Automated Stateful Protocol Verification

Andreas V. Hess* Sebastian Mödersheim*
Achim D. Brucker† Anders Schlichtkrull‡

March 17, 2025

*DTU Compute, Technical University of Denmark, Lyngby, Denmark
{avhe, samo}@dtu.dk

† Department of Computer Science, University of Exeter, Exeter, UK
a.brucker@exeter.ac.uk

‡ Department of Computer Science, Aalborg University, Copenhagen, Denmark
andsch@cs.aau.dk

Abstract

In protocol verification we observe a wide spectrum from fully automated methods to interactive theorem proving with proof assistants like Isabelle/HOL. In this AFP entry, we present a fully-automated approach for verifying stateful security protocols, i.e., protocols with mutable state that may span several sessions. The approach supports reachability goals like secrecy and authentication. We also include a simple user-friendly transaction-based protocol specification language that is embedded into Isabelle.

Keywords: Fully automated verification, stateful security protocols

Contents

1 Introduction

In protocol verification we observe a wide spectrum from fully automated methods to interactive theorem proving with proof assistants like Isabelle/HOL. The latter provide overwhelmingly high assurance of the correctness, which automated methods often cannot: due to their complexity, bugs in such automated verification tools are likely and thus the risk of erroneously verifying a flawed protocol is non-negligible. There are a few works that try to combine advantages from both ends of the spectrum: a high degree of automation and assurance.

Inspired by [1], we present here a first step towards achieving this for a more challenging class of protocols, namely those that work with a mutable long-term state. To our knowledge this is the first approach that achieves fully automated verification of stateful protocols in an LCF-style theorem prover. The approach also includes a simple user-friendly transaction-based protocol specification language embedded into Isabelle, and can also leverage a number of existing results such as soundness of a typed model (see, e.g., [3–5]) and compositionality (see, e.g., [3, 6]). The Isabelle formalization extends the AFP entry on stateful protocol composition and typing [7].

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. chapter 2 provides a manual of our automated protocol verification tool, called PSPSP, that is provided as part of this AFP entry. Thereafter, the structure of this document follows the theory dependencies (see Figure 1.1): After introducing the formal framework for verifying stateful security protocols (chapter 3), we continue with the setup for supporting the high-level protocol specifications language for security protocols (the Trac format) and the implementation of the fully automated proof tactics (chapter 4). Finally, we present examples (chapter 5).

Acknowledgments This work was supported by the Sapere-Aude project “Composec: Secure Composition of Distributed Systems”, grant 4184-00334B of the Danish Council for Independent Research, by the EU H2020 project no. 700321 “LIGHTest: Lightweight Infrastructure for Global Heterogeneous Trust management in support of an open Ecosystem of Trust schemes” (lightest.eu) and by the “CyberSec4Europe” European Union’s Horizon 2020 research and innovation programme under grant agreement No 830929.

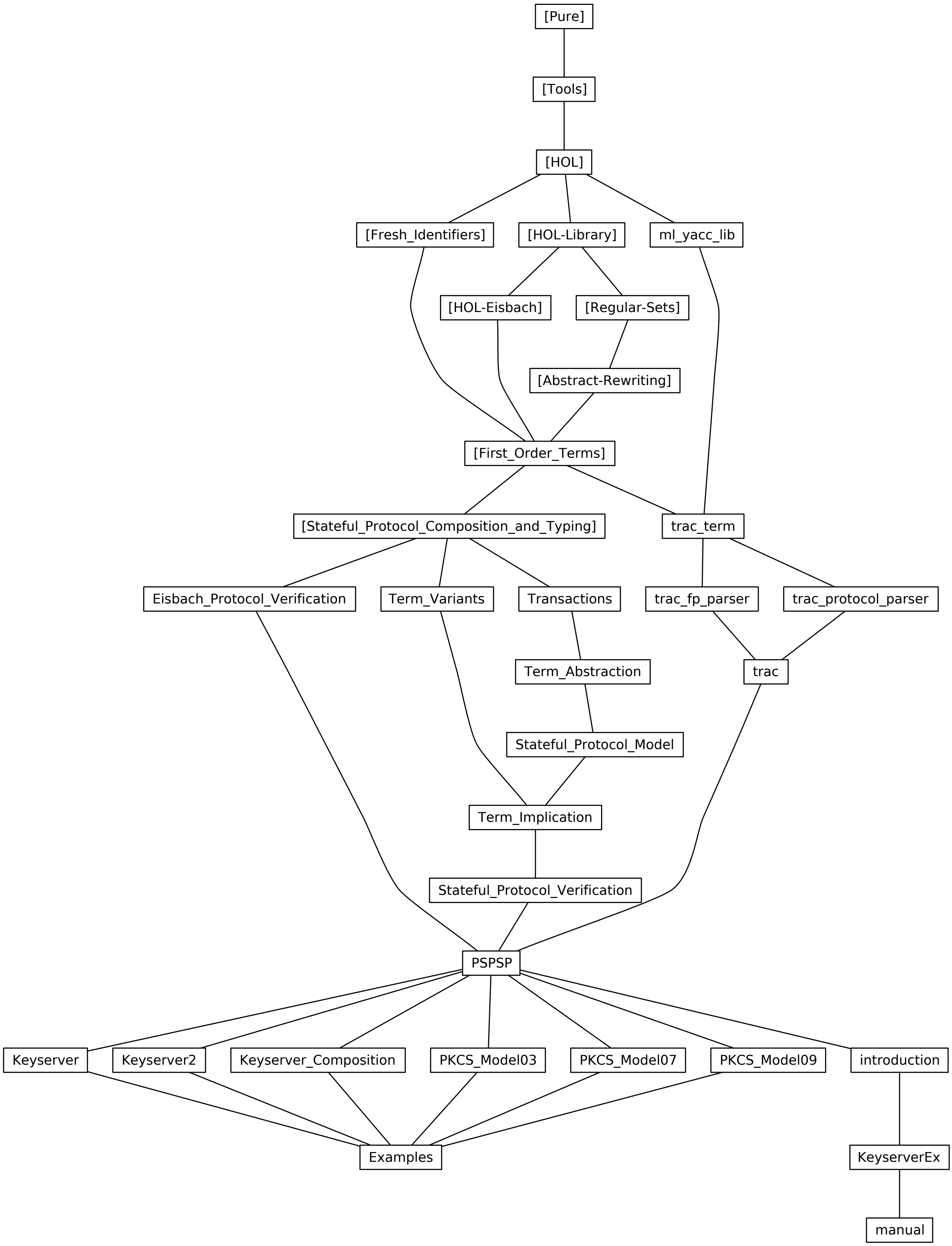


Figure 1.1: The Dependency Graph of the Isabelle Theories.

2 The PSPSP Manual

2.1 Introduction

In this section, we describe the installation and use of Isabelle/PSPSP, the system implementing the approach described in our CSF submission.

Isabelle/PSPSP is built on top of the latest version of Isabelle/HOL [8]. While Isabelle is widely perceived as an interactive theorem prover for HOL (Higher-order Logic), we would mention that Isabelle can be understood as a framework that provides various extension points. In our work, we make use of this fact by extending Isabelle/HOL with:

- a formalization of the protocol-independent aspects of our approach that is based on a large formalization (the session is called `Automated_Stateful_Protocol_Verification`) of security protocols in Isabelle/HOL that, among others, includes proofs for typing results and protocol compositionality. The main entry for the security analysis of concrete protocols using Isabelle/PSPSP is the theory *`Automated_Stateful_Protocol_Verification.PSPSP`*.
- an encoder (datatype package) that translates a high-level protocol specification (called "trac") into HOL. This datatype package provides the high-level command **trac**.
- a command (called **compute_fixpoint**) that computes an over-approximation of all messages that a security protocol can generate.
- a command that, for a specific class of protocols, can fully-automatically prove their security (**protocol_security_proof**).
- a command that generates a list of proof obligations (sub-goals) for proving the security of the specified protocol interactively (**manual_protocol_security_proof**).
- several proof methods that either can be used interactively or that are used internally by the fully automated poof setup (**protocol_security_proof**).

2.2 Installation

Isabelle/PSPSP extends Isabelle/HOL. Thus, the first step is to install Isabelle. Moreover, we make use of the Archive of Formal Proofs (AFP), which needs to be installed in a second step. Finally, we need to register the new Isabelle components and compile the session heaps for faster start up.

2.2.1 Installing Isabelle

Isabelle can be downloaded from the Isabelle website (`http://isabelle.in.tum.de/`). Detailed installation instructions for all supported operating systems are available at `https://isabelle.in.tum.de/installation.html`.

2.2.2 Installing the Archive of Formal Proofs

After installing Isabelle, we now need to install the AFP (Archive of Formal Proofs). The AFP (`https://www.isa-afp.org`) is a large library of Isabelle formalizations. Please install the latest version, following the instructions from `https://www.isa-afp.org/using.html`.

2.2.3 Compiling Session Heaps and Final Setup

We recommend[1] to "compile" Isabelle/PSPSP (in Isabelle lingo: building the session heaps) on the command line. This can be done by executing (please take care of the full qualified path of the `isabelle` binary for your operating system):

```bash
achim@logicalhacking:~$ isabelle build -b Automated_Stateful_Protocol_Verification
Building Pure ...
Finished Pure (0:00:50 elapsed time, 0:00:50 cpu time, factor 1.00)
Building HOL ...
Finished HOL (0:09:50 elapsed time, 0:31:02 cpu time, factor 3.16)
Building HOL-Library ...
Finished HOL-Library (0:04:49 elapsed time, 0:24:43 cpu time, factor 5.13)
Building Abstract-Rewriting ...
Finished Abstract-Rewriting (0:01:28 elapsed time, 0:04:00 cpu time, factor 2.71)
Building First_Order_Terms ...
Finished First_Order_Terms (0:00:47 elapsed time, 0:01:54 cpu time, factor 2.39)
Building Stateful_Protocol_Composition_and_Typing ...
Finished Stateful_Protocol_Composition_and_Typing (0:08:18 elapsed time, 0:36:38 cpu time,  \
    factor 4.41)
Building Automated_Stateful_Protocol_Verification ...
Finished Automated_Stateful_Protocol_Verification (0:15:11 elapsed time, 0:50:57 cpu time,  \
    factor 3.36)
0:41:46 elapsed time, 2:30:06 cpu time, factor 3.59
achim@logicalhacking:~$
```

Isabelle will build all sessions that are required. Note that you might have already some of the heaps available and, hence, only a subset of the list shown above might be build on your system.

Finally, please start the (graphical) Isabelle application by clicking on the Isabelle icon (macOS) or by starting `Isabelle2021-1` (this example is for Isabelle version 2021-1) on the command line (Linux and macOS):

```bash
achim@logicalhacking:~$ ./Isabelle2021-1/Isabelle2021-1
```

and select the session `Automated_Stateful_Protocol_Verification`. For doing so, you need to select the "Theories"-pane on the right hand side and select the session from drop-down menu (see Figure 2.1). To persist this configuration, you need to restart Isabelle, i.e., please close Isabelle/jEdit now. On the next start, `Automated_Stateful_Protocol_Verification` will be the default session.

2.3 A Brief Overview of Isabelle/PSPSP

In this section, we briefly explain how to use Isabelle/PSPSP for proving the security of protocols. As Isabelle/PSPSP is build on top of Isabelle/HOL, the overall user interface and the high-level language (called Isar) are inherited from Isabelle. We refer the reader to [8] and the system manuals that are part of the Isabelle distribution. The latter are accessible within Isabelle/jEdit in the documentation pane on the left-hand side of the main window .

In the following, we will illustrate the use of our system by analyzing a simple keyserver protocol (this theory is stored in the file `PSPSP-Manual/KeyserverEx.thy`. When loading this theory in Isabelle/jEdit, please ensure that the session `Automated_Stateful_Protocol_Verification` is active (this session provides Isabelle/PSPSP).

When done, please move the text cursor to the section "Proof of Security". There are some orange question marks at the side of some lines. These are the comments from Isabelle that indicate the timing results we ask for: when moving the cursor to the corresponding line, and selecting the `Output`-Tab on the bottom of the Isabelle window (ensure that there is a tick-mark on "Auto update"), you see the timing information provided by Isabelle for each step. Your Isabelle should look similar to Figure 2.2.

[1] The sessions should also be build automatically on the start of Isabelle's graphical user interface Isabelle/jEdit. For this, it is important that you select the session `Automated_Stateful_Protocol_Verification` as described in the following paragraph and *restart* Isabelle. For us, building on the command line has easier to reproduce on different machines.

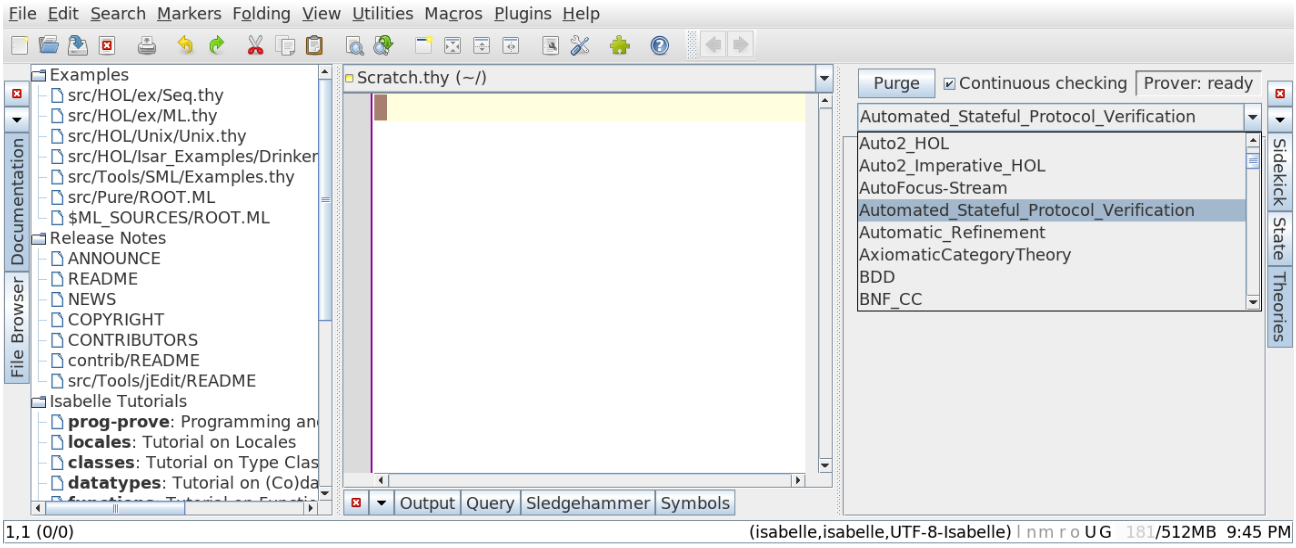


Figure 2.1: Isabelle/jEdit on its first startup. Please click on the "Theories" tab on the right hand side and select the session "Automated_Stateful_Protocol_Verification."

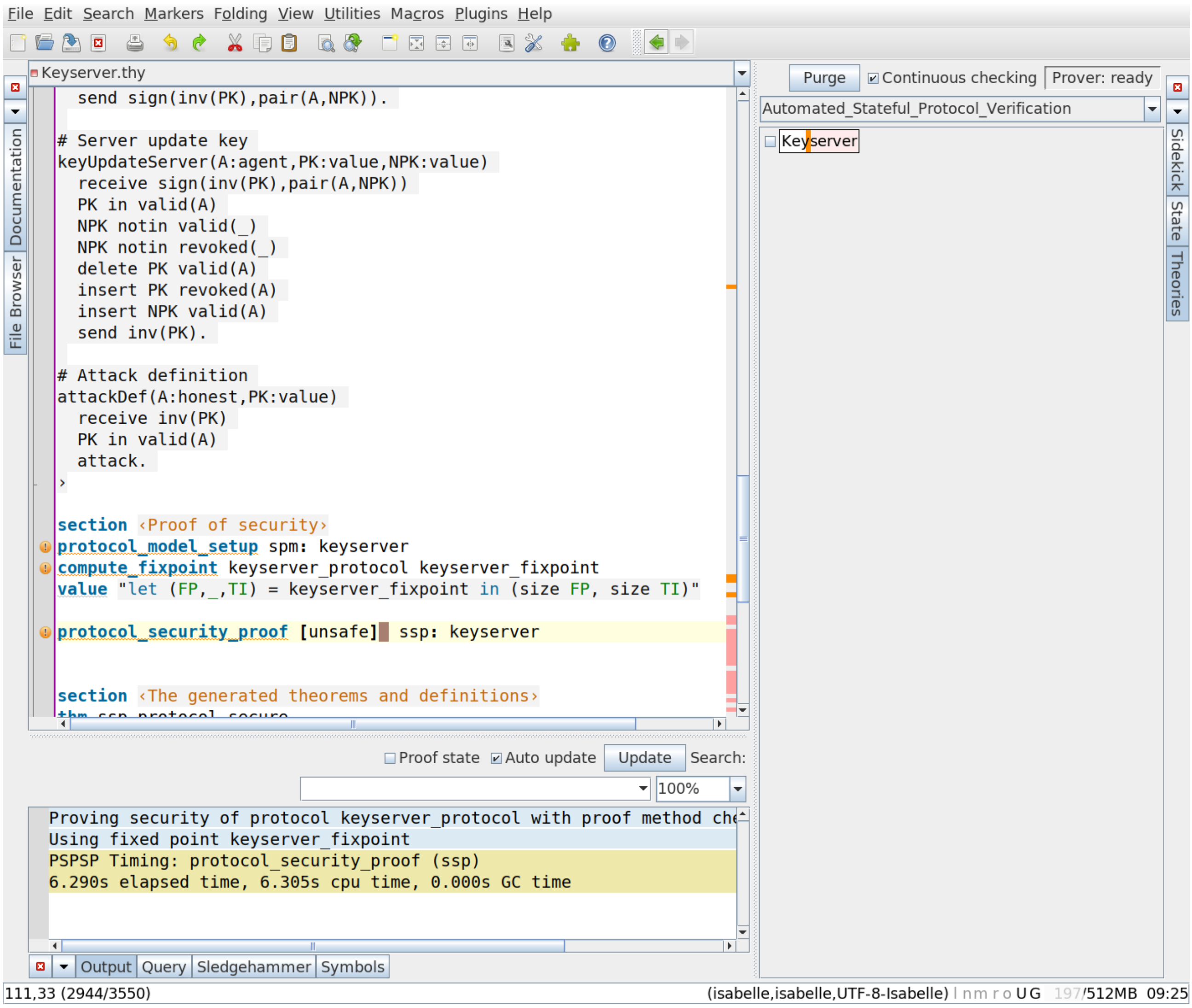


Figure 2.2: Opening KeyserverEx.thy in Isabelle/jEdit.

The Isabelle IDE (called Isabelle/jEdit) is a front-end for Isabelle that supports most features known from IDEs for programming languages. The input area (in the middle of the upper part of the window) supports, e.g., auto completion, syntax highlighting, and automated proof generation as well as interactive proof development. The lower part shows the current output (response) with respect to the cursor position.

We will now briefly explain this example in more detail. First, we start with the theory header: As in Isabelle/HOL, formalization happens within theories. A theory is a unit with a name that can import other theories. Consider the following theory header:

```
theory
  KeyserverEx
imports
  Automated_Stateful_Protocol_Verification.PSPSP
  begin
```

which opens a new theory `KeyserverEx` that is based on the top-level theory of Isabelle/PSPSP, called *Automated_Stateful_Protocol_Verification.PSPSP*. Within this theory, we can use all definitions and tools provided by Isabelle/PSPSP. For example, Isabelle/PSPSP provides a mechanism for measuring the run-time of certain commands. This mechanism can be turned on as follows:

```
declare [[pspsp_timing]]
```

2.3.1 Protocol Specification

The protocol is specified using a domain-specific language that, e.g., could also be used by a security protocol model checker. We call this language “trac” and provide a dedicated environment (command) **trac** for it:

```
trac‹
Protocol: Keyserver

Enumerations:
honest = {a,b,c}
dishonest = {i}
agent = honest ++ dishonest

Sets:
ring/1 valid/1 revoked/1 deleted/1

Functions:
Public sign/2 crypt/2 pair/2
Private inv/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
pair(X,Y) -> X,Y

Transactions:
# Out-of-band registration
outOfBand(A:honest)
  new PK
  insert PK ring(A)
  insert PK valid(A)
  send PK.

# Out-of-band registration (for dishonest users; they reveal their private keys to the intruder)
outOfBandD(A:dishonest)
  new PK
  insert PK valid(A)
  send PK
  send inv(PK).

# User update key
keyUpdateUser(A:honest,PK:value)
```

```
  PK in ring(A)
  new NPK
  delete PK ring(A)
  insert PK deleted(A)
  insert NPK ring(A)
  send sign(inv(PK),pair(A,NPK)).

# Server update key
keyUpdateServer(A:agent,PK:value,NPK:value)
  receive sign(inv(PK),pair(A,NPK))
  PK in valid(A)
  NPK notin valid(_)
  NPK notin revoked(_)
  delete PK valid(A)
  insert PK revoked(A)
  insert NPK valid(A)
  send inv(PK).

# Attack definition
attackDef(A:honest,PK:value)
  receive inv(PK)
  PK in valid(A)
  attack.
›
```

The command **trac** automatically translates this specification into a family of formal HOL definitions. Moreover, basic properties of these definitions are also already proven automatically (i.e., without any user interaction): for this simple example, already over 350 definitions and theorems are automatically generated, respectively, formally proven. For example, the following induction rule is derived:

```
⟦?P sign; ?P crypt; ?P pair; ?P Keyserver_fun.inv; ?P PrivFunSec;
 ⋀uu_. ?P (enum uu_)⟧
⟹ ?P ?a0.0
```

2.3.2 Protocol Model Setup

Next, we show that the defined protocol satisfies the requirement of our protocol model (technically, this is done by instantiating several Isabelle locales, resulting in over 1750 theorems "for free."). The underlying instantiation proofs are fully automated by our tool:

protocol_model_setup *spm: Keyserver*

2.3.3 Fixed Point Computation

Now we compute the fixed-point:

compute_fixpoint *Keyserver_protocol Keyserver_fixpoint*

We can inspect the fixed-point with the following command:

thm *Keyserver_fixpoint_def*

Moreover, we can use Isabelle's **value**-command to compute its size:

value *"let (FP,_,TI) = Keyserver_fixpoint in (size FP, size TI)"*

2.3.4 Proof of Security

After these steps, all definitions and auxiliary lemmas for the security proof are available. Note that the security proof will fail, if any of the previous commands did fail. A failing command is sometimes hard to spot for non Isabelle experts: the status bar next to the scroll bar on the right-hand side of the window should not have any "'dark red" markers.

We can do a fully automated security proof using a new command **protocol_security_proof**:

protocol_security_proof *ssp: Keyserver*

This command proves the security of the protocol using only Isabelle's simplifier (and, hence, everything is checked by Isabelle's LCF-style kernel).

Moreover, we provide two alternative configuration, one using an approach called "normalization by evaluation" (nbe) and one using Isabelle's code generator for direct code evaluation (eval). Please see section 2.5 and Isabelle's code generator manual [2] for details.

protocol_security_proof *[nbe] ssp: Keyserver*

While the stack of code that needs to be trusted for the normalization by evaluation is much smaller than for the direct code evaluation, direct code evaluation is usually much faster:

protocol_security_proof *[eval] ssp: Keyserver*

Moreover, there is the option to only generate the proof obligations (as sub-goals) for an interactive security proof:

```
manual_protocol_security_proof ssp: Keyserver
  for Keyserver_protocol Keyserver_fixpoint
  apply check_protocol_intro
  subgoal by (timeit code_simp)
  subgoal by (timeit eval)
  subgoal by (timeit code_simp)
  subgoal by (timeit normalization)
  subgoal by (timeit code_simp)
  done
```

Such an interactive proof allows us to interactively inspect intermediate proof states or to use protocol-specific proof strategies (e.g., only partially unfolding the fixed-point).

2.3.5 Inspecting the Generated Theorems and Definitions

We can inspect the generated proofs using the **thm** command:

```
thm ssp.protocol_secure
thm spm.constraint_model_def
thm spm.reachable_constraints.simps

thm Keyserver_enum_consts.nchotomy
thm Keyserver_sets.nchotomy
thm Keyserver_fun.nchotomy
thm Keyserver_atom.nchotomy
thm Keyserver_arity.simps
thm Keyserver_sets_arity.simps
thm Keyserver_public.simps
thm Keyserver_Γ.simps
thm Keyserver_Ana.simps

thm Keyserver_protocol_def
thm Keyserver_transaction_intruderValueGen_def
thm Keyserver_transaction_outOfBand_def
thm Keyserver_transaction_outOfBandD_def
thm Keyserver_transaction_keyUpdateUser_def
thm Keyserver_transaction_keyUpdateServer_def
thm Keyserver_transaction_attackDef_def

thm Keyserver_fixpoint_def
```

Finally, the theory needs to be closed:

end

2.4 Common Pitfalls

This section explains some common pitfalls, along with solutions, that one may encounter when writing trac specifications.

2.4.1 Using Value-Typed Database-Parameters in Database-Expressions

Due to the nature of the abstraction that is at the core of our verification approach it is simply not possible to use value-typed variables in parameters to databases. Hence, a trac specification with the following transaction would be rejected:

```
f(PK:value,A:value)
   PK in db(A).
```

As an alternative one could declare A with a type—say, `agent`—that is itself declared in the `Enumerations` section of the trac specification:

```
Enumerations:
agent = {a,b,c}

Transactions:
f(PK:value,A:agent)
   PK in db(A).
```

2.4.2 Not Ordering the Action Sequences in Transactions Correctly

The actions of a transaction should occur in the correct order; first receive actions, then database checks, then new actions and database updates, and finally send actions.

Hence, the following is an invalid transaction:

```
invalid(PK:value)
   send f(PK)
   receive g(PK).
```

whereas the following is valid:

```
valid(PK:value)
   receive f(PK)
   send g(PK).
```

2.4.3 Declaring Ill-Formed Analysis Rules

Each analysis rule must either be of the form

```
Ana(f(X1,...,Xn)) ? t1,...,tk -> Y1,...,Ym
```

or of the form

```
Ana(f(X1,...,Xn)) -> Y1,...,Ym
```

where `f` is a function symbol of arity `n`, the variables `Xi` are all distinct, the variables occurring in the `ti` terms are among the `Xi` variables, and the variables `Yi` are among the `Xi` variables.

2.4.4 Declaring Public Constants of Type Value

It is not possible to directly refer to constants of type value. A possible workaround is to instead add a transaction that generates fresh values and releases them to the intruder (thereby making them "public"):

```
freshPublicValues():
   new K
   send K.
```

It is usually beneficial to ensure that all fresh values are inserted into a database before being transmitted over the network. In this example one could use a database that is not used anywhere else:

```
freshPublicValues():
   new K
   insert K publicvalues
   send K.
```

Under the set-based abstraction this prevents accidentally identifying values produced from this transaction with values produced elsewhere in the protocol, since they are now identified with their own unique abstract value `{publicvalues}` instead of the more common "empty" abstract value `{}`.

2.4.5 Forgetting to Terminate Transactions With a period

Transactions must end with a period. Forgetting this period may result in a confusing error message from the parser. For instance, suppose that we have the following `Transaction` section where we forgot to terminate the `valueProducer` transaction:

```
valueProducer()
   new PK
   send PK

attackDef(PK:value)
   attack.
```

This could result in an error message like the following:

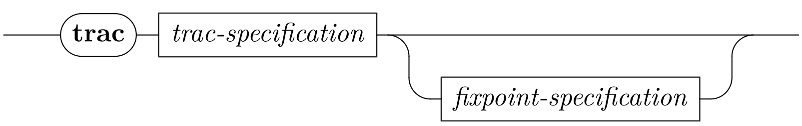
```
Error, line .... 14.13, syntax error: deleting COLON LOWER_STRING_LITERAL
```

2.5 Reference Manual

In this section, we briefly introduce the syntax of the most important commands and methods of Isabelle/PSPSP. We follow, in our presentation, the style of the Isabelle/Isar manual [9]. For details about the standard Isabelle commands and methods, we refer to the reader to this manual [9].

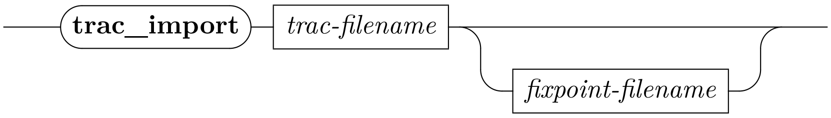
2.5.1 Top-Level Isabelle Commands

trac



This command takes a protocol in the trac language as argument. The command translates this high-level protocol specification into a family of HOL definitions and also proves already a number of basic properties over these definitions. The generated definitions are all prefixed with the name of the protocol, as given as part of the trac specification (e.g., if `keyserver` is the protocol name given in the trac specification, then `keyserver_protocol` will refer to the generated HOL constant that represents the transactions of the protocol). As an optional argument the command can take a fixed point in the trac language and generate HOL definitions for it as well.

trac_import

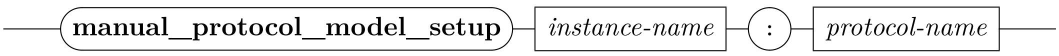


This command takes the name of a trac protocol specification file as argument, and, optionally, the name of a file with a fixed point in the trac language. The command loads the protocol and (optionally) fixed-point specifications from the files and then executes the **trac** command on these specifications.

protocol_model_setup

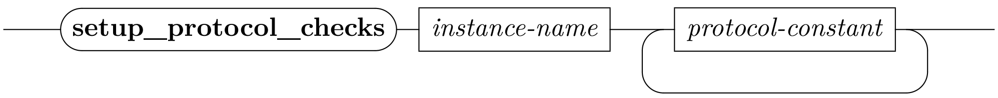
This command takes two arguments: the name that should be used to refer to the protocol model, and the name of the protocol (as given in the trac specification). In general, this command proves a large number of properties over the protocol specification that are later used by our security proof. In particular, the command does internally instantiation proofs showing, e.g., that the protocol specification satisfies the requirements of the typed model of [5].

manual_protocol_model_setup



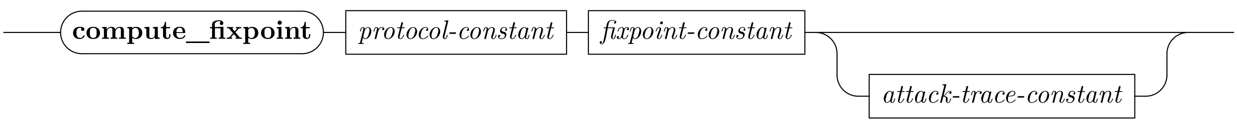
This command allows to interactively set up the protocol model. As the fully automated version, it takes the name for the protocol model and the protocol name as arguments but it does not execute a proof. Instead, it generates a proof state with the necessary proof obligations. It is the responsibility of the user to discharge these proof obligations. Application of this command results in a regular Isabelle proof state and, hence, all proof methods of Isabelle can be used.

setup_protocol_checks



This command declares attributes for the definitions of the `protocol_constant` HOL constants given as arguments (among other constants used in the automated proofs of protocol security). This can later be used to expand the proof obligations when proving fixed-point coverage with **manual_protocol_security_proof** and using the proof methods `coverage_check_intro` and `check_protocol_intro'`. The command takes as arguments the name of the protocol model instance given at an earlier point to the **manual_protocol_model_setup** command, and a non-empty sequence of protocol HOL constant names for which the command will perform its setup.

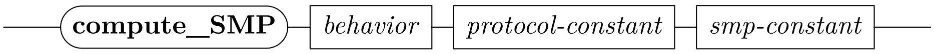
compute_fixpoint



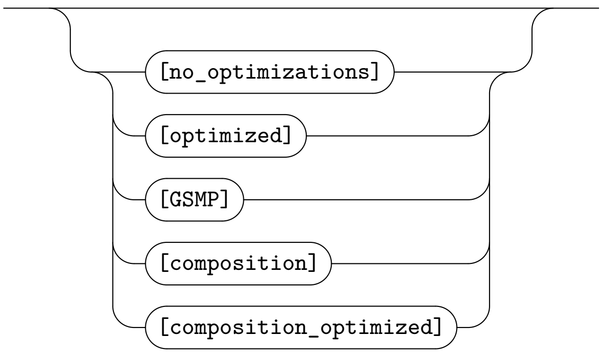
This command computes the fixed-point of a protocol. It takes two arguments, first the name of the HOL constant representing the protocol (usually the name given in the trac specification suffixed with `_protocol`), second, the name that should be used for the constant to which the generated fixed point is bound. The algorithm for computing the fixed-point has been specified in HOL. Internally, Isabelle's code generator is used for deriving an SML implementation that is actually used. Note that our approach *does not* rely on the correctness of this algorithm neither on the correctness of the code generator.

The command can optionally generate a HOL constant that represents an attack trace, bound to the HOL constant `attack_trace_constant`. The attack trace can later be given to the **print_attack_trace** to print it. This optional argument, `attack_trace_constant`, should only be given if the computed fixed point will contain an attack signal.

compute_SMP



behavior

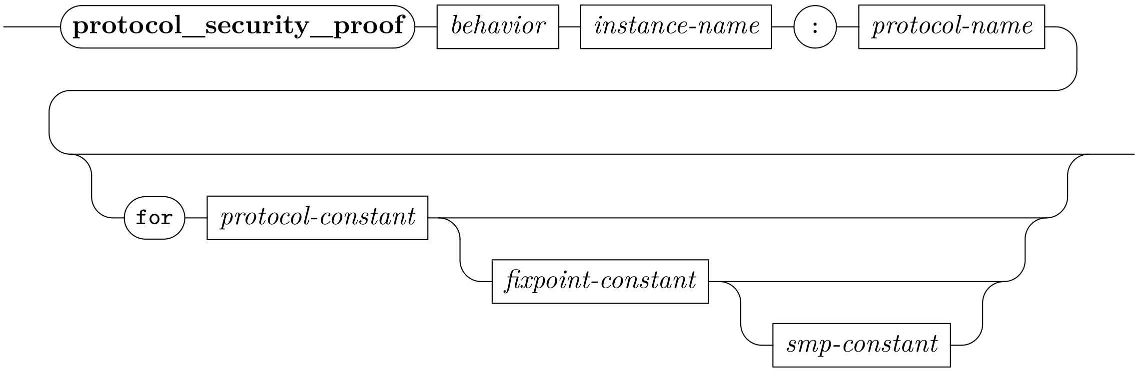


This command computes a finite representation of the Sub-Message Patterns (SMP) set of the protocol. This set is used to automatically prove the conditions of the typing result of [5] (named type-flaw resistance) during a security proof. It takes two mandatory arguments; first the protocol name (as given in the trac specification) and, second, the name that should be used for the constant to which the generated SMP set is bound.

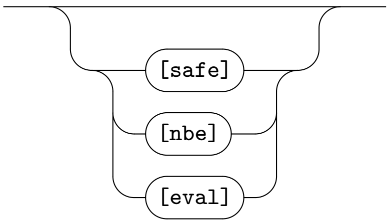
The optional argument can take the following values:

- `[no_optimizations]`: Computes the finite SMP representation set without any optimizations (this is the default setting).
- `[optimized]`: Applies optimizations to reduce the size of the computed set, but this might not be sound.
- `[GSMP]`: Computes a set suitable for use in checking GSMP disjointness (see the **protocol_composition_proof** command for further information).
- `[composition]`: Computes a set suitable for checking type-flaw resistance of composed protocols (see the **protocol_composition_proof** command for further information).
- `[composition_optimized]`: This is an optimized variant of the previous setting.

protocol_security_proof



behavior



This command executes the formal security proof for the given security protocol. Its internal behavior can be configured using one of the following three options:

- *[safe]* (default): use Isabelle's simplifier to prove the goal by symbolic evaluation. In this mode, all proof steps are checked by Isabelle's LCF-style kernel.
- *[nbe]*: use normalization by evaluation, a partial symbolic evaluation which permits also normalization of functions and uninterpreted symbols. This setup uses the well-tested default configuration of Isabelle's code generator for HOL. While the stack of code to be trusted is considerable, we consider this still a highly trustworthy setup, as it cannot be influenced by end-user configurations of the code generator.
- *[eval]*: use Isabelle's code-generator for evaluating the proof goal on the SML-level. While this is, by far, the fastest setup, it depends on the full-blown code-generator setup. As we do not modify the code-generator setup in our formalization, we consider the setup to be nearly as trustworthy as the normalization by evaluation setup. Still, end-user configurations of the code generator could, inadvertently, introduce inconsistencies.

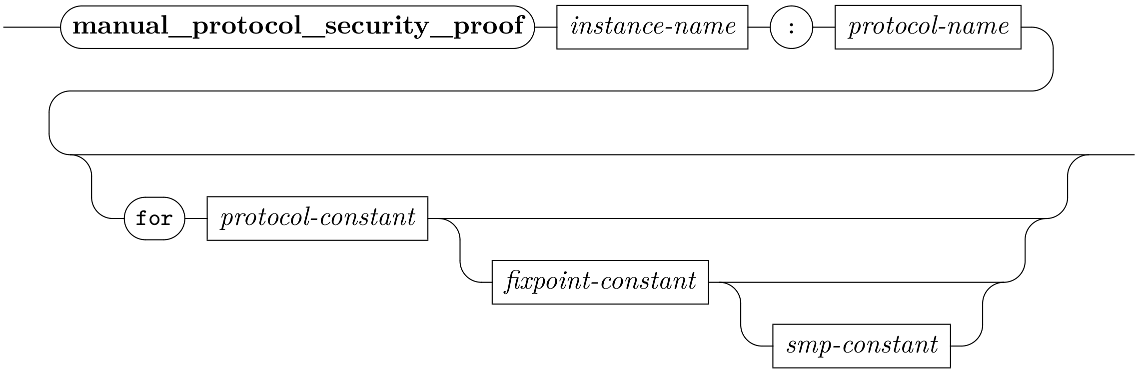
For a detailed discussion of these three modes and the different software stacks that need to be trusted, we refer the reader to the tutorial describing the code generator [2, Section 5.1].

The remaining arguments are the following:

- *instance_name*: The name that should be used to refer to the instance of the *secure_stateful_protocol* locale that is interpreted by this command.
- *protocol_name*: The name of the trac protocol to be proven secure, as given in the protocol specification. By default, if no other arguments are given, the command will use the HOL constants *protocol_name_protocol* and *protocol_name_fixpoint* for the protocol respectively fixed point used in the security proof.
- *protocol_constant* (optional): The name of the HOL constant that represents the protocol to be proven secure.
- *fixpoint_constant* (optional): The name of the HOL constant that represents the fixed point that is used in the security proof.
- *smp_constant* (optional): The name of the HOL constant that represents the SMP set of the protocol.

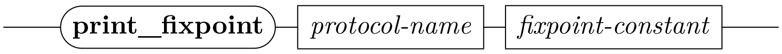
After successful execution of this command, the security theorem of the protocol is available as *instance_name.protocol_secure*. The corollary *instance_name.protocol_welltyped_secure* is the version of the security theorem restricted to the typed model.

manual_protocol_security_proof



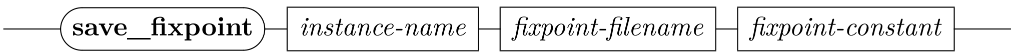
This command allows to interactively prove the security of a protocol. As the fully automated version, it takes the protocol name as argument but it does not execute a proof. Instead, it generates a proof state with the necessary proof obligations. It is the responsibility of the user to discharge these proof obligations. Application of this command results in a regular Isabelle proof state and, hence, all proof methods of Isabelle can be used.

print_fixpoint



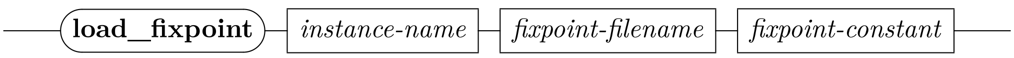
This command translates the given HOL constant fixed point into the trac-language and then prints it. It takes as arguments, first, the name of the protocol (as given in the trac specification), and, second, the name of the HOL constant that represents a fixed point.

save_fixpoint



This command translates the given HOL constant fixed point into the trac-language and then saves it to the file whose name is given as argument. It takes as arguments the name of the interpreted protocol model, the name of the HOL constant for the fixed point, and the output filename.

load_fixpoint

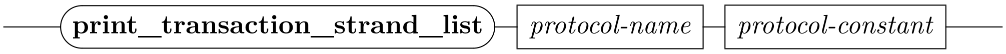


This command loads a trac fixed-point from a file. It takes as arguments the name of the interpreted protocol model, the input filename, and the name of the HOL constant to be defined.

print_transaction_strand

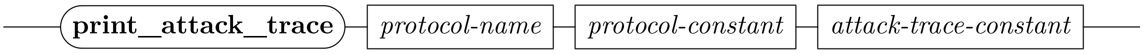
This command takes a HOL constant that represents a transaction, translates it into a syntax that is similar to trac, and then prints it. It takes as arguments the name of the name of the trac specification and the HOL constant to be printed.

print_transaction_strand_list



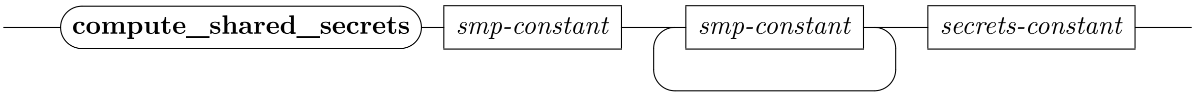
This command takes a HOL constant that represents a list of transactions (such as a protocol HOL constant), translates it into a syntax that is similar to trac, and then prints it. It takes as arguments the name of the name of the trac specification and the HOL constant to be printed.

print_attack_trace



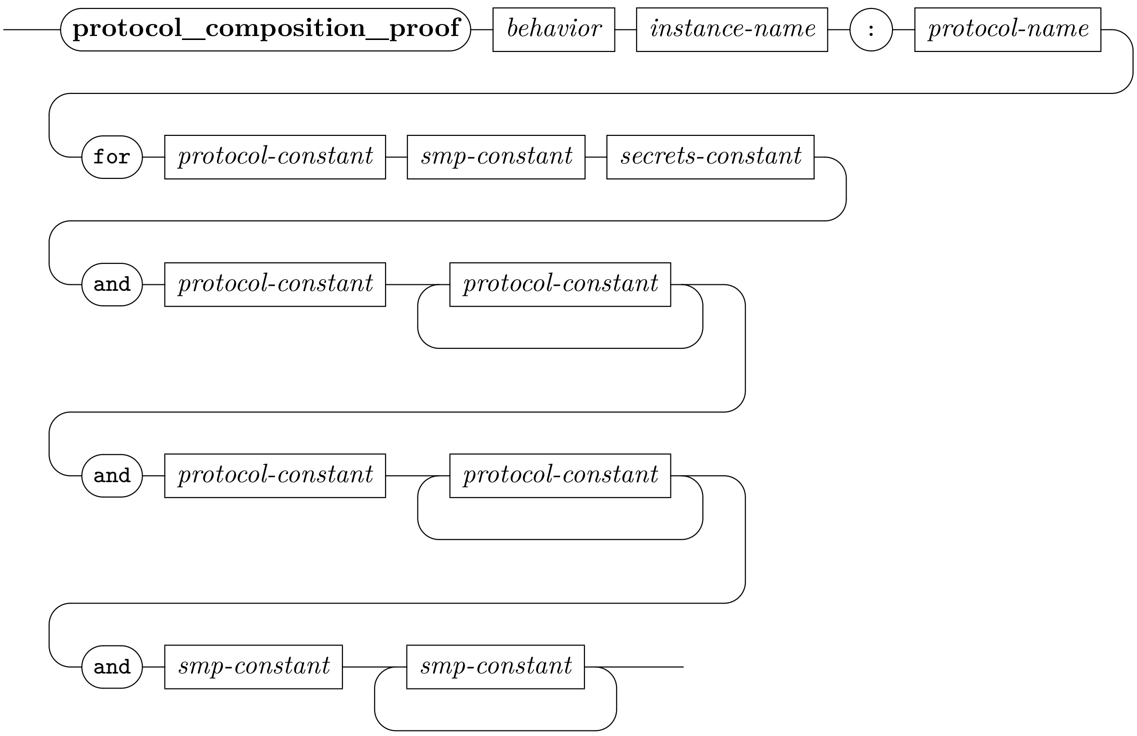
This command takes a HOL constant that represents an attack trace, translates it into a syntax that is similar to trac, and then prints it. It takes as arguments the name of the trac specification, the protocol HOL constant that has an attack, and the HOL constant for the attack trace.

compute_shared_secrets

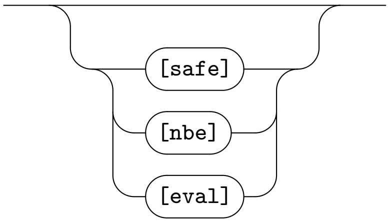


This command computes a finite representation of the terms that are in the intersection of two or more GSMP sets. This is useful to compute the set of secrets that are shared between two or more protocols. It takes as arguments a sequence of SMP HOL constants and the name of the HOL constant to which the output should be bound.

protocol_composition_proof



behavior



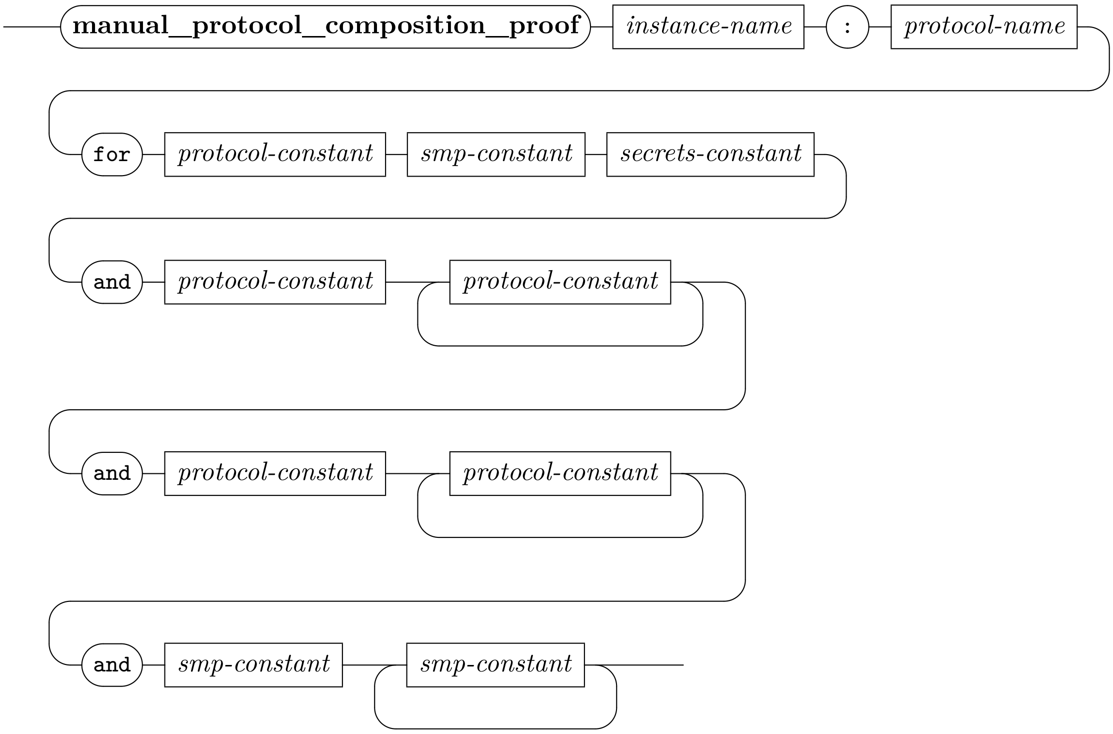
This command applies the compositionality theorem of [6] to the protocols given as arguments and automatically proves the syntactic conditions required for composition.

After successful execution of this command the theorem
instance_name.composed_protocol_preserves_component_goals
is available which states the following: if the nth component protocol is secure (in a typed model), and all component protocols are leakage-free, then the goals of the nth component protocol hold in the composed protocol as well (also in an untyped model).

The command takes the following arguments:

- An optional argument to set the behavior of the internal automated proof (see **protocol_security_proof** for an explanation).
- The name to which the interpreted locale for the composition should be bound.
- The name of the protocol as given in the trac-specification.
- The HOL constant representing the composed protocol. Usually *protocol_name_protocol* where *protocol_name* is the name of the protocol as given in the trac-specification.
- The HOL constant representing the SMP set of the composed protocol, usually computed using the **compute_SMP** command with the *[composition]* or *[composition_optimized]* optional argument.
- The HOL constant representing the shared secrets between the component protocols, usually computed with the **compute_shared_secrets** command.
- A sequence of two or more HOL constants representing the component protocols. Their union must evaluate to the same term as the composed protocol HOL constant given as an earlier argument. Usually this sequence is of the form *protocol_name_protocol_N*, for each *N*, where *N* is the name of the nth component protocol as given in the trac-specification.
- A sequence of two or more HOL constants representing the composition of each component protocol composed with the abstractions of the other component protocols. It is important that the ordering of these arguments matches the ordering of the component protocols as given in the previous sequence of arguments, and that the length matches the length of the previous sequence: the nth element of this sequence must represent the composition of the nth element from the sequence of component protocols, composed with the abstraction of all the other component protocols. Usually this sequence is of the form *protocol_name_protocol_N_with_star_projs*, for each *N*, where *N* is the name of the nth component protocol as given in the trac-specification.
- A sequence of two or more HOL constants that represent the Ground Sub-Message Patterns (GSMP) of the component protocols, usually computed using the **compute_SMP** with the *[GSMP]* optional argument. The number of elements in this sequence, and their ordering, must again match the previous sequence of arguments.

manual_protocol_composition_proof



This command allows to interactively prove the composition of protocols. As the fully automated version, it takes the protocol name as argument but it does not execute a proof. Instead, it generates a proof state with the necessary proof obligations. It is the responsibility of the user to discharge these proof obligations. Application of this command results in a regular Isabelle proof state and, hence, all proof methods of Isabelle can be used.

2.5.2 Proof Methods

In addition to the Isar commands discussed in the previous section, Isabelle/PSPSP also provides a number of proof methods such as *`check_protocol_intro`* or *`coverage_check_unfold`* or *`composable_protocols_intro`*. These domain specific proof methods are used internally by, e.g., the command **protocol_security_proof** and can also be used in interactive mode.

3 Stateful Protocol Verification

3.1 Protocol Transactions

```
theory Transactions
  imports
    Stateful_Protocol_Composition_and_Typing.Typed_Model
    Stateful_Protocol_Composition_and_Typing.Labeled_Stateful_Strands
begin
```

3.1.1 Definitions

```
datatype 'b prot_atom =
  is_Atom: Atom 'b
| Value
| SetType
| AttackType
| Bottom
| OccursSecType
| AbsValue

datatype ('a,'b,'c,'d) prot_fun =
  Fu (the_Fu: 'a)
| Set (the_Set: 'c)
| Val (the_Val: "nat")
| Abs (the_Abs: "'c set")
| Attack (the_Attack_label: "'d strand_label")
| Pair
| PubConst (the_PubConst_type: "'b prot_atom") nat
| OccursFact
| OccursSec

definition "is_Fun_Set t ≡ is_Fun t ∧ args t = [] ∧ is_Set (the_Fun t)"

definition "is_Fun_Attack t ≡ is_Fun t ∧ args t = [] ∧ is_Attack (the_Fun t)"

definition "is_PubConstValue f ≡ is_PubConst f ∧ the_PubConst_type f = Value"

abbreviation occurs where
  "occurs t ≡ Fun OccursFact [Fun OccursSec [], t]"

type_synonym ('a,'b,'c,'d) prot_term_type = "(('a,'b,'c,'d) prot_fun,'b prot_atom) term_type"

type_synonym ('a,'b,'c,'d) prot_var = "('a,'b,'c,'d) prot_term_type × nat"

type_synonym ('a,'b,'c,'d) prot_term = "(('a,'b,'c,'d) prot_fun,('a,'b,'c,'d) prot_var) term"
type_synonym ('a,'b,'c,'d) prot_terms = "('a,'b,'c,'d) prot_term set"

type_synonym ('a,'b,'c,'d) prot_subst = "(('a,'b,'c,'d) prot_fun, ('a,'b,'c,'d) prot_var) subst"

type_synonym ('a,'b,'c,'d) prot_strand_step =
  "(('a,'b,'c,'d) prot_fun, ('a,'b,'c,'d) prot_var, 'd) labeled_stateful_strand_step"
type_synonym ('a,'b,'c,'d) prot_strand = "('a,'b,'c,'d) prot_strand_step list"
type_synonym ('a,'b,'c,'d) prot_constr = "('a,'b,'c,'d) prot_strand_step list"


datatype ('a,'b,'c,'d) prot_transaction =
```

```
  Transaction
    (transaction_decl:    "unit ⇒ (('a,'b,'c,'d) prot_var × 'a set) list")
    (transaction_fresh:   "('a,'b,'c,'d) prot_var list")
    (transaction_receive: "('a,'b,'c,'d) prot_strand")
    (transaction_checks:  "('a,'b,'c,'d) prot_strand")
    (transaction_updates: "('a,'b,'c,'d) prot_strand")
    (transaction_send:    "('a,'b,'c,'d) prot_strand")
```

definition *transaction_strand* **where**
```
  "transaction_strand T ≡
    transaction_receive T@transaction_checks T@
    transaction_updates T@transaction_send T"
```

fun *transaction_proj* **where**
```
  "transaction_proj l (Transaction A B C D E F) = (
  let f = proj l
  in Transaction A B (f C) (f D) (f E) (f F))"
```

fun *transaction_star_proj* **where**
```
  "transaction_star_proj (Transaction A B C D E F) = (
  let f = filter has_LabelS
  in Transaction A B (f C) (f D) (f E) (f F))"
```

abbreviation *fv_transaction* **where**
  "fv_transaction T ≡ fv_{lsst} (transaction_strand T)"

abbreviation *bvars_transaction* **where**
  "bvars_transaction T ≡ $bvars_{lsst}$ (transaction_strand T)"

abbreviation *vars_transaction* **where**
  "vars_transaction T ≡ $vars_{lsst}$ (transaction_strand T)"

abbreviation *trms_transaction* **where**
  "trms_transaction T ≡ $trms_{lsst}$ (transaction_strand T)"

abbreviation *setops_transaction* **where**
  "setops_transaction T ≡ $setops_{sst}$ (unlabel (transaction_strand T))"

definition *wellformed_transaction* **where**
  "wellformed_transaction T ≡
    list_all is_Receive (unlabel (transaction_receive T)) ∧
    list_all is_Check_or_Assignment (unlabel (transaction_checks T)) ∧
    list_all is_Update (unlabel (transaction_updates T)) ∧
    list_all is_Send (unlabel (transaction_send T)) ∧
    distinct (map fst (transaction_decl T ())) ∧
    distinct (transaction_fresh T) ∧
    set (transaction_fresh T) ∩ fst ` set (transaction_decl T ()) = {} ∧
    set (transaction_fresh T) ∩ fv_{lsst} (transaction_receive T) = {} ∧
    set (transaction_fresh T) ∩ fv_{lsst} (transaction_checks T) = {} ∧
    set (transaction_fresh T) ∩ bvars_transaction T = {} ∧
    fv_transaction T ∩ bvars_transaction T = {} ∧
    wf'_{sst} (fst ` set (transaction_decl T ()) ∪ set (transaction_fresh T))
          (unlabel ($dual_{lsst}$ (transaction_strand T)))"

type_synonym *('a,'b,'c,'d) prot = "('a,'b,'c,'d) prot_transaction list"*

abbreviation *Var_Value_term* (<$\langle$_: value$\rangle_v$>) **where**
  "$\langle$n: value$\rangle_v$ ≡ Var (Var Value, n)::('a,'b,'c,'d) prot_term"

abbreviation *Var_SetType_term* (<$\langle$_: SetType$\rangle_v$>) **where**
  "$\langle$n: SetType$\rangle_v$ ≡ Var (Var SetType, n)::('a,'b,'c,'d) prot_term"

abbreviation *Var_AttackType_term* (<$\langle$_: AttackType$\rangle_v$>) **where**

```
  "⟨n: AttackType⟩_v ≡ Var (Var AttackType, n)::('a,'b,'c,'d) prot_term"

abbreviation Var_Atom_term (‹⟨_: _⟩_v›) where
  "⟨n: a⟩_v ≡ Var (Var (Atom a), n)::('a,'b,'c,'d) prot_term"

abbreviation Var_Comp_Fu_term (‹⟨_: _⟨_⟩⟩_v›) where
  "⟨n: f⟨T⟩⟩_v ≡ Var (Fun (Fu f) T, n)::('a,'b,'c,'d) prot_term"

abbreviation TAtom_Atom_term (‹⟨_⟩_τa›) where
  "⟨a⟩_τa ≡ Var (Atom a)::('a,'b,'c,'d) prot_term_type"

abbreviation TComp_Fu_term (‹⟨_ _⟩_τ›) where
  "⟨f T⟩_τ ≡ Fun (Fu f) T::('a,'b,'c,'d) prot_term_type"

abbreviation Fun_Fu_term (‹⟨_ _⟩_t›) where
  "⟨f T⟩_t ≡ Fun (Fu f) T::('a,'b,'c,'d) prot_term"

abbreviation Fun_Fu_const_term (‹⟨_⟩_c›) where
  "⟨c⟩_c ≡ Fun (Fu c) []::('a,'b,'c,'d) prot_term"

abbreviation Fun_Set_const_term (‹⟨_⟩_s›) where
  "⟨f⟩_s ≡ Fun (Set f) []::('a,'b,'c,'d) prot_term"

abbreviation Fun_Set_composed_term (‹⟨_⟨_⟩⟩_s›) where
  "⟨f⟨T⟩⟩_s ≡ Fun (Set f) T::('a,'b,'c,'d) prot_term"

abbreviation Fun_Abs_const_term (‹⟨_⟩_abs›) where
  "⟨a⟩_abs ≡ Fun (Abs a) []::('a,'b,'c,'d) prot_term"

abbreviation Fun_Attack_const_term (‹attack⟨_⟩›) where
  "attack⟨n⟩ ≡ Fun (Attack n) []::('a,'b,'c,'d) prot_term"

abbreviation prot_transaction1 (‹transaction_1 _ _ new _ _ _›) where
  "transaction_1 (S1::('a,'b,'c,'d) prot_strand) S2 new (B::('a,'b,'c,'d) prot_term list) S3 S4
  ≡ Transaction (λ(). []) (map the_Var B) S1 S2 S3 S4"

abbreviation prot_transaction2 (‹transaction_2 _ _  _ _›) where
  "transaction_2 (S1::('a,'b,'c,'d) prot_strand) S2 S3 S4
  ≡ Transaction (λ(). []) [] S1 S2 S3 S4"
```

3.1.2 Lemmata

```
lemma prot_atom_UNIV:
  "(UNIV::'b prot_atom set) = range Atom ∪ {Value, SetType, AttackType, Bottom, OccursSecType,
AbsValue}"
proof -
  have "a ∈ range Atom ∨ a = Value ∨ a = SetType ∨ a = AttackType ∨
        a = Bottom ∨ a = OccursSecType ∨ a = AbsValue"
    for a::"'b prot_atom"
    by (cases a) auto
  thus ?thesis by auto
qed

instance prot_atom::(finite) finite
by intro_classes (simp add: prot_atom_UNIV)

instantiation prot_atom::(enum) enum
begin
definition "enum_prot_atom == map Atom enum_class.enum@[Value, SetType, AttackType, Bottom,
OccursSecType, AbsValue]"
definition "enum_all_prot_atom P == list_all P (map Atom enum_class.enum@[Value, SetType, AttackType,
Bottom, OccursSecType, AbsValue])"
definition "enum_ex_prot_atom P == list_ex P (map Atom enum_class.enum@[Value, SetType, AttackType,
```

```
Bottom, OccursSecType, AbsValue])"

instance
proof intro_classes
  have *: "set (map Atom (enum_class.enum::'a list)) = range Atom"
          "distinct (enum_class.enum::'a list)"
    using UNIV_enum enum_distinct by auto

  show "(UNIV::'a prot_atom set) = set enum_class.enum"
    using *(1) by (simp add: prot_atom_UNIV enum_prot_atom_def)

  have "set (map Atom enum_class.enum) ∩ set [Value, SetType, AttackType, Bottom, OccursSecType,
AbsValue] = {}" by auto
  moreover have "inj_on Atom (set (enum_class.enum::'a list))" unfolding inj_on_def by auto
  hence "distinct (map Atom (enum_class.enum::'a list))" by (metis *(2) distinct_map)
  ultimately show "distinct (enum_class.enum::'a prot_atom list)" by (simp add: enum_prot_atom_def)

  have "Ball UNIV P ⟷ Ball (range Atom) P ∧ Ball {Value, SetType, AttackType, Bottom,
OccursSecType, AbsValue} P"
    for P::"'a prot_atom ⇒ bool"
    by (metis prot_atom_UNIV UNIV_I UnE)
  thus "enum_class.enum_all P = Ball (UNIV::'a prot_atom set) P" for P
    using *(1) Ball_set[of "map Atom enum_class.enum" P]
    by (auto simp add: enum_all_prot_atom_def)

  have "Bex UNIV P ⟷ Bex (range Atom) P ∨ Bex {Value, SetType, AttackType, Bottom, OccursSecType,
AbsValue} P"
    for P::"'a prot_atom ⇒ bool"
    by (metis prot_atom_UNIV UNIV_I UnE)
  thus "enum_class.enum_ex P = Bex (UNIV::'a prot_atom set) P" for P
    using *(1) Bex_set[of "map Atom enum_class.enum" P]
    by (auto simp add: enum_ex_prot_atom_def)
qed
end

lemma wellformed_transaction_cases:
  assumes "wellformed_transaction T"
  shows
      "(l,x) ∈ set (transaction_receive T) ⟹ ∃t. x = receive⟨t⟩" (is "?A ⟹ ?A'")
      "(l,x) ∈ set (transaction_checks T) ⟹
              (∃ac t s. x = ⟨ac: t ≐ s⟩) ∨ (∃ac t s. x = ⟨ac: t ∈ s⟩) ∨
              (∃X F G. x = ∀X⟨∨≠: F ∨∉: G⟩)"
          (is "?B ⟹ ?B'")
      "(l,x) ∈ set (transaction_updates T) ⟹
              (∃t s. x = insert⟨t,s⟩) ∨ (∃t s. x = delete⟨t,s⟩)" (is "?C ⟹ ?C'")
      "(l,x) ∈ set (transaction_send T) ⟹ ∃t. x = send⟨t⟩" (is "?D ⟹ ?D'")
proof -
  have a:
      "list_all is_Receive (unlabel (transaction_receive T))"
      "list_all is_Check_or_Assignment (unlabel (transaction_checks T))"
      "list_all is_Update (unlabel (transaction_updates T))"
      "list_all is_Send (unlabel (transaction_send T))"
    using assms unfolding wellformed_transaction_def by metis+

  note b = Ball_set unlabel_in
  note c = stateful_strand_step.collapse

  show "?A ⟹ ?A'" by (metis (mono_tags, lifting) a(1) b c(2))
  show "?B ⟹ ?B'" by (metis (no_types, lifting) a(2) b c(3,6,7))
  show "?C ⟹ ?C'" by (metis (mono_tags, lifting) a(3) b c(4,5))
  show "?D ⟹ ?D'" by (metis (mono_tags, lifting) a(4) b c(1))
qed
```

```
lemma wellformed_transaction_unlabel_cases:
  assumes "wellformed_transaction T"
  shows
      "x ∈ set (unlabel (transaction_receive T)) ⟹ ∃t. x = receive⟨t⟩" (is "?A ⟹ ?A'")
      "x ∈ set (unlabel (transaction_checks T)) ⟹
              (∃ac t s. x = ⟨ac: t ≐ s⟩) ∨ (∃ac t s. x = ⟨ac: t ∈ s⟩) ∨
              (∃X F G. x = ∀X⟨∨≠: F ∨∉: G⟩)"
        (is "?B ⟹ ?B'")
      "x ∈ set (unlabel (transaction_updates T)) ⟹
              (∃t s. x = insert⟨t,s⟩) ∨ (∃t s. x = delete⟨t,s⟩)" (is "?C ⟹ ?C'")
      "x ∈ set (unlabel (transaction_send T)) ⟹ ∃t. x = send⟨t⟩" (is "?D ⟹ ?D'")
proof -
  have a:
      "list_all is_Receive (unlabel (transaction_receive T))"
      "list_all is_Check_or_Assignment (unlabel (transaction_checks T))"
      "list_all is_Update (unlabel (transaction_updates T))"
      "list_all is_Send (unlabel (transaction_send T))"
    using assms unfolding wellformed_transaction_def by metis+

  note b = Ball_set
  note c = stateful_strand_step.collapse

  show "?A ⟹ ?A'" by (metis (mono_tags, lifting) a(1) b c(2))
  show "?B ⟹ ?B'" by (metis (no_types, lifting) a(2) b c(3,6,7))
  show "?C ⟹ ?C'" by (metis (mono_tags, lifting) a(3) b c(4,5))
  show "?D ⟹ ?D'" by (metis (mono_tags, lifting) a(4) b c(1))
qed

lemma transaction_strand_subsets[simp]:
  "set (transaction_receive T) ⊆ set (transaction_strand T)"
  "set (transaction_checks T) ⊆ set (transaction_strand T)"
  "set (transaction_updates T) ⊆ set (transaction_strand T)"
  "set (transaction_send T) ⊆ set (transaction_strand T)"
  "set (unlabel (transaction_receive T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_checks T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_updates T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_send T)) ⊆ set (unlabel (transaction_strand T))"
unfolding transaction_strand_def unlabel_def by force+

lemma transaction_strand_subst_subsets[simp]:
  "set (transaction_receive T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_checks T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_updates T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_send T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (unlabel (transaction_receive T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_checks T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_updates T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_send T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
unfolding transaction_strand_def unlabel_def subst_apply_labeled_stateful_strand_def by force+

lemma transaction_strand_dual_unfold:
  defines "f ≡ λS. dual_lsst S"
  shows "f (transaction_strand T) =
         f (transaction_receive T)@f (transaction_checks T)@
         f (transaction_updates T)@f (transaction_send T)"
using dual_lsst_append unfolding f_def transaction_strand_def by auto

lemma transaction_strand_unlabel_dual_unfold:
  defines "f ≡ λS. unlabel (dual_lsst S)"
  shows "f (transaction_strand T) =
         f (transaction_receive T)@f (transaction_checks T)@
         f (transaction_updates T)@f (transaction_send T)"
using unlabel_append dual_lsst_append unfolding f_def transaction_strand_def by auto
```

```
lemma transaction_dual_subst_unfold:
  "dual_lsst (transaction_strand T ·_lsst ϑ) =
    dual_lsst (transaction_receive T ·_lsst ϑ)@
    dual_lsst (transaction_checks T ·_lsst ϑ)@
    dual_lsst (transaction_updates T ·_lsst ϑ)@
    dual_lsst (transaction_send T ·_lsst ϑ)"
by (simp add: transaction_strand_def dual_lsst_append subst_lsst_append)

lemma transaction_dual_subst_unlabel_unfold:
  "unlabel (dual_lsst (transaction_strand T ·_lsst ϑ)) =
    unlabel (dual_lsst (transaction_receive T ·_lsst ϑ))@
    unlabel (dual_lsst (transaction_checks T ·_lsst ϑ))@
    unlabel (dual_lsst (transaction_updates T ·_lsst ϑ))@
    unlabel (dual_lsst (transaction_send T ·_lsst ϑ))"
by (simp add: transaction_dual_subst_unfold unlabel_append)

lemma trms_transaction_unfold:
  "trms_transaction T =
      trms_lsst (transaction_receive T) ∪ trms_lsst (transaction_checks T) ∪
      trms_lsst (transaction_updates T) ∪ trms_lsst (transaction_send T)"
by (metis trms_sst_append unlabel_append append_assoc transaction_strand_def)

lemma trms_transaction_subst_unfold:
  "trms_lsst (transaction_strand T ·_lsst ϑ) =
      trms_lsst (transaction_receive T ·_lsst ϑ) ∪ trms_lsst (transaction_checks T ·_lsst ϑ) ∪
      trms_lsst (transaction_updates T ·_lsst ϑ) ∪ trms_lsst (transaction_send T ·_lsst ϑ)"
by (metis trms_sst_append unlabel_append append_assoc transaction_strand_def subst_lsst_append)

lemma vars_transaction_unfold:
  "vars_transaction T =
      vars_lsst (transaction_receive T) ∪ vars_lsst (transaction_checks T) ∪
      vars_lsst (transaction_updates T) ∪ vars_lsst (transaction_send T)"
by (metis vars_sst_append unlabel_append append_assoc transaction_strand_def)

lemma vars_transaction_subst_unfold:
  "vars_lsst (transaction_strand T ·_lsst ϑ) =
      vars_lsst (transaction_receive T ·_lsst ϑ) ∪ vars_lsst (transaction_checks T ·_lsst ϑ) ∪
      vars_lsst (transaction_updates T ·_lsst ϑ) ∪ vars_lsst (transaction_send T ·_lsst ϑ)"
by (metis vars_sst_append unlabel_append append_assoc transaction_strand_def subst_lsst_append)

lemma fv_transaction_unfold:
  "fv_transaction T =
      fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_checks T) ∪
      fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)"
by (metis fv_sst_append unlabel_append append_assoc transaction_strand_def)

lemma fv_transaction_subst_unfold:
  "fv_lsst (transaction_strand T ·_lsst ϑ) =
      fv_lsst (transaction_receive T ·_lsst ϑ) ∪ fv_lsst (transaction_checks T ·_lsst ϑ) ∪
      fv_lsst (transaction_updates T ·_lsst ϑ) ∪ fv_lsst (transaction_send T ·_lsst ϑ)"
by (metis fv_sst_append unlabel_append append_assoc transaction_strand_def subst_lsst_append)

lemma bvars_transaction_unfold:
  "bvars_transaction T =
      bvars_lsst (transaction_receive T) ∪ bvars_lsst (transaction_checks T) ∪
      bvars_lsst (transaction_updates T) ∪ bvars_lsst (transaction_send T)"
by (metis bvars_sst_append unlabel_append append_assoc transaction_strand_def)

lemma bvars_transaction_subst_unfold:
  "bvars_lsst (transaction_strand T ·_lsst ϑ) =
      bvars_lsst (transaction_receive T ·_lsst ϑ) ∪ bvars_lsst (transaction_checks T ·_lsst ϑ) ∪
      bvars_lsst (transaction_updates T ·_lsst ϑ) ∪ bvars_lsst (transaction_send T ·_lsst ϑ)"
```

```
by (metis bvars_sst_append unlabel_append append_assoc transaction_strand_def subst_lsst_append)

lemma bvars_wellformed_transaction_unfold:
  assumes "wellformed_transaction T"
  shows "bvars_transaction T = bvars_lsst (transaction_checks T)" (is ?A)
    and "bvars_lsst (transaction_receive T) = {}" (is ?B)
    and "bvars_lsst (transaction_updates T) = {}" (is ?C)
    and "bvars_lsst (transaction_send T) = {}" (is ?D)
proof -
  have 0: "list_all is_Receive (unlabel (transaction_receive T))"
          "list_all is_Update (unlabel (transaction_updates T))"
          "list_all is_Send (unlabel (transaction_send T))"
    using assms unfolding wellformed_transaction_def by metis+

  have "filter is_NegChecks (unlabel (transaction_receive T)) = []"
       "filter is_NegChecks (unlabel (transaction_updates T)) = []"
       "filter is_NegChecks (unlabel (transaction_send T)) = []"
    using list_all_filter_nil[OF 0(1), of is_NegChecks]
          list_all_filter_nil[OF 0(2), of is_NegChecks]
          list_all_filter_nil[OF 0(3), of is_NegChecks]
          stateful_strand_step.distinct_disc(11,21,29,35,39,41)
    by blast+
  thus ?A ?B ?C ?D
    using bvars_transaction_unfold[of T]
          bvars_sst_NegChecks[of "unlabel (transaction_receive T)"]
          bvars_sst_NegChecks[of "unlabel (transaction_updates T)"]
          bvars_sst_NegChecks[of "unlabel (transaction_send T)"]
    by (metis bvars_sst_def UnionE emptyE list.set(1) list.simps(8) subsetI subset_Un_eq sup_commute)+
qed

lemma transaction_strand_memberD[dest]:
  assumes "x ∈ set (transaction_strand T)"
  shows "x ∈ set (transaction_receive T) ∨ x ∈ set (transaction_checks T) ∨
         x ∈ set (transaction_updates T) ∨ x ∈ set (transaction_send T)"
using assms by (simp add: transaction_strand_def)

lemma transaction_strand_unlabel_memberD[dest]:
  assumes "x ∈ set (unlabel (transaction_strand T))"
  shows "x ∈ set (unlabel (transaction_receive T)) ∨ x ∈ set (unlabel (transaction_checks T)) ∨
         x ∈ set (unlabel (transaction_updates T)) ∨ x ∈ set (unlabel (transaction_send T))"
using assms by (simp add: unlabel_def transaction_strand_def)

lemma wellformed_transaction_strand_memberD[dest]:
  assumes "wellformed_transaction T" and "(l,x) ∈ set (transaction_strand T)"
  shows
    "x = receive⟨ts⟩ ⟹ (l,x) ∈ set (transaction_receive T)" (is "?A ⟹ ?A'")
    "x = select⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (is "?B ⟹ ?B'")
    "x = ⟨t == s⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (is "?C ⟹ ?C'")
    "x = ⟨t in s⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (is "?D ⟹ ?D'")
    "x = ∀X⟨∨≠: F ∨∉: G⟩  ⟹ (l,x) ∈ set (transaction_checks T)" (is "?E ⟹ ?E'")
    "x = insert⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_updates T)" (is "?F ⟹ ?F'")
    "x = delete⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_updates T)" (is "?G ⟹ ?G'")
    "x = send⟨ts⟩ ⟹ (l,x) ∈ set (transaction_send T)" (is "?H ⟹ ?H'")
    "x = ⟨ac: t ≐ s⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (is "?I ⟹ ?I'")
    "x = ⟨ac: t ∈ s⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (is "?J ⟹ ?J'")
proof -
  have "(l,x) ∈ set (transaction_receive T) ∨ (l,x) ∈ set (transaction_checks T) ∨
        (l,x) ∈ set (transaction_updates T) ∨ (l,x) ∈ set (transaction_send T)"
    using assms(2) by auto
  thus "?A ⟹ ?A'" "?B ⟹ ?B'" "?C ⟹ ?C'" "?D ⟹ ?D'" "?E ⟹ ?E'"
       "?F ⟹ ?F'" "?G ⟹ ?G'" "?H ⟹ ?H'" "?I ⟹ ?I'" "?J ⟹ ?J'"
    using wellformed_transaction_cases[OF assms(1)] by fast+
qed
```

```
lemma wellformed_transaction_strand_unlabel_memberD[dest]:
  assumes "wellformed_transaction T" and "x ∈ set (unlabel (transaction_strand T))"
  shows
    "x = receive⟨ts⟩ ⟹ x ∈ set (unlabel (transaction_receive T))" (is "?A ⟹ ?A'")
    "x = select⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (is "?B ⟹ ?B'")
    "x = ⟨t == s⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (is "?C ⟹ ?C'")
    "x = ⟨t in s⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (is "?D ⟹ ?D'")
    "x = ∀X⟨∨≠: F ∨∉: G⟩  ⟹ x ∈ set (unlabel (transaction_checks T))" (is "?E ⟹ ?E'")
    "x = insert⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_updates T))" (is "?F ⟹ ?F'")
    "x = delete⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_updates T))" (is "?G ⟹ ?G'")
    "x = send⟨ts⟩ ⟹ x ∈ set (unlabel (transaction_send T))" (is "?H ⟹ ?H'")
    "x = ⟨ac: t ≐ s⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (is "?I ⟹ ?I'")
    "x = ⟨ac: t ∈ s⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (is "?J ⟹ ?J'")
proof -
  have "x ∈ set (unlabel (transaction_receive T)) ∨ x ∈ set (unlabel (transaction_checks T)) ∨
        x ∈ set (unlabel (transaction_updates T)) ∨ x ∈ set (unlabel (transaction_send T))"
    using assms(2) by auto
  thus "?A ⟹ ?A'" "?B ⟹ ?B'" "?C ⟹ ?C'" "?D ⟹ ?D'" "?E ⟹ ?E'"
       "?F ⟹ ?F'" "?G ⟹ ?G'" "?H ⟹ ?H'" "?I ⟹ ?I'" "?J ⟹ ?J'"
    using wellformed_transaction_unlabel_cases[OF assms(1)] by fast+
qed

lemma wellformed_transaction_send_receive_trm_cases:
  assumes T: "wellformed_transaction T"
  shows "t ∈ trms_lsst (transaction_receive T) ⟹ ∃ts. t ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel
(transaction_receive T))"
    and "t ∈ trms_lsst (transaction_send T) ⟹ ∃ts. t ∈ set ts ∧ send⟨ts⟩ ∈ set (unlabel
(transaction_send T))"
using wellformed_transaction_unlabel_cases(1,4)[OF T]
      trms_sst_in[of t "unlabel (transaction_receive T)"]
      trms_sst_in[of t "unlabel (transaction_send T)"]
by fastforce+

lemma wellformed_transaction_send_receive_subst_trm_cases:
  assumes T: "wellformed_transaction T"
  shows "t ∈ trms_lsst (transaction_receive T) ·set ϑ ⟹ ∃ts. t ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel
(transaction_receive T ·lsst ϑ))"
    and "t ∈ trms_lsst (transaction_send T) ·set ϑ ⟹ ∃ts. t ∈ set ts ∧ send⟨ts⟩ ∈ set (unlabel
(transaction_send T ·lsst ϑ))"
proof -
  assume "t ∈ trms_lsst (transaction_receive T) ·set ϑ"
  then obtain s where s: "s ∈ trms_lsst (transaction_receive T)" "t = s · ϑ"
    by blast
  hence "∃ts. s ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel (transaction_receive T))"
    using wellformed_transaction_send_receive_trm_cases(1)[OF T] by simp
  thus "∃ts. t ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel (transaction_receive T ·lsst ϑ))"
    using s(2) unlabel_subst[of _ ϑ] subst_set_map[of s _ ϑ]
          stateful_strand_step_subst_inI(2)[of _ "unlabel (transaction_receive T)" ϑ]
    by metis
next
  assume "t ∈ trms_lsst (transaction_send T) ·set ϑ"
  then obtain s where s: "s ∈ trms_lsst (transaction_send T)" "t = s · ϑ"
    by blast
  hence "∃ts. s ∈ set ts ∧ send⟨ts⟩ ∈ set (unlabel (transaction_send T))"
    using wellformed_transaction_send_receive_trm_cases(2)[OF T] by simp
  thus "∃ts. t ∈ set ts ∧ send⟨ts⟩ ∈ set (unlabel (transaction_send T ·lsst ϑ))"
    using s(2) unlabel_subst[of _ ϑ] subst_set_map[of s _ ϑ]
          stateful_strand_step_subst_inI(1)[of _ "unlabel (transaction_send T)" ϑ]
    by metis
qed

lemma wellformed_transaction_send_receive_fv_subset:
```

```
  assumes T: "wellformed_transaction T"
  shows "t ∈ trms_lsst (transaction_receive T) ⟹ fv t ⊆ fv_transaction T" (is "?A ⟹ ?A'")
    and "t ∈ trms_lsst (transaction_send T) ⟹ fv t ⊆ fv_transaction T" (is "?B ⟹ ?B'")
proof -
  let ?P = "∃ts. t ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel (transaction_strand T))"
  let ?Q = "∃ts. t ∈ set ts ∧ send⟨ts⟩ ∈ set (unlabel (transaction_strand T))"

  have *: "t ∈ trms_lsst (transaction_receive T) ⟹ ?P" "t ∈ trms_lsst (transaction_send T) ⟹ ?Q"
    using wellformed_transaction_send_receive_trm_cases[OF T, of t]
    unfolding transaction_strand_def by force+

  show "?A ⟹ ?A'" using *(1) by (induct "transaction_strand T") (simp, force)
  show "?B ⟹ ?B'" using *(2) by (induct "transaction_strand T") (simp, force)
qed

lemma dual_wellformed_transaction_ident_cases[dest]:
  "list_all is_Assignment (unlabel S) ⟹ dual_lsst S = S"
  "list_all is_Check (unlabel S) ⟹ dual_lsst S = S"
  "list_all is_Update (unlabel S) ⟹ dual_lsst S = S"
proof (induction S)
  case (Cons s S)
  obtain l x where s: "s = (l,x)" by force
  { case 1 thus ?case using Cons s unfolding unlabel_def dual_lsst_def by (cases x) auto }
  { case 2 thus ?case using Cons s unfolding unlabel_def dual_lsst_def by (cases x) auto }
  { case 3 thus ?case using Cons s unfolding unlabel_def dual_lsst_def by (cases x) auto }
qed simp_all

lemma wellformed_transaction_wf_sst:
  fixes T::"('a, 'b, 'c, 'd) prot_transaction"
  assumes T: "wellformed_transaction T"
  shows "wf'_sst (fst ` set (transaction_decl T ()) ∪ set (transaction_fresh T))
              (unlabel (dual_lsst (transaction_strand T)))"
    and "fv_transaction T ∩ bvars_transaction T = {}"
using T unfolding wellformed_transaction_def by simp_all

lemma dual_wellformed_transaction_ident_cases'[dest]:
  assumes "wellformed_transaction T"
  shows "dual_lsst (transaction_checks T) = transaction_checks T" (is ?A)
        "dual_lsst (transaction_updates T) = transaction_updates T" (is ?B)
proof -
  have "list_all is_Check_or_Assignment (unlabel (transaction_checks T))"
       "list_all is_Update (unlabel (transaction_updates T))"
    using assms is_Check_or_Assignment_iff unfolding wellformed_transaction_def by auto
  thus ?A ?B
    using dual_lsst_list_all_same(9)[of "transaction_checks T"]
          dual_lsst_list_all_same(8)[of "transaction_updates T"]
    by (blast, blast)
qed

lemma dual_transaction_strand:
  assumes "wellformed_transaction T"
  shows "dual_lsst (transaction_strand T) =
         dual_lsst (transaction_receive T)@transaction_checks T@
         transaction_updates T@dual_lsst (transaction_send T)"
using dual_wellformed_transaction_ident_cases'[OF assms] dual_lsst_append
unfolding transaction_strand_def by metis

lemma dual_unlabel_transaction_strand:
  assumes "wellformed_transaction T"
  shows "unlabel (dual_lsst (transaction_strand T)) =
         (unlabel (dual_lsst (transaction_receive T)))@(unlabel (transaction_checks T))@
         (unlabel (transaction_updates T))@(unlabel (dual_lsst (transaction_send T)))"
using dual_transaction_strand[OF assms] by (simp add: unlabel_def)
```

```
lemma dual_transaction_strand_subst:
  assumes "wellformed_transaction T"
  shows "dual_lsst (transaction_strand T ·_lsst δ) =
          (dual_lsst (transaction_receive T)@transaction_checks T@
          transaction_updates T@dual_lsst (transaction_send T)) ·_lsst δ"
proof -
  have "dual_lsst (transaction_strand T ·_lsst δ) = dual_lsst (transaction_strand T) ·_lsst δ"
    using dual_lsst_subst by metis
  thus ?thesis using dual_transaction_strand[OF assms] by argo
qed

lemma dual_transaction_ik_is_transaction_send:
  assumes "wellformed_transaction T"
  shows "ik_sst (unlabel (dual_lsst (transaction_strand T))) = trms_sst (unlabel (transaction_send T))"
    (is "?A = ?B")
proof -
  { fix t assume "t ∈ ?A"
    then obtain ts where ts:
        "t ∈ set ts" "receive⟨ts⟩ ∈ set (unlabel (dual_lsst (transaction_strand T)))"
      by (auto simp add: ik_sst_def)
    hence *: "send⟨ts⟩ ∈ set (unlabel (transaction_strand T))"
      using dual_lsst_unlabel_steps_iff(1) by metis
    have "t ∈ ?B"
      using ts(1) wellformed_transaction_strand_unlabel_memberD(8)[OF assms *, of ts] by force
  } moreover {
    fix t assume "t ∈ ?B"
    then obtain ts where ts:
        "t ∈ set ts" "send⟨ts⟩ ∈ set (unlabel (transaction_send T))"
      using wellformed_transaction_unlabel_cases(4)[OF assms] by fastforce
    hence "receive⟨ts⟩ ∈ set (unlabel (dual_lsst (transaction_send T)))"
      using dual_lsst_unlabel_steps_iff(1) by metis
    hence "receive⟨ts⟩ ∈ set (unlabel (dual_lsst (transaction_strand T)))"
      using dual_unlabel_transaction_strand[OF assms] by simp
    hence "t ∈ ?A" using ts(1) by (auto simp add: ik_sst_def)
  } ultimately show "?A = ?B" by auto
qed

lemma dual_transaction_ik_is_transaction_send':
  fixes δ::"('a,'b,'c,'d) prot_subst"
  assumes "wellformed_transaction T"
  shows "ik_sst (unlabel (dual_lsst (transaction_strand T ·_lsst δ))) =
         trms_sst (unlabel (transaction_send T)) ·_set δ" (is "?A = ?B")
using dual_transaction_ik_is_transaction_send[OF assms]
      subst_lsst_unlabel[of "dual_lsst (transaction_strand T)" δ]
      ik_sst_subst[of "unlabel (dual_lsst (transaction_strand T))" δ]
      dual_lsst_subst[of "transaction_strand T" δ]
by auto

lemma db_sst_transaction_prefix_eq:
  assumes T: "wellformed_transaction T"
    and S: "prefix S (transaction_receive T@transaction_checks T)"
  shows "db_lsst A = db_lsst (A@dual_lsst (S ·_lsst δ))"
proof -
  let ?T1 = "transaction_receive T"
  let ?T2 = "transaction_checks T"

  have *: "prefix (unlabel S) (unlabel (?T1@?T2))" using S prefix_unlabel by blast

  have "list_all is_Receive (unlabel ?T1)"
       "list_all is_Check_or_Assignment (unlabel ?T2)"
    using T by (simp_all add: wellformed_transaction_def)
  hence "∀b ∈ set (unlabel ?T1). ¬is_Insert b ∧ ¬is_Delete b"
```

```
      "∀b ∈ set (unlabel ?T2). ¬is_Insert b ∧ ¬is_Delete b"
    by (metis (mono_tags, lifting) Ball_set stateful_strand_step.distinct_disc(16,18),
        metis (mono_tags, lifting) Ball_set stateful_strand_step.distinct_disc(24,26,33,35,37,39))
  hence "∀b ∈ set (unlabel (?T1@?T2)). ¬is_Insert b ∧ ¬is_Delete b"
    by (auto simp add: unlabel_def)
  hence "∀b ∈ set (unlabel S). ¬is_Insert b ∧ ¬is_Delete b"
    using * unfolding prefix_def by fastforce
  hence "∀b ∈ set (unlabel (dual_lsst S) ·sst δ). ¬is_Insert b ∧ ¬is_Delete b"
  proof (induction S)
    case (Cons a S)
    then obtain l b where "a = (l,b)" by (metis surj_pair)
    thus ?case
      using Cons unfolding dual_lsst_def unlabel_def subst_apply_stateful_strand_def
      by (cases b) auto
  qed simp
  hence **: "∀b ∈ set (unlabel (dual_lsst (S ·lsst δ))). ¬is_Insert b ∧ ¬is_Delete b"
    by (metis dual_lsst_subst_unlabel)

  show ?thesis
    using db_sst_no_upd_append[OF **] unlabel_append
    unfolding db_sst_def by metis
qed

lemma db_lsst_dual_lsst_set_ex:
   assumes "d ∈ set (db'_lsst (dual_lsst A ·lsst ϑ) ℐ D)"
    "∀t u. insert⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃s. u = Fun (Set s) [])"
    "∀t u. delete⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃s. u = Fun (Set s) [])"
    "∀d ∈ set D. ∃s. snd d = Fun (Set s) []"
  shows "∃s. snd d = Fun (Set s) []"
  using assms
proof (induction A arbitrary: D)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)

  have 1: "unlabel (dual_lsst (a#A) ·lsst ϑ) = receive⟨ts ·list ϑ⟩#unlabel (dual_lsst A ·lsst ϑ)"
    when "b = send⟨ts⟩" for ts
    by (simp add: a that subst_lsst_unlabel_cons)

  have 2: "unlabel (dual_lsst (a#A) ·lsst ϑ) = send⟨ts ·list ϑ⟩#unlabel (dual_lsst A ·lsst ϑ)"
    when "b = receive⟨ts⟩" for ts
    by (simp add: a that subst_lsst_unlabel_cons)

  have 3: "unlabel (dual_lsst (a#A) ·lsst ϑ) = (b ·sstp ϑ)#unlabel (dual_lsst A ·lsst ϑ)"
    when "∄ts. b = send⟨ts⟩ ∨ b = receive⟨ts⟩"
    using a that dual_lsst_Cons subst_lsst_unlabel_cons[of l b]
    by (cases b) auto

  show ?case using 1 2 3 a Cons by (cases b) fastforce+
qed simp

lemma is_Fun_SetE[elim]:
  assumes t: "is_Fun_Set t"
  obtains s where "t = Fun (Set s) []"
proof (cases t)
  case (Fun f T)
  then obtain s where "f = Set s" using t unfolding is_Fun_Set_def by (cases f) force+
  moreover have "T = []" using Fun t unfolding is_Fun_Set_def by (cases T) auto
  ultimately show ?thesis using Fun that by fast
qed (use t is_Fun_Set_def in fast)

lemma Fun_Set_InSet_iff:
  "(u = ⟨a: Var x ∈ Fun (Set s) []⟩) ⟷
   (is_InSet u ∧ is_Var (the_elem_term u) ∧ is_Fun_Set (the_set_term u) ∧
```

```
    the_Set (the_Fun (the_set_term u)) = s ∧ the_Var (the_elem_term u) = x ∧ the_check u = a)"
  (is "?A ⟷ ?B")
proof
  show "?A ⟹ ?B" unfolding is_Fun_Set_def by auto

  assume B: ?B
  thus ?A
  proof (cases u)
    case (InSet b t t')
    hence "b = a" "t = Var x" "t' = Fun (Set s) []"
      using B by (simp, fastforce, fastforce)
    thus ?thesis using InSet by fast
  qed auto
qed

lemma Fun_Set_NotInSet_iff:
  "(u = ⟨Var x not in Fun (Set s) []⟩) ⟷
   (is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1 ∧
    is_Var (fst (hd (the_ins u))) ∧ is_Fun_Set (snd (hd (the_ins u)))) ∧
    the_Set (the_Fun (snd (hd (the_ins u)))) = s ∧ the_Var (fst (hd (the_ins u))) = x"
  (is "?A ⟷ ?B")
proof
  show "?A ⟹ ?B" unfolding is_Fun_Set_def by auto

  assume B: ?B
  show ?A
  proof (cases u)
    case (NegChecks X F F')
    hence "X = []" "F = []"
      using B by auto
    moreover have "fst (hd (the_ins u)) = Var x" "snd (hd (the_ins u)) = Fun (Set s) []"
      using B is_Fun_SetE[of "snd (hd (the_ins u))"]
      by (force, fastforce)
    hence "F' = [(Var x, Fun (Set s) [])]"
      using NegChecks B by (cases "the_ins u") auto
    ultimately show ?thesis using NegChecks by fast
  qed (use B in auto)
qed

lemma is_Fun_Set_exi: "is_Fun_Set x ⟷ (∃s. x = Fun (Set s) [])"
by (metis prot_fun.collapse(2) term.collapse(2) prot_fun.disc(11) term.disc(2)
          term.sel(2,4) is_Fun_Set_def un_Fun1_def)

lemma is_Fun_Set_subst:
  assumes "is_Fun_Set S'"
  shows "is_Fun_Set (S' · σ)"
using assms by (fastforce simp add: is_Fun_Set_def)

lemma is_Update_in_transaction_updates:
  assumes tu: "is_Update t"
  assumes t: "t ∈ set (unlabel (transaction_strand TT))"
  assumes vt: "wellformed_transaction TT"
  shows "t ∈ set (unlabel (transaction_updates TT))"
using t tu vt unfolding transaction_strand_def wellformed_transaction_def list_all_iff
by (auto simp add: unlabel_append)

lemma transaction_proj_member:
  assumes "T ∈ set P"
  shows "transaction_proj n T ∈ set (map (transaction_proj n) P)"
using assms by simp

lemma transaction_strand_proj:
  "transaction_strand (transaction_proj n T) = proj n (transaction_strand T)"
```

```
proof -
  obtain A B C D E F where "T = Transaction A B C D E F" by (cases T) simp
  thus ?thesis
    using transaction_proj.simps[of n A B C D E F]
    unfolding transaction_strand_def proj_def Let_def by auto
qed

lemma transaction_proj_decl_eq:
  "transaction_decl (transaction_proj n T) = transaction_decl T"
proof -
  obtain A B C D E F where "T = Transaction A B C D E F" by (cases T) simp
  thus ?thesis
    using transaction_proj.simps[of n A B C D E F]
    unfolding proj_def Let_def by auto
qed

lemma transaction_proj_fresh_eq:
  "transaction_fresh (transaction_proj n T) = transaction_fresh T"
proof -
  obtain A B C D E F where "T = Transaction A B C D E F" by (cases T) simp
  thus ?thesis
    using transaction_proj.simps[of n A B C D E F]
    unfolding proj_def Let_def by auto
qed

lemma transaction_proj_trms_subset:
  "trms_transaction (transaction_proj n T) ⊆ trms_transaction T"
proof -
  obtain A B C D E F where "T = Transaction A B C D E F" by (cases T) simp
  thus ?thesis
    using transaction_proj.simps[of n A B C D E F] trms_sst_proj_subset(1)[of n]
    unfolding transaction_fresh_def Let_def transaction_strand_def by auto
qed

lemma transaction_proj_vars_subset:
  "vars_transaction (transaction_proj n T) ⊆ vars_transaction T"
proof -
  obtain A B C D E F where "T = Transaction A B C D E F" by (cases T) simp
  thus ?thesis
    using transaction_proj.simps[of n A B C D E F]
          sst_vars_proj_subset(3)[of n "transaction_strand T"]
    unfolding transaction_fresh_def Let_def transaction_strand_def by simp
qed

lemma transaction_proj_labels:
  fixes T::"('a,'b,'c,'d) prot_transaction"
  shows "list_all (λa. has_LabelN l a ∨ has_LabelS a) (transaction_strand (transaction_proj l T))"
proof -
  define h where "h ≡ λa::('a,'b,'c,'d) prot_strand_step. has_LabelN l a ∨ has_LabelS a"
  let ?f = "filter h"
  let ?g = "list_all h"

  obtain T1 T2 T3 T4 T5 T6 where T: "T = Transaction T1 T2 T3 T4 T5 T6" by (cases T) simp

  note 0 = transaction_proj.simps[unfolded Let_def, of l T1 T2 T3 T4 T5 T6]

  show ?thesis using T 0 unfolding list_all_iff proj_def by auto
qed

lemma transaction_proj_ident_iff:
  fixes T::"('a,'b,'c,'d) prot_transaction"
  shows "list_all (λa. has_LabelN l a ∨ has_LabelS a) (transaction_strand T) ⟷
         transaction_proj l T = T"
```

```
    (is "?A ⟷ ?B")
proof
  obtain T1 T2 T3 T4 T5 T6 where T: "T = Transaction T1 T2 T3 T4 T5 T6" by (cases T) simp
  hence "transaction_strand T = T3@T4@T5@T6" unfolding transaction_strand_def by simp
  thus "?A ⟹ ?B"
    using T transaction_proj.simps[unfolded Let_def, of l T1 T2 T3 T4 T5 T6]
    unfolding list_all_iff proj_def by auto

  show "?B ⟹ ?A" using transaction_proj_labels[of l T] by presburger
qed

lemma transaction_proj_idem:
  fixes T::"('a,'b,'c,'d) prot_transaction"
  shows "transaction_proj l (transaction_proj l T) = transaction_proj l T"
by (meson transaction_proj_ident_iff transaction_proj_labels)

lemma transaction_proj_ball_subst:
  assumes
    "set Ps = (λn. map (transaction_proj n) P) ` set L"
    "∀p ∈ set Ps. Q p"
  shows "∀l ∈ set L. Q (map (transaction_proj l) P)"
using assms by auto

lemma transaction_star_proj_has_star_labels:
  "list_all has_LabelS (transaction_strand (transaction_star_proj T))"
proof -
  let ?f = "filter has_LabelS"

  obtain T1 T2 T3 T4 T5 T6 where T: "T = Transaction T1 T2 T3 T4 T5 T6" by (cases T) simp
  hence T': "transaction_strand (transaction_star_proj T) = ?f T3@?f T4@?f T5@?f T6"
    using transaction_star_proj.simps[unfolded Let_def, of T1 T2 T3 T4 T5 T6]
    unfolding transaction_strand_def by auto

  show ?thesis using Ball_set unfolding T' by fastforce
qed

lemma transaction_star_proj_ident_iff:
  "list_all has_LabelS (transaction_strand T) ⟷ transaction_star_proj T = T" (is "?A ⟷ ?B")
proof
  obtain T1 T2 T3 T4 T5 T6 where T: "T = Transaction T1 T2 T3 T4 T5 T6" by (cases T) simp
  hence T': "transaction_strand T = T3@T4@T5@T6" unfolding transaction_strand_def by simp

  show "?A ⟹ ?B" using T T' unfolding list_all_iff by auto
  show "?B ⟹ ?A" using transaction_star_proj_has_star_labels[of T] by auto
qed

lemma transaction_star_proj_negates_transaction_proj:
  "transaction_star_proj (transaction_proj l T) = transaction_star_proj T" (is "?A l T")
  "k ≠ l ⟹ transaction_proj k (transaction_proj l T) = transaction_star_proj T" (is "?B ⟹ ?B'")
proof -
  show "?A l T" for l T
  proof -
    obtain T1 T2 T3 T4 T5 T6 where T: "T = Transaction T1 T2 T3 T4 T5 T6" by (cases T) simp
    thus ?thesis
      by (metis dbproj_def transaction_proj.simps transaction_star_proj.simps proj_dbproj(2))
  qed
  thus "?B ⟹ ?B'"
    by (metis (no_types) has_LabelS_proj_iff_not_has_LabelN proj_elims_label
        transaction_star_proj_ident_iff transaction_strand_proj)
qed

lemma transaction_updates_send_ex_iff:
  fixes T::"('a,'b,'c,'d) prot_transaction"
```

```
  assumes "list_all is_Receive (unlabel (transaction_receive T))"
          "list_all is_Check_or_Assignment (unlabel (transaction_checks T))"
          "list_all is_Update (unlabel (transaction_updates T))"
          "list_all is_Send (unlabel (transaction_send T))"
  shows "transaction_updates T ≠ [] ∨ transaction_send T ≠ [] ⟷
         list_ex (λa. is_Send (snd a) ∨ is_Update (snd a)) (transaction_strand T)"
proof -
  define f where "f ≡ λa::('a,'b,'c,'d) prot_strand_step. is_Send (snd a) ∨ is_Update (snd a)"

  have 0: "list_all (λa. ¬(f a)) (transaction_receive T)"
          "list_all (λa. ¬(f a)) (transaction_checks T)"
          "list_all (λa. (f a)) (transaction_updates T)"
          "list_all (λa. (f a)) (transaction_send T)"
    using assms unfolding list_all_iff unlabel_def f_def by auto

  have 1:
      "list_ex f (transaction_strand T) ⟷
       list_ex f (transaction_updates T) ∨ list_ex f (transaction_send T)"
    using 0 unfolding list_all_iff list_ex_iff transaction_strand_def by auto

  have 2: "A ≠ [] ⟷ list_ex f A" when "list_all f A" for A
    using that by (induct A) auto

  thus ?thesis
    using 1 2[OF 0(3)] 2[OF 0(4)] unfolding list_all_iff list_ex_iff f_def[symmetric] by meson
qed

end
```

3.2 Term Abstraction

```
theory Term_Abstraction
  imports Transactions
begin
```

3.2.1 Definitions

```
fun to_abs (‹α₀›) where
  "α₀ [] _ = {}"
| "α₀ ((Fun (Val m) [],Fun (Set s) S)#D) n =
    (if m = n then insert s (α₀ D n) else α₀ D n)"
| "α₀ (_#D) n = α₀ D n"

fun abs_apply_term (infixl ‹·α› 67) where
  "Var x ·α α = Var x"
| "Fun (Val n) T ·α α = Fun (Abs (α n)) (map (λt. t ·α α) T)"
| "Fun f T ·α α = Fun f (map (λt. t ·α α) T)"

definition abs_apply_list (infixl ‹·αlist› 67) where
  "M ·αlist α ≡ map (λt. t ·α α) M"

definition abs_apply_terms (infixl ‹·αset› 67) where
  "M ·αset α ≡ (λt. t ·α α) ` M"

definition abs_apply_pairs (infixl ‹·αpairs› 67) where
  "F ·αpairs α ≡ map (λ(s,t). (s ·α α, t ·α α)) F"

definition abs_apply_strand_step (infixl ‹·αstp› 67) where
  "s ·αstp α ≡ (case s of
    (l,send⟨ts⟩) ⇒ (l,send⟨ts ·αlist α⟩)
  | (l,receive⟨ts⟩) ⇒ (l,receive⟨ts ·αlist α⟩)
  | (l,⟨ac: t ≐ t'⟩) ⇒ (l,⟨ac: (t ·α α) ≐ (t' ·α α)⟩)
```

```
  | (l,insert⟨t,t'⟩) ⇒ (l,insert⟨t ·α α,t' ·α α⟩)
  | (l,delete⟨t,t'⟩) ⇒ (l,delete⟨t ·α α,t' ·α α⟩)
  | (l,⟨ac: t ∈ t'⟩) ⇒ (l,⟨ac: (t ·α α) ∈ (t' ·α α)⟩)
  | (l,∀X⟨∨≠: F ∨∉: F'⟩) ⇒ (l,∀X⟨∨≠: (F ·αpairs α) ∨∉: (F' ·αpairs α)⟩))"

definition abs_apply_strand (infixl ‹·αst› 67) where
  "S ·αst α ≡ map (λx. x ·αstp α) S"
```

3.2.2 Lemmata

```
lemma to_abs_alt_def:
  "α0 D n = {s. ∃S. (Fun (Val n) [], Fun (Set s) S) ∈ set D}"
by (induct D n rule: to_abs.induct) auto

lemma abs_term_apply_const[simp]:
  "is_Val f ⟹ Fun f [] ·α a = Fun (Abs (a (the_Val f))) []"
  "¬is_Val f ⟹ Fun f [] ·α a = Fun f []"
by (cases f; auto)+

lemma abs_fv: "fv (t ·α a) = fv t"
by (induct t a rule: abs_apply_term.induct) auto

lemma abs_eq_if_no_Val:
  assumes "∀f ∈ funs_term t. ¬is_Val f"
  shows "t ·α a = t ·α b"
using assms
proof (induction t)
  case (Fun f T) thus ?case by (cases f) simp_all
qed simp

lemma abs_list_set_is_set_abs_set: "set (M ·αlist α) = (set M) ·αset α"
unfolding abs_apply_list_def abs_apply_terms_def by simp

lemma abs_set_empty[simp]: "{} ·αset α = {}"
unfolding abs_apply_terms_def by simp

lemma abs_in:
  assumes "t ∈ M"
  shows "t ·α α ∈ M ·αset α"
using assms unfolding abs_apply_terms_def
by (induct t α rule: abs_apply_term.induct) blast+

lemma abs_set_union: "(A ∪ B) ·αset a = (A ·αset a) ∪ (B ·αset a)"
unfolding abs_apply_terms_def
by auto

lemma abs_subterms: "subterms (t ·α α) = subterms t ·αset α"
proof (induction t)
  case (Fun f T) thus ?case by (cases f) (auto simp add: abs_apply_terms_def)
qed (simp add: abs_apply_terms_def)

lemma abs_subterms_in: "s ∈ subterms t ⟹ s ·α a ∈ subterms (t ·α a)"
proof (induction t)
  case (Fun f T) thus ?case by (cases f) auto
qed simp

lemma abs_ik_append: "(iksst (A@B) ·set I) ·αset a = (iksst A ·set I) ·αset a ∪ (iksst B ·set I) ·αset a"
unfolding abs_apply_terms_def iksst_def
by fastforce

lemma to_abs_in:
  assumes "(Fun (Val n) [], Fun (Set s) []) ∈ set D"
  shows "s ∈ α0 D n"
```

```
using assms by (induct rule: to_abs.induct) auto

lemma to_abs_empty_iff_notin_db:
  "Fun (Val n) [] ·α α0 D = Fun (Abs {}) [] ⟷ (∄s S. (Fun (Val n) [], Fun (Set s) S) ∈ set D)"
by (simp add: to_abs_alt_def)

lemma to_abs_list_insert:
  assumes "Fun (Val n) [] ≠ t"
  shows "α0 D n = α0 (List.insert (t,s) D) n"
using assms to_abs_alt_def[of D n] to_abs_alt_def[of "List.insert (t,s) D" n]
by auto

lemma to_abs_list_insert':
  "insert s (α0 D n) = α0 (List.insert (Fun (Val n) [], Fun (Set s) S) D) n"
using to_abs_alt_def[of D n]
      to_abs_alt_def[of "List.insert (Fun (Val n) [], Fun (Set s) S) D" n]
by auto

lemma to_abs_list_remove_all:
  assumes "Fun (Val n) [] ≠ t"
  shows "α0 D n = α0 (List.removeAll (t,s) D) n"
using assms to_abs_alt_def[of D n] to_abs_alt_def[of "List.removeAll (t,s) D" n]
by auto

lemma to_abs_list_remove_all':
  "α0 D n - {s} = α0 (filter (λd. ∄S. d = (Fun (Val n) [], Fun (Set s) S)) D) n"
using to_abs_alt_def[of D n]
      to_abs_alt_def[of "filter (λd. ∄S. d = (Fun (Val n) [], Fun (Set s) S)) D" n]
by auto

lemma to_abs_db_sst_append:
  assumes "∀u s. insert⟨u, s⟩ ∈ set B ⟶ Fun (Val n) [] ≠ u · I"
    and "∀u s. delete⟨u, s⟩ ∈ set B ⟶ Fun (Val n) [] ≠ u · I"
  shows "α0 (db'_sst A I D) n = α0 (db'_sst (A@B) I D) n"
using assms
proof (induction B rule: List.rev_induct)
  case (snoc b B)
  hence IH: "α0 (db'_sst A I D) n = α0 (db'_sst (A@B) I D) n" by auto
  have *: "∀u s. b = insert⟨u,s⟩ ⟶ Fun (Val n) [] ≠ u · I"
          "∀u s. b = delete⟨u,s⟩ ⟶ Fun (Val n) [] ≠ u · I"
    using snoc.prems by simp_all
  show ?case
  proof (cases b)
    case (Insert u s)
    hence **: "db'_sst (A@B@[b]) I D = List.insert (u · I,s · I) (db'_sst (A@B) I D)"
      using db_sst_append[of "A@B" "[b]"] by simp
    have "Fun (Val n) [] ≠ u · I" using *(1) Insert by auto
    thus ?thesis using IH ** to_abs_list_insert by metis
  next
    case (Delete u s)
    hence **: "db'_sst (A@B@[b]) I D = List.removeAll (u · I,s · I) (db'_sst (A@B) I D)"
      using db_sst_append[of "A@B" "[b]"] by simp
    have "Fun (Val n) [] ≠ u · I" using *(2) Delete by auto
    thus ?thesis using IH ** to_abs_list_remove_all by metis
  qed (simp_all add: db_sst_no_upd_append[of "[b]" "A@B"] IH)
qed simp

lemma to_abs_neq_imp_db_update:
  assumes "α0 (db_sst A I) n ≠ α0 (db_sst (A@B) I) n"
  shows "∃u s. u · I = Fun (Val n) [] ∧ (insert⟨u,s⟩ ∈ set B ∨ delete⟨u,s⟩ ∈ set B)"
proof -
  { fix D have ?thesis when "α0 D n ≠ α0 (db'_sst B I D) n" using that
    proof (induction B I D rule: db'_sst.induct)
```

```
      case 2 thus ?case
        by (metis db'ₛₛₜ.simps(2) list.set_intros(1,2) subst_apply_pair_pair to_abs_list_insert)
    next
      case 3 thus ?case
        by (metis db'ₛₛₜ.simps(3) list.set_intros(1,2) subst_apply_pair_pair to_abs_list_remove_all)
    qed simp_all
  } thus ?thesis using assms by (metis dbₛₛₜ_append dbₛₛₜ_def)
qed

lemma abs_term_subst_eq:
  fixes δ ϑ::"(('a,'b,'c,'d) prot_fun, ('e,'f prot_atom) term × nat) subst"
  assumes "∀x ∈ fv t. δ x ·α a = ϑ x ·α b"
    and "∄n T. Fun (Val n) T ∈ subterms t"
  shows "t · δ ·α a = t · ϑ ·α b"
using assms
proof (induction t)
  case (Fun f T) thus ?case
  proof (cases f)
    case (Val n)
    hence False using Fun.prems(2) by blast
    thus ?thesis by metis
  qed auto
qed simp

lemma abs_term_subst_eq':
  fixes δ ϑ::"(('a,'b,'c,'d) prot_fun, ('e,'f prot_atom) term × nat) subst"
  assumes "∀x ∈ fv t. δ x ·α a = ϑ x"
    and "∄n T. Fun (Val n) T ∈ subterms t"
  shows "t · δ ·α a = t · ϑ"
using assms
proof (induction t)
  case (Fun f T) thus ?case
  proof (cases f)
    case (Val n)
    hence False using Fun.prems(2) by blast
    thus ?thesis by metis
  qed auto
qed simp

lemma abs_val_in_funs_term:
  assumes "f ∈ funs_term t" "is_Val f"
  shows "Abs (α (the_Val f)) ∈ funs_term (t ·α α)"
using assms by (induct t α rule: abs_apply_term.induct) auto

end
```

3.3 Stateful Protocol Model

```
theory Stateful_Protocol_Model
  imports Stateful_Protocol_Composition_and_Typing.Stateful_Compositionality
          Transactions Term_Abstraction
begin
```

3.3.1 Locale Setup

```
locale stateful_protocol_model =
  fixes arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
```

```
    and label_witness2::"'lbl"
  assumes Ana_f_assm1: "∀f. let (K, M) = Ana_f f in (∀k ∈ subterms_set (set K).
      is_Fun k ⟶ (is_Fu (the_Fun k)) ∧ length (args k) = arity_f (the_Fu (the_Fun k)))"
    and Ana_f_assm2: "∀f. let (K, M) = Ana_f f in ∀i ∈ fv_set (set K) ∪ set M. i < arity_f f"
    and public_f_assm: "∀f. arity_f f > (0::nat) ⟶ public_f f"
    and Γ_f_assm: "∀f. arity_f f = (0::nat) ⟶ Γ_f f ≠ None"
    and label_witness_assm: "label_witness1 ≠ label_witness2"
begin

lemma Ana_f_assm1_alt:
  assumes "Ana_f f = (K,M)" "k ∈ subterms_set (set K)"
  shows "(∃x. k = Var x) ∨ (∃h T. k = Fun (Fu h) T ∧ length T = arity_f h)"
proof (cases k)
  case (Fun g T)
  let ?P = "λk. is_Fun k ⟶ is_Fu (the_Fun k) ∧ length (args k) = arity_f (the_Fu (the_Fun k))"
  let ?Q = "λK M. ∀k ∈ subterms_set (set K). ?P k"

  have "?Q (fst (Ana_f f)) (snd (Ana_f f))" using Ana_f_assm1 split_beta[of ?Q "Ana_f f"] by meson
  hence "?Q K M" using assms(1) by simp
  hence "?P k" using assms(2) by blast
  thus ?thesis using Fun by (cases g) auto
qed simp

lemma Ana_f_assm2_alt:
  assumes "Ana_f f = (K,M)" "i ∈ fv_set (set K) ∪ set M"
  shows "i < arity_f f"
using Ana_f_assm2 assms by fastforce
```

3.3.2 Definitions

```
fun arity where
  "arity (Fu f) = arity_f f"
| "arity (Set s) = arity_s s"
| "arity (Val _) = 0"
| "arity (Abs _) = 0"
| "arity Pair = 2"
| "arity (Attack _) = 0"
| "arity OccursFact = 2"
| "arity OccursSec = 0"
| "arity (PubConst _ _) = 0"

fun public where
  "public (Fu f) = public_f f"
| "public (Set s) = (arity_s s > 0)"
| "public (Val n) = False"
| "public (Abs _) = False"
| "public Pair = True"
| "public (Attack _) = False"
| "public OccursFact = True"
| "public OccursSec = False"
| "public (PubConst _ _) = True"

fun Ana where
  "Ana (Fun (Fu f) T) = (
    if arity_f f = length T ∧ arity_f f > 0
    then let (K,M) = Ana_f f in (K ·_list (!) T, map ((!) T) M)
    else ([], []))"
| "Ana _ = ([], [])"

definition Γ_v where
  "Γ_v v ≡ (
    if (∀t ∈ subterms (fst v).
          case t of (TComp f T) ⇒ arity f > 0 ∧ arity f = length T | _ ⇒ True)
```

```
    then fst v
    else TAtom Bottom)"

fun Γ where
  "Γ (Var v) = Γ_v v"
| "Γ (Fun f T) = (
    if arity f = 0
    then case f of
      (Fu g) ⇒ TAtom (case Γ_f g of Some a ⇒ Atom a | None ⇒ Bottom)
    | (Val _) ⇒ TAtom Value
    | (Abs _) ⇒ TAtom AbsValue
    | (Set _) ⇒ TAtom SetType
    | (Attack _) ⇒ TAtom AttackType
    | OccursSec ⇒ TAtom OccursSecType
    | (PubConst a _) ⇒ TAtom a
    | _ ⇒ TAtom Bottom
    else TComp f (map Γ T))"

lemma Γ_consts_simps[simp]:
  "arity_f g = 0 ⟹ Γ (Fun (Fu g) []::('fun,'atom,'sets,'lbl) prot_term)
                   = TAtom (case Γ_f g of Some a ⇒ Atom a | None ⇒ Bottom)"
  "Γ (Fun (Val n) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom Value"
  "Γ (Fun (Abs b) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom AbsValue"
  "arity_s s = 0 ⟹ Γ (Fun (Set s) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom SetType"
  "Γ (Fun (Attack x) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom AttackType"
  "Γ (Fun OccursSec []::('fun,'atom,'sets,'lbl) prot_term) = TAtom OccursSecType"
  "Γ (Fun (PubConst a t) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom a"
by simp+

lemma Γ_Fu_simps[simp]:
  "arity_f f ≠ 0 ⟹ Γ (Fun (Fu f) T) = TComp (Fu f) (map Γ T)" (is "?A1 ⟹ ?A2")
  "arity_f f = 0 ⟹ Γ_f f = Some a ⟹ Γ (Fun (Fu f) T) = TAtom (Atom a)" (is "?B1 ⟹ ?B2 ⟹ ?B3")
  "arity_f f = 0 ⟹ Γ_f f = None ⟹ Γ (Fun (Fu f) T) = TAtom Bottom" (is "?C1 ⟹ ?C2 ⟹ ?C3")
  "Γ (Fun (Fu f) T) ≠ TAtom Value" (is ?D)
  "Γ (Fun (Fu f) T) ≠ TAtom AttackType" (is ?E)
  "Γ (Fun (Fu f) T) ≠ TAtom OccursSecType" (is ?F)
proof -
  show "?A1 ⟹ ?A2" by simp
  show "?B1 ⟹ ?B2 ⟹ ?B3" by simp
  show "?C1 ⟹ ?C2 ⟹ ?C3" by simp
  show ?D by (cases "Γ_f f") simp_all
  show ?E by (cases "Γ_f f") simp_all
  show ?F by (cases "Γ_f f") simp_all
qed

lemma Γ_Set_simps[simp]:
  "arity_s s ≠ 0 ⟹ Γ (Fun (Set s) T) = TComp (Set s) (map Γ T)"
  "Γ (Fun (Set s) T) = TAtom SetType ∨ Γ (Fun (Set s) T) = TComp (Set s) (map Γ T)"
  "Γ (Fun (Set s) T) ≠ TAtom Value"
  "Γ (Fun (Set s) T) ≠ TAtom (Atom a)"
  "Γ (Fun (Set s) T) ≠ TAtom AttackType"
  "Γ (Fun (Set s) T) ≠ TAtom OccursSecType"
  "Γ (Fun (Set s) T) ≠ TAtom Bottom"
by auto
```

3.3.3 Locale Interpretations

```
lemma Ana_Fu_cases:
  assumes "Ana (Fun f T) = (K,M)"
    and "f = Fu g"
    and "Ana_f g = (K',M')"
  shows "(K,M) = (if arity_f g = length T ∧ arity_f g > 0
                 then (K' ·_list (!) T, map ((!) T) M')
```

```
                 else ([],[]))" (is ?A)
    and "(K,M) = (K' ·_list (!) T, map ((!) T) M') ∨ (K,M) = ([],[])" (is ?B)
proof -
  show ?A using assms by (cases "arity_f g = length T ∧ arity_f g > 0") auto
  thus ?B by metis
qed

lemma Ana_Fu_intro:
  assumes "arity_f f = length T" "arity_f f > 0"
    and "Ana_f f = (K',M')"
  shows "Ana (Fun (Fu f) T) = (K' ·_list (!) T, map ((!) T) M')"
using assms by simp

lemma Ana_Fu_elim:
  assumes "Ana (Fun f T) = (K,M)"
    and "f = Fu g"
    and "Ana_f g = (K',M')"
    and "(K,M) ≠ ([],[])"
  shows "arity_f g = length T" (is ?A)
    and "(K,M) = (K' ·_list (!) T, map ((!) T) M')" (is ?B)
proof -
  show ?A using assms by force
  moreover have "arity_f g > 0" using assms by force
  ultimately show ?B using assms by auto
qed

lemma Ana_nonempty_inv:
  assumes "Ana t ≠ ([],[])"
  shows "∃f T. t = Fun (Fu f) T ∧ arity_f f = length T ∧ arity_f f > 0 ∧
              (∃K M. Ana_f f = (K, M) ∧ Ana t = (K ·_list (!) T, map ((!) T) M))"
using assms
proof (induction t rule: Ana.induct)
  case (1 f T)
  hence *: "arity_f f = length T" "0 < arity_f f"
           "Ana (Fun (Fu f) T) = (case Ana_f f of (K, M) ⇒ (K ·_list (!) T, map ((!) T) M))"
    using Ana.simps(1)[of f T] unfolding Let_def by metis+

  obtain K M where **: "Ana_f f = (K, M)" by (metis surj_pair)
  hence "Ana (Fun (Fu f) T) = (K ·_list (!) T, map ((!) T) M)" using *(3) by simp
  thus ?case using ** *(1,2) by blast
qed simp_all

lemma assm1:
  assumes "Ana t = (K,M)"
  shows "fv_set (set K) ⊆ fv t"
using assms
proof (induction t rule: term.induct)
  case (Fun f T)
  have aux: "fv_set (set K ·_set (!) T) ⊆ fv_set (set T)"
    when K: "∀i ∈ fv_set (set K). i < length T"
    for K::"(('fun,'atom,'sets,'lbl) prot_fun, nat) term list"
  proof
    fix x assume "x ∈ fv_set (set K ·_set (!) T)"
    then obtain k where k: "k ∈ set K" "x ∈ fv (k · (!) T)" by auto
    have "∀i ∈ fv k. i < length T" using K k(1) by simp
    thus "x ∈ fv_set (set T)"
      by (metis (no_types, lifting) k(2) contra_subsetD fv_set_mono image_subsetI nth_mem
                                    subst_apply_fv_unfold)
  qed

  { fix g assume f: "f = Fu g" and K: "K ≠ []"
    obtain K' M' where *: "Ana_f g = (K',M')" by force
    have "(K, M) ≠ ([], [])" using K by simp
```

```
    hence "(K, M) = (K' ·_list (!) T, map ((!) T) M')" "arity_f g = length T"
      using Ana_Fu_cases(1)[OF Fun.prems f *]
      by presburger+
    hence ?case using aux[of K'] Ana_f_assm2_alt[OF *] by auto
  } thus ?case using Fun by (cases f) fastforce+
qed simp

lemma assm2:
  assumes "Ana t = (K,M)"
  and "⋀g S'. Fun g S' ⊑ t ⟹ length S' = arity g"
  and "k ∈ set K"
  and "Fun f T' ⊑ k"
  shows "length T' = arity f"
using assms
proof (induction t rule: term.induct)
  case (Fun g T)
  obtain h where 2: "g = Fu h"
    using Fun.prems(1,3) by (cases g) auto
  obtain K' M' where 1: "Ana_f h = (K',M')" by force
  have "(K,M) ≠ ([],[])" using Fun.prems(3) by auto
  hence "(K,M) = (K' ·_list (!) T, map ((!) T) M')"
        "⋀i. i ∈ fv_set (set K') ∪ set M' ⟹ i < length T"
    using Ana_Fu_cases(1)[OF Fun.prems(1) 2 1] Ana_f_assm2_alt[OF 1]
    by presburger+
  hence "K = K' ·_list (!) T" and 3: "∀i∈fv_set (set K'). i < length T" by simp_all
  then obtain k' where k': "k' ∈ set K'" "k = k' · (!) T" using Fun.prems(3) by force
  hence 4: "Fun f T' ∈ subterms (k' · (!) T)" "fv k' ⊆ fv_set (set K')"
    using Fun.prems(4) by auto
  show ?case
  proof (cases "∃i ∈ fv k'. Fun f T' ∈ subterms (T ! i)")
    case True
    hence "Fun f T' ∈ subterms_set (set T)" using k' Fun.prems(4) 3 by auto
    thus ?thesis using Fun.prems(2) by auto
  next
    case False
    then obtain S where "Fun f S ∈ subterms k'" "Fun f T' = Fun f S · (!) T"
      using k'(2) Fun.prems(4) subterm_subst_not_img_subterm by force
    thus ?thesis using Ana_f_assm1_alt[OF 1, of "Fun f S"] k'(1) by (cases f) auto
  qed
qed simp

lemma assm4:
  assumes "Ana (Fun f T) = (K, M)"
  shows "set M ⊆ set T"
using assms
proof (cases f)
  case (Fu g)
  obtain K' M' where *: "Ana_f g = (K',M')" by force
  have "M = [] ∨ (arity_f g = length T ∧ M = map ((!) T) M')"
    using Ana_Fu_cases(1)[OF assms Fu *]
    by (meson prod.inject)
  thus ?thesis using Ana_f_assm2_alt[OF *] by auto
qed auto

lemma assm5: "Ana t = (K,M) ⟹ K ≠ [] ∨ M ≠ [] ⟹ Ana (t · δ) = (K ·_list δ, M ·_list δ)"
proof (induction t rule: term.induct)
  case (Fun f T) thus ?case
  proof (cases f)
    case (Fu g)
    obtain K' M' where *: "Ana_f g = (K',M')" by force
    have **: "K = K' ·_list (!) T" "M = map ((!) T) M'"
             "arity_f g = length T" "∀i ∈ fv_set (set K') ∪ set M'. i < arity_f g" "0 < arity_f g"
      using Fun.prems(2) Ana_Fu_cases(1)[OF Fun.prems(1) Fu *] Ana_f_assm2_alt[OF *]
```

```
      by (meson prod.inject)+

    have ***: "∀i ∈ fv_set (set K'). i < length T" "∀i ∈ set M'. i < length T" using **(3,4) by auto

    have "K ·list δ = K' ·list (!) (map (λt. t · δ) T)"
         "M ·list δ = map ((!) (map (λt. t · δ) T)) M'"
      using subst_idx_map[OF ***(2), of δ]
            subst_idx_map'[OF ***(1), of δ]
            **(1,2)
      by fast+
    thus ?thesis using Fu * **(3,5) by auto
  qed auto
qed simp

sublocale intruder_model arity public Ana
apply unfold_locales
by (metis assm1, metis assm2, rule Ana.simps, metis assm4, metis assm5)

adhoc_overloading INTRUDER_SYNTH ⇌ intruder_synth
adhoc_overloading INTRUDER_DEDUCT ⇌ intruder_deduct

lemma assm6: "arity c = 0 ⟹ ∃a. ∀X. Γ (Fun c X) = TAtom a" by (cases c) auto

lemma assm7: "0 < arity f ⟹ Γ (Fun f T) = TComp f (map Γ T)" by auto

lemma assm8: "infinite {c. Γ (Fun c []::('fun,'atom,'sets,'lbl) prot_term) = TAtom a ∧ public c}"
  (is "?P a")
proof -
  let ?T = "λf. (range f)::('fun,'atom,'sets,'lbl) prot_fun set"
  let ?A = "λf. ∀x::nat ∈ UNIV. ∀y::nat ∈ UNIV. (f x = f y) = (x = y)"
  let ?B = "λf. ∀x::nat ∈ UNIV. f x ∈ ?T f"
  let ?C = "λf. ∀y::('fun,'atom,'sets,'lbl) prot_fun ∈ ?T f. ∃x ∈ UNIV. y = f x"
  let ?D = "λf b. ?T f ⊆ {c. Γ (Fun c []::('fun,'atom,'sets,'lbl) prot_term) = TAtom b ∧ public c}"

  have sub_lmm: "?P b" when "?A f" "?C f" "?C f" "?D f b" for b f
  proof -
    have "∃g::nat ⇒ ('fun,'atom,'sets,'lbl) prot_fun. bij_betw g UNIV (?T f)"
      using bij_betwI'[of UNIV f "?T f"] that(1,2,3) by blast
    hence "infinite (?T f)" by (metis nat_not_finite bij_betw_finite)
    thus ?thesis using infinite_super[OF that(4)] by blast
  qed

  show ?thesis
  proof (cases a)
    case (Atom b) thus ?thesis using sub_lmm[of "PubConst (Atom b)" a] by force
  next
    case Value thus ?thesis using sub_lmm[of "PubConst Value" a] by force
  next
    case SetType thus ?thesis using sub_lmm[of "PubConst SetType" a] by fastforce
  next
    case AttackType thus ?thesis using sub_lmm[of "PubConst AttackType" a] by fastforce
  next
    case Bottom thus ?thesis using sub_lmm[of "PubConst Bottom" a] by fastforce
  next
    case OccursSecType thus ?thesis using sub_lmm[of "PubConst OccursSecType" a] by fastforce
  next
    case AbsValue thus ?thesis using sub_lmm[of "PubConst AbsValue" a] by force
  qed
qed

lemma assm9: "TComp f T ⊑ Γ t ⟹ arity f > 0"
proof (induction t rule: term.induct)
  case (Var x)
```

```
  hence "Γ (Var x) ≠ TAtom Bottom" by force
  hence "∀t ∈ subterms (fst x). case t of
            TComp f T ⇒ arity f > 0 ∧ arity f = length T
          | _ ⇒ True"
    using Var Γ.simps(1)[of x] unfolding Γ_v_def by meson
  thus ?case using Var by (fastforce simp add: Γ_v_def)
next
  case (Fun g S)
  have "arity g ≠ 0" using Fun.prems Var_subtermeq assm6 by force
  thus ?case using Fun by (cases "TComp f T = TComp g (map Γ S)") auto
qed

lemma assm10: "wf_trm (Γ (Var x))"
unfolding wf_trm_def by (auto simp add: Γ_v_def)

lemma assm11: "arity f > 0 ⟹ public f" using public_f_assm by (cases f) auto

lemma assm12: "Γ (Var (τ, n)) = Γ (Var (τ, m))" by (simp add: Γ_v_def)

lemma assm13: "arity c = 0 ⟹ Ana (Fun c T) = ([],[])" by (cases c) simp_all

lemma assm14:
  assumes "Ana (Fun f T) = (K,M)"
  shows "Ana (Fun f T · δ) = (K ·_list δ, M ·_list δ)"
proof -
  show ?thesis
  proof (cases "(K, M) = ([],[])")
    case True
    { fix g assume f: "f = Fu g"
      obtain K' M' where "Ana_f g = (K',M')" by force
      hence ?thesis using assms f True by auto
    } thus ?thesis using True assms by (cases f) auto
  next
    case False
    then obtain g where **: "f = Fu g" using assms by (cases f) auto
    obtain K' M' where *: "Ana_f g = (K',M')" by force
    have ***: "K = K' ·_list (!) T" "M = map ((!) T) M'" "arity_f g = length T"
              "∀i ∈ fv_set (set K') ∪ set M'. i < arity_f g"
      using Ana_Fu_cases(1)[OF assms ** *] False Ana_f_assm2_alt[OF *]
      by (meson prod.inject)+
    have ****: "∀i∈fv_set (set K'). i < length T" "∀i∈set M'. i < length T" using ***(3,4) by auto
    have "K ·_list δ = K' ·_list (!) (map (λt. t · δ) T)"
         "M ·_list δ = map ((!) (map (λt. t · δ) T)) M'"
      using subst_idx_map[OF ****(2), of δ]
            subst_idx_map'[OF ****(1), of δ]
            ***(1,2)
      by auto
    thus ?thesis using assms * ** ***(3) by auto
  qed
qed

sublocale labeled_stateful_typing' arity public Ana Γ Pair label_witness1 label_witness2
  apply unfold_locales
  subgoal by (metis assm6)
  subgoal by (metis assm7)
  subgoal by (metis assm9)
  subgoal by (rule assm10)
  subgoal by (metis assm12)
  subgoal by (metis assm13)
  subgoal by (metis assm14)
  subgoal by (rule label_witness_assm)
  subgoal by (rule arity.simps(5))
  subgoal by (metis assm14)
```

```
  subgoal by (metis assm8)
  subgoal by (metis assm11)
  done
```

3.3.4 The Protocol Transition System, Defined in Terms of the Reachable Constraints

```
definition transaction_decl_subst where
  "transaction_decl_subst (ξ::('fun,'atom,'sets,'lbl) prot_subst) T ≡
    subst_domain ξ = fst ` set (transaction_decl T ()) ∧
    (∀(x,cs) ∈ set (transaction_decl T ()). ∃c ∈ cs.
      ξ x = Fun (Fu c) [] ∧
      arity (Fu c::('fun,'atom,'sets,'lbl) prot_fun) = 0) ∧
    wt_subst ξ"

definition transaction_fresh_subst where
  "transaction_fresh_subst σ T M ≡
    subst_domain σ = set (transaction_fresh T) ∧
    (∀t ∈ subst_range σ. ∃c. t = Fun c [] ∧ ¬public c ∧ arity c = 0) ∧
    (∀t ∈ subst_range σ. t ∉ subterms_set M) ∧
    (∀t ∈ subst_range σ. t ∉ subterms_set (trms_transaction T)) ∧
    wt_subst σ ∧ inj_on σ (subst_domain σ)"


definition transaction_renaming_subst where
  "transaction_renaming_subst α P X ≡
    ∃n ≥ max_var_set (⋃(vars_transaction ` set P) ∪ X). α = var_rename n"

definition (in intruder_model) constraint_model where
  "constraint_model ℐ 𝒜 ≡
    constr_sem_stateful ℐ (unlabel 𝒜) ∧
    interpretation_subst ℐ ∧
    wf_trms (subst_range ℐ)"

definition (in typed_model) welltyped_constraint_model where
  "welltyped_constraint_model ℐ 𝒜 ≡ wt_subst ℐ ∧ constraint_model ℐ 𝒜"
```

The set of symbolic constraints reachable in any symbolic run of the protocol *P*.

ξ instantiates the "declared variables" of transaction *T* with ground terms. σ instantiates the fresh variables of transaction *T* with fresh terms. α is a variable-renaming whose range consists of fresh variables.

```
inductive_set reachable_constraints::
  "('fun,'atom,'sets,'lbl) prot ⇒ ('fun,'atom,'sets,'lbl) prot_constr set"
  for P::"('fun,'atom,'sets,'lbl) prot"
where
  init[simp]:
  "[] ∈ reachable_constraints P"
| step:
  "⟦𝒜 ∈ reachable_constraints P;
    T ∈ set P;
    transaction_decl_subst ξ T;
    transaction_fresh_subst σ T (trms_lsst 𝒜);
    transaction_renaming_subst α P (vars_lsst 𝒜)
   ⟧ ⟹ 𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α) ∈ reachable_constraints P"
```

3.3.5 Minor Lemmata

```
lemma Γv_TAtom[simp]: "Γv (TAtom a, n) = TAtom a"
unfolding Γv_def by simp

lemma Γv_TAtom':
  assumes "a ≠ Bottom"
  shows "Γv (τ, n) = TAtom a ⟷ τ = TAtom a"
proof
  assume "Γv (τ, n) = TAtom a"
```

```
  thus "τ = TAtom a" by (metis (no_types, lifting) assms Γv_def fst_conv term.inject(1))
qed simp

lemma Γv_TAtom_inv:
  "Γv x = TAtom (Atom a) ⟹ ∃m. x = (TAtom (Atom a), m)"
  "Γv x = TAtom Value ⟹ ∃m. x = (TAtom Value, m)"
  "Γv x = TAtom SetType ⟹ ∃m. x = (TAtom SetType, m)"
  "Γv x = TAtom AttackType ⟹ ∃m. x = (TAtom AttackType, m)"
  "Γv x = TAtom OccursSecType ⟹ ∃m. x = (TAtom OccursSecType, m)"
by (metis Γv_TAtom' surj_pair prot_atom.distinct(7),
    metis Γv_TAtom' surj_pair prot_atom.distinct(18),
    metis Γv_TAtom' surj_pair prot_atom.distinct(26),
    metis Γv_TAtom' surj_pair prot_atom.distinct(32),
    metis Γv_TAtom' surj_pair prot_atom.distinct(38))

lemma Γv_TAtom'':
  "(fst x = TAtom (Atom a)) = (Γv x = TAtom (Atom a))" (is "?A = ?A'")
  "(fst x = TAtom Value) = (Γv x = TAtom Value)" (is "?B = ?B'")
  "(fst x = TAtom SetType) = (Γv x = TAtom SetType)" (is "?C = ?C'")
  "(fst x = TAtom AttackType) = (Γv x = TAtom AttackType)" (is "?D = ?D'")
  "(fst x = TAtom OccursSecType) = (Γv x = TAtom OccursSecType)" (is "?E = ?E'")
proof -
  have 1: "?A ⟹ ?A'" "?B ⟹ ?B'" "?C ⟹ ?C'" "?D ⟹ ?D'" "?E ⟹ ?E'"
    by (metis Γv_TAtom prod.collapse)+

  have 2: "?A' ⟹ ?A" "?B' ⟹ ?B" "?C' ⟹ ?C" "?D' ⟹ ?D" "?E' ⟹ ?E"
    using Γv_TAtom Γv_TAtom_inv(1) apply fastforce
    using Γv_TAtom Γv_TAtom_inv(2) apply fastforce
    using Γv_TAtom Γv_TAtom_inv(3) apply fastforce
    using Γv_TAtom Γv_TAtom_inv(4) apply fastforce
    using Γv_TAtom Γv_TAtom_inv(5) by fastforce

  show "?A = ?A'" "?B = ?B'" "?C = ?C'" "?D = ?D'" "?E = ?E'"
    using 1 2 by metis+
qed

lemma Γv_Var_image:
  "Γv ` X = Γ ` Var ` X"
by force

lemma Γ_Fu_const:
  assumes "arityf g = 0"
  shows "∃a. Γ (Fun (Fu g) T) = TAtom (Atom a)"
proof -
  have "Γf g ≠ None" using assms Γf_assm by blast
  thus ?thesis using assms by force
qed

lemma Fun_Value_type_inv:
  fixes T::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes "Γ (Fun f T) = TAtom Value"
  shows "(∃n. f = Val n) ∨ (∃bs. f = Abs bs) ∨ (∃n. f = PubConst Value n)"
proof -
  have *: "arity f = 0" by (metis const_type_inv assms)
  show ?thesis  using assms
  proof (cases f)
    case (Fu g)
    hence "arityf g = 0" using * by simp
    hence False using Fu Γ_Fu_const[of g T] assms by auto
    thus ?thesis by metis
  next
    case (Set s)
    hence "aritys s = 0" using * by simp
```

```
    hence False using Set assms by auto
    thus ?thesis by metis
  qed simp_all
qed

lemma Ana_f_keys_not_val_terms:
  assumes "Ana_f f = (K, T)"
    and "k ∈ set K"
    and "g ∈ funs_term k"
  shows "¬is_Val g"
    and "¬is_PubConstValue g"
    and "¬is_Abs g"
proof -
  { assume "is_Val g"
    then obtain n S where *: "Fun (Val n) S ∈ subterms_set (set K)"
      using assms(2) funs_term_Fun_subterm[OF assms(3)]
      by (cases g) auto
    hence False using Ana_f_assm1_alt[OF assms(1) *] by simp
  } moreover {
    assume "is_PubConstValue g"
    then obtain n S where *: "Fun (PubConst Value n) S ∈ subterms_set (set K)"
      using assms(2) funs_term_Fun_subterm[OF assms(3)]
      unfolding is_PubConstValue_def by (cases g) auto
    hence False using Ana_f_assm1_alt[OF assms(1) *] by simp
  } moreover {
    assume "is_Abs g"
    then obtain a S where *: "Fun (Abs a) S ∈ subterms_set (set K)"
      using assms(2) funs_term_Fun_subterm[OF assms(3)]
      by (cases g) auto
    hence False using Ana_f_assm1_alt[OF assms(1) *] by simp
  } ultimately show "¬is_Val g" "¬is_PubConstValue g" "¬is_Abs g" by metis+
qed

lemma Ana_f_keys_not_pairs:
  assumes "Ana_f f = (K, T)"
    and "k ∈ set K"
    and "g ∈ funs_term k"
  shows "g ≠ Pair"
proof
  assume "g = Pair"
  then obtain S where *: "Fun Pair S ∈ subterms_set (set K)"
    using assms(2) funs_term_Fun_subterm[OF assms(3)]
    by (cases g) auto
  show False using Ana_f_assm1_alt[OF assms(1) *] by simp
qed

lemma Ana_Fu_keys_funs_term_subset:
  fixes K::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
  shows "⋃(funs_term ` set K) ⊆ ⋃(funs_term ` set K') ∪ funs_term (Fun (Fu f) S)"
proof -
  { fix k assume k: "k ∈ set K"
    then obtain k' where k':
        "k' ∈ set K'" "k = k' · (!) S" "arity_f f = length S"
        "subterms k' ⊆ subterms_set (set K')"
      using assms Ana_Fu_elim[OF assms(1) _ assms(2)] by fastforce

    have 1: "funs_term k' ⊆ ⋃(funs_term ` set K')" using k'(1) by auto

    have "i < length S" when "i ∈ fv k'" for i
      using that Ana_f_assm2_alt[OF assms(2), of i] k'(1,3)
      by auto
```

```
    hence 2: "funs_term (S ! i) ⊆ funs_term (Fun (Fu f) S)" when "i ∈ fv k'" for i
      using that by force

    have "funs_term k ⊆ ⋃(funs_term ` set K') ∪ funs_term (Fun (Fu f) S)"
      using funs_term_subst[of k' "(!) S"] k'(2) 1 2 by fast
  } thus ?thesis by blast
qed

lemma Ana_Fu_keys_not_pubval_terms:
  fixes k::"('fun,'atom,'sets,'lbl) prot_term"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
    and "k ∈ set K"
    and "∀g ∈ funs_term (Fun (Fu f) S). ¬is_PubConstValue g"
  shows "∀g ∈ funs_term k. ¬is_PubConstValue g"
using assms(3,4) Ana_f_keys_not_val_terms(1,2)[OF assms(2)]
      Ana_Fu_keys_funs_term_subset[OF assms(1,2)]
by blast

lemma Ana_Fu_keys_not_abs_terms:
  fixes k::"('fun,'atom,'sets,'lbl) prot_term"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
    and "k ∈ set K"
    and "∀g ∈ funs_term (Fun (Fu f) S). ¬is_Abs g"
  shows "∀g ∈ funs_term k. ¬is_Abs g"
using assms(3,4) Ana_f_keys_not_val_terms(3)[OF assms(2)]
      Ana_Fu_keys_funs_term_subset[OF assms(1,2)]
by blast

lemma Ana_Fu_keys_not_pairs:
  fixes k::"('fun,'atom,'sets,'lbl) prot_term"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
    and "k ∈ set K"
    and "∀g ∈ funs_term (Fun (Fu f) S). g ≠ Pair"
  shows "∀g ∈ funs_term k. g ≠ Pair"
using assms(3,4) Ana_f_keys_not_pairs[OF assms(2)]
      Ana_Fu_keys_funs_term_subset[OF assms(1,2)]
by blast

lemma Ana_Fu_keys_length_eq:
  assumes "length T = length S"
  shows "length (fst (Ana (Fun (Fu f) T))) = length (fst (Ana (Fun (Fu f) S)))"
proof (cases "arity_f f = length T ∧ arity_f f > 0")
  case True thus ?thesis using assms by (cases "Ana_f f") auto
next
  case False thus ?thesis using assms by force
qed

lemma Ana_key_PubConstValue_subterm_in_term:
  fixes k::"('fun,'atom,'sets,'lbl) prot_term"
  assumes KR: "Ana t = (K, R)"
    and k: "k ∈ set K"
    and n: "Fun (PubConst Value n) [] ⊑ k"
  shows "Fun (PubConst Value n) [] ⊑ t"
proof (cases t)
  case (Var x) thus ?thesis using KR k n by force
next
  case (Fun f ts)
  note t = this
  then obtain g where f: "f = Fu g" using KR k by (cases f) auto
  obtain K' R' where KR': "Ana_f g = (K', R')" by fastforce
```

```
  have K: "K = K' ·list (!) ts"
    using k Ana_Fu_elim(2)[OF KR[unfolded t] f KR'] by force

  obtain k' where k': "k' ∈ set K'" "k = k' · (!) ts" using k K by auto

  have 0: "¬(Fun (PubConst Value n) [] ⊑ k')"
  proof
    assume *: "Fun (PubConst Value n) [] ⊑ k'"
    have **: "PubConst Value n ∈ funs_term k'"
      using funs_term_Fun_subterm'[OF *] by (cases k') auto
    show False
      using Ana_f_keys_not_val_terms(2)[OF KR' k'(1) **]
      unfolding is_PubConstValue_def by force
  qed
  hence "∃i ∈ fv k'. Fun (PubConst Value n) [] ⊑ ts ! i"
    by (metis n const_subterm_subst_var_obtain k'(2))
  then obtain i where i: "i ∈ fv k'" "Fun (PubConst Value n) [] ⊑ ts ! i" by blast

  have "i < length ts"
    using i(1) KR' k'(1) Ana_f_assm2_alt[OF KR', of i]
          Ana_Fu_elim(1)[OF KR[unfolded t] f KR'] k
    by fastforce
  thus ?thesis using i(2) unfolding t by force
qed

lemma deduct_occurs_in_ik:
  fixes t::"('fun,'atom,'sets,'lbl) prot_term"
  assumes t: "M ⊢ occurs t"
    and M: "∀s ∈ subterms_set M. OccursFact ∉ ⋃(funs_term ` set (snd (Ana s)))"
           "∀s ∈ subterms_set M. OccursSec ∉ ⋃(funs_term ` set (snd (Ana s)))"
           "Fun OccursSec [] ∉ M"
  shows "occurs t ∈ M"
using private_fun_deduct_in_ik''[of M OccursFact "[Fun OccursSec [], t]" OccursSec] t M
by fastforce

lemma deduct_val_const_swap:
  fixes ϑ σ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "M ·set ϑ ⊢ t · ϑ"
    and "∀x ∈ fv_set M ∪ fv t. (∃n. ϑ x = Fun (Val n) []) ∨ (∃n. ϑ x = Fun (PubConst Value n) [])"
    and "∀x ∈ fv_set M ∪ fv t. (∃n. σ x = Fun (Val n) [])"
    and "∀x ∈ fv_set M ∪ fv t. (∃n. ϑ x = Fun (PubConst Value n) []) ⟶ σ x ∈ M ∪ N"
    and "∀x ∈ fv_set M ∪ fv t. (∃n. ϑ x = Fun (Val n) []) ⟶ ϑ x = σ x"
    and "∀x ∈ fv_set M ∪ fv t. ∀y ∈ fv_set M ∪ fv t. ϑ x = ϑ y ⟷ σ x = σ y"
    and "∀n. ¬(Fun (PubConst Value n) [] ⊑set insert t M)"
  shows "(M ·set σ) ∪ N ⊢ t · σ"
proof -
  obtain n where n: "intruder_deduct_num (M ·set ϑ) n (t · ϑ)"
    using assms(1) deduct_num_if_deduct by blast
  hence "∃m ≤ n. intruder_deduct_num ((M ·set σ) ∪ N) m (t · σ)" using assms(2-)
  proof (induction n arbitrary: t rule: nat_less_induct)
    case (1 n)
    note prems = "1.prems"
    note IH = "1.IH"

    show ?case
    proof (cases "t · ϑ ∈ M ·set ϑ")
      case True
      note 2 = this
      have 3: "∀x ∈ fv_set M ∪ fv t. ∃c. ϑ x = Fun c []"
              "∀x ∈ fv_set M ∪ fv t. ∃c. σ x = Fun c []"
        using prems(2,3) by (blast, blast)
      have "t · σ ∈ M ·set σ"
```

```
        using subst_const_swap_eq_mem[OF 2 _ 3 prems(6)] prems(2,5,7) by metis
      thus ?thesis using intruder_deduct_num.AxiomN by auto
  next
    case False
    then obtain n' where n: "n = Suc n'" using prems(1) deduct_zero_in_ik by (cases n) fast+

    have M_subterms_eq:
        "subterms_set (M ·_set ϑ) = subterms_set M ·_set ϑ"
        "subterms_set (M ·_set σ) = subterms_set M ·_set σ"
      subgoal using prems(2) subterms_subst''[of M ϑ] by blast
      subgoal using prems(3) subterms_subst''[of M σ] by blast
      done

    from deduct_inv[OF prems(1)] show ?thesis
    proof (elim disjE)
      assume "t · ϑ ∈ M ·_set ϑ" thus ?thesis using False by argo
    next
      assume "∃f ts. t · ϑ = Fun f ts ∧ public f ∧ length ts = arity f ∧
                  (∀t ∈ set ts. ∃l < n. intruder_deduct_num (M ·_set ϑ) l t)"
      then obtain f ts where t:
          "t · ϑ = Fun f ts" "public f" "length ts = arity f"
          "∀t ∈ set ts. ∃l < n. intruder_deduct_num (M ·_set ϑ) l t"
        by blast

      show ?thesis
      proof (cases t)
        case (Var x)
        hence ts: "ts = []" and f: "∃c. f = PubConst Value c"
          using t(1,2) prems(2) by (force, auto)
        have "σ x ∈ M ∪ N" using prems(4) Var f ts t(1) by auto
        moreover have "fv (σ x) = {}" using prems(3) Var by auto
        hence "σ x ∈ M ·_set σ" when "σ x ∈ M" using that subst_ground_ident[of "σ x" σ] by force
        ultimately have "σ x ∈ (M ·_set σ) ∪ N" by fast
        thus ?thesis using intruder_deduct_num.AxiomN Var by force
      next
        case (Fun g ss)
        hence f: "f = g" and ts: "ts = ss ·_list ϑ" using t(1) by auto

        have ss: "∃l < n. intruder_deduct_num (M ·_set ϑ) l (s · ϑ)" when s: "s ∈ set ss" for s
          using t(4) ts s by auto

        have IH': "∃l < n. intruder_deduct_num ((M ·_set σ) ∪ N) l (s · σ)"
          when s: "s ∈ set ss" for s
        proof -
          obtain l where l: "l < n" "intruder_deduct_num (M ·_set ϑ) l (s · ϑ)"
            using ss s by blast

          have *: "fv s ⊆ fv t" "subterms_set (insert s M) ⊆ subterms_set (insert t M)"
            using s unfolding Fun f ts by auto

          have "∃l' ≤ l. intruder_deduct_num ((M ·_set σ) ∪ N) l' (s · σ)"
          proof -
            have "∀x ∈ fv_set M ∪ fv s.
                    (∃n. ϑ x = Fun (Val n) []) ∨ (∃n. ϑ x = Fun (PubConst Value n) [])"
                 "∀x ∈ fv_set M ∪ fv s. ∃n. σ x = Fun (Val n) []"
                 "∀x ∈ fv_set M ∪ fv s. (∃n. ϑ x = Fun (PubConst Value n) []) ⟶ σ x ∈ M ∪ N"
                 "∀x ∈ fv_set M ∪ fv s. (∃n. ϑ x = Fun (Val n) []) ⟶ ϑ x = σ x"
                 "∀x ∈ fv_set M ∪ fv s. ∀y ∈ fv_set M ∪ fv s. ϑ x = ϑ y ⟷ σ x = σ y"
                 "∀n. Fun (PubConst Value n) [] ∉ subterms_set (insert s M)"
              subgoal using prems(2) *(1) by blast
              subgoal using prems(3) *(1) by blast
              subgoal using prems(4) *(1) by blast
              subgoal using prems(5) *(1) by blast
```

```
          subgoal using prems(6) *(1) by blast
          subgoal using prems(7) *(2) by blast
          done
        thus ?thesis using IH l by presburger
      qed
      then obtain l' where l': "l' ≤ l" "intruder_deduct_num ((M ·set σ) ∪ N) l' (s · σ)"
        by blast

      have "l' < n" using l'(1) l(1) by linarith
      thus ?thesis using l'(2) by blast
    qed

    have g: "length (ss ·list σ) = arity g" "public g"
      using t(2,3) unfolding f ts by auto

    let ?P = "λs l. l < n ∧ intruder_deduct_num ((M ·set σ) ∪ N) l s"
    define steps where "steps ≡ λs. SOME l. ?P s l"

    have 2: "steps (s · σ) < n" "intruder_deduct_num ((M ·set σ) ∪ N) (steps (s · σ)) (s · σ)"
      when s: "s ∈ set ss" for s
      using someI_ex[OF IH'[OF s]] unfolding steps_def by (blast, blast)

    have 3: "Suc (Max (insert 0 (steps ` set (ss ·list σ)))) ≤ n"
    proof (cases "ss = []")
      case True show ?thesis unfolding True n by simp
    next
      case False thus ?thesis
        using 2 Max_nat_finite_lt[of "set (ss ·list σ)" steps n] by (simp add: Suc_leI)
    qed

    show ?thesis
      using intruder_deduct_num.ComposeN[OF g, of "(M ·set σ) ∪ N" steps] 2(2) 3
      unfolding Fun by auto
  qed
next
  assume "∃s ∈ subtermsset (M ·set ϑ).
            (∃l < n. intruder_deduct_num (M ·set ϑ) l s) ∧
            (∀k ∈ set (fst (Ana s)). ∃l < n. intruder_deduct_num (M ·set ϑ) l k) ∧
            t · ϑ ∈ set (snd (Ana s))"
  then obtain s l
      where s:
        "s ∈ subtermsset M ·set ϑ"
        "∀k ∈ set (fst (Ana s)). ∃l < n. intruder_deduct_num (M ·set ϑ) l k"
        "t · ϑ ∈ set (snd (Ana s))"
      and l: "l < n" "intruder_deduct_num (M ·set ϑ) l s"
    by (metis (no_types, lifting) M_subterms_eq(1))

  obtain u where u: "u ⊑set M" "s = u · ϑ" using s(1) by blast

  have u_fv: "fv u ⊆ fvset M" by (metis fv_subset_subterms u(1))

  have "∄x. u = Var x"
  proof
    assume "∃x. u = Var x"
    then obtain x where x: "u = Var x" by blast
    then obtain c where c: "s = Fun c []" using u prems(2) u_fv by auto
    thus False using s(3) Ana_subterm by (cases "Ana s") force
  qed
  then obtain f ts where u': "u = Fun f ts" by (cases u) auto

  obtain K R where KR: "Ana u = (K,R)" by (metis surj_pair)

  have KR': "Ana s = (K ·list ϑ, R ·list ϑ)"
```

```
    using KR Ana_subst'[OF KR[unfolded u'], of ϑ] unfolding u(2) u' by blast
  hence s':
      "∀k ∈ set K. ∃l < n. intruder_deduct_num (M ·set ϑ) l (k · ϑ)"
      "t · ϑ ∈ set (R ·list ϑ)"
    using s(2,3) by auto

  have IH1: "∃l < n. intruder_deduct_num ((M ·set σ) ∪ N) l (u · σ)"
  proof -
    have "subterms u ⊆ subtermsset M" using u(1) subterms_subset by auto
    hence "subtermsset (insert u M) = subtermsset M" by blast
    hence *: "subtermsset (insert u M) ⊆ subtermsset (insert t M)" by auto

    have "∃l' ≤ l. intruder_deduct_num ((M ·set σ) ∪ N) l' (u · σ)"
    proof -
      have "∀x ∈ fvset M ∪ fv u.
              (∃n. ϑ x = Fun (Val n) []) ∨ (∃n. ϑ x = Fun (PubConst Value n) [])"
           "∀x ∈ fvset M ∪ fv u. ∃n. σ x = Fun (Val n) []"
           "∀x ∈ fvset M ∪ fv u. (∃n. ϑ x = Fun (PubConst Value n) []) ⟶ σ x ∈ M ∪ N"
           "∀x ∈ fvset M ∪ fv u. (∃n. ϑ x = Fun (Val n) []) ⟶ ϑ x = σ x"
           "∀x ∈ fvset M ∪ fv u. ∀y ∈ fvset M ∪ fv u. ϑ x = ϑ y ⟷ σ x = σ y"
           "∀n. Fun (PubConst Value n) [] ∉ subtermsset (insert u M)"
        subgoal using prems(2) u_fv by blast
        subgoal using prems(3) u_fv by blast
        subgoal using prems(4) u_fv by blast
        subgoal using prems(5) u_fv by blast
        subgoal using prems(6) u_fv by blast
        subgoal using prems(7) * by blast
        done
      thus ?thesis using IH l unfolding u(2) by presburger
    qed
    then obtain l' where l': "l' ≤ l" "intruder_deduct_num ((M ·set σ) ∪ N) l' (u · σ)"
      by blast

    have "l' < n" using l'(1) l(1) by linarith
    thus ?thesis using l'(2) by blast
  qed

  have IH2: "∃l < n. intruder_deduct_num ((M ·set σ) ∪ N) l (k · σ)" when k: "k ∈ set K" for k
    using k IH prems(2-) Anaf_keys_not_val_terms s'(1) KR u(1)
  proof -
    have *: "fv k ⊆ fvset M" using k KR Ana_keys_fv u(1) fv_subset_subterms by blast

    have **: "Fun (PubConst Value n) [] ⊑set M" when "Fun (PubConst Value n) [] ⊑ k" for n
      using in_subterms_subset_Union[OF u(1)]
            Ana_key_PubConstValue_subterm_in_term[OF KR k that]
      by fast

    obtain lk where lk: "lk < n" "intruder_deduct_num (M ·set ϑ) lk (k · ϑ)"
      using s'(1) k by fast

    have "∃l' ≤ lk. intruder_deduct_num ((M ·set σ) ∪ N) l' (k · σ)"
    proof -
      have "∀x ∈ fvset M ∪ fv k.
              (∃n. ϑ x = Fun (Val n) []) ∨ (∃n. ϑ x = Fun (PubConst Value n) [])"
           "∀x ∈ fvset M ∪ fv k. ∃n. σ x = Fun (Val n) []"
           "∀x ∈ fvset M ∪ fv k. (∃n. ϑ x = Fun (PubConst Value n) []) ⟶ σ x ∈ M ∪ N"
           "∀x ∈ fvset M ∪ fv k. (∃n. ϑ x = Fun (Val n) []) ⟶ ϑ x = σ x"
           "∀x ∈ fvset M ∪ fv k. ∀y ∈ fvset M ∪ fv k. ϑ x = ϑ y ⟷ σ x = σ y"
           "∀n. Fun (PubConst Value n) [] ∉ subtermsset (insert k M)"
        subgoal using prems(2) * by blast
        subgoal using prems(3) * by blast
        subgoal using prems(4) * by blast
        subgoal using prems(5) * by blast
```

```
      subgoal using prems(6) * by blast
      subgoal using prems(7) ** by blast
      done
    thus ?thesis using IH lk by presburger
  qed
  then obtain lk' where lk': "lk' ≤ lk" "intruder_deduct_num ((M ·set σ) ∪ N) lk' (k · σ)"
    by blast

  have "lk' < n" using lk'(1) lk(1) by linarith
  thus ?thesis using lk'(2) by blast
qed

have KR'': "Ana (u · σ) = (K ·list σ, R ·list σ)"
  using Ana_subst' KR unfolding u' by blast

obtain r where r: "r ∈ set R" "t · ϑ = r · ϑ"
  using s'(2) by fastforce

have r': "t · σ ∈ set (R ·list σ)"
proof -
  have r_subterm_u: "r ⊑ u" using r(1) KR Ana_subterm by blast

  have r_fv: "fv r ⊆ fvset M"
    by (meson r_subterm_u u(1) fv_subset_subterms in_mono in_subterms_subset_Union)

  have t_subterms_M: "subterms t ⊆ subtermsset (insert t M)"
    by blast

  have r_subterm_M: "subterms r ⊆ subtermsset (insert t M)"
    using subterms_subset[OF r_subterm_u] in_subterms_subset_Union[OF u(1)]
    by (auto intro: subtermsset_mono)

  have *: "∀x ∈ fv t ∪ fv r. ϑ x = σ x ∨ ¬(ϑ x ⊑ t) ∧ ¬(ϑ x ⊑ r)"
  proof
    fix x assume "x ∈ fv t ∪ fv r"
    hence "x ∈ fvset M ∪ fv t" using r_fv by blast
    thus "ϑ x = σ x ∨ ¬(ϑ x ⊑ t) ∧ ¬(ϑ x ⊑ r)"
      using prems(2,5,7) r_subterm_M t_subterms_M
      by (metis (no_types, opaque_lifting) in_mono)
  qed

  have **: "∀x ∈ fv t ∪ fv r. ∃c. ϑ x = Fun c []"
           "∀x ∈ fv t ∪ fv r. ∃c. σ x = Fun c []"
           "∀x ∈ fv t ∪ fv r. ∀y ∈ fv t ∪ fv r. ϑ x = ϑ y ⟷ σ x = σ y"
    subgoal using prems(2) r_fv by blast
    subgoal using prems(3) r_fv by blast
    subgoal using prems(6) r_fv by blast
    done

  have "t · σ = r · σ" by (rule subst_const_swap_eq'[OF r(2) * **])
  thus ?thesis using r(1) by simp
qed

obtain l1 where l1: "l1 < n" "intruder_deduct_num ((M ·set σ) ∪ N) l1 (u · σ)"
  using IH1 by blast

let ?P = "λs l. l < n ∧ intruder_deduct_num ((M ·set σ) ∪ N) l s"
define steps where "steps ≡ λs. SOME l. ?P s l"

have 2: "steps (k · σ) < n" "intruder_deduct_num ((M ·set σ) ∪ N) (steps (k · σ)) (k · σ)"
  when k: "k ∈ set K" for k
  using someI_ex[OF IH2[OF k]] unfolding steps_def by (blast, blast)
```

```
        have 3: "Suc (Max (insert l1 (steps ` set (K ·list σ)))) ≤ n"
        proof (cases "K = []")
          case True show ?thesis using l1(1) unfolding True n by simp
        next
          case False thus ?thesis
            using l1(1) 2 Max_nat_finite_lt[of "set (K ·list σ)" steps n] by (simp add: Suc_leI)
        qed

        have IH2': "intruder_deduct_num ((M ·set σ) ∪ N) (steps k) k"
          when k: "k ∈ set (K ·list σ)" for k
          using IH2 k 2 by auto

        show ?thesis
          using l1(1) intruder_deduct_num.DecomposeN[OF l1(2) KR'' IH2' r'] 3 by fast
      qed
    qed
  qed
  thus ?thesis using deduct_if_deduct_num by blast
qed

lemma constraint_model_Nil:
  assumes I: "interpretationsubst I" "wftrms (subst_range I)"
  shows "constraint_model I []"
using I unfolding constraint_model_def by simp

lemma welltyped_constraint_model_Nil:
  assumes I: "wtsubst I" "interpretationsubst I" "wftrms (subst_range I)"
  shows "welltyped_constraint_model I []"
using I(1) constraint_model_Nil[OF I(2,3)] unfolding welltyped_constraint_model_def by simp

lemma constraint_model_prefix:
  assumes "constraint_model I (A@B)"
  shows "constraint_model I A"
by (metis assms strand_sem_append_stateful unlabel_append constraint_model_def)

lemma welltyped_constraint_model_prefix:
  assumes "welltyped_constraint_model I (A@B)"
  shows "welltyped_constraint_model I A"
by (metis assms constraint_model_prefix welltyped_constraint_model_def)

lemma welltyped_constraint_model_deduct_append:
  assumes "welltyped_constraint_model I A"
    and "iklsst A ·set I ⊢ s · I"
  shows "welltyped_constraint_model I (A@[(l,send⟨[s]⟩)])"
using assms strand_sem_append_stateful[of "{}" "{}" "unlabel A" _ I]
unfolding welltyped_constraint_model_def constraint_model_def by simp

lemma welltyped_constraint_model_deduct_split:
  assumes "welltyped_constraint_model I (A@[(l,send⟨[s]⟩)])"
  shows "welltyped_constraint_model I A"
    and "iklsst A ·set I ⊢ s · I"
using assms strand_sem_append_stateful[of "{}" "{}" "unlabel A" _ I]
unfolding welltyped_constraint_model_def constraint_model_def by simp_all

lemma welltyped_constraint_model_deduct_iff:
  "welltyped_constraint_model I (A@[(l,send⟨[s]⟩)]) ⟷
    welltyped_constraint_model I A ∧ iklsst A ·set I ⊢ s · I"
by (metis welltyped_constraint_model_deduct_append welltyped_constraint_model_deduct_split)

lemma welltyped_constraint_model_attack_if_receive_attack:
  assumes I: "welltyped_constraint_model ℐ 𝒜"
    and rcv_attack: "receive⟨ts⟩ ∈ set (unlabel 𝒜)" "attack⟨n⟩ ∈ set ts"
  shows "welltyped_constraint_model ℐ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
```

```
proof -
  have "ik_lsst 𝒜 ·_set ℐ ⊢ attack⟨n⟩"
    using rcv_attack in_ik_sst_iff[of "attack⟨n⟩" "unlabel 𝒜"]
          ideduct_subst[OF intruder_deduct.Axiom[of "attack⟨n⟩" "ik_lsst 𝒜"], of ℐ]
    by auto
  thus ?thesis
    using I strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "[send⟨[attack⟨n⟩]⟩]" ℐ]
    unfolding welltyped_constraint_model_def constraint_model_def by auto
qed

lemma constraint_model_Val_is_Value_term:
  assumes "welltyped_constraint_model I A"
    and "t · I = Fun (Val n) []"
  shows "t = Fun (Val n) [] ∨ (∃m. t = Var (TAtom Value, m))"
proof -
  have "wt_subst I" using assms(1) unfolding welltyped_constraint_model_def by simp
  moreover have "Γ (Fun (Val n) []) = TAtom Value" by auto
  ultimately have *: "Γ t = TAtom Value" by (metis (no_types) assms(2) wt_subst_trm'')

  show ?thesis
  proof (cases t)
    case (Var x)
    obtain τ m where x: "x = (τ, m)" by (metis surj_pair)
    have "Γ_v x = TAtom Value" using * Var by auto
    hence "τ = TAtom Value" using x Γ_v_TAtom'[of Value τ m] by simp
    thus ?thesis using x Var by metis
  next
    case (Fun f T) thus ?thesis using assms(2) by auto
  qed
qed

lemma wellformed_transaction_sem_receives:
  fixes T::"('fun,'atom,'sets,'lbl) prot_transaction"
  assumes T_valid: "wellformed_transaction T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
    and s: "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T ·_lsst ϑ))"
  shows "∀t ∈ set ts. IK ⊢ t · ℐ"
proof -
  let ?R = "unlabel (dual_lsst (transaction_receive T ·_lsst ϑ))"
  let ?S = "λA. unlabel (dual_lsst (A ·_lsst ϑ))"
  let ?S' = "?S (transaction_receive T)"

  obtain l B s where B:
      "(l,send⟨ts⟩) = dual_lsstp ((l,s) ·_lsstp ϑ)"
      "prefix ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ]) (transaction_receive T ·_lsst ϑ)"
    using s dual_lsst_unlabel_steps_iff(2)[of ts "transaction_receive T ·_lsst ϑ"]
          dual_lsst_in_set_prefix_obtain_subst[of "send⟨ts⟩" "transaction_receive T" ϑ]
    by blast

  have 1: "unlabel (dual_lsst ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ])) = unlabel (dual_lsst (B ·_lsst ϑ))@[send⟨ts⟩]"
    using B(1) unlabel_append dual_lsstp_subst dual_lsst_subst singleton_lst_proj(4)
          dual_lsst_subst_snoc subst_lsst_append subst_lsst_singleton
    by (metis (no_types, lifting) subst_apply_labeled_stateful_strand_step.simps)

  have "strand_sem_stateful IK DB ?S' ℐ"
    using ℐ strand_sem_append_stateful[of IK DB _ _ ℐ] transaction_dual_subst_unfold[of T ϑ]
    by fastforce
  hence "strand_sem_stateful IK DB (unlabel (dual_lsst (B ·_lsst ϑ))@[send⟨ts⟩]) ℐ"
    using B 1 unfolding prefix_def unlabel_def
    by (metis dual_lsst_def map_append strand_sem_append_stateful)
  hence t_deduct: "∀t ∈ set ts. IK ∪ (ik_lsst (dual_lsst (B ·_lsst ϑ)) ·_set ℐ) ⊢ t · ℐ"
    using strand_sem_append_stateful[of IK DB "unlabel (dual_lsst (B ·_lsst ϑ))" "[send⟨ts⟩]" ℐ]
    by simp
```

```
  have "∀s ∈ set (unlabel (transaction_receive T)). ∃t. s = receive⟨t⟩"
    using T_valid wellformed_transaction_unlabel_cases(1)[OF T_valid] by auto
  moreover { fix A::"('fun,'atom,'sets,'lbl) prot_strand" and ϑ
    assume "∀s ∈ set (unlabel A). ∃t. s = receive⟨t⟩"
    hence "∀s ∈ set (unlabel (A ·$_{lsst}$ ϑ)). ∃t. s = receive⟨t⟩"
    proof (induction A)
      case (Cons a A) thus ?case using subst_lsst_cons[of a A ϑ] by (cases a) auto
    qed simp
    hence "∀s ∈ set (unlabel (A ·$_{lsst}$ ϑ)). ∃t. s = receive⟨t⟩"
      by (simp add: list.pred_set is_Receive_def)
    hence "∀s ∈ set (unlabel (dual$_{lsst}$ (A ·$_{lsst}$ ϑ))). ∃t. s = send⟨t⟩"
      by (metis dual$_{lsst}$_memberD dual$_{lsstp}$_inv(2) unlabel_in unlabel_mem_has_label)
  }
  ultimately have "∀s ∈ set ?R. ∃ts. s = send⟨ts⟩" by simp
  hence "ik$_{sst}$ ?R = {}" unfolding unlabel_def ik$_{sst}$_def by fast
  hence "ik$_{lsst}$ (dual$_{lsst}$ (B ·$_{lsst}$ ϑ)) = {}"
    using B(2) 1 ik$_{sst}$_append dual$_{lsst}$_append
    by (metis (no_types, lifting) Un_empty map_append prefix_def unlabel_def)
  thus ?thesis using t_deduct by simp
qed

lemma wellformed_transaction_sem_pos_checks:
  assumes T_valid: "wellformed_transaction T"
    and 𝓘: "strand_sem_stateful IK DB (unlabel (dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ϑ))) 𝓘"
  shows "⟨ac: t ∈ u⟩ ∈ set (unlabel (transaction_checks T ·$_{lsst}$ ϑ)) ⟹ (t · 𝓘, u · 𝓘) ∈ DB"
    and "⟨ac: t ≐ u⟩ ∈ set (unlabel (transaction_checks T ·$_{lsst}$ ϑ)) ⟹ t · 𝓘 = u · 𝓘"
proof -
  let ?s = "⟨ac: t ∈ u⟩"
  let ?s' = "⟨ac: t ≐ u⟩"
  let ?C = "set (unlabel (transaction_checks T ·$_{lsst}$ ϑ))"
  let ?R = "transaction_receive T@transaction_checks T"
  let ?R' = "unlabel (dual$_{lsst}$ (?R ·$_{lsst}$ ϑ))"
  let ?S = "λA. unlabel (dual$_{lsst}$ (A ·$_{lsst}$ ϑ))"
  let ?S' = "?S (transaction_receive T)@?S (transaction_checks T)"
  let ?P = "λa. is_Receive a ∨ is_Check_or_Assignment a"
  let ?Q = "λa. is_Send a ∨ is_Check_or_Assignment a"
  let ?dbupd = "λB. dbupd$_{sst}$ (unlabel (dual$_{lsst}$ (B ·$_{lsst}$ ϑ))) 𝓘 DB"

  have s_in: "?s ∈ ?C ⟹ ?s ∈ set (unlabel (?R ·$_{lsst}$ ϑ))"
             "?s' ∈ ?C ⟹ ?s' ∈ set (unlabel (?R ·$_{lsst}$ ϑ))"
    using subst_lsst_append[of "transaction_receive T"]
          unlabel_append[of "transaction_receive T"]
    by auto

  have 1: "unlabel (dual$_{lsst}$ ((B ·$_{lsst}$ ϑ)@[(l,s) ·$_{lsstp}$ ϑ])) = unlabel (dual$_{lsst}$ (B ·$_{lsst}$ ϑ))@[s']"
    when B: "(l,s') = dual$_{lsstp}$ ((l,s) ·$_{lsstp}$ ϑ)" "s' = ⟨ac: t ∈ u⟩ ∨ s' = ⟨ac: t ≐ u⟩"
    for l s s' and B::"('fun,'atom,'sets,'lbl) prot_strand"
    using B unlabel_append dual$_{lsstp}$_subst dual$_{lsst}$_subst singleton_lst_proj(4)
          dual$_{lsst}$_subst_snoc subst_lsst_append subst_lsst_singleton
    by (metis (no_types, lifting) subst_apply_labeled_stateful_strand_step.simps)


  have 2: "strand_sem_stateful IK DB (unlabel (dual$_{lsst}$ (B ·$_{lsst}$ ϑ))@[s']) 𝓘"
    when B: "(l,s') = dual$_{lsstp}$ ((l,s) ·$_{lsstp}$ ϑ)"
            "prefix ((B ·$_{lsst}$ ϑ)@[(l,s) ·$_{lsstp}$ ϑ]) (?R ·$_{lsst}$ ϑ)"
            "s' = ⟨ac: t ∈ u⟩ ∨ s' = ⟨ac: t ≐ u⟩"
    for l s s' and B::"('fun,'atom,'sets,'lbl) prot_strand"
  proof -
    have "strand_sem_stateful IK DB ?S' 𝓘"
      using 𝓘 strand_sem_append_stateful[of IK DB _ _ 𝓘] transaction_dual_subst_unfold[of T ϑ]
      by fastforce
    thus ?thesis
```

```
    using B(2) 1[OF B(1,3)] strand_sem_append_stateful subst_lsst_append
    unfolding prefix_def unlabel_def dual_{lsst}_def by (metis (no_types) map_append)
qed

have s_sem:
    "?s ∈ ?C ⟹ (l,?s) = dual_{lsstp} ((l,s) ·_{lsstp} ϑ) ⟹ (t · ℐ, u · ℐ) ∈ ?dbupd B"
    "?s' ∈ ?C ⟹ (l,?s') = dual_{lsstp} ((l,s) ·_{lsstp} ϑ) ⟹ t · ℐ = u · ℐ"
  when B: "prefix ((B ·_{lsst} ϑ)@[(l,s) ·_{lsstp} ϑ]) (?R ·_{lsst} ϑ)"
  for l s and B::"('fun,'atom,'sets,'lbl) prot_strand"
  using 2[OF _ B] strand_sem_append_stateful[of IK DB "unlabel (dual_{lsst} (B ·_{lsst} ϑ))" _ ℐ]
  by (fastforce, fastforce)

have 3: "∀a ∈ set (unlabel (dual_{lsst} (B ·_{lsst} ϑ))). ¬is_Insert a ∧ ¬is_Delete a"
  when B: "prefix ((B ·_{lsst} ϑ)@[(l,s) ·_{lsstp} ϑ]) (?R ·_{lsst} ϑ)"
  for l s and B::"('fun,'atom,'sets,'lbl) prot_strand"
proof -
  have "∀a ∈ set (unlabel (dual_{lsst} (B ·_{lsst} ϑ))). ?Q a"
  proof
    fix a assume a: "a ∈ set (unlabel (dual_{lsst} (B ·_{lsst} ϑ)))"

    have "?P a" when a: "a ∈ set (unlabel ?R)" for a
      using a wellformed_transaction_unlabel_cases(1,2)[OF T_valid]
      unfolding unlabel_def by fastforce
    hence "?P a" when a: "a ∈ set (unlabel (?R ·_{lsst} ϑ))" for a
      using a stateful_strand_step_cases_subst(2,11)[of _ ϑ] subst_lsst_unlabel[of ?R ϑ]
      unfolding subst_apply_stateful_strand_def by auto
    hence B_P: "∀a ∈ set (unlabel (B ·_{lsst} ϑ)). ?P a"
      using unlabel_mono[OF set_mono_prefix[OF append_prefixD[OF B]]]  by blast

    obtain l where "(l,a) ∈ set (dual_{lsst} (B ·_{lsst} ϑ))"
      using a by (meson unlabel_mem_has_label)
    then obtain b where b: "(l,b) ∈ set (B ·_{lsst} ϑ)" "dual_{lsstp} (l,b) = (l,a)"
      using dual_{lsst}_memberD by blast
    hence "?P b" using B_P unfolding unlabel_def by fastforce
    thus "?Q a" using dual_{lsstp}_inv[OF b(2)] by (cases b) auto
  qed
  thus ?thesis by fastforce
qed

show "(t · ℐ, u · ℐ) ∈ DB" when s: "?s ∈ ?C"
proof -
  obtain l B s where B:
      "(l,?s) = dual_{lsstp} ((l,s) ·_{lsstp} ϑ)"
      "prefix ((B ·_{lsst} ϑ)@[(l,s) ·_{lsstp} ϑ]) (?R ·_{lsst} ϑ)"
    using s_in(1)[OF s] dual_{lsst}_unlabel_steps_iff(6)[of _ t u]
          dual_{lsst}_in_set_prefix_obtain_subst[of ?s ?R ϑ]
    by blast

  show ?thesis
    using 3[OF B(2)] s_sem(1)[OF B(2) s B(1)]
          dbupd_{sst}_no_upd[of "unlabel (dual_{lsst} (B ·_{lsst} ϑ))" ℐ DB]
    by simp
qed

show "t · ℐ = u · ℐ" when s: "?s' ∈ ?C"
proof -
  obtain l B s where B:
      "(l,?s') = dual_{lsstp} ((l,s) ·_{lsstp} ϑ)"
      "prefix ((B ·_{lsst} ϑ)@[(l,s) ·_{lsstp} ϑ]) (?R ·_{lsst} ϑ)"
    using s_in(2)[OF s] dual_{lsst}_unlabel_steps_iff(3)[of _ t u]
          dual_{lsst}_in_set_prefix_obtain_subst[of ?s' ?R ϑ]
    by blast
```

```
    show ?thesis
      using 3[OF B(2)] s_sem(2)[OF B(2) s B(1)]
            dbupd_sst_no_upd[of "unlabel (dual_lsst (B ·_lsst ϑ))" ℐ DB]
      by simp
  qed
qed

lemma wellformed_transaction_sem_neg_checks:
  assumes T_valid: "wellformed_transaction T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
    and "NegChecks X F G ∈ set (unlabel (transaction_checks T ·_lsst ϑ))"
  shows "negchecks_model ℐ DB X F G"
proof -
  let ?s = "NegChecks X F G"
  let ?R = "transaction_receive T@transaction_checks T"
  let ?R' = "unlabel (dual_lsst (?R ·_lsst ϑ))"
  let ?S = "λA. unlabel (dual_lsst (A ·_lsst ϑ))"
  let ?S' = "?S (transaction_receive T)@?S (transaction_checks T)"
  let ?P = "λa. is_Receive a ∨ is_Check_or_Assignment a"
  let ?Q = "λa. is_Send a ∨ is_Check_or_Assignment a"
  let ?U = "λδ. subst_domain δ = set X ∧ ground (subst_range δ)"

  have s: "?s ∈ set (unlabel (?R ·_lsst ϑ))"
    using assms(3) subst_lsst_append[of "transaction_receive T"]
          unlabel_append[of "transaction_receive T"]
    by auto

  obtain l B s where B:
      "(l,?s) = dual_lsstp ((l,s) ·_lsstp ϑ)"
      "prefix ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ]) (?R ·_lsst ϑ)"
    using s dual_lsst_unlabel_steps_iff(7)[of X F G]
          dual_lsst_in_set_prefix_obtain_subst[of ?s ?R ϑ]
    by blast

  have 1: "unlabel (dual_lsst ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ])) = unlabel (dual_lsst (B ·_lsst ϑ))@[?s]"
    using B(1) unlabel_append dual_lsstp_subst dual_lsst_subst singleton_lst_proj(4)
          dual_lsst_subst_snoc subst_lsst_append subst_lsst_singleton
    by (metis (no_types, lifting) subst_apply_labeled_stateful_strand_step.simps)

  have "strand_sem_stateful IK DB ?S' ℐ"
    using ℐ strand_sem_append_stateful[of IK DB _ _ ℐ] transaction_dual_subst_unfold[of T ϑ]
    by fastforce
  hence "strand_sem_stateful IK DB (unlabel (dual_lsst (B ·_lsst ϑ))@[?s]) ℐ"
    using B 1 strand_sem_append_stateful subst_lsst_append
    unfolding prefix_def unlabel_def dual_lsst_def
    by (metis (no_types) map_append)
  hence s_sem: "negchecks_model ℐ (dbupd_sst (unlabel (dual_lsst (B ·_lsst ϑ))) ℐ DB) X F G"
    using strand_sem_append_stateful[of IK DB "unlabel (dual_lsst (B ·_lsst ϑ))" "[?s]" ℐ]
    by fastforce

  have "∀a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ))). ?Q a"
  proof
    fix a assume a: "a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ)))"

    have "?P a" when a: "a ∈ set (unlabel ?R)" for a
      using a wellformed_transaction_unlabel_cases(1,2,3)[OF T_valid]
      unfolding unlabel_def by fastforce
    hence "?P a" when a: "a ∈ set (unlabel (?R ·_lsst ϑ))" for a
      using a stateful_strand_step_cases_subst(2,11)[of _ ϑ] subst_lsst_unlabel[of ?R ϑ]
      unfolding subst_apply_stateful_strand_def by auto
    hence B_P: "∀a ∈ set (unlabel (B ·_lsst ϑ)). ?P a"
      using unlabel_mono[OF set_mono_prefix[OF append_prefixD[OF B(2)]]]
      by blast
```

```
    obtain l where "(l,a) ∈ set (dual_lsst (B ·_lsst ϑ))"
      using a by (meson unlabel_mem_has_label)
    then obtain b where b: "(l,b) ∈ set (B ·_lsst ϑ)" "dual_lsstp (l,b) = (l,a)"
      using dual_lsst_memberD by blast
    hence "?P b" using B_P unfolding unlabel_def by fastforce
    thus "?Q a" using dual_lsstp_inv[OF b(2)] by (cases b) auto
  qed
  hence "∀a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ))). ¬is_Insert a ∧ ¬is_Delete a" by fastforce
  thus ?thesis using dbupd_sst_no_upd[of "unlabel (dual_lsst (B ·_lsst ϑ))" ℐ DB] s_sem by simp
qed

lemma wellformed_transaction_sem_neg_checks':
  assumes T_valid: "wellformed_transaction T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
    and c: "NegChecks X [] [(t,u)] ∈ set (unlabel (transaction_checks T ·_lsst ϑ))"
  shows "∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶ (t · δ · ℐ, u · δ · ℐ) ∉ DB" (is ?A)
    and "X = [] ⟹ (t · ℐ, u · ℐ) ∉ DB" (is "?B ⟹ ?B'")
proof -
  show ?A
    using wellformed_transaction_sem_neg_checks[OF T_valid ℐ c]
    unfolding negchecks_model_def by auto
  moreover have "δ = Var" "t · δ = t"
    when "subst_domain δ = set []" for t and δ::"('fun, 'atom, 'sets, 'lbl) prot_subst"
    using that by auto
  moreover have "subst_domain Var = set []" "range_vars Var = {}"
    by simp_all
  ultimately show "?B ⟹ ?B'" unfolding range_vars_alt_def by metis
qed

lemma wellformed_transaction_sem_iff:
  fixes T ϑ
  defines "A ≡ unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))"
    and "rm ≡ λX. rm_vars (set X)"
  assumes T: "wellformed_transaction T"
    and I: " interpretation_subst I" "wf_trms (subst_range I)"
  shows "strand_sem_stateful M D A I ⟷ (
    (∀l ts. (l,receive⟨ts⟩) ∈ set (transaction_receive T) ⟶ (∀t ∈ set ts. M ⊢ t · ϑ · I)) ∧
    (∀l ac t s. (l,⟨ac: t ≐ s⟩) ∈ set (transaction_checks T) ⟶ t · ϑ · I = s · ϑ · I) ∧
    (∀l ac t s. (l,⟨ac: t ∈ s⟩) ∈ set (transaction_checks T) ⟶ (t · ϑ · I, s · ϑ · I) ∈ D) ∧
    (∀l X F G. (l,∀X⟨∨≠: F ∨∉: G⟩) ∈ set (transaction_checks T) ⟶
      (∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶
            (∃(t,s) ∈ set F. t · rm X ϑ · δ · I ≠ s · rm X ϑ · δ · I) ∨
            (∃(t,s) ∈ set G. (t · rm X ϑ · δ · I, s · rm X ϑ · δ · I) ∉ D))))"
    (is "?A ⟷ ?B")
proof
  note 0 = A_def transaction_dual_subst_unlabel_unfold
  note 1 = wellformed_transaction_sem_receives[OF T, of M D ϑ I, unfolded A_def[symmetric]]
           wellformed_transaction_sem_pos_checks[OF T, of M D ϑ I, unfolded A_def[symmetric]]
           wellformed_transaction_sem_neg_checks[OF T, of M D ϑ I, unfolded A_def[symmetric]]
  note 2 = stateful_strand_step_subst_inI[OF unlabel_in]
  note 3 = unlabel_subst
  note 4 = strand_sem_append_stateful[of M D _ _ I]

  let ?C = "λT. unlabel (dual_lsst (T ·_lsst ϑ))"
  let ?P = "λX δ. subst_domain δ = set X ∧ ground (subst_range δ)"
  let ?sem = "λM D T. strand_sem_stateful M D (?C T) I"
  let ?negchecks = "λX F G. ∀δ. ?P X δ ⟶
                              (∃(t,s) ∈ set F. t · rm X ϑ · δ · I ≠ s · rm X ϑ · δ · I) ∨
                              (∃(t,s) ∈ set G. (t · rm X ϑ · δ · I, s · rm X ϑ · δ · I) ∉ D)"

  have "list_all is_Receive (unlabel (transaction_receive T))"
       "list_all is_Check_or_Assignment (unlabel (transaction_checks T))"
```

```
      "list_all is_Update (unlabel (transaction_updates T))"
      "list_all is_Send (unlabel (transaction_send T))"
    using T unfolding wellformed_transaction_def by (blast, blast, blast, blast)
  hence 5: "list_all is_Send (?C (transaction_receive T))"
           "list_all is_Check_or_Assignment (?C (transaction_checks T))"
           "list_all is_Update (?C (transaction_updates T))"
           "list_all is_Receive (?C (transaction_send T))"
    by (metis (no_types) subst_sst_list_all(2) unlabel_subst dual_{lsst}_list_all(1),
        metis (no_types) subst_sst_list_all(11) unlabel_subst dual_{lsst}_list_all(11),
        metis (no_types) subst_sst_list_all(10) unlabel_subst dual_{lsst}_list_all(10),
        metis (no_types) subst_sst_list_all(1) unlabel_subst dual_{lsst}_list_all(2))

  have "∀a ∈ set (?C (transaction_receive T)). ¬is_Receive a ∧ ¬is_Insert a ∧ ¬is_Delete a"
       "∀a ∈ set (?C (transaction_checks T)). ¬is_Receive a ∧ ¬is_Insert a ∧ ¬is_Delete a"
    using 5(1,2) unfolding list_all_iff by (blast,blast)
  hence 6:
      "M ∪ (ik_{sst} (?C (transaction_receive T)) ·_{set} I) = M"
      "dbupd_{sst} (?C (transaction_receive T)) I D = D"
      "M ∪ (ik_{sst} (?C (transaction_checks T)) ·_{set} I) = M"
      "dbupd_{sst} (?C (transaction_checks T)) I D = D"
    by (metis ik_{sst}_snoc_no_receive_empty sup_bot.right_neutral, metis dbupd_{sst}_no_upd,
        metis ik_{sst}_snoc_no_receive_empty sup_bot.right_neutral, metis dbupd_{sst}_no_upd)

  have ?B when A: ?A
  proof -
    have "M ⊢ t · ϑ · I"
      when "(l, receive⟨ts⟩) ∈ set (transaction_receive T)" "t ∈ set ts" for l ts t
      using that(2) 1(1)[OF A, of "ts ·_{list} ϑ"] 2(2)[OF that(1)] unfolding 3 by auto
    moreover have "t · ϑ · I = s · ϑ · I"
      when "(l, ⟨ac: t ≐ s⟩) ∈ set (transaction_checks T)" for l ac t s
      using 1(3)[OF A] 2(3)[OF that] unfolding 3 by blast
    moreover have "(t · ϑ · I, s · ϑ · I) ∈ D"
      when "(l, ⟨ac: t ∈ s⟩) ∈ set (transaction_checks T)" for l ac t s
      using 1(2)[OF A] 2(6)[OF that] unfolding 3 by blast
    moreover have "?negchecks X F G"
      when "(l, ∀X⟨∨≠: F ∨∉: G⟩) ∈ set (transaction_checks T)" for l X F G
      using 1(4)[OF A 2(7)[OF that, of ϑ, unfolded 3]]
      unfolding negchecks_model_def rm_def subst_apply_pairs_def by fastforce
    ultimately show ?B by blast
  qed
  thus "?A ⟹ ?B" by fast

  have ?A when B: ?B
  proof -
    have 7: "∀t ∈ set ts. M ⊢ t · I" when ts: "send⟨ts⟩ ∈ set (?C (transaction_receive T))" for ts
    proof -
      obtain l ss where "(l,receive⟨ss⟩) ∈ set (transaction_receive T)" "ts = ss ·_{list} ϑ"
        by (metis ts dual_{lsst}_unlabel_steps_iff(2) subst_lsst_memD(1) unlabel_mem_has_label)
      thus ?thesis using B by auto
    qed

    have 8: "t · I = s · I" when ts: "⟨ac: t ≐ s⟩ ∈ set (?C (transaction_checks T))" for ac t s
    proof -
      obtain l t' s' where "(l,⟨ac: t' ≐ s'⟩) ∈ set (transaction_checks T)" "t = t' · ϑ" "s = s' · ϑ"
        by (metis ts dual_{lsst}_unlabel_steps_iff(3) subst_lsst_memD(3) unlabel_mem_has_label)
      thus ?thesis using B by auto
    qed

    have 9: "(t · I, s · I) ∈ D" when ts: "⟨ac: t ∈ s⟩ ∈ set (?C (transaction_checks T))" for ac t s
    proof -
      obtain l t' s' where "(l,⟨ac: t' ∈ s'⟩) ∈ set (transaction_checks T)" "t = t' · ϑ" "s = s' · ϑ"
        by (metis ts dual_{lsst}_unlabel_steps_iff(6) subst_lsst_memD(6) unlabel_mem_has_label)
      thus ?thesis using B by auto
```

```
    qed

    have 10: "negchecks_model I D X F G"
      when ts: "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (?C (transaction_checks T))" for X F G
    proof -
      obtain l F' G' where *:
          "(l,∀X⟨∨≠: F' ∨∉: G'⟩) ∈ set (transaction_checks T)"
          "F = F' ·pairs rm_vars (set X) ϑ" "G = G' ·pairs rm_vars (set X) ϑ"
        using unlabel_mem_has_label[OF iffD2[OF duallsst_unlabel_steps_iff(7) ts]]
              subst_lsst_memD(7)[of _ X F G "transaction_checks T" ϑ]
        by fast

      have "?negchecks X F' G'" using *(1) B by blast
      moreover have "∃(t,s) ∈ set F. t · δ ∘s I ≠ s · δ ∘s I"
        when "(t,s) ∈ set F'" "t · rm X ϑ · δ · I ≠ s · rm X ϑ · δ · I" for δ t s
        using that unfolding rm_def *(2) subst_apply_pairs_def by force
      moreover have "∃(t,s) ∈ set G. (t,s) ·p δ ∘s I ∉ D"
        when "(t,s) ∈ set G'" "(t · rm X ϑ · δ · I, s · rm X ϑ · δ · I) ∉ D" for δ t s
        using that unfolding rm_def *(3) subst_apply_pairs_def by force
      ultimately show ?thesis
        unfolding negchecks_model_def by auto
    qed

    have "?sem M D (transaction_receive T)"
      using 7 strand_sem_stateful_if_sends_deduct[OF 5(1)] by blast
    moreover have "?sem M D (transaction_checks T)"
      using 8 9 10 strand_sem_stateful_if_checks[OF 5(2)] by blast
    moreover have "?sem M D (transaction_updates T)" for M D
      using 5(3) strand_sem_stateful_if_no_send_or_check unfolding list_all_iff by blast
    moreover have "?sem M D (transaction_send T)" for M D
      using 5(4) strand_sem_stateful_if_no_send_or_check unfolding list_all_iff by blast
    ultimately show ?thesis
      using 4[of "?C (transaction_receive T)"
                 "?C (transaction_checks T)@?C (transaction_updates T)@?C (transaction_send T)"]
            4[of "?C (transaction_checks T)" "?C (transaction_updates T)@?C (transaction_send T)"]
            4[of "?C (transaction_updates T)" "?C (transaction_send T)"]
      unfolding 0 6 by blast
  qed
  thus "?B ⟹ ?A" by fast
qed

lemma wellformed_transaction_unlabel_sem_iff:
  fixes T ϑ
  defines "A ≡ unlabel (duallsst (transaction_strand T ·lsst ϑ))"
    and "rm ≡ λX. rm_vars (set X)"
  assumes T: "wellformed_transaction T"
    and I: " interpretationsubst I" "wftrms (subst_range I)"
  shows "strand_sem_stateful M D A I ⟷ (
    (∀ts. receive⟨ts⟩ ∈ set (unlabel (transaction_receive T)) ⟶ (∀t ∈ set ts. M ⊢ t · ϑ · I)) ∧
    (∀ac t s. ⟨ac: t ≐ s⟩ ∈ set (unlabel (transaction_checks T)) ⟶ t · ϑ · I = s · ϑ · I) ∧
    (∀ac t s. ⟨ac: t ∈ s⟩ ∈ set (unlabel (transaction_checks T)) ⟶ (t · ϑ · I, s · ϑ · I) ∈ D) ∧
    (∀X F G. ∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel (transaction_checks T)) ⟶
      (∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶
            (∃(t,s) ∈ set F. t · rm X ϑ · δ · I ≠ s · rm X ϑ · δ · I) ∨
            (∃(t,s) ∈ set G. (t · rm X ϑ · δ · I, s · rm X ϑ · δ · I) ∉ D))))"
using wellformed_transaction_sem_iff[OF T I, of M D ϑ]
      unlabel_in[of _ _ "transaction_receive T"] unlabel_mem_has_label[of _ "transaction_receive T"]
      unlabel_in[of _ _ "transaction_checks T"] unlabel_mem_has_label[of _ "transaction_checks T"]
unfolding A_def[symmetric] rm_def by meson

lemma dual_transaction_ik_is_transaction_send'':
  fixes δ 𝓘::"('a,'b,'c,'d) prot_subst"
  assumes "wellformed_transaction T"
```

```
  shows "(ik_sst (unlabel (dual_lsst (transaction_strand T ·_lsst δ))) ·_set ℐ) ·_αset a =
         (trms_sst (unlabel (transaction_send T)) ·_set δ ·_set ℐ) ·_αset a" (is "?A = ?B")
using dual_transaction_ik_is_transaction_send[OF assms]
      subst_lsst_unlabel[of "dual_lsst (transaction_strand T)" δ]
      ik_sst_subst[of "unlabel (dual_lsst (transaction_strand T))" δ]
      dual_lsst_subst[of "transaction_strand T" δ]
by (auto simp add: abs_apply_terms_def)

lemma while_prot_terms_fun_mono:
  "mono (λM'. M ∪ ⋃(subterms ` M') ∪ ⋃((set ∘ fst ∘ Ana) ` M'))"
unfolding mono_def by fast

lemma while_prot_terms_SMP_overapprox:
  fixes M::"('fun,'atom,'sets,'lbl) prot_terms"
  assumes N_supset: "M ∪ ⋃(subterms ` N) ∪ ⋃((set ∘ fst ∘ Ana) ` N) ⊆ N"
    and Value_vars_only: "∀x ∈ fv_set N. Γ_v x = TAtom Value"
  shows "SMP M ⊆ {a · δ | a δ. a ∈ N ∧ wt_subst δ ∧ wf_trms (subst_range δ)}"
proof -
  define f where "f ≡ λM'. M ∪ ⋃(subterms ` M') ∪ ⋃((set ∘ fst ∘ Ana) ` M')"
  define S where "S ≡ {a · δ | a δ. a ∈ N ∧ wt_subst δ ∧ wf_trms (subst_range δ)}"

  note 0 = Value_vars_only

  have "t ∈ S" when "t ∈ SMP M" for t
  using that
  proof (induction t rule: SMP.induct)
    case (MP t)
    hence "t ∈ N" "wt_subst Var" "wf_trms (subst_range Var)" using N_supset by auto
    hence "t · Var ∈ S" unfolding S_def by blast
    thus ?case by simp
  next
    case (Subterm t t')
    then obtain δ a where a: "a · δ = t" "a ∈ N" "wt_subst δ" "wf_trms (subst_range δ)"
      by (auto simp add: S_def)
    hence "∀x ∈ fv a. ∃τ. Γ (Var x) = TAtom τ" using 0 by auto
    hence *: "∀x ∈ fv a. (∃f. δ x = Fun f []) ∨ (∃y. δ x = Var y)"
      using a(3) TAtom_term_cases[OF wf_trm_subst_rangeD[OF a(4)]]
      by (metis wt_subst_def)
    obtain b where b: "b · δ = t'" "b ∈ subterms a"
      using subterms_subst_subterm[OF *, of t'] Subterm.hyps(2) a(1)
      by fast
    hence "b ∈ N" using N_supset a(2) by blast
    thus ?case using a b(1) unfolding S_def by blast
  next
    case (Substitution t ϑ)
    then obtain δ a where a: "a · δ = t" "a ∈ N" "wt_subst δ" "wf_trms (subst_range δ)"
      by (auto simp add: S_def)
    have "wt_subst (δ ∘_s ϑ)" "wf_trms (subst_range (δ ∘_s ϑ))"
      by (fact wt_subst_compose[OF a(3) Substitution.hyps(2)],
          fact wf_trms_subst_compose[OF a(4) Substitution.hyps(3)])
    moreover have "t · ϑ = a · δ ∘_s ϑ" using a(1) subst_subst_compose[of a δ ϑ] by simp
    ultimately show ?case using a(2) unfolding S_def by blast
  next
    case (Ana t K T k)
    then obtain δ a where a: "a · δ = t" "a ∈ N" "wt_subst δ" "wf_trms (subst_range δ)"
      by (auto simp add: S_def)
    obtain Ka Ta where a': "Ana a = (Ka,Ta)" by force
    have *: "K = Ka ·_list δ"
    proof (cases a)
      case (Var x)
      then obtain g U where gU: "t = Fun g U"
        using a(1) Ana.hyps(2,3) Ana_var
        by (cases t) simp_all
```

```
    have "Γ (Var x) = TAtom Value" using Var a(2) 0 by auto
    hence "Γ (Fun g U) = TAtom Value"
      using a(1,3) Var gU wt_subst_trm''[OF a(3), of a]
      by argo
    thus ?thesis using gU Fun_Value_type_inv Ana.hyps(2,3) by fastforce
  next
    case (Fun g U) thus ?thesis using a(1) a' Ana.hyps(2) Ana_subst'[of g U] by simp
  qed
  then obtain ka where ka: "k = ka · δ" "ka ∈ set Ka" using Ana.hyps(3) by auto
  have "ka ∈ set ((fst ∘ Ana) a)" using ka(2) a' by simp
  hence "ka ∈ N" using a(2) N_supset by auto
  thus ?case using ka a(3,4) unfolding S_def by blast
 qed
 thus ?thesis unfolding S_def by blast
qed
```

3.3.6 Admissible Transactions

```
definition admissible_transaction_checks where
  "admissible_transaction_checks T ≡
    ∀x ∈ set (unlabel (transaction_checks T)).
      (is_InSet x ⟶
          is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
          fst (the_Var (the_elem_term x)) = TAtom Value) ∧
      (is_NegChecks x ⟶
          bvars_sstp x = [] ∧
          ((the_eqs x = [] ∧ length (the_ins x) = 1) ∨
           (the_ins x = [] ∧ length (the_eqs x) = 1))) ∧
      (is_NegChecks x ∧ the_eqs x = [] ⟶ (let h = hd (the_ins x) in
          is_Var (fst h) ∧ is_Fun_Set (snd h) ∧
          fst (the_Var (fst h)) = TAtom Value))"

definition admissible_transaction_updates where
  "admissible_transaction_updates T ≡
    ∀x ∈ set (unlabel (transaction_updates T)).
      is_Update x ∧ is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
      fst (the_Var (the_elem_term x)) = TAtom Value"

definition admissible_transaction_terms where
  "admissible_transaction_terms T ≡
    wf_trms' arity (trms_lsst (transaction_strand T)) ∧
    (∀f ∈ ⋃(funs_term ` trms_transaction T).
      ¬is_Val f ∧ ¬is_Abs f ∧ ¬is_PubConst f ∧ f ≠ Pair) ∧
    (∀r ∈ set (unlabel (transaction_strand T)).
      (∃f ∈ ⋃(funs_term ` (trms_sstp r)). is_Attack f) ⟶
        transaction_fresh T = [] ∧
        is_Send r ∧ length (the_msgs r) = 1 ∧ is_Fun_Attack (hd (the_msgs r)))"

definition admissible_transaction_send_occurs_form where
  "admissible_transaction_send_occurs_form T ≡ (
    let snds = transaction_send T;
        frsh = transaction_fresh T
    in ∀t ∈ trms_lsst snds. OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t ⟶
        (∃x ∈ set frsh. t = occurs (Var x))
)"

definition admissible_transaction_occurs_checks where
  "admissible_transaction_occurs_checks T ≡ (
    let occ_in = λx S. occurs (Var x) ∈ set (the_msgs (hd (unlabel S)));
        rcvs = transaction_receive T;
        snds = transaction_send T;
        frsh = transaction_fresh T;
        fvs = fv_transaction T
```

```
    in admissible_transaction_send_occurs_form T ∧
       ((∃x ∈ fvs - set frsh. fst x = TAtom Value) ⟶ (
          rcvs ≠ [] ∧ is_Receive (hd (unlabel rcvs)) ∧
          (∀x ∈ fvs - set frsh. fst x = TAtom Value ⟶ occ_in x rcvs))) ∧
       (frsh ≠ [] ⟶ (
          snds ≠ [] ∧ is_Send (hd (unlabel snds)) ∧
          (∀x ∈ set frsh. occ_in x snds)))
)"
```

definition `admissible_transaction_no_occurs_msgs` **where**

```
  "admissible_transaction_no_occurs_msgs T ≡ (
    let no_occ = λt. is_Fun t ⟶ the_Fun t ≠ OccursFact;
        rcvs = transaction_receive T;
        snds = transaction_send T
    in list_all (λa. is_Receive (snd a) ⟶ list_all no_occ (the_msgs (snd a))) rcvs ∧
       list_all (λa. is_Send (snd a)    ⟶ list_all no_occ (the_msgs (snd a))) snds
)"
```

definition `admissible_transaction'` **where**

```
  "admissible_transaction' T ≡ (
    wellformed_transaction T ∧
    transaction_decl T () = [] ∧
    list_all (λx. fst x = TAtom Value) (transaction_fresh T) ∧
    (∀x ∈ vars_transaction T. is_Var (fst x) ∧ (the_Var (fst x) = Value)) ∧
    bvars_lsst (transaction_strand T) = {} ∧
    set (transaction_fresh T) ⊆
      fv_lsst (filter (is_Insert ∘ snd) (transaction_updates T)) ∪ fv_lsst (transaction_send T) ∧
    (∀x ∈ fv_transaction T - set (transaction_fresh T).
     ∀y ∈ fv_transaction T - set (transaction_fresh T).
      x ≠ y ⟶ ⟨Var x != Var y⟩ ∈ set (unlabel (transaction_checks T)) ∨
               ⟨Var y != Var x⟩ ∈ set (unlabel (transaction_checks T))) ∧
    fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T) - set (transaction_fresh T)
      ⊆ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_checks T) ∧
    (∀r ∈ set (unlabel (transaction_checks T)).
      is_Equality r ⟶ fv (the_rhs r) ⊆ fv_lsst (transaction_receive T)) ∧
    fv_lsst (transaction_checks T) ⊆
      fv_lsst (transaction_receive T) ∪
      fv_lsst (filter (λs. is_InSet (snd s) ∧ the_check (snd s) = Assign) (transaction_checks T)) ∧
    list_all (λa. is_Receive (snd a) ⟶ the_msgs (snd a) ≠ []) (transaction_receive T) ∧
    list_all (λa. is_Send (snd a) ⟶ the_msgs (snd a) ≠ []) (transaction_send T) ∧
    admissible_transaction_checks T ∧
    admissible_transaction_updates T ∧
    admissible_transaction_terms T ∧
    admissible_transaction_send_occurs_form T
)"
```

definition `admissible_transaction` **where**

```
  "admissible_transaction T ≡
    admissible_transaction' T ∧
    admissible_transaction_no_occurs_msgs T"
```

definition `has_initial_value_producing_transaction` **where**

```
  "has_initial_value_producing_transaction P ≡
    let f = λs.
      list_all (λT. list_all (λa. ((is_Delete a ∨ is_InSet a) ⟶ the_set_term a ≠ ⟨s⟩_s) ∧
                                  (is_NegChecks a ⟶ list_all (λ(_,t). t ≠ ⟨s⟩_s) (the_ins a)))
                             (unlabel (transaction_checks T@transaction_updates T)))
                 P
    in list_ex (λT.
        length (transaction_fresh T) = 1 ∧ transaction_receive T = [] ∧
        transaction_checks T = [] ∧ length (transaction_send T) = 1 ∧
        (let x = hd (transaction_fresh T); a = hd (transaction_send T); u = transaction_updates T
         in is_Send (snd a) ∧ Var x ∈ set (the_msgs (snd a)) ∧
```

```
          fv_set (set (the_msgs (snd a))) = {x} ∧
          (u ≠ [] ⟶ (
            let b = hd u; c = snd b
            in tl u = [] ∧ is_Insert c ∧ the_elem_term c = Var x ∧
               is_Fun_Set (the_set_term c) ∧ f (the_Set (the_Fun (the_set_term c))))))
      ) P"


lemma admissible_transaction_is_wellformed_transaction:
  assumes "admissible_transaction' T"
  shows "wellformed_transaction T"
    and "admissible_transaction_checks T"
    and "admissible_transaction_updates T"
    and "admissible_transaction_terms T"
    and "admissible_transaction_send_occurs_form T"
using assms unfolding admissible_transaction'_def by blast+

lemma admissible_transaction_no_occurs_msgsE:
  assumes T: "admissible_transaction' T" "admissible_transaction_no_occurs_msgs T"
  shows "∀ts. send⟨ts⟩ ∈ set (unlabel (transaction_strand T)) ∨
             receive⟨ts⟩ ∈ set (unlabel (transaction_strand T)) ⟶
               (∀t s. t ∈ set ts ⟶ t ≠ occurs s)"
proof -
  note 1 = admissible_transaction_is_wellformed_transaction(1)[OF T(1)]

  have 2: "send⟨ts⟩ ∈ set (unlabel (transaction_send T))"
    when "send⟨ts⟩ ∈ set (unlabel (transaction_strand T))" for ts
    using wellformed_transaction_strand_unlabel_memberD(8)[OF 1 that] by fast

  have 3: "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T))"
    when "receive⟨ts⟩ ∈ set (unlabel (transaction_strand T))" for ts
    using wellformed_transaction_strand_unlabel_memberD(1)[OF 1 that] by fast

  show ?thesis
    using T(2) 2 3 wellformed_transaction_unlabel_cases(1,4)[OF 1]
    unfolding admissible_transaction_no_occurs_msgs_def Let_def list_all_iff
    by (metis sndI stateful_strand_step.discI(1,2) stateful_strand_step.sel(1,2)
             term.discI(2) term.sel(2) unlabel_mem_has_label)
qed

lemma admissible_transactionE:
  assumes T: "admissible_transaction' T"
  shows "transaction_decl T () = []" (is ?A)
    and "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value" (is ?B)
    and "∀x ∈ vars_lsst (transaction_strand T). Γ_v x = TAtom Value" (is ?C)
    and "bvars_lsst (transaction_strand T) = {}" (is ?D1)
    and "fv_transaction T ∩ bvars_transaction T = {}" (is ?D2)
    and "set (transaction_fresh T) ⊆
          fv_lsst (filter (is_Insert ∘ snd) (transaction_updates T)) ∪ fv_lsst (transaction_send T)"
      (is ?E)
    and "set (transaction_fresh T) ⊆ fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)"
      (is ?F)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀y ∈ fv_transaction T - set (transaction_fresh T).
          x ≠ y ⟶ ⟨Var x != Var y⟩ ∈ set (unlabel (transaction_checks T)) ∨
                    ⟨Var y != Var x⟩ ∈ set (unlabel (transaction_checks T))"
      (is ?G)
    and "∀x ∈ fv_lsst (transaction_checks T).
          x ∈ fv_lsst (transaction_receive T) ∨
          (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ x ∈ fv t ∪ fv s)"
      (is ?H)
    and "fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T) - set (transaction_fresh T) ⊆
          fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_checks T)"
```

```
      (is ?I)
    and "∀x ∈ set (unlabel (transaction_checks T)).
          is_Equality x ⟶ fv (the_rhs x) ⊆ fv_lsst (transaction_receive T)"
      (is ?J)
    and "set (transaction_fresh T) ∩ fv_lsst (transaction_receive T) = {}" (is ?K1)
    and "set (transaction_fresh T) ∩ fv_lsst (transaction_checks T) = {}" (is ?K2)
    and "list_all (λx. fst x = Var Value) (transaction_fresh T)" (is ?K3)
    and "∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γ_v x" (is ?K4)
    and "∀l ts. (l,receive⟨ts⟩) ∈ set (transaction_receive T) ⟶ ts ≠ []" (is ?L1)
    and "∀l ts. (l,send⟨ts⟩) ∈ set (transaction_send T) ⟶ ts ≠ []" (is ?L2)
proof -
  show ?A ?D1 ?D2 ?G ?I ?J ?K3
    using T unfolding admissible_transaction'_def
    by (blast, blast, blast, blast, blast, blast, blast)

  have "list_all (λa. is_Receive (snd a) ⟶ the_msgs (snd a) ≠ []) (transaction_receive T)"
       "list_all (λa. is_Send (snd a) ⟶ the_msgs (snd a) ≠ []) (transaction_send T)"
    using T unfolding admissible_transaction'_def by auto
  thus ?L1 ?L2 unfolding list_all_iff by (force,force)

  have "list_all (λx. fst x = Var Value) (transaction_fresh T)"
       "∀x∈vars_transaction T. is_Var (fst x) ∧ the_Var (fst x) = Value"
    using T unfolding admissible_transaction'_def by (blast, blast)
  thus ?B ?C ?K4 using Γ_v_TAtom''(2) unfolding list_all_iff by (blast, force, force)

  show ?E using T unfolding admissible_transaction'_def by argo
  thus ?F unfolding unlabel_def by auto

  show ?K1 ?K2
    using T unfolding admissible_transaction'_def wellformed_transaction_def by (argo, argo)

  let ?selects = "filter (λs. is_InSet (snd s) ∧ the_check (snd s) = Assign) (transaction_checks T)"

  show ?H
  proof
    fix x assume "x ∈ fv_lsst (transaction_checks T)"
    hence "x ∈ fv_lsst (transaction_receive T) ∨ x ∈ fv_lsst ?selects"
      using T unfolding admissible_transaction'_def by blast
    thus "x ∈ fv_lsst (transaction_receive T) ∨
          (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ x ∈ fv t ∪ fv s)"
    proof
      assume "x ∈ fv_lsst ?selects"
      then obtain r where r: "x ∈ fv_sstp r" "r ∈ set (unlabel (transaction_checks T))"
                             "is_InSet r" "the_check r = Assign"
        unfolding unlabel_def by force
      thus ?thesis by (cases r) auto
    qed simp
  qed
qed

lemma admissible_transactionE':
  assumes T: "admissible_transaction T"
  shows "admissible_transaction' T" (is ?A)
    and "admissible_transaction_no_occurs_msgs T" (is ?B)
    and "∀ts. send⟨ts⟩ ∈ set (unlabel (transaction_strand T)) ∨
              receive⟨ts⟩ ∈ set (unlabel (transaction_strand T)) ⟶
                (∀t s. t ∈ set ts ⟶ t ≠ occurs s)"
      (is ?C)
proof -
  show 0: ?A ?B using T unfolding admissible_transaction_def by (blast, blast)
  show ?C using admissible_transaction_no_occurs_msgsE[OF 0] by blast
qed
```

```
lemma transaction_inserts_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_updates T"
    and "insert⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n)"
    and "∃u. s = Fun (Set u) []"
proof -
  let ?x = "insert⟨t,s⟩"

  have "?x ∈ set (unlabel (transaction_updates T))"
    using assms(3) wellformed_transaction_unlabel_cases[OF T_valid, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)
  hence *: "is_Var (the_elem_term ?x)" "fst (the_Var (the_elem_term ?x)) = TAtom Value"
          "is_Fun (the_set_term ?x)" "args (the_set_term ?x) = []"
          "is_Set (the_Fun (the_set_term ?x))"
    using assms(2) unfolding admissible_transaction_updates_def is_Fun_Set_def by fastforce+

  show "∃n. t = Var (TAtom Value, n)" using *(1,2) by (cases t) auto
  show "∃u. s = Fun (Set u) []" using *(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma transaction_deletes_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_updates T"
    and "delete⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n)"
    and "∃u. s = Fun (Set u) []"
proof -
  let ?x = "delete⟨t,s⟩"

  have "?x ∈ set (unlabel (transaction_updates T))"
    using assms(3) wellformed_transaction_unlabel_cases[OF T_valid, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)
  hence *: "is_Var (the_elem_term ?x)" "fst (the_Var (the_elem_term ?x)) = TAtom Value"
          "is_Fun (the_set_term ?x)" "args (the_set_term ?x) = []"
          "is_Set (the_Fun (the_set_term ?x))"
    using assms(2) unfolding admissible_transaction_updates_def is_Fun_Set_def by fastforce+

  show "∃n. t = Var (TAtom Value, n)" using *(1,2) by (cases t) auto
  show "∃u. s = Fun (Set u) []" using *(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma transaction_selects_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_checks T"
    and "select⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n) ∧ (TAtom Value, n) ∉ set (transaction_fresh T)" (is ?A)
    and "∃u. s = Fun (Set u) []" (is ?B)
proof -
  let ?x = "select⟨t,s⟩"

  have *: "?x ∈ set (unlabel (transaction_checks T))"
    using assms(3) wellformed_transaction_unlabel_cases[OF T_valid, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)

  have **: "is_Var (the_elem_term ?x)" "fst (the_Var (the_elem_term ?x)) = TAtom Value"
          "is_Fun (the_set_term ?x)" "args (the_set_term ?x) = []"
          "is_Set (the_Fun (the_set_term ?x))"
    using * assms(2) unfolding admissible_transaction_checks_def is_Fun_Set_def by fastforce+

  have "fv_sstp ?x ⊆ fv_lsst (transaction_checks T)"
    using * by force
  hence ***: "fv_sstp ?x ∩ set (transaction_fresh T) = {}"
```

```
    using T_valid unfolding wellformed_transaction_def by fast

  show ?A using **(1,2) *** by (cases t) auto
  show ?B using **(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma transaction_inset_checks_are_Value_vars:
  assumes T_valid: "admissible_transaction' T"
    and t: "⟨t in s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n) ∧ (TAtom Value, n) ∉ set (transaction_fresh T)" (is ?A)
    and "∃u. s = Fun (Set u) []" (is ?B)
proof -
  let ?x = "⟨t in s⟩"

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_valid]
  note T_adm_checks = admissible_transaction_is_wellformed_transaction(2)[OF T_valid]

  have *: "?x ∈ set (unlabel (transaction_checks T))"
    using t wellformed_transaction_unlabel_cases[OF T_wf, of ?x]
    unfolding transaction_strand_def unlabel_def by fastforce

  have **: "is_Var (the_elem_term ?x)" "fst (the_Var (the_elem_term ?x)) = TAtom Value"
           "is_Fun (the_set_term ?x)" "args (the_set_term ?x) = []"
           "is_Set (the_Fun (the_set_term ?x))"
    using * T_adm_checks unfolding admissible_transaction_checks_def is_Fun_Set_def by fastforce+

  have "fv_sstp ?x ⊆ fv_lsst (transaction_checks T)"
    using * by force
  hence ***: "fv_sstp ?x ∩ set (transaction_fresh T) = {}"
    using T_wf unfolding wellformed_transaction_def by fast

  show ?A using **(1,2) *** by (cases t) auto
  show ?B using **(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma transaction_notinset_checks_are_Value_vars:
  assumes T_adm: "admissible_transaction' T"
    and FG: "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel (transaction_strand T))"
    and t: "(t,s) ∈ set G"
  shows "∃n. t = Var (TAtom Value, n) ∧ (TAtom Value, n) ∉ set (transaction_fresh T)" (is ?A)
    and "∃u. s = Fun (Set u) []" (is ?B)
    and "F = []" (is ?C)
    and "G = [(t,s)]" (is ?D)
proof -
  let ?x = "∀X⟨∨≠: F ∨∉: G⟩"

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]
  note T_adm_checks = admissible_transaction_is_wellformed_transaction(2)[OF T_adm]

  have 0: "?x ∈ set (unlabel (transaction_checks T))"
    using FG wellformed_transaction_unlabel_cases[OF T_wf, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)
  hence 1: "F = [] ∧ length G = 1"
    using T_adm_checks t unfolding admissible_transaction_checks_def by fastforce
  hence "hd G = (t,s)" using t by (cases "the_ins ?x") auto
  hence **: "is_Var t" "fst (the_Var t) = TAtom Value" "is_Fun s" "args s = []" "is_Set (the_Fun s)"
    using 1 Set.bspec[OF T_adm_checks[unfolded admissible_transaction_checks_def] 0]
    unfolding is_Fun_Set_def by auto

  show ?C using 1 by blast
  show ?D using 1 t by force

  have "fv_sstp ?x ⊆ fv_lsst (transaction_checks T)"
```

```
      "set (bvars_sstp ?x) ⊆ bvars_lsst (transaction_checks T)"
    using 0 by force+
  moreover have
      "set (transaction_fresh T) ∩ fv_lsst (transaction_receive T) = {}"
      "set (transaction_fresh T) ∩ fv_lsst (transaction_checks T) = {}"
    using T_wf unfolding wellformed_transaction_def by fast+
  ultimately have
      "fv_sstp ?x ∩ set (transaction_fresh T) = {}"
      "set (bvars_sstp ?x) ∩ set (transaction_fresh T) = {}"
    using admissible_transactionE(7)[OF T_adm]
          wellformed_transaction_wf_sst(2)[OF T_wf]
          fv_transaction_unfold[of T] bvars_transaction_unfold[of T]
    by (blast, blast)
  hence ***: "fv t ∩ set (transaction_fresh T) = {}"
    using t by auto

  show ?A using **(1,2) *** by (cases t) auto
  show ?B using **(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma transaction_noteqs_checks_case:
  assumes T_adm: "admissible_transaction' T"
    and FG: "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel (transaction_strand T))"
    and G: "G = []"
  shows "∃t s. F = [(t,s)]" (is ?A)
proof -
  let ?x = "∀X⟨∨≠: F ∨∉: G⟩"

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]
  note T_adm_checks = admissible_transaction_is_wellformed_transaction(2)[OF T_adm]

  have "?x ∈ set (unlabel (transaction_checks T))"
    using FG wellformed_transaction_unlabel_cases[OF T_wf, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)
  hence "length F = 1"
    using T_adm_checks unfolding admissible_transaction_checks_def G by fastforce
  thus ?thesis by fast
qed

lemma admissible_transaction_fresh_vars_notin:
  assumes T: "admissible_transaction' T"
    and x: "x ∈ set (transaction_fresh T)"
  shows "x ∉ fv_lsst (transaction_receive T)" (is ?A)
    and "x ∉ fv_lsst (transaction_checks T)" (is ?B)
    and "x ∉ vars_lsst (transaction_receive T)" (is ?C)
    and "x ∉ vars_lsst (transaction_checks T)" (is ?D)
    and "x ∉ bvars_lsst (transaction_receive T)" (is ?E)
    and "x ∉ bvars_lsst (transaction_checks T)" (is ?F)
proof -
  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T]

  have 0:
      "set (transaction_fresh T) ⊆ fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)"
      "set (transaction_fresh T) ∩ fv_lsst (transaction_receive T) = {}"
      "set (transaction_fresh T) ∩ fv_lsst (transaction_checks T) = {}"
      "fv_transaction T ∩ bvars_transaction T = {}"
    using admissible_transactionE[OF T] by argo+

  have 1: "set (transaction_fresh T) ∩ bvars_lsst (transaction_checks T) = {}"
    using 0(1,4) fv_transaction_unfold[of T] bvars_transaction_unfold[of T] by blast

  have 2:
      "vars_lsst (transaction_receive T) = fv_lsst (transaction_receive T)"
```

```
      "bvars_lsst (transaction_receive T) = {}"
    using bvars_wellformed_transaction_unfold[OF T_wf]
          vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_receive T)"]
    by blast+

  show ?A ?B ?C ?E ?F using 0 1 2 x by (fast, fast, fast, fast, fast)

  show ?D using 0(3) 1 x vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_checks T)"] by fast
qed

lemma admissible_transaction_fv_in_receives_or_selects:
  assumes T: "admissible_transaction' T"
    and x: "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "x ∈ fv_lsst (transaction_receive T) ∨
         (x ∈ fv_lsst (transaction_checks T) ∧
          (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ x ∈ fv t ∪ fv s))"
proof -
  have "x ∈ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_checks T) ∪
           fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)"
    using x(1) fv_sst_append unlabel_append
    by (metis transaction_strand_def append_assoc)
  thus ?thesis using x(2) admissible_transactionE(9,10)[OF T] by blast
qed

lemma admissible_transaction_fv_in_receives_or_selects':
  assumes T: "admissible_transaction' T"
    and x: "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "(∃ts. receive⟨ts⟩ ∈ set (unlabel (transaction_receive T)) ∧ x ∈ fv_set (set ts)) ∨
         (∃s. select⟨Var x, s⟩ ∈ set (unlabel (transaction_checks T)))"
proof (cases "x ∈ fv_lsst (transaction_receive T)")
  case True thus ?thesis
    using wellformed_transaction_unlabel_cases(1)[
            OF admissible_transaction_is_wellformed_transaction(1)[OF T]]
    by force
next
  case False
  then obtain t s where t: "select⟨t,s⟩ ∈ set (unlabel (transaction_checks T))" "x ∈ fv t ∪ fv s"
    using admissible_transaction_fv_in_receives_or_selects[OF T x] by blast

  have t': "select⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
    using t(1) unfolding transaction_strand_def by simp

  show ?thesis
    using t transaction_selects_are_Value_vars[
              OF admissible_transaction_is_wellformed_transaction(1,2)[OF T] t']
    by force
qed

lemma admissible_transaction_fv_in_receives_or_selects_subst:
  assumes T: "admissible_transaction' T"
    and x: "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "(∃ts. receive⟨ts⟩ ∈ set (unlabel (transaction_receive T ·lsst ϑ)) ∧ ϑ x ⊑set set ts) ∨
         (∃s. select⟨ϑ x, s⟩ ∈ set (unlabel (transaction_checks T ·lsst ϑ)))"
proof -
  note 0 = admissible_transaction_fv_in_receives_or_selects'[OF T x]

  have 1: "ϑ x ⊑set set (ts ·list ϑ)" when ts: "x ∈ fv_set (set ts)" for ts
    using that subst_mono_fv[of x _ ϑ] by auto

  have 2: "receive⟨ts ·list ϑ⟩ ∈ set (A ·sst ϑ)" when "receive⟨ts⟩ ∈ set A" for ts A
    using that by fast

  have 3: "select⟨t · ϑ,s · ϑ⟩ ∈ set (A ·sst ϑ)" when "select⟨t,s⟩ ∈ set A" for t s A
```

```
    using that by fast

  show ?thesis
    using 0 1 2[of _ "unlabel (transaction_receive T)"]
          3[of _ _ "unlabel (transaction_checks T)"]
    unfolding unlabel_subst by (metis eval_term.simps(1))
qed

lemma admissible_transaction_fv_in_receives_or_selects_dual_subst:
  defines "f ≡ λS. unlabel (dual_lsst S)"
  assumes T: "admissible_transaction' T"
    and x: "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "(∃ts. send⟨ts⟩ ∈ set (f (transaction_receive T ·_lsst ϑ)) ∧ ϑ x ⊑_set set ts) ∨
         (∃s. select⟨ϑ x, s⟩ ∈ set (f (transaction_checks T ·_lsst ϑ)))"
using admissible_transaction_fv_in_receives_or_selects_subst[OF T x, of ϑ]
by (metis (no_types, lifting) f_def dual_lsst_unlabel_steps_iff(2) dual_lsst_unlabel_steps_iff(6))

lemma admissible_transaction_decl_subst_empty':
  assumes T: "transaction_decl T () = []"
    and ξ: "transaction_decl_subst ξ T"
  shows "ξ = Var"
proof -
  have "subst_domain ξ = {}"
    using ξ T unfolding transaction_decl_subst_def by auto
  thus ?thesis by auto
qed

lemma admissible_transaction_decl_subst_empty:
  assumes T: "admissible_transaction' T"
    and ξ: "transaction_decl_subst ξ T"
  shows "ξ = Var"
by (rule admissible_transaction_decl_subst_empty'[OF admissible_transactionE(1)[OF T] ξ])

lemma admissible_transaction_no_bvars:
  assumes "admissible_transaction' T"
  shows "fv_transaction T = vars_transaction T"
    and "bvars_transaction T = {}"
using admissible_transactionE(4)[OF assms]
      bvars_wellformed_transaction_unfold vars_sst_is_fv_sst_bvars_sst
by (fast, fast)

lemma admissible_transactions_fv_bvars_disj:
  assumes "∀T ∈ set P. admissible_transaction' T"
  shows "(⋃T ∈ set P. fv_transaction T) ∩ (⋃T ∈ set P. bvars_transaction T) = {}"
using assms admissible_transaction_no_bvars(2) by fast

lemma admissible_transaction_occurs_fv_types:
  assumes "admissible_transaction' T"
    and "x ∈ vars_transaction T"
  shows "∃a. Γ (Var x) = TAtom a ∧ Γ (Var x) ≠ TAtom OccursSecType"
proof -
  have "is_Var (fst x)" "the_Var (fst x) = Value"
    using assms unfolding admissible_transaction'_def by blast+
  thus ?thesis using Γ_v_TAtom''(2)[of x] by force
qed

lemma admissible_transaction_Value_vars_are_fv:
  assumes "admissible_transaction' T"
    and "x ∈ vars_transaction T"
    and "Γ_v x = TAtom Value"
  shows "x ∈ fv_transaction T"
using assms(2,3) Γ_v_TAtom''(2)[of x] vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_strand T)"]
      admissible_transactionE(4)[OF assms(1)]
```

```
by fast

lemma transaction_receive_deduct:
  assumes T_wf: "wellformed_transaction T"
    and ℐ: "constraint_model ℐ (A@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T M"
    and α: "transaction_renaming_subst α P X"
    and t: "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T ·_lsst ξ ∘_s σ ∘_s α))"
  shows "∀t ∈ set ts. ik_lsst A ·_set ℐ ⊢ t · ℐ"
proof -
  define ϑ where "ϑ ≡ ξ ∘_s σ ∘_s α"

  have t': "send⟨ts⟩ ∈ set (unlabel (dual_lsst (transaction_receive T ·_lsst ϑ)))"
    using t dual_lsst_unlabel_steps_iff(2) unfolding ϑ_def by blast
  then obtain T1 T2 where T: "unlabel (dual_lsst (transaction_receive T ·_lsst ϑ)) = T1@send⟨ts⟩#T2"
    using t' by (meson split_list)

  have "constr_sem_stateful ℐ (unlabel A@unlabel (dual_lsst (transaction_strand T ·_lsst ϑ)))"
    using ℐ unlabel_append[of A] unfolding constraint_model_def ϑ_def by simp
  hence "constr_sem_stateful ℐ (unlabel A@T1@[send⟨ts⟩])"
    using strand_sem_append_stateful[of "{}" "{}" "unlabel A@T1@[send⟨ts⟩]" _ ℐ]
          transaction_dual_subst_unlabel_unfold[of T ϑ] T
    by (metis append.assoc append_Cons append_Nil)
  hence "∀t ∈ set ts. ik_sst (unlabel A@T1) ·_set ℐ ⊢ t · ℐ"
    using strand_sem_append_stateful[of "{}" "{}" "unlabel A@T1" "[send⟨ts⟩]" ℐ] T
    by force
  moreover have "¬is_Receive x"
    when x: "x ∈ set (unlabel (dual_lsst (transaction_receive T ·_lsst ϑ)))" for x
  proof -
    have *: "is_Receive a" when "a ∈ set (unlabel (transaction_receive T))" for a
      using T_wf Ball_set[of "unlabel (transaction_receive T)" is_Receive] that
      unfolding wellformed_transaction_def
      by blast

    obtain l where l: "(l,x) ∈ set (dual_lsst (transaction_receive T ·_lsst ϑ))"
      using x unfolding unlabel_def by fastforce
    then obtain ly where ly: "ly ∈ set (transaction_receive T ·_lsst ϑ)" "(l,x) = dual_lsstp ly"
      unfolding dual_lsst_def by auto

    obtain j y where j: "ly = (j,y)" by (metis surj_pair)
    hence "j = l" using ly(2) by (cases y) auto
    hence y: "(l,y) ∈ set (transaction_receive T ·_lsst ϑ)" "(l,x) = dual_lsstp (l,y)"
      by (metis j ly(1), metis j ly(2))

    obtain z where z:
        "z ∈ set (unlabel (transaction_receive T))"
        "(l,z) ∈ set (transaction_receive T)"
        "(l,y) = (l,z) ·_lsstp ϑ"
      using y(1) unfolding subst_apply_labeled_stateful_strand_def unlabel_def by force

    have "is_Receive y" using *[OF z(1)] z(3) by (cases z) auto
    thus "¬is_Receive x" using l y by (cases y) auto
  qed
  hence "¬is_Receive x" when "x ∈ set T1" for x using T that by simp
  hence "ik_sst T1 = {}" unfolding ik_sst_def is_Receive_def by fast
  hence "ik_sst (unlabel A@T1) = ik_lsst A" using ik_sst_append[of "unlabel A" T1] by simp
  ultimately show ?thesis by (simp add: ϑ_def)
qed

lemma transaction_checks_db:
  assumes T: "admissible_transaction' T"
    and ℐ: "constraint_model ℐ (A@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α))"
```

```
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T M"
    and α: "transaction_renaming_subst α P X"
  shows "⟨Var (TAtom Value, n) in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
          ⟹ (α (TAtom Value, n) · ℐ, Fun (Set s) []) ∈ set (db_lsst A ℐ)"
      (is "?A ⟹ ?B")
    and "⟨Var (TAtom Value, n) not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
          ⟹ (α (TAtom Value, n) · ℐ, Fun (Set s) []) ∉ set (db_lsst A ℐ)"
      (is "?C ⟹ ?D")
proof -
  let ?x = "λn. (TAtom Value, n)"
  let ?s = "Fun (Set s) []"
  let ?T = "transaction_receive T@transaction_checks T"
  let ?T' = "?T ·_lsst ξ ∘_s σ ∘_s α"
  let ?S = "λS. transaction_receive T@S"
  let ?S' = "λS. ?S S ·_lsst ξ ∘_s σ ∘_s α"

  note ξ_empty = admissible_transaction_decl_subst_empty[OF T ξ]

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T]

  have "constr_sem_stateful ℐ (unlabel (A@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)))"
    using ℐ unfolding constraint_model_def by simp
  moreover have
      "dual_lsst (transaction_strand T ·_lsst δ) =
       dual_lsst (?S (T1@[c]) ·_lsst δ)@
       dual_lsst (T2@transaction_updates T@transaction_send T ·_lsst δ)"
    when "transaction_checks T = T1@c#T2" for T1 T2 c δ
    using that dual_lsst_append subst_lsst_append
    unfolding transaction_strand_def
    by (metis append.assoc append_Cons append_Nil)
  ultimately have T'_model: "constr_sem_stateful ℐ (unlabel (A@dual_lsst (?S' (T1@[(l,c)]))))"
    when "transaction_checks T = T1@(l,c)#T2" for T1 T2 l c
    using strand_sem_append_stateful[of _ _ _ _ ℐ]
    by (simp add: that transaction_strand_def)

  show "?A ⟹ ?B"
  proof -
    assume a: ?A
    hence *: "⟨Var (?x n) in ?s⟩ ∈ set (unlabel ?T)"
      unfolding transaction_strand_def unlabel_def by simp
    then obtain l T1 T2 where T1: "transaction_checks T = T1@(l,⟨Var (?x n) in ?s⟩)#T2"
      by (metis a split_list unlabel_mem_has_label)

    have "?x n ∈ fv_lsst (transaction_checks T)"
      using a by force
    hence "?x n ∉ set (transaction_fresh T)"
      using a admissible_transaction_fresh_vars_notin[OF T] by fast
    hence "unlabel (A@dual_lsst (?S' (T1@[(l,⟨Var (?x n) in ?s⟩)]))) =
          unlabel (A@dual_lsst (?S' T1))@[⟨α (?x n) in ?s⟩]"
      using T a σ dual_lsst_append subst_lsst_append unlabel_append ξ_empty
      by (fastforce simp add: transaction_fresh_subst_def unlabel_def dual_lsst_def
                              subst_apply_labeled_stateful_strand_def subst_compose)
    moreover have "db_sst (unlabel A) = db_sst (unlabel (A@dual_lsst (?S' T1)))"
      by (simp add: T1 db_sst_transaction_prefix_eq[OF T_wf] del: unlabel_append)
    ultimately have "∃M. strand_sem_stateful M (set (db_sst (unlabel A) ℐ)) [⟨α (?x n) in ?s⟩] ℐ"
      using T'_model[OF T1] db_sst_set_is_dbupd_sst[of _ ℐ] strand_sem_append_stateful[of _ _ _ _ ℐ]
      by (simp add: db_sst_def del: unlabel_append)
    thus ?B by simp
  qed

  show "?C ⟹ ?D"
  proof -
```

```
    assume a: ?C
    hence *: "⟨Var (?x n) not in ?s⟩ ∈ set (unlabel ?T)"
      unfolding transaction_strand_def unlabel_def by simp
    then obtain l T1 T2 where T1: "transaction_checks T = T1@(l,⟨Var (?x n) not in ?s⟩)#T2"
      by (metis a split_list unlabel_mem_has_label)

    have "?x n ∈ vars_sstp ⟨Var (?x n) not in ?s⟩"
      using vars_sstp_cases(9)[of "[]" "Var (?x n)" ?s] by auto
    hence "?x n ∈ vars_lsst (transaction_checks T)"
      using a unfolding vars_sst_def by force
    hence "?x n ∉ set (transaction_fresh T)"
      using a admissible_transaction_fresh_vars_notin[OF T] by fast
    hence "unlabel (A@dual_lsst (?S' (T1@[(l,⟨Var (?x n) not in ?s⟩)]))) =
          unlabel (A@dual_lsst (?S' T1))@[⟨α (?x n) not in ?s⟩]"
      using T a σ dual_lsst_append subst_lsst_append unlabel_append ξ_empty
      by (fastforce simp add: transaction_fresh_subst_def unlabel_def dual_lsst_def
                           subst_apply_labeled_stateful_strand_def subst_compose)
    moreover have "db_sst (unlabel A) = db_sst (unlabel (A@dual_lsst (?S' T1)))"
      by (simp add: T1 db_sst_transaction_prefix_eq[OF T_wf] del: unlabel_append)
    ultimately have "∃M. strand_sem_stateful M (set (db_sst (unlabel A) ℐ)) [⟨α (?x n) not in ?s⟩] ℐ"
      using T'_model[OF T1] db_sst_set_is_dbupd_sst[of _ ℐ] strand_sem_append_stateful[of _ _ _ _ ℐ]
      by (simp add: db_sst_def del: unlabel_append)
    thus ?D using stateful_strand_sem_NegChecks_no_bvars(1)[of _ _ _ ?s ℐ] by simp
  qed
qed

lemma transaction_selects_db:
  assumes T: "admissible_transaction' T"
    and ℐ: "constraint_model ℐ (A@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T M"
    and α: "transaction_renaming_subst α P X"
  shows "select⟨Var (TAtom Value, n), Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
          ⟹ (α (TAtom Value, n) · ℐ, Fun (Set s) []) ∈ set (db_lsst A ℐ)"
      (is "?A ⟹ ?B")
proof -
  let ?x = "λn. (TAtom Value, n)"
  let ?s = "Fun (Set s) []"
  let ?T = "transaction_receive T@transaction_checks T"
  let ?T' = "?T ·lsst ξ ∘s σ ∘s α"
  let ?S = "λS. transaction_receive T@S"
  let ?S' = "λS. ?S S ·lsst ξ ∘s σ ∘s α"

  note ξ_empty = admissible_transaction_decl_subst_empty[OF T ξ]

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T]

  have "constr_sem_stateful ℐ (unlabel (A@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)))"
    using ℐ unfolding constraint_model_def by simp
  moreover have
      "dual_lsst (transaction_strand T ·lsst δ) =
       dual_lsst (?S (T1@[c]) ·lsst δ)@
       dual_lsst (T2@transaction_updates T@transaction_send T ·lsst δ)"
    when "transaction_checks T = T1@c#T2" for T1 T2 c δ
    using that dual_lsst_append subst_lsst_append
    unfolding transaction_strand_def by (metis append.assoc append_Cons append_Nil)
  ultimately have T'_model: "constr_sem_stateful ℐ (unlabel (A@dual_lsst (?S' (T1@[(l,c)]))))"
    when "transaction_checks T = T1@(l,c)#T2" for T1 T2 l c
    using strand_sem_append_stateful[of _ _ _ _ ℐ]
    by (simp add: that transaction_strand_def)

  show "?A ⟹ ?B"
  proof -
```

```
    assume a: ?A
    hence *: "select⟨Var (?x n), ?s⟩ ∈ set (unlabel ?T)"
      unfolding transaction_strand_def unlabel_def by simp
    then obtain l T1 T2 where T1: "transaction_checks T = T1@(l,select⟨Var (?x n), ?s⟩)#T2"
      by (metis a split_list unlabel_mem_has_label)

    have "?x n ∈ fv_lsst (transaction_checks T)"
      using a by force
    hence "?x n ∉ set (transaction_fresh T)"
      using a admissible_transaction_fresh_vars_notin[OF T] by fast
    hence "unlabel (A@dual_lsst (?S' (T1@[(l,select⟨Var (?x n), ?s⟩)]))) =
           unlabel (A@dual_lsst (?S' T1))@[select⟨α (?x n), ?s⟩]"
      using T a σ dual_lsst_append subst_lsst_append unlabel_append ξ_empty
      by (fastforce simp add: transaction_fresh_subst_def unlabel_def dual_lsst_def
                              subst_apply_labeled_stateful_strand_def subst_compose)
    moreover have "db_sst (unlabel A) = db_sst (unlabel (A@dual_lsst (?S' T1)))"
      by (simp add: T1 db_sst_transaction_prefix_eq[OF T_wf] del: unlabel_append)
    ultimately have "∃M. strand_sem_stateful M (set (db_sst (unlabel A) ℐ)) [⟨α (?x n) in ?s⟩] ℐ"
      using T'_model[OF T1] db_sst_set_is_dbupd_sst[of _ ℐ] strand_sem_append_stateful[of _ _ _ _ ℐ]
      by (simp add: db_sst_def del: unlabel_append)
    thus ?B by simp
  qed
qed

lemma admissible_transaction_terms_no_Value_consts:
  assumes "admissible_transaction_terms T"
    and "t ∈ subterms_set (trms_lsst (transaction_strand T))"
  shows "∄a T. t = Fun (Val a) T" (is ?A)
    and "∄a T. t = Fun (Abs a) T" (is ?B)
    and "∄a T. t = Fun (PubConst Value a) T" (is ?C)
proof -
  have "¬is_Val f" "¬is_Abs f" "¬is_PubConstValue f"
    when "f ∈ ⋃(funs_term ` (trms_transaction T))" for f
    using that assms(1)[unfolded admissible_transaction_terms_def]
    unfolding is_PubConstValue_def by (blast,blast,blast)
  moreover have "f ∈ ⋃(funs_term ` (trms_transaction T))"
    when "f ∈ funs_term t" for f
    using that assms(2) funs_term_subterms_eq(2)[of "trms_transaction T"] by blast+
  ultimately have *: "¬is_Val f" "¬is_Abs f" "¬is_PubConstValue f"
    when "f ∈ funs_term t" for f
    using that by presburger+

  show ?A using *(1) by force
  show ?B using *(2) by force
  show ?C using *(3) unfolding is_PubConstValue_def by force
qed

lemma admissible_transactions_no_Value_consts:
  assumes "admissible_transaction' T"
    and "t ∈ subterms_set (trms_lsst (transaction_strand T))"
  shows "∄a T. t = Fun (Val a) T" (is ?A)
    and "∄a T. t = Fun (Abs a) T" (is ?B)
    and "∄a T. t = Fun (PubConst Value a) T" (is ?C)
using admissible_transaction_terms_no_Value_consts[OF
        admissible_transaction_is_wellformed_transaction(4)[OF assms(1)] assms(2)]
by auto

lemma admissible_transactions_no_Value_consts':
  assumes "admissible_transaction' T"
    and "t ∈ trms_lsst (transaction_strand T)"
  shows "∄a T. Fun (Val a) T ∈ subterms t"
    and "∄a T. Fun (Abs a) T ∈ subterms t"
using admissible_transactions_no_Value_consts[OF assms(1)] assms(2) by fast+
```

```
lemma admissible_transactions_no_Value_consts'':
  assumes "admissible_transaction' T"
  shows "∀n. PubConst Value n ∉ ⋃(funs_term ` trms_transaction T)"
    and "∀n. Abs n ∉ ⋃(funs_term ` trms_transaction T)"
using assms
unfolding admissible_transaction'_def admissible_transaction_terms_def
by (meson prot_fun.discI(6), meson prot_fun.discI(4))

lemma admissible_transactions_no_PubConsts:
  assumes "admissible_transaction' T"
    and "t ∈ subterms_set (trms_lsst (transaction_strand T))"
  shows "∄a n T. t = Fun (PubConst a n) T"
proof -
  have "¬is_PubConst f"
    when "f ∈ ⋃(funs_term ` (trms_transaction T))" for f
    using that conjunct1[OF conjunct2[OF admissible_transaction_is_wellformed_transaction(4)[
            OF assms(1), unfolded admissible_transaction_terms_def]]]
    by blast
  moreover have "f ∈ ⋃(funs_term ` (trms_transaction T))"
    when "f ∈ funs_term t" for f
    using that assms(2) funs_term_subterms_eq(2)[of "trms_transaction T"] by blast+
  ultimately have *: "¬is_PubConst f"
    when "f ∈ funs_term t" for f
    using that by presburger+

  show ?thesis using * by force
qed

lemma admissible_transactions_no_PubConsts':
  assumes "admissible_transaction' T"
    and "t ∈ trms_lsst (transaction_strand T)"
  shows "∄a n T. Fun (PubConst a n) T ∈ subterms t"
using admissible_transactions_no_PubConsts[OF assms(1)] assms(2) by fast+

lemma admissible_transaction_strand_step_cases:
  assumes T_adm: "admissible_transaction' T"
  shows "r ∈ set (unlabel (transaction_receive T)) ⟹ ∃t. r = receive⟨t⟩"
        (is "?A ⟹ ?A'")
    and "r ∈ set (unlabel (transaction_checks T)) ⟹
            (∃x s. (r = ⟨Var x in Fun (Set s) []⟩ ∨ r = select⟨Var x, Fun (Set s) []⟩ ∨
                    r = ⟨Var x not in Fun (Set s) []⟩) ∧
                    fst x = TAtom Value ∧ x ∈ fv_transaction T - set (transaction_fresh T)) ∨
            (∃s t. r = ⟨s == t⟩ ∨ r = ⟨s := t⟩ ∨ r = ⟨s != t⟩)"
        (is "?B ⟹ ?B'")
    and "r ∈ set (unlabel (transaction_updates T)) ⟹
            ∃x s. (r = insert⟨Var x, Fun (Set s) []⟩ ∨ r = delete⟨Var x, Fun (Set s) []⟩) ∧
                  fst x = TAtom Value"
        (is "?C ⟹ ?C'")
    and "r ∈ set (unlabel (transaction_send T)) ⟹ ∃t. r = send⟨t⟩"
        (is "?D ⟹ ?D'")
proof -
  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

  show "?A ⟹ ?A'"
    using T_wf Ball_set[of "unlabel (transaction_receive T)" is_Receive]
    unfolding wellformed_transaction_def is_Receive_def
    by blast

  show "?D ⟹ ?D'"
    using T_wf Ball_set[of "unlabel (transaction_send T)" is_Send]
    unfolding wellformed_transaction_def is_Send_def
    by blast
```

```
show "?B ⟹ ?B'"
proof -
  assume r: ?B
  note adm_checks = admissible_transaction_is_wellformed_transaction(1,2)[OF T_adm]

  have fv_r1: "fv_sstp r ⊆ fv_transaction T"
    using r fv_transaction_unfold[of T] by auto

  have fv_r2: "fv_sstp r ∩ set (transaction_fresh T) = {}"
    using r T_wf unfolding wellformed_transaction_def by fastforce

  have "list_all is_Check_or_Assignment (unlabel (transaction_checks T))"
    using adm_checks(1) unfolding wellformed_transaction_def by blast
  hence "is_InSet r ∨ is_Equality r ∨ is_NegChecks r"
    using r unfolding list_all_iff by blast
  thus ?B'
  proof (elim disjE conjE)
    assume *: "is_InSet r"
    hence **: "is_Var (the_elem_term r)" "is_Fun (the_set_term r)"
              "is_Set (the_Fun (the_set_term r))" "args (the_set_term r) = []"
              "fst (the_Var (the_elem_term r)) = TAtom Value"
      using r adm_checks unfolding admissible_transaction_checks_def is_Fun_Set_def
      by fast+

    obtain ac rt rs where r': "r = ⟨ac: rt ∈ rs⟩" using * by (cases r) auto
    obtain x where x: "rt = Var x" "fst x = TAtom Value" using **(1,5) r' by auto
    obtain f S where fS: "rs = Fun f S" using **(2) r' by auto
    obtain s where s: "f = Set s" using **(3) fS r' by (cases f) auto
    hence S: "S = []" using **(4) fS r' by auto

    show ?B' using r' x fS s S fv_r1 fv_r2 by (cases ac) simp_all
  next
    assume *: "is_NegChecks r"
    hence **: "bvars_sstp r = []"
              "(the_eqs r = [] ∧ length (the_ins r) = 1) ∨
               (the_ins r = [] ∧ length (the_eqs r) = 1)"
      using r adm_checks unfolding admissible_transaction_checks_def by fast+
    show ?B' using **(2)
    proof (elim disjE conjE)
      assume ***: "the_eqs r = []" "length (the_ins r) = 1"
      then obtain t s where ts: "the_ins r = [(t,s)]" by (cases "the_ins r") auto
      hence "hd (the_ins r) = (t,s)" by simp
      hence ****: "is_Var (fst (t,s))" "is_Fun (snd (t,s))"
                  "is_Set (the_Fun (snd (t,s)))" "args (snd (t,s)) = []"
                  "fst (the_Var (fst (t,s))) = TAtom Value"
        using * ***(1) Set.bspec[OF adm_checks(2)[unfolded admissible_transaction_checks_def] r]
        unfolding is_Fun_Set_def by simp_all
      obtain x where x: "t = Var x" "fst x = TAtom Value" using ts ****(1,5) by (cases t) simp_all
      obtain f S where fS: "s = Fun f S" using ts ****(2) by (cases s) simp_all
      obtain ss where ss: "f = Set ss" using fS ****(3) by (cases f) simp_all
      have S: "S = []" using ts fS ss ****(4) by simp

      show ?B' using ts x fS ss S *** **(1) * fv_r1 fv_r2 by (cases r) auto
    next
      assume ***: "the_ins r = []" "length (the_eqs r) = 1"
      then obtain t s where "the_eqs r = [(t,s)]" by (cases "the_eqs r") auto
      thus ?B' using *** **(1) * by (cases r) auto
    qed
  qed (auto simp add: is_Equality_def the_check_def intro: poscheckvariant.exhaust)
qed

show "?C ⟹ ?C'"
```

```
  proof -
    assume r: ?C
    note adm_upds = admissible_transaction_is_wellformed_transaction(3)[OF T_adm]

    have *: "is_Update r" "is_Var (the_elem_term r)" "is_Fun (the_set_term r)"
            "is_Set (the_Fun (the_set_term r))" "args (the_set_term r) = []"
            "fst (the_Var (the_elem_term r)) = TAtom Value"
      using r adm_upds unfolding admissible_transaction_updates_def is_Fun_Set_def by fast+

    obtain t s where ts: "r = insert⟨t,s⟩ ∨ r = delete⟨t,s⟩" using *(1) by (cases r) auto
    obtain x where x: "t = Var x" "fst x = TAtom Value" using ts *(2,6) by (cases t) auto
    obtain f T where fT: "s = Fun f T" using ts *(3) by (cases s) auto
    obtain ss where ss: "f = Set ss" using ts fT *(4) by (cases f) fastforce+
    have T: "T = []" using ts fT *(5) ss by (cases T) auto

    show ?C'
      using ts x fT ss T by blast
  qed
qed

lemma protocol_transaction_vars_TAtom_typed:
  assumes T_adm: "admissible_transaction' T"
  shows "∀x ∈ vars_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    and "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    and "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
proof -
  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

  show "∀x ∈ vars_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    using admissible_transactionE(3)[OF T_adm] by fast
  thus "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    using vars_sst_is_fv_sst_bvars_sst by fast

  show "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
    using admissible_transactionE(2)[OF T_adm] by argo
qed

lemma protocol_transactions_no_pubconsts:
  assumes "admissible_transaction' T"
  shows "Fun (Val n) S ∉ subterms_set (trms_transaction T)"
    and "Fun (PubConst Value n) S ∉ subterms_set (trms_transaction T)"
using assms admissible_transactions_no_Value_consts(1,3) by (blast, blast)

lemma protocol_transactions_no_abss:
  assumes "admissible_transaction' T"
  shows "Fun (Abs n) S ∉ subterms_set (trms_transaction T)"
using assms admissible_transactions_no_Value_consts(2)
by fast

lemma admissible_transaction_strand_sem_fv_ineq:
  assumes T_adm: "admissible_transaction' T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
    and x: "x ∈ fv_transaction T - set (transaction_fresh T)"
    and y: "y ∈ fv_transaction T - set (transaction_fresh T)"
    and x_not_y: "x ≠ y"
  shows "ϑ x · ℐ ≠ ϑ y · ℐ"
proof -
  have "⟨Var x != Var y⟩ ∈ set (unlabel (transaction_checks T)) ∨
        ⟨Var y != Var x⟩ ∈ set (unlabel (transaction_checks T))"
    using x y x_not_y admissible_transactionE(8)[OF T_adm] by auto
  hence "⟨Var x != Var y⟩ ∈ set (unlabel (transaction_strand T)) ∨
         ⟨Var y != Var x⟩ ∈ set (unlabel (transaction_strand T))"
    unfolding transaction_strand_def unlabel_def by auto
```

```
  hence "⟨ϑ x != ϑ y⟩ ∈ set (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ∨
         ⟨ϑ y != ϑ x⟩ ∈ set (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ)))"
    using stateful_strand_step_subst_inI(8)[of _ _ "unlabel (transaction_strand T)" ϑ]
          subst_lsst_unlabel[of "transaction_strand T" ϑ]
          dual_lsst_unlabel_steps_iff(7)[of "[]" _ "[]"]
    by force
  then obtain B where B:
      "prefix (B@[⟨ϑ x != ϑ y⟩]) (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ∨
       prefix (B@[⟨ϑ y != ϑ x⟩]) (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ)))"
    unfolding prefix_def
    by (metis (no_types, opaque_lifting) append.assoc append_Cons append_Nil split_list)
  thus ?thesis
    using ℐ strand_sem_append_stateful[of IK DB _ _ ℐ]
          stateful_strand_sem_NegChecks_no_bvars(2)
    unfolding prefix_def
    by metis
qed

lemma admissible_transaction_sem_iff:
  fixes ϑ and T::"('fun,'atom,'sets,'lbl) prot_transaction"
  defines "A ≡ unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))"
  assumes T: "admissible_transaction' T"
    and I: " interpretation_subst I" "wf_trms (subst_range I)"
  shows "strand_sem_stateful M D A I ⟷ (
    (∀l ts. (l,receive⟨ts⟩) ∈ set (transaction_receive T) ⟶ (∀t ∈ set ts. M ⊢ t · ϑ · I)) ∧
    (∀l ac t s. (l,⟨ac: t ≐ s⟩) ∈ set (transaction_checks T) ⟶ t · ϑ · I = s · ϑ · I) ∧
    (∀l ac t s. (l,⟨ac: t ∈ s⟩) ∈ set (transaction_checks T) ⟶ (t · ϑ · I, s · ϑ · I) ∈ D) ∧
    (∀l t s. (l,⟨t != s⟩) ∈ set (transaction_checks T) ⟶ t · ϑ · I ≠ s · ϑ · I) ∧
    (∀l t s. (l,⟨t not in s⟩) ∈ set (transaction_checks T) ⟶ (t · ϑ · I, s · ϑ · I) ∉ D))"
    (is "?A ⟷ ?B")
proof -
  define P where "P ≡
    λX. λδ::('fun,'atom,'sets,'lbl) prot_subst. subst_domain δ = set X ∧ ground (subst_range δ)"
  define rm where "rm ≡ λX. λδ::('fun,'atom,'sets,'lbl) prot_subst. rm_vars (set X) δ"
  define chks where "chks ≡ transaction_checks T"
  define q1 where "q1 ≡ λt X δ. t · rm X ϑ · δ · I"
  define q2 where "q2 ≡ λt. t · ϑ · I"

  note 0 = admissible_transaction_is_wellformed_transaction[OF T]
  note 1 = wellformed_transaction_sem_iff[OF 0(1) I, of M D ϑ, unfolded A_def[symmetric]]
  note 2 = admissible_transactionE[OF T]

  have 3: "rm X ϑ = ϑ" when "X = []" for X using that unfolding rm_def by auto

  have 4: "P X δ ⟷ δ = Var" when "X = []" for X and δ
    using that unfolding P_def by auto

  have 5: "∃t s. ∀X⟨∨≠: F ∨∉: G⟩ = ⟨t != s⟩ ∨ ∀X⟨∨≠: F ∨∉: G⟩ = ⟨t not in s⟩"
    when X: "(l, ∀X⟨∨≠: F ∨∉: G⟩) ∈ set chks" for l X F G
  proof -
    have *: "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel (transaction_strand T))"
      using X transaction_strand_subsets(2)[of T] unlabel_in unfolding chks_def by fast
    hence **: "X = []" using 2(4) by auto

    note *** = transaction_notinset_checks_are_Value_vars(3,4)[OF T *]
               transaction_noteqs_checks_case[OF T *]

    show ?thesis
    proof (cases "G = []")
      case True thus ?thesis using ** ***(3) by blast
    next
      case False
      then obtain t s where g: "(t,s) ∈ set G" by (cases G) auto
```

```
      show ?thesis using ** ***(1,2)[OF g] by blast
    qed
  qed

  have 6: "q1 t X δ = q2 t" when "P X δ" "X = []" for X δ t
    using that 3 4 unfolding q1_def q2_def by simp

  let ?negcheck_sem = "λX F G. ∀δ. P X δ ⟶
                             (∃(t,s) ∈ set F. q1 t X δ ≠ q1 s X δ) ∨
                             (∃(t,s) ∈ set G. (q1 t X δ, q1 s X δ) ∉ D)"

  have "(∀l X F G. (l,∀X⟨∨≠: F ∨∉: G⟩) ∈ set chks ⟶ ?negcheck_sem X F G) ⟷
        ((∀l t s. (l,⟨t != s⟩) ∈ set chks ⟶ q2 t ≠ q2 s) ∧
         (∀l t s. (l,⟨t not in s⟩) ∈ set chks ⟶ (q2 t, q2 s) ∉ D))"
    (is "?A ⟷ ?B")
  proof
    have "q2 t ≠ q2 s" when t: "(l,⟨t != s⟩) ∈ set chks" ?A for l t s
    proof -
      have "?negcheck_sem [] [(t,s)] []" using t by blast
      thus ?thesis using 4[of "[]"] 6[of "[]"] by force
    qed
    moreover have "(q2 t, q2 s) ∉ D" when t: "(l,⟨t not in s⟩) ∈ set chks" ?A for l t s
    proof -
      have "?negcheck_sem [] [] [(t,s)]" using t by blast
      thus ?thesis using 4[of "[]"] 6[of "[]"] by force
    qed
    ultimately show "?A ⟹ ?B" by blast

    have "?negcheck_sem X F G"
      when t: "(l,∀X⟨∨≠: F ∨∉: G⟩) ∈ set chks" ?B for l X F G
    proof -
      obtain t s where ts: "(X = [] ∧ F = [(t,s)] ∧ G = []) ∨ (X = [] ∧ F = [] ∧ G = [(t,s)])"
        using 5[OF t(1)] by blast
      hence "(X = [] ∧ F = [(t,s)] ∧ G = [] ∧ q2 t ≠ q2 s) ∨
             (X = [] ∧ F = [] ∧ G = [(t,s)] ∧ (q2 t, q2 s) ∉ D)" using t by blast
      thus ?thesis using 4[of "[]"] 6[of "[]"] by fastforce
    qed
    thus "?B ⟹ ?A" by simp
  qed
  thus ?thesis using 1 unfolding rm_def chks_def P_def q1_def q2_def by simp
qed

lemma admissible_transaction_terms_wf_trms:
  assumes "admissible_transaction_terms T"
  shows "wf_trms (trms_transaction T)"
by (rule conjunct1[OF assms[unfolded admissible_transaction_terms_def wf_trms_code[symmetric]]])

lemma admissible_transactions_wf_trms:
  assumes "admissible_transaction' T"
  shows "wf_trms (trms_transaction T)"
proof -
  have "admissible_transaction_terms T" using assms[unfolded admissible_transaction'_def] by fast
  thus ?thesis by (metis admissible_transaction_terms_wf_trms)
qed

lemma admissible_transaction_no_Ana_Attack:
  assumes "admissible_transaction_terms T"
    and "t ∈ subterms_set (trms_transaction T)"
  shows "attack⟨n⟩ ∉ set (snd (Ana t))"
proof -
  obtain r where r: "r ∈ set (unlabel (transaction_strand T))" "t ∈ subterms_set (trms_sstp r)"
    using assms(2) by force
```

```
  obtain K M where t: "Ana t = (K, M)"
    by (metis surj_pair)

  show ?thesis
  proof
    assume n: "attack⟨n⟩ ∈ set (snd (Ana t))"
    hence "attack⟨n⟩ ∈ set M" using t by simp
    hence n': "attack⟨n⟩ ∈ subterms_set (trms_sstp r)"
      using Ana_subterm[OF t] r(2) subterms_subset by fast
    hence "∃f ∈ ⋃(funs_term ` trms_sstp r). is_Attack f"
      using funs_term_Fun_subterm' unfolding is_Attack_def by fast
    hence "is_Send r" "length (the_msgs r) = 1" "is_Fun (hd (the_msgs r))"
          "is_Attack (the_Fun (hd (the_msgs r)))" "args (hd (the_msgs r)) = []"
      using assms(1) r(1) unfolding admissible_transaction_terms_def is_Fun_Attack_def by metis+
    hence "t = attack⟨n⟩"
      using n' r(2) unfolding is_Send_def is_Attack_def by (cases "the_msgs r") auto
    thus False using n by fastforce
  qed
qed

lemma admissible_transaction_Value_vars:
  assumes T: "admissible_transaction' T"
    and x: "x ∈ fv_transaction T"
  shows "Γ_v x = TAtom Value"
proof -
  have "x ∈ vars_transaction T"
    using x vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_strand T)"]
    by blast
  thus "Γ_v x = TAtom Value"
    using admissible_transactionE(3)[OF T] by simp
qed

lemma admissible_transaction_occurs_checksE1:
  assumes T: "admissible_transaction_occurs_checks T"
    and x: "x ∈ fv_transaction T - set (transaction_fresh T)" "Γ_v x = TAtom Value"
  obtains l ts S where
    "transaction_receive T = (l,receive⟨ts⟩)#S" "occurs (Var x) ∈ set ts"
proof -
  let ?rcvs = "transaction_receive T"
  let ?frsh = "transaction_fresh T"
  let ?fvs = "fv_transaction T"

  have *: "?rcvs ≠ []" "is_Receive (hd (unlabel ?rcvs))"
          "∀x ∈ ?fvs - set ?frsh. Γ_v x = TAtom Value ⟶
            occurs (Var x) ∈ set (the_msgs (hd (unlabel ?rcvs)))"
    using T x unfolding admissible_transaction_occurs_checks_def Γ_v_TAtom''(2) by meson+

  obtain r S where S: "?rcvs = r#S"
    using *(1) by (cases ?rcvs) auto

  obtain l ts where r: "r = (l,receive⟨ts⟩)"
    by (metis *(1,2) S list.map_sel(1) list.sel(1) prod.collapse is_Receive_def unlabel_def)

  have 0: "occurs (Var x) ∈ set ts" using *(3) x S r by fastforce

  show ?thesis using that[unfolded S r, of l ts S] 0 by blast
qed

lemma admissible_transaction_occurs_checksE2:
  assumes T: "admissible_transaction_occurs_checks T"
    and x: "x ∈ set (transaction_fresh T)"
  obtains l ts S where
    "transaction_send T = (l,send⟨ts⟩)#S" "occurs (Var x) ∈ set ts"
```

```
proof -
  let ?snds = "transaction_send T"
  let ?frsh = "transaction_fresh T"
  let ?fvs = "fv_transaction T"
  define ts where "ts ≡ the_msgs (hd (unlabel ?snds))"

  let ?P = "∀x ∈ set ?frsh. occurs (Var x) ∈ set ts"

  have "?frsh ≠ []" using x by auto
  hence *: "?snds ≠ []" "is_Send (hd (unlabel ?snds))" "?P"
    using T x unfolding admissible_transaction_occurs_checks_def ts_def by meson+

  obtain r S where S: "?snds = r#S"
    using *(1) by (cases ?snds) auto

  obtain l where r: "r = (l,send⟨ts⟩)"
    by (metis *(1,2) S list.map_sel(1) list.sel(1) prod.collapse unlabel_def ts_def
              stateful_strand_step.collapse(1))

  have ts: "occurs (Var x) ∈ set ts"
    using x *(3) unfolding S by auto

  show ?thesis using that[unfolded S r, of l ts S] ts by blast
qed

lemma admissible_transaction_occurs_checksE3:
  assumes T: "admissible_transaction_occurs_checks T"
    and t: "OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t" "t ∈ set ts"
    and ts: "send⟨ts⟩ ∈ set (unlabel (transaction_send T))"
  obtains x where "t = occurs (Var x)" "x ∈ set (transaction_fresh T)"
proof -
  let ?P = "λt. ∃x ∈ set (transaction_fresh T). t = occurs (Var x)"

  have "?P t"
    when "t ∈ trms_lsst (transaction_send T)" "OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t"
    for t
    using assms that
    unfolding admissible_transaction_occurs_checks_def
              admissible_transaction_send_occurs_form_def
    by metis
  moreover have "t ∈ trms_lsst (transaction_send T)"
    using t(2) ts unfolding trms_sst_def by fastforce
  ultimately have "?P t" using t(1) by blast
  thus thesis using that by blast
qed

lemma admissible_transaction_occurs_checksE4:
  assumes T: "admissible_transaction_occurs_checks T"
    and ts: "send⟨ts⟩ ∈ set (unlabel (transaction_send T))"
    and t: "occurs t ∈ set ts"
  obtains x where "t = Var x" "x ∈ set (transaction_fresh T)"
using admissible_transaction_occurs_checksE3[OF T _ t ts] by auto

lemma admissible_transaction_occurs_checksE5:
  assumes T: "admissible_transaction_occurs_checks T"
  shows "Fun OccursSec [] ∉ trms_lsst (transaction_send T)"
proof -
  have "∃x ∈ set (transaction_fresh T). t = occurs (Var x)"
    when "t ∈ trms_lsst (transaction_send T)" "OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t"
    for t
    using assms that
    unfolding admissible_transaction_occurs_checks_def
              admissible_transaction_send_occurs_form_def
```

```
    by metis
  thus ?thesis by fastforce
qed

lemma admissible_transaction_occurs_checksE6:
  assumes T: "admissible_transaction_occurs_checks T"
    and t: "t ⊑set trmslsst (transaction_send T)"
  shows "Fun OccursSec [] ∉ set (snd (Ana t))" (is ?A)
    and "occurs k ∉ set (snd (Ana t))" (is ?B)
proof -
  obtain t' where t': "t' ∈ trmslsst (transaction_send T)" "t ⊑ t'" using t by blast
  have "?A ∧ ?B"
  proof (rule ccontr)
    assume *: "¬(?A ∧ ?B)"
    hence "OccursSec ∈ funs_term t' ∨ OccursFact ∈ funs_term t'"
      by (meson t'(2) Ana_subterm' funs_term_Fun_subterm' term.order.trans)
    then obtain x where x: "x ∈ set (transaction_fresh T)" "t' = occurs (Var x)"
      using t'(1) T
      unfolding admissible_transaction_occurs_checks_def
                admissible_transaction_send_occurs_form_def
      by metis
    have "t = occurs (Var x) ∨ t = Var x ∨ t = Fun OccursSec []" using x(2) t'(2) by auto
    thus False using * by fastforce
  qed
  thus ?A ?B by simp_all
qed

lemma has_initial_value_producing_transactionE:
  fixes P::"('fun,'atom,'sets,'lbl) prot"
  assumes P: "has_initial_value_producing_transaction P"
    and P_adm: "∀T ∈ set P. admissible_transaction' T"
  obtains T x s ts upds l l' where
    "Γv x = TAtom Value" "Var x ∈ set ts" "fvset (set ts) = {x}"
    "∀n. ¬(Fun (Val n) [] ⊑set set ts)" "T ∈ set P"
    "T = Transaction (λ(). []) [x] [] [] upds [(l, send⟨ts⟩)]"
    "upds = [] ∨ (upds = [(l', insert⟨Var x, ⟨s⟩s⟩)] ∧
      (∀T ∈ set P. ∀(l,a) ∈ set (transaction_strand T). ∀t.
        a ≠ select⟨t, ⟨s⟩s⟩ ∧ a ≠ ⟨t in ⟨s⟩s⟩ ∧ a ≠ ⟨t not in ⟨s⟩s⟩ ∧ a ≠ delete⟨t, ⟨s⟩s⟩)) ∨
     T = Transaction (λ(). []) [x] [] [] [] [(l, send⟨ts⟩)]"
proof -
  define f where "f ≡ λs.
      list_all (λT. list_all (λa. ((is_Delete a ∨ is_InSet a) ⟶ the_set_term a ≠ ⟨s⟩s) ∧
                                  (is_NegChecks a ⟶ list_all (λ(_,t). t ≠ ⟨s⟩s) (the_ins a)))
                             (unlabel (transaction_checks T@transaction_updates T)))
              P"

  obtain T where T0:
      "T ∈ set P"
      "length (transaction_fresh T) = 1" "transaction_receive T = []"
      "transaction_checks T = []" "length (transaction_send T) = 1"
      "let x = hd (transaction_fresh T); a = hd (transaction_send T); u = transaction_updates T
       in is_Send (snd a) ∧ Var x ∈ set (the_msgs (snd a)) ∧
          fvset (set (the_msgs (snd a))) = {x} ∧
          (u ≠ [] ⟶ (
            let b = hd u; c = snd b
            in tl u = [] ∧ is_Insert c ∧ the_elem_term c = Var x ∧
               is_Fun_Set (the_set_term c) ∧ f (the_Set (the_Fun (the_set_term c)))))"
    using P unfolding has_initial_value_producing_transaction_def Let_def list_ex_iff f_def by blast

  obtain x upds ts h s l l' where T1:
      "T = Transaction h [x] [] [] upds [(l, send⟨ts⟩)]"
      "Var x ∈ set ts" "fvset (set ts) = {x}"
      "upds = [] ∨ (upds = [(l', insert⟨Var x, ⟨s⟩s⟩)] ∧ f s)"
```

```
proof (cases T)
  case T: (Transaction A B C D E F)

  obtain x where B: "B = [x]" using T0(2) unfolding T by (cases B) auto
  have C: "C = []" using T0(3) unfolding T by simp
  have D: "D = []" using T0(4) unfolding T by simp
  obtain l a where F: "F = [(l,a)]" using T0(5) unfolding T by fastforce
  obtain ts where ts: "a = send⟨ts⟩" using T0(6) unfolding T F by (cases a) auto
  obtain k u where E: "E = [] ∨ E = [(k,u)]" using T0(6) unfolding T by (cases E) fastforce+
  have x: "Var x ∈ set ts" "fv_set (set ts) = {x}" using T0(6) unfolding T B F ts by auto

  from E show ?thesis
  proof
    assume E': "E = [(k,u)]"
    obtain t t' where u: "u = insert⟨t,t'⟩" using T0(6) unfolding T E' by (cases u) auto
    have t: "t = Var x" using T0(6) unfolding T B E' u Let_def by simp
    obtain s where t': "t' = ⟨s⟩_s" and s: "f s" using T0(6) unfolding T B E' u Let_def by auto
    show ?thesis using that[OF T[unfolded B C D F ts E' u t t'] x] s by blast
  qed (use that[OF T[unfolded B C D F ts] x] in blast)
qed

note T_adm = bspec[OF P_adm T0(1)]

have "x ∈ set (transaction_fresh T)" using T1(1) by fastforce
hence x: "Γ_v x = TAtom Value" using admissible_transactionE(2)[OF T_adm] by fast

have "set ts ⊆ trms_transaction T" unfolding T1(1) trms_transaction_unfold by simp
hence ts: "∀n. ¬(Fun (Val n) [] ⊑_set set ts)"
  using admissible_transactions_no_Value_consts[OF T_adm] by blast

have "a ≠ select⟨t, ⟨s⟩_s⟩ ∧ a ≠ ⟨t in ⟨s⟩_s⟩ ∧ a ≠ ⟨t not in ⟨s⟩_s⟩ ∧ a ≠ delete⟨t, ⟨s⟩_s⟩"
  when upds: "upds = [(k, insert⟨Var x,⟨s⟩_s⟩)]"
    and T': "T' ∈ set P" and la: "(l,a) ∈ set (transaction_strand T')"
  for T' l k a t
proof -
  note T'_wf = admissible_transaction_is_wellformed_transaction(1)[OF bspec[OF P_adm T']]

  have "a ∈ set (unlabel (transaction_checks T'@transaction_updates T'))"
    when a': "is_Check_or_Assignment a ∨ is_Update a"
    using that wellformed_transaction_strand_unlabel_memberD[OF T'_wf unlabel_in[OF la]]
    by (cases a) auto
  note 0 = this T1(4) T'

  note 1 = upds f_def list_all_iff

  show ?thesis
  proof (cases a)
    case (Delete t' s') thus ?thesis using 0 unfolding 1 by fastforce
  next
    case (InSet ac t' s') thus ?thesis using 0 unfolding 1 by fastforce
  next
    case (NegChecks X F G) thus ?thesis using 0 unfolding 1 by fastforce
  qed auto
qed
hence s: "∀T ∈ set P. ∀(l,a) ∈ set (transaction_strand T). ∀t.
     a ≠ select⟨t, ⟨s⟩_s⟩ ∧ a ≠ ⟨t in ⟨s⟩_s⟩ ∧ a ≠ ⟨t not in ⟨s⟩_s⟩ ∧ a ≠ delete⟨t, ⟨s⟩_s⟩"
  when upds: "upds = [(k, insert⟨Var x,⟨s⟩_s⟩)]" for k
  using upds by force

have h: "h = (λ(). [])"
proof -
  have "transaction_decl T = h" using T1(1) by fastforce
  hence "h a = []" for a using admissible_transactionE(1)[OF T_adm] by simp
```

```
    thus ?thesis using ext[of h "λ(). []"] by (metis case_unit_Unity)
  qed

  show ?thesis using that[OF x T1(2,3) ts T0(1)] T1(1,4) s unfolding h by auto
qed

lemma has_initial_value_producing_transaction_update_send_ex_filter:
  fixes P::"('a,'b,'c,'d) prot"
  defines "f ≡ λT. transaction_fresh T = [] ⟶
                    list_ex (λa. is_Update (snd a) ∨ is_Send (snd a)) (transaction_strand T)"
  assumes P: "has_initial_value_producing_transaction P"
  shows "has_initial_value_producing_transaction (filter f P)"
proof -
  define g where "g ≡ λP::('a,'b,'c,'d) prot. λs.
      list_all (λT. list_all (λa. ((is_Delete a ∨ is_InSet a) ⟶ the_set_term a ≠ ⟨s⟩s) ∧
                                  (is_NegChecks a ⟶ list_all (λ(_,t). t ≠ ⟨s⟩s) (the_ins a)))
                             (unlabel (transaction_checks T@transaction_updates T)))
              P"

  let ?Q = "λP T.
     let x = hd (transaction_fresh T); a = hd (transaction_send T); u = transaction_updates T
     in is_Send (snd a) ∧ Var x ∈ set (the_msgs (snd a)) ∧
        fv_set (set (the_msgs (snd a))) = {x} ∧
        (u ≠ [] ⟶ (
          let b = hd u; c = snd b
          in tl u = [] ∧ is_Insert c ∧ the_elem_term c = Var x ∧
             is_Fun_Set (the_set_term c) ∧ g P (the_Set (the_Fun (the_set_term c)))))"

  have "set (filter f P) ⊆ set P" by simp
  hence "list_all h P ⟹ list_all h (filter f P)" for h unfolding list_all_iff by blast
  hence g_f_subset: "g P s ⟹ g (filter f P) s" for s unfolding g_def by blast

  obtain T where T:
    "T ∈ set P" "length (transaction_fresh T) = 1" "transaction_receive T = []"
    "transaction_checks T = []" "length (transaction_send T) = 1" "?Q P T"
    using P unfolding has_initial_value_producing_transaction_def Let_def list_ex_iff g_def by blast

  obtain x where x: "transaction_fresh T = [x]" using T(2) by blast
  obtain a where a: "transaction_send T = [a]" using T(5) by blast
  obtain l b where b: "a = (l,b)" by (cases a) auto
  obtain ts where ts: "b = send⟨ts⟩" using T(6) unfolding Let_def a b by (cases b) auto

  have "T ∈ set (filter f P)" using T(1) x a unfolding b ts f_def by auto
  moreover have "?Q (filter f P) T" using T(6) g_f_subset by meson
  ultimately show ?thesis
    using T(2-5)
    unfolding has_initial_value_producing_transaction_def Let_def list_ex_iff g_def
    by blast
qed
```

3.3.7 Lemmata: Renaming, Declaration, and Fresh Substitutions

```
lemma transaction_decl_subst_empty_inv:
  assumes "transaction_decl_subst Var T"
  shows "transaction_decl T () = []"
using assms unfolding transaction_decl_subst_def subst_domain_Var by blast

lemma transaction_decl_subst_domain:
  fixes ξ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst ξ T"
  shows "subst_domain ξ = fst ` set (transaction_decl T ())"
using assms unfolding transaction_decl_subst_def by argo
```

```
lemma transaction_decl_subst_grounds_domain:
  fixes ξ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst ξ T"
    and "x ∈ fst ` set (transaction_decl T ())"
  shows "fv (ξ x) = {}"
proof -
  obtain c where "ξ x = Fun c []"
    using assms unfolding transaction_decl_subst_def by force
  thus ?thesis by simp
qed

lemma transaction_decl_subst_range_vars_empty:
  fixes ξ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst ξ T"
  shows "range_vars ξ = {}"
using assms unfolding transaction_decl_subst_def range_vars_def by auto

lemma transaction_decl_subst_wt:
  fixes ξ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst ξ T"
  shows "wt_subst ξ"
using assms unfolding transaction_decl_subst_def by blast

lemma transaction_decl_subst_is_wf_trm:
  fixes ξ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst ξ P"
  shows "wf_trm (ξ v)"
proof (cases "v ∈ subst_domain ξ")
  case True thus ?thesis using assms unfolding transaction_decl_subst_def by fastforce
qed auto

lemma transaction_decl_subst_range_wf_trms:
  fixes ξ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst ξ P"
  shows "wf_trms (subst_range ξ)"
by (metis transaction_decl_subst_is_wf_trm[OF assms] wf_trm_subst_range_iff)

lemma transaction_renaming_subst_is_renaming:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes α: "transaction_renaming_subst α P A"
  shows "∃m. ∀τ n. α (τ,n) = Var (τ,n+Suc m)" (is ?A)
    and "∃y. α x = Var y" (is ?B)
    and "α x ≠ Var x" (is ?C)
    and "subst_domain α = UNIV" (is ?D)
    and "subst_range α ⊆ range Var" (is ?E)
    and "fv (t · α) ⊆ range_vars α" (is ?F)
proof -
  show 0: ?A using α unfolding transaction_renaming_subst_def var_rename_def by force
  show ?B using α unfolding transaction_renaming_subst_def var_rename_def by blast
  show ?C using 0 by (cases x) auto
  show 1: ?D using 0 by fastforce
  show ?E using 0 by auto
  show ?F by (induct t) (auto simp add: 1 subst_dom_vars_in_subst subst_fv_imgI)
qed

lemma transaction_renaming_subst_is_injective:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "inj α"
proof (intro injI)
  fix x y::"('fun,'atom,'sets,'lbl) prot_var"
  obtain τx nx where x: "x = (τx,nx)" by (metis surj_pair)
  obtain τy ny where y: "y = (τy,ny)" by (metis surj_pair)
```

```
  obtain m where m: "∀τ. ∀n. α (τ, n) = Var (τ, n + Suc m)"
    using transaction_renaming_subst_is_renaming(1)[OF assms] by blast

  assume "α x = α y"
  hence "τx = τy" "nx = ny" using x y m by simp_all
  thus "x = y" using x y by argo
qed

lemma transaction_renaming_subst_vars_disj:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst α P (vars_lsst A)"
  shows "fv_set (α ` (⋃(vars_transaction ` set P))) ∩ (⋃(vars_transaction ` set P)) = {}" (is ?A)
    and "fv_set (α ` vars_lsst A) ∩ vars_lsst A = {}" (is ?B)
    and "T ∈ set P ⟹ vars_transaction T ∩ range_vars α = {}" (is "T ∈ set P ⟹ ?C1")
    and "T ∈ set P ⟹ bvars_transaction T ∩ range_vars α = {}" (is "T ∈ set P ⟹ ?C2")
    and "T ∈ set P ⟹ fv_transaction T ∩ range_vars α = {}" (is "T ∈ set P ⟹ ?C3")
    and "vars_lsst A ∩ range_vars α = {}" (is ?D1)
    and "bvars_lsst A ∩ range_vars α = {}" (is ?D2)
    and "fv_lsst A ∩ range_vars α = {}" (is ?D3)
proof -
  define X where "X ≡ ⋃(vars_transaction ` set P) ∪ vars_lsst A"

  have 1: "finite X" by (simp add: X_def)

  obtain n where n: "n ≥ max_var_set X" "α = var_rename n"
    using assms unfolding transaction_renaming_subst_def X_def by force
  hence 2: "∀x ∈ X. snd x < Suc n"
    using less_Suc_max_var_set[OF _ 1] unfolding var_rename_def by fastforce

  have 3: "x ∉ fv_set (α ` X)" "fv (α x) ∩ X = {}" "x ∉ range_vars α" when x: "x ∈ X" for x
    using 2 x n unfolding var_rename_def by force+

  show ?A ?B using 3(1,2) unfolding X_def by auto

  show ?C1 when T: "T ∈ set P" using T 3(3) unfolding X_def by blast
  thus ?C2 ?C3 when T: "T ∈ set P"
    using T by (simp_all add: disjoint_iff_not_equal vars_sst_is_fv_sst_bvars_sst)

  show ?D1 using 3(3) unfolding X_def by auto
  thus ?D2 ?D3 by (simp_all add: disjoint_iff_not_equal vars_sst_is_fv_sst_bvars_sst)
qed

lemma transaction_renaming_subst_wt:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst α P X"
  shows "wt_subst α"
proof -
  { fix x::"('fun,'atom,'sets,'lbl) prot_var"
    obtain τ n where x: "x = (τ,n)" by force
    then obtain m where m: "α x = Var (τ,m)"
      using assms transaction_renaming_subst_is_renaming(1) by force
    hence "Γ (α x) = Γ_v x" using x by (simp add: Γ_v_def)
  } thus ?thesis by (simp add: wt_subst_def)
qed

lemma transaction_renaming_subst_is_wf_trm:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst α P X"
  shows "wf_trm (α v)"
proof -
  obtain τ n where "v = (τ, n)" by force
  then obtain m where "α v = Var (τ, n + Suc m)"
```

```
    using transaction_renaming_subst_is_renaming(1)[OF assms]
    by force
  thus ?thesis by (metis wf_trm_Var)
qed

lemma transaction_renaming_subst_range_wf_trms:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst α P X"
  shows "wf_trms (subst_range α)"
by (metis transaction_renaming_subst_is_wf_trm[OF assms] wf_trm_subst_range_iff)

lemma transaction_renaming_subst_range_notin_vars:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst α P (vars_lsst A)"
  shows "∃y. α x = Var y ∧ y ∉ ⋃(vars_transaction ` set P) ∪ vars_lsst A"
proof -
  obtain τ n where x: "x = (τ,n)" by (metis surj_pair)

  define y where "y ≡ λm. (τ,n+Suc m)"

  have "∃m ≥ max_var_set (⋃(vars_transaction ` set P) ∪ vars_lsst A). α x = Var (y m)"
    using assms x by (auto simp add: y_def transaction_renaming_subst_def var_rename_def)
  moreover have "finite (⋃(vars_transaction ` set P) ∪ vars_lsst A)" by auto
  ultimately show ?thesis using x unfolding y_def by force
qed

lemma transaction_renaming_subst_var_obtain:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes α: "transaction_renaming_subst α P X"
  shows "x ∈ fv_sst (S ·_sst α) ⟹ ∃y. α y = Var x" (is "?A1 ⟹ ?B1")
    and "x ∈ fv (t · α) ⟹ ∃y ∈ fv t. α y = Var x" (is "?A2 ⟹ ?B2")
proof -
  assume x: ?A1
  obtain y where y: "y ∈ fv_sst S" "x ∈ fv (α y)" using fv_sst_subst_obtain_var[OF x] by force
  thus ?B1 using transaction_renaming_subst_is_renaming(2)[OF α, of y] by fastforce
next
  assume x: ?A2
  obtain y where y: "y ∈ fv t" "x ∈ fv (α y)" using fv_subst_obtain_var[OF x] by force
  thus ?B2 using transaction_renaming_subst_is_renaming(2)[OF α, of y] by fastforce
qed

lemma transaction_renaming_subst_set_eq:
  assumes "set P1 = set P2"
  shows "transaction_renaming_subst α P1 X = transaction_renaming_subst α P2 X" (is "?A = ?B")
using assms unfolding transaction_renaming_subst_def by presburger

lemma transaction_renaming_subst_vars_transaction_neq:
  assumes T: "T ∈ set P"
    and α: "transaction_renaming_subst α P vars"
    and vars:"finite vars"
    and x: "x ∈ vars_transaction T"
  shows "α y ≠ Var x"
proof -
  have "∃n. α = var_rename n ∧ n ≥ max_var_set (⋃(vars_transaction ` set P))"
    using T α vars x unfolding transaction_renaming_subst_def by auto
  then obtain n where n_p: "α = var_rename n" "n ≥ max_var_set (⋃(vars_transaction ` set P))"
    by blast
  moreover
  have "⋃(vars_transaction ` set P) ⊇ vars_transaction T"
    using T by blast
  ultimately
  have n_gt: "n ≥ max_var_set (vars_transaction T)"
    by auto
```

```
  obtain a b where ab: "x = (a,b)"
    by (cases x) auto
  obtain c d where cd: "y = (c,d)"
    by (cases y) auto

  have nb: "n ≥ b"
    using n_gt x ab
    by auto

  have "α y = α (c, d)"
    using cd by auto
  moreover
  have "... = Var (c, Suc (d + n))"
    unfolding n_p(1) unfolding var_rename_def by simp
  moreover
  have "... ≠ Var x"
    using nb ab by auto
  ultimately
  show ?thesis
    by auto
qed

lemma transaction_renaming_subst_fv_disj:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst α P (vars_lsst A)"
  shows "fv_set (α ` fv_lsst A) ∩ fv_lsst  A = {}"
proof -
  have "fv_set (α ` vars_lsst A) ∩ vars_lsst  A = {}"
    using assms transaction_renaming_subst_vars_disj(2) by blast
  moreover
  have "fv_lsst A ⊆ vars_lsst A"
    by (simp add: vars_sst_is_fv_sst_bvars_sst)
  ultimately
  show ?thesis
    by auto
qed

lemma transaction_fresh_subst_is_wf_trm:
  fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_fresh_subst σ T X"
  shows "wf_trm (σ v)"
proof (cases "v ∈ subst_domain σ")
  case True
  then obtain c where "σ v = Fun c []" "arity c = 0"
    using assms unfolding transaction_fresh_subst_def
    by force
  thus ?thesis by auto
qed auto

lemma transaction_fresh_subst_wt:
  fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_fresh_subst σ T X"
  shows "wt_subst σ"
using assms unfolding transaction_fresh_subst_def by blast

lemma transaction_fresh_subst_domain:
  fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_fresh_subst σ T X"
  shows "subst_domain σ = set (transaction_fresh T)"
using assms unfolding transaction_fresh_subst_def by fast

lemma transaction_fresh_subst_range_wf_trms:
  fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
```

```
  assumes "transaction_fresh_subst σ T X"
  shows "wf_trms (subst_range σ)"
by (metis transaction_fresh_subst_is_wf_trm[OF assms] wf_trm_subst_range_iff)

lemma transaction_fresh_subst_range_fresh:
  fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_fresh_subst σ T M"
  shows "∀t ∈ subst_range σ. t ∉ subterms_set M"
    and "∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst (transaction_strand T))"
using assms unfolding transaction_fresh_subst_def by meson+

lemma transaction_fresh_subst_sends_to_val:
  fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes σ: "transaction_fresh_subst σ T X"
    and y: "y ∈ set (transaction_fresh T)" "Γv y = TAtom Value"
  obtains n where "σ y = Fun (Val n) []" "Fun (Val n) [] ∈ subst_range σ"
proof -
  have "σ y ∈ subst_range σ" using assms unfolding transaction_fresh_subst_def by simp

  obtain c where c: "σ y = Fun c []" "¬public c" "arity c = 0"
    using σ y(1) unfolding transaction_fresh_subst_def by fastforce

  have "Γ (σ y) = TAtom Value"
    using σ y(2) Γv_TAtom''(2)[of y] wt_subst_trm''[of σ "Var y"]
    unfolding transaction_fresh_subst_def by simp
  then obtain n where "c = Val n"
    using c by (cases c) (auto split: option.splits)
  thus ?thesis
    using c that unfolding transaction_fresh_subst_def
    by fastforce
qed

lemma transaction_fresh_subst_sends_to_val':
  fixes σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_fresh_subst σ T X"
    and "y ∈ set (transaction_fresh T)" "Γv y = TAtom Value"
  obtains n where "(σ ∘s α) y · I = Fun (Val n) []" "Fun (Val n) [] ∈ subst_range σ"
proof -
  obtain n where "σ y = Fun (Val n) []" "Fun (Val n) [] ∈ subst_range σ"
    using transaction_fresh_subst_sends_to_val[OF assms] by force
  thus ?thesis using that by (fastforce simp add: subst_compose_def)
qed

lemma transaction_fresh_subst_grounds_domain:
  fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_fresh_subst σ T X"
    and "y ∈ set (transaction_fresh T)"
  shows "fv (σ y) = {}"
proof -
  obtain c where "σ y = Fun c []"
    using assms unfolding transaction_fresh_subst_def by force
  thus ?thesis by simp
qed

lemma transaction_fresh_subst_range_vars_empty:
  fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_fresh_subst σ T X"
  shows "range_vars σ = {}"
proof -
  have "fv t = {}" when "t ∈ subst_range σ" for t
    using assms that unfolding transaction_fresh_subst_def by fastforce
  thus ?thesis unfolding range_vars_def by blast
qed
```

```
lemma transaction_decl_fresh_renaming_substs_range:
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T M"
    and α: "transaction_renaming_subst α P X"
  shows "x ∈ fst ` set (transaction_decl T ()) ⟹
          ∃c. (ξ ∘s σ ∘s α) x = Fun c [] ∧ arity c = 0"
    and "x ∉ fst ` set (transaction_decl T ()) ⟹
          x ∈ set (transaction_fresh T) ⟹
            ∃c. (ξ ∘s σ ∘s α) x = Fun c [] ∧ ¬public c ∧ arity c = 0"
    and "x ∉ fst ` set (transaction_decl T ()) ⟹
          x ∈ set (transaction_fresh T) ⟹
          fst x = TAtom Value ⟹
            ∃n. (ξ ∘s σ ∘s α) x = Fun (Val n) []"
    and "x ∉ fst ` set (transaction_decl T ()) ⟹
          x ∉ set (transaction_fresh T) ⟹
            ∃y. (ξ ∘s σ ∘s α) x = Var y"
proof -
  assume "x ∈ fst ` set (transaction_decl T ())"
  then obtain c where c: "ξ x = Fun c []" "arity c = 0"
    using ξ unfolding transaction_decl_subst_def by fastforce
  thus "∃c. (ξ ∘s σ ∘s α) x = Fun c [] ∧ arity c = 0"
    using subst_compose[of "ξ ∘s σ" α x] subst_compose[of ξ σ x] by simp
next
  assume x: "x ∉ fst ` set (transaction_decl T ())"
            "x ∈ set (transaction_fresh T)"

  have *: "(ξ ∘s σ) x = σ x"
    using x(1) ξ unfolding transaction_decl_subst_def
    by (metis (no_types, opaque_lifting) subst_comp_notin_dom_eq)
  then obtain c where c: "(ξ ∘s σ) x = Fun c []" "¬public c" "arity c = 0"
    using σ x(2) unfolding transaction_fresh_subst_def by fastforce
  thus "∃c. (ξ ∘s σ ∘s α) x = Fun c [] ∧ ¬public c ∧ arity c = 0"
    using subst_compose[of "ξ ∘s σ" α x] subst_compose[of ξ σ x] by simp

  assume "fst x = TAtom Value"
  hence "Γ ((ξ ∘s σ) x) = TAtom Value"
    using * σ Γv_TAtom''(2)[of x] wt_subst_trm''[of σ "Var x"]
    unfolding transaction_fresh_subst_def by simp
  then obtain n where "c = Val n"
    using c by (cases c) (auto split: option.splits)
  thus "∃n. (ξ ∘sσ ∘s α) x = Fun (Val n) []"
    using c subst_compose[of "ξ ∘s σ" α x] subst_compose[of ξ σ x] by simp
next
  assume "x ∉ fst ` set (transaction_decl T ())"
         "x ∉ set (transaction_fresh T)"
  hence "(ξ ∘s σ) x = Var x"
    using ξ σ
    unfolding transaction_decl_subst_def transaction_fresh_subst_def
    by (metis (no_types, opaque_lifting) subst_comp_notin_dom_eq subst_domI)
  thus "∃y. (ξ ∘s σ ∘s α) x = Var y"
    using transaction_renaming_subst_is_renaming(1)[OF α]
          subst_compose[of "ξ ∘s σ" α x] subst_compose[of ξ σ x]
    by (cases x) force
qed

lemma transaction_decl_fresh_renaming_substs_range':
  fixes σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T M"
    and α: "transaction_renaming_subst α P X"
    and t: "t ∈ subst_range (ξ ∘s σ ∘s α)"
```

```
  shows "(∃c. t = Fun c [] ∧ arity c = 0) ∨ (∃x. t = Var x)"
    and "ξ = Var ⟹ (∃c. t = Fun c [] ∧ ¬public c ∧ arity c = 0) ∨ (∃x. t = Var x)"
    and "ξ = Var ⟹ ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value ⟹
                   (∃n. t = Fun (Val n) []) ∨ (∃x. t = Var x)"
    and "ξ = Var ⟹ is_Fun t ⟹ t ∈ subst_range σ"
proof -
  obtain x where x: "x ∈ subst_domain (ξ ∘s σ ∘s α)" "(ξ ∘s σ ∘s α) x = t"
    using t by auto

  note 0 = x transaction_decl_fresh_renaming_substs_range[OF ξ σ α, of x]

  show "(∃c. t = Fun c [] ∧ arity c = 0) ∨ (∃x. t = Var x)"
    using 0 unfolding Γv_TAtom'' by auto

  assume 1: "ξ = Var"

  note 2 = transaction_decl_subst_empty_inv[OF ξ[unfolded 1]]

  show 3: "(∃c. t = Fun c [] ∧ ¬public c ∧ arity c = 0) ∨ (∃x. t = Var x)"
    using 0 2 unfolding Γv_TAtom'' by auto

  show "(∃n. t = Fun (Val n) []) ∨ (∃x. t = Var x)"
    when "∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
    using 0 2 that unfolding Γv_TAtom'' by auto

  show "t ∈ subst_range σ" when t': "is_Fun t"
  proof -
    obtain x where x: "(σ ∘s α) x = t" using t 1 by auto

    show ?thesis
    proof (cases "x ∈ subst_domain σ")
      case True thus ?thesis
        by (metis subst_dom_vars_in_subst subst_ground_ident_compose(1) subst_imgI x
                  transaction_fresh_subst_grounds_domain[OF σ]
                  transaction_fresh_subst_domain[OF σ])
    next
      case False thus ?thesis
        by (metis (no_types, lifting) subst_compose_def subst_domI term.disc(1) that
                  transaction_renaming_subst_is_renaming(5)[OF α] var_renaming_is_Fun_iff x)
    qed
  qed
qed

lemma transaction_decl_fresh_renaming_substs_range'':
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trms_lsst 𝒜)"
    and α: "transaction_renaming_subst α P (vars_lsst 𝒜)"
    and y: "y ∈ fv ((ξ ∘s σ ∘s α) x)"
  shows "ξ x = Var x"
    and "σ x = Var x"
    and "α x = Var y"
    and "(ξ ∘s σ ∘s α) x = Var y"
proof -
  have "∃z. z ∈ fv (ξ x)" by (metis y subst_compose_fv')
  hence "x ∉ subst_domain ξ"
    using y transaction_decl_subst_domain[OF ξ]
          transaction_decl_subst_grounds_domain[OF ξ, of x]
    by blast
  thus 0: "ξ x = Var x" by blast
  hence "y ∈ fv ((σ ∘s α) x)" using y by (simp add: subst_compose)
  hence "∃z. z ∈ fv (σ x)" by (metis subst_compose_fv')
  hence "x ∉ subst_domain σ"
```

```
    using y transaction_fresh_subst_domain[OF σ]
          transaction_fresh_subst_grounds_domain[OF σ, of x]
    by blast
  thus 1: "σ x = Var x" by blast

  show "α x = Var y" "(ξ ∘s σ ∘s α) x = Var y"
    using 0 1 y transaction_renaming_subst_is_renaming(2)[OF α, of x]
    unfolding subst_compose_def by (fastforce,fastforce)
qed

lemma transaction_decl_fresh_renaming_substs_vars_subset:
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trmslsst 𝒜)"
    and α: "transaction_renaming_subst α P (varslsst 𝒜)"
  shows "⋃(fv_transaction ` set P) ⊆ subst_domain (ξ ∘s σ ∘s α)" (is ?A)
    and "fvlsst 𝒜 ⊆ subst_domain (ξ ∘s σ ∘s α)" (is ?B)
    and "T' ∈ set P ⟹ fv_transaction T' ⊆ subst_domain (ξ ∘s σ ∘s α)" (is "T' ∈ set P ⟹ ?C")
    and "T' ∈ set P ⟹ fvlsst (transaction_strand T' ·lsst (ξ ∘s σ ∘s α)) ⊆ range_vars (ξ ∘s σ ∘s α)"
      (is "T' ∈ set P ⟹ ?D")
proof -
  have *: "x ∈ subst_domain (ξ ∘s σ ∘s α)" for x
  proof (cases "x ∈ subst_domain ξ")
    case True thus ?thesis
      using transaction_decl_subst_domain[OF ξ] transaction_decl_subst_grounds_domain[OF ξ]
      by (simp add: subst_domI subst_dom_vars_in_subst subst_ground_ident_compose(1))
  next
    case False
    hence ξ_x_eq: "(ξ ∘s σ ∘s α) x = (σ ∘s α) x" by (auto simp add: subst_compose)

    show ?thesis
    proof (cases "x ∈ subst_domain σ")
      case True
      hence "x ∉ {x. ∃y. σ x = Var y ∧ α y = Var x}"
        using transaction_fresh_subst_domain[OF σ]
              transaction_fresh_subst_grounds_domain[OF σ, of x]
        by auto
      hence "x ∈ subst_domain (σ ∘s α)" using subst_domain_subst_compose[of σ α] by blast
      thus ?thesis using ξ_x_eq subst_dom_vars_in_subst by fastforce
    next
      case False
      hence "(σ ∘s α) x = α x" unfolding subst_compose_def by fastforce
      moreover have "α x ≠ Var x"
        using transaction_renaming_subst_is_renaming(1)[OF α] by (cases x) auto
      ultimately show ?thesis using ξ_x_eq by fastforce
    qed
  qed

  show ?A ?B using * by blast+

  show ?C when T: "T' ∈ set P" using T * by blast
  hence "fvsst (unlabel (transaction_strand T') ·sst ξ ∘s σ ∘s α) ⊆ range_vars (ξ ∘s σ ∘s α)"
    when T: "T' ∈ set P"
    using T fvsst_subst_subset_range_vars_if_subset_domain by blast
  thus ?D when T: "T' ∈ set P" by (metis T unlabel_subst)
qed

lemma transaction_decl_fresh_renaming_substs_vars_disj:
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trmslsst 𝒜)"
    and α: "transaction_renaming_subst α P (varslsst 𝒜)"
  shows "fvset ((ξ ∘s σ ∘s α) ` (⋃(vars_transaction ` set P))) ∩ (⋃(vars_transaction ` set P)) = {}"
```

```
      (is ?A)
    and "x ∈ ⋃(vars_transaction ` set P) ⟹ fv ((ξ ∘s σ ∘s α) x) ∩ (⋃(vars_transaction ` set P)) =
{}"
      (is "?B' ⟹ ?B")
    and "T' ∈ set P ⟹ vars_transaction T' ∩ range_vars (ξ ∘s σ ∘s α) = {}" (is "T' ∈ set P ⟹
?C1")
    and "T' ∈ set P ⟹ bvars_transaction T' ∩ range_vars (ξ ∘s σ ∘s α) = {}" (is "T' ∈ set P ⟹
?C2")
    and "T' ∈ set P ⟹ fv_transaction T' ∩ range_vars (ξ ∘s σ ∘s α) = {}" (is "T' ∈ set P ⟹ ?C3")
    and "varslsst 𝒜 ∩ range_vars (ξ ∘s σ ∘s α) = {}" (is ?D1)
    and "bvarslsst 𝒜 ∩ range_vars (ξ ∘s σ ∘s α) = {}" (is ?D2)
    and "fvlsst 𝒜 ∩ range_vars (ξ ∘s σ ∘s α) = {}" (is ?D3)
    and "range_vars ξ = {}" (is ?E1)
    and "range_vars σ = {}" (is ?E2)
    and "range_vars (ξ ∘s σ ∘s α) ⊆ range_vars α" (is ?E3)
proof -
  note 0 = transaction_renaming_subst_vars_disj[OF α]
  define ϑ where "ϑ = ξ ∘s σ ∘s α"
  show ?A
  proof (cases "fvset ((ξ ∘s σ ∘s α) ` (⋃(vars_transaction ` set P))) = {}")
    case False
    hence "∀x ∈ (⋃(vars_transaction ` set P)). (ξ ∘s σ ∘s α) x = α x ∨ fv ((ξ ∘s σ ∘s α) x) = {}"
      using transaction_decl_fresh_renaming_substs_range''[OF ξ σ α] by auto
    thus ?thesis using 0(1) unfolding ϑ_def[symmetric] by force
  qed blast
  thus "?B' ⟹ ?B" by auto

  show ?E1 ?E2
    using transaction_fresh_subst_grounds_domain[OF σ]
          transaction_decl_subst_grounds_domain[OF ξ]
    unfolding transaction_fresh_subst_domain[OF σ, symmetric]
              transaction_decl_subst_domain[OF ξ, symmetric]
    by (fastforce, fastforce)
  thus 1: ?E3
    using range_vars_subst_compose_subset[of ξ σ]
          range_vars_subst_compose_subset[of "ξ ∘s σ" α]
    by blast

  show ?C1 ?C2 ?C3 when T: "T' ∈ set P" using T 1 0(3,4,5)[of T'] by blast+

  show ?D1 ?D2 ?D3 using 1 0(6,7,8) by blast+
qed

lemma transaction_decl_fresh_renaming_substs_trms:
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trmslsst 𝒜)"
    and α: "transaction_renaming_subst α P (varslsst 𝒜)"
    and "bvarslsst S ∩ subst_domain ξ = {}"
    and "bvarslsst S ∩ subst_domain σ = {}"
    and "bvarslsst S ∩ subst_domain α = {}"
  shows "subtermsset (trmslsst (S ·lsst (ξ ∘s σ ∘s α))) = subtermsset (trmslsst S) ·set (ξ ∘s σ ∘s α)"
proof -
  have 1: "∀x ∈ fvset (trmslsst S). (∃f. (ξ ∘s σ ∘s α) x = Fun f []) ∨ (∃y. (ξ ∘s σ ∘s α) x = Var y)"
    using transaction_decl_fresh_renaming_substs_range'[OF ξ σ α] by blast

  have 2: "bvarslsst S ∩ subst_domain (ξ ∘s σ ∘s α) = {}"
    using assms(4-6) subst_domain_compose[of ξ σ] subst_domain_compose[of "ξ ∘s σ" α] by blast

  show ?thesis using subterms_subst_lsst[OF 1 2] by simp
qed

lemma transaction_decl_fresh_renaming_substs_wt:
```

```
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst ξ T" "transaction_fresh_subst σ T M"
          "transaction_renaming_subst α P X"
  shows "wt_{subst} (ξ ∘_s σ ∘_s α)"
using transaction_renaming_subst_wt[OF assms(3)]
      transaction_fresh_subst_wt[OF assms(2)]
      transaction_decl_subst_wt[OF assms(1)]
by (metis wt_subst_compose)

lemma transaction_decl_fresh_renaming_substs_range_wf_trms:
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst ξ T" "transaction_fresh_subst σ T M"
          "transaction_renaming_subst α P X"
  shows "wf_{trms} (subst_range (ξ ∘_s σ ∘_s α))"
using transaction_renaming_subst_range_wf_trms[OF assms(3)]
      transaction_fresh_subst_range_wf_trms[OF assms(2)]
      transaction_decl_subst_range_wf_trms[OF assms(1)]
      wf_trms_subst_compose[of ξ σ]
      wf_trms_subst_compose[of "ξ ∘_s σ" α]
by metis

lemma transaction_decl_fresh_renaming_substs_fv:
  fixes σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T M"
    and α: "transaction_renaming_subst α P X"
    and x: "x ∈ fv_{lsst} (dual_{lsst} (transaction_strand T ·_{lsst} ξ ∘_s σ ∘_s α))"
  shows "∃y ∈ fv_transaction T - set (transaction_fresh T). (ξ ∘_s σ ∘_s α) y = Var x"
proof -
  have "x ∈ fv_{sst} (unlabel (transaction_strand T) ·_{sst} ξ ∘_s σ ∘_s α)"
    using x fv_{sst}_unlabel_dual_{lsst}_eq[of "transaction_strand T ·_{lsst} ξ ∘_s σ ∘_s α"]
          unlabel_subst[of "transaction_strand T" "ξ ∘_s σ ∘_s α"]
    by argo
  then obtain y where "y ∈ fv_transaction T" "x ∈ fv ((ξ ∘_s σ ∘_s α) y)"
    by (metis fv_{sst}_subst_obtain_var)
  thus ?thesis
    using transaction_decl_fresh_renaming_substs_range[OF ξ σ α, of y]
    by (cases "y ∈ set (transaction_fresh T)") force+
qed

lemma transaction_decl_fresh_renaming_substs_range_no_attack_const:
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T M"
    and α: "transaction_renaming_subst α P X"
    and T: "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    and t: "t ∈ subst_range (ξ ∘_s σ ∘_s α)"
  shows "∄n. t = attack⟨n⟩"
proof -
  note ξσα_wt = transaction_decl_fresh_renaming_substs_wt[OF ξ σ α]

  obtain x where x: "(ξ ∘_s σ ∘_s α) x = t" using t by auto

  have x_type: "Γ (Var x) = Γ (Var x · ξ)" "Γ (Var x) = Γ (Var x · ξ ∘_s σ ∘_s α)"
    using ξ wt_subst_trm''[of ξ "Var x"] wt_subst_trm''[OF ξσα_wt, of "Var x"]
    unfolding transaction_decl_subst_def by (blast, blast)

  show ?thesis
  proof (cases t)
    case (Fun f S)
    hence "x ∈ set (transaction_fresh T) ∨ x ∈ fst ` set (transaction_decl T ())"
      using transaction_decl_fresh_renaming_substs_range[OF ξ σ α, of x] x by force
    thus ?thesis
```

```
    proof
      assume "x ∈ set (transaction_fresh T)"
      hence "Γ t = TAtom Value ∨ (∃a. Γ t = TAtom (Atom a))"
        using T x_type(2) x by (metis Γ.simps(1) eval_term.simps(1))
      thus ?thesis by auto
    next
      assume "x ∈ fst ` set (transaction_decl T ())"
      then obtain c where c: "ξ x = Fun (Fu c) []" "arity_f c = 0"
        using ξ unfolding transaction_decl_subst_def by auto

      have "Γ t = TAtom Bottom ∨ (∃a. Γ t = TAtom (Atom a))"
        using c(1) Γ_consts_simps(1)[OF c(2)] x x_type
              eval_term.simps(1)[of _ x ξ] eval_term.simps(1)[of _ x "ξ ∘s σ ∘s α"]
        by (cases "Γ_f c") simp_all
      thus ?thesis by auto
    qed
  qed simp
qed

lemma transaction_decl_fresh_renaming_substs_occurs_fact_send_receive:
  fixes t::"('fun,'atom,'sets,'lbl) prot_term"
  assumes ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T M"
    and α: "transaction_renaming_subst α P X"
    and T: "admissible_transaction' T"
    and t: "occurs t ∈ set ts"
  shows "send⟨ts⟩ ∈ set (unlabel (transaction_strand T ·lsst ξ ∘s σ ∘s α))
          ⟹ ∃ts' s. send⟨ts'⟩ ∈ set (unlabel (transaction_send T)) ∧
                      occurs s ∈ set ts' ∧ t = s · ξ ∘s σ ∘s α"
      (is "?A ⟹ ?A'")
    and "receive⟨ts⟩ ∈ set (unlabel (transaction_strand T ·lsst ξ ∘s σ ∘s α))
          ⟹ ∃ts' s. receive⟨ts'⟩ ∈ set (unlabel (transaction_receive T)) ∧
                      occurs s ∈ set ts' ∧ t = s · ξ ∘s σ ∘s α"
      (is "?B ⟹ ?B'")
proof -
  assume ?A
  then obtain s ts' where s:
      "s ∈ set ts'" "send⟨ts'⟩ ∈ set (unlabel (transaction_strand T))" "occurs t = s · ξ ∘s σ ∘s α"
    using t stateful_strand_step_mem_substD(1)[
            of ts "unlabel (transaction_strand T)" "ξ ∘s σ ∘s α"]
          unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
    by auto

  note ξ_empty = admissible_transaction_decl_subst_empty[OF T ξ]

  have T_decl_notin: "x ∉ fst ` set (transaction_decl T ())" for x
    using transaction_decl_subst_empty_inv[OF ξ[unfolded ξ_empty]] by simp

  note 0 = s(3) transaction_decl_fresh_renaming_substs_range[OF ξ σ α]

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T]
  note T_fresh = admissible_transactionE(14)[OF T]

  have "∃u. s = occurs u"
  proof (cases s)
    case (Var x)
    hence "(∃c. s · ξ ∘s σ ∘s α = Fun c []) ∨ (∃y. s · ξ ∘s σ ∘s α = Var y)"
      using 0(2-5)[of x] ξ_empty by (auto simp del: subst_subst_compose)
    thus ?thesis
      using 0(1) by simp
  next
    case (Fun f T)
    hence 1: "f = OccursFact" "length T = 2" "T ! 0 · ξ ∘s σ ∘s α = Fun OccursSec []"
```

```
            "T ! 1 · ξ ∘s σ ∘s α = t"
      using 0(1) by auto
    have "T ! 0 = Fun OccursSec []"
    proof (cases "T ! 0")
      case (Var x) thus ?thesis
        using 0(2-5)[of x] 1(3) T_fresh T_decl_notin
        unfolding list_all_iff by (auto simp del: subst_subst_compose)
    qed (use 1(3) in simp)
    thus ?thesis using Fun 1 0(1) by (auto simp del: subst_subst_compose)
  qed
  then obtain u where u: "s = occurs u" by force
  hence "t = u · ξ ∘s σ ∘s α" using s(3) by fastforce
  thus ?A' using s u wellformed_transaction_strand_unlabel_memberD(8)[OF T_wf] by metis
next
  assume ?B
  then obtain s ts' where s:
      "s ∈ set ts'" "receive⟨ts'⟩ ∈ set (unlabel (transaction_strand T))" "occurs t = s · ξ ∘s σ ∘s α"
    using t stateful_strand_step_mem_substD(2)[
            of ts "unlabel (transaction_strand T)" "ξ ∘s σ ∘s α"]
          unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
    by auto

  note ξ_empty = admissible_transaction_decl_subst_empty[OF T ξ]

  have T_decl_notin: "x ∉ fst ` set (transaction_decl T ())" for x
    using transaction_decl_subst_empty_inv[OF ξ[unfolded ξ_empty]] by simp

  note 0 = s(3) transaction_decl_fresh_renaming_substs_range[OF ξ σ α]

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T]
  note T_fresh = admissible_transactionE(14)[OF T]

  have "∃u. s = occurs u"
  proof (cases s)
    case (Var x)
    hence "(∃c. s · ξ ∘s σ ∘s α = Fun c []) ∨ (∃y. s · ξ ∘s σ ∘s α = Var y)"
      using 0(2-5)[of x] ξ_empty by (auto simp del: subst_subst_compose)
    thus ?thesis
      using 0(1) by simp
  next
    case (Fun f T)
    hence 1: "f = OccursFact" "length T = 2" "T ! 0 · ξ ∘s σ ∘s α = Fun OccursSec []"
             "T ! 1 · ξ ∘s σ ∘s α = t"
      using 0(1) by auto
    have "T ! 0 = Fun OccursSec []"
    proof (cases "T ! 0")
      case (Var x) thus ?thesis
        using 0(2-5)[of x] 1(3) T_fresh T_decl_notin
        unfolding list_all_iff by (auto simp del: subst_subst_compose)
    qed (use 1(3) in simp)
    thus ?thesis using Fun 1 0(1) by (auto simp del: subst_subst_compose)
  qed
  then obtain u where u: "s = occurs u" by force
  hence "t = u · ξ ∘s σ ∘s α" using s(3) by fastforce
  thus ?B' using s u wellformed_transaction_strand_unlabel_memberD(1)[OF T_wf] by metis
qed

lemma transaction_decl_subst_proj:
  assumes "transaction_decl_subst ξ T"
  shows "transaction_decl_subst ξ (transaction_proj n T)"
using assms transaction_proj_decl_eq[of n T]
unfolding transaction_decl_subst_def by presburger
```

```
lemma transaction_fresh_subst_proj:
  assumes "transaction_fresh_subst σ T (trms_lsst A)"
  shows "transaction_fresh_subst σ (transaction_proj n T) (trms_lsst (proj n A))"
using assms transaction_proj_fresh_eq[of n T]
      contra_subsetD[OF subterms_set_mono[OF transaction_proj_trms_subset[of n T]]]
      contra_subsetD[OF subterms_set_mono[OF trms_sst_proj_subset(1)[of n A]]]
unfolding transaction_fresh_subst_def by metis

lemma transaction_renaming_subst_proj:
  assumes "transaction_renaming_subst α P (vars_lsst A)"
  shows "transaction_renaming_subst α (map (transaction_proj n) P) (vars_lsst (proj n A))"
proof -
  let ?X = "λP A. ⋃(vars_transaction ` set P) ∪ vars_lsst A"
  define Y where "Y ≡ ?X (map (transaction_proj n) P) (proj n A)"
  define Z where "Z ≡ ?X P A"

  have "Y ⊆ Z"
    using sst_vars_proj_subset(3)[of n A] transaction_proj_vars_subset[of n]
    unfolding Y_def Z_def by fastforce
  hence "insert 0 (snd ` Y) ⊆ insert 0 (snd ` Z)" by blast
  moreover have "finite (insert 0 (snd ` Z))" "finite (insert 0 (snd ` Y))"
    unfolding Y_def Z_def by auto
  ultimately have 0: "max_var_set Y ≤ max_var_set Z" using Max_mono by blast

  have "∃n≥max_var_set Z. α = var_rename n"
    using assms unfolding transaction_renaming_subst_def Z_def by blast
  hence "∃n≥max_var_set Y. α = var_rename n" using 0 le_trans by fast
  thus ?thesis unfolding transaction_renaming_subst_def Y_def by blast
qed

lemma transaction_decl_fresh_renaming_substs_wf_sst:
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes T: "wf'_sst (fst ` set (transaction_decl T ()) ∪ set (transaction_fresh T))
                  (unlabel (dual_lsst (transaction_strand T)))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trms_lsst 𝒜)"
    and α: "transaction_renaming_subst α P (vars_lsst 𝒜)"
  shows "wf'_sst {} (unlabel (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)))"
proof -
  have 0: "range_vars ξ ∩ bvars_lsst (dual_lsst (transaction_strand T)) = {}"
          "range_vars σ ∩ bvars_lsst (dual_lsst (transaction_strand T) ·_lsst ξ) = {}"
          "ground (ξ ` (fst ` set (transaction_decl T ())))"
          "ground (σ ` set (transaction_fresh T))"
          "ground (α ` {})"
    using transaction_decl_subst_domain[OF ξ]
          transaction_decl_subst_grounds_domain[OF ξ]
          transaction_decl_subst_range_vars_empty[OF ξ]
          transaction_fresh_subst_range_vars_empty[OF σ]
          transaction_fresh_subst_domain[OF σ]
          transaction_fresh_subst_grounds_domain[OF σ]
    by (simp, simp, simp, simp, simp)

  have 1: "fv_set (ξ ` set (transaction_fresh T)) ⊆ set (transaction_fresh T)" (is "?A ⊆ ?B")
  proof
    fix x assume x: "x ∈ ?A"
    then obtain y where y: "y ∈ set (transaction_fresh T)" "x ∈ fv (ξ y)" by auto
    hence "y ∉ subst_domain ξ"
      using transaction_decl_subst_domain[OF ξ]
            transaction_decl_subst_grounds_domain[OF ξ]
      by fast
    thus "x ∈ ?B" using x y by auto
  qed
```

```
  let ?X = "fst ` set (transaction_decl T ()) ∪ set (transaction_fresh T)"

  have "fv_set (α ` fv_set (σ ` fv_set (ξ ` ?X))) = {}" using 0(3-5) 1 by auto
  hence "wf'_sst {} (((unlabel (dual_lsst (transaction_strand T)) ·_sst ξ) ·_sst σ) ·_sst α)"
    by (metis wf_sst_subst_apply[OF wf_sst_subst_apply[OF wf_sst_subst_apply[OF T]]])
  thus ?thesis
    using dual_lsst_subst unlabel_subst
          labeled_stateful_strand_subst_comp[OF 0(1), of "σ ∘_s α"]
          labeled_stateful_strand_subst_comp[OF 0(2), of α]
          subst_compose_assoc[of ξ σ α]
    by metis
qed

lemma admissible_transaction_decl_fresh_renaming_subst_not_occurs:
  fixes ξ σ α
  defines "ϑ ≡ ξ ∘_s σ ∘_s α"
  assumes T_adm: "admissible_transaction' T"
    and ξσα:
      "transaction_decl_subst ξ T"
      "transaction_fresh_subst σ T (trms_lsst 𝒜)"
      "transaction_renaming_subst α P (vars_lsst 𝒜)"
  shows "∄t. ϑ x = occurs t"
    and "ϑ x ≠ Fun OccursSec []"
proof -
  note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm ξσα(1)]
  note T_fresh_val = admissible_transactionE(2)[OF T_adm]

  show "∄t. ϑ x = occurs t" for x
    using transaction_decl_fresh_renaming_substs_range'(1)[OF ξσα]
    unfolding ϑ_def[symmetric] by (cases "x ∈ subst_domain ϑ") (force,force)

  show "ϑ x ≠ Fun OccursSec []" for x
    using transaction_decl_fresh_renaming_substs_range'(3)[
            OF ξσα _ ξ_empty T_fresh_val, of "ϑ x"]
    unfolding ϑ_def[symmetric] by (cases "x ∈ subst_domain ϑ") auto
qed
```

3.3.8 Lemmata: Reachable Constraints

```
lemma reachable_constraints_as_transaction_lists:
  fixes f
  defines "f ≡ λ(T,ξ,σ,α). dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
    and "g ≡ concat ∘ map f"
  assumes A: "A ∈ reachable_constraints P"
  obtains Ts where "A = g Ts"
    and "∀B. prefix B Ts ⟶ g B ∈ reachable_constraints P"
    and "∀B T ξ σ α. prefix (B@[(T,ξ,σ,α)]) Ts ⟶
                      T ∈ set P ∧ transaction_decl_subst ξ T ∧
                      transaction_fresh_subst σ T (trms_lsst (g B)) ∧
                      transaction_renaming_subst α P (vars_lsst (g B))"
proof -
  let ?P1 = "λA Ts. A = g Ts"
  let ?P2 = "λTs. ∀B. prefix B Ts ⟶ g B ∈ reachable_constraints P"
  let ?P3 = "λTs. ∀B T ξ σ α. prefix (B@[(T,ξ,σ,α)]) Ts ⟶
                      T ∈ set P ∧ transaction_decl_subst ξ T ∧
                      transaction_fresh_subst σ T (trms_lsst (g B)) ∧
                      transaction_renaming_subst α P (vars_lsst (g B))"

  have "∃Ts. ?P1 A Ts ∧ ?P2 Ts ∧ ?P3 Ts" using A
  proof (induction A rule: reachable_constraints.induct)
    case init
    have "?P1 [] []" "?P2 []" "?P3 []" unfolding g_def f_def by simp_all
    thus ?case by blast
```

```
  next
    case (step A T ξ σ α)
    let ?A' = "A@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
    obtain Ts where Ts: "?P1 A Ts" "?P2 Ts" "?P3 Ts" using step.IH by blast

    have 1: "?P1 ?A' (Ts@[(T,ξ,σ,α)])"
      using Ts(1) unfolding g_def f_def by simp

    have 2: "?P2 (Ts@[(T,ξ,σ,α)])"
    proof (intro allI impI)
      fix B assume "prefix B (Ts@[(T,ξ,σ,α)])"
      hence "prefix B Ts ∨ B = Ts@[(T,ξ,σ,α)]" by fastforce
      thus "g B ∈ reachable_constraints P "
        using Ts(1,2) reachable_constraints.step[OF step.hyps]
        unfolding g_def f_def by auto
    qed

    have 3: "?P3 (Ts@[(T,ξ,σ,α)])"
      using Ts(1,3) step.hyps(2-5) unfolding g_def f_def by auto

    show ?case using 1 2 3 by blast
  qed
  thus thesis using that by blast
qed

lemma reachable_constraints_transaction_action_obtain:
  assumes A: "A ∈ reachable_constraints P"
    and a: "a ∈ set A"
  obtains T b B α σ ξ
  where "prefix (B@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) A"
    and "T ∈ set P" "transaction_decl_subst ξ T" "transaction_fresh_subst σ T (trms_lsst B)"
        "transaction_renaming_subst α P (vars_lsst B)"
    and "b ∈ set (transaction_strand T)" "a = dual_lsstp b ·_lsstp ξ ∘_s σ ∘_s α" "fst a = fst b"
proof -
  define f where "f ≡ λ(T,ξ,σ::('fun,'atom,'sets,'lbl) prot_subst,α).
                        dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
  define g where "g ≡ concat ∘ map f"

  obtain Ts where Ts:
      "A = g Ts" "∀B. prefix B Ts ⟶ g B ∈ reachable_constraints P"
      "∀B T ξ σ α. prefix (B@[(T,ξ,σ,α)]) Ts ⟶
            T ∈ set P ∧ transaction_decl_subst ξ T ∧
            transaction_fresh_subst σ T (trms_lsst (g B)) ∧
            transaction_renaming_subst α P (vars_lsst (g B))"
    using reachable_constraints_as_transaction_lists[OF A] unfolding g_def f_def by blast

  obtain T α ξ σ where T: "(T,ξ,σ,α) ∈ set Ts" "a ∈ set (f (T,ξ,σ,α))"
    using Ts(1) a unfolding g_def by auto

  obtain B where B: "prefix (B@[(T,ξ,σ,α)]) Ts"
    using T(1) by (meson prefix_snoc_in_iff)

  obtain b where b:
      "b ∈ set (transaction_strand T)" "a = dual_lsstp b ·_lsstp ξ ∘_s σ ∘_s α" "fst a = fst b"
    using T(2) dual_lsst_subst[of "transaction_strand T" "ξ ∘_s σ ∘_s α"]
          dual_lsst_memberD'[of a "transaction_strand T" "ξ ∘_s σ ∘_s α" thesis]
    unfolding f_def by simp

  have 0: "prefix (g B@f (T, ξ, σ, α)) A"
    using concat_map_mono_prefix[OF B, of f] unfolding g_def Ts(1) by simp

  have 1: "T ∈ set P" "transaction_decl_subst ξ T" "transaction_fresh_subst σ T (trms_lsst (g B))"
        "transaction_renaming_subst α P (vars_lsst (g B))"
```

```
    using B Ts(3) by (blast,blast,blast,blast)

  show thesis using 0[unfolded f_def] that[OF _ 1 b] by fast
qed

lemma reachable_constraints_unlabel_eq:
  defines "transaction_unlabel_eq ≡ λT1 T2.
           transaction_decl T1     =         transaction_decl T2 ∧
           transaction_fresh T1    =         transaction_fresh T2 ∧
    unlabel (transaction_receive T1) = unlabel (transaction_receive T2) ∧
    unlabel (transaction_checks T1)  = unlabel (transaction_checks T2) ∧
    unlabel (transaction_updates T1) = unlabel (transaction_updates T2) ∧
    unlabel (transaction_send T1)    = unlabel (transaction_send T2)"
  assumes Peq: "list_all2 transaction_unlabel_eq P1 P2"
  shows "unlabel ` reachable_constraints P1 = unlabel ` reachable_constraints P2" (is "?A = ?B")
proof (intro antisym subsetI)
  have "transaction_unlabel_eq T2 T1 = transaction_unlabel_eq T1 T2" for T1 T2
    unfolding transaction_unlabel_eq_def by argo
  hence Peq': "list_all2 transaction_unlabel_eq P2 P1"
    using Peq list_all2_sym by metis

  have 0: "unlabel (transaction_strand T1) = unlabel (transaction_strand T2)"
    when "transaction_unlabel_eq T1 T2" for T1 T2
    using that unfolding transaction_unlabel_eq_def transaction_strand_def by force

  have "vars_transaction T1 = vars_transaction T2" when "transaction_unlabel_eq T1 T2" for T1 T2
    using 0[OF that] by simp
  hence "vars_transaction (P1 ! i) = vars_transaction (P2 ! i)" when "i < length P1" for i
    using that Peq list_all2_conv_all_nth by blast
  moreover have "length P1 = length P2" using Peq unfolding list_all2_iff by argo
  ultimately have 1: "⋃(vars_transaction ` set P1) = ⋃(vars_transaction ` set P2)"
    using in_set_conv_nth[of _ P1] in_set_conv_nth[of _ P2] by fastforce

  have 2:
      "transaction_decl_subst ξ T1 ⟹ transaction_decl_subst ξ T2" (is "?A1 ⟹ ?A2")
      "transaction_fresh_subst σ T1 (trms_lsst 𝒜) ⟹ transaction_fresh_subst σ T2 (trms_lsst ℬ)"
        (is "?B1 ⟹ ?B2")
      "transaction_renaming_subst α P1 (vars_lsst 𝒜) ⟹ transaction_renaming_subst α P2 (vars_lsst ℬ)"
        (is "?C1 ⟹ ?C2")
      "transaction_renaming_subst α P2 (vars_lsst 𝒜) ⟹ transaction_renaming_subst α P1 (vars_lsst ℬ)"
        (is "?D1 ⟹ ?D2")
    when "transaction_unlabel_eq T1 T2" "unlabel 𝒜 = unlabel ℬ"
    for T1 T2::"('fun,'atom,'sets,'lbl) prot_transaction"
      and 𝒜 ℬ::"('fun,'atom,'sets,'lbl) prot_strand"
      and ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
  proof -
    have *: "transaction_decl T1 = transaction_decl T2"
          "transaction_fresh T1 = transaction_fresh T2"
          "trms_transaction T1 = trms_transaction T2"
      using that unfolding transaction_unlabel_eq_def transaction_strand_def by force+

    show "?A1 ⟹ ?A2" using *(1) unfolding transaction_decl_subst_def by argo
    show "?B1 ⟹ ?B2" using that(2) *(2,3) unfolding transaction_fresh_subst_def by force
    show "?C1 ⟹ ?C2" using that(2) 1 unfolding transaction_renaming_subst_def by metis
    show "?D1 ⟹ ?D2" using that(2) 1 unfolding transaction_renaming_subst_def by metis
  qed

  have 3: "unlabel (dual_lsst (transaction_strand T1 ·lsst ϑ)) =
          unlabel (dual_lsst (transaction_strand T2 ·lsst ϑ))"
    when "transaction_unlabel_eq T1 T2" for T1 T2 ϑ
    using 0[OF that] unlabel_subst[of _ ϑ] dual_lsst_unlabel_cong by metis

  have "∃ℬ ∈ reachable_constraints P2. unlabel 𝒜 = unlabel ℬ"
```

```
    when "𝒜 ∈ reachable_constraints P1" for 𝒜 using that
  proof (induction 𝒜 rule: reachable_constraints.induct)
    case (step 𝒜 T ξ σ α)
    obtain ℬ where IH: "ℬ ∈ reachable_constraints P2" "unlabel 𝒜 = unlabel ℬ"
      by (meson step.IH)

    obtain T' where T': "T' ∈ set P2" "transaction_unlabel_eq T T'"
      using list_all2_in_set_ex[OF Peq step.hyps(2)] by auto

    show ?case
      using 3[OF T'(2), of "ξ ∘s σ ∘s α"] IH(2) reachable_constraints.step[OF IH(1) T'(1)]
            2[OF T'(2) IH(2)] step.hyps(3-5)
      by (metis unlabel_append[of 𝒜] unlabel_append[of ℬ])
  qed (simp add: unlabel_def)
  thus "𝒜 ∈ ?A ⟹ 𝒜 ∈ ?B" for 𝒜 by fast

  have "∃ℬ ∈ reachable_constraints P1. unlabel 𝒜 = unlabel ℬ"
    when "𝒜 ∈ reachable_constraints P2" for 𝒜 using that
  proof (induction 𝒜 rule: reachable_constraints.induct)
    case (step 𝒜 T ξ σ α)
    obtain ℬ where IH: "ℬ ∈ reachable_constraints P1" "unlabel 𝒜 = unlabel ℬ"
      by (meson step.IH)

    obtain T' where T': "T' ∈ set P1" "transaction_unlabel_eq T T'"
      using list_all2_in_set_ex[OF Peq' step.hyps(2)] by auto

    show ?case
      using 3[OF T'(2), of "ξ ∘s σ ∘s α"] IH(2) reachable_constraints.step[OF IH(1) T'(1)]
            2[OF T'(2) IH(2)] step.hyps(3-5)
      by (metis unlabel_append[of 𝒜] unlabel_append[of ℬ])
  qed (simp add: unlabel_def)
  thus "𝒜 ∈ ?B ⟹ 𝒜 ∈ ?A" for 𝒜 by fast
qed

lemma reachable_constraints_set_eq:
  assumes "set P1 = set P2"
  shows "reachable_constraints P1 = reachable_constraints P2" (is "?A = ?B")
proof (intro antisym subsetI)
  note 0 = assms transaction_renaming_subst_set_eq[OF assms]
  note 1 = reachable_constraints.intros

  show "𝒜 ∈ ?A ⟹ 𝒜 ∈ ?B" for 𝒜
    by (induct 𝒜 rule: reachable_constraints.induct) (auto simp add: 0 intro: 1)

  show "𝒜 ∈ ?B ⟹ 𝒜 ∈ ?A" for 𝒜
    by (induct 𝒜 rule: reachable_constraints.induct) (auto simp add: 0 intro: 1)
qed

lemma reachable_constraints_set_subst:
  assumes "set P1 = set P2"
    and "Q (reachable_constraints P1)"
  shows "Q (reachable_constraints P2)"
by (rule subst[of _ _ Q, OF reachable_constraints_set_eq[OF assms(1)] assms(2)])

lemma reachable_constraints_wf_trms:
  assumes "∀T ∈ set P. wf_trms (trms_transaction T)"
    and "𝒜 ∈ reachable_constraints P"
  shows "wf_trms (trms_lsst 𝒜)"
using assms(2)
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  have "wf_trms (trms_transaction T)"
    using assms(1) step.hyps(2) by blast
```

```
  hence "wf_{trms} (trms_transaction T ·_{set} ξ ∘_s σ ∘_s α)"
    using transaction_decl_fresh_renaming_substs_range_wf_trms[OF step.hyps(3-5)]
    by (metis wf_trms_subst)
  hence "wf_{trms} (trms_{lsst} (transaction_strand T ·_{lsst} ξ ∘_s σ ∘_s α))"
    using wf_{trms}_trms_{sst}_subst unlabel_subst[of "transaction_strand T" "ξ ∘_s σ ∘_s α"] by metis
  hence "wf_{trms} (trms_{lsst} (dual_{lsst} (transaction_strand T ·_{lsst} ξ ∘_s σ ∘_s α)))"
    using trms_{sst}_unlabel_dual_{lsst}_eq by blast
  thus ?case using step.IH unlabel_append[of 𝒜] trms_{sst}_append[of "unlabel 𝒜"] by auto
qed simp

lemma reachable_constraints_var_types_in_transactions:
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T).
              Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
  shows "Γ_v ` fv_{lsst} 𝒜 ⊆ (⋃T ∈ set P. Γ_v ` fv_transaction T)" (is "?A 𝒜")
    and "Γ_v ` bvars_{lsst} 𝒜 ⊆ (⋃T ∈ set P. Γ_v ` bvars_transaction T)" (is "?B 𝒜")
    and "Γ_v ` vars_{lsst} 𝒜 ⊆ (⋃T ∈ set P. Γ_v ` vars_transaction T)" (is "?C 𝒜")
using 𝒜
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  define T' where "T' ≡ dual_{lsst} (transaction_strand T ·_{lsst} ξ ∘_s σ ∘_s α)"

  note 2 = transaction_decl_fresh_renaming_substs_wt[OF step.hyps(3-5)]

  have 3: "∀t ∈ subst_range (ξ ∘_s σ ∘_s α). fv t = {} ∨ (∃x. t = Var x)"
    using transaction_decl_fresh_renaming_substs_range'(1)[OF step.hyps(3-5)]
    by fastforce

  have "fv_{lsst} T' = fv_{lsst} (transaction_strand T ·_{lsst} ξ ∘_s σ ∘_s α)"
       "bvars_{lsst} T' = bvars_{lsst} (transaction_strand T ·_{lsst} ξ ∘_s σ ∘_s α)"
       "vars_{lsst} T' = vars_{lsst} (transaction_strand T ·_{lsst} ξ ∘_s σ ∘_s α)"
    unfolding T'_def
    by (metis fv_{sst}_unlabel_dual_{lsst}_eq,
        metis bvars_{sst}_unlabel_dual_{lsst}_eq,
        metis vars_{sst}_unlabel_dual_{lsst}_eq)
  hence "Γ ` Var ` fv_{lsst} T' ⊆ Γ ` Var ` fv_transaction T"
        "Γ ` Var ` bvars_{lsst} T' = Γ ` Var ` bvars_transaction T"
        "Γ ` Var ` vars_{lsst} T' ⊆ Γ ` Var ` vars_transaction T"
    using wt_subst_lsst_vars_type_subset[OF 2 3, of "transaction_strand T"]
    by argo+
  hence "Γ_v ` fv_{lsst} T' ⊆ Γ_v ` fv_transaction T"
        "Γ_v ` bvars_{lsst} T' = Γ_v ` bvars_transaction T"
        "Γ_v ` vars_{lsst} T' ⊆ Γ_v ` vars_transaction T"
    by (metis Γ_v_Var_image)+
  hence 4: "Γ_v ` fv_{lsst} T' ⊆ (⋃T ∈ set P. Γ_v ` fv_transaction T)"
           "Γ_v ` bvars_{lsst} T' ⊆ (⋃T ∈ set P. Γ_v ` bvars_transaction T)"
           "Γ_v ` vars_{lsst} T' ⊆ (⋃T ∈ set P. Γ_v ` vars_transaction T)"
    using step.hyps(2) by fast+

  have 5: "Γ_v ` fv_{lsst} (𝒜 @ T') = (Γ_v ` fv_{lsst} 𝒜) ∪ (Γ_v ` fv_{lsst} T')"
          "Γ_v ` bvars_{lsst} (𝒜 @ T') = (Γ_v ` bvars_{lsst} 𝒜) ∪ (Γ_v ` bvars_{lsst} T')"
          "Γ_v ` vars_{lsst} (𝒜 @ T') = (Γ_v ` vars_{lsst} 𝒜) ∪ (Γ_v ` vars_{lsst} T')"
    using unlabel_append[of 𝒜 T']
          fv_{sst}_append[of "unlabel 𝒜" "unlabel T'"]
          bvars_{sst}_append[of "unlabel 𝒜" "unlabel T'"]
          vars_{sst}_append[of "unlabel 𝒜" "unlabel T'"]
    by auto

  { case 1 thus ?case
      using step.IH(1) 4(1) 5(1)
      unfolding T'_def by (simp del: subst_subst_compose fv_{sst}_def)
  }
```

```
  { case 2 thus ?case
      using step.IH(2) 4(2) 5(2)
      unfolding T'_def by (simp del: subst_subst_compose bvars_sst_def)
  }

  { case 3 thus ?case
      using step.IH(3) 4(3) 5(3)
      unfolding T'_def by (simp del: subst_subst_compose)
  }
qed simp_all

lemma reachable_constraints_no_bvars:
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. bvars_lsst (transaction_strand T) = {}"
  shows "bvars_lsst 𝒜 = {}"
using assms proof (induction)
  case init
  then show ?case
    unfolding unlabel_def by auto
next
  case (step 𝒜 T ξ σ α)
  then have "bvars_lsst 𝒜 = {}"
    by metis
  moreover
  have "bvars_lsst (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) = {}"
    using step by (metis bvars_lsst_subst bvars_sst_unlabel_dual_lsst_eq)
  ultimately
  show ?case
    using bvars_sst_append unlabel_append by (metis sup_bot.left_neutral)
qed

lemma reachable_constraints_fv_bvars_disj:
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and P: "∀S ∈ set P. admissible_transaction' S"
  shows "fv_lsst 𝒜 ∩ bvars_lsst 𝒜 = {}"
proof -
  let ?X = "⋃T ∈ set P. bvars_transaction T"

  note 0 = admissible_transactions_fv_bvars_disj[OF P]

  have 1: "bvars_lsst 𝒜 ⊆ ?X" using 𝒜_reach
  proof (induction 𝒜 rule: reachable_constraints.induct)
    case (step 𝒜 T ξ σ α)
    have "bvars_lsst (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) = bvars_transaction T"
      using bvars_sst_subst[of "unlabel (transaction_strand T)" "ξ ∘_s σ ∘_s α"]
            bvars_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·_lsst ξ ∘_s σ ∘_s α"]
            dual_lsst_subst[of "transaction_strand T" "ξ ∘_s σ ∘_s α"]
            unlabel_subst[of "transaction_strand T" "ξ ∘_s σ ∘_s α"]
      by argo
    hence "bvars_lsst (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) ⊆ ?X"
      using step.hyps(2)
      by blast
    thus ?case
      using step.IH bvars_sst_append
      by auto
  qed (simp add: unlabel_def)

  have 2: "fv_lsst 𝒜 ∩ ?X = {}" using 𝒜_reach
  proof (induction 𝒜 rule: reachable_constraints.induct)
    case (step 𝒜 T ξ σ α)
    have "x ≠ y" when x: "x ∈ ?X" and y: "y ∈ fv_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)" for x
```

```
y
    proof -
      obtain y' where y': "y' ∈ fv_transaction T" "y ∈ fv ((ξ ∘s σ ∘s α) y')"
        using y unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
        by (metis fv_sst_subst_obtain_var)

      have "y ∉ ⋃(vars_transaction ` set P)"
        using transaction_decl_fresh_renaming_substs_range''[OF step.hyps(3-5) y'(2)]
              transaction_renaming_subst_range_notin_vars[OF step.hyps(5), of y']
        by auto
      thus ?thesis using x vars_sst_is_fv_sst_bvars_sst by fast
    qed
    hence "fv_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α) ∩ ?X = {}"
      by blast
    thus ?case
      using step.IH
            fv_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst ξ ∘s σ ∘s α"]
            dual_lsst_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
            unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
            fv_sst_append[of "unlabel 𝒜" "unlabel (transaction_strand T ·lsst ξ ∘s σ ∘s α)"]
            unlabel_append[of 𝒜 "transaction_strand T"]
      by force
  qed (simp add: unlabel_def)

  show ?thesis using 0 1 2 by blast
qed

lemma reachable_constraints_vars_TAtom_typed:
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and x: "x ∈ vars_lsst 𝒜"
  shows "Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
proof -
  have 𝒜_wf_trms: "wf_trms (trms_lsst 𝒜)"
    by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P 𝒜_reach)

  have T_adm: "admissible_transaction' T" when "T ∈ set P" for T
    by (meson that Ball_set P)

  have "∀T∈set P. ∀x∈set (transaction_fresh T). Γv x = TAtom Value"
    using protocol_transaction_vars_TAtom_typed(3) P by blast
  hence *: "Γv ` vars_lsst 𝒜 ⊆ (⋃T∈set P. Γv ` vars_transaction T)"
    using reachable_constraints_var_types_in_transactions[of 𝒜 P, OF 𝒜_reach] by auto

  have "Γv ` vars_lsst 𝒜 ⊆ TAtom ` insert Value (range Atom)"
  proof -
    have "Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
      when "T ∈ set P" "x ∈ vars_transaction T" for T x
      using that protocol_transaction_vars_TAtom_typed(1)[of T] P
            admissible_transactionE(5)
      by blast
    hence "(⋃T∈set P. Γv ` vars_transaction T) ⊆ TAtom ` insert Value (range Atom)"
      using P by blast
    thus "Γv ` vars_lsst 𝒜 ⊆ TAtom ` insert Value (range Atom)"
      using * by auto
  qed
  thus ?thesis using x by auto
qed

lemma reachable_constraints_vars_not_attack_typed:
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
```

```
    and P: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T).
              Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
           "∀T ∈ set P. ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γv x"
    and x: "x ∈ varslsst 𝒜"
  shows "¬TAtom AttackType ⊑ Γv x"
using reachable_constraints_var_types_in_transactions(3)[OF 𝒜_reach P(1)] P(2) x by fastforce

lemma reachable_constraints_Value_vars_are_fv:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and x: "x ∈ varslsst 𝒜"
    and "Γv x = TAtom Value"
  shows "x ∈ fvlsst 𝒜"
proof -
  have "∀T∈set P. bvars_transaction T = {}"
    using P admissible_transactionE(4) by metis
  hence 𝒜_no_bvars: "bvarslsst 𝒜 = {}"
    using reachable_constraints_no_bvars[OF 𝒜_reach] by metis
  thus ?thesis using x varssst_is_fvsst_bvarssst[of "unlabel 𝒜"] by blast
qed

lemma reachable_constraints_subterms_subst:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction' T"
  shows "subtermsset (trmslsst (𝒜 ·lsst ℐ)) = (subtermsset (trmslsst 𝒜)) ·set ℐ"
proof -
  have 𝒜_wftrms: "wftrms (trmslsst 𝒜)"
    by (metis reachable_constraints_wftrms admissible_transactions_wftrms P 𝒜_reach)

  from ℐ have ℐ': "welltyped_constraint_model ℐ 𝒜"
    using welltyped_constraint_model_prefix by auto

  have 1: "∀x ∈ fvset (trmslsst 𝒜). (∃f. ℐ x = Fun f []) ∨ (∃y. ℐ x = Var y)"
  proof
    fix x
    assume xa: "x ∈ fvset (trmslsst 𝒜)"
    have "∃f T. ℐ x = Fun f T"
      using ℐ interpretation_grounds[of ℐ "Var x"]
      unfolding welltyped_constraint_model_def constraint_model_def
      by (cases "ℐ x") auto
    then obtain f T where fT_p: "ℐ x = Fun f T"
      by auto
    hence "wftrm (Fun f T)"
      using ℐ
      unfolding welltyped_constraint_model_def constraint_model_def
      using wf_trm_subst_rangeD
      by metis
    moreover
    have "x ∈ varslsst 𝒜"
      using xa var_subterm_trmssst_is_varssst[of x "unlabel 𝒜"] vars_iff_subtermeq[of x]
      by auto
    hence "∃a. Γv x = TAtom a"
      using reachable_constraints_vars_TAtom_typed[OF 𝒜_reach P] by blast
    hence "∃a. Γ (Var x) = TAtom a"
      by simp
    hence "∃a. Γ (Fun f T) = TAtom a"
      by (metis (no_types, opaque_lifting) ℐ' welltyped_constraint_model_def fT_p wtsubst_def)
    ultimately show "(∃f. ℐ x = Fun f []) ∨ (∃y. ℐ x = Var y)"
      using TAtom_term_cases fT_p by metis
  qed

  have "∀T∈set P. bvars_transaction T = {}"
```

```
    using assms admissible_transactionE(4) by metis
  then have "bvars_lsst 𝒜 = {}"
    using reachable_constraints_no_bvars assms by metis
  then have 2: "bvars_lsst 𝒜 ∩ subst_domain ℐ = {}"
    by auto

  show ?thesis
    using subterms_subst_lsst[OF _ 2] 1
    by simp
qed

lemma reachable_constraints_val_funs_private':
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction_terms T"
           "∀T ∈ set P. transaction_decl T () = []"
           "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
    and f: "f ∈ ⋃(funs_term ` trms_lsst 𝒜)"
  shows "¬is_PubConstValue f"
    and "¬is_Abs f"
proof -
  have "¬is_PubConstValue f ∧ ¬is_Abs f" using 𝒜_reach f
  proof (induction 𝒜 rule: reachable_constraints.induct)
    case (step 𝒜 T ξ σ α)
    let ?T' = "unlabel (transaction_strand T) ·_sst ξ ∘_s σ ∘_s α"
    let ?T'' = "transaction_strand T ·_lsst ξ ∘_s σ ∘_s α"

    note ξ_empty =
      admissible_transaction_decl_subst_empty'[OF bspec[OF P(2) step.hyps(2)] step.hyps(3)]

    have T: "admissible_transaction_terms T"
      using P(1) step.hyps(2) by metis

    have T_fresh: "∀x ∈ set (transaction_fresh T). fst x = TAtom Value" when "T ∈ set P" for T
      using P that admissible_transactionE(14) unfolding list_all_iff Γ_v_TAtom'' by fast

    show ?thesis using step
    proof (cases "f ∈ ⋃(funs_term ` trms_lsst 𝒜)")
      case False
      then obtain t where t: "t ∈ trms_sst ?T'" "f ∈ funs_term t"
        using step.prems trms_sst_unlabel_dual_lsst_eq[of ?T'']
              trms_sst_append[of "unlabel 𝒜" "unlabel (dual_lsst ?T'')"]
              unlabel_append[of 𝒜 "dual_lsst ?T''"] unlabel_subst[of "transaction_strand T"]
        by fastforce
      show ?thesis using trms_sst_funs_term_cases[OF t]
      proof
        assume "∃u ∈ trms_transaction T. f ∈ funs_term u"
        thus ?thesis
          using conjunct1[OF conjunct2[OF T[unfolded admissible_transaction_terms_def]]]
          unfolding is_PubConstValue_def by blast
      next
        assume "∃x ∈ fv_transaction T. f ∈ funs_term ((ξ ∘_s σ ∘_s α) x)"
        then obtain x where "x ∈ fv_transaction T" "f ∈ funs_term ((ξ ∘_s σ ∘_s α) x)" by force
        thus ?thesis
          using transaction_decl_fresh_renaming_substs_range'(3)[
                  OF step.hyps(3-5) _ ξ_empty T_fresh[OF step.hyps(2), unfolded Γ_v_TAtom''(2)]]
          unfolding is_PubConstValue_def
          by (metis (no_types, lifting) funs_term_Fun_subterm prot_fun.disc(30,48) subst_imgI
                subtermeq_Var_const(2) term.distinct(1) term.inject(2) term.set_cases(1))
      qed
    qed simp
  qed simp
  thus "¬is_PubConstValue f" "¬is_Abs f" by simp_all
```

```
qed

lemma reachable_constraints_val_funs_private:
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and f: "f ∈ ⋃(funs_term ` trms_lsst 𝒜)"
  shows "¬is_PubConstValue f"
    and "¬is_Abs f"
using P reachable_constraints_val_funs_private'[OF 𝒜_reach _ _ _ f]
      admissible_transaction_is_wellformed_transaction(4)
      admissible_transactionE(1,14)
unfolding list_all_iff Γ_v_TAtom''
by (blast,fast)

lemma reachable_constraints_occurs_fact_ik_case:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and occ: "occurs t ∈ ik_lsst A"
  shows "∃n. t = Fun (Val n) []"
using 𝒜_reach occ
proof (induction A rule: reachable_constraints.induct)
  case (step A T ξ σ α)
  define ϑ where "ϑ ≡ ξ ∘s σ ∘s α"

  have T_adm: "admissible_transaction' T" using P step.hyps(2) by blast
  hence T: "wellformed_transaction T" "admissible_transaction_occurs_checks T"
    using admissible_transaction_is_wellformed_transaction(1) P_occ step.hyps(2) by (blast,blast)

  have T_fresh: "∀x ∈ set (transaction_fresh T). fst x = TAtom Value"
    using admissible_transactionE(14)[OF T_adm] unfolding list_all_iff by fast

  note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm step.hyps(3)]

  have ξ_dom_empty: "z ∉ fst ` set (transaction_decl T ())" for z
    using transaction_decl_subst_empty_inv[OF step.hyps(3)[unfolded ξ_empty]] by simp

  show ?case
  proof (cases "occurs t ∈ ik_lsst A")
    case False
    hence "occurs t ∈ ik_lsst (dual_lsst (transaction_strand T ·_lsst ϑ))"
      using step.prems unfolding ϑ_def by simp
    hence "∃ts. occurs t ∈ set ts ∧
             receive⟨ts⟩ ∈ set (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ)))"
      unfolding ik_sst_def by force
    hence "∃ts. occurs t ∈ set ts ∧
             send⟨ts⟩ ∈ set (unlabel (transaction_strand T ·_lsst ϑ))"
      using dual_lsst_unlabel_steps_iff(1) by blast
    then obtain ts s where s:
        "s ∈ set ts" "send⟨ts⟩ ∈ set (unlabel (transaction_strand T))" "s · ϑ = occurs t"
      using stateful_strand_step_mem_substD(1)[of _ "unlabel (transaction_strand T)" ϑ]
            unlabel_subst[of "transaction_strand T" ϑ]
      by force

    note 0 = transaction_decl_fresh_renaming_substs_range[OF step.hyps(3-5)]

    have 1: "send⟨ts⟩ ∈ set (unlabel (transaction_send T))"
      using s(2) wellformed_transaction_strand_unlabel_memberD(8)[OF T(1)] by blast

    have 2: "is_Send (send⟨ts⟩)"
      unfolding is_Send_def by simp
```

```
    have 3: "∃u. s = occurs u"
    proof -
      { fix z
        have "(∃n. ϑ z = Fun (Val n) []) ∨ (∃y. ϑ z = Var y)"
          using 0(3,4) T_fresh ξ_dom_empty unfolding ϑ_def by blast
        hence "∄u. ϑ z = occurs u" "ϑ z ≠ Fun OccursSec []" by auto
      } note * = this

      obtain u u' where T: "s = Fun OccursFact [u,u']"
        using *(1) s(3) by (cases s) auto
      thus ?thesis using *(2) s(3) by (cases u) auto
    qed

    obtain x where x: "x ∈ set (transaction_fresh T)" "s = occurs (Var x)"
      using 3 s(1) admissible_transaction_occurs_checksE4[OF T(2) 1] by metis

    have "t = ϑ x"
      using s(3) x(2) by auto
    thus ?thesis
      using 0(3)[OF ξ_dom_empty x(1)] x(1) T_fresh unfolding ϑ_def by fast
  qed (simp add: step.IH)
qed simp

lemma reachable_constraints_occurs_fact_send_ex:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and x: "Γᵥ x = TAtom Value" "x ∈ fv_lsst A"
  shows "∃ts. occurs (Var x) ∈ set ts ∧ send⟨ts⟩ ∈ set (unlabel A)"
using 𝒜_reach x(2)
proof (induction A rule: reachable_constraints.induct)
  case (step A T ξ σ α)
  note ξ_empty = admissible_transaction_decl_subst_empty[OF bspec[OF P step.hyps(2)] step.hyps(3)]
  note T = bspec[OF P_occ step.hyps(2)]

  show ?case
  proof (cases "x ∈ fv_lsst A")
    case True
    show ?thesis
      using step.IH[OF True] unlabel_append[of A]
      by auto
  next
    case False
    then obtain y where y:
        "y ∈ fv_transaction T - set (transaction_fresh T)" "(ξ ∘s σ ∘s α) y = Var x"
      using transaction_decl_fresh_renaming_substs_fv[OF step.hyps(3-5), of x]
            step.prems(1) fv_sst_append[of "unlabel A"] unlabel_append[of A]
      by auto

    have "σ y = Var y" using y(1) step.hyps(4) unfolding transaction_fresh_subst_def by auto
    hence "α y = Var x" using y(2) unfolding subst_compose_def ξ_empty by simp
    hence y_val: "fst y = TAtom Value" "Γᵥ y = TAtom Value"
      using x(1) Γᵥ_TAtom''[of x] Γᵥ_TAtom''[of y]
            wt_subst_trm''[OF transaction_renaming_subst_wt[OF step.hyps(5)], of "Var y"]
      by force+

    obtain ts where ts:
        "occurs (Var y) ∈ set ts" "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T))"
      using admissible_transaction_occurs_checksE1[OF T y(1) y_val(2)]
      by (metis list.set_intros(1) unlabel_Cons(1))
    hence "receive⟨ts⟩ ∈ set (unlabel (transaction_strand T))"
```

```
      using transaction_strand_subsets(5) by blast
    hence *: "receive⟨ts ·list ξ ∘s σ ∘s α⟩ ∈ set (unlabel (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
      using unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
            stateful_strand_step_subst_inI(2)[of _ _ "ξ ∘s σ ∘s α"]
      by force

    have "occurs (Var y) · ξ ∘s σ ∘s α = occurs (Var x)"
      using y(2) by (auto simp del: subst_subst_compose)
    hence **: "occurs (Var x) ∈ set ts ·set ξ ∘s σ ∘s α" using ts(1) by force

    have "send⟨ts ·list ξ ∘s σ ∘s α⟩ ∈ set (unlabel (duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)))"
      using * duallsst_unlabel_steps_iff(2) by blast
    thus ?thesis using ** unlabel_append[of A] by force
  qed
qed simp

lemma reachable_constraints_dblsst_set_args_empty:
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
    and PP: "list_all wellformed_transaction P"
    and admissible_transaction_updates:
      "let f = (λT. ∀x ∈ set (unlabel (transaction_updates T)).
                      is_Update x ∧ is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
                      fst (the_Var (the_elem_term x)) = TAtom Value)
      in list_all f P"
    and d: "(t, s) ∈ set (dblsst 𝒜 ℐ)"
  shows "∃ss. s = Fun (Set ss) []"
  using 𝒜 d
proof (induction)
  case (step 𝒜 TT ξ σ α)
  let ?TT = "transaction_strand TT ·lsst ξ ∘s σ ∘s α"
  let ?TTu = "unlabel ?TT"
  let ?TTd = "duallsst ?TT"
  let ?TTdu = "unlabel ?TTd"

  from step(6) have "(t, s) ∈ set (db'sst ?TTdu ℐ (db'sst (unlabel 𝒜) ℐ []))"
    by (metis dbsst_append dbsst_def step.prems unlabel_append)
  hence "(t, s) ∈ set (db'sst (unlabel 𝒜) ℐ []) ∨
    (∃t' s'. insert⟨t',s'⟩ ∈ set ?TTdu ∧ t = t' · ℐ ∧ s = s' · ℐ)"
    using dbsst_in_cases[of t "s" ?TTdu ℐ] by metis
  thus ?case
  proof
    assume "∃t' s'. insert⟨t',s'⟩ ∈ set ?TTdu ∧ t = t' · ℐ ∧ s = s' · ℐ"
    then obtain t' s' where t's'_p: "insert⟨t',s'⟩ ∈ set ?TTdu" "t = t' · ℐ" "s = s' · ℐ" by metis
    then obtain lll where "(lll, insert⟨t',s'⟩) ∈ set ?TTd" by (meson unlabel_mem_has_label)
    hence "(lll, insert⟨t',s'⟩) ∈ set (transaction_strand TT ·lsst ξ ∘s σ ∘s α)"
      using duallsst_steps_iff(4) by blast
    hence "insert⟨t',s'⟩ ∈ set ?TTu" by (meson unlabel_in)
    hence "insert⟨t',s'⟩ ∈ set ((unlabel (transaction_strand TT)) ·sst ξ ∘s σ ∘s α)"
      by (simp add: subst_lsst_unlabel)
    hence "insert⟨t',s'⟩ ∈ (λx. x ·sstp ξ ∘s σ ∘s α) ` set (unlabel (transaction_strand TT))"
      unfolding subst_apply_stateful_strand_def by auto
    then obtain u where
        "u ∈ set (unlabel (transaction_strand TT)) ∧ u ·sstp ξ ∘s σ ∘s α = insert⟨t',s'⟩"
      by auto
    hence "∃t'' s''. insert⟨t'',s''⟩ ∈ set (unlabel (transaction_strand TT)) ∧
                  t' = t'' · ξ ∘s σ ∘s α ∧ s' = s'' · ξ ∘s σ ∘s α"
      by  (cases u) auto
    then obtain t'' s'' where t''s''_p:
        "insert⟨t'',s''⟩ ∈ set (unlabel (transaction_strand TT)) ∧
          t' = t'' · ξ ∘s σ ∘s α ∧ s' = s'' · ξ ∘s σ ∘s α"
      by auto
    hence "insert⟨t'',s''⟩ ∈ set (unlabel (transaction_updates TT))"
      using is_Update_in_transaction_updates[of "insert⟨t'',s''⟩" TT]
```

```
      using PP step(2) unfolding list_all_iff by auto
    moreover have "∀x∈set (unlabel (transaction_updates TT)). is_Fun_Set (the_set_term x)"
      using step(2) admissible_transaction_updates unfolding is_Fun_Set_def list_all_iff by auto
    ultimately have "is_Fun_Set (the_set_term (insert⟨t'',s''⟩))" by auto
    moreover have "s' = s'' · ξ ∘s σ ∘s α" using t''s''_p by blast
    ultimately have "is_Fun_Set (the_set_term (insert⟨t',s'⟩))" by (auto simp add: is_Fun_Set_subst)
    hence "is_Fun_Set s" by (simp add: t's'_p(3) is_Fun_Set_subst)
    thus ?case using is_Fun_Set_exi by auto
  qed (auto simp add: step db_sst_def)
qed auto

lemma reachable_constraints_occurs_fact_ik_ground:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and t: "occurs t ∈ ik_lsst A"
  shows "fv (occurs t) = {}"
proof -
  have 0: "admissible_transaction' T"
    when "T ∈ set P" for T
    using P that unfolding list_all_iff by simp

  note 1 = admissible_transaction_is_wellformed_transaction(1)[OF 0] bspec[OF P_occ]

  have 2: "ik_lsst (A@dual_lsst (transaction_strand T ·lsst ϑ)) =
           (ik_lsst A) ∪ (trms_lsst (transaction_send T) ·set ϑ)"
    when "T ∈ set P" for T ϑ and A::"('fun,'atom,'sets,'lbl) prot_constr"
    using dual_transaction_ik_is_transaction_send'[OF 1(1)[OF that]] by fastforce

  show ?thesis using 𝒜_reach t
  proof (induction A rule: reachable_constraints.induct)
    case (step A T ξ σ α)
    note ξ_empty = admissible_transaction_decl_subst_empty[OF 0[OF step.hyps(2)] step.hyps(3)]

    from step show ?case
    proof (cases "occurs t ∈ ik_lsst A")
      case False
      hence "occurs t ∈ trms_lsst (transaction_send T) ·set ξ ∘s σ ∘s α"
        using 2[OF step.hyps(2)] step.prems ξ_empty by blast
      then obtain ts where ts:
          "occurs t ∈ set ts" "send⟨ts⟩ ∈ set (unlabel (transaction_send T ·lsst ξ ∘s σ ∘s α))"
        using wellformed_transaction_send_receive_subst_trm_cases(2)[OF 1(1)[OF step.hyps(2)]]
        by blast
      then obtain ts' s where s:
          "occurs s ∈ set ts'" "send⟨ts'⟩ ∈ set (unlabel (transaction_send T))" "t = s · ξ ∘s σ ∘s α"
        using transaction_decl_fresh_renaming_substs_occurs_fact_send_receive(1)[
                OF step.hyps(3-5) 0[OF step.hyps(2)] ts(1)]
              transaction_strand_subst_subsets(8)[of T "ξ ∘s σ ∘s α"]
        by blast

      obtain x where x: "x ∈ set (transaction_fresh T)" "s = Var x"
        using admissible_transaction_occurs_checksE4[OF 1(2)[OF step.hyps(2)] s(2,1)] by metis

      have "fv t = {}"
        using transaction_decl_fresh_renaming_substs_range(2)[OF step.hyps(3-5) _ x(1)]
              s(3) x(2) transaction_decl_subst_empty_inv[OF step.hyps(3)[unfolded ξ_empty]]
        by (auto simp del: subst_subst_compose)
      thus ?thesis by simp
    qed simp
  qed simp
qed
```

```
lemma reachable_constraints_occurs_fact_ik_funs_terms:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
  shows "∀s ∈ subterms_set (ik_lsst A ·_set I). OccursFact ∉ ⋃(funs_term ` set (snd (Ana s)))" (is "?A A")
    and "∀s ∈ subterms_set (ik_lsst A ·_set I). OccursSec ∉ ⋃(funs_term ` set (snd (Ana s)))" (is "?B A")
    and "Fun OccursSec [] ∉ ik_lsst A ·_set I" (is "?C A")
    and "∀x ∈ vars_lsst A. I x ≠ Fun OccursSec []" (is "?D A")
proof -
  have T_adm: "admissible_transaction' T" when "T ∈ set P" for T
    using P that unfolding list_all_iff by simp

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

  note T_occ = bspec[OF P_occ]

  note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm]

  have ℐ_wt: "wt_subst I" by (metis ℐ welltyped_constraint_model_def)

  have ℐ_wf_trms: "wf_trms (subst_range I)"
    by (metis ℐ welltyped_constraint_model_def constraint_model_def)

  have ℐ_grounds: "fv (I x) = {}" "∃f T. I x = Fun f T" for x
    using ℐ interpretation_grounds[of I, of "Var x"] empty_fv_exists_fun[of "I x"]
    unfolding welltyped_constraint_model_def constraint_model_def by auto

  have 00: "fv_set (trms_lsst (transaction_send T)) ⊆ vars_transaction T"
          "fv_set (subterms_set (trms_lsst (transaction_send T))) = fv_set (trms_lsst (transaction_send T))"
    for T::"('fun,'atom,'sets,'lbl) prot_transaction"
    using fv_trms_sst_subset(1)[of "unlabel (transaction_send T)"] vars_transaction_unfold
          fv_subterms_set[of "trms_lsst (transaction_send T)"]
    by blast+

  have 0: "∀x ∈ fv_set (trms_lsst (transaction_send T)). ∃a. Γ (Var x) = TAtom a"
         "∀x ∈ fv_set (trms_lsst (transaction_send T)). Γ (Var x) ≠ TAtom OccursSecType"
         "∀x ∈ fv_set (subterms_set (trms_lsst (transaction_send T))). ∃a. Γ (Var x) = TAtom a"
         "∀x ∈ fv_set (subterms_set (trms_lsst (transaction_send T))). Γ (Var x) ≠ TAtom OccursSecType"
         "∀x ∈ vars_transaction T. ∃a. Γ (Var x) = TAtom a"
         "∀x ∈ vars_transaction T. Γ (Var x) ≠ TAtom OccursSecType"
    when "T ∈ set P" for T
    using admissible_transaction_occurs_fv_types[OF T_adm[OF that]] 00
    by blast+

  note T_fresh_type = admissible_transactionE(2)[OF T_adm]

  have 1: "ik_lsst (A@dual_lsst (transaction_strand T ·_lsst ϑ)) ·_set I =
          (ik_lsst A ·_set I) ∪ (trms_lsst (transaction_send T) ·_set ϑ ·_set I)"
    when "T ∈ set P" for T ϑ and A::"('fun,'atom,'sets,'lbl) prot_constr"
    using dual_transaction_ik_is_transaction_send'[OF T_wf[OF that]]
    by fastforce

  have 2: "subterms_set (trms_lsst (transaction_send T) ·_set ϑ ·_set I) =
          subterms_set (trms_lsst (transaction_send T)) ·_set ϑ ·_set I"
    when "T ∈ set P" and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)" for T ϑ
    using wt_subst_TAtom_subterms_set_subst[OF wt_subst_compose[OF ϑ(1) ℐ_wt] 0(1)[OF that(1)]]
          wf_trm_subst_rangeD[OF wf_trms_subst_compose[OF ϑ(2) ℐ_wf_trms]]
    by auto

  have 3: "wt_subst (ξ ∘_s σ ∘_s α)" "wf_trms (subst_range (ξ ∘_s σ ∘_s α))"
    when "T ∈ set P" "transaction_decl_subst ξ T"
```

```
      "transaction_fresh_subst σ T (trms_lsst A)" "transaction_renaming_subst α P (vars_lsst A)"
  for ξ σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
  using protocol_transaction_vars_TAtom_typed(3)[of T] P that(1)
        transaction_decl_fresh_renaming_substs_wt[OF that(2-4)]
        transaction_decl_fresh_renaming_substs_range_wf_trms[OF that(2-4)]
        wf_trms_subst_compose
  by simp_all

have 4: "∀s ∈ subterms_set (trms_lsst (transaction_send T)).
            OccursFact ∉ ⋃(funs_term ` set (snd (Ana s))) ∧
            OccursSec ∉ ⋃(funs_term ` set (snd (Ana s)))"
  when T: "T ∈ set P" for T
proof
  fix t assume t: "t ∈ subterms_set (trms_lsst (transaction_send T))"
  then obtain ts s where s:
      "send⟨ts⟩ ∈ set (unlabel (transaction_send T))" "s ∈ set ts" "t ∈ subterms s"
    using wellformed_transaction_unlabel_cases(4)[OF T_wf[OF T]]
    by fastforce

  have s_occ: "∃x. s = occurs (Var x)" when "OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t"
    using that s(1) subtermeq_imp_funs_term_subset[OF s(3)]
          admissible_transaction_occurs_checksE3[OF T_occ[OF T] _ s(2)]
    by blast

  obtain K T' where K: "Ana t = (K,T')" by force

  show "OccursFact ∉ ⋃(funs_term ` set (snd (Ana t))) ∧
        OccursSec ∉ ⋃(funs_term ` set (snd (Ana t)))"
  proof (rule ccontr)
    assume "¬(OccursFact ∉ ⋃(funs_term ` set (snd (Ana t))) ∧
              OccursSec ∉ ⋃(funs_term ` set (snd (Ana t))))"
    hence a: "OccursFact ∈ ⋃(funs_term ` set (snd (Ana t))) ∨
              OccursSec ∈ ⋃(funs_term ` set (snd (Ana t)))"
      by simp
    hence "OccursFact ∈ ⋃(funs_term ` set T') ∨ OccursSec ∈ ⋃(funs_term ` set T')"
      using K by simp
    hence "OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t"
      using Ana_subterm[OF K] funs_term_subterms_eq(1)[of t] by blast
    then obtain x where x: "t ∈ subterms (occurs (Var x))"
      using s(3) s_occ by blast
    thus False using a by fastforce
  qed
qed

have 5: "OccursFact ∉ ⋃(funs_term ` subst_range (ξ ∘s σ ∘s α))"
        "OccursSec ∉ ⋃(funs_term ` subst_range (ξ ∘s σ ∘s α))"
    when ξσα: "transaction_decl_subst ξ T" "transaction_fresh_subst σ T (trms_lsst A)"
              "transaction_renaming_subst α P (vars_lsst A)"
    and T: "T ∈ set P"
  for ξ σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
proof -
  have "OccursFact ∉ funs_term t" "OccursSec ∉ funs_term t"
    when "t ∈ subst_range (ξ ∘s σ ∘s α)" for t
    using transaction_decl_fresh_renaming_substs_range'(3)[
            OF ξσα that ξ_empty[OF T ξσα(1)] T_fresh_type[OF T]]
    by auto
  thus "OccursFact ∉ ⋃(funs_term ` subst_range (ξ ∘s σ ∘s α))"
       "OccursSec ∉ ⋃(funs_term ` subst_range (ξ ∘s σ ∘s α))"
    by blast+
qed
```

```
have 6: "I x ≠ Fun OccursSec []" "∄t. I x = occurs t" "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
  when T: "T ∈ set P"
    and ξσα: "transaction_decl_subst ξ T" "transaction_fresh_subst σ T (trms_lsst A)"
             "transaction_renaming_subst α P (vars_lsst A)"
    and x: "Var x ∈ trms_lsst (transaction_send T) ·_set ξ ∘_s σ ∘_s α"
  for x ξ σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
proof -
  obtain t where t: "t ∈ trms_lsst (transaction_send T)" "t · (ξ ∘_s σ ∘_s α) = Var x"
    using x by force
  then obtain y where y: "t = Var y" by (cases t) auto

  have "∃a. Γ t = TAtom a ∧ a ≠ OccursSecType"
    using 0(1,2)[OF T] t(1) y
    by force
  thus "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
    using wt_subst_trm''[OF 3(1)[OF T ξσα]] wt_subst_trm''[OF ℐ_wt] t(2)
    by (metis eval_term.simps(1))
  thus "I x ≠ Fun OccursSec []" "∄t. I x = occurs t"
    by auto
qed

have 7: "I x ≠ Fun OccursSec []" "∄t. I x = occurs t" "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
  when T: "T ∈ set P"
    and ξσα: "transaction_decl_subst ξ T" "transaction_fresh_subst σ T (trms_lsst A)"
             "transaction_renaming_subst α P (vars_lsst A)"
    and x: "x ∈ fv_set ((ξ ∘_s σ ∘_s α) ` vars_transaction T)"
  for x ξ σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
proof -
  obtain y where y: "y ∈ vars_transaction T" "x ∈ fv ((ξ ∘_s σ ∘_s α) y)"
    using x by auto
  hence y': "(ξ ∘_s σ ∘_s α) y = Var x"
    using transaction_decl_fresh_renaming_substs_range'(3)[
          OF ξσα _ ξ_empty[OF T ξσα(1)] T_fresh_type[OF T]]
    by (cases "(ξ ∘_s σ ∘_s α) y ∈ subst_range (ξ ∘_s σ ∘_s α)") force+

  have "∃a. Γ (Var y) = TAtom a ∧ a ≠ OccursSecType"
    using 0(5,6)[OF T] y
    by force
  thus "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
    using wt_subst_trm''[OF 3(1)[OF T ξσα]] wt_subst_trm''[OF ℐ_wt] y'
    by (metis eval_term.simps(1))
  thus "I x ≠ Fun OccursSec []" "∄t. I x = occurs t"
    by auto
qed

have 8: "I x ≠ Fun OccursSec []" "∄t. I x = occurs t" "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
  when T: "T ∈ set P"
    and ξσα: "transaction_decl_subst ξ T" "transaction_fresh_subst σ T (trms_lsst A)"
             "transaction_renaming_subst α P (vars_lsst A)"
    and x: "Var x ∈ subterms_set (trms_lsst (transaction_send T)) ·_set ξ ∘_s σ ∘_s α"
  for x ξ σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
proof -
  obtain t where t: "t ∈ subterms_set (trms_lsst (transaction_send T))" "t · (ξ ∘_s σ ∘_s α) = Var x"
    using x by force
  then obtain y where y: "t = Var y" by (cases t) auto

  have "∃a. Γ t = TAtom a ∧ a ≠ OccursSecType"
    using 0(3,4)[OF T] t(1) y
    by force
  thus "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
```

```
      using wt_subst_trm''[OF 3(1)[OF T ξσα]] wt_subst_trm''[OF ℐ_wt] t(2)
      by (metis eval_term.simps(1))
    thus "I x ≠ Fun OccursSec []" "∄t. I x = occurs t"
      by auto
  qed

  have s_fv: "fv s ⊆ fv_set ((ξ ∘s σ ∘s α) ` vars_transaction T)"
    when s: "s ∈ subterms_set (trms_lsst (transaction_send T)) ·set ξ ∘s σ ∘s α"
      and T: "T ∈ set P"
    for s and ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
      and T::"('fun,'atom,'sets,'lbl) prot_transaction"
  proof
    fix x assume "x ∈ fv s"
    hence "x ∈ fv_set (subterms_set (trms_lsst (transaction_send T)) ·set ξ ∘s σ ∘s α)"
      using s by auto
    hence *: "x ∈ fv_set (trms_lsst (transaction_send T) ·set ξ ∘s σ ∘s α)"
      using fv_subterms_set_subst' by fast
    have **: "list_all is_Send (unlabel (transaction_send T))"
      using T_wf[OF T] unfolding wellformed_transaction_def by blast
    have "x ∈ fv_set ((ξ ∘s σ ∘s α) ` vars_lsst (transaction_send T))"
    proof -
      obtain t where t: "t ∈ trms_lsst (transaction_send T)" "x ∈ fv (t · ξ ∘s σ ∘s α)"
        using * by fastforce
      hence "fv t ⊆ vars_lsst (transaction_send T)"
        using fv_trms_sst_subset(1)[of "unlabel (transaction_send T)"]
        by auto
      thus ?thesis using t(2) subst_apply_fv_subset by fast
    qed
    thus "x ∈ fv_set ((ξ ∘s σ ∘s α) ` vars_transaction T)"
      using vars_transaction_unfold[of T] by fastforce
  qed

  show "?A A" using 𝒜_reach
  proof (induction A rule: reachable_constraints.induct)
    case (step A T ξ σ α)
    have *: "∀s ∈ subterms_set (trms_lsst (transaction_send T)).
              OccursFact ∉ ⋃(funs_term ` set (snd (Ana s)))"
      using 4[OF step.hyps(2)] by blast

    have "∀s ∈ subterms_set (trms_lsst (transaction_send T)) ·set ξ ∘s σ ∘s α ·set I.
            OccursFact ∉ ⋃(funs_term ` set (snd (Ana s)))"
    proof
      fix t assume t: "t ∈ subterms_set (trms_lsst (transaction_send T)) ·set ξ ∘s σ ∘s α ·set I"
      then obtain s u where su:
          "s ∈ subterms_set (trms_lsst (transaction_send T)) ·set ξ ∘s σ ∘s α" "s · I = t"
          "u ∈ subterms_set (trms_lsst (transaction_send T))" "u · ξ ∘s σ ∘s α = s"
        by force

      obtain Ku Tu where KTu: "Ana u = (Ku,Tu)" by force

      have *: "OccursFact ∉ ⋃(funs_term ` set Tu)"
              "OccursFact ∉ ⋃(funs_term ` subst_range (ξ ∘s σ ∘s α))"
              "OccursFact ∉ ⋃(funs_term ` ⋃(((set ∘ snd ∘ Ana) ` subst_range (ξ ∘s σ ∘s α))))"
        using transaction_decl_fresh_renaming_substs_range'(3)[
                OF step.hyps(3-5) _ ξ_empty[OF step.hyps(2,3)] T_fresh_type[OF step.hyps(2)]]
              4[OF step.hyps(2)] su(3) KTu
        by (fastforce,fastforce,fastforce)

      have "OccursFact ∉ ⋃(funs_term ` set (Tu ·list ξ ∘s σ ∘s α))"
      proof -
        { fix f assume f: "f ∈ ⋃(funs_term ` set (Tu ·list ξ ∘s σ ∘s α))"
          then obtain tf where tf: "tf ∈ set Tu" "f ∈ funs_term (tf · ξ ∘s σ ∘s α)" by force
          hence "f ∈ funs_term tf ∨ f ∈ ⋃(funs_term ` subst_range (ξ ∘s σ ∘s α))"
```

```
          using funs_term_subst[of tf "ξ ∘s σ ∘s α"] by force
        hence "f ≠ OccursFact" using *(1,2) tf(1) by blast
      } thus ?thesis by metis
    qed
    hence **: "OccursFact ∉ ⋃(funs_term ` set (snd (Ana s)))"
    proof (cases u)
      case (Var xu)
      hence "s = (ξ ∘s σ ∘s α) xu" using su(4) by (metis eval_term.simps(1))
      thus ?thesis using *(3) by fastforce
    qed (use su(4) KTu Ana_subst'[of _ _ Ku Tu "ξ ∘s σ ∘s α"] in simp)

    show "OccursFact ∉ ⋃(funs_term ` set (snd (Ana t)))"
    proof (cases s)
      case (Var sx)
      then obtain a where a: "Γ (I sx) = Var a"
        using su(1) 8(3)[OF step.hyps(2-5), of sx] by fast
      hence "Ana (I sx) = ([],[])" by (metis I_grounds(2) const_type_inv[THEN Ana_const])
      thus ?thesis using Var su(2) by simp
    next
      case (Fun f S)
      hence snd_Ana_t: "snd (Ana t) = snd (Ana s) ·list I"
        using su(2) Ana_subst'[of f S _ "snd (Ana s)" I] by (cases "Ana s") simp_all

      { fix g assume "g ∈ ⋃(funs_term ` set (snd (Ana t)))"
        hence "g ∈ ⋃(funs_term ` set (snd (Ana s))) ∨
               (∃x ∈ fvset (set (snd (Ana s))). g ∈ funs_term (I x))"
          using snd_Ana_t funs_term_subst[of _ I] by auto
        hence "g ≠ OccursFact"
        proof
          assume "∃x ∈ fvset (set (snd (Ana s))). g ∈ funs_term (I x)"
          then obtain x where x: "x ∈ fvset (set (snd (Ana s)))" "g ∈ funs_term (I x)" by force
          have "x ∈ fv s" using x(1) Ana_vars(2)[of s] by (cases "Ana s") auto
          hence "x ∈ fvset ((ξ ∘s σ ∘s α) ` vars_transaction T)"
            using s_fv[OF su(1) step.hyps(2)] by blast
          then obtain a h U where h:
              "I x = Fun h U" "Γ (I x) = Var a" "a ≠ OccursSecType" "arity h = 0"
            using I_grounds(2) 7(3)[OF step.hyps(2-5)] const_type_inv
            by metis
          hence "h ≠ OccursFact" by auto
          moreover have "U = []" using h(1,2,4) const_type_inv_wf[of h U a] I_wftrms by fastforce
          ultimately show ?thesis using h(1) x(2) by auto
        qed (use ** in blast)
      } thus ?thesis by blast
    qed
  qed
  thus ?case
    using step.IH step.prems 1[OF step.hyps(2), of A "ξ ∘s σ ∘s α"]
          2[OF step.hyps(2) 3[OF step.hyps(2-5)]]
    by auto
qed simp

show "?B A" using A_reach
proof (induction A rule: reachable_constraints.induct)
  case (step A T ξ σ α)
  have "∀s ∈ subtermsset (trmslsst (transaction_send T)) ·set ξ ∘s σ ∘s α ·set I.
          OccursSec ∉ ⋃(funs_term ` set (snd (Ana s)))"
  proof
    fix t assume t: "t ∈ subtermsset (trmslsst (transaction_send T)) ·set ξ ∘s σ ∘s α ·set I"
    then obtain s u where su:
        "s ∈ subtermsset (trmslsst (transaction_send T)) ·set ξ ∘s σ ∘s α" "s · I = t"
        "u ∈ subtermsset (trmslsst (transaction_send T))" "u · ξ ∘s σ ∘s α = s"
      by force
```

```
  obtain Ku Tu where KTu: "Ana u = (Ku,Tu)" by force

  have *: "OccursSec ∉ ⋃(funs_term ` set Tu)"
          "OccursSec ∉ ⋃(funs_term ` subst_range (ξ ∘s σ ∘s α))"
          "OccursSec ∉ ⋃(funs_term ` ⋃(((set ∘ snd ∘ Ana) ` subst_range (ξ ∘s σ ∘s α))))"
    using transaction_decl_fresh_renaming_substs_range'(3)[
            OF step.hyps(3-5) _ ξ_empty[OF step.hyps(2,3)] T_fresh_type[OF step.hyps(2)]]
          4[OF step.hyps(2)] su(3) KTu
    by (fastforce,fastforce,fastforce)

  have "OccursSec ∉ ⋃(funs_term ` set (Tu ·list ξ ∘s σ ∘s α))"
  proof -
    { fix f assume f: "f ∈ ⋃(funs_term ` set (Tu ·list ξ ∘s σ ∘s α))"
      then obtain tf where tf: "tf ∈ set Tu" "f ∈ funs_term (tf · ξ ∘s σ ∘s α)" by force
      hence "f ∈ funs_term tf ∨ f ∈ ⋃(funs_term ` subst_range (ξ ∘s σ ∘s α))"
        using funs_term_subst[of tf "ξ ∘s σ ∘s α"] by force
      hence "f ≠ OccursSec" using *(1,2) tf(1) by blast
    } thus ?thesis by metis
  qed
  hence **: "OccursSec ∉ ⋃(funs_term ` set (snd (Ana s)))"
  proof (cases u)
    case (Var xu)
    hence "s = (ξ ∘s σ ∘s α) xu" using su(4) by (metis eval_term.simps(1))
    thus ?thesis using *(3) by fastforce
  qed (use su(4) KTu Ana_subst'[of _ _ Ku Tu "ξ ∘s σ ∘s α"] in simp)

  show "OccursSec ∉ ⋃(funs_term ` set (snd (Ana t)))"
  proof (cases s)
    case (Var sx)
    then obtain a where a: "Γ (I sx) = Var a"
      using su(1) 8(3)[OF step.hyps(2-5), of sx] by fast
    hence "Ana (I sx) = ([],[])" by (metis I_grounds(2) const_type_inv[THEN Ana_const])
    thus ?thesis using Var su(2) by simp
  next
    case (Fun f S)
    hence snd_Ana_t: "snd (Ana t) = snd (Ana s) ·list I"
      using su(2) Ana_subst'[of f S _ "snd (Ana s)" I] by (cases "Ana s") simp_all

    { fix g assume "g ∈ ⋃(funs_term ` set (snd (Ana t)))"
      hence "g ∈ ⋃(funs_term ` set (snd (Ana s))) ∨
             (∃x ∈ fvset (set (snd (Ana s))). g ∈ funs_term (I x))"
        using snd_Ana_t funs_term_subst[of _ I] by auto
      hence "g ≠ OccursSec"
      proof
        assume "∃x ∈ fvset (set (snd (Ana s))). g ∈ funs_term (I x)"
        then obtain x where x: "x ∈ fvset (set (snd (Ana s)))" "g ∈ funs_term (I x)" by force
        have "x ∈ fv s" using x(1) Ana_vars(2)[of s] by (cases "Ana s") auto
        hence "x ∈ fvset ((ξ ∘s σ ∘s α) ` vars_transaction T)"
          using s_fv[OF su(1) step.hyps(2)] by blast
        then obtain a h U where h:
            "I x = Fun h U" "Γ (I x) = Var a" "a ≠ OccursSecType" "arity h = 0"
          using I_grounds(2) 7(3)[OF step.hyps(2-5)] const_type_inv
          by metis
        hence "h ≠ OccursSec" by auto
        moreover have "U = []" using h(1,2,4) const_type_inv_wf[of h U a] I_wftrms by fastforce
        ultimately show ?thesis using h(1) x(2) by auto
      qed (use ** in blast)
    } thus ?thesis by blast
  qed
qed
thus ?case
  using step.IH step.prems 1[OF step.hyps(2), of A "ξ ∘s σ ∘s α"]
        2[OF step.hyps(2) 3[OF step.hyps(2-5)]]
```

```
      by auto
  qed simp

  show "?C A" using 𝒜_reach
  proof (induction A rule: reachable_constraints.induct)
    case (step A T ξ σ α)
    have *: "Fun OccursSec [] ∉ trms_lsst (transaction_send T)"
      using admissible_transaction_occurs_checksE5[OF T_occ[OF step.hyps(2)]] by blast

    have **: "Fun OccursSec [] ∉ subst_range (ξ ∘s σ ∘s α)"
      using transaction_decl_fresh_renaming_substs_range'(3)[
              OF step.hyps(3-5) _ ξ_empty[OF step.hyps(2,3)] T_fresh_type[OF step.hyps(2)]]
      by auto

    have "Fun OccursSec [] ∉ trms_lsst (transaction_send T) ·set ξ ∘s σ ∘s α ·set I"
    proof
      assume "Fun OccursSec [] ∈ trms_lsst (transaction_send T) ·set ξ ∘s σ ∘s α ·set I"
      then obtain s where "s ∈ trms_lsst (transaction_send T) ·set ξ ∘s σ ∘s α" "s · I = Fun OccursSec
[]"
        by force
      moreover have "Fun OccursSec [] ∉ trms_lsst (transaction_send T) ·set ξ ∘s σ ∘s α"
      proof
        assume "Fun OccursSec [] ∈ trms_lsst (transaction_send T) ·set ξ ∘s σ ∘s α"
        then obtain u where "u ∈ trms_lsst (transaction_send T)" "u · ξ ∘s σ ∘s α = Fun OccursSec []"
          by force
        thus False using * ** by (cases u) (force simp del: subst_subst_compose)+
      qed
      ultimately show False using 6[OF step.hyps(2-5)] by (cases s) auto
    qed
    thus ?case using step.IH step.prems 1[OF step.hyps(2), of A "ξ ∘s σ ∘s α"] by fast
  qed simp

  show "?D A" using 𝒜_reach
  proof (induction A rule: reachable_constraints.induct)
    case (step A T ξ σ α)
    { fix x assume x: "x ∈ vars_lsst (dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
      hence x': "x ∈ vars_sst (unlabel (transaction_strand T) ·sst ξ ∘s σ ∘s α)"
        by (metis vars_sst_unlabel_dual_lsst_eq unlabel_subst)
      hence "x ∈ vars_transaction T ∨ x ∈ fv_set ((ξ ∘s σ ∘s α) ` vars_transaction T)"
        using vars_sst_subst_cases[OF x'] by metis
      moreover have "I x ≠ Fun OccursSec []" when "x ∈ vars_transaction T"
        using that 0(5,6)[OF step.hyps(2)] wt_subst_trm''[OF ℐ_wt, of "Var x"]
        by fastforce
      ultimately have "I x ≠ Fun OccursSec []"
        using 7(1)[OF step.hyps(2-5), of x]
        by blast
    } thus ?case using step.IH by auto
  qed simp
qed

lemma reachable_constraints_occurs_fact_ik_subst_aux:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and t: "t ∈ ik_lsst A" "t · I = occurs s"
  shows "∃u. t = occurs u"
proof -
  have "wt_subst I"
    using ℐ unfolding welltyped_constraint_model_def constraint_model_def by metis
  hence 0: "Γ t = Γ (occurs s)"
    using t(2) wt_subst_trm'' by metis
```

```
  have 1: "Γᵥ ` fv_lsst A ⊆ (⋃T ∈ set P. Γᵥ ` fv_transaction T)"
          "∀T ∈ set P. ∀x ∈ fv_transaction T. Γᵥ x = TAtom Value ∨ (∃a. Γᵥ x = TAtom (Atom a))"
    using reachable_constraints_var_types_in_transactions(1)[OF 𝒜_reach]
          protocol_transaction_vars_TAtom_typed(2,3) P
    by fast+

  show ?thesis
  proof (cases t)
    case (Var x)
    thus ?thesis
      using 0 1 t(1) var_subterm_ik_sst_is_fv_sst[of x "unlabel A"]
      by fastforce
  next
    case (Fun f T)
    hence 2: "f = OccursFact" "length T = Suc (Suc 0)" "T ! 0 · I = Fun OccursSec []"
      using t(2) by auto

    have "T ! 0 = Fun OccursSec []"
    proof (cases "T ! 0")
      case (Var y)
      hence "I y = Fun OccursSec []" using Fun 2(3) by simp
      moreover have "Var y ∈ set T" using Var 2(2) length_Suc_conv[of T 1] by auto
      hence "y ∈ fv_set (ik_lsst A)" using Fun t(1) by force
      hence "y ∈ vars_lsst A"
        using fv_ik_subset_fv_sst'[of "unlabel A"] vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
        by blast
      ultimately have False
        using reachable_constraints_occurs_fact_ik_funs_terms(4)[OF 𝒜_reach ℐ P P_occ]
        by blast
      thus ?thesis by simp
    qed (use 2(3) in simp)
    moreover have "∃u u'. T = [u,u']"
      using iffD1[OF length_Suc_conv 2(2)] iffD1[OF length_Suc_conv[of _ 0]] length_0_conv by fast
    ultimately show ?thesis using Fun 2(1,2) by force
  qed
qed

lemma reachable_constraints_occurs_fact_ik_subst:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and t: "occurs t ∈ ik_lsst A ·set I"
  shows "occurs t ∈ ik_lsst A"
proof -
  have ℐ_wt: "wt_subst I"
    using ℐ unfolding welltyped_constraint_model_def constraint_model_def by metis

  obtain s where s: "s ∈ ik_lsst A" "s · I = occurs t"
    using t by auto
  hence u: "∃u. s = occurs u"
    using ℐ_wt reachable_constraints_occurs_fact_ik_subst_aux[OF 𝒜_reach ℐ P P_occ]
    by blast
  hence "fv s = {}"
    using reachable_constraints_occurs_fact_ik_ground[OF 𝒜_reach P P_occ] s
    by fast
  thus ?thesis
    using s u subst_ground_ident[of s I]
    by argo
qed

lemma reachable_constraints_occurs_fact_send_in_ik:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
```

```
    and 𝓘: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and x: "occurs (Var x) ∈ set ts" "send⟨ts⟩ ∈ set (unlabel A)"
  shows "occurs (I x) ∈ ik_lsst A"
using 𝓐_reach 𝓘 x
proof (induction A rule: reachable_constraints.induct)
  case (step A T ξ σ α)
  define ϑ where "ϑ ≡ ξ ∘s σ ∘s α"
  define T' where "T' ≡ dual_lsst (transaction_strand T ·lsst ϑ)"

  have T_adm: "admissible_transaction' T"
    using P step.hyps(2) unfolding list_all_iff by blast

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]
  note T_adm_occ = bspec[OF P_occ]

  have 𝓘_is_T_model: "strand_sem_stateful (ik_lsst A ·set I) (set (db_lsst A I)) (unlabel T') I"
    using step.prems unlabel_append[of A T'] db_sst_set_is_dbupd_sst[of "unlabel A" I "[]"]
          strand_sem_append_stateful[of "{}" "{}" "unlabel A" "unlabel T'" I]
    by (simp add: T'_def ϑ_def welltyped_constraint_model_def constraint_model_def db_sst_def)

  show ?case
  proof (cases "send⟨ts⟩ ∈ set (unlabel A)")
    case False
    hence "send⟨ts⟩ ∈ set (unlabel T')"
      using step.prems(3) unfolding T'_def ϑ_def by simp
    hence "receive⟨ts⟩ ∈ set (unlabel (transaction_strand T ·lsst ϑ))"
      using dual_lsst_unlabel_steps_iff(2) unfolding T'_def by blast
    then obtain y ts' where y:
        "receive⟨ts'⟩ ∈ set (unlabel (transaction_receive T))"
        "ϑ y = Var x" "occurs (Var y) ∈ set ts'"
      using transaction_decl_fresh_renaming_substs_occurs_fact_send_receive(2)[
              OF step.hyps(3-5) T_adm]
            subst_to_var_is_var[of _ ϑ x] step.prems(2)
      unfolding ϑ_def by (metis eval_term.simps(1))
    hence "occurs (Var y) · ϑ ∈ set ts' ·set ϑ"
          "receive⟨ts' ·list ϑ⟩ ∈ set (unlabel (transaction_receive T ·lsst ϑ))"
      using subst_lsst_unlabel_member[of "receive⟨ts'⟩" "transaction_receive T" ϑ]
      by fastforce+
    hence "ik_lsst A ·set I ⊢ occurs (Var y) · ϑ · I"
      using wellformed_transaction_sem_receives[
              OF T_wf, of "ik_lsst A ·set I" "set (db_lsst A I)" ϑ I "ts' ·list ϑ"]
            𝓘_is_T_model
      unfolding T'_def list_all_iff by fastforce
    hence *: "ik_lsst A ·set I ⊢ occurs (ϑ y · I)"
      by auto

    have "occurs (ϑ y · I) ∈ ik_lsst A"
      using deduct_occurs_in_ik[OF *]
            reachable_constraints_occurs_fact_ik_subst[
              OF step.hyps(1) welltyped_constraint_model_prefix[OF step.prems(1)] P P_occ,
              of "ϑ y · I"]
            reachable_constraints_occurs_fact_ik_funs_terms[
              OF step.hyps(1) welltyped_constraint_model_prefix[OF step.prems(1)] P P_occ]
      by blast
    thus ?thesis using y(2) by simp
  qed (simp add: step.IH[OF welltyped_constraint_model_prefix[OF step.prems(1)]] step.prems(2))
qed simp

lemma reachable_constraints_occurs_fact_deduct_in_ik:
  assumes 𝓐_reach: "A ∈ reachable_constraints P"
    and 𝓘: "welltyped_constraint_model I A"
```

```
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and k: "ik_lsst A ·set I ⊢ occurs k"
  shows "occurs k ∈ ik_lsst A ·set I"
    and "occurs k ∈ ik_lsst A"
using reachable_constraints_occurs_fact_ik_funs_terms(1-3)[OF 𝒜_reach ℐ P P_occ]
      reachable_constraints_occurs_fact_ik_subst[OF 𝒜_reach ℐ P P_occ]
      deduct_occurs_in_ik[OF k]
by (presburger, presburger)

lemma reachable_constraints_fv_bvars_subset:
  assumes A: "A ∈ reachable_constraints P"
  shows "bvars_lsst A ⊆ (⋃T ∈ set P. bvars_transaction T)"
using assms
proof (induction A rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  let ?T' = "transaction_strand T ·lsst ξ ∘s σ ∘s α"

  show ?case
    using step.IH step.hyps(2)
          bvars_sst_unlabel_dual_lsst_eq[of ?T']
          bvars_lsst_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
          bvars_sst_append[of "unlabel 𝒜" "unlabel (dual_lsst ?T')"]
          unlabel_append[of 𝒜 "dual_lsst ?T'"]
    by (metis (no_types, lifting) SUP_upper Un_subset_iff)
qed simp

lemma reachable_constraints_fv_disj:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes A: "A ∈ reachable_constraints P"
  shows "fv_lsst A ∩ (⋃T ∈ set P. bvars_transaction T) = {}"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  define T' where "T' ≡ transaction_strand T ·lsst ξ ∘s σ ∘s α"
  define X where "X ≡ ⋃T ∈ set P. bvars_transaction T"
  have "fv_lsst T' ∩ X = {}"
    using transaction_decl_fresh_renaming_substs_vars_disj(4)[OF step.hyps(3-5)]
          transaction_decl_fresh_renaming_substs_vars_subset(4)[OF step.hyps(3-5,2)]
    unfolding T'_def X_def by blast
  hence "fv_lsst (𝒜@dual_lsst T') ∩ X = {}"
    using step.IH[unfolded X_def[symmetric]] fv_sst_unlabel_dual_lsst_eq[of T'] by auto
  thus ?case unfolding T'_def X_def by blast
qed simp


lemma reachable_constraints_fv_bvars_disj':
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes P: "∀T ∈ set P. wellformed_transaction T"
    and A: "A ∈ reachable_constraints P"
  shows "fv_lsst A ∩ bvars_lsst A = {}"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  define T' where "T' ≡ dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"

  note 0 = transaction_decl_fresh_renaming_substs_vars_disj[OF step.hyps(3-5)]
  note 1 = transaction_decl_fresh_renaming_substs_vars_subset[OF step.hyps(3-5)]

  have 2: "bvars_lsst 𝒜 ∩ fv_lsst T' = {}"
    using 0(7) 1(4)[OF step.hyps(2)] fv_sst_unlabel_dual_lsst_eq
    unfolding T'_def by (metis (no_types) disjoint_iff_not_equal subset_iff)
```

```
  have "bvars_lsst T' ⊆ ⋃(bvars_transaction ` set P)"
       "fv_lsst 𝒜 ∩ ⋃(bvars_transaction ` set P) = {}"
    using reachable_constraints_fv_bvars_subset[OF reachable_constraints.step[OF step.hyps]]
          reachable_constraints_fv_disj[OF reachable_constraints.step[OF step.hyps]]
    unfolding T'_def by auto
  hence 3: "fv_lsst 𝒜 ∩ bvars_lsst T' = {}" by blast

  have "fv_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α) ∩ bvars_transaction T = {}"
    using 0(4)[OF step.hyps(2)] 1(4)[OF step.hyps(2)] by blast
  hence 4: "fv_lsst T' ∩ bvars_lsst T' = {}"
    by (metis (no_types) T'_def fv_sst_unlabel_dual_lsst_eq bvars_sst_unlabel_dual_lsst_eq
              unlabel_subst bvars_sst_subst)

  have "fv_lsst (𝒜@T') ∩ bvars_lsst (𝒜@T') = {}"
    using 2 3 4 step.IH
    unfolding unlabel_append[of 𝒜 T']
              fv_sst_append[of "unlabel 𝒜" "unlabel T'"]
              bvars_sst_append[of "unlabel 𝒜" "unlabel T'"]
    by fast
  thus ?case unfolding T'_def by blast
qed simp

lemma reachable_constraints_wf:
  assumes P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
    and A: "A ∈ reachable_constraints P"
  shows "wf_sst (unlabel A)"
    and "wf_trms (trms_lsst A)"
proof -
  let ?X = "λT. fst ` set (transaction_decl T ()) ∪ set (transaction_fresh T)"

  have "wellformed_transaction T"
    when "T ∈ set P" for T
    using P(1) that by fast+
  hence 0: "wf'_sst (?X T) (unlabel (dual_lsst (transaction_strand T)))"
          "fv_lsst (dual_lsst (transaction_strand T)) ∩ bvars_lsst (dual_lsst (transaction_strand T)) = {}"
          "wf_trms (trms_transaction T)"
    when T: "T ∈ set P" for T
    unfolding admissible_transaction_terms_def
    by (metis T wellformed_transaction_wf_sst(1),
        metis T wellformed_transaction_wf_sst(2) fv_sst_unlabel_dual_lsst_eq bvars_sst_unlabel_dual_lsst_eq,
        metis T wf_trms_code P(2))

  from A have "wf_sst (unlabel A) ∧ wf_trms (trms_lsst A)"
  proof (induction A rule: reachable_constraints.induct)
    case (step A T ξ σ α)
    let ?T' = "dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"

    have IH: "wf'_sst {} (unlabel A)" "fv_lsst A ∩ bvars_lsst A = {}" "wf_trms (trms_lsst A)"
      using step.IH by metis+

    have 1: "wf'_sst {} (unlabel (A@?T'))"
      using transaction_decl_fresh_renaming_substs_wf_sst[OF 0(1)[OF step.hyps(2)] step.hyps(3-5)]
            wf_sst_vars_mono[of "{}"] wf_sst_append[OF IH(1)]
      by simp

    have 2: "fv_lsst (A@?T') ∩ bvars_lsst (A@?T') = {}"
      using reachable_constraints_fv_bvars_disj'[OF P(1)]
            reachable_constraints.step[OF step.hyps]
      by blast

    have "wf_trms (trms_lsst ?T')"
```

```
        using trms_sst_unlabel_dual_lsst_eq unlabel_subst
              wf_trms_subst[
                OF transaction_decl_fresh_renaming_substs_range_wf_trms[OF step.hyps(3-5)],
                THEN wf_trms_trms_sst_subst,
                OF 0(3)[OF step.hyps(2)]]
        by metis
      hence 3: "wf_trms (trms_lsst (A@?T'))"
        using IH(3) by auto

      show ?case using 1 2 3 by force
    qed simp
    thus "wf_sst (unlabel A)" "wf_trms (trms_lsst A)" by metis+
qed

lemma reachable_constraints_no_Ana_attack:
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. wellformed_transaction T"
           "∀T ∈ set P. admissible_transaction_terms T"
           "∀T ∈ set P. ∀x ∈ set (transaction_fresh T).
              Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    and t: "t ∈ subterms_set (ik_lsst 𝒜)"
  shows "attack⟨n⟩ ∉ set (snd (Ana t))"
proof -
  have T_adm_term: "admissible_transaction_terms T" when "T ∈ set P" for T
    using P that by blast

  have T_wf: "wellformed_transaction T" when "T ∈ set P" for T
    using P that by blast

  have T_fresh: "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    when "T ∈ set P" for T
    using P(3) that by fast

  show ?thesis
  using 𝒜 t
  proof (induction 𝒜 rule: reachable_constraints.induct)
    case (step A T ξ σ α) thus ?case
    proof (cases "t ∈ subterms_set (ik_lsst A)")
      case False
      hence "t ∈ subterms_set (ik_lsst (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)))"
        using step.prems by simp
      hence "t ∈ subterms_set (trms_lsst (transaction_send T) ·_set ξ ∘_s σ ∘_s α)"
        using dual_transaction_ik_is_transaction_send'[OF T_wf[OF step.hyps(2)]]
        by metis
      hence "t ∈ subterms_set (trms_lsst (transaction_send T)) ·_set ξ ∘_s σ ∘_s α"
        using transaction_decl_fresh_renaming_substs_trms[
                OF step.hyps(3-5), of "transaction_send T"]
              wellformed_transaction_unlabel_cases(4)[OF T_wf[OF step.hyps(2)]]
        by fastforce
      then obtain s where s: "s ∈ subterms_set (trms_lsst (transaction_send T))" "t = s · ξ ∘_s σ ∘_s α"
        by force
      hence s': "attack⟨n⟩ ∉ set (snd (Ana s))"
        using admissible_transaction_no_Ana_Attack[OF T_adm_term[OF step.hyps(2)]]
              trms_transaction_unfold[of T]
        by blast

      note * = transaction_decl_fresh_renaming_substs_range'(1-3)[OF step.hyps(3-5)]
               transaction_decl_fresh_renaming_substs_range_no_attack_const[
                 OF step.hyps(3-5) T_fresh[OF step.hyps(2)]]

      show ?thesis
      proof
        assume n: "attack⟨n⟩ ∈ set (snd (Ana t))"
```

```
        thus False
        proof (cases s)
          case (Var x)
          hence "(∃c. t = Fun c []) ∨ (∃y. t = Var y)"
            using *(1)[of t] n s(2) by (force simp del: subst_subst_compose)
          thus ?thesis using n Ana_subterm' by fastforce
        next
          case (Fun f S)
          hence "attack⟨n⟩ ∈ set (snd (Ana s)) ·set ξ ∘s σ ∘s α"
            using Ana_subst'[of f S _ "snd (Ana s)" "ξ ∘s σ ∘s α"] s(2) s' n
            by (cases "Ana s") auto
          hence "attack⟨n⟩ ∈ set (snd (Ana s)) ∨ attack⟨n⟩ ∈ subst_range (ξ ∘s σ ∘s α)"
            using const_mem_subst_cases' by fast
          thus ?thesis using *(4) s' by fast
        qed
      qed
    qed simp
  qed simp
qed

lemma reachable_constraints_receive_attack_if_attack:
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. wellformed_transaction T"
           "∀T ∈ set P. admissible_transaction_terms T"
           "∀T ∈ set P. ∀x ∈ set (transaction_fresh T).
              Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
           "∀T ∈ set P. ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γv x"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and l: "iklsst 𝒜 ·set ℐ ⊢ attack⟨l⟩"
  shows "attack⟨l⟩ ∈ iklsst 𝒜 ·set ℐ"
    and "receive⟨[attack⟨l⟩]⟩ ∈ set (unlabel 𝒜)"
    and "∀T ∈ set P. ∀s ∈ set (transaction_strand T).
              is_Send (snd s) ∧ length (the_msgs (snd s)) = 1 ∧
              is_Fun_Attack (hd (the_msgs (snd s)))
              ⟶ the_Attack_label (the_Fun (hd (the_msgs (snd s)))) = fst s
          ⟹ (l,receive⟨[attack⟨l⟩]⟩) ∈ set 𝒜" (is "?Q ⟹ (l,receive⟨[attack⟨l⟩]⟩) ∈ set 𝒜")
proof -
  have ℐ': "constr_sem_stateful ℐ (unlabel 𝒜)" "interpretationsubst ℐ"
           "wftrms (subst_range ℐ)" "wtsubst ℐ"
    using ℐ unfolding welltyped_constraint_model_def constraint_model_def by metis+

  have 0: "wftrms (iklsst 𝒜 ·set ℐ)"
    when 𝒜: "𝒜 ∈ reachable_constraints P" for 𝒜
    using reachable_constraints_wftrms[OF _ 𝒜] admissible_transaction_terms_wftrms P(2)
          iksst_trmssst_subset[of "unlabel 𝒜"] wf_trms_subst[OF ℐ'(3)]
    by fast

  have 1: "∀x ∈ fvset (iklsst 𝒜). ¬TAtom AttackType ⊑ Γv x"
    when 𝒜: "𝒜 ∈ reachable_constraints P" for 𝒜
    using reachable_constraints_vars_not_attack_typed[OF 𝒜 P(3,4)]
          fv_ik_subset_vars_sst'[of "unlabel 𝒜"]
    by fast

  have 2: "attack⟨l⟩ ∉ set (snd (Ana t)) ·set ℐ" when t: "t ∈ subtermsset (iklsst 𝒜)" for t
  proof
    assume "attack⟨l⟩ ∈ set (snd (Ana t)) ·set ℐ"
    then obtain s where s: "s ∈ set (snd (Ana t))" "s · ℐ = attack⟨l⟩" by force

    obtain x where x: "s = Var x"
      by (cases s) (use s reachable_constraints_no_Ana_attack[OF 𝒜 P(1-3) t] in auto)

    have "x ∈ fv t" using x Ana_subterm'[OF s(1)] vars_iff_subtermeq by force
    hence "x ∈ fvset (iklsst 𝒜)" using t fv_subterms by fastforce
```

```
  hence "Γv x ≠ TAtom AttackType" using 1[OF 𝒜] by fastforce
  thus False using s(2) x wt_subst_trm''[OF ℐ'(4), of "Var x"] by fastforce
qed

have 3: "attack⟨l⟩ ∉ set (snd (Ana t))" when t: "t ∈ subtermsset (iklsst 𝒜 ·set ℐ)" for t
proof
  assume "attack⟨l⟩ ∈ set (snd (Ana t))"
  then obtain s where s:
      "s ∈ subtermsset (ℐ ` fvset (iklsst 𝒜))" "attack⟨l⟩ ∈ set (snd (Ana s))"
    using Ana_subst_subterms_cases[OF t] 2 by fast
  then obtain x where x: "x ∈ fvset (iklsst 𝒜)" "s ⊑ ℐ x" by force
  hence "ℐ x ∈ subtermsset (iklsst 𝒜 ·set ℐ)"
    using var_is_subterm[of x] subterms_subst_subset'[of ℐ "iklsst 𝒜"]
    by force
  hence *: "wftrm (ℐ x)" "wftrm s"
    using wf_trms_subterms[OF 0[OF 𝒜]] wf_trm_subtermeq[OF _ x(2)]
    by auto

  show False
    using term.order_trans[
            OF subtermeq_imp_subtermtypeeq[OF *(2) Ana_subterm'[OF s(2)]]
               subtermeq_imp_subtermtypeeq[OF *(1) x(2)]]
          1[OF 𝒜] x(1) wt_subst_trm''[OF ℐ'(4), of "Var x"]
    by force
qed

have 4: "t = attack⟨n⟩"
  when t: "t · ξ ∘s σ ∘s α = attack⟨n⟩"
    and hyps: "transaction_decl_subst ξ T"
              "transaction_fresh_subst σ T (trmslsst 𝒜)"
              "transaction_renaming_subst α P (varslsst 𝒜)"
    and T: "∀x ∈ set (transaction_fresh T). Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
  for t n
    and T::"('fun, 'atom, 'sets, 'lbl) prot_transaction"
    and ξ σ α::"('fun, 'atom, 'sets, 'lbl) prot_subst"
    and 𝒜::"('fun, 'atom, 'sets, 'lbl) prot_strand"
proof (cases t)
  case (Var x)
  hence "attack⟨n⟩ ∈ subst_range (ξ ∘s σ ∘s α)"
    by (metis (no_types, lifting) t eval_term.simps(1) subst_imgI term.distinct(1))
  thus ?thesis
    using transaction_decl_fresh_renaming_substs_range_no_attack_const[OF hyps T]
    by blast
qed (use t in simp)

have 5: "∃ts'. ts = ts' ·list ϑ ∧ (l,send⟨ts'⟩) ∈ set (transaction_strand T)"
  when ts: "(l,receive⟨ts⟩) ∈ set (duallsst (transaction_strand T ·lsst ϑ))"
  for l ts ϑ and T::"('fun, 'atom, 'sets, 'lbl) prot_transaction"
  using subst_lsst_memD(2)[OF ts[unfolded duallsst_steps_iff(1)[symmetric]]]
  by auto

have 6: "l' = l" when "(l',receive⟨[attack⟨l⟩]⟩) ∈ set 𝒜" and Q: "?Q" for l'
  using 𝒜 that(1)
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α) show ?case
  proof (cases "(l',receive⟨[attack⟨l⟩]⟩) ∈ set 𝒜")
    case False
    hence *: "(l',receive⟨[attack⟨l⟩]⟩) ∈ set (duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
      using step.prems by simp
    have "(l',send⟨[attack⟨l⟩]⟩) ∈ set (transaction_strand T)"
      using 4[OF _ step.hyps(3-5)] P(3) step.hyps(2) 5[OF *] by force
    thus ?thesis using Q step.hyps(2) unfolding is_Fun_Attack_def by fastforce
  qed (use step.IH in simp)
```

```
  qed simp

  have 7: "∃t. ts = [t] ∧ t = attack⟨l⟩"
    when ts: "receive⟨ts⟩ ∈ set (unlabel 𝒜)" "attack⟨l⟩ ∈ set ts ·set ℐ" for ts
    using 𝒜 ts(1)
  proof (induction 𝒜 rule: reachable_constraints.induct)
    case (step 𝒜 T ξ σ α)
    obtain t where t: "t ∈ set ts" "attack⟨l⟩ = t · ℐ" using ts(2) by blast
    hence t_in_ik: "t ∈ iklsst (𝒜 @ duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
      using step.prems(1) in_iksst_iff[of t] by blast

    have t_attack_eq: "t = attack⟨l⟩"
    proof (cases t)
      case (Var x)
      hence "TAtom AttackType ∉ subterms (Γ t)"
        using t_in_ik 1[OF reachable_constraints.step[OF step.hyps]] by fastforce
      thus ?thesis using t(2) wt_subst_trm''[OF ℐ'(4), of t] by force
    qed (use t(2) in simp)

    show ?case
    proof (cases "receive⟨ts⟩ ∈ set (unlabel 𝒜)")
      case False
      then obtain l' where l':
          "(l', receive⟨ts⟩) ∈ set (duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
        using step.prems(1) unfolding unlabel_def by force
      then obtain ts' where ts':
          "ts = ts' ·list ξ ∘s σ ∘s α" "(l', send⟨ts'⟩) ∈ set (transaction_strand T)"
        using 5 by meson
      then obtain t' where t': "t' ∈ set ts'" "t' · ξ ∘s σ ∘s α = attack⟨l⟩"
        using t(1) t_attack_eq by force

      note * = t'(1) 4[OF t'(2) step.hyps(3-5)]

      have "send⟨ts'⟩ ∈ set (unlabel (transaction_strand T))"
        using ts'(2) step.hyps(2) P(2) unfolding unlabel_def by force
      hence "length ts' = 1"
        using step.hyps(2) P(2,3) * unfolding admissible_transaction_terms_def by fastforce
      hence "ts' = [attack⟨l⟩]" using * P(3) step.hyps(2) by (cases ts') auto
      thus ?thesis by (simp add: ts'(1))
    qed (use step.IH in simp)
  qed simp

  show "attack⟨l⟩ ∈ iklsst 𝒜 ·set ℐ"
    using private_const_deduct[OF _ l] 3 by simp
  then obtain ts where ts: "receive⟨ts⟩ ∈ set (unlabel 𝒜)" "attack⟨l⟩ ∈ set ts ·set ℐ"
    using in_iklsst_iff[of _ 𝒜] unfolding unlabel_def by force
  then obtain t where "ts = [t]" "t = attack⟨l⟩"
    using 7 by blast
  thus "receive⟨[attack⟨l⟩]⟩ ∈ set (unlabel 𝒜)"
    using ts(1) by blast
  hence "∃l'. (l', receive⟨[attack⟨l⟩]⟩) ∈ set 𝒜"
    unfolding unlabel_def by fastforce
  thus "(l,receive⟨[attack⟨l⟩]⟩) ∈ set 𝒜" when ?Q
    using that 6 by fast
qed

lemma reachable_constraints_receive_attack_if_attack':
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and n: "iklsst 𝒜 ·set ℐ ⊢ attack⟨n⟩"
  shows "attack⟨n⟩ ∈ iklsst 𝒜 ·set ℐ"
    and "receive⟨[attack⟨n⟩]⟩ ∈ set (unlabel 𝒜)"
```

```
proof -
  have P': "∀T ∈ set P. wellformed_transaction T"
           "∀T ∈ set P. admissible_transaction_terms T"
           "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
           "∀T ∈ set P. ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γv x"
    using admissible_transaction_is_wellformed_transaction(1,4)[OF bspec[OF P]]
          admissible_transactionE(2,15)[OF bspec[OF P]]
    by (blast, blast, blast, blast)

  show "attack⟨n⟩ ∈ iklsst 𝒜 ·set ℐ" "receive⟨[attack⟨n⟩]⟩ ∈ set (unlabel 𝒜)"
    using reachable_constraints_receive_attack_if_attack(1,2)[OF 𝒜 P'(1,2) _ P'(4) ℐ n] P'(3)
    by (metis, metis)
qed

lemma constraint_model_Value_term_is_Val:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and x: "Γv x = TAtom Value" "x ∈ fvlsst A"
  shows "∃n. I x = Fun (Val n) []"
using reachable_constraints_occurs_fact_send_ex[OF 𝒜_reach P P_occ x]
      reachable_constraints_occurs_fact_send_in_ik[OF 𝒜_reach ℐ P P_occ]
      reachable_constraints_occurs_fact_ik_case[OF 𝒜_reach P P_occ]
by fast

lemma constraint_model_Value_term_is_Val':
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and x: "(TAtom Value, m) ∈ fvlsst A"
  shows "∃n. I (TAtom Value, m) = Fun (Val n) []"
using constraint_model_Value_term_is_Val[OF 𝒜_reach ℐ P P_occ _ x] by simp


lemma constraint_model_Value_var_in_constr_prefix:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
  shows "∀x ∈ fvlsst 𝒜. Γv x = TAtom Value ⟶ (∃B. prefix B 𝒜 ∧ x ∉ fvlsst B ∧ ℐ x ⊑set trmslsst
B)"
    (is "∀x ∈ ?X 𝒜. ?R x ⟶ ?Q x 𝒜")
using 𝒜_reach ℐ
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  let ?P = "λ𝒜. ∀x ∈ ?X 𝒜. ?R x ⟶ ?Q x 𝒜"

  define T' where "T' ≡ duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"

  have IH: "?P 𝒜" using step welltyped_constraint_model_prefix by fast

  note ξ_empty = admissible_transaction_decl_subst_empty[OF bspec[OF P step.hyps(2)] step.hyps(3)]

  have T_adm: "admissible_transaction' T" by (metis P step.hyps(2))

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

  have ℐ_is_T_model: "strand_sem_stateful (iklsst 𝒜 ·set ℐ) (set (dblsst 𝒜 ℐ)) (unlabel T') ℐ"
    using step.prems unlabel_append[of 𝒜 T'] dbsst_set_is_dbupdsst[of "unlabel 𝒜" ℐ "[]"]
          strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "unlabel T'" ℐ]
    by (simp add: T'_def welltyped_constraint_model_def constraint_model_def dbsst_def)
```

```
have ℐ_interp: "interpretation_subst ℐ"
  and ℐ_wt: "wt_subst ℐ"
  and ℐ_wf_trms: "wf_trms (subst_range ℐ)"
  by (metis ℐ welltyped_constraint_model_def constraint_model_def,
      metis ℐ welltyped_constraint_model_def,
      metis ℐ welltyped_constraint_model_def constraint_model_def)

have 1: "?Q x 𝒜" when x: "x ∈ fv_lsst T'" "Γ_v x = TAtom Value" for x
proof -
  obtain n where n: "ℐ x = Fun n []" "is_Val n" "¬public n"
    using constraint_model_Value_term_is_Val[
            OF reachable_constraints.step[OF step.hyps] step.prems P P_occ x(2)]
          x(1) fv_sst_append[of "unlabel 𝒜" "unlabel T'"] unlabel_append[of 𝒜 T']
    unfolding T'_def by force

  have "x ∈ fv_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
    using x(1) fv_sst_unlabel_dual_lsst_eq unfolding T'_def by fastforce
  then obtain y where y: "y ∈ fv_lsst (transaction_strand T)" "x ∈ fv ((ξ ∘_s σ ∘_s α) y)"
    using fv_sst_subst_obtain_var[of x "unlabel (transaction_strand T)" "ξ ∘_s σ ∘_s α"]
          unlabel_subst[of "transaction_strand T" "ξ ∘_s σ ∘_s α"]
    by auto

  have y_x: "(ξ ∘_s σ ∘_s α) y = Var x" and y_not_fresh: "y ∉ set (transaction_fresh T)"
    using y(2) transaction_decl_fresh_renaming_substs_range[OF step.hyps(3-5), of y]
    by (force, fastforce)

  have "Γ ((ξ ∘_s σ ∘_s α) y) = TAtom Value" using x(2) y_x by simp
  moreover have "wt_subst (ξ ∘_s σ ∘_s α)"
    by (rule transaction_decl_fresh_renaming_substs_wt[OF step.hyps(3-5)])
  ultimately have y_val: "Γ_v y = TAtom Value"
    by (metis wt_subst_def Γ.simps(1))

  have "Fun n [] = (ξ ∘_s σ ∘_s α) y · ℐ" using n y_x by simp
  hence y_n: "Fun n [] = (ξ ∘_s σ ∘_s α ∘_s ℐ) y"
    by (metis subst_subst_compose[of "Var y" "ξ ∘_s σ ∘_s α" ℐ] eval_term.simps(1))

  have 𝒜_ik_ℐ_vals: "∀x ∈ fv_set (ik_lsst 𝒜). ∃f. ℐ x = Fun f []"
  proof -
    have "∃a. Γ (ℐ x) = Var a" when "x ∈ fv_lsst 𝒜" for x
      using that reachable_constraints_vars_TAtom_typed[OF step.hyps(1) P, of x]
            vars_sst_is_fv_sst_bvars_sst[of "unlabel 𝒜"] wt_subst_trm''[OF ℐ_wt, of "Var x"]
      by force
    hence "∃f. ℐ x = Fun f []" when "x ∈ fv_lsst 𝒜" for x
      using that wf_trm_subst[OF ℐ_wf_trms, of "Var x"] wf_trm_Var[of x] const_type_inv_wf
            empty_fv_exists_fun[OF interpretation_grounds[OF ℐ_interp], of "Var x"]
      by (metis eval_term.simps(1)[of _ x ℐ])
    thus ?thesis
      using fv_ik_subset_fv_sst'[of "unlabel 𝒜"] vars_sst_is_fv_sst_bvars_sst[of "unlabel 𝒜"]
      by blast
  qed
  hence 𝒜_subterms_subst_cong: "subterms_set (ik_lsst 𝒜 ·_set ℐ) = subterms_set (ik_lsst 𝒜) ·_set ℐ"
    by (metis ik_sst_subst[of "unlabel 𝒜" ℐ] unlabel_subst[of 𝒜 ℐ] subterms_subst_lsst_ik[of 𝒜 ℐ])

  have x_nin_𝒜: "x ∉ fv_lsst 𝒜"
  proof -
    have "x ∈ fv_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
      using x(1) fv_sst_unlabel_dual_lsst_eq unfolding T'_def by fast
    hence "x ∈ fv_sst ((unlabel (transaction_strand T) ·_sst σ) ·_sst α)"
      using transaction_fresh_subst_grounds_domain[OF step.hyps(4)] step.hyps(4)
            labeled_stateful_strand_subst_comp[of σ "transaction_strand T" α]
            unlabel_subst[of "transaction_strand T ·_lsst σ" α]
            unlabel_subst[of "transaction_strand T" σ]
```

```
      by (simp add: ξ_empty transaction_fresh_subst_def range_vars_alt_def)
    then obtain y where "α y = Var x"
      using transaction_renaming_subst_var_obtain(1)[OF step.hyps(5)] by blast
    thus ?thesis
      using transaction_renaming_subst_range_notin_vars[OF step.hyps(5), of y]
            vars_sst_is_fv_sst_bvars_sst[of "unlabel 𝒜"]
      by auto
  qed

  from admissible_transaction_fv_in_receives_or_selects[OF T_adm y(1) y_not_fresh]
  have n_cases: "Fun n [] ⊑set trms_lsst 𝒜 ∨ (∃z ∈ fv_lsst 𝒜. Γv z = TAtom Value ∧ ℐ z = Fun n [])"
  proof
    assume y_in: "y ∈ fv_lsst (transaction_receive T)"
    then obtain ts where ts:
        "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T))" "y ∈ fv_set (set ts)"
      using admissible_transaction_strand_step_cases(1)[OF T_adm]
      by force
    hence ts_deduct: "list_all (λt. ik_lsst 𝒜 ·set ℐ ⊢ t · ξ ∘s σ ∘s α · ℐ) ts"
      using wellformed_transaction_sem_receives[
              OF T_wf, of "ik_lsst 𝒜 ·set ℐ" "set (db_lsst 𝒜 ℐ)" "ξ ∘s σ ∘s α" ℐ "ts ·list ξ ∘s σ ∘s α"]
            ℐ_is_T_model
            subst_lsst_unlabel_member[of "receive⟨ts⟩" "transaction_receive T" "ξ ∘s σ ∘s α"]
      unfolding T'_def list_all_iff by force

    obtain ty where ty: "ty ∈ set ts" "y ∈ fv ty" using ts(2) by fastforce

    have "Fun n [] ⊑set ik_lsst 𝒜 ∨ (∃z ∈ fv_set (ik_lsst 𝒜). Γv z = TAtom Value ∧ ℐ z = Fun n [])"
    proof -
      have "Fun n [] ⊑ ty · ξ ∘s σ ∘s α · ℐ"
        using imageI[of "Var y" "subterms ty" "λx. x · ξ ∘s σ ∘s α ∘s ℐ"]
              var_is_subterm[OF ty(2)] subterms_subst_subset[of "ξ ∘s σ ∘s α ∘s ℐ" ty]
              subst_subst_compose[of ty "ξ ∘s σ ∘s α" ℐ] y_n
        by (auto simp del: subst_subst_compose)
      hence "Fun n [] ⊑set ik_lsst 𝒜 ·set ℐ"
        using ty(1) private_fun_deduct_in_ik[of _ _ n "[]"] n(2,3) ts_deduct
        unfolding is_Val_def is_Abs_def list_all_iff by fast
      hence "Fun n [] ⊑set ik_lsst 𝒜 ∨ (∃z ∈ fv_set (ik_lsst 𝒜). ℐ z = Fun n [])"
        using const_subterm_subst_cases[of n _ ℐ] 𝒜_ik_ℐ_vals by fastforce
      thus ?thesis
        using ℐ_wt n(2) unfolding wt_subst_def is_Val_def is_Abs_def by force
    qed
    thus ?thesis
      using fv_ik_subset_fv_sst' ik_sst_trms_sst_subset[of "unlabel 𝒜"] 𝒜_subterms_subst_cong
      by fast
  next
    assume y_in: "y ∈ fv_lsst (transaction_checks T) ∧
                 (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ y ∈ fv t ∪ fv s)"
    then obtain s where s: "select⟨Var y,Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))"
      using admissible_transaction_strand_step_cases(2)[OF T_adm] by force
    hence "select⟨(ξ ∘s σ ∘s α) y, Fun (Set s) []⟩ ∈
            set (unlabel (transaction_checks T ·lsst ξ ∘s σ ∘s α))"
      using subst_lsst_unlabel_member
      by fastforce
    hence n_in_db: "(Fun n [], Fun (Set s) []) ∈ set (db'_sst (unlabel 𝒜) ℐ [])"
      using wellformed_transaction_sem_pos_checks(1)[
              OF T_wf, of "ik_lsst 𝒜 ·set ℐ" "set (db_lsst 𝒜 ℐ)" "ξ ∘s σ ∘s α" ℐ
                          assign "(ξ ∘s σ ∘s α) y" "Fun (Set s) []"]
            ℐ_is_T_model n y_x
      unfolding T'_def db_sst_def
      by fastforce

    obtain tn sn where tsn: "insert⟨tn,sn⟩ ∈ set (unlabel 𝒜)" "Fun n [] = tn · ℐ"
      using db_sst_in_cases[OF n_in_db] by force
```

```
      have "Fun n [] = tn ∨ (∃z. Γᵥ z = TAtom Value ∧ tn = Var z)"
        using ℐ_wt tsn(2) n(2) unfolding wt_subst_def is_Val_def is_Abs_def by (cases tn) auto
      moreover have "tn ∈ subterms_set (trms_lsst 𝒜)" "fv tn ⊆ fv_lsst 𝒜"
        using tsn(1) in_subterms_Union by force+
      ultimately show ?thesis using tsn(2) by auto
    qed

    from n_cases show ?thesis
    proof
      assume "∃z ∈ fv_lsst 𝒜. Γᵥ z = TAtom Value ∧ ℐ z = Fun n []"
      then obtain B where B: "prefix B 𝒜" "Fun n [] ∈ subterms_set (trms_lsst B)"
        by (metis IH n(1))
      thus ?thesis
        using n x_nin_𝒜 trms_sst_unlabel_prefix_subset(1)[of B]
        by (metis (no_types, opaque_lifting) self_append_conv subset_iff subterms_set_mono prefix_def)
    qed (use n x_nin_𝒜 in fastforce)
  qed

  have "?P (𝒜@T')"
  proof (intro ballI impI)
    fix x assume x: "x ∈ fv_lsst (𝒜@T')" "Γᵥ x = TAtom Value"
    show "?Q x (𝒜@T')"
    proof (cases "x ∈ fv_lsst 𝒜")
      case False
      hence "x ∈ fv_lsst T'" using x(1) unlabel_append[of 𝒜] fv_sst_append[of "unlabel 𝒜"] by simp
      then obtain B where B: "prefix B 𝒜" "x ∉ fv_lsst B" "ℐ x ∈ subterms_set (trms_lsst B)"
        using x(2) 1 by blast
      thus ?thesis using prefix_prefix by fast
    qed (use x(2) IH prefix_prefix in fast)
  qed
  thus ?case unfolding T'_def by blast
qed simp

lemma constraint_model_Val_const_in_constr_prefix:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. wellformed_transaction T"
           "∀T ∈ set P. admissible_transaction_terms T"
    and n: "Fun (Val n) [] ⊑_set ik_lsst 𝒜 ·_set ℐ"
  shows "Fun (Val n) [] ⊑_set trms_lsst 𝒜"
proof -
  have *: "wf_sst (unlabel 𝒜)"
          "constr_sem_stateful ℐ (unlabel 𝒜)"
          "interpretation_subst ℐ"
          "wf_trms (subst_range ℐ)"
          "wt_subst ℐ"
    using reachable_constraints_wf(1)[OF P(1) _ 𝒜_reach]
          admissible_transaction_terms_wf_trms ℐ P(2) n
    unfolding welltyped_constraint_model_def constraint_model_def wf_trms_code by fast+

  show ?thesis
    using constraint_model_priv_const_in_constr_prefix[OF * _ _ n]
    by simp
qed

lemma constraint_model_Val_const_in_constr_prefix':
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and n: "Fun (Val n) [] ⊑_set ik_lsst 𝒜 ·_set ℐ"
  shows "Fun (Val n) [] ⊑_set trms_lsst 𝒜"
using constraint_model_Val_const_in_constr_prefix[OF 𝒜_reach ℐ _ _ n]
```

```
    P admissible_transaction_is_wellformed_transaction(1,4)
by fast

lemma constraint_model_Value_in_constr_prefix_fresh_action':
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction_terms T"
           "∀T ∈ set P. transaction_decl T () = []"
           "∀T ∈ set P. bvars_transaction T = {}"
    and n: "Fun (Val n) [] ⊑set trmslsst A"
  obtains B T ξ σ α where "prefix (B@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) A"
    and "B ∈ reachable_constraints P" "T ∈ set P" "transaction_decl_subst ξ T"
        "transaction_fresh_subst σ T (trmslsst B)" "transaction_renaming_subst α P (varslsst B)"
    and "Fun (Val n) [] ∈ subst_range σ"
proof -
  define f where "f ≡
    λ(T::('fun, 'atom, 'sets, 'lbl) prot_transaction,
      ξ::('fun, 'atom, 'sets, 'lbl) prot_subst,
      σ::('fun, 'atom, 'sets, 'lbl) prot_subst,
      α::('fun, 'atom, 'sets, 'lbl) prot_subst).
        duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"

  define g where "g ≡ concat ∘ map f"

  obtain Ts where Ts:
      "A = g Ts" "∀B. prefix B Ts ⟶ g B ∈ reachable_constraints P"
      "∀B T ξ σ α. prefix (B@[(T,ξ,σ,α)]) Ts ⟶
                        T ∈ set P ∧ transaction_decl_subst ξ T ∧
                        transaction_fresh_subst σ T (trmslsst (g B)) ∧
                        transaction_renaming_subst α P (varslsst (g B))"
    using reachable_constraints_as_transaction_lists[OF A] unfolding g_def f_def by blast

  obtain T ξ σ α where T:
      "(T, ξ, σ, α) ∈ set Ts" "Fun (Val n) [] ⊑set trmslsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
    using n trmssst_unlabel_duallsst_eq unlabel_subst
    unfolding Ts(1) g_def f_def unlabel_def trmssst_def
    by fastforce

  obtain B where B:
      "prefix (B@[(T, ξ, σ, α)]) Ts" "g B ∈ reachable_constraints P" "T ∈ set P"
      "transaction_decl_subst ξ T" "transaction_fresh_subst σ T (trmslsst (g B))"
      "transaction_renaming_subst α P (varslsst (g B))"
  proof -
    obtain B where "∃C. B@(T, ξ, σ, α)#C = Ts" by (metis T(1) split_list)
    thus ?thesis using Ts(2-) that[of B] by auto
  qed

  note T_adm_terms = bspec[OF P(1) B(3)]
  note T_decl_empty = bspec[OF P(2) B(3)]
  note T_no_bvars = bspec[OF P(3) B(3)]
  note ξ_empty = admissible_transaction_decl_subst_empty'[OF T_decl_empty B(4)]

  have "trmssst (unlabel (transaction_strand T) ·sst ξ ∘s σ ∘s α) = trms_transaction T ·set ξ ∘s σ ∘s α"
    using trmssst_subst[of _ "ξ ∘s σ ∘s α"] T_no_bvars by blast
  hence "Fun (Val n) [] ⊑set trms_transaction T ·set ξ ∘s σ ∘s α"
    by (metis T(2) unlabel_subst)
  hence "Fun (Val n) [] ⊑set subst_range (ξ ∘s σ ∘s α)"
    using admissible_transaction_terms_no_Value_consts(1)[OF T_adm_terms]
          const_subterms_subst_cases'[of "Val n" "ξ ∘s σ ∘s α" "trms_transaction T"]
    by blast
  then obtain tn where tn: "tn ∈ subst_range (ξ ∘s σ ∘s α)" "Fun (Val n) [] ⊑ tn" "is_Fun tn"
    by fastforce
```

```
  have "Fun (Val n) [] ∈ subst_range σ"
    using tn(1-) transaction_decl_fresh_renaming_substs_range'(2,4)[OF B(4-6) tn(1) ξ_empty]
    by fastforce
  moreover have "prefix (g B@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) A"
    using Ts(1) B(1) unfolding g_def f_def prefix_def by fastforce
  ultimately show thesis using that B(2-) by blast
qed

lemma constraint_model_Value_in_constr_prefix_fresh_action:
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
  assumes A: "A ∈ reachable_constraints P"
    and P_adm: "∀T ∈ set P. admissible_transaction' T"
    and n: "Fun (Val n) [] ⊑_set trms_lsst A"
  obtains B T ξ σ α where "prefix (B@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) A"
    and "B ∈ reachable_constraints P" "T ∈ set P" "transaction_decl_subst ξ T"
        "transaction_fresh_subst σ T (trms_lsst B)" "transaction_renaming_subst α P (vars_lsst B)"
    and "Fun (Val n) [] ∈ subst_range σ"
proof -
  have P: "∀T ∈ set P. admissible_transaction_terms T"
          "∀T ∈ set P. transaction_decl T () = []"
          "∀T ∈ set P. bvars_transaction T = {}"
  using P_adm admissible_transactionE(1) admissible_transaction_no_bvars(2)
        admissible_transaction_is_wellformed_transaction(4)
  by (blast,blast,blast)

  show ?thesis using that constraint_model_Value_in_constr_prefix_fresh_action'[OF A P n] by blast
qed

lemma reachable_constraints_occurs_fact_ik_case':
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and val: "Fun (Val n) [] ⊑_set trms_lsst A"
  shows "occurs (Fun (Val n) []) ∈ ik_lsst A"
proof -
  obtain B T ξ σ α where B:
      "prefix (B@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) A"
      "B ∈ reachable_constraints P"
      "T ∈ set P"
      "transaction_decl_subst ξ T"
      "transaction_fresh_subst σ T (trms_lsst B)"
      "transaction_renaming_subst α P (vars_lsst B)"
      "Fun (Val n) [] ∈ subst_range σ"
    using constraint_model_Value_in_constr_prefix_fresh_action[OF 𝒜_reach P val]
    by blast

  define ϑ where "ϑ ≡ ξ ∘_s σ ∘_s α"

  have T_adm: "admissible_transaction' T" using P B(3) by blast
  hence T_wf: "wellformed_transaction T" "admissible_transaction_occurs_checks T"
    using admissible_transaction_is_wellformed_transaction(1) bspec[OF P_occ B(3)] by (blast,blast)

  obtain x where x: "x ∈ set (transaction_fresh T)" "ϑ x = Fun (Val n) []"
    using transaction_fresh_subst_domain[OF B(5)] B(7)
          admissible_transaction_decl_subst_empty[OF T_adm B(4)]
    by (force simp add: subst_compose ϑ_def)

  obtain ts where ts: "send⟨ts⟩ ∈ set (unlabel (transaction_send T))" "occurs (Var x) ∈ set ts"
    using admissible_transaction_occurs_checksE2[OF T_wf(2) x(1)]
    by (metis (mono_tags, lifting) list.set_intros(1) unlabel_Cons(1))

  have "occurs (Var x) ∈ trms_lsst (transaction_send T)"
```

```
    using ts by force
  hence "occurs (Var x) · ϑ ∈ ik_lsst (dual_lsst (transaction_strand T ·_lsst ϑ))"
    using dual_transaction_ik_is_transaction_send'[OF T_wf(1), of ϑ] by fast
  hence "occurs (Fun (Val n) []) ∈ ik_lsst (dual_lsst (transaction_strand T ·_lsst ϑ))"
    using x(2) by simp
  thus ?thesis
    using B(1)[unfolded ϑ_def[symmetric]]
          unlabel_append[of B "dual_lsst (transaction_strand T ·_lsst ϑ)"]
          ik_sst_append[of "unlabel B" "unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))"]
    unfolding prefix_def by force
qed

lemma reachable_constraints_occurs_fact_ik_case'':
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ A"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and val: "Fun (Val n) [] ⊑ t" "ik_lsst A ·_set ℐ ⊢ t"
  shows "occurs (Fun (Val n) []) ∈ ik_lsst A"
proof -
  obtain f ts where t: "t = Fun f ts" using val(1) by (cases t) simp_all

  show ?thesis
    using private_fun_deduct_in_ik[OF val(2,1)[unfolded t]]
          constraint_model_Val_const_in_constr_prefix'[OF 𝒜_reach ℐ P, of n]
          reachable_constraints_occurs_fact_ik_case'[OF 𝒜_reach P P_occ, of n]
    by fastforce
qed

lemma admissible_transaction_occurs_checks_prop:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and f: "f ∈ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
  shows "¬is_PubConstValue f"
    and "¬is_Abs f"
proof -
  obtain x where x: "x ∈ fv_lsst 𝒜" "f ∈ funs_term (ℐ x)" using f by force
  obtain T where T: "Fun f T ⊑ ℐ x" using funs_term_Fun_subterm[OF x(2)] by force

  have ℐ_interp: "interpretation_subst ℐ"
    and ℐ_wt: "wt_subst ℐ"
    and ℐ_wf_trms: "wf_trms (subst_range ℐ)"
    by (metis ℐ welltyped_constraint_model_def constraint_model_def,
        metis ℐ welltyped_constraint_model_def,
        metis ℐ welltyped_constraint_model_def constraint_model_def)

  note 0 = x(1) reachable_constraints_vars_TAtom_typed[OF 𝒜_reach P, of x]
           vars_sst_is_fv_sst_bvars_sst[of "unlabel 𝒜"]

  have 1: "Γ (Var x) = Γ (ℐ x)" using wt_subst_trm''[OF ℐ_wt, of "Var x"] by simp
  hence "∃a. Γ (ℐ x) = Var a" using 0 by force
  hence "∃f. ℐ x = Fun f []"
    using x(1) wf_trm_subst[OF ℐ_wf_trms, of "Var x"] wf_trm_Var[of x] const_type_inv_wf
          empty_fv_exists_fun[OF interpretation_grounds[OF ℐ_interp], of "Var x"]
    by (metis eval_term.simps(1)[of _ x ℐ])
  hence 2: "ℐ x = Fun f []" using x(2) by force

  have 3: "Γ_v x ≠ TAtom AbsValue" using 0 by force

  have "¬is_PubConstValue f ∧ ¬is_Abs f"
```

```
  proof (cases "Γ_v x = TAtom Value")
    case True
    then obtain B where B: "prefix B 𝒜" "x ∉ fv_lsst B" "ℐ x ∈ subterms_set (trms_lsst B)"
      using constraint_model_Value_var_in_constr_prefix[OF 𝒜_reach ℐ P P_occ] x(1)
      by fast

    have "ℐ x ∈ subterms_set (trms_lsst 𝒜)"
      using B(1,3) trms_sst_append[of "unlabel B"] unlabel_append[of B]
      unfolding prefix_def by auto
    hence "f ∈ ⋃(funs_term ` trms_lsst 𝒜)"
      using x(2) funs_term_subterms_eq(2)[of "trms_lsst 𝒜"] by blast
    thus ?thesis
      using reachable_constraints_val_funs_private[OF 𝒜_reach P]
      by blast+
  next
    case False thus ?thesis using x 1 2 3 unfolding is_PubConstValue_def by (cases f) auto
  qed
  thus "¬is_PubConstValue f" "¬is_Abs f" by metis+
qed

lemma admissible_transaction_occurs_checks_prop':
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and f: "f ∈ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
  shows "∄n. f = PubConst Value n"
    and "∄n. f = Abs n"
using admissible_transaction_occurs_checks_prop[OF 𝒜_reach ℐ P P_occ f]
unfolding is_PubConstValue_def by auto

lemma transaction_var_becomes_Val:
  assumes 𝒜_reach: "𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α) ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trms_lsst 𝒜)"
    and α: "transaction_renaming_subst α P (vars_lsst 𝒜)"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and T: "T ∈ set P"
    and x: "x ∈ fv_transaction T" "fst x = TAtom Value"
  shows "∃n. Fun (Val n) [] = (ξ ∘_s σ ∘_s α) x · ℐ"
proof -
  obtain m where m: "x = (TAtom Value, m)" by (metis x(2) eq_fst_iff)

  note ξ_empty = admissible_transaction_decl_subst_empty[OF bspec[OF P T] ξ]

  have x_not_bvar: "x ∉ bvars_transaction T" "fv ((ξ ∘_s σ ∘_s α) x) ∩ bvars_transaction T = {}"
    using x(1) admissible_transactions_fv_bvars_disj[OF P] T
          transaction_decl_fresh_renaming_substs_vars_disj(2)[OF ξ σ α, of x]
          vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_strand T)"]
    by (blast, blast)

  have σx_type: "Γ (σ x) = TAtom Value"
    using σ x Γ_v_TAtom''(2)[of x] wt_subst_trm''[of σ "Var x"]
    unfolding transaction_fresh_subst_def by simp

  show ?thesis
  proof (cases "x ∈ subst_domain σ")
    case True
    then obtain c where c: "σ x = Fun c []" "¬public c" "arity c = 0"
      using σ unfolding transaction_fresh_subst_def by fastforce
    then obtain n where n: "c = Val n" using σx_type by (cases c) (auto split: option.splits)
```

```
    show ?thesis using c n subst_compose[of σ α x] ξ_empty by simp
  next
    case False
    hence "σ x = Var x" by auto
    then obtain n where n: "(σ ∘s α) x = Var (TAtom Value, n)"
      using m transaction_renaming_subst_is_renaming(1)[OF α] subst_compose[of σ α x]
      by force
    hence "(TAtom Value, n) ∈ fv_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
      using x_not_bvar fv_sst_subst_fv_subset[OF x(1), of "ξ ∘s σ ∘s α"]
            unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"] ξ_empty
      by force
    hence "∃n'. I (TAtom Value, n) = Fun (Val n') []"
      using constraint_model_Value_term_is_Val'[OF A_reach I P P_occ, of n] x
            fv_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst ξ ∘s σ ∘s α"]
            fv_sst_append[of "unlabel A"] unlabel_append[of A]
      by fastforce
    thus ?thesis using n ξ_empty by simp
  qed
qed

lemma reachable_constraints_SMP_subset:
  assumes A: "A ∈ reachable_constraints P"
  shows "SMP (trms_lsst A) ⊆ SMP (⋃T ∈ set P. trms_transaction T)" (is "?A A")
    and "SMP (pair`setops_sst (unlabel A)) ⊆ SMP (⋃T∈set P. pair`setops_transaction T)" (is "?B A")
proof -
  have "?A A ∧ ?B A" using A
  proof (induction A rule: reachable_constraints.induct)
    case (step A T ξ σ α)
    define T' where "T' ≡ transaction_strand T ·lsst ξ ∘s σ ∘s α"
    define M where "M ≡ ⋃T ∈ set P. trms_transaction T"
    define N where "N ≡ ⋃T ∈ set P. pair ` setops_transaction T"

    let ?P = "λt. ∃s δ. s ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = s · δ"
    let ?Q = "λt. ∃s δ. s ∈ N ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = s · δ"

    have IH: "SMP (trms_lsst A) ⊆ SMP M" "SMP (pair ` setops_sst (unlabel A)) ⊆ SMP N"
      using step.IH by (metis M_def, metis N_def)

    note ξσα_wt = transaction_decl_fresh_renaming_substs_wt[OF step.hyps(3-5)]
    note ξσα_wf = transaction_decl_fresh_renaming_substs_range_wf_trms[OF step.hyps(3-5)]

    have 0: "SMP (trms_lsst (A@dual_lsst T')) = SMP (trms_lsst A) ∪ SMP (trms_lsst T')"
            "SMP (pair ` setops_sst (unlabel (A@dual_lsst T'))) =
              SMP (pair ` setops_sst (unlabel A)) ∪ SMP (pair ` setops_sst (unlabel T'))"
      using trms_sst_unlabel_dual_lsst_eq[of T']
            setops_sst_unlabel_dual_lsst_eq[of T']
            trms_sst_append[of "unlabel A" "unlabel (dual_lsst T')"]
            setops_sst_append[of "unlabel A" "unlabel (dual_lsst T')"]
            unlabel_append[of A "dual_lsst T'"]
            image_Un[of pair "setops_sst (unlabel A)" "setops_sst (unlabel T')"]
            SMP_union[of "trms_lsst A" "trms_lsst T'"]
            SMP_union[of "pair ` setops_sst (unlabel A)" "pair ` setops_sst (unlabel T')"]
      by argo+

    have 1: "SMP (trms_lsst T') ⊆ SMP M"
    proof (intro SMP_subset_I ballI)
      fix t show "t ∈ trms_lsst T' ⟹ ?P t"
        using trms_sst_wt_subst_ex[OF ξσα_wt ξσα_wf, of t "unlabel (transaction_strand T)"]
              unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"] step.hyps(2)
        unfolding T'_def M_def by auto
    qed

    have 2: "SMP (pair ` setops_sst (unlabel T')) ⊆ SMP N"
```

```
    proof (intro SMP_subset_I ballI)
      fix t show "t ∈ pair ` setops_sst (unlabel T') ⟹ ?Q t"
        using setops_sst_wt_subst_ex[OF ξσα_wt ξσα_wf, of t "unlabel (transaction_strand T)"]
              unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"] step.hyps(2)
        unfolding T'_def N_def by auto
    qed

    have "SMP (trms_lsst (A@dual_lsst T')) ⊆ SMP M"
         "SMP (pair ` setops_sst (unlabel (A@dual_lsst T'))) ⊆ SMP N"
      using 0 1 2 IH by blast+
    thus ?case unfolding T'_def M_def N_def by blast
  qed (simp add: setops_sst_def)
  thus "?A 𝒜" "?B 𝒜" by metis+
qed

lemma reachable_constraints_no_Pair_fun':
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
           "∀T ∈ set P. transaction_decl T () = []"
           "∀T ∈ set P. admissible_transaction_terms T"
           "∀T ∈ set P. ∀x ∈ vars_transaction T. Γv x = TAtom Value ∨ (∃a. Γv x = ⟨a⟩τa)"
  shows "Pair ∉ ⋃(funs_term ` SMP (trms_lsst A))"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step A T ξ σ α)
  define T' where "T' ≡ dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"

  note T_fresh_type = bspec[OF P(1) step.hyps(2)]

  note ξ_empty =
    admissible_transaction_decl_subst_empty'[OF bspec[OF P(2) step.hyps(2)] step.hyps(3)]

  note T_adm_terms = bspec[OF P(3) step.hyps(2)]

  note ξσα_wt = transaction_decl_fresh_renaming_substs_wt[OF step.hyps(3-5)]
  note ξσα_wf = transaction_decl_fresh_renaming_substs_range_wf_trms[OF step.hyps(3-5)]

  have 0: "SMP (trms_lsst (A@T')) = SMP (trms_lsst A) ∪ SMP (trms_lsst T')"
    using SMP_union[of "trms_lsst A" "trms_lsst T'"]
          unlabel_append[of A T'] trms_sst_append[of "unlabel A" "unlabel T'"]
    by simp

  have 1: "wf_trms (trms_lsst T')"
    using reachable_constraints_wf_trms[OF _ reachable_constraints.step[OF step.hyps]]
          admissible_transaction_terms_wf_trms P(3)
          trms_sst_append[of "unlabel A"] unlabel_append[of A]
    unfolding T'_def by force

  have 2: "Pair ∉ ⋃(funs_term ` (subst_range (ξ ∘s σ ∘s α)))"
    using transaction_decl_fresh_renaming_substs_range'(3)[
            OF step.hyps(3-5) _ ξ_empty T_fresh_type]
    by force

  have "Pair ∉ ⋃(funs_term ` (trms_transaction T))"
    using T_adm_terms unfolding admissible_transaction_terms_def by blast
  hence "Pair ∉ funs_term t"
    when t: "t ∈ trms_sst (unlabel (transaction_strand T) ·sst ξ ∘s σ ∘s α)" for t
    using 2 trms_sst_funs_term_cases[OF t]
    by force
  hence 3: "Pair ∉ funs_term t" when t: "t ∈ trms_lsst T'" for t
    using t unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
          trms_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst ξ ∘s σ ∘s α"]
    unfolding T'_def by metis
```

```
  have "∃a. Γᵥ x = TAtom a" when "x ∈ vars_transaction T" for x
    using that protocol_transaction_vars_TAtom_typed(1) bspec[OF P(4) step.hyps(2)]
    by fast
  hence "∃a. Γᵥ x = TAtom a" when "x ∈ vars_sst (unlabel (transaction_strand T) ·sst ξ ∘s σ ∘s α)" for
x
    using wt_subst_fv_set_termtype_subterm[OF _ ξσα_wt ξσα_wf, of x "vars_transaction T"]
          vars_sst_subst_cases[OF that]
    by fastforce
  hence "∃a. Γᵥ x = TAtom a" when "x ∈ vars_lsst T'" for x
    using that unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
          vars_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst ξ ∘s σ ∘s α"]
    unfolding T'_def
    by simp
  hence "∃a. Γᵥ x = TAtom a" when "x ∈ fv_set (trms_lsst T')" for x
    using that fv_trms_sst_subset(1) by fast
  hence "Pair ∉ funs_term (Γ (Var x))" when "x ∈ fv_set (trms_lsst T')" for x
    using that by fastforce
  moreover have "Pair ∈ funs_term s"
    when s: "Ana s = (K, M)" "Pair ∈ ⋃(funs_term ` set K)"
    for s::"('fun,'atom,'sets,'lbl) prot_term" and K M
  proof (cases s)
    case (Fun f S) thus ?thesis using s Ana_Fu_keys_not_pairs[of _ S K M] by (cases f) force+
  qed (use s in simp)
  ultimately have "Pair ∉ funs_term t" when t: "t ∈ SMP (trms_lsst T')" for t
    using t 3 SMP_funs_term[OF t _ _ 1, of Pair] funs_term_type_iff by fastforce
  thus ?case using 0 step.IH(1) unfolding T'_def by blast
qed simp

lemma reachable_constraints_no_Pair_fun:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
  shows "Pair ∉ ⋃(funs_term ` SMP (trms_lsst A))"
using reachable_constraints_no_Pair_fun'[OF A]
      P admissible_transactionE(1,2,3)
      admissible_transaction_is_wellformed_transaction(4)
by blast

lemma reachable_constraints_setops_form:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and t: "t ∈ pair ` setops_sst (unlabel A)"
  shows "∃c s. t = pair (c, Fun (Set s) []) ∧ Γ c = TAtom Value"
using A t
proof (induction A rule: reachable_constraints.induct)
  case (step A T ξ σ α)

  have T_adm: "admissible_transaction' T" when "T ∈ set P" for T
    using P that unfolding list_all_iff by simp

  note T_adm' = admissible_transaction_is_wellformed_transaction(2,3)[OF T_adm]
  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

  note ξσα_wt = transaction_decl_fresh_renaming_substs_wt[OF step.hyps(3-5)]
  note ξσα_wf = transaction_decl_fresh_renaming_substs_range_wf_trms[OF step.hyps(3-5)]

  show ?case using step.IH
  proof (cases "t ∈ pair ` setops_sst (unlabel A)")
    case False
    hence "t ∈ pair ` setops_sst (unlabel (transaction_strand T) ·sst ξ ∘s σ ∘s α)"
      using step.prems setops_sst_append unlabel_append
            setops_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst ξ ∘s σ ∘s α"]
            unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
```

```
        by fastforce
      then obtain t' δ where t':
          "t' ∈ pair ` setops_sst (unlabel (transaction_strand T))"
          "wt_subst δ" "wf_trms (subst_range δ)" "t = t' · δ"
        using setops_sst_wt_subst_ex[OF ξσα_wt ξσα_wf] by blast
      then obtain s s' where s: "t' = pair (s,s')"
        using setops_sst_are_pairs by fastforce
      moreover have "InSet ac s s' = InSet Assign s s' ∨ InSet ac s s' = InSet Check s s'" for ac
        by (cases ac) simp_all
      ultimately have "∃n. s = Var (Var Value, n)" "∃u. s' = Fun (Set u) []"
        using t'(1) setops_sst_member_iff[of s s' "unlabel (transaction_strand T)"]
              pair_in_pair_image_iff[of s s']
              transaction_inserts_are_Value_vars[
                OF T_wf[OF step.hyps(2)] T_adm'(2)[OF step.hyps(2)], of s s']
              transaction_deletes_are_Value_vars[
                OF T_wf[OF step.hyps(2)] T_adm'(2)[OF step.hyps(2)], of s s']
              transaction_selects_are_Value_vars[
                OF T_wf[OF step.hyps(2)] T_adm'(1)[OF step.hyps(2)], of s s']
              transaction_inset_checks_are_Value_vars[
                OF T_adm[OF step.hyps(2)], of s s']
              transaction_notinset_checks_are_Value_vars[
                OF T_adm[OF step.hyps(2)], of _ _ _ s s']
        by metis+
      then obtain ss n where ss: "t = pair (δ (Var Value, n), Fun (Set ss) [])"
        using t'(4) s unfolding pair_def by force

      have "Γ (δ (Var Value, n)) = TAtom Value" "wf_trm (δ (Var Value, n))"
        using t'(2) wt_subst_trm''[OF t'(2), of "Var (Var Value, n)"] apply simp
        using t'(3) by (cases "(Var Value, n) ∈ subst_domain δ") auto
      thus ?thesis using ss by blast
    qed simp
qed (simp add: setops_sst_def)

lemma reachable_constraints_insert_delete_form:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and t: "insert⟨t,s⟩ ∈ set (unlabel A) ∨ delete⟨t,s⟩ ∈ set (unlabel A)" (is "?Q t s A")
  shows "∃k. s = Fun (Set k) []" (is ?A)
    and "Γ t = TAtom Value" (is ?B)
    and "(∃x. t = Var x) ∨ (∃n. t = Fun (Val n) [])" (is ?C)
proof -
  have 0: "pair (t,s) ∈ pair ` setops_sst (unlabel A)" using t unfolding setops_sst_def by force

  show 1: ?A ?B using reachable_constraints_setops_form[OF A P 0] by (fast,fast)

  show ?C using A t
  proof (induction A rule: reachable_constraints.induct)
    case (step 𝒜 T ξ σ α)
    let ?T' = "transaction_strand T ·_lsst ξ ∘_s σ ∘_s α"

    note T_adm = bspec[OF P step.hyps(2)]
    note T_wf = admissible_transaction_is_wellformed_transaction(1,3)[OF T_adm]

    have "?Q t s 𝒜 ∨ ?Q t s ?T'"
      using step.prems dual_lsst_unlabel_steps_iff(4,5)[of t s ?T']
      unfolding unlabel_append by auto
    thus ?case
    proof
      assume "?Q t s ?T'"
      then obtain u v where u: "?Q u v (transaction_strand T)" "t = u · ξ ∘_s σ ∘_s α"
        by (metis (no_types, lifting) stateful_strand_step_mem_substD(4,5) unlabel_subst)

      obtain x where x: "u = Var x"
```

```
      using u(1) transaction_inserts_are_Value_vars(1)[OF T_wf, of u v]
            transaction_deletes_are_Value_vars(1)[OF T_wf, of u v]
      by fastforce

    show ?case
      using u(2) x
            transaction_decl_fresh_renaming_substs_range'(3)[
              OF step.hyps(3,4,5) _
                 admissible_transaction_decl_subst_empty[OF T_adm step.hyps(3)]
                 admissible_transactionE(2)[OF T_adm],
              of t]
      by (cases "t ∈ subst_range (ξ ∘s σ ∘s α)")
         (blast, metis eval_term.simps(1) subst_imgI)
  qed (use step.IH in fastforce)
 qed simp
qed

lemma reachable_constraints_setops_type:
 fixes t::"('fun,'atom,'sets,'lbl) prot_term"
 assumes A: "A ∈ reachable_constraints P"
   and P: "∀T ∈ set P. admissible_transaction' T"
   and t: "t ∈ pair ` setops_sst (unlabel A)"
 shows "Γ t = TComp Pair [TAtom Value, TAtom SetType]"
proof -
 obtain s c where s: "t = pair (c, Fun (Set s) [])" "Γ c = TAtom Value"
   using reachable_constraints_setops_form[OF A P t] by force
 hence "(Fun (Set s) []::('fun,'atom,'sets,'lbl) prot_term) ∈ trms_lsst A"
   using t setops_sst_member_iff[of c "Fun (Set s) []" "unlabel A"]
   by force
 hence "wf_trm (Fun (Set s) []::('fun,'atom,'sets,'lbl) prot_term)"
   using reachable_constraints_wf(2) P A admissible_transaction_is_wellformed_transaction(1,4)
   unfolding admissible_transaction_terms_def by blast
 hence "arity (Set s) = 0" unfolding wf_trm_def by simp
 thus ?thesis using s unfolding pair_def by fastforce
qed

lemma reachable_constraints_setops_same_type_if_unifiable:
 assumes A: "A ∈ reachable_constraints P"
   and P: "∀T ∈ set P. admissible_transaction' T"
 shows "∀s ∈ pair ` setops_sst (unlabel A). ∀t ∈ pair ` setops_sst (unlabel A).
          (∃δ. Unifier δ s t) ⟶ Γ s = Γ t"
   (is "?P A")
using reachable_constraints_setops_type[OF A P] by simp

lemma reachable_constraints_setops_unifiable_if_wt_instance_unifiable:
 assumes A: "A ∈ reachable_constraints P"
   and P: "∀T ∈ set P. admissible_transaction' T"
 shows "∀s ∈ pair ` setops_sst (unlabel A). ∀t ∈ pair ` setops_sst (unlabel A).
          (∃σ ϑ ϱ. wt_subst σ ∧ wt_subst ϑ ∧ wf_trms (subst_range σ) ∧ wf_trms (subst_range ϑ) ∧
                Unifier ϱ (s · σ) (t · ϑ))
          ⟶ (∃δ. Unifier δ s t)"
proof (intro ballI impI)
 fix s t assume st: "s ∈ pair ` setops_sst (unlabel A)" "t ∈ pair ` setops_sst (unlabel A)" and
   "∃σ ϑ ϱ. wt_subst σ ∧ wt_subst ϑ ∧ wf_trms (subst_range σ) ∧ wf_trms (subst_range ϑ) ∧
          Unifier ϱ (s · σ) (t · ϑ)"
 then obtain σ ϑ ϱ where σ:
     "wt_subst σ" "wt_subst ϑ" "wf_trms (subst_range σ)" "wf_trms (subst_range ϑ)"
     "Unifier ϱ (s · σ) (t · ϑ)"
   by force

 obtain fs ft cs ct where c:
     "s = pair (cs, Fun (Set fs) [])" "t = pair (ct, Fun (Set ft) [])"
     "Γ cs = TAtom Value" "Γ ct = TAtom Value"
```

```
    using reachable_constraints_setops_form[OF A P st(1)]
          reachable_constraints_setops_form[OF A P st(2)]
    by force

  have "cs ∈ subterms_set (trms_lsst A)" "ct ∈ subterms_set (trms_lsst A)"
    using c(1,2) setops_subterm_trms[OF st(1), of cs] setops_subterm_trms[OF st(2), of ct]
          Fun_param_is_subterm[of cs "args s"] Fun_param_is_subterm[of ct "args t"]
    unfolding pair_def by simp_all
  moreover have
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
    using P admissible_transaction_is_wellformed_transaction(1,4)
    unfolding admissible_transaction_terms_def by fast+
  ultimately have *: "wf_trm cs" "wf_trm ct"
    using reachable_constraints_wf(2)[OF _ _ A] wf_trms_subterms by blast+

  have "(∃x. cs = Var x) ∨ (∃c d. cs = Fun c [])"
    using const_type_inv_wf c(3) *(1) by (cases cs) auto
  moreover have "(∃x. ct = Var x) ∨ (∃c d. ct = Fun c [])"
    using const_type_inv_wf c(4) *(2) by (cases ct) auto
  ultimately show "∃δ. Unifier δ s t"
    using reachable_constraints_setops_form[OF A P] reachable_constraints_setops_type[OF A P] st σ c
    unfolding pair_def by auto
qed

lemma reachable_constraints_tfr:
  assumes M:
      "M ≡ ⋃T ∈ set P. trms_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. admissible_transaction T"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "tfr_sst (unlabel 𝒜)"
using 𝒜
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step A T ξ σ α)
  define T' where "T' ≡ dual_lsst (transaction_strand T ·_lsst ξ ∘s σ ∘s α)"

  have P': "∀T ∈ set P. admissible_transaction' T" using P(1) admissible_transactionE'(1) by blast

  have AT'_reach: "A@T' ∈ reachable_constraints P"
    using reachable_constraints.step[OF step.hyps] unfolding T'_def by metis

  note T_adm = bspec[OF P(1) step.hyps(2)]
  note T_adm' = bspec[OF P'(1) step.hyps(2)]

  note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm' step.hyps(3)]

  note ξσα_wt = transaction_decl_fresh_renaming_substs_wt[OF step.hyps(3-5)]
  note ξσα_wf = transaction_decl_fresh_renaming_substs_range_wf_trms[OF step.hyps(3-5)]

  have ξσα_bvars_disj: "bvars_lsst (transaction_strand T) ∩ range_vars (ξ ∘s σ ∘s α) = {}"
    using transaction_decl_fresh_renaming_substs_vars_disj(4)[OF step.hyps(3,4,5,2)] ξ_empty
    by simp

  have wf_trms_M: "wf_trms M"
    using admissible_transactions_wf_trms P'(1) unfolding M(1) by blast

  have "tfr_set (trms_lsst (A@T'))"
```

```
    using reachable_constraints_SMP_subset(1)[OF AT'_reach]
          tfr_subset(3)[OF M(4), of "trms_lsst (A@T')"]
          SMP_SMP_subset[of M N] SMP_I'[OF wf_trms_M M(5,2)]
    unfolding M(1) by blast
  moreover have "∀p. Ana (pair p) = ([],[])" unfolding pair_def by auto
  ultimately have 1: "tfr_set (trms_lsst (A@T') ∪ pair ` setops_sst (unlabel (A@T')))"
    using tfr_setops_if_tfr_trms[of "unlabel (A@T')"]
          reachable_constraints_no_Pair_fun[OF AT'_reach P']
          reachable_constraints_setops_same_type_if_unifiable[OF AT'_reach P']
          reachable_constraints_setops_unifiable_if_wt_instance_unifiable[OF AT'_reach P']
    by blast

  have "list_all tfr_sstp (unlabel (transaction_strand T))"
    using step.hyps(2) P(2) tfr_sstp_is_comp_tfr_sstp
    unfolding comp_tfr_sst_def tfr_sst_def by fastforce
  hence "list_all tfr_sstp (unlabel T')"
    using tfr_sstp_all_wt_subst_apply[OF _ ξσα_wt ξσα_wf ξσα_bvars_disj]
          dual_lsst_tfr_sstp[of "transaction_strand T ·lsst ξ ∘s σ ∘s α"]
          unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
    unfolding T'_def by argo
  hence 2: "list_all tfr_sstp (unlabel (A@T'))"
    using step.IH unlabel_append
    unfolding tfr_sst_def by auto

  have "tfr_sst (unlabel (A@T'))" using 1 2 by (metis tfr_sst_def)
  thus ?case by (metis T'_def)
qed simp

lemma reachable_constraints_tfr':
  assumes M:
      "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ` setops_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "tfr_sst (unlabel 𝒜)"
using 𝒜
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step A T ξ σ α)
  define T' where "T' ≡ dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"

  have AT'_reach: "A@T' ∈ reachable_constraints P"
    using reachable_constraints.step[OF step.hyps] unfolding T'_def by metis

  note ξσα_wt = transaction_decl_fresh_renaming_substs_wt[OF step.hyps(3-5)]
  note ξσα_wf = transaction_decl_fresh_renaming_substs_range_wf_trms[OF step.hyps(3-5)]

  have ξσα_bvars_disj: "bvars_lsst (transaction_strand T) ∩ range_vars (ξ ∘s σ ∘s α) = {}"
    by (rule transaction_decl_fresh_renaming_substs_vars_disj(4)[OF step.hyps(3,4,5,2)])

  have wf_trms_M: "wf_trms M"
    using P(1) setops_sst_wf_trms(2) unfolding M(1) pair_code wf_trms_code[symmetric] by fast

  have "SMP (trms_lsst (A@T')) ⊆ SMP M" "SMP (pair ` setops_sst (unlabel (A@T'))) ⊆ SMP M"
    using reachable_constraints_SMP_subset[OF AT'_reach]
          SMP_mono[of "⋃T ∈ set P. trms_transaction T" M]
          SMP_mono[of "⋃T ∈ set P. pair ` setops_transaction T" M]
    unfolding M(1) pair_code[symmetric] by blast+
  hence 1: "tfr_set (trms_lsst (A@T') ∪ pair ` setops_sst (unlabel (A@T')))"
```

```
    using tfr_subset(3)[OF M(4), of "trms_lsst (A@T') ∪ pair ` setops_sst (unlabel (A@T'))"]
          SMP_union[of "trms_lsst (A@T')" "pair ` setops_sst (unlabel (A@T'))"]
          SMP_SMP_subset[of M N] SMP_I'[OF wf_trms_M M(5,2)]
    by blast

  have "list_all tfr_sstp (unlabel (transaction_strand T))"
    using step.hyps(2) P(2) tfr_sstp_is_comp_tfr_sstp
    unfolding comp_tfr_sst_def tfr_sst_def by fastforce
  hence "list_all tfr_sstp (unlabel T')"
    using tfr_sstp_all_wt_subst_apply[OF _ ξσα_wt ξσα_wf ξσα_bvars_disj]
          dual_lsst_tfr_sstp[of "transaction_strand T ·lsst ξ ∘s σ ∘s α"]
          unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
    unfolding T'_def by argo
  hence 2: "list_all tfr_sstp (unlabel (A@T'))"
    using step.IH unlabel_append
    unfolding tfr_sst_def by auto

  have "tfr_sst (unlabel (A@T'))" using 1 2 by (metis tfr_sst_def)
  thus ?case by (metis T'_def)
qed simp

lemma reachable_constraints_typing_cond_sst:
  assumes M:
      "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ` setops_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "typing_cond_sst (unlabel 𝒜)"
using reachable_constraints_wf[OF P(1,2) 𝒜] reachable_constraints_tfr'[OF M P(2,3) 𝒜]
unfolding typing_cond_sst_def by blast

context
begin
private lemma reachable_constraints_typing_result_aux:
  assumes 0: "wf_sst (unlabel 𝒜)" "tfr_sst (unlabel 𝒜)" "wf_trms (trms_lsst 𝒜)"
  shows "wf_sst (unlabel (𝒜@[(l,send⟨[attack⟨n⟩]⟩)]))" "tfr_sst (unlabel (𝒜@[(l,send⟨[attack⟨n⟩]⟩)]))"
        "wf_trms (trms_lsst (𝒜@[(l,send⟨[attack⟨n⟩]⟩)]))"
proof -
  let ?n = "[(l,send⟨[attack⟨n⟩]⟩)]"
  let ?A = "𝒜@?n"

  show "wf_sst (unlabel ?A)"
    using 0(1) wf_sst_append_suffix'[of "{}" "unlabel 𝒜" "unlabel ?n"] unlabel_append[of 𝒜 ?n]
    by simp

  show "wf_trms (trms_lsst ?A)"
    using 0(3) trms_sst_append[of "unlabel 𝒜" "unlabel ?n"] unlabel_append[of 𝒜 ?n]
    by fastforce

  have "∀t ∈ trms_lsst ?n ∪ pair ` setops_sst (unlabel ?n). ∃c. t = Fun c []"
       "∀t ∈ trms_lsst ?n ∪ pair ` setops_sst (unlabel ?n). Ana t = ([],[])"
    by (simp_all add: setops_sst_def)
  hence "tfr_set (trms_lsst 𝒜 ∪ pair ` setops_sst (unlabel 𝒜) ∪
              (trms_lsst ?n ∪ pair ` setops_sst (unlabel ?n)))"
    using 0(2) tfr_consts_mono unfolding tfr_sst_def by blast
  hence "tfr_set (trms_lsst (𝒜@?n) ∪ pair ` setops_sst (unlabel (𝒜@?n)))"
    using unlabel_append[of 𝒜 ?n] trms_sst_append[of "unlabel 𝒜" "unlabel ?n"]
```

```
          setops_sst_append[of "unlabel 𝒜" "unlabel ?n"]
    by (simp add: setops_sst_def)
  thus "tfr_sst (unlabel ?A)"
    using 0(2) unlabel_append[of ?A ?n]
    unfolding tfr_sst_def by auto
qed

lemma reachable_constraints_typing_result:
  fixes P
  assumes M:
      "has_all_wt_instances_of Γ (⋃T ∈ set P. trms_transaction T) N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. admissible_transaction T"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and A: "𝒜 ∈ reachable_constraints P"
    and ℐ: "constraint_model ℐ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
  shows "∃ℐ_τ. welltyped_constraint_model ℐ_τ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
proof -
  have I:
      "interpretation_subst ℐ" "wf_trms (subst_range ℐ)"
      "constr_sem_stateful ℐ (unlabel (𝒜@[(l, send⟨[attack⟨n⟩]⟩)]))"
    using ℐ unfolding constraint_model_def by metis+

  note 0 = admissible_transaction_is_wellformed_transaction(1,4)[OF admissible_transactionE'(1)]

  have 1: "∀T ∈ set P. wellformed_transaction T"
    using P(1) 0(1) by blast

  have 2: "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
    using P(1) 0(2) unfolding admissible_transaction_terms_def by blast

  have 3: "wf_sst (unlabel 𝒜)" "tfr_sst (unlabel 𝒜)" "wf_trms (trms_lsst 𝒜)"
    using reachable_constraints_tfr[OF _ M P A] reachable_constraints_wf[OF 1(1) 2 A] by metis+

  show ?thesis
    using stateful_typing_result[OF reachable_constraints_typing_result_aux[OF 3] I(1,3)]
    by (metis welltyped_constraint_model_def constraint_model_def)
qed

lemma reachable_constraints_typing_result':
  fixes P
  assumes M:
      "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ` setops_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and A: "𝒜 ∈ reachable_constraints P"
    and ℐ: "constraint_model ℐ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
  shows "∃ℐ_τ. welltyped_constraint_model ℐ_τ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
proof -
  have I:
      "interpretation_subst ℐ" "wf_trms (subst_range ℐ)"
      "constr_sem_stateful ℐ (unlabel (𝒜@[(l, send⟨[attack⟨n⟩]⟩)]))"
    using ℐ unfolding constraint_model_def by metis+
```

```
  have 0: "wf_sst (unlabel 𝒜)" "tfr_sst (unlabel 𝒜)" "wf_trms (trms_lsst 𝒜)"
    using reachable_constraints_tfr'[OF M P(2-3) A]
          reachable_constraints_wf[OF P(1,2) A]
    by metis+

  show ?thesis
    using stateful_typing_result[OF reachable_constraints_typing_result_aux[OF 0] I(1,3)]
    by (metis welltyped_constraint_model_def constraint_model_def)
qed
end

lemma reachable_constraints_transaction_proj:
  assumes "𝒜 ∈ reachable_constraints P"
  shows "proj n 𝒜 ∈ reachable_constraints (map (transaction_proj n) P)"
using assms
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α) show ?case
    using step.hyps(2) reachable_constraints.step[OF
            step.IH _ transaction_decl_subst_proj[OF step.hyps(3)]
            transaction_fresh_subst_proj[OF step.hyps(4)]
            transaction_renaming_subst_proj[OF step.hyps(5)]]
    by (simp add: proj_dual_lsst proj_subst transaction_strand_proj)
qed (simp add: reachable_constraints.init)

context
begin
private lemma reachable_constraints_par_comp_lsst_aux:
  fixes P
  defines "Ts ≡ concat (map transaction_strand P)"
  assumes A: "A ∈ reachable_constraints P"
  shows "∀b ∈ set (dual_lsst A). ∃a ∈ set Ts. ∃δ. b = a ·_lsstp δ ∧
      wt_subst δ ∧ wf_trms (subst_range δ) ∧
      (∀t ∈ subst_range δ. (∃x. t = Var x) ∨ (∃c. t = Fun c []))"
    (is "∀b ∈ set (dual_lsst A). ∃a ∈ set Ts. ?P b a")
using A
proof (induction A rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  define Q where "Q ≡ ?P"
  define ϑ where "ϑ ≡ ξ ∘_s σ ∘_s α"

  let ?R = "λA Ts. ∀b ∈ set A. ∃a ∈ set Ts. Q b a"

  have "wt_subst ϑ" "wf_trms (subst_range ϑ)"
       "∀t ∈ subst_range ϑ. (∃x. t = Var x) ∨ (∃c. t = Fun c [])"
    using transaction_decl_fresh_renaming_substs_wt[OF step.hyps(3-5)]
          transaction_decl_fresh_renaming_substs_range_wf_trms[OF step.hyps(3-5)]
          transaction_decl_fresh_renaming_substs_range'(1)[OF step.hyps(3-5)]
    unfolding ϑ_def by (metis,metis,fastforce)
  hence "?R (dual_lsst (dual_lsst (transaction_strand T)) ·_lsst ϑ) (transaction_strand T)"
    using dual_lsst_self_inverse[of "transaction_strand T"]
    by (auto simp add: Q_def subst_apply_labeled_stateful_strand_def)
  hence "?R (dual_lsst (dual_lsst (transaction_strand T ·_lsst ϑ))) (transaction_strand T)"
    by (metis dual_lsst_subst)
  hence "?R (dual_lsst (dual_lsst (transaction_strand T ·_lsst ϑ))) Ts"
    using step.hyps(2) unfolding Ts_def dual_lsst_def by fastforce
  thus ?case using step.IH unfolding Q_def ϑ_def by auto
qed simp

lemma reachable_constraints_par_comp_lsst:
  fixes P
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Ts ≡ concat (map transaction_strand P)"
  assumes P_pc: "comp_par_comp_lsst public arity Ana Γ Pair Ts M S"
```

```
    and A: "A ∈ reachable_constraints P"
  shows "par_comp_lsst A ((f S) - {m. intruder_synth {} m})"
using par_comp_lsst_if_comp_par_comp_lsst'[OF P_pc, of "dual_lsst A", THEN par_comp_lsst_dual_lsst]
      reachable_constraints_par_comp_lsst_aux[OF A, unfolded Ts_def[symmetric]]
unfolding f_def dual_lsst_self_inverse by fast
end

lemma reachable_constraints_par_comp_constr:
  fixes P f S
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Ts ≡ concat (map transaction_strand P)"
    and "Sec ≡ f S - {m. intruder_synth {} m}"
    and "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ` setops_transaction T"
  assumes M:
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
      "comp_par_comp_lsst public arity Ana Γ Pair Ts M_fun S"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
    and ℐ: "constraint_model ℐ 𝒜"
  shows "∃ℐτ. welltyped_constraint_model ℐτ 𝒜 ∧
            ((∀n. welltyped_constraint_model ℐτ (proj n 𝒜)) ∨
             (∃𝒜' l t. prefix 𝒜' 𝒜 ∧ suffix [(l, receive⟨t⟩)] 𝒜' ∧ strand_leaks_lsst 𝒜' Sec ℐτ))"
proof -
  have ℐ': "constr_sem_stateful ℐ (unlabel 𝒜)" "interpretation_subst ℐ"
    using ℐ unfolding constraint_model_def by blast+

  show ?thesis
    using reachable_constraints_par_comp_lsst[OF P(4)[unfolded Ts_def] 𝒜]
          reachable_constraints_typing_cond_sst[OF M_def M P(1-3) 𝒜]
          par_comp_constr_stateful[OF _ _ ℐ', of Sec]
    unfolding f_def Sec_def welltyped_constraint_model_def constraint_model_def by blast
qed

lemma reachable_constraints_component_leaks_if_composed_leaks:
  fixes Sec Q
  defines "leaks ≡ λ𝒜. ∃ℐτ 𝒜'.
    Q ℐτ ∧ prefix 𝒜' 𝒜 ∧ (∃l' t. suffix [(l', receive⟨t⟩)] 𝒜') ∧ strand_leaks_lsst 𝒜' Sec ℐτ"
  assumes Sec: "∀s ∈ Sec. ¬{} ⊢c s" "ground Sec"
    and composed_leaks: "∃𝒜 ∈ reachable_constraints Ps. leaks 𝒜"
  shows "∃l. ∃𝒜 ∈ reachable_constraints (map (transaction_proj l) Ps). leaks 𝒜"
proof -
  from composed_leaks obtain 𝒜 ℐτ 𝒜' s n where
      𝒜: "𝒜 ∈ reachable_constraints Ps" and
      𝒜': "prefix 𝒜' 𝒜" "constr_sem_stateful ℐτ (proj_unl n 𝒜'@[send⟨[s]⟩])" and
      ℐτ: "Q ℐτ" and
      s: "s ∈ Sec - declassified_lsst 𝒜' ℐτ"
    unfolding leaks_def strand_leaks_lsst_def by fast

  have "¬{} ⊢c s" "s · ℐτ = s" using s Sec by auto
  then obtain B k' u where
      "constr_sem_stateful ℐτ (proj_unl n B@[send⟨[s]⟩])"
      "prefix (proj n B) (proj n 𝒜)" "suffix [(k', receive⟨u⟩)] (proj n B)"
      "s ∈ Sec - declassified_lsst (proj n B) ℐτ"
    using constr_sem_stateful_proj_priv_term_prefix_obtain[OF 𝒜' s]
    unfolding welltyped_constraint_model_def constraint_model_def by metis
  thus ?thesis
    using ℐτ reachable_constraints_transaction_proj[OF 𝒜, of n] proj_idem[of n B]
```

```
    unfolding leaks_def strand_leaks_lsst_def
    by metis
qed

lemma reachable_constraints_preserves_labels:
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "∀a ∈ set 𝒜. ∃T ∈ set P. ∃b ∈ set (transaction_strand T). fst b = fst a"
    (is "∀a ∈ set 𝒜. ∃T ∈ set P. ?P T a")
using 𝒜
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  have "∀a ∈ set (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)). ?P T a"
  proof
    fix a assume a: "a ∈ set (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α))"
    then obtain b where b: "b ∈ set (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)" "a = dual_lsstp b"
      unfolding dual_lsst_def by auto
    then obtain c where c: "c ∈ set (transaction_strand T)" "b = c ·_lsstp ξ ∘_s σ ∘_s α"
      unfolding subst_apply_labeled_stateful_strand_def by auto

    have "?P T c" using c(1) by blast
    hence "?P T b" using c(2) by (simp add: subst_lsstp_fst_eq)
    thus "?P T a" using b(2) dual_lsstp_fst_eq[of b] by presburger
  qed
  thus ?case using step.IH step.hyps(2) by (metis Un_iff set_append)
qed simp

lemma reachable_constraints_preserves_labels':
  assumes P: "∀T ∈ set P. ∀a ∈ set (transaction_strand T). has_LabelN l a ∨ has_LabelS a"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "∀a ∈ set 𝒜. has_LabelN l a ∨ has_LabelS a"
using reachable_constraints_preserves_labels[OF 𝒜] P by fastforce

lemma reachable_constraints_transaction_proj_proj_eq:
  assumes 𝒜: "𝒜 ∈ reachable_constraints (map (transaction_proj l) P)"
  shows "proj l 𝒜 = 𝒜"
    and "prefix 𝒜' 𝒜 ⟹ proj l 𝒜' = 𝒜'"
using 𝒜
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  let ?T = "dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
  note A = reachable_constraints.step[OF step.hyps]

  have P: "∀T ∈ set (map (transaction_proj l) P).
           ∀a ∈ set (transaction_strand T). has_LabelN l a ∨ has_LabelS a"
    using transaction_proj_labels[of l] unfolding list_all_iff by auto

  note * = reachable_constraints_preserves_labels'[OF P A]

  have **: "∀a ∈ set 𝒜'. has_LabelN l a ∨ has_LabelS a"
    when "∀a ∈ set B. has_LabelN l a ∨ has_LabelS a" "prefix 𝒜' B" for B
    using that assms unfolding prefix_def by auto

  note *** = proj_ident[unfolded list_all_iff]

  { case 1 thus ?case using *[THEN ***] by blast }
  { case 2 thus ?case using *[THEN **, THEN ***] by blast }
qed (simp_all add: reachable_constraints.init)

lemma reachable_constraints_transaction_proj_star_proj:
  assumes 𝒜: "𝒜 ∈ reachable_constraints (map (transaction_proj l) P)"
    and k_neq_l: "k ≠ l"
  shows "proj k 𝒜 ∈ reachable_constraints (map transaction_star_proj P)"
using 𝒜
```

```
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  have "map (transaction_proj k) (map (transaction_proj l) P) = map transaction_star_proj P"
    using transaction_star_proj_negates_transaction_proj(2)[OF k_neq_l]
    by fastforce
  thus ?case
    using reachable_constraints_transaction_proj[OF reachable_constraints.step[OF step.hyps], of k]
    by argo
qed (simp add: reachable_constraints.init)

lemma reachable_constraints_aligned_prefix_ex:
  fixes P
  defines "f ≡ λT.
    list_all is_Receive (unlabel (transaction_receive T)) ∧
    list_all is_Check_or_Assignment (unlabel (transaction_checks T)) ∧
    list_all is_Update (unlabel (transaction_updates T)) ∧
    list_all is_Send (unlabel (transaction_send T))"
  assumes P: "list_all f P" "list_all ((list_all (Not ∘ has_LabelS)) ∘ tl ∘ transaction_send) P"
    and s: "¬{} ⊢c s" "fv s = {}"
    and A: "A ∈ reachable_constraints P" "prefix B A"
    and B: "∃l ts. suffix [(l, receive⟨ts⟩)] B"
           "constr_sem_stateful ℐ (unlabel B@[send⟨[s]⟩])"
  shows "∃C ∈ reachable_constraints P.
          prefix C A ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] C) ∧
          declassified_lsst B ℐ = declassified_lsst C ℐ ∧
          constr_sem_stateful ℐ (unlabel C@[send⟨[s]⟩])"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step A T ξ σ α)
  define ϑ where "ϑ ≡ ξ ∘s σ ∘s α"

  let ?T = "dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"

  note AT_reach = reachable_constraints.step[OF step.hyps]

  obtain lb tsb B' where B': "B = B'@[(lb, receive⟨tsb⟩)]" using B(1) unfolding suffix_def by blast

  define decl_ik where "decl_ik ≡ λS::('fun,'atom,'sets,'lbl) prot_strand.
    ⋃{set ts |ts. ⟨⋆, receive⟨ts⟩⟩ ∈ set S} ·set ℐ"

  have decl_ik_append: "decl_ik (M@N) = decl_ik M ∪ decl_ik N" for M N
    unfolding decl_ik_def by fastforce

  have "⟨⋆, receive⟨ts⟩⟩ ∉ set N"
    when "⋆ ∉ fst ` set N" for ts and N::"('fun, 'atom, 'sets, 'lbl) prot_strand"
    using that by force
  hence decl_ik_star: "decl_ik M = decl_ik (M@N)" when "⋆ ∉ fst ` set N" for M N
    using that unfolding decl_ik_def by simp

  have decl_decl_ik: "declassified_lsst S ℐ = {t. decl_ik S ⊢ t}" for S
    unfolding declassified_lsst_alt_def decl_ik_def by blast

  have "f T" using P(1) step.hyps(2) by (simp add: list_all_iff)
  hence "list_all is_Send (unlabel (dual_lsst (transaction_receive T ·lsst ϑ)))"
        "list_all is_Check_or_Assignment (unlabel (dual_lsst (transaction_checks T ·lsst ϑ)))"
        "list_all is_Update (unlabel (dual_lsst (transaction_updates T ·lsst ϑ)))"
        "list_all is_Receive (unlabel (dual_lsst (transaction_send T ·lsst ϑ)))"
    using subst_sst_list_all(2)[of "unlabel (transaction_receive T)" ϑ]
          subst_sst_list_all(11)[of "unlabel (transaction_checks T)" ϑ]
          subst_sst_list_all(10)[of "unlabel (transaction_updates T)" ϑ]
          subst_sst_list_all(1)[of "unlabel (transaction_send T)" ϑ]
          dual_lsst_list_all(1)[of "transaction_receive T ·lsst ϑ"]
          dual_lsst_list_all(11)[of "transaction_checks T ·lsst ϑ"]
```

```
        dual_lsst_list_all(10)[of "transaction_updates T ·lsst ϑ"]
        dual_lsst_list_all(2)[of "transaction_send T ·lsst ϑ"]
  unfolding f_def by (metis unlabel_subst[of _ ϑ])+
hence "¬list_ex is_Receive (unlabel (dual_lsst (transaction_receive T ·lsst ϑ)))"
      "¬list_ex is_Receive (unlabel (dual_lsst (transaction_checks T ·lsst ϑ)))"
      "¬list_ex is_Receive (unlabel (dual_lsst (transaction_updates T ·lsst ϑ)))"
      "list_all is_Receive (unlabel (dual_lsst (transaction_send T ·lsst ϑ)))"
  unfolding list_ex_iff list_all_iff by blast+
then obtain TA TB where T:
    "?T = TA@TB" "¬list_ex is_Receive (unlabel TA)" "list_all is_Receive (unlabel TB)"
    "TB = dual_lsst (transaction_send T ·lsst ϑ)"
  using transaction_dual_subst_unfold[of T ϑ] unfolding ϑ_def by fastforce

have 0: "prefix B (A@TA@TB)" using step.prems B' T by argo

have 1: "prefix B A" when "prefix B (A@TA)"
  using that T(2) B' prefix_prefix_inv
  unfolding list_ex_iff unlabel_def by fastforce

have 2: "⋆ ∉ fst ` set TB2" when "TB = TB1@(l,x)#TB2" for TB1 l x TB2
proof -
  have "k ≠ ⋆" when "k ∈ set (map fst (tl (transaction_send T)))" for k
    using that P(2) step.hyps(2) unfolding list_all_iff by auto
  hence "k ≠ ⋆" when "k ∈ set (map fst (tl TB))" for k
    using that subst_lsst_map_fst_eq[of "tl (transaction_send T)" ϑ]
          dual_lsst_map_fst_eq[of "tl (transaction_send T ·lsst ϑ)"]
    unfolding T(4) dual_lsst_tl subst_lsst_tl by simp
  moreover have "set TB2 ⊆ set (tl TB)"
    using that
    by (metis (no_types, lifting) append_eq_append_conv2 order.eq_iff list.sel(3) self_append_conv
              set_mono_suffix suffix_appendI suffix_tl tl_append2)
  ultimately show ?thesis by auto
qed

have 3: "declassified_lsst TB ℐ = declassified_lsst (TB1@[(l,x)]) ℐ"
  when "TB = TB1@(l,x)#TB2" for TB1 l x TB2
  using decl_ik_star[OF 2[OF that], of "TB1@[(l,x)]"] unfolding that decl_decl_ik by simp

show ?case
proof (cases "prefix B A")
  case False
  hence 4: "¬prefix B (A@TA)" using 1 by blast

  have 5: "∃l ts. suffix [(l, receive⟨ts⟩)] (A@?T)"
  proof -
    have "(lb, receive⟨tsb⟩) ∈ set TB"
      using 0 4 prefix_prefix_inv[OF _ suffixI[OF B'], of "A@TA" TB] by (metis append_assoc)
    hence "receive⟨tsb⟩ ∈ set (unlabel TB)"
      unfolding unlabel_def by force
    hence "∃ts. suffix [receive⟨ts⟩] (unlabel TB)"
      using T(3) unfolding list_all_iff is_Receive_def suffix_def
      by (metis in_set_conv_decomp list.distinct(1) list.set_cases rev_exhaust)
    then obtain TB' ts where "unlabel TB = TB'@[receive⟨ts⟩]" unfolding suffix_def by blast
    then obtain TB'' x where "TB = TB''@[x]" "snd x = receive⟨ts⟩"
      by (metis (no_types, opaque_lifting) append1_eq_conv list.distinct(1) rev_exhaust
            rotate1.simps(2) rotate1_is_Nil_conv unlabel_Cons(2) unlabel_append unlabel_nil)
    then obtain l where "suffix [(l, receive⟨ts⟩)] TB"
      by (metis surj_pair prod.sel(2) suffix_def)
    thus ?thesis
      using T(4) transaction_dual_subst_unfold[of T ϑ]
            suffix_append[of "[(l, receive⟨ts⟩)]"]
      unfolding ϑ_def by metis
  qed
```

```
    obtain TB1 where TB:
        "B = A@TA@TB1@[(lb, receive⟨tsb⟩)]" "prefix (TB1@[(lb, receive⟨tsb⟩)]) TB"
      using 0 4 B' prefix_snoc_obtain[of B' "(lb, receive⟨tsb⟩)" "A@TA" TB thesis]
      by (metis append_assoc)
    then obtain TB2 where TB2: "TB = TB1@(lb, receive⟨tsb⟩)#TB2"
      unfolding prefix_def by fastforce
    hence TB2': "list_all is_Receive (unlabel TB2)"
      using T(3) unfolding list_all_iff is_Receive_def proj_def unlabel_def by auto

    have 6: "constr_sem_stateful 𝓘 (unlabel B)" "ik_lsst B ·_set 𝓘 ⊢ s · 𝓘"
      using B(2) strand_sem_append_stateful[of "{}" "{}" "unlabel B" "[send⟨[s]⟩]" 𝓘]
      by auto

    have "constr_sem_stateful 𝓘 (unlabel (A@TA@TB1@[(lb, receive⟨tsb⟩)]))"
      using 6(1) TB(1) by blast
    hence "constr_sem_stateful 𝓘 (unlabel (A@?T))"
      using T(1) TB2 strand_sem_receive_prepend_stateful[
              of "{}" "{}" "unlabel (A@TA@TB1@[(lb, receive⟨tsb⟩)])" 𝓘,
              OF _ TB2']
      by auto
    moreover have "set (unlabel B) ⊆ set (unlabel (A@?T))"
      using step.prems(1) unfolding prefix_def by force
    hence "ik_lsst (A@?T) ·_set 𝓘 ⊢ s · 𝓘"
      using ideduct_mono[OF 6(2)] subst_all_mono[of _ _ 𝓘]
            ik_sst_set_subset[of "unlabel B" "unlabel (A@?T)"]
      by meson
    ultimately have 7: "constr_sem_stateful 𝓘 (unlabel (A@?T)@[send⟨[s]⟩])"
      using strand_sem_append_stateful[of "{}" "{}" "unlabel (A@?T)" "[send⟨[s]⟩]" 𝓘]
      by auto

    have "declassified_lsst B 𝓘 = declassified_lsst (A@?T) 𝓘"
    proof -
      have "declassified_lsst TB 𝓘 = declassified_lsst (TB1@[(lb, receive⟨tsb⟩)]) 𝓘"
        using 3[of _ lb "receive⟨tsb⟩"] TB(2) unfolding prefix_def by auto
      hence "(decl_ik TB ⊢ t) ⟷ decl_ik (TB1@[(lb,receive⟨tsb⟩)]) ⊢ t" for t
        unfolding TB(1) T(1) decl_decl_ik by blast
      hence "(decl_ik (A@TA@TB) ⊢ t) ⟷ decl_ik (A@TA@TB1@[(lb,receive⟨tsb⟩)]) ⊢ t" for t
        using ideduct_mono_eq[of "decl_ik TB" "decl_ik (TB1@[(lb,receive⟨tsb⟩)])" "decl_ik (A@TA)"]
        by (metis decl_ik_append[of "A@TA"] Un_commute[of _ "decl_ik (A@TA)"] append_assoc)
      thus ?thesis unfolding TB(1) T(1) decl_decl_ik by blast
    qed
    thus ?thesis using step.prems AT_reach B(1) 5 7 by force
  qed (use step.IH prefix_append in blast)
qed (use B(1) suffix_def in simp)

lemma reachable_constraints_secure_if_filter_secure_case:
  fixes f l n
    and P::"('fun,'atom,'sets,'lbl) prot_transaction list"
  defines "has_attack ≡ λP.
      ∃𝒜 ∈ reachable_constraints P. ∃𝓘. constraint_model 𝓘 (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
    and "f ≡ λT. list_ex (λa. is_Update (snd a) ∨ is_Send (snd a)) (transaction_strand T)"
    and "g ≡ λT. transaction_fresh T = [] ⟶ f T"
  assumes att: "has_attack P"
  shows "has_attack (filter g P)"
proof -
  let ?attack = "λA I. constraint_model I (A@[(l, send⟨[attack⟨n⟩]⟩)])"

  define constr' where "constr' ≡
    λ(T::('fun,'atom,'sets,'lbl) prot_transaction,ξ::('fun,'atom,'sets,'lbl) prot_subst,
      σ::('fun,'atom,'sets,'lbl) prot_subst,α::('fun,'atom,'sets,'lbl) prot_subst).
      dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
```

```
define constr where "constr ≡ λTs. concat (map constr' Ts)"

define h where "h ≡
  λ(T::('fun,'atom,'sets,'lbl) prot_transaction,_::('fun,'atom,'sets,'lbl) prot_subst,
    _::('fun,'atom,'sets,'lbl) prot_subst,_::('fun,'atom,'sets,'lbl) prot_subst).
    g T"

obtain A I where A: "A ∈ reachable_constraints P" "?attack A I"
  using att unfolding has_attack_def by blast

obtain Ts where Ts:
    "A = constr Ts"
    "∀B. prefix B Ts ⟶ constr B ∈ reachable_constraints P"
    "∀B T ξ σ α. prefix (B@[(T,ξ,σ,α)]) Ts ⟶
        T ∈ set P ∧ transaction_decl_subst ξ T ∧
        transaction_fresh_subst σ T (trms_lsst (constr B)) ∧
        transaction_renaming_subst α P (vars_lsst (constr B))"
  using reachable_constraints_as_transaction_lists[OF A(1)] constr_def constr'_def by auto

define B where "B ≡ constr (filter h Ts)"

have Ts': "T ∈ set P" when "(T,ξ,σ,α) ∈ set Ts" for T ξ σ α
  using that Ts(3) by (meson prefix_snoc_in_iff)

have constr_Cons: "constr (p#Ts) = constr' p@constr Ts" for p Ts unfolding constr_def by simp

have constr_snoc: "constr (Ts@[p]) = constr Ts@constr' p" for p Ts unfolding constr_def by simp

have 0: "?attack B I" when A_att: "?attack A I"
proof -
  have not_f_T_case: "ik_lsst (constr' p) = {}" "⋀D. dbupd_sst (unlabel (constr' p)) I D = D"
    when not_f_T: "¬(f T)" and p: "p = (T,ξ,σ,α)" for p T ξ σ α
  proof -
    have constr_p: "constr' p = dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
      unfolding p constr'_def by fast

    have "¬is_Receive a" when a: "(l,a) ∈ set (constr' p)" for l a
    proof
      assume "is_Receive a"
      then obtain ts where ts: "a = receive⟨ts⟩" by (cases a) auto
      then obtain ts' where ts':
        "(l,send⟨ts'⟩) ∈ set (transaction_strand T)" "ts = ts' ·_list ξ ∘_s σ ∘_s α"
        using a dual_lsst_steps_iff(1)[of l ts] subst_lsst_memD(2)[of l _ "transaction_strand T"]
        unfolding constr_p by blast
      thus False using not_f_T unfolding f_def list_ex_iff by fastforce
    qed
    thus "ik_lsst (constr' p) = {}" using in_ik_lsst_iff by fastforce

    have "¬is_Update a" when a: "(l,a) ∈ set (constr' p)" for l a
    proof
      assume "is_Update a"
      then obtain t s where ts: "a = insert⟨t,s⟩ ∨ a = delete⟨t,s⟩" by (cases a) auto
      then obtain t' s' where ts':
        "(l,insert⟨t',s'⟩) ∈ set (transaction_strand T) ∨
             (l,delete⟨t',s'⟩) ∈ set (transaction_strand T)"
        "t = t' · ξ ∘_s σ ∘_s α" "s = s' · ξ ∘_s σ ∘_s α"
        using a dual_lsst_steps_iff(4,5)[of l]
          subst_lsst_memD(4,5)[of l _ _ "transaction_strand T"]
        unfolding constr_p by blast
      thus False using not_f_T unfolding f_def list_ex_iff by fastforce
    qed
    thus "⋀D. dbupd_sst (unlabel (constr' p)) I D = D"
      using dbupd_sst_no_upd[of "unlabel (constr' p)" I] by (meson unlabel_mem_has_label)
```

```
qed

have *: "strand_sem_stateful M D (unlabel B) I"
  when "strand_sem_stateful M D (unlabel A) I" for M D
  using that Ts' unfolding Ts(1) B_def
proof (induction Ts arbitrary: M D)
  case (Cons p Ts)
  obtain T ξ σ α where p: "p = (T,ξ,σ,α)" by (cases p) simp
  have T_in: "T ∈ set P" using Cons.prems(2) unfolding p by fastforce

  let ?M' = "M ∪ (ik_lsst (constr' p) ·_set I)"
  let ?D' = "dbupd_sst (unlabel (constr' p)) I D"

  have p_sem: "strand_sem_stateful M D (unlabel (constr' p)) I"
    and IH: "strand_sem_stateful ?M' ?D' (unlabel (constr (filter h Ts))) I"
    using Cons.IH[of ?M' ?D'] Cons.prems
          strand_sem_append_stateful[of M D "unlabel (constr' p)" "unlabel (constr Ts)" I]
    unfolding constr_Cons unlabel_append by fastforce+

  show ?case
  proof (cases "T ∈ set (filter g P)")
    case True
    hence h_p: "filter h (p#Ts) = p#filter h Ts" unfolding h_def p by simp
    show ?thesis
      using p_sem IH strand_sem_append_stateful[of M D "unlabel (constr' p)" _ I]
      unfolding h_p constr_Cons unlabel_append by blast
  next
    case False
    hence not_f: "¬(f T)"
      and not_h: "¬(h p)"
      using T_in unfolding g_def h_def p by auto

    show ?thesis
      using not_h not_f_T_case[OF not_f p] IH
      unfolding constr_Cons unlabel_append by auto
  qed
qed simp

have **: "ik_lsst B = ik_lsst A"
proof
  show "ik_lsst B ⊆ ik_lsst A"
    unfolding Ts(1) B_def constr_def by (induct Ts) (auto simp add: ik_sst_def)

  show "ik_lsst A ⊆ ik_lsst B" using Ts' unfolding Ts(1) B_def
  proof (induction Ts)
    case (Cons p Ts)
    obtain T ξ σ α where p: "p = (T,ξ,σ,α)" by (cases p) simp
    have T_in: "T ∈ set P" using Cons.prems unfolding p by fastforce

    have IH: "ik_lsst (constr Ts) ⊆ ik_lsst (constr (filter h Ts))"
      using Cons.IH Cons.prems by auto

    show ?case
    proof (cases "T ∈ set (filter g P)")
      case True
      hence h_p: "filter h (p#Ts) = p#filter h Ts" unfolding h_def p by simp
      show ?thesis
        using IH unfolding h_p constr_Cons unlabel_append ik_sst_append by blast
    next
      case False
      hence not_f: "¬(f T)"
        and not_h: "¬(h p)"
        using T_in unfolding g_def h_def p by auto
```

```
        show ?thesis
          using not_h not_f_T_case[OF not_f p] IH
          unfolding constr_Cons unlabel_append by auto
      qed
    qed simp
  qed

  show ?thesis
    using A_att *[of "{}" "{}"] ** strand_sem_stateful_if_sends_deduct
          strand_sem_append_stateful[of "{}" "{}" _ "unlabel [(l, send⟨[attack⟨n⟩]⟩)]" I]
    unfolding constraint_model_def unlabel_append by force
qed

have 1: "B ∈ reachable_constraints (filter g P)"
  using A(1) Ts(2,3) unfolding Ts(1) B_def
proof (induction Ts rule: List.rev_induct)
  case (snoc p Ts)
  obtain T ξ σ α where p: "p = (T,ξ,σ,α)" by (cases p) simp

  have constr_p: "constr' p = dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
    unfolding constr'_def p by fastforce

  have T_in: "T ∈ set P"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trms_lsst (constr Ts))"
    and α: "transaction_renaming_subst α P (vars_lsst (constr Ts))"
    using snoc.prems(3) unfolding p by fast+

  have "transaction_fresh_subst s t bb"
    when "transaction_fresh_subst s t aa" "bb ⊆ aa"
    for s t bb aa using that unfolding transaction_fresh_subst_def by fast

  have "trms_lsst (constr (filter h Ts)) ⊆ trms_lsst (constr Ts)"
    unfolding constr_def unlabel_def by fastforce
  hence σ': "transaction_fresh_subst σ T (trms_lsst (constr (filter h Ts)))"
    using σ unfolding transaction_fresh_subst_def by fast

  have "vars_lsst (constr (filter h Ts)) ⊆ vars_lsst (constr Ts)"
    unfolding constr_def unlabel_def vars_sst_def by auto
  hence α': "transaction_renaming_subst α (filter g P) (vars_lsst (constr (filter h Ts)))"
    using α unfolding transaction_renaming_subst_def by auto

  have IH: "constr (filter h Ts) ∈ reachable_constraints (filter g P)"
    using snoc.prems snoc.IH by simp

  show ?case
  proof (cases "h p")
    case True
    hence h_p: "filter h (Ts@[p]) = filter h Ts@[p]" by fastforce
    have T_in': "T ∈ set (filter g P)" using T_in True unfolding h_def p by fastforce
    show ?thesis
      using IH reachable_constraints.step[OF IH T_in' ξ σ' α']
      unfolding h_p constr_snoc constr_p by fast
  next
    case False thus ?thesis using IH by fastforce
  qed
qed (simp add: constr_def)

show ?thesis using 0 1 A(2) unfolding has_attack_def by blast
qed

lemma reachable_constraints_fv_Value_typed:
```

```
  assumes P: "∀T ∈ set P. admissible_transaction' T"
    and A: "𝒜 ∈ reachable_constraints P"
    and x: "x ∈ fv_lsst 𝒜"
  shows "Γ_v x = TAtom Value"
proof -
  obtain T where T: "T ∈ set P" "Γ_v x ∈ Γ_v ` fv_transaction T"
    using x P(1) reachable_constraints_var_types_in_transactions(1)[OF A(1)]
          admissible_transactionE(2)
    by blast

  show ?thesis
    using T(2) admissible_transactionE(3)[OF bspec[OF P(1) T(1)]]
          vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_strand T)"]
    by force
qed

lemma reachable_constraints_fv_Value_const_cases:
  assumes P: "∀T ∈ set P. admissible_transaction' T"
    and A: "𝒜 ∈ reachable_constraints P"
    and I: "welltyped_constraint_model ℐ 𝒜"
    and x: "x ∈ fv_lsst 𝒜"
  shows "(∃n. ℐ x = Fun (Val n) []) ∨ (∃n. ℐ x = Fun (PubConst Value n) [])"
proof -
  have x': "Γ (ℐ x) = TAtom Value" "fv (ℐ x) = {}" "wf_trm (ℐ x)"
    using reachable_constraints_fv_Value_typed[OF P A x] I wt_subst_trm''[of ℐ "Var x"]
    unfolding welltyped_constraint_model_def constraint_model_def by auto

  obtain f where f: "arity f = 0" "ℐ x = Fun f []"
    using TAtom_term_cases[OF x'(3,1)] x' const_type_inv_wf[of _ _ Value] by (cases "ℐ x") force+

  show ?thesis
  proof (cases f)
    case (Fu g) thus ?thesis by (metis f(2) x'(1) Γ_Fu_simps(4)[of g "[]"])
  qed (use f x'(1) in auto)
qed

lemma reachable_constraints_receive_attack_if_attack'':
  assumes P: "∀T ∈ set P. admissible_transaction' T"
    and A: "𝒜 ∈ reachable_constraints P"
    and wt_attack: "welltyped_constraint_model ℐ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
  shows "receive⟨[attack⟨n⟩]⟩ ∈ set (unlabel 𝒜)"
proof -
  have I: "welltyped_constraint_model ℐ 𝒜"
    using welltyped_constraint_model_prefix wt_attack by blast

  show ?thesis
    using wt_attack strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "[send⟨[attack⟨n⟩]⟩]" ℐ]
          reachable_constraints_receive_attack_if_attack'(2)[OF A(1) P I]
    unfolding welltyped_constraint_model_def constraint_model_def by simp
qed

context
begin

private lemma reachable_constraints_initial_value_transaction_aux:
  fixes P::"('fun,'atom,'sets,'lbl) prot" and N::"nat set"
  assumes P: "∀T ∈ set P. admissible_transaction' T"
    and A: "A ∈ reachable_constraints P"
    and P':
      "∀T ∈ set P. ∀(l,a) ∈ set (transaction_strand T). ∀t.
         a ≠ select⟨t,⟨k⟩_s⟩ ∧ a ≠ ⟨t in ⟨k⟩_s⟩ ∧ a ≠ ⟨t not in ⟨k⟩_s⟩ ∧ a ≠ delete⟨t,⟨k⟩_s⟩"
    shows "(l,⟨ac: t ∈ s⟩) ∈ set A ⟹ (∃u. s = ⟨u⟩_s ∧ u ≠ k)" (is "?A A ⟹ ?Q A")
      and "(l,⟨t not in s⟩) ∈ set A ⟹ (∃u. s = ⟨u⟩_s ∧ u ≠ k)" (is "?B A ⟹ ?Q A")
```

```
    and "(l,delete⟨t,s⟩) ∈ set A ⟹ (∃u. s = ⟨u⟩ₛ ∧ u ≠ k)" (is "?C A ⟹ ?Q A")
proof -
  have "(?A A ⟶ ?Q A) ∧ (?B A ⟶ ?Q A) ∧ (?C A ⟶ ?Q A)" (is "?D A") using A
  proof (induction A rule: reachable_constraints.induct)
    case (step 𝒜 T ξ σ α)
    define ϑ where "ϑ ≡ ξ ∘ₛ σ ∘ₛ α"
    let ?T' = "dual_lsst (transaction_strand T ·_lsst ϑ)"

    note T_adm = bspec[OF P step.hyps(2)]
    note T_wf = admissible_transaction_is_wellformed_transaction[OF T_adm]
    note T_P' = bspec[OF P' step.hyps(2)]

    have 0: "?Q ?T'" when A: "?A ?T'"
    proof -
      obtain t' s' where t:
          "(l, ⟨ac: t' ∈ s'⟩) ∈ set (transaction_strand T)" "t = t' · ϑ" "s = s' · ϑ"
        using A dual_lsst_steps_iff(6) subst_lsst_memD(6) by blast
      obtain u where u: "s' = ⟨u⟩ₛ"
        using transaction_selects_are_Value_vars[OF T_wf(1,2), of t' s']
              transaction_inset_checks_are_Value_vars[OF T_adm, of t' s']
              unlabel_in[OF t(1)]
        by (cases ac) auto
      show ?thesis using T_P' t(1,3) unfolding u by (cases ac) auto
    qed

    have 1: "?Q ?T'" when B: "?B ?T'"
    proof -
      obtain t' s' where t:
          "(l, ⟨t' not in s'⟩) ∈ set (transaction_strand T)" "t = t' · ϑ" "s = s' · ϑ"
        using B dual_lsst_steps_iff(7) subst_lsst_memD(9) by blast
      obtain u where u: "s' = ⟨u⟩ₛ"
        using transaction_notinset_checks_are_Value_vars(2)[OF T_adm unlabel_in[OF t(1)]]
        by fastforce
      show ?thesis using T_P' t(1,3) unfolding u by auto
    qed

    have 2: "?Q ?T'" when C: "?C ?T'"
    proof -
      obtain t' s' where t:
          "(l, delete⟨t', s'⟩) ∈ set (transaction_strand T)" "t = t' · ϑ" "s = s' · ϑ"
        using C dual_lsst_steps_iff(5) subst_lsst_memD(5) by blast
      obtain u where u: "s' = ⟨u⟩ₛ"
        using transaction_deletes_are_Value_vars(2)[OF T_wf(1,3) unlabel_in[OF t(1)]] by blast
      show ?thesis using T_P' t(1,3) unfolding u by auto
    qed

    show ?case using 0 1 2 step.IH unfolding ϑ_def by auto
  qed simp
  thus "?A A ⟹ ?Q A" "?B A ⟹ ?Q A" "?C A ⟹ ?Q A" by fast+
qed

lemma reachable_constraints_initial_value_transaction:
  fixes P::"('fun,'atom,'sets,'lbl) prot" and N::"nat set" and k A T_upds
  defines "checks_not_k ≡ λB.
            T_upds ≠ [] ⟶ (
             (∀l t s. (l,⟨t in s⟩) ∈ set (A@B) ⟶ (∃u. s = ⟨u⟩ₛ ∧ u ≠ k)) ∧
             (∀l t s. (l,⟨t not in s⟩) ∈ set (A@B) ⟶ (∃u. s = ⟨u⟩ₛ ∧ u ≠ k)) ∧
             (∀l t s. (l,delete⟨t,s⟩) ∈ set (A@B) ⟶ (∃u. s = ⟨u⟩ₛ ∧ u ≠ k)))"
  assumes P: "∀T ∈ set P. admissible_transaction' T"
    and A: "A ∈ reachable_constraints P"
    and N: "finite N" "∀n ∈ N. ¬(Fun (Val n) [] ⊑_set trms_lsst A)"
    and T:
     "T ∈ set P" "Var x ∈ set T_ts" "Γ_v x = TAtom Value" "fv_set (set T_ts) = {x}"
```

```
      "∀n. ¬(Fun (Val n) [] ⊑set set T_ts)"
      "T = Transaction (λ(). []) [x] [] [] T_upds [(l1,send⟨T_ts⟩)]"
      "T_upds = [] ∨
       (T_upds = [(l2,insert⟨Var x, ⟨k⟩s⟩)] ∧
        (∀T ∈ set P. ∀(l,a) ∈ set (transaction_strand T). ∀t.
          a ≠ select⟨t,⟨k⟩s⟩ ∧ a ≠ ⟨t in ⟨k⟩s⟩ ∧ a ≠ ⟨t not in ⟨k⟩s⟩ ∧ a ≠ delete⟨t,⟨k⟩s⟩))"
  shows "∃B. A@B ∈ reachable_constraints P ∧ B ∈ reachable_constraints P ∧ varslsst B = {} ∧
            (T_upds = [] ⟶ list_all is_Receive (unlabel B)) ∧
            (T_upds ≠ [] ⟶ list_all (λa. is_Insert a ∨ is_Receive a) (unlabel B)) ∧
            (∀n. Fun (Val n) [] ⊑set trmslsst A ⟶ Fun (Val n) [] ∉ iklsst B) ∧
            (∀n. Fun (Val n) [] ⊑set trmslsst B ⟶ Fun (Val n) [] ∈ iklsst B) ∧
            N = {n. Fun (Val n) [] ∈ iklsst B} ∧
            checks_not_k B ∧
            (∀l a. (l,a) ∈ set B ∧ is_Insert a ⟶
                  (l = l2 ∧ (∃n. a = insert⟨Fun (Val n) [],⟨k⟩s⟩)))"
    (is "∃B. A@B ∈ ?reach P ∧ B ∈ ?reach P ∧ ?Q1 B ∧ ?Q2 B ∧ ?Q3 B ∧ ?Q4 B ∧ ?Q5 B ∧ ?Q6 N B ∧
            checks_not_k B ∧ ?Q7 B")
using N
proof (induction N rule: finite_induct)
  case empty
  define B where "B ≡ []::('fun,'atom,'sets,'lbl) prot_constr"

  have 0: "A@B ∈ ?reach P" "B ∈ ?reach P"
    using A unfolding B_def by auto

  have 1: "?Q1 B" "?Q2 B" "?Q3 B" "?Q4 B" "?Q6 {} B"
    unfolding B_def by auto

  have 2: "checks_not_k B"
    using reachable_constraints_initial_value_transaction_aux[OF P 0(1)] T(7)
    unfolding checks_not_k_def by presburger

  have 3: "?Q5 B" "?Q7 B"
    unfolding B_def by simp_all

  show ?case using 0 1 2 3 by blast
next
  case (insert n N)
  obtain B where B:
      "A@B ∈ reachable_constraints P" "B ∈ reachable_constraints P"
      "?Q1 B" "?Q2 B" "?Q3 B" "?Q4 B" "?Q5 B" "?Q6 N B" "checks_not_k B" "?Q7 B"
    using insert.IH insert.prems by blast

  define ξ where "ξ ≡ Var::('fun,'atom,'sets,'lbl) prot_subst"

  define σ where "σ ≡ Var(x := Fun (Val n) []):: ('fun,'atom,'sets,'lbl) prot_subst"

  have σ: "transaction_fresh_subst σ T (trmslsst (A@B))"
  proof (unfold transaction_fresh_subst_def; intro conjI)
    have "subst_range σ = {Fun (Val n) []}" unfolding σ_def by simp
    moreover have "Fun (Val n) [] ∉ subtermsset (trmslsst (A@B))"
      using insert.prems insert.hyps(2) B(7,8) iksst_trmssst_subset[of "unlabel B"]
      unfolding unlabel_append trmssst_append by blast
    ultimately show "∀t ∈ subst_range σ. t ∉ subtermsset (trmslsst (A@B))" by fastforce
  next
    show "subst_domain σ = set (transaction_fresh T)" using T(6) unfolding σ_def by auto
  next
    show "∀t ∈ subst_range σ. t ∉ subtermsset (trms_transaction T)"
      using T(5,7) unfolding σ_def T(6) by fastforce
  qed (force simp add: T(3) σ_def wtsubst_def)+
  hence σ': "transaction_fresh_subst σ T (trmslsst B)"
    unfolding transaction_fresh_subst_def by fastforce
```

```
have ξ: "transaction_decl_subst ξ T"
  using T(6) unfolding ξ_def transaction_decl_subst_def by fastforce

obtain α::"('fun,'atom,'sets,'lbl) prot_subst" where α:
    "transaction_renaming_subst α P (vars_lsst (A@B))"
  unfolding transaction_renaming_subst_def by blast
hence α': "transaction_renaming_subst α P (vars_lsst B)"
  unfolding transaction_renaming_subst_def by auto

define ϑ where "ϑ ≡ ξ ∘s σ ∘s α"

define C where "C ≡ dual_lsst (transaction_strand T ·lsst ϑ)"

have ϑx: "ϑ x = Fun (Val n) []" unfolding ϑ_def ξ_def σ_def subst_compose by force

have "dual_lsst (transaction_receive T ·lsst ϑ) = []" "dual_lsst (transaction_checks T ·lsst ϑ) = []"
  using T(6) unfolding ϑ_def ξ_def σ_def by auto
moreover have
    "(T_upds = [] ∧ dual_lsst (transaction_updates T ·lsst ϑ) = []) ∨
     (T_upds ≠ [] ∧ dual_lsst (transaction_updates T ·lsst ϑ) = [(l2,insert⟨ϑ x, ⟨k⟩s⟩)])"
  using subst_lsst_cons[of "(l2,insert⟨Var x,⟨k⟩s⟩)" "[]" ϑ] T(6,7) unfolding dual_lsst_subst by auto
hence "(T_upds = [] ∧ dual_lsst (transaction_updates T ·lsst ϑ) = []) ∨
       (T_upds ≠ [] ∧ dual_lsst (transaction_updates T ·lsst ϑ) = [(l2,insert⟨Fun (Val n) [],⟨k⟩s⟩)])"
  unfolding ϑ_def ξ_def σ_def by (auto simp: subst_compose)
moreover have "dual_lsst (transaction_send T ·lsst ϑ) = [(l1, receive⟨T_ts ·list ϑ⟩)]"
  using subst_lsst_cons[of "(l1, receive⟨T_ts⟩)" "[]" ϑ] T(6) unfolding dual_lsst_subst by auto
hence "dual_lsst (transaction_send T ·lsst ϑ) = [(l1, receive⟨T_ts ·list ϑ⟩)]"
  by auto
ultimately have C:
    "(T_upds = [] ∧ C = [(l1, receive⟨T_ts ·list ϑ⟩)]) ∨
     (T_upds ≠ [] ∧ C = [(l2, insert⟨Fun (Val n) [],⟨k⟩s⟩), (l1, receive⟨T_ts ·list ϑ⟩)])"
  unfolding C_def transaction_dual_subst_unfold by force

have C': "Fun (Val n) [] ∈ set (T_ts ·list ϑ)" "Fun (Val n) [] ∈ ik_lsst C"
  using T(2) in_ik_sst_iff[of _ "unlabel C"] C
  unfolding ϑ_def ξ_def σ_def by (force, force)

have "fv (t · ϑ) = {}" when t: "t ⊑set set T_ts" for t
proof -
  have  "fv t ⊆ {x}" using t T(4) fv_subset_subterms by blast
  hence "fv (t · ξ ∘s σ) = {}" unfolding ξ_def σ_def by (induct t) auto
  thus ?thesis unfolding ϑ_def by (metis subst_ground_ident_compose(2))
qed
hence 1: "ground (set (T_ts ·list ϑ))" by auto

have 2: "m = n" when m: "Fun (Val m) [] ⊑set ik_lsst C" for m
proof -
  have "Fun (Val m) [] ⊑set set (T_ts ·list ϑ)"
    using m C in_ik_sst_iff[of _ "unlabel C"] by fastforce
  hence *: "Fun (Val m) [] ⊑set set T_ts ·set ϑ" by simp
  show ?thesis using const_subterms_subst_cases[OF *] T(4,5) ϑx by fastforce
qed

have C_trms: "trms_lsst C ⊆ {Fun (Val n) [],⟨k⟩s} ∪ ik_lsst C"
  using C in_ik_sst_iff[of _ "unlabel C"] by fastforce

have 3: "m = n" when m: "Fun (Val m) [] ⊑set trms_lsst C" for m
  using m 2[of m] C_trms by fastforce

have Q1: "?Q1 (B@C)" using B(3) C 1 by auto
have Q2: "?Q2 (B@C)" using B(4) C by force
have Q3: "?Q3 (B@C)" using B(5) C by force
have Q4: "?Q4 (B@C)" using B(6) insert.prems C 2 unfolding unlabel_append ik_sst_append by blast
```

```
  have Q5: "?Q5 (B@C)" using B(7) C' 3 unfolding unlabel_append ik_sst_append trms_sst_append by blast
  have Q6: "?Q6 (insert n N) (B@C)" using B(8) C' 2 unfolding unlabel_append ik_sst_append by blast
  have Q7: "?Q7 (B@C)" using B(10) C by fastforce
  have Q8: "checks_not_k (B@C)" using B(9) C unfolding checks_not_k_def by force

  have "B@C ∈ reachable_constraints P" "A@B@C ∈ reachable_constraints P"
    using reachable_constraints.step[OF B(1) T(1) ξ σ α]
          reachable_constraints.step[OF B(2) T(1) ξ σ' α']
    unfolding ϑ_def[symmetric] C_def[symmetric] by simp_all
  thus ?case using Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 by blast
qed

end
```

3.3.9 Equivalence Between the Symbolic Protocol Model and a Ground Protocol Model

```
context
begin
```

Intermediate Step: Equivalence to a Ground Protocol Model with Renaming

```
private definition "priv_consts_of X = {t. t ⊑_set X ∧ (∃c. t = Fun c [] ∧ ¬public c ∧ arity c = 0)}"

private fun mk_symb where
  "mk_symb (ξ, σ, I, T, α) = dual_lsst((transaction_strand T) ·_lsst ξ ∘_s σ ∘_s α)"

private fun T_symb :: " _ ⇒ ('fun,'atom,'sets,'lbl) prot_constr" where
  "T_symb w = concat (map mk_symb w)"


private definition "narrow σ S = (λx. if x ∈ S then σ x else Var x)"

private fun mk_invαI where
  "mk_invαI n (ξ, σ, I, T) =
    narrow ((var_rename_inv n) ∘_s I) (fv_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s var_rename n))"

private fun invαI where
  "invαI ns w = foldl (∘_s) Var (map2 mk_invαI ns w)"

private fun mk_I where
  "mk_I (ξ, σ, I, T, α) = narrow I (fv_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α))"

private fun comb_I where
  "comb_I w = fold (∘_s) (map mk_I w) (λx. Fun OccursSec [])"

private abbreviation "ground_term t ≡ ground {t}"

private lemma ground_term_def2: "ground_term t ⟷ (fv t = {})"
  by auto

private definition "ground_strand s ≡ fv_lsst s = {}"

private fun ground_step :: "(_, _) stateful_strand_step ⇒ bool" where
  "ground_step s ⟷ fv_sstp s = {}"

private fun ground_lstep :: "_ strand_label × (_, _) stateful_strand_step ⇒ bool" where
  "ground_lstep (l,s) ⟷ fv_sstp s = {}"


private inductive_set ground_protocol_states_aux::
  "('fun,'atom,'sets,'lbl) prot ⇒
   (('fun,'atom,'sets,'lbl) prot_terms ×
    (('fun,'atom,'sets,'lbl) prot_term × ('fun,'atom,'sets,'lbl) prot_term) set
```

```
    × _ set × _ set × _ list) set"
  for P::"('fun,'atom,'sets,'lbl) prot"
where
  init:
  "({},{},{},{},[]) ∈ ground_protocol_states_aux P"
| step:
  "⟦(IK,DB,trms,vars,w) ∈ ground_protocol_states_aux P;
    T ∈ set P;
    transaction_decl_subst ξ T;
    transaction_fresh_subst σ T trms;
    transaction_renaming_subst α P vars;
    A = dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α);
    strand_sem_stateful IK DB (unlabel A) I;
    interpretation_subst I;
    wf_trms (subst_range I)
   ⟧ ⟹ (IK ∪ ((ik_lsst A) ·_set I), dbupd_sst (unlabel A) I DB,
          trms ∪ trms_lsst A, vars ∪ vars_lsst A,
          w@[(ξ, σ, I, T, α)]) ∈ ground_protocol_states_aux P"

private lemma T_symb_append':
  " T_symb (w@w') = T_symb w @ T_symb w'"
proof (induction w arbitrary: w')
  case Nil
  then show ?case
    by auto
next
  case (Cons a w)
  then show ?case
    by auto
qed

private lemma T_symb_append:
  "T_symb (w@[(ξ, σ, I, T, α)]) = T_symb w @ dual_lsst((transaction_strand T) ·_lsst ξ ∘_s σ ∘_s α)"
using T_symb_append'[of w "[(ξ, σ, I, T,α)]"] by auto

private lemma ground_step_subst:
  assumes "ground_step a"
  shows "a = a ·_sstp σ"
using assms
proof (induction a)
  case (NegChecks Y F F')
  then have FY: "fv_pairs F - set Y = {}"
    unfolding ground_step.simps
    unfolding fv_sstp.simps by auto
  {
    have "∀t s. (t,s) ∈ set F ⟶ t · (rm_vars (set Y) σ) = t"
    proof (rule, rule, rule)
      fix t s
      assume "(t, s) ∈ set F"
      then show "t · rm_vars (set Y) σ = t"
        using FY by fastforce
    qed
    moreover
    have "∀t s. (t,s) ∈ set F ⟶ s · (rm_vars (set Y) σ) = s"
    proof (rule, rule, rule)
      fix t s
      assume "(t, s) ∈ set F"
      then show "s · rm_vars (set Y) σ = s"
        using FY by fastforce
    qed
    ultimately
    have "∀f ∈ set F. f ·_p (rm_vars (set Y) σ) = f"
      by auto
```

```
    then have "F = F ·pairs rm_vars (set Y) σ"
      by (metis (no_types, lifting) map_idI split_cong subst_apply_pairs_def)
  }
  moreover
  from NegChecks have F'Y: "fvpairs F' - set Y = {}"
    unfolding ground_step.simps
    unfolding fvsstp.simps by auto
  {
    have "∀t s. (t,s) ∈ set F' ⟶ t · (rm_vars (set Y) σ) = t"
    proof (rule, rule, rule)
      fix t s
      assume "(t, s) ∈ set F'"
      then show "t · rm_vars (set Y) σ = t"
        using F'Y by fastforce
    qed
    moreover
    have "∀t s. (t,s) ∈ set F' ⟶ s · (rm_vars (set Y) σ) = s"
    proof (rule, rule, rule)
      fix t s
      assume "(t, s) ∈ set F'"
      then show "s · rm_vars (set Y) σ = s"
        using F'Y by fastforce
    qed
    ultimately
    have "∀f ∈ set F'. f ·p (rm_vars (set Y) σ) = f"
      by auto
    then have "F' = F' ·pairs rm_vars (set Y) σ"
      by (simp add: map_idI subst_apply_pairs_def)
  }
  ultimately
  show ?case
    by simp
qed (auto simp add: map_idI subst_ground_ident)

private lemma ground_lstep_subst:
  assumes "ground_lstep a"
  shows "a = a ·lsstp σ"
using assms by (cases a) (auto simp add: ground_step_subst)

private lemma subst_apply_term_rm_vars_swap:
  assumes "∀x∈fv t - set X. I x = I' x"
  shows "t · rm_vars (set X) I = t · rm_vars (set X) I'"
using assms by (induction t) auto

private lemma subst_apply_pairs_rm_vars_swap:
  assumes "∀x∈⋃ (fvpair ` set ps) - set X. I x = I' x"
  shows "ps ·pairs rm_vars (set X) I = ps ·pairs rm_vars (set X) I'"
proof -
  have "∀p ∈ set ps. p ·p rm_vars (set X) I = p  ·p rm_vars (set X) I'"
  proof
    fix p
    assume "p ∈ set ps"
    obtain t s where "p = (t,s)"
      by (cases p) auto
    have "∀x∈fv t - set X. I x = I' x"
      by (metis DiffD1 DiffD2 DiffI ‹p = (t, s)› ‹p ∈ set ps› assms fvpairs.elims fvpairs_inI(4))
    then have "t · rm_vars (set X) I = t · rm_vars (set X) I'"
      using subst_apply_term_rm_vars_swap by blast
    have "∀x∈fv s - set X. I x = I' x"
      by (metis DiffD1 DiffD2 DiffI ‹p = (t, s)› ‹p ∈ set ps› assms fvpairs.elims fvpairs_inI(5))
    then have "s · rm_vars (set X) I = s · rm_vars (set X) I'"
      using subst_apply_term_rm_vars_swap by blast
    show "p ·p rm_vars (set X) I = p ·p rm_vars (set X) I'"
```

```
      using ‹p = (t, s)› ‹s · rm_vars (set X) I = s · rm_vars (set X) I'›
            ‹t · rm_vars (set X) I = t · rm_vars (set X) I'›
      by fastforce
  qed
  then show ?thesis
    unfolding subst_apply_pairs_def by auto
qed

private lemma subst_apply_stateful_strand_step_swap:
  assumes "∀x∈fv_sstp T. I x = I' x"
  shows "T ·_sstp I = T ·_sstp I'"
  using assms
proof (induction T)
  case (Send ts)
  then show ?case
    using term_subst_eq by fastforce
next
  case (NegChecks X F G)
  then have "∀x ∈ ⋃(fv_pair ` set F) ∪ ⋃(fv_pair ` set G) - set X. I x = I' x"
    by auto
  then show ?case
    using subst_apply_pairs_rm_vars_swap[of F]
          subst_apply_pairs_rm_vars_swap[of G]
    by auto
qed (simp_all add: term_subst_eq_conv)

private lemma subst_apply_labeled_stateful_strand_step_swap:
  assumes "∀x ∈ fv_sstp (snd T). I x = I' x"
  shows "T ·_lsstp I = T ·_lsstp I'"
using assms subst_apply_stateful_strand_step_swap
by (metis prod.exhaust_sel subst_apply_labeled_stateful_strand_step.simps)

private lemma subst_apply_labeled_stateful_strand_swap:
  assumes "∀x ∈ fv_lsst T. I x = I' x"
  shows "T ·_lsst I = T ·_lsst I'"
  using assms
proof (induction T)
  case Nil
  then show ?case
    by auto
next
  case (Cons a T)
  then show ?case
    using subst_apply_labeled_stateful_strand_step_swap
    by (metis UnCI fv_sst_Cons subst_lsst_cons unlabel_Cons(2))
qed

private lemma transaction_renaming_subst_not_in_fv_lsst:
  fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes "x ∈ fv_lsst A"
  assumes "transaction_renaming_subst α P (vars_lsst A)"
  shows "x ∉ fv_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
proof -
  have 0: "x ∉ fv_lsst (transaction_strand T ·_lsst ξ ∘_s α)"
    when x: "x ∈ vars_lsst A"
      and α: "transaction_renaming_subst α P (vars_lsst A)"
    for x
      and A::"('fun,'atom,'sets,'lbl) prot_constr"
      and ξ α::"('fun,'atom,'sets,'lbl) prot_subst"
  proof -
    have "x ∉ range_vars α"
      using x α transaction_renaming_subst_vars_disj(6) by blast
```

```
    moreover
    have "subst_domain α = UNIV"
      using α transaction_renaming_subst_is_renaming(4) by blast
    ultimately
    show ?thesis
      using subst_fv_dom_img_subset[of _ α] fv_sst_subst_obtain_var subst_compose unlabel_subst
      by (metis (no_types, opaque_lifting) subset_iff top_greatest)
  qed

  have 1: "x ∉ fv_lsst (transaction_strand T ·_lsst ξ ∘_s α)"
    when x: "x ∈ fv_lsst A"
      and α: "transaction_renaming_subst α P (vars_lsst A)"
    for x
      and A::"('fun,'atom,'sets,'lbl) prot_constr"
      and ξ α::"('fun,'atom,'sets,'lbl) prot_subst"
    using α x 0 by (metis Un_iff vars_sst_is_fv_sst_bvars_sst)

  show ?thesis using 1 assms by metis
qed

private lemma wf_comb_I_Nil: "wf_trms (subst_range (comb_I []))"
  by auto

private lemma comb_I_append:
  "comb_I (w @ [(ξ, σ, I, T, α)]) = (mk_I (ξ, σ, I, T, α) ∘_s (comb_I w))"
by auto

private lemma reachable_constraints_if_ground_protocol_states_aux:
  assumes "(IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P"
  shows "T_symb w ∈ reachable_constraints P
         ∧ constr_sem_stateful (comb_I w) (unlabel (T_symb w))
         ∧ IK = ik_lsst ((T_symb w) ·_lsst (comb_I w))
         ∧ DB = dbupd_sst (unlabel ((T_symb w))) (comb_I w) {}
         ∧ trms = trms_lsst (T_symb w)
         ∧ vars = vars_lsst (T_symb w)
         ∧ interpretation_subst (comb_I w)
         ∧ wf_trms (subst_range (comb_I w))"
  using assms
proof (induction rule: ground_protocol_states_aux.induct)
  case init
  show ?case
    using wf_comb_I_Nil by auto
next
  case (step IK DB trms vars w T ξ σ α A I)
  then have step': "T_symb w ∈ reachable_constraints P"
    "strand_sem_stateful {} {} (unlabel (T_symb w)) (comb_I w)"
    "IK = ik_lsst (T_symb w ·_lsst comb_I w)"
    "DB = dbupd_sst (unlabel (T_symb w)) (comb_I w) {}"
    "trms = trms_lsst (T_symb w)"
    "vars = vars_lsst (T_symb w)"
    "interpretation_subst (comb_I w)"
    "wf_trms (subst_range (comb_I w))"
    by auto

  define w' where "w' = w @ [(ξ, σ, I, T, α)]"

  have interps_w: "∀x ∈ fv_lsst (T_symb w). (comb_I w) x = (comb_I w') x"
  proof
    fix x
    assume "x ∈ fv_lsst (T_symb w)"
    then have "x ∉ fv_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
      using step(5) transaction_renaming_subst_not_in_fv_lsst unfolding step'(6) by blast
    then have "mk_I (ξ, σ, I, T, α) x = Var x"
```

```
      unfolding mk_I.simps narrow_def by metis
    then have "comb_I w x = (mk_I (ξ, σ, I, T, α) ∘s (comb_I (w))) x"
      by (simp add: subst_compose)
    then show "comb_I w x = comb_I w' x"
      unfolding w'_def by auto
  qed

  have interps_T: "∀x ∈ fvsst (unlabel (mk_symb (ξ, σ, I, T, α))). I x = (comb_I w') x"
  proof
    fix x
    assume "x ∈ fvsst (unlabel (mk_symb (ξ, σ, I, T, α)))"
    then have a: "x ∈ (fvlsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
      by (metis fvsst_unlabel_duallsst_eq mk_symb.simps)
    have "(comb_I w') x = (mk_I (ξ, σ, I, T, α) ∘s (comb_I (w))) x"
      unfolding w'_def by auto
    also
    have "... = ((mk_I (ξ, σ, I, T, α)) x) · comb_I w"
      unfolding subst_compose by auto
    also
    have "... = (narrow I (fvlsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) x) · comb_I w"
      using a by auto
    also
    have "... = (I x) · comb_I w"
      by (metis a narrow_def)
    also
    have "... = I x"
      by (metis UNIV_I ground_subst_range_empty_fv step.hyps(8) subst_compose
               subst_ground_ident_compose(1))
    finally
    show "I x = (comb_I w') x"
      by auto
  qed

  have "T_symb w' ∈ reachable_constraints P"
  proof -
    have "T_symb w ∈ reachable_constraints P"
      using step'(1) .
    moreover
    have "T ∈ set P"
      using step(2) by auto
    moreover
    have "transaction_decl_subst ξ T"
      using step(3) by auto
    moreover
    have "transaction_fresh_subst σ T (trmslsst (T_symb w))"
      using step(4) step'(5) by auto
    moreover
    have "transaction_renaming_subst α P (varslsst (T_symb w))"
      using step(5) step'(6) by auto
    ultimately
    have "(T_symb w) @ mk_symb (ξ, σ, I, T, α) ∈ reachable_constraints P"
      using reachable_constraints.step[of "T_symb w" P T ξ σ α] by auto
    then show "T_symb w' ∈ reachable_constraints P"
      unfolding w'_def by auto
  qed
  moreover
  have "strand_sem_stateful {} {} (unlabel (T_symb w')) (comb_I w')"
  proof -
    have "strand_sem_stateful {} {} (unlabel (T_symb w)) (comb_I w')"
    proof -
      have "strand_sem_stateful {} {} (unlabel (T_symb w)) (comb_I w)"
        using step'(2) by auto
      then show "strand_sem_stateful {} {} (unlabel (T_symb w)) (comb_I w')"
```

```
    using interps_w strand_sem_model_swap by blast
  qed
  moreover
  have "strand_sem_stateful
          (ik_lsst (T_symb w) ·_set comb_I w')
          (dbupd_sst (unlabel (T_symb w)) (comb_I w') {})
          (unlabel (mk_symb (ξ, σ, I, T, α)))
          (comb_I w')"
  proof -
    have "A = (mk_symb (ξ, σ, I, T, α))"
      unfolding step(6) by auto
    moreover
    have "strand_sem_stateful
            (ik_lsst (T_symb w ·_lsst comb_I w))
            (dbupd_sst (unlabel (T_symb w)) (comb_I w) {})
            (unlabel A)
            I"
      using step'(3) step'(4) step.hyps(7) by force
    moreover
    {
      have "∀x∈fv_set (ik_lsst (T_symb w)). comb_I w x = comb_I w' x"
        using interps_w by (metis fv_ik_sst_is_fv_sst)
      then have "∀x∈fv t. comb_I w x = comb_I w' x" when t: "t ∈ ik_lsst (T_symb w)" for t
        using t by auto
      then have "t · comb_I w' = t · comb_I w" when t: "t ∈ ik_lsst (T_symb w)" for t
        using t term_subst_eq[of t "comb_I w'" "comb_I w"] by metis
      then have "ik_lsst (T_symb w) ·_set comb_I w' = ik_lsst (T_symb w) ·_set comb_I w"
        by auto
      also
      have "... = ik_lsst (T_symb w ·_lsst comb_I w)"
        by (metis ik_sst_subst unlabel_subst)
      finally
      have "ik_lsst (T_symb w) ·_set comb_I w' = ik_lsst (T_symb w ·_lsst comb_I w)"
        by auto
    }
    moreover
    {
      have "dbupd_sst (unlabel (T_symb w)) (comb_I w) {} =
            dbupd_sst (unlabel (T_symb w)) (comb_I w') {}"
        by (metis db_sst_subst_swap[OF interps_w] db_sst_set_is_dbupd_sst empty_set)
    }
    ultimately
    have "strand_sem_stateful
            (ik_lsst (T_symb w) ·_set comb_I w')
            (dbupd_sst (unlabel (T_symb w)) (comb_I w') {})
            (unlabel (mk_symb (ξ, σ, I, T, α)))
            I"
      by force
    then show "strand_sem_stateful
                (ik_lsst (T_symb w) ·_set comb_I w')
                (dbupd_sst (unlabel (T_symb w)) (comb_I w') {})
                (unlabel (mk_symb (ξ, σ, I, T, α)))
                (comb_I w')"
      using interps_T strand_sem_model_swap[of "unlabel (mk_symb (ξ, σ, I, T, α))" I "comb_I w'"]
      by force
  qed
  ultimately
  show "strand_sem_stateful {} {} (unlabel (T_symb w')) (comb_I w')"
    using strand_sem_append_stateful[
            of "{}" "{}" "unlabel (T_symb w)" "unlabel (mk_symb (ξ, σ, I, T, α))" "(comb_I w')"]
    unfolding w'_def by auto
qed
moreover
```

```
have "IK ∪ (ik_lsst A ·set I) = ik_lsst (T_symb w' ·lsst comb_I w')"
proof -
  have AI: "ik_lsst (A ·lsst I) = ik_lsst A ·set I"
    by (metis ik_sst_subst unlabel_subst)

  have "ik_lsst (T_symb w' ·lsst comb_I w') =
         ik_lsst (T_symb w ·lsst comb_I w') ∪ ik_lsst (T_symb [(ξ, σ, I, T, α)] ·lsst comb_I w')"
    unfolding w'_def by (simp add: subst_lsst_append)
  also
  have "... = ik_lsst (T_symb w ·lsst comb_I w') ∪ ik_lsst (T_symb [(ξ, σ, I, T, α)] ·lsst I)"
    by (metis T_symb_append T_symb_append' interps_T mk_symb.simps self_append_conv2
              subst_apply_labeled_stateful_strand_swap)
  also
  have "... = ik_lsst (T_symb w ·lsst comb_I w) ∪ ik_lsst (T_symb [(ξ, σ, I, T, α)] ·lsst I)"
    by (metis interps_w subst_apply_labeled_stateful_strand_swap)
  also
  have "... = IK ∪ ik_lsst (A ·lsst I)"
    using step'(3) step.hyps(6) by auto
  also
  have "... = IK ∪ (ik_lsst A  ·set I)"
    unfolding AI by auto
  finally
  show "IK ∪ (ik_lsst A ·set I) = ik_lsst (T_symb w' ·lsst comb_I w')"
    using step'(3) step(6) T_symb.simps mk_symb.simps by auto
qed
moreover
have "dbupd_sst (unlabel A) I DB = dbupd_sst (unlabel (T_symb w')) (comb_I w') {}"
proof -
  have "dbupd_sst (unlabel A) I DB =
        dbupd_sst (unlabel A) I (dbupd_sst (unlabel (T_symb w)) (comb_I w) {})"
    using step'(4) by auto
  moreover
  have "... = dbupd_sst (unlabel (dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))) I
                     (dbupd_sst (unlabel (T_symb w)) (comb_I w) {})"
    using step(6) by auto
  moreover
  have "... = dbupd_sst (unlabel (mk_symb (ξ, σ, I, T, α))) I
                     (dbupd_sst (unlabel (T_symb w)) (comb_I w) {})"
    by auto
  moreover
  have "... = dbupd_sst (unlabel (mk_symb (ξ, σ, I, T, α))) (comb_I w')
                     (dbupd_sst (unlabel (T_symb w)) (comb_I w) {})"
    by (metis (no_types, lifting) db_sst_subst_swap[OF interps_T] db_sst_set_is_dbupd_sst empty_set)
  moreover
  have "... = dbupd_sst (unlabel (mk_symb (ξ, σ, I, T, α))) (comb_I w')
                     (dbupd_sst (unlabel (T_symb w)) (comb_I w') {})"
    by (metis (no_types, lifting) db_sst_subst_swap[OF interps_w] db_sst_set_is_dbupd_sst empty_set)
  moreover
  have "... = dbupd_sst (unlabel (T_symb w) @ unlabel (mk_symb (ξ, σ, I, T, α))) (comb_I w') {}"
    using dbupd_sst_append by metis
  moreover
  have "... = dbupd_sst (unlabel ((T_symb w) @ mk_symb (ξ, σ, I, T, α))) (comb_I w') {}"
    by auto
  moreover
  have "... = dbupd_sst (unlabel (T_symb w')) (comb_I w') {}"
    unfolding w'_def by auto
  ultimately
  show "dbupd_sst (unlabel A) I DB = dbupd_sst (unlabel (T_symb w')) (comb_I w') {}"
    by auto
qed
moreover
have "trms ∪ trms_lsst A = trms_lsst (T_symb w')"
proof -
```

```
    have "trms ∪ trms_lsst A = trms_lsst (T_symb w) ∪ trms_lsst A"
      using step'(5) by auto
    moreover
    have "... = trms_lsst (T_symb w) ∪ trms_lsst (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α))"
      using step(6) by auto
    moreover
    have "... = trms_lsst (T_symb w) ∪ trms_lsst (T_symb [(ξ, σ, I, T, α)])"
      by auto
    moreover
    have "... = trms_lsst (T_symb w @ T_symb [(ξ, σ, I, T, α)])"
      by auto
    moreover
    have "... = trms_lsst (T_symb w')"
      unfolding w'_def by auto
    ultimately
    show "trms ∪ trms_lsst A = trms_lsst (T_symb w')"
      by auto
  qed
  moreover
  have "vars ∪ vars_lsst A = vars_lsst (T_symb w')"
  proof -
    have "vars ∪ vars_lsst A = vars_lsst (T_symb w) ∪ vars_lsst A"
      using step'(6) by fastforce
    moreover
    have "... = vars_lsst (T_symb w) ∪ vars_lsst (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α))"
      using step(6) by auto
    moreover
    have "... = vars_lsst (T_symb w) ∪ vars_lsst (T_symb [(ξ, σ, I, T, α)])"
      by auto
    moreover
    have "... = vars_lsst (T_symb w @ T_symb [(ξ, σ, I, T, α)])"
      by auto
    moreover
    have "... = vars_lsst (T_symb w')"
      unfolding w'_def by auto
    ultimately
    show "vars ∪ vars_lsst A = vars_lsst (T_symb w')"
      using step(6) by auto
  qed
  moreover
  have interp_comb_I_w': "interpretation_subst (comb_I w')"
    using interpretation_comp(1) step'(7) unfolding w'_def by auto
  moreover
  have "wf_trms (subst_range (comb_I w'))"
  proof
    fix t
    assume "t ∈ subst_range (comb_I w')"
    then have "∃x. x ∈ subst_domain (comb_I w') ∧ t = comb_I w' x"
      by auto
    then obtain x where "x ∈ subst_domain (comb_I w')" "t = comb_I w' x"
      by auto
    then show "wf_trm t"
      by (metis (no_types, lifting) w'_def interp_comb_I_w' comb_I_append ground_subst_dom_iff_img
                mk_I.simps narrow_def step'(8) step.hyps(8) step.hyps(9) subst_compose_def
                wf_trm_Var wf_trm_subst)
  qed
  ultimately
  show ?case
    unfolding w'_def by auto
qed

private lemma ground_protocol_states_aux_if_reachable_constraints:
  assumes "A ∈ reachable_constraints P"
```

```
  assumes "constr_sem_stateful I (unlabel A)"
  assumes "interpretation_subst I"
  assumes "wf_trms (subst_range I)"
  shows "∃w. (ik_lsst A ·_set I, dbupd_sst (unlabel A) I {}, trms_lsst A, vars_lsst A, w)
            ∈ ground_protocol_states_aux P"
using assms
proof (induction rule: reachable_constraints.induct)
  case init
  then show ?case
    using ground_protocol_states_aux.init by auto
next
  case (step 𝒜 T ξ σ α)
  have "∃w. (ik_lsst 𝒜 ·_set I, dbupd_sst (unlabel 𝒜) I {}, trms_lsst 𝒜, vars_lsst 𝒜, w)
            ∈ ground_protocol_states_aux P"
    by (metis local.step(6,7,8,9) step.IH strand_sem_append_stateful unlabel_append)
  then obtain w where w_p:
      "(ik_lsst 𝒜 ·_set I, dbupd_sst (unlabel 𝒜) I {}, trms_lsst 𝒜, vars_lsst 𝒜, w)
        ∈ ground_protocol_states_aux P"
    by auto

  define w' where "w' = w@[(ξ, σ, I, T, α)]"
  define 𝒜' where "𝒜' = 𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"

  let ?T = "unlabel (dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α))"

  have "T ∈ set P"
    using step.hyps(2) .
  moreover
  have "transaction_decl_subst ξ T"
    using step.hyps(3) .
  moreover
  have "transaction_fresh_subst σ T (trms_lsst 𝒜)"
    using step.hyps(4) .
  moreover
  have "transaction_renaming_subst α P (vars_lsst 𝒜)"
    using step.hyps(5) .
  moreover
  have "strand_sem_stateful (ik_lsst 𝒜 ·_set I) (dbupd_sst (unlabel 𝒜) I {}) ?T I"
    using step(7) strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" ?T I]
    by auto
  moreover
  have "interpretation_subst I"
    using assms(3) .
  moreover
  have "wf_trms (subst_range I)"
    using step.prems(3) by fastforce
  ultimately
  have "((ik_lsst 𝒜 ·_set I) ∪ (ik_sst ?T ·_set I),
         dbupd_sst ?T I (dbupd_sst (unlabel 𝒜) I {}),
         trms_lsst 𝒜 ∪ trms_sst ?T,
         vars_lsst 𝒜 ∪ vars_sst ?T,
         w@[(ξ, σ, I, T, α)])
      ∈ ground_protocol_states_aux P"
    using ground_protocol_states_aux.step[
            OF w_p,
            of T ξ σ α "dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)" I]
    by metis
  moreover
  have "ik_lsst 𝒜' ·_set I = (ik_lsst 𝒜 ·_set I) ∪ (ik_sst ?T ·_set I)"
    unfolding 𝒜'_def by auto
  moreover
  have "dbupd_sst (unlabel 𝒜') I {} = dbupd_sst ?T I (dbupd_sst (unlabel 𝒜) I {})"
    unfolding 𝒜'_def by (simp add: dbupd_sst_append)
```

```
  moreover
  have "trmslsst 𝒜' = trmslsst 𝒜 ∪ trmssst ?T"
    unfolding 𝒜'_def by auto
  moreover
  have "varslsst 𝒜' = varslsst 𝒜 ∪ varssst ?T"
    unfolding 𝒜'_def by auto
  ultimately
  have "(iklsst 𝒜' ·set I, dbupdsst (unlabel 𝒜') I {}, trmslsst 𝒜', varslsst 𝒜', w')
          ∈ ground_protocol_states_aux P"
    using w'_def by auto
  then show ?case
    unfolding 𝒜'_def w'_def by auto
qed

private lemma protocol_model_equivalence_aux1:
  "{(IK, DB) | IK DB. ∃w trms vars. (IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P} =
   {(iklsst (A ·lsst I), dbupdsst (unlabel A) I {}) | A I.
      A ∈ reachable_constraints P ∧ strand_sem_stateful {} {} (unlabel A) I ∧
      interpretationsubst I ∧ wftrms (subst_range I)}"
proof (rule; rule; rule)
  fix IK DB
  assume "(IK, DB) ∈
          {(IK, DB) | IK DB. ∃w trms vars. (IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P}"
  then have "∃w trms vars. (IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P"
    by auto
  then obtain w trms vars where "(IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P"
    by auto
  then have reachable:
      "T_symb w ∈ reachable_constraints P"
      "strand_sem_stateful {} {} (unlabel (T_symb w)) (comb_I w)"
      "IK = iklsst (T_symb w ·lsst comb_I w)"
      "DB = dbupdsst (unlabel (T_symb w)) (comb_I w) {}"
      "trms = trmslsst (T_symb w)"
      "vars = varslsst (T_symb w)"
      "interpretationsubst (comb_I w)"
      "wftrms (subst_range (comb_I w))"
    using reachable_constraints_if_ground_protocol_states_aux[of IK DB trms vars w P] by auto
  then have
      "IK = iklsst (T_symb w ·lsst (comb_I w))"
      "DB = dbupdsst (unlabel (T_symb w)) (comb_I w) {}"
      "T_symb w ∈ reachable_constraints P"
      "strand_sem_stateful {} {} (unlabel (T_symb w)) (comb_I w)"
      "interpretationsubst (comb_I w) ∧ wftrms (subst_range (comb_I w))"
    by auto
  then show "∃A I. (IK, DB) = (iklsst (A ·lsst I), dbupdsst (unlabel A) I {}) ∧
                 A ∈ reachable_constraints P ∧ strand_sem_stateful {} {} (unlabel A) I ∧
                 interpretationsubst I ∧ wftrms (subst_range I)"
    by blast
next
  fix IK DB
  assume "(IK, DB) ∈
          {(iklsst (A ·lsst I), dbupdsst (unlabel A) I {}) | A I.
            A ∈ reachable_constraints P ∧ strand_sem_stateful {} {} (unlabel A) I ∧
            interpretationsubst I ∧ wftrms (subst_range I)}"
  then obtain A I where A_I_p:
     "IK = iklsst (A ·lsst I)"
     "DB = dbupdsst (unlabel A) I {}"
     "A ∈ reachable_constraints P"
     "strand_sem_stateful {} {} (unlabel A) I"
     "interpretationsubst I"
     "wftrms (subst_range I)"
    by auto
  then have "∃w. (iklsst A ·set I, dbupdsst (unlabel A) I {}, trmslsst A, varslsst A, w)
```

```
                 ∈ ground_protocol_states_aux P"
    using ground_protocol_states_aux_if_reachable_constraints[of A P I] by auto
  then have "∃w. (ik_lsst A ·set I, DB, trms_lsst A, vars_lsst A, w) ∈ ground_protocol_states_aux P"
    using A_I_p by blast
  then have "∃w. (ik_sst (unlabel A ·sst I), DB, trms_lsst A, vars_lsst A, w) ∈ ground_protocol_states_aux
P"
    by (simp add: ik_sst_subst)
  then have "(∃w trms vars. (IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P)"
    by (metis (no_types) A_I_p(1) unlabel_subst)
  then show "∃IK' DB'. (IK, DB) = (IK', DB') ∧
                      (∃w trms vars. (IK', DB', trms, vars, w) ∈ ground_protocol_states_aux P)"
    by auto
qed
```

The Protocol Model Equivalence Proof

```
private lemma subst_ground_term_ident:
  assumes "ground_term t"
  shows "t · I = t"
using assms by (simp add: subst_ground_ident)

private lemma subst_comp_rm_vars_eq:
  fixes δ :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes α :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "subst_domain δ = set X ∧ ground (subst_range δ)"
  shows "(δ ∘s α) = (δ ∘s (rm_vars (set X) α))"
proof (rule ext)
  fix x
  show "(δ ∘s α) x = (δ ∘s rm_vars (set X) α) x"
  proof (cases "x ∈ set X")
    case True
    have gt: "ground_term (δ x)"
      using True assms by auto

    have "(δ ∘s α) x = (δ x) · α"
      using subst_compose by metis
    also
    have "... = δ x"
      using gt subst_ground_term_ident by blast
    also
    have "... = (δ x) · (rm_vars (set X) α)"
      using gt subst_ground_term_ident by fastforce
    also
    have "... = (δ ∘s (rm_vars (set X) α)) x"
      using subst_compose by metis
    ultimately
    show ?thesis
      by auto
  next
    case False
    have delta_x: "δ x = Var x"
      using False assms by blast
    have "(rm_vars (set X) α) x = α x"
      using False by auto

    have "(δ ∘s α) x = (δ x) · α"
      using subst_compose by metis
    also
    have "... = (Var x) · α"
      using delta_x by presburger
    also
    have "... = (Var x) · (rm_vars (set X) α)"
```

```
      using False by force
    also
    have "... = (δ x) · (rm_vars (set X) α)"
      using delta_x by presburger
    also
    have "... = (δ ∘s (rm_vars (set X) α)) x"
      using subst_compose by metis
    finally
    show ?thesis
      by auto
  qed
qed

private lemma subst_comp_rm_vars_commute:
  assumes "∀x∈set X. ∀y. α y ≠ Var x"
  assumes "subst_range α ⊆ range Var"
  assumes "subst_domain δ = set X"
  assumes "ground (subst_range δ)"
  shows "(δ ∘s (rm_vars (set X) α)) = (rm_vars (set X) α ∘s δ)"
proof (rule ext)
  fix x
  show "(δ ∘s rm_vars (set X) α) x = (rm_vars (set X) α ∘s δ) x"
  proof (cases "x ∈ set X")
    case True
    then have gt: "ground_term (δ x)"
      using True assms(3,4) by auto

    have "(δ ∘s (rm_vars (set X) α)) x = δ x · rm_vars (set X) α"
      by (simp add: subst_compose)
    also
    have "... = δ x"
      using gt by auto
    also
    have "... = ((rm_vars (set X) α) x) · δ"
      by (simp add: True)
    also
    have "... = (rm_vars (set X) α ∘s δ) x"
      by (simp add: subst_compose)
    finally
    show ?thesis .
  next
    case False
    have δ_x: "δ x = Var x"
      using False assms(3) by blast
    obtain y where y_p: "α x = Var y"
      by (meson assms(2) image_iff subsetD subst_imgI)
    then have "y ∉ set X"
      using assms(1) by blast
    then show ?thesis
      using assms(3,4) subst_domI False δ_x y_p
      by (metis (mono_tags, lifting) subst_comp_notin_dom_eq subst_compose)
  qed
qed

private lemma negchecks_model_substitution_lemma_1:
  fixes α :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "negchecks_model (α ∘s I) DB X F F'"
  assumes "subst_range α ⊆ range Var"
  assumes "∀x∈set X. ∀y. α y ≠ Var x"
  shows  "negchecks_model I DB X (F ·pairs rm_vars (set X) α) (F' ·pairs rm_vars (set X) α)"
  unfolding negchecks_model_def
proof (rule, rule)
```

```
  fix δ :: "('fun,'atom,'sets,'lbl) prot_subst"
  assume a: "subst_domain δ = set X ∧ ground (subst_range δ)"

  have "(∃(t, s)∈set F. t · δ ∘s (α ∘s I) ≠ s · δ ∘s (α ∘s I)) ∨
          (∃(t, s)∈set F'. (t, s) ·p δ ∘s (α ∘s I) ∉ DB)"
    using a assms(1) unfolding negchecks_model_def by auto
  then show "(∃(t, s)∈set (F ·pairs rm_vars (set X) α). t · δ ∘s I ≠ s · δ ∘s I) ∨
               (∃(t, s)∈set (F' ·pairs rm_vars (set X) α). (t, s) ·p δ ∘s I ∉ DB)"
  proof
    assume "∃(t, s)∈set F. t · δ ∘s (α ∘s I) ≠ s · δ ∘s (α ∘s I)"
    then obtain t s where t_s_p: "(t, s)∈set F" "t · δ ∘s (α ∘s I) ≠ s · δ ∘s (α ∘s I)"
      by auto
    from this(2) have "t · δ ∘s ((rm_vars (set X) α) ∘s I) ≠ s · δ ∘s ((rm_vars (set X) α) ∘s I)"
      using assms(3) a using subst_comp_rm_vars_eq[of δ X α] subst_compose_assoc
      by (metis (no_types, lifting))
    then have "t · (rm_vars (set X) α) ∘s  ( δ ∘s I) ≠ s · (rm_vars (set X) α) ∘s (δ ∘s I)"
      using subst_comp_rm_vars_commute[of X α δ, OF assms(3) assms(2)] a
      by (metis (no_types, lifting) subst_compose_assoc[symmetric])
    then have "t · (rm_vars (set X) α) · ( δ ∘s I) ≠ s · (rm_vars (set X) α) · (δ ∘s I)"
      by auto
    moreover
    have "(t · rm_vars (set X) α, s · rm_vars (set X) α) ∈ set (F ·pairs rm_vars (set X) α)"
      using subst_apply_pairs_pset_subst t_s_p(1) by fastforce
    ultimately
    have "∃(t, s)∈set (F ·pairs rm_vars (set X) α). t · δ ∘s I ≠ s · δ ∘s I"
      by auto
    then show ?thesis
      by auto
  next
    assume "∃(t, s)∈set F'. (t, s) ·p δ ∘s (α ∘s I) ∉ DB"
    then obtain t s where t_s_p: "(t, s) ∈ set F'" "(t, s) ·p δ ∘s (α ∘s I) ∉ DB"
      by auto
    from this(2) have "(t, s) ·p δ ∘s (rm_vars (set X) α ∘s I) ∉ DB"
      using assms(3) a subst_comp_rm_vars_eq[OF a]
      by (metis (no_types, lifting) case_prod_conv subst_subst_compose)
    then have "(t, s) ·p rm_vars (set X) α ∘s (δ ∘s I) ∉ DB"
      using a subst_comp_rm_vars_commute[of X α δ, OF assms(3) assms(2)]
      by (metis (no_types, lifting) case_prod_conv subst_compose_assoc)
    then have "(t · rm_vars (set X) α, s · rm_vars (set X) α) ·p δ ∘s I ∉ DB"
      by auto
    moreover
    have "(t · rm_vars (set X) α, s · rm_vars (set X) α) ∈ set (F' ·pairs rm_vars (set X) α)"
      using t_s_p(1) subst_apply_pairs_pset_subst by fastforce
    ultimately
    have "(∃(t, s) ∈ set (F' ·pairs rm_vars (set X) α). (t, s) ·p δ ∘s I ∉ DB)"
      by auto
    then show ?thesis
      by auto
  qed
qed

private lemma negchecks_model_substitution_lemma_2:
  fixes α :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "negchecks_model I DB X (F ·pairs rm_vars (set X) α) (F' ·pairs rm_vars (set X) α)"
  assumes "subst_range α ⊆ range Var"
  assumes "∀x∈set X. ∀y. α y ≠ Var x"
  shows "negchecks_model (α ∘s I) DB X F F'"
  unfolding negchecks_model_def
proof (rule, rule)
  fix δ :: "('fun,'atom,'sets,'lbl) prot_subst"
  assume a: "subst_domain δ = set X ∧ ground (subst_range δ)"
```

```
  have "(∃(t, s)∈set (F ·pairs rm_vars (set X) α). t · δ ∘s  I ≠ s · δ ∘s (I)) ∨
          (∃(t, s)∈set (F' ·pairs rm_vars (set X) α). (t, s) ·p δ ∘s I ∉ DB)"
    using a assms(1)unfolding negchecks_model_def by auto
  then show "(∃(t, s)∈set F. t · δ ∘s (α ∘s I) ≠ s · δ ∘s (α ∘s I)) ∨
                (∃(t, s)∈set F'. (t, s) ·p δ ∘s (α ∘s I) ∉ DB)"
  proof
    assume "∃(t, s)∈set (F ·pairs rm_vars (set X) α). t · δ ∘s  I ≠ s · δ ∘s (I)"
    then obtain t s where t_s_p: "(t, s) ∈ set (F ·pairs rm_vars (set X) α)" "t · δ ∘s I ≠ s · δ ∘s I"
      by auto
    then have "∃t' s'. t = t' · rm_vars (set X) α ∧ s = s' · rm_vars (set X) α ∧ (t',s') ∈ set F"
      unfolding subst_apply_pairs_def by auto
    then obtain t' s' where t'_s'_p:
      "t = t' · rm_vars (set X) α"
      "s = s' · rm_vars (set X) α"
      "(t',s') ∈ set F"
      by auto
    then have "t' · rm_vars (set X) α · δ ∘s I ≠ s' · rm_vars (set X) α · δ ∘s I"
      using t_s_p by auto
    then have "t' ·  δ · rm_vars (set X) α ∘s I ≠ s' · δ · rm_vars (set X) α  ∘s I"
      using a subst_comp_rm_vars_commute[OF assms(3,2)] by (metis (no_types, lifting) subst_subst)
    then have "t' ·  δ · α ∘s I ≠ s' · δ · α  ∘s I"
      using subst_comp_rm_vars_eq[OF a] by (metis (no_types, lifting) subst_subst)
    moreover
    from t_s_p(1) have "(t', s') ∈ set F"
      using subst_apply_pairs_pset_subst t'_s'_p by fastforce
    ultimately
    have "∃(t, s)∈set F. t · δ ∘s (α ∘s I) ≠ s · δ ∘s (α ∘s I)"
      by auto
    then show ?thesis
      by auto
  next
    assume "∃(t, s)∈set (F' ·pairs rm_vars (set X) α). (t, s) ·p δ ∘s I ∉ DB"
    then obtain t s where t_s_p:
      "(t, s) ∈ set (F' ·pairs rm_vars (set X) α)"
      "(t · δ ∘s I, s · δ ∘s I) ∉ DB"
      by auto
    then have "∃t' s'. t = t' · rm_vars (set X) α ∧ s = s' · rm_vars (set X) α ∧ (t',s') ∈ set F'"
      unfolding subst_apply_pairs_def by auto
    then obtain t' s' where t'_s'_p:
      "t = t' · rm_vars (set X) α"
      "s = s' · rm_vars (set X) α"
      "(t',s') ∈ set F'"
      by auto
    then have "(t' · rm_vars (set X) α · δ ∘s I, s' · rm_vars (set X) α · δ ∘s I) ∉ DB"
      using t_s_p by auto
    then have "(t' ·  δ · rm_vars (set X) α ∘s I, s' · δ · rm_vars (set X) α  ∘s I) ∉ DB"
      using a subst_comp_rm_vars_commute[OF assms(3,2)]
      by (metis (no_types, lifting) subst_subst)
    then have "(t' ·  δ · α ∘s I , s' · δ · α  ∘s I) ∉ DB"
      using subst_comp_rm_vars_eq[OF a] by (metis (no_types, lifting) subst_subst)
    moreover
    from t_s_p(1) have "(t', s') ∈ set F'"
      using subst_apply_pairs_pset_subst t'_s'_p by fastforce
    ultimately
    have "∃(t, s)∈set F'. (t, s) ·p δ ∘s (α ∘s I) ∉ DB"
      by auto
    then show ?thesis
      by auto
  qed
qed

private lemma negchecks_model_substitution_lemma:
  fixes α :: "('fun,'atom,'sets,'lbl) prot_subst"
```

```
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "subst_range α ⊆ range Var"
  assumes "∀x∈set X. ∀y. α y ≠ Var x"
  shows "negchecks_model (α ∘s I) DB X F F' ⟷
           negchecks_model I DB X (F ·pairs rm_vars (set X) α) (F' ·pairs rm_vars (set X) α)"
using assms negchecks_model_substitution_lemma_1[of α I DB X F F']
      negchecks_model_substitution_lemma_2[of I DB X F α F'] assms
by auto

private lemma strand_sem_stateful_substitution_lemma:
  fixes α :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "subst_range α ⊆ range Var"
  assumes "∀x ∈ bvarssst T. ∀y. α y ≠ Var x"
  shows "strand_sem_stateful IK DB (T ·sst α) I = strand_sem_stateful IK DB T (α ∘s I)"
using assms
proof (induction T arbitrary: IK DB)
  case Nil
  then show ?case by auto
next
  case (Cons a T)
  then show ?case
  proof (induction a)
    case (Receive ts)
    have "((λx. x · α · I) ` (set ts)) ∪ IK = ((λt. t · α) ` set ts ·set I) ∪ IK"
      by blast
    then show ?case
      using Receive by (force simp add: subst_sst_cons)
  next
    case (NegChecks X F F')

    have bounds: "∀x∈bvarssst T. ∀y. α y ≠ Var x"
      using NegChecks by auto

    have "∀x∈bvarssst ([∀X⟨∨≠: F ∨∉: F'⟩]). ∀y. α y ≠ Var x"
      using NegChecks by auto
    then have bounds2: "∀x∈set X. ∀y. α y ≠ Var x"
      by simp

    have "negchecks_model I DB X (F ·pairs rm_vars (set X) α) (F' ·pairs rm_vars (set X) α) ⟷
            negchecks_model (α ∘s I) DB X F F'"
      using NegChecks.prems(2) bounds2 negchecks_model_substitution_lemma by blast
    moreover
    have "strand_sem_stateful IK DB (T ·sst α) I ⟷ strand_sem_stateful IK DB T (α ∘s I)"
      using Cons NegChecks(2) bounds by blast
    ultimately
    show ?case
      by (simp add: subst_sst_cons)
  qed (force simp add: subst_sst_cons)+
qed

private lemma ground_subst_rm_vars_subst_compose_dist:
  assumes "ground (subst_range ξσ)"
  shows "(rm_vars (set X) (ξσ ∘s α)) = (rm_vars (set X) ξσ ∘s rm_vars (set X) α)"
proof (rule ext)
  fix x
  show "rm_vars (set X) (ξσ ∘s α) x = (rm_vars (set X) ξσ ∘s rm_vars (set X) α) x"
  proof (cases "x ∈ set X")
    case True
    then show ?thesis
      by (simp add: subst_compose)
  next
    case False
```

```
    note False_outer = False
    show ?thesis
    proof (cases "x ∈ subst_domain ξσ")
      case True
      then show ?thesis
        by (metis (mono_tags, lifting) False assms ground_subst_range_empty_fv
              subst_ground_ident_compose(1))
    next
      case False
      have "ξσ x = Var x"
        using False by blast
      then show ?thesis
        using False_outer by (simp add: subst_compose)
    qed
  qed
qed

private lemma stateful_strand_ground_subst_comp:
  assumes "ground (subst_range ξσ)"
  shows "T ·sst ξσ ∘s α = (T ·sst ξσ) ·sst α"
using assms by (meson disjoint_iff ground_subst_no_var stateful_strand_subst_comp)

private lemma labelled_stateful_strand_ground_subst_comp:
  assumes "ground (subst_range ξσ)"
  shows "T ·lsst ξσ ∘s α = (T ·lsst ξσ) ·lsst α"
using assms by (metis Int_empty_left ground_range_vars labeled_stateful_strand_subst_comp)

private lemma transaction_fresh_subst_ground_subst_range:
  assumes "transaction_fresh_subst σ T trms"
  shows "ground (subst_range σ)"
using assms by (metis range_vars_alt_def transaction_fresh_subst_range_vars_empty)

private lemma transaction_decl_subst_ground_subst_range:
  assumes "transaction_decl_subst ξ T"
  shows "ground (subst_range ξ)"
proof -
  have ξ_ground: "∀x ∈ subst_domain ξ. ground_term (ξ x)"
    using assms transaction_decl_subst_domain transaction_decl_subst_grounds_domain by force
  show ?thesis
  proof (rule ccontr)
    assume "fvset (subst_range ξ) ≠ {}"
    then have "∃x ∈ subst_domain ξ. fv (ξ x) ≠ {}"
      by auto
    then obtain x where x_p: "x ∈ subst_domain ξ ∧ fv (ξ x) ≠ {}"
      by meson
    moreover
    have "ground_term (ξ x)"
      using ξ_ground x_p by auto
    ultimately
    show "False"
      by auto
  qed
qed

private lemma fresh_transaction_decl_subst_ground_subst_range:
  assumes "transaction_fresh_subst σ T trms"
  assumes "transaction_decl_subst ξ T"
  shows "ground (subst_range (ξ ∘s σ))"
proof -
  have "ground (subst_range ξ)"
    using assms transaction_decl_subst_ground_subst_range by blast
  moreover
  have "ground (subst_range σ)"
```

```
    using assms
    using transaction_fresh_subst_ground_subst_range by blast
  ultimately
  show "ground (subst_range (ξ ∘s σ))"
    by (metis (no_types, opaque_lifting) Diff_iff all_not_in_conv empty_iff empty_subsetI
          range_vars_alt_def range_vars_subst_compose_subset subset_antisym sup_bot.right_neutral)
qed

private lemma strand_sem_stateful_substitution_lemma':
  assumes "transaction_renaming_subst α P vars"
  assumes "transaction_fresh_subst σ T trms"
  assumes "transaction_decl_subst ξ T"
  assumes "finite vars"
  assumes "T ∈ set P"
  shows "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))) I
    ⟷ strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·lsst ξ ∘s σ))) (α ∘s I)"
proof -
  have α_Var: "subst_range α ⊆ range Var"
    using assms(1) transaction_renaming_subst_is_renaming(5) by blast
  have "(∀x∈vars_lsst (transaction_strand T). ∀y. α y ≠ Var x)"
    using assms(4,2) transaction_renaming_subst_vars_transaction_neq assms(1) assms(5) by blast
  then have "(∀x∈bvars_lsst (transaction_strand T). ∀y. α y ≠ Var x)"
    by (metis UnCI vars_sst_is_fv_sst_bvars_sst)
  then have T_Vars: "(∀x∈bvars_lsst (dual_lsst (transaction_strand T ·lsst ξ ∘s σ)). ∀y. α y ≠ Var x)"
    by (metis bvars_lsst_subst bvars_sst_unlabel_dual_lsst_eq)

  have ground_ξ_σ: "ground (subst_range (ξ ∘s σ))"
    using fresh_transaction_decl_subst_ground_subst_range
    using assms(2) assms(3) by blast

  from assms(1) ground_ξ_σ have
    "unlabel (dual_lsst (transaction_strand T) ·lsst ξ ∘s σ ∘s α) =
    unlabel ((dual_lsst (transaction_strand T) ·lsst ξ ∘s σ) ·lsst α)"
    using stateful_strand_ground_subst_comp[of _ "unlabel (dual_lsst (transaction_strand T))"]
    by (simp add: dual_lsst_subst unlabel_subst)
  then show ?thesis
    using strand_sem_stateful_substitution_lemma α_Var T_Vars
    by (metis dual_lsst_subst subst_lsst_unlabel)
qed

inductive_set ground_protocol_states::
  "('fun,'atom,'sets,'lbl) prot ⇒
   (('fun,'atom,'sets,'lbl) prot_terms ×
    (('fun,'atom,'sets,'lbl) prot_term × ('fun,'atom,'sets,'lbl) prot_term) set
    ×
   _ set
   ) set"
  for P::"('fun,'atom,'sets,'lbl) prot"
where
  init:
  "({},{},{}) ∈ ground_protocol_states P"
| step:
  "⟦(IK,DB,consts) ∈ ground_protocol_states P;
    T ∈ set P;
    transaction_decl_subst ξ T;
    transaction_fresh_subst σ T consts;
    A = dual_lsst (transaction_strand T ·lsst ξ ∘s σ);
    strand_sem_stateful IK DB (unlabel A) I;
    interpretation_subst I;
    wf_trms (subst_range I)
   ⟧ ⟹ (IK ∪ ((ik_lsst A) ·set I), dbupd_sst (unlabel A) I DB,
          consts ∪ {t. t ⊑set trms_lsst A ∧ (∃c. t = Fun c [] ∧ ¬public c ∧ arity c = 0)})
        ∈ ground_protocol_states P"
```

```
private lemma transaction_fresh_subst_priv_consts_of_iff:
  "transaction_fresh_subst σ T (priv_consts_of trms) ⟷ transaction_fresh_subst σ T trms"
proof (cases "transaction_fresh_subst σ T (priv_consts_of trms) ∨ transaction_fresh_subst σ T trms")
  case True
  then have "∀t ∈ subst_range σ. ∃c. t = Fun c [] ∧ ¬ public c ∧ arity c = 0"
    unfolding transaction_fresh_subst_def by auto
  have "(∀t ∈ subst_range σ. t ∈ subterms_set (priv_consts_of trms) ⟷ t ∈ subterms_set trms)"
  proof
    fix t
    assume "t ∈ subst_range σ"
    then obtain c where c_p: "t = Fun c [] ∧ ¬ public c ∧ arity c = 0"
      using ‹∀t∈subst_range σ. ∃c. t = Fun c [] ∧ ¬ public c ∧ arity c = 0› by blast
    then have "Fun c [] ∈ subterms_set (priv_consts_of trms) ⟷ Fun c [] ∈ subterms_set trms"
      unfolding priv_consts_of_def by auto
    then show "t ∈ subterms_set (priv_consts_of trms) ⟷ t ∈ subterms_set trms"
      using c_p by auto
  qed
  then show ?thesis
    using transaction_fresh_subst_def by force
next
  case False
  then show ?thesis by auto
qed

private lemma transaction_renaming_subst_inv:
  assumes "transaction_renaming_subst α P X "
  shows "∃αinv. α ∘s αinv = Var ∧  wf_trms (subst_range αinv)"
using var_rename_inv_comp transaction_renaming_subst_def assms subst_apply_term_empty subst_term_eqI
by (metis var_rename_wf_trms_range(2))

private lemma priv_consts_of_union_distr:
  "priv_consts_of (trms1 ∪ trms2) = priv_consts_of trms1 ∪ priv_consts_of trms2"
unfolding priv_consts_of_def by auto

private lemma ground_protocol_states_aux_finite_vars:
  assumes "(IK,DB,trms,vars,w) ∈ ground_protocol_states_aux P"
  shows "finite vars"
using assms by (induction rule: ground_protocol_states_aux.induct) auto

private lemma dbupd_sst_substitution_lemma:
  "dbupd_sst T (α ∘s I) DB = dbupd_sst (T ·sst α) I DB"
proof (induction T arbitrary: DB)
  case Nil
  then show ?case
    by auto
next
  case (Cons a T)
  then show ?case
    by (induction a) (simp_all add: subst_apply_stateful_strand_def)
qed

private lemma subst_Var_const_subterm_subst:
  assumes "subst_range α ⊆ range Var"
  shows "Fun c [] ⊑ t ⟷ Fun c [] ⊑ t · α"
  using assms
proof (induction t)
  case (Var x)
  then show ?case
    by (metis is_Var_def subtermeq_Var_const(1) term.discI(2) var_renaming_is_Fun_iff)
next
  case (Fun f ts)
```

```
  then show ?case
    by auto
qed

private lemma subst_Var_priv_consts_of:
  assumes "subst_range α ⊆ range Var"
  shows "priv_consts_of T = priv_consts_of (T ·set α)"
proof (rule antisym; rule subsetI)
  fix x
  assume "x ∈ priv_consts_of T"
  then obtain t c where t_c_p: "t ∈ T ∧ x ⊑ t ∧ x = Fun c [] ∧ ¬ public c ∧ arity c = 0"
    unfolding priv_consts_of_def by auto
  moreover
  have "x ⊑ t · α"
    using t_c_p by (meson assms subst_Var_const_subterm_subst)
  ultimately
  show "x ∈ priv_consts_of (T ·set α)"
    unfolding priv_consts_of_def by auto
next
  fix x
  assume "x ∈ priv_consts_of (T ·set α)"
  then obtain t c where t_c_p: "t ∈ T ∧ x ⊑ t · α ∧ x = Fun c [] ∧ ¬ public c ∧ arity c = 0"
    unfolding priv_consts_of_def by auto
  moreover
  have "x ⊑ t"
    using t_c_p by (meson assms subst_Var_const_subterm_subst)
  ultimately
  show "x ∈ priv_consts_of T"
    unfolding priv_consts_of_def by auto
qed

private lemma fst_set_subst_apply_set:
  "fst ` set F ·set α = fst ` (set F ·pset α)"
by (induction F) auto

private lemma snd_set_subst_apply_set:
  "snd ` set F ·set α = snd ` (set F ·pset α)"
by (induction F) auto

private lemma trms_pairs_fst_snd:
  "trms_pairs F = fst ` set F ∪ snd ` set F"
by (auto simp add: rev_image_eqI)

private lemma priv_consts_of_trms_sstp_range_Var:
  assumes "subst_range α ⊆ range Var"
  shows "priv_consts_of (trms_sstp a) = priv_consts_of (trms_sstp (a ·sstp α))"
  using assms
proof (induction a)
  case (NegChecks X F F')
  have α_subs_rng_Var: "subst_range (rm_vars (set X) α) ⊆ range Var"
    using assms by auto

  have "priv_consts_of (trms_pairs F) = priv_consts_of (fst ` set F ∪ snd ` set F)"
    using trms_pairs_fst_snd by metis
  also
  have "... = priv_consts_of (fst ` set F) ∪ priv_consts_of (snd ` set F)"
    using priv_consts_of_union_distr by blast
  also
  have "... = priv_consts_of ((fst ` set F) ·set rm_vars (set X) α) ∪
              priv_consts_of ((snd ` set F) ·set rm_vars (set X) α)"
    using α_subs_rng_Var subst_Var_priv_consts_of[of "rm_vars (set X) α" "fst ` set F"]
          subst_Var_priv_consts_of[of "rm_vars (set X) α" "snd ` set F"]
    by auto
```

```
  also
  have "... = priv_consts_of (((fst ` set F) ·set rm_vars (set X) α) ∪
               ((snd ` set F) ·set rm_vars (set X) α))"
    using priv_consts_of_union_distr by auto
  also
  have "... = priv_consts_of (fst ` set (F ·pairs rm_vars (set X) α) ∪
               snd ` set (F ·pairs rm_vars (set X) α))"
    unfolding subst_apply_pairs_def fst_set_subst_apply_set snd_set_subst_apply_set by simp
  also
  have "... = priv_consts_of (trms_pairs (F ·pairs rm_vars (set X) α))"
    using trms_pairs_fst_snd[of "F ·pairs rm_vars (set X) α"]
    by metis
  finally
  have 1: "priv_consts_of (trms_pairs F) = priv_consts_of (trms_pairs (F ·pairs rm_vars (set X) α))"
    by auto

  have "priv_consts_of (trms_pairs F') = priv_consts_of (fst ` set F' ∪ snd ` set F')"
    using trms_pairs_fst_snd by metis
  also
  have "... = priv_consts_of (fst ` set F') ∪ priv_consts_of (snd ` set F')"
    using priv_consts_of_union_distr by blast
  also
  have "... = priv_consts_of ((fst ` set F') ·set rm_vars (set X) α) ∪
              priv_consts_of ((snd ` set F') ·set rm_vars (set X) α)"
    using subst_Var_priv_consts_of[of "rm_vars (set X) α" "fst ` set F'"] α_subs_rng_Var
          subst_Var_priv_consts_of[of "rm_vars (set X) α" "snd ` set F'"]
    by auto
  also
  have "... = priv_consts_of ((fst ` set F' ·set rm_vars (set X) α) ∪
               (snd ` set F' ·set rm_vars (set X) α))"
    using priv_consts_of_union_distr by auto
  also
  have "... = priv_consts_of (fst ` set (F' ·pairs rm_vars (set X) α) ∪
               snd ` set (F' ·pairs rm_vars (set X) α))"
    unfolding subst_apply_pairs_def fst_set_subst_apply_set snd_set_subst_apply_set by simp
  also
  have "... = priv_consts_of (trms_pairs (F' ·pairs rm_vars (set X) α))"
    using trms_pairs_fst_snd[of "F' ·pairs rm_vars (set X) α"]
    by metis
  finally have 2: "priv_consts_of (trms_pairs F') = priv_consts_of (trms_pairs (F' ·pairs rm_vars (set X)
α))"
    by auto

  show ?case
    using 1 2 by (simp add: priv_consts_of_union_distr)
qed (use subst_Var_priv_consts_of[of _ "{_, _}", OF assms] subst_Var_priv_consts_of[OF assms] in auto)

private lemma priv_consts_of_trms_sst_range_Var:
  assumes "subst_range α ⊆ range Var"
  shows "priv_consts_of (trms_sst T) = priv_consts_of (trms_sst (T ·sst α))"
proof (induction T)
  case Nil
  then show ?case by auto
next
  case (Cons a T)
  have "priv_consts_of (trms_sst (a # T)) = priv_consts_of (trms_sst [a] ∪ trms_sst T)"
    by simp
  also
  have "... = priv_consts_of (trms_sst [a]) ∪ priv_consts_of (trms_sst T)"
    using priv_consts_of_union_distr by simp
  also
  have "... = priv_consts_of (trms_sstp a) ∪ priv_consts_of (trms_sst T)"
    by simp
```

```
  also
  have "... = priv_consts_of (trms_sstp (a ·_sstp α)) ∪ priv_consts_of (trms_sst T)"
    using priv_consts_of_trms_sstp_range_Var[OF assms] by simp
  also
  have "... = priv_consts_of (trms_sst ([a] ·_sst α)) ∪ priv_consts_of (trms_sst T)"
    by (simp add: subst_apply_stateful_strand_def)
  also
  have "... = priv_consts_of (trms_sst ([a] ·_sst α)) ∪ priv_consts_of (trms_sst (T ·_sst α))"
    using local.Cons by simp
  also
  have "... = priv_consts_of (trms_sst (a # T ·_sst α))"
    by (simp add: priv_consts_of_union_distr subst_sst_cons)
  finally
  show ?case
    by simp
qed

private lemma priv_consts_of_trms_lsst_range_Var:
  assumes "subst_range α ⊆ range Var"
  shows "priv_consts_of (trms_lsst T) = priv_consts_of (trms_lsst (T ·_lsst α))"
using priv_consts_of_trms_sst_range_Var[of α "unlabel T"]
by (metis assms unlabel_subst)

private lemma transaction_renaming_subst_range:
  assumes "transaction_renaming_subst α P vars"
  shows "subst_range α ⊆ range Var"
using assms unfolding transaction_renaming_subst_def var_rename_def by auto

private lemma protocol_models_equiv3':
  assumes "(IK,DB,trms,vars,w) ∈ ground_protocol_states_aux P"
  shows "(IK,DB, priv_consts_of trms) ∈ ground_protocol_states P"
  using assms
proof (induction rule: ground_protocol_states_aux.induct)
  case init
  then show ?case
    using ground_protocol_states.init unfolding priv_consts_of_def by force
next
  case (step IK DB trms vars w T ξ σ α A I)

  have fin_vars: "finite vars"
    using ground_protocol_states_aux_finite_vars step by auto

  have ground_ξσ: "ground (subst_range (ξ ∘_s σ))"
    using fresh_transaction_decl_subst_ground_subst_range using step.hyps(3) step.hyps(4) by blast

  have α_Var: "subst_range α ⊆ range Var"
    using step(5) transaction_renaming_subst_range by metis

  define I' where "I' = α ∘_s I"
  define A' where "A' = dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ)"

  have "(IK, DB, priv_consts_of trms) ∈ ground_protocol_states P"
    using step by force
  moreover
  have T_in_P: "T ∈ set P"
     using step by force
  moreover
  have "transaction_decl_subst ξ T"
     using step by force
  moreover
  have "transaction_fresh_subst σ T (priv_consts_of trms)"
     using step transaction_fresh_subst_priv_consts_of_iff by force
  moreover
```

```
have "A' = dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ)"
  using A'_def .
moreover
have "strand_sem_stateful IK DB (unlabel A') I'"
  using step(7) step(4) step(5) step(3) T_in_P fin_vars unfolding A'_def I'_def step(6)
  using strand_sem_stateful_substitution_lemma'
  by auto
moreover
have "interpretation_subst I'"
  using step(8) unfolding I'_def
  by (meson interpretation_comp(1))
moreover
have "wf_trms (subst_range I')"
 using step(9) unfolding I'_def
 using step.hyps(5) transaction_renaming_subst_range_wf_trms wf_trms_subst_compose by blast
ultimately
have "(IK ∪ (ik_lsst A' ·_set I'), dbupd_sst (unlabel A') I' DB,
         priv_consts_of trms ∪ priv_consts_of (trms_lsst A')) ∈ ground_protocol_states P"
  using ground_protocol_states.step[of IK DB "priv_consts_of trms" P T ξ σ A' I']
  unfolding priv_consts_of_def by blast
moreover
have "ik_lsst A' ·_set I' = ik_lsst A ·_set I"
proof -
  have "ik_lsst A' ·_set I' = ik_lsst A' ·_set α ∘_s I"
    unfolding A'_def I'_def step(6) by auto
  also
  have "... = ik_lsst ((A' ·_lsst α) ·_lsst I)"
    unfolding unlabel_subst[symmetric] ik_sst_subst by auto
  also
  have "... = ik_lsst (A ·_lsst I)"
    unfolding A'_def step(6)
    using labelled_stateful_strand_ground_subst_comp[of _ "transaction_strand T", OF ground_ξσ]
    by (simp add: dual_lsst_subst)
  also
  have "... = ik_lsst A ·_set I"
    by (metis ik_sst_subst unlabel_subst)
  finally
  show "ik_lsst A' ·_set I' = ik_lsst A ·_set I"
    by auto
qed
moreover
have "dbupd_sst (unlabel A') I' DB = dbupd_sst (unlabel A) I DB"
proof -
  have "dbupd_sst (unlabel A') I' DB =
          dbupd_sst (unlabel A') (α ∘_s I) DB"
    unfolding A'_def I'_def step(6) using step by auto
  also
  have "... = dbupd_sst (unlabel A' ·_sst α) I DB"
    using dbupd_sst_substitution_lemma by metis
  also
  have "... = dbupd_sst (unlabel A) I DB"
    unfolding A'_def step(6)
    using stateful_strand_ground_subst_comp[of _ "unlabel (dual_lsst (transaction_strand T))"]
      ground_ξσ by (simp add: dual_lsst_subst_unlabel)
  finally
  show "dbupd_sst (unlabel A') I' DB = dbupd_sst (unlabel A) I DB"
    by auto
qed
moreover
have "priv_consts_of (trms_lsst A') = priv_consts_of (trms_lsst A)"
  by (metis (no_types, lifting) A'_def α_Var priv_consts_of_trms_lsst_range_Var ground_ξσ
       labelled_stateful_strand_ground_subst_comp step.hyps(6) trms_sst_unlabel_dual_lsst_eq)
ultimately
```

```
  show ?case
    using priv_consts_of_union_distr by metis
qed

private lemma protocol_models_equiv4':
  assumes "(IK, DB, csts) ∈ ground_protocol_states P"
  shows "∃trms w vars. (IK,DB,trms,vars,w) ∈ ground_protocol_states_aux P
                  ∧ csts = priv_consts_of trms
                  ∧ vars = vars_lsst (T_symb w)"
  using assms
proof (induction rule: ground_protocol_states.induct)
  case init
  have "({}, {}, {}, {}, []) ∈ ground_protocol_states_aux P"
    using ground_protocol_states_aux.init by blast
  moreover
  have "{} = priv_consts_of {}"
    unfolding priv_consts_of_def by auto
  moreover
  have "{} = vars_lsst (T_symb [])"
    by auto
  ultimately
  show ?case
    by metis
next
  case (step IK DB "consts" T ξ σ A I)
  then obtain trms w vars where trms_w_vars_p:
    "(IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P"
    "consts = priv_consts_of trms"
    "vars = vars_lsst (T_symb w)"
    by auto

  have "∃α. transaction_renaming_subst α P vars"
    unfolding transaction_renaming_subst_def by blast

  then obtain α :: "('fun,'atom,'sets,'lbl) prot_subst"
    where α_p: "transaction_renaming_subst α P vars"
    by blast
  then obtain αinv where αinv_p: "α ∘s αinv = Var ∧ wf_trms (subst_range αinv)"
    using transaction_renaming_subst_inv[of α P vars] by auto

  define A' where "A' = dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
  define I' where "I' = αinv ∘s I"
  define trms' where "trms' = trms ∪ trms_lsst A'"
  define vars' where "vars' = vars ∪ vars_lsst A'"
  define w' where "w' = w @ [(ξ, σ, I', T, α)]"
  define IK' where "IK' = IK ∪ (ik_lsst A ·set I)"
  define DB' where "DB' = dbupd_sst (unlabel A) I DB"

  have P_state: "(IK', DB' , trms', vars', w') ∈ ground_protocol_states_aux P"
  proof -
    have 1: "(IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P"
      using ‹(IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P› by blast
    moreover
    have "T ∈ set P"
      using step(2) .
    moreover
    have "transaction_decl_subst ξ T"
      using step(3) .
    moreover
    have fresh_σ: "transaction_fresh_subst σ T trms"
      using step(4) trms_w_vars_p(2)
      using transaction_fresh_subst_priv_consts_of_iff by auto
    moreover
```

```
  have "transaction_renaming_subst α P vars"
    using α_p .
  moreover
  have "A' = dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
    unfolding A'_def by auto
  moreover
  have "strand_sem_stateful IK DB (unlabel A') I'"
  proof -
    have fin_vars: "finite vars"
      using 1 ground_protocol_states_aux_finite_vars by blast
    show "strand_sem_stateful IK DB (unlabel A') I'"
      using step(6) strand_sem_stateful_substitution_lemma'[OF α_p fresh_σ step(3) fin_vars]
        step(2) unfolding A'_def step(5)
      by (metis (no_types, lifting) I'_def αinv_p subst_compose_assoc var_comp(2))
  qed
  moreover
  have "interpretation_subst I'"
    using step(7) by (simp add: I'_def interpretation_substI subst_compose)
  moreover
  have "wf_trms (subst_range I')"
    using I'_def αinv_p step.hyps(8) wf_trms_subst_compose by blast
  moreover
  have "dbupd_sst (unlabel A) I DB = dbupd_sst (unlabel A') I' DB"
  proof -
    have "dbupd_sst (unlabel A) I DB = dbupd_sst (unlabel A) (α ∘_s αinv ∘_s I) DB"
      by (simp add: αinv_p)
    also
    have "... = dbupd_sst (unlabel A') (αinv ∘_s I) DB"
      unfolding A'_def step(5)
      by (metis (no_types, lifting) dbupd_sst_substitution_lemma dual_lsst_subst_unlabel
           subst_compose_assoc)
    also
    have "... = dbupd_sst (unlabel A') I' DB"
      unfolding A'_def I'_def by auto
    finally
    show ?thesis
      by auto
  qed
  moreover
  have "IK ∪ (ik_lsst A ·_set I) = IK ∪ (ik_lsst A' ·_set I')"
  proof -
    have "IK ∪ (ik_lsst A ·_set I) = IK ∪ (ik_lsst A ·_set (α  ∘_s αinv) ∘_s I)"
      using αinv_p by auto
    also
    have "... = IK ∪ (ik_lsst A' ·_set I')"
      unfolding A'_def step(5) unlabel_subst[symmetric] ik_sst_subst dual_lsst_subst I'_def by auto
    finally
    show ?thesis
      by auto
  qed
  ultimately
  show "(IK', DB' , trms', vars', w') ∈ ground_protocol_states_aux P"
    using ground_protocol_states_aux.step[of IK DB trms vars w P T ξ σ α A' I']
    unfolding trms'_def vars'_def w'_def IK'_def DB'_def by auto
qed
moreover
have "consts ∪ priv_consts_of (trms_lsst A) = priv_consts_of trms'"
proof -
  have α_Var: "subst_range α ⊆ range Var"
    using α_p transaction_renaming_subst_range by blast

  have ground_ξσ: "ground (subst_range (ξ ∘_s σ))"
    using fresh_transaction_decl_subst_ground_subst_range using step.hyps(3) step.hyps(4) by blast
```

```
    have "consts ∪ priv_consts_of (trms_lsst A) = (priv_consts_of trms) ∪ priv_consts_of (trms_lsst A)"
      using trms_w_vars_p(2) by blast
    also
    have "... = (priv_consts_of trms) ∪ priv_consts_of (trms_lsst (A ·_lsst α))"
      using priv_consts_of_trms_lsst_range_Var[of α, OF α_Var, of A] by auto
    also
    have "... = (priv_consts_of trms) ∪ priv_consts_of (trms_lsst A')"
      using step(5) A'_def ground_ξσ
      using labelled_stateful_strand_ground_subst_comp[of _ "(dual_lsst (transaction_strand T))"]
      by (simp add: dual_lsst_subst)
    also
    have "... = priv_consts_of (trms ∪ trms_lsst A')"
      using priv_consts_of_union_distr by blast
    also
    have "... = priv_consts_of trms'"
      unfolding trms'_def by auto
    finally
    show "?thesis"
      by auto
  qed
  moreover
  have "vars' = vars_lsst (T_symb w')"
    using P_state reachable_constraints_if_ground_protocol_states_aux by auto
  ultimately
  show ?case
    unfolding DB'_def IK'_def priv_consts_of_def[symmetric] by metis
qed

private lemma protocol_model_equivalence_aux2:
  "{(IK, DB) | IK DB. ∃csts. (IK, DB, csts) ∈ ground_protocol_states P} =
   {(IK, DB) | IK DB. ∃w trms vars. (IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P}"
using protocol_models_equiv4' protocol_models_equiv3' by meson

theorem protocol_model_equivalence:
  "{(IK, DB) | IK DB. ∃csts. (IK, DB, csts) ∈ ground_protocol_states P} =
   {(ik_lsst (A ·_lsst I), dbupd_sst (unlabel A) I {}) | A I.
      A ∈ reachable_constraints P ∧ strand_sem_stateful {} {} (unlabel A) I ∧
      interpretation_subst I ∧ wf_trms (subst_range I)}"
using protocol_model_equivalence_aux2 protocol_model_equivalence_aux1 by auto

end

end

end
```

3.4 Term Variants

```
theory Term_Variants
  imports Stateful_Protocol_Composition_and_Typing.Intruder_Deduction
begin

fun term_variants where
  "term_variants P (Var x) = [Var x]"
| "term_variants P (Fun f T) = (
  let S = product_lists (map (term_variants P) T)
  in map (Fun f) S@concat (map (λg. map (Fun g) S) (P f)))"

inductive term_variants_pred for P where
  term_variants_Var:
  "term_variants_pred P (Var x) (Var x)"
```

```
| term_variants_P:
  "⟦length T = length S; ⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i); g ∈ set (P f)⟧
   ⟹ term_variants_pred P (Fun f T) (Fun g S)"
| term_variants_Fun:
  "⟦length T = length S; ⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i)⟧
   ⟹ term_variants_pred P (Fun f T) (Fun f S)"

lemma term_variants_pred_inv:
  assumes "term_variants_pred P (Fun f T) (Fun h S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i)"
    and "f ≠ h ⟹ h ∈ set (P f)"
using assms by (auto elim: term_variants_pred.cases)

lemma term_variants_pred_inv':
  assumes "term_variants_pred P (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ term_variants_pred P (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ the_Fun t ∈ set (P f)"
    and "P ≡ (λ_. [])(g := [h]) ⟹ f ≠ the_Fun t ⟹ f = g ∧ the_Fun t = h"
using assms by (auto elim: term_variants_pred.cases)

lemma term_variants_pred_inv'':
  assumes "term_variants_pred P t (Fun f T)"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ term_variants_pred P (args t ! i) (T ! i)"
    and "f ≠ the_Fun t ⟹ f ∈ set (P (the_Fun t))"
    and "P ≡ (λ_. [])(g := [h]) ⟹ f ≠ the_Fun t ⟹ f = h ∧ the_Fun t = g"
using assms by (auto elim: term_variants_pred.cases)

lemma term_variants_pred_inv_Var:
  "term_variants_pred P (Var x) t ⟷ t = Var x"
  "term_variants_pred P t (Var x) ⟷ t = Var x"
by (auto intro: term_variants_Var elim: term_variants_pred.cases)

lemma term_variants_pred_inv_const:
  "term_variants_pred P (Fun c []) t ⟷ ((∃g ∈ set (P c). t = Fun g []) ∨ (t = Fun c []))"
by (auto intro: term_variants_P term_variants_Fun elim: term_variants_pred.cases)

lemma term_variants_pred_refl: "term_variants_pred P t t"
by (induct t) (auto intro: term_variants_pred.intros)

lemma term_variants_pred_refl_inv:
  assumes st: "term_variants_pred P s t"
    and P: "∀f. ∀g ∈ set (P f). f = g"
  shows "s = t"
  using st P
proof (induction s t rule: term_variants_pred.induct)
  case (term_variants_Var x) thus ?case by blast
next
  case (term_variants_P T S g f)
  hence "T ! i = S ! i" when i: "i < length T" for i using i by blast
  hence "T = S" using term_variants_P.hyps(1) by (simp add: nth_equalityI)
  thus ?case using term_variants_P.prems term_variants_P.hyps(3) by fast
next
  case (term_variants_Fun T S f)
  hence "T ! i = S ! i" when i: "i < length T" for i using i by blast
  hence "T = S" using term_variants_Fun.hyps(1) by (simp add: nth_equalityI)
  thus ?case by fast
qed
```

```
lemma term_variants_pred_const:
  assumes "b ∈ set (P a)"
  shows "term_variants_pred P (Fun a []) (Fun b [])"
using term_variants_P[of "[]" "[]"] assms by simp

lemma term_variants_pred_const_cases:
  "P a ≠ [] ⟹ term_variants_pred P (Fun a []) t ⟷
                (t = Fun a [] ∨ (∃b ∈ set (P a). t = Fun b []))"
  "P a = [] ⟹ term_variants_pred P (Fun a []) t ⟷ t = Fun a []"
using term_variants_pred_inv_const[of P] by auto

lemma term_variants_pred_param:
  assumes "term_variants_pred P t s"
    and fg: "f = g ∨ g ∈ set (P f)"
  shows "term_variants_pred P (Fun f (S@t#T)) (Fun g (S@s#T))"
proof -
  have 1: "length (S@t#T) = length (S@s#T)" by simp

  have "term_variants_pred P (T ! i) (T ! i)" "term_variants_pred P (S ! i) (S ! i)" for i
    by (metis term_variants_pred_refl)+
  hence 2: "term_variants_pred P ((S@t#T) ! i) ((S@s#T) ! i)" for i
    by (simp add: assms nth_Cons' nth_append)

  show ?thesis by (metis term_variants_Fun[OF 1 2] term_variants_P[OF 1 2] fg)
qed

lemma term_variants_pred_Cons:
  assumes t: "term_variants_pred P t s"
    and T: "term_variants_pred P (Fun f T) (Fun f S)"
    and fg: "f = g ∨ g ∈ set (P f)"
  shows "term_variants_pred P (Fun f (t#T)) (Fun g (s#S))"
proof -
  have 1: "length (t#T) = length (s#S)"
       and "⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i)"
    using term_variants_pred_inv[OF T] by simp_all
  hence 2: "⋀i. i < length (t#T) ⟹ term_variants_pred P ((t#T) ! i) ((s#S) ! i)"
    by (metis t One_nat_def diff_less length_Cons less_Suc_eq less_imp_diff_less nth_Cons'
              zero_less_Suc)

  show ?thesis using 1 2 fg by (auto intro: term_variants_pred.intros)
qed

lemma term_variants_pred_dense:
  fixes P Q::"'a set" and fs gs::"'a list"
  defines "P_fs x ≡ if x ∈ P then fs else []"
    and "P_gs x ≡ if x ∈ P then gs else []"
    and "Q_fs x ≡ if x ∈ Q then fs else []"
  assumes ut: "term_variants_pred P_fs u t"
    and g: "g ∈ Q" "g ∈ set gs"
  shows "∃s. term_variants_pred P_gs u s ∧ term_variants_pred Q_fs s t"
proof -
  define F where "F ≡ λ(P::'a set) (fs::'a list) x. if x ∈ P then fs else []"

  show ?thesis using ut g P_fs_def unfolding P_gs_def Q_fs_def
  proof (induction u t arbitrary: g gs rule: term_variants_pred.induct)
    case (term_variants_Var h x) thus ?case
      by (auto intro: term_variants_pred.term_variants_Var)
  next
    case (term_variants_P T S h' h g gs)
    note hyps = term_variants_P.hyps(1,2,4,5,6,7)
    note IH = term_variants_P.hyps(3)

    have "∃s. term_variants_pred (F P gs) (T ! i) s ∧ term_variants_pred (F Q fs) s (S ! i)"
```

```
      when i: "i < length T" for i
      using IH[OF i hyps(4,5,6)] unfolding F_def by presburger
    then obtain U where U:
        "length T = length U" "⋀i. i < length T ⟹ term_variants_pred (F P gs) (T ! i) (U ! i)"
        "length U = length S" "⋀i. i < length U ⟹ term_variants_pred (F Q fs) (U ! i) (S ! i)"
      using hyps(1) Skolem_list_nth[of _ "λi s. term_variants_pred (F P gs) (T ! i) s ∧
                                              term_variants_pred (F Q fs) s (S ! i)"]
      by (metis (no_types))

    show ?case
      using term_variants_pred.term_variants_P[OF U(1,2), of g h]
            term_variants_pred.term_variants_P[OF U(3,4), of h' g]
            hyps(3)[unfolded hyps(6)] hyps(4,5)
      unfolding F_def by force
  next
    case (term_variants_Fun T S h' g gs)
    note hyps = term_variants_Fun.hyps(1,2,4,5,6)
    note IH = term_variants_Fun.hyps(3)

    have "∃s. term_variants_pred (F P gs) (T ! i) s ∧ term_variants_pred (F Q fs) s (S ! i)"
      when i: "i < length T" for i
      using IH[OF i hyps(3,4,5)] unfolding F_def by presburger
    then obtain U where U:
        "length T = length U" "⋀i. i < length T ⟹ term_variants_pred (F P gs) (T ! i) (U ! i)"
        "length U = length S" "⋀i. i < length U ⟹ term_variants_pred (F Q fs) (U ! i) (S ! i)"
      using hyps(1) Skolem_list_nth[of _ "λi s. term_variants_pred (F P gs) (T ! i) s ∧
                                              term_variants_pred (F Q fs) s (S ! i)"]
      by (metis (no_types))

    thus ?case
      using term_variants_pred.term_variants_Fun[OF U(1,2)]
            term_variants_pred.term_variants_Fun[OF U(3,4)]
      unfolding F_def by meson
  qed
qed

lemma term_variants_pred_dense':
  assumes ut: "term_variants_pred ((λ_. [])(a := [b])) u t"
  shows "∃s. term_variants_pred ((λ_. [])(a := [c])) u s ∧
             term_variants_pred ((λ_. [])(c := [b])) s t"
using ut term_variants_pred_dense[of "{a}" "[b]" u t c "{c}" "[c]"]
unfolding fun_upd_def by simp

lemma term_variants_pred_eq_case:
  fixes t s::"('a,'b) term"
  assumes "term_variants_pred P t s" "∀f ∈ funs_term t. P f = []"
  shows "t = s"
using assms
proof (induction t s rule: term_variants_pred.induct)
  case (term_variants_Fun T S f) thus ?case
    using subtermeq_imp_funs_term_subset[OF Fun_param_in_subterms[OF nth_mem], of _ T f]
          nth_equalityI[of T S]
    by blast
qed (simp_all add: term_variants_pred_refl)

lemma term_variants_pred_subst:
  assumes "term_variants_pred P t s"
  shows "term_variants_pred P (t · δ) (s · δ)"
using assms
proof (induction t s rule: term_variants_pred.induct)
  case (term_variants_P T S f g)
  have 1: "length (map (λt. t · δ) T) = length (map (λt. t · δ) S)"
    using term_variants_P.hyps
```

```
    by simp

  have 2: "term_variants_pred P ((map (λt. t · δ) T) ! i) ((map (λt. t · δ) S) ! i)"
    when "i < length (map (λt. t · δ) T)" for i
    using term_variants_P that
    by fastforce

  show ?case
    using term_variants_pred.term_variants_P[OF 1 2 term_variants_P.hyps(3)]
    by fastforce
next
  case (term_variants_Fun T S f)
  have 1: "length (map (λt. t · δ) T) = length (map (λt. t · δ) S)"
    using term_variants_Fun.hyps
    by simp

  have 2: "term_variants_pred P ((map (λt. t · δ) T) ! i) ((map (λt. t · δ) S) ! i)"
    when "i < length (map (λt. t · δ) T)" for i
    using term_variants_Fun that
    by fastforce

  show ?case
    using term_variants_pred.term_variants_Fun[OF 1 2]
    by fastforce
qed (simp add: term_variants_pred_refl)

lemma term_variants_pred_subst':
  fixes t s::"('a,'b) term" and δ::"('a,'b) subst"
  assumes "term_variants_pred P (t · δ) s"
    and "∀x ∈ fv t ∪ fv s. (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ P f = [])"
  shows "∃u. term_variants_pred P t u ∧ s = u · δ"
using assms
proof (induction "t · δ" s arbitrary: t rule: term_variants_pred.induct)
  case (term_variants_Var x g) thus ?case using term_variants_pred_refl by fast
next
  case (term_variants_P T S g f) show ?case
  proof (cases t)
    case (Var x) thus ?thesis
      using term_variants_P.hyps(4,5) term_variants_P.prems
      by fastforce
  next
    case (Fun h U)
    hence 1: "h = f" "T = map (λs. s · δ) U" "length U = length T"
      using term_variants_P.hyps(5) by simp_all
    hence 2: "T ! i = U ! i · δ" when "i < length T" for i
      using that by simp

    have "∀x ∈ fv (U ! i) ∪ fv (S ! i). (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ P f = [])"
      when "i < length U" for i
      using that Fun term_variants_P.prems term_variants_P.hyps(1) 1(3)
      by force
    hence IH: "∀i < length U. ∃u. term_variants_pred P (U ! i) u ∧ S ! i = u · δ"
      by (metis 1(3) term_variants_P.hyps(3)[OF _ 2])

    have "∃V. length U = length V ∧ S = map (λv. v · δ) V ∧
              (∀i < length U. term_variants_pred P (U ! i) (V ! i))"
      using term_variants_P.hyps(1) 1(3) subst_term_list_obtain[OF IH] by metis
    then obtain V where V: "length U = length V" "S = map (λv. v · δ) V"
                          "⋀i. i < length U ⟹ term_variants_pred P (U ! i) (V ! i)"
      by blast

    have "term_variants_pred P (Fun f U) (Fun g V)"
      by (metis term_variants_pred.term_variants_P[OF V(1,3) term_variants_P.hyps(4)])
```

```
    moreover have "Fun g S = Fun g V · δ" using V(2) by simp
    ultimately show ?thesis using term_variants_P.hyps(1,4) Fun 1 by blast
  qed
next
  case (term_variants_Fun T S f t) show ?case
  proof (cases t)
    case (Var x)
    hence "T = []" "P f = []" using term_variants_Fun.hyps(4) term_variants_Fun.prems by fastforce+
    thus ?thesis using term_variants_pred_refl Var term_variants_Fun.hyps(1,4) by fastforce
  next
    case (Fun h U)
    hence 1: "h = f" "T = map (λs. s · δ) U" "length U = length T"
      using term_variants_Fun.hyps(4) by simp_all
    hence 2: "T ! i = U ! i · δ" when "i < length T" for i
      using that by simp

    have "∀x ∈ fv (U ! i) ∪ fv (S ! i). (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ P f = [])"
      when "i < length U" for i
      using that Fun term_variants_Fun.prems term_variants_Fun.hyps(1) 1(3)
      by force
    hence IH: "∀i < length U. ∃u. term_variants_pred P (U ! i) u ∧ S ! i = u · δ"
      by (metis 1(3) term_variants_Fun.hyps(3)[OF _ 2 ])

    have "∃V. length U = length V ∧ S = map (λv. v · δ) V ∧
            (∀i < length U. term_variants_pred P (U ! i) (V ! i))"
      using term_variants_Fun.hyps(1) 1(3) subst_term_list_obtain[OF IH] by metis
    then obtain V where V: "length U = length V" "S = map (λv. v · δ) V"
                          "⋀i. i < length U ⟹ term_variants_pred P (U ! i) (V ! i)"
      by blast

    have "term_variants_pred P (Fun f U) (Fun f V)"
      by (metis term_variants_pred.term_variants_Fun[OF V(1,3)])
    moreover have "Fun f S = Fun f V · δ" using V(2) by simp
    ultimately show ?thesis using term_variants_Fun.hyps(1) Fun 1 by blast
  qed
qed

lemma term_variants_pred_subst'':
  assumes "∀x ∈ fv t. term_variants_pred P (δ x) (ϑ x)"
  shows "term_variants_pred P (t · δ) (t · ϑ)"
using assms
proof (induction t)
  case (Fun f ts) thus ?case
    using term_variants_Fun[of "map (λt. t · δ) ts" "map (λt. t · ϑ) ts" P f] by force
qed simp

lemma term_variants_pred_iff_in_term_variants:
  fixes t::"('a,'b) term"
  shows "term_variants_pred P t s ⟷ s ∈ set (term_variants P t)"
    (is "?A t s ⟷ ?B t s")
proof
  define U where "U ≡ λP (T::('a,'b) term list). product_lists (map (term_variants P) T)"

  have a:
      "g ∈ set (P f) ⟹ set (map (Fun g) (U P T)) ⊆ set (term_variants P (Fun f T))"
      "set (map (Fun f) (U P T)) ⊆ set (term_variants P (Fun f T))"
    for f P g and T::"('a,'b) term list"
    using term_variants.simps(2)[of P f T]
    unfolding U_def Let_def by auto

  have b: "∃S ∈ set (U P T). s = Fun f S ∨ (∃g ∈ set (P f). s = Fun g S)"
    when "s ∈ set (term_variants P (Fun f T))" for P T f s
    using that by (cases "P f") (auto simp add: U_def Let_def)
```

```
  have c: "length T = length S" when "S ∈ set (U P T)" for S P T
    using that unfolding U_def
    by (simp add: in_set_product_lists_length)

  show "?A t s ⟹ ?B t s"
  proof (induction t s rule: term_variants_pred.induct)
    case (term_variants_P T S g f)
    note hyps = term_variants_P.hyps
    note IH = term_variants_P.IH

    have "S ∈ set (U P T)"
      using IH hyps(1) product_lists_in_set_nth'[of _ S]
      unfolding U_def by simp
    thus ?case using a(1)[of _ P, OF hyps(3)] by auto
  next
    case (term_variants_Fun T S f)
    note hyps = term_variants_Fun.hyps
    note IH = term_variants_Fun.IH

    have "S ∈ set (U P T)"
      using IH hyps(1) product_lists_in_set_nth'[of _ S]
      unfolding U_def by simp
    thus ?case using a(2)[of f P T] by (cases "P f") auto
  qed (simp add: term_variants_Var)

  show "?B t s ⟹ ?A t s"
  proof (induction P t arbitrary: s rule: term_variants.induct)
    case (2 P f T)
    obtain S where S:
        "s = Fun f S ∨ (∃g ∈ set (P f). s = Fun g S)"
        "S ∈ set (U P T)" "length T = length S"
      using c b[OF "2.prems"] by blast

    have "∀i < length T. term_variants_pred P (T ! i) (S ! i)"
      using "2.IH" S product_lists_in_set_nth by (fastforce simp add: U_def)
    thus ?case using S by (auto intro: term_variants_pred.intros)
  qed (simp add: term_variants_Var)
qed

lemma term_variants_pred_finite:
  "finite {s. term_variants_pred P t s}"
using term_variants_pred_iff_in_term_variants[of P t]
by simp

lemma term_variants_pred_fv_eq:
  assumes "term_variants_pred P s t"
  shows "fv s = fv t"
using assms
by (induct rule: term_variants_pred.induct)
   (metis, metis fv_eq_FunI, metis fv_eq_FunI)

lemma (in intruder_model) term_variants_pred_wf_trms:
  assumes "term_variants_pred P s t"
    and "⋀f g. g ∈ set (P f) ⟹ arity f = arity g"
    and "wf_trm s"
  shows "wf_trm t"
using assms
apply (induction rule: term_variants_pred.induct, simp)
by (metis (no_types) wf_trmI wf_trm_arity in_set_conv_nth wf_trm_param_idx)+

lemma term_variants_pred_funs_term:
  assumes "term_variants_pred P s t"
```

```
    and "f ∈ funs_term t"
  shows "f ∈ funs_term s ∨ (∃g ∈ funs_term s. f ∈ set (P g))"
  using assms
proof (induction rule: term_variants_pred.induct)
  case (term_variants_P T S g h) thus ?case
  proof (cases "f = g")
    case False
    then obtain s where "s ∈ set S" "f ∈ funs_term s"
      using funs_term_subterms_eq(1)[of "Fun g S"] term_variants_P.prems by auto
    thus ?thesis
      using term_variants_P.IH term_variants_P.hyps(1) in_set_conv_nth[of s S] by force
  qed simp
next
  case (term_variants_Fun T S h) thus ?case
  proof (cases "f = h")
    case False
    then obtain s where "s ∈ set S" "f ∈ funs_term s"
      using funs_term_subterms_eq(1)[of "Fun h S"] term_variants_Fun.prems by auto
    thus ?thesis
      using term_variants_Fun.IH term_variants_Fun.hyps(1) in_set_conv_nth[of s S] by force
  qed simp
qed fast

end
```

3.5 Term Implication

```
theory Term_Implication
  imports Stateful_Protocol_Model Term_Variants
begin
```

3.5.1 Single Term Implications

```
definition timpl_apply_term (‹⟨_ --» _⟩⟨_⟩›) where
  "⟨a --» b⟩⟨t⟩ ≡ term_variants ((λ_. [])(Abs a := [Abs b])) t"

definition timpl_apply_terms (‹⟨_ --» _⟩⟨_⟩_set›) where
  "⟨a --» b⟩⟨M⟩_set ≡ ⋃((set o timpl_apply_term a b) ` M)"

lemma timpl_apply_Fun:
  assumes "⋀i. i < length T ⟹ S ! i ∈ set ⟨a --» b⟩⟨T ! i⟩"
    and "length T = length S"
  shows "Fun f S ∈ set ⟨a --» b⟩⟨Fun f T⟩"
using assms term_variants_Fun term_variants_pred_iff_in_term_variants
by (metis timpl_apply_term_def)

lemma timpl_apply_Abs:
  assumes "⋀i. i < length T ⟹ S ! i ∈ set ⟨a --» b⟩⟨T ! i⟩"
    and "length T = length S"
  shows "Fun (Abs b) S ∈ set ⟨a --» b⟩⟨Fun (Abs a) T⟩"
using assms(1) term_variants_P[OF assms(2), of "(λ_. [])(Abs a := [Abs b])" "Abs b" "Abs a"]
unfolding timpl_apply_term_def term_variants_pred_iff_in_term_variants[symmetric]
by fastforce

lemma timpl_apply_refl: "t ∈ set ⟨a --» b⟩⟨t⟩"
unfolding timpl_apply_term_def
by (metis term_variants_pred_refl term_variants_pred_iff_in_term_variants)

lemma timpl_apply_const: "Fun (Abs b) [] ∈ set ⟨a --» b⟩⟨Fun (Abs a) []⟩"
using term_variants_pred_iff_in_term_variants term_variants_pred_const
unfolding timpl_apply_term_def by auto
```

```
lemma timpl_apply_const':
  "c = a ⟹ set ⟨a --» b⟩⟨Fun (Abs c) []⟩ = {Fun (Abs b) [], Fun (Abs c) []}"
  "c ≠ a ⟹ set ⟨a --» b⟩⟨Fun (Abs c) []⟩ = {Fun (Abs c) []}"
using term_variants_pred_const_cases[of "(λ_. [])(Abs a := [Abs b])" "Abs c"]
      term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
unfolding timpl_apply_term_def by auto

lemma timpl_apply_term_subst:
  "s ∈ set ⟨a --» b⟩⟨t⟩ ⟹ s · δ ∈ set ⟨a --» b⟩⟨t · δ⟩"
by (metis term_variants_pred_iff_in_term_variants term_variants_pred_subst timpl_apply_term_def)

lemma timpl_apply_inv:
  assumes "Fun h S ∈ set ⟨a --» b⟩⟨Fun f T⟩"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ S ! i ∈ set ⟨a --» b⟩⟨T ! i⟩"
    and "f ≠ h ⟹ f = Abs a ∧ h = Abs b"
using assms term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
unfolding timpl_apply_term_def
by (metis (full_types) term_variants_pred_inv(1),
    metis (full_types) term_variants_pred_inv(2),
    fastforce dest: term_variants_pred_inv(3))

lemma timpl_apply_inv':
  assumes "s ∈ set ⟨a --» b⟩⟨Fun f T⟩"
  shows "∃g S. s = Fun g S"
proof -
  have *: "term_variants_pred ((λ_. [])(Abs a := [Abs b])) (Fun f T) s"
    using assms term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
    unfolding timpl_apply_term_def by force
  show ?thesis using term_variants_pred.cases[OF *, of ?thesis] by fastforce
qed

lemma timpl_apply_term_Var_iff:
  "Var x ∈ set ⟨a --» b⟩⟨t⟩ ⟷ t = Var x"
using term_variants_pred_inv_Var term_variants_pred_iff_in_term_variants
unfolding timpl_apply_term_def by metis
```

3.5.2 Term Implication Closure

```
inductive_set timpl_closure for t TI where
  FP: "t ∈ timpl_closure t TI"
| TI: "⟦u ∈ timpl_closure t TI; (a,b) ∈ TI; term_variants_pred ((λ_. [])(Abs a := [Abs b])) u s⟧
       ⟹ s ∈ timpl_closure t TI"

definition "timpl_closure_set M TI ≡ (⋃t ∈ M. timpl_closure t TI)"

inductive_set timpl_closure'_step for TI where
  "⟦(a,b) ∈ TI; term_variants_pred ((λ_. [])(Abs a := [Abs b])) t s⟧
    ⟹ (t,s) ∈ timpl_closure'_step TI"

definition "timpl_closure' TI ≡ (timpl_closure'_step TI)*"

definition comp_timpl_closure where
  "comp_timpl_closure FP TI ≡
    let f = λX. FP ∪ (⋃x ∈ X. ⋃(a,b) ∈ TI. set ⟨a --» b⟩⟨x⟩)
    in while (λX. f X ≠ X) f {}"

definition comp_timpl_closure_list where
  "comp_timpl_closure_list FP TI ≡
    let f = λX. remdups (concat (map (λx. concat (map (λ(a,b). ⟨a --» b⟩⟨x⟩) TI)) X)@X)
    in while (λX. set (f X) ≠ set X) f FP"

lemma timpl_closure_setI:
```

```
  "t ∈ M ⟹ t ∈ timpl_closure_set M TI"
unfolding timpl_closure_set_def by (auto intro: timpl_closure.FP)

lemma timpl_closure_set_empty_timpls:
  "timpl_closure t {} = {t}" (is "?A = ?B")
proof (intro subset_antisym subsetI)
  fix s show "s ∈ ?A ⟹ s ∈ ?B"
    by (induct s rule: timpl_closure.induct) auto
qed (simp add: timpl_closure.FP)

lemmas timpl_closure_set_is_timpl_closure_union = meta_eq_to_obj_eq[OF timpl_closure_set_def]

lemma term_variants_pred_eq_case_Abs:
  fixes a b
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes "term_variants_pred P t s" "∀f ∈ funs_term s. ¬is_Abs f"
  shows "t = s"
using assms(2,3)
proof (induction t s rule: term_variants_pred.induct)
  case (term_variants_Fun T S f)
  have "¬is_Abs h" when i: "i < length S" and h: "h ∈ funs_term (S ! i)" for i h
    using i h term_variants_Fun.prems by auto
  hence "T ! i = S ! i" when i: "i < length T" for i
    using i term_variants_Fun.hyps(1) term_variants_Fun.IH by auto
  hence "T = S" using term_variants_Fun.hyps(1) nth_equalityI[of T S] by fast
  thus ?case using term_variants_Fun.hyps(1) by blast
qed (simp_all add: term_variants_pred_refl P_def)

lemma timpl_closure'_step_inv:
  assumes "(t,s) ∈ timpl_closure'_step TI"
  obtains a b where "(a,b) ∈ TI" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) t s"
using assms by (auto elim: timpl_closure'_step.cases)

lemma timpl_closure_mono:
  assumes "TI ⊆ TI'"
  shows "timpl_closure t TI ⊆ timpl_closure t TI'"
proof
  fix s show "s ∈ timpl_closure t TI ⟹ s ∈ timpl_closure t TI'"
    apply (induct rule: timpl_closure.induct)
    using assms by (auto intro: timpl_closure.intros)
qed

lemma timpl_closure_set_mono:
  assumes "M ⊆ M'" "TI ⊆ TI'"
  shows "timpl_closure_set M TI ⊆ timpl_closure_set M' TI'"
using assms(1) timpl_closure_mono[OF assms(2)] unfolding timpl_closure_set_def by fast

lemma timpl_closure_idem:
  "timpl_closure_set (timpl_closure t TI) TI = timpl_closure t TI" (is "?A = ?B")
proof
  have "s ∈ timpl_closure t TI"
    when "s ∈ timpl_closure u TI" "u ∈ timpl_closure t TI"
    for s u
    using that
    by (induction rule: timpl_closure.induct)
       (auto intro: timpl_closure.intros)
  thus "?A ⊆ ?B" unfolding timpl_closure_set_def by blast

  show "?B ⊆ ?A"
    unfolding timpl_closure_set_def
    by (blast intro: timpl_closure.FP)
qed
```

```
lemma timpl_closure_set_idem:
  "timpl_closure_set (timpl_closure_set M TI) TI = timpl_closure_set M TI"
using timpl_closure_idem[of _ TI]unfolding timpl_closure_set_def by auto

lemma timpl_closure_set_mono_timpl_closure_set:
  assumes N: "N ⊆ timpl_closure_set M TI"
  shows "timpl_closure_set N TI ⊆ timpl_closure_set M TI"
using timpl_closure_set_mono[OF N, of TI TI] timpl_closure_set_idem[of M TI]
by simp

lemma timpl_closure_is_timpl_closure':
  "s ∈ timpl_closure t TI ⟷ (t,s) ∈ timpl_closure' TI"
proof
  show "s ∈ timpl_closure t TI ⟹ (t,s) ∈ timpl_closure' TI"
    unfolding timpl_closure'_def
    by (induct rule: timpl_closure.induct)
       (auto intro: rtrancl_into_rtrancl timpl_closure'_step.intros)

  show "(t,s) ∈ timpl_closure' TI ⟹ s ∈ timpl_closure t TI"
    unfolding timpl_closure'_def
    by (induct rule: rtrancl_induct)
       (auto dest: timpl_closure'_step_inv
             intro: timpl_closure.FP timpl_closure.TI)
qed

lemma timpl_closure'_mono:
  assumes "TI ⊆ TI'"
  shows "timpl_closure' TI ⊆ timpl_closure' TI'"
using timpl_closure_mono[OF assms]
      timpl_closure_is_timpl_closure'[of _ _ TI]
      timpl_closure_is_timpl_closure'[of _ _ TI']
by fast

lemma timpl_closureton_is_timpl_closure:
  "timpl_closure_set {t} TI = timpl_closure t TI"
by (simp add: timpl_closure_set_is_timpl_closure_union)

lemma timpl_closure'_timpls_trancl_subset:
  "timpl_closure' (c⁺) ⊆ timpl_closure' c"
unfolding timpl_closure'_def
proof
  fix s t::"(('a,'b,'c,'d) prot_fun,'e) term"
  show "(s,t) ∈ (timpl_closure'_step (c⁺))* ⟹ (s,t) ∈ (timpl_closure'_step c)*"
  proof (induction rule: rtrancl_induct)
    case (step u t)
    obtain a b where ab:
        "(a,b) ∈ c⁺" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u t"
      using step.hyps(2) timpl_closure'_step_inv by blast
    hence "(u,t) ∈ (timpl_closure'_step c)*"
    proof (induction arbitrary: t rule: trancl_induct)
      case (step d e)
      obtain s where s:
          "term_variants_pred ((λ_. [])(Abs a := [Abs d])) u s"
          "term_variants_pred ((λ_. [])(Abs d := [Abs e])) s t"
        using term_variants_pred_dense'[OF step.prems, of "Abs d"] by blast

      have "(u,s) ∈ (timpl_closure'_step c)*"
           "(s,t) ∈ timpl_closure'_step c"
        using step.hyps(2) s(2) step.IH[OF s(1)]
        by (auto intro: timpl_closure'_step.intros)
      thus ?case by simp
    qed (auto intro: timpl_closure'_step.intros)
    thus ?case using step.IH by simp
```

```
  qed simp
qed

lemma timpl_closure'_timpls_trancl_subset':
  "timpl_closure' {(a,b) ∈ c⁺. a ≠ b} ⊆ timpl_closure' c"
using timpl_closure'_timpls_trancl_subset
      timpl_closure'_mono[of "{(a,b) ∈ c⁺. a ≠ b}" "c⁺"]
by fast

lemma timpl_closure_set_timpls_trancl_subset:
  "timpl_closure_set M (c⁺) ⊆ timpl_closure_set M c"
using timpl_closure'_timpls_trancl_subset[of c]
      timpl_closure_is_timpl_closure'[of _ _ c]
      timpl_closure_is_timpl_closure'[of _ _ "c⁺"]
      timpl_closure_set_is_timpl_closure_union[of M c]
      timpl_closure_set_is_timpl_closure_union[of M "c⁺"]
by fastforce

lemma timpl_closure_set_timpls_trancl_subset':
  "timpl_closure_set M {(a,b) ∈ c⁺. a ≠ b} ⊆ timpl_closure_set M c"
using timpl_closure'_timpls_trancl_subset'[of c]
      timpl_closure_is_timpl_closure'[of _ _ c]
      timpl_closure_is_timpl_closure'[of _ _ "{(a,b) ∈ c⁺. a ≠ b}"]
      timpl_closure_set_is_timpl_closure_union[of M c]
      timpl_closure_set_is_timpl_closure_union[of M "{(a,b) ∈ c⁺. a ≠ b}"]
by fastforce

lemma timpl_closure'_timpls_trancl_supset':
  "timpl_closure' c ⊆ timpl_closure' {(a,b) ∈ c⁺. a ≠ b}"
unfolding timpl_closure'_def
proof
  let ?cl = "{(a,b) ∈ c⁺. a ≠ b}"

  fix s t::"(('a,'b,'c,'d) prot_fun,'e) term"
  show "(s,t) ∈ (timpl_closure'_step c)* ⟹ (s,t) ∈ (timpl_closure'_step ?cl)*"
  proof (induction rule: rtrancl_induct)
    case (step u t)
    obtain a b where ab:
        "(a,b) ∈ c" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u t"
      using step.hyps(2) timpl_closure'_step_inv by blast
    hence "(a,b) ∈ c⁺" by simp
    hence "(u,t) ∈ (timpl_closure'_step ?cl)*" using ab(2)
    proof (induction arbitrary: t rule: trancl_induct)
      case (base d) show ?case
      proof (cases "a = d")
        case True thus ?thesis
          using base term_variants_pred_refl_inv[of _ u t]
          by force
      next
        case False thus ?thesis
          using base timpl_closure'_step.intros[of a d ?cl]
          by fast
      qed
    next
      case (step d e)
      obtain s where s:
          "term_variants_pred ((λ_. [])(Abs a := [Abs d])) u s"
          "term_variants_pred ((λ_. [])(Abs d := [Abs e])) s t"
        using term_variants_pred_dense'[OF step.prems, of "Abs d"] by blast

      show ?case
      proof (cases "d = e")
        case True
```

```
        thus ?thesis
          using step.prems step.IH[of t]
          by blast
      next
        case False
        hence "(u,s) ∈ (timpl_closure'_step ?cl)*"
              "(s,t) ∈ timpl_closure'_step ?cl"
          using step.hyps(2) s(2) step.IH[OF s(1)]
          by (auto intro: timpl_closure'_step.intros)
        thus ?thesis by simp
      qed
    qed
    thus ?case using step.IH by simp
  qed simp
qed

lemma timpl_closure'_timpls_trancl_supset:
  "timpl_closure' c ⊆ timpl_closure' (c+)"
using timpl_closure'_timpls_trancl_supset'[of c]
      timpl_closure'_mono[of "{(a,b) ∈ c+. a ≠ b}" "c+"]
by fast

lemma timpl_closure'_timpls_trancl_eq:
  "timpl_closure' (c+) = timpl_closure' c"
using timpl_closure'_timpls_trancl_subset timpl_closure'_timpls_trancl_supset
by blast

lemma timpl_closure'_timpls_trancl_eq':
  "timpl_closure' {(a,b) ∈ c+. a ≠ b} = timpl_closure' c"
using timpl_closure'_timpls_trancl_subset' timpl_closure'_timpls_trancl_supset'
by blast

lemma timpl_closure'_timpls_rtrancl_subset:
  "timpl_closure' (c*) ⊆ timpl_closure' c"
unfolding timpl_closure'_def
proof
  fix s t::"(('a,'b,'c,'d) prot_fun,'e) term"
  show "(s,t) ∈ (timpl_closure'_step (c*))* ⟹ (s,t) ∈ (timpl_closure'_step c)*"
  proof (induction rule: rtrancl_induct)
    case (step u t)
    obtain a b where ab:
        "(a,b) ∈ c*" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u t"
      using step.hyps(2) timpl_closure'_step_inv by blast
    hence "(u,t) ∈ (timpl_closure'_step c)*"
    proof (induction arbitrary: t rule: rtrancl_induct)
      case base
      hence "u = t" using term_variants_pred_refl_inv by fastforce
      thus ?case by simp
    next
      case (step d e)
      obtain s where s:
          "term_variants_pred ((λ_. [])(Abs a := [Abs d])) u s"
          "term_variants_pred ((λ_. [])(Abs d := [Abs e])) s t"
        using term_variants_pred_dense'[OF step.prems, of "Abs d"] by blast

      have "(u,s) ∈ (timpl_closure'_step c)*"
           "(s,t) ∈ timpl_closure'_step c"
        using step.hyps(2) s(2) step.IH[OF s(1)]
        by (auto intro: timpl_closure'_step.intros)
      thus ?case by simp
    qed
    thus ?case using step.IH by simp
  qed simp
```

```
qed

lemma timpl_closure'_timpls_rtrancl_supset:
  "timpl_closure' c ⊆ timpl_closure' (c*)"
unfolding timpl_closure'_def
proof
  fix s t::"(('a,'b,'c,'d) prot_fun,'e) term"
  show "(s,t) ∈ (timpl_closure'_step c)* ⟹ (s,t) ∈ (timpl_closure'_step (c*))*"
  proof (induction rule: rtrancl_induct)
    case (step u t)
    obtain a b where ab:
        "(a,b) ∈ c" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u t"
      using step.hyps(2) timpl_closure'_step_inv by blast
    hence "(a,b) ∈ c*" by simp
    hence "(u,t) ∈ (timpl_closure'_step (c*))*" using ab(2)
    proof (induction arbitrary: t rule: rtrancl_induct)
      case (base t) thus ?case using term_variants_pred_refl_inv[of _ u t] by fastforce
    next
      case (step d e)
      obtain s where s:
          "term_variants_pred ((λ_. [])(Abs a := [Abs d])) u s"
          "term_variants_pred ((λ_. [])(Abs d := [Abs e])) s t"
        using term_variants_pred_dense'[OF step.prems, of "Abs d"] by blast

      show ?case
      proof (cases "d = e")
        case True
        thus ?thesis
          using step.prems step.IH[of t]
          by blast
      next
        case False
        hence "(u,s) ∈ (timpl_closure'_step (c*))*"
              "(s,t) ∈ timpl_closure'_step (c*)"
          using step.hyps(2) s(2) step.IH[OF s(1)]
          by (auto intro: timpl_closure'_step.intros)
        thus ?thesis by simp
      qed
    qed
    thus ?case using step.IH by simp
  qed simp
qed

lemma timpl_closure'_timpls_rtrancl_eq:
  "timpl_closure' (c*) = timpl_closure' c"
using timpl_closure'_timpls_rtrancl_subset timpl_closure'_timpls_rtrancl_supset
by blast

lemma timpl_closure_timpls_trancl_eq:
  "timpl_closure t (c+) = timpl_closure t c"
using timpl_closure'_timpls_trancl_eq[of c]
      timpl_closure_is_timpl_closure'[of _ _ c]
      timpl_closure_is_timpl_closure'[of _ _ "c+"]
by fastforce

lemma timpl_closure_set_timpls_trancl_eq:
  "timpl_closure_set M (c+) = timpl_closure_set M c"
using timpl_closure_timpls_trancl_eq
      timpl_closure_set_is_timpl_closure_union[of M c]
      timpl_closure_set_is_timpl_closure_union[of M "c+"]
by fastforce

lemma timpl_closure_set_timpls_trancl_eq':
```

```
  "timpl_closure_set M {(a,b) ∈ c⁺. a ≠ b} = timpl_closure_set M c"
using timpl_closure'_timpls_trancl_eq'[of c]
      timpl_closure_is_timpl_closure'[of _ _ c]
      timpl_closure_is_timpl_closure'[of _ _ "{(a,b) ∈ c⁺. a ≠ b}"]
      timpl_closure_set_is_timpl_closure_union[of M c]
      timpl_closure_set_is_timpl_closure_union[of M "{(a,b) ∈ c⁺. a ≠ b}"]
by fastforce

lemma timpl_closure_Var_in_iff:
  "Var x ∈ timpl_closure t TI ⟷ t = Var x" (is "?A ⟷ ?B")
proof
  have "s ∈ timpl_closure t TI ⟹ s = Var x ⟹ s = t" for s
    apply (induction rule: timpl_closure.induct)
    by (simp, metis term_variants_pred_inv_Var(2))
  thus "?A ⟹ ?B" by blast
qed (blast intro: timpl_closure.FP)

lemma timpl_closure_set_Var_in_iff:
  "Var x ∈ timpl_closure_set M TI ⟷ Var x ∈ M"
unfolding timpl_closure_set_def by (simp add: timpl_closure_Var_in_iff[of x _ TI])

lemma timpl_closure_Var_inv:
  assumes "t ∈ timpl_closure (Var x) TI"
  shows "t = Var x"
using assms
proof (induction rule: timpl_closure.induct)
  case (TI u a b s) thus ?case using term_variants_pred_inv_Var by fast
qed simp

lemma timpls_Un_mono: "mono (λX. FP ∪ (⋃x ∈ X. ⋃(a,b) ∈ TI. set ⟨a --» b⟩⟨x⟩))"
by (auto intro!: monoI)

lemma timpl_closure_set_lfp:
  fixes M TI
  defines "f ≡ λX. M ∪ (⋃x ∈ X. ⋃(a,b) ∈ TI. set ⟨a --» b⟩⟨x⟩)"
  shows "lfp f = timpl_closure_set M TI"
proof
  note 0 = timpls_Un_mono[of M TI, unfolded f_def[symmetric]]

  let ?N = "timpl_closure_set M TI"

  show "lfp f ⊆ ?N"
  proof (induction rule: lfp_induct)
    case 2 thus ?case
    proof
      fix t assume "t ∈ f (lfp f ∩ ?N)"
      hence "t ∈ M ∨ t ∈ (⋃x ∈ ?N. ⋃(a,b) ∈ TI. set ⟨a --» b⟩⟨x⟩)" (is "?A ∨ ?B")
        unfolding f_def by blast
      thus "t ∈ ?N"
      proof
        assume ?B
        then obtain s a b where s: "s ∈ ?N" "(a,b) ∈ TI" "t ∈ set ⟨a --» b⟩⟨s⟩" by blast
        thus ?thesis
          using term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])" s]
          unfolding timpl_closure_set_def timpl_apply_term_def
          by (auto intro: timpl_closure.intros)
      qed (auto simp add: timpl_closure_set_def intro: timpl_closure.intros)
    qed
  qed (rule 0)

  have "t ∈ lfp f" when t: "t ∈ timpl_closure s TI" and s: "s ∈ M" for t s
    using t
  proof (induction t rule: timpl_closure.induct)
```

```
    case (TI u a b v) thus ?case
      using term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
            lfp_fixpoint[OF 0]
      unfolding timpl_apply_term_def f_def by fastforce
  qed (use s lfp_fixpoint[OF 0] f_def in blast)
  thus "?N ⊆ lfp f" unfolding timpl_closure_set_def by blast
qed

lemma timpl_closure_set_supset:
  assumes "∀t ∈ FP. t ∈ closure"
  and "∀t ∈ closure. ∀(a,b) ∈ TI. ∀s ∈ set ⟨a --» b⟩⟨t⟩. s ∈ closure"
  shows "timpl_closure_set FP TI ⊆ closure"
proof -
  have "t ∈ closure" when t: "t ∈ timpl_closure s TI" and s: "s ∈ FP" for t s
    using t
  proof (induction rule: timpl_closure.induct)
    case FP thus ?case using s assms(1) by blast
  next
    case (TI u a b s') thus ?case
      using assms(2) term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
      unfolding timpl_apply_term_def by fastforce
  qed
  thus ?thesis unfolding timpl_closure_set_def by blast
qed

lemma timpl_closure_set_supset':
  assumes "∀t ∈ FP. ∀(a,b) ∈ TI. ∀s ∈ set ⟨a --» b⟩⟨t⟩. s ∈ FP"
  shows "timpl_closure_set FP TI ⊆ FP"
using timpl_closure_set_supset[OF _ assms] by blast

lemma timpl_closure'_param:
  assumes "(t,s) ∈ timpl_closure' c"
    and fg: "f = g ∨ (∃a b. (a,b) ∈ c ∧ f = Abs a ∧ g = Abs b)"
  shows "(Fun f (S@t#T), Fun g (S@s#T)) ∈ timpl_closure' c"
using assms(1) unfolding timpl_closure'_def
proof (induction rule: rtrancl_induct)
  case base thus ?case
  proof (cases "f = g")
    case False
    then obtain a b where ab: "(a,b) ∈ c" "f = Abs a" "g = Abs b"
      using fg by blast
    show ?thesis
      using term_variants_pred_param[OF term_variants_pred_refl[of "(λ_. [])(Abs a := [Abs b])" t]]
            timpl_closure'_step.intros[OF ab(1)] ab(2,3)
      by fastforce
  qed simp
next
  case (step u s)
  obtain a b where ab: "(a,b) ∈ c" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u s"
    using timpl_closure'_step_inv[OF step.hyps(2)] by blast
  have "(Fun g (S@u#T), Fun g (S@s#T)) ∈ timpl_closure'_step c"
      using ab(1) term_variants_pred_param[OF ab(2), of g g S T]
      by (auto simp add: timpl_closure'_def intro: timpl_closure'_step.intros)
  thus ?case using rtrancl_into_rtrancl[OF step.IH] fg by blast
qed

lemma timpl_closure'_param':
  assumes "(t,s) ∈ timpl_closure' c"
  shows "(Fun f (S@t#T), Fun f (S@s#T)) ∈ timpl_closure' c"
using timpl_closure'_param[OF assms] by simp

lemma timpl_closure_FunI:
  assumes IH: "⋀i. i < length T ⟹ (T ! i, S ! i) ∈ timpl_closure' c"
```

```
    and len: "length T = length S"
    and fg: "f = g ∨ (∃a b. (a,b) ∈ c⁺ ∧ f = Abs a ∧ g = Abs b)"
  shows "(Fun f T, Fun g S) ∈ timpl_closure' c"
proof -
  have aux: "(Fun f T, Fun g (take n S@drop n T)) ∈ timpl_closure' c"
    when "n ≤ length T" for n
    using that
  proof (induction n)
    case 0
    have "(T ! n, T ! n) ∈ timpl_closure' c" when n: "n < length T" for n
      using n unfolding timpl_closure'_def by simp
    hence "(Fun f T, Fun g T) ∈ timpl_closure' c"
    proof (cases "f = g")
      case False
      then obtain a b where ab: "(a, b) ∈ c⁺" "f = Abs a" "g = Abs b"
        using fg by blast
      show ?thesis
        using timpl_closure'_step.intros[OF ab(1), of "Fun f T" "Fun g T"] ab(2,3)
              term_variants_P[OF _ term_variants_pred_refl[of "(λ_. [])(Abs a := [Abs b])"],
                              of T g f]
              timpl_closure'_timpls_trancl_eq
        unfolding timpl_closure'_def
        by (metis fun_upd_same list.set_intros(1) r_into_rtrancl)
    qed (simp add: timpl_closure'_def)
    thus ?case by simp
  next
    case (Suc n)
    hence IH': "(Fun f T, Fun g (take n S@drop n T)) ∈ timpl_closure' c"
      and n: "n < length T" "n < length S"
      by (simp_all add: len)

    obtain T1 T2 where T: "T = T1@T ! n#T2" "length T1 = n"
      using length_prefix_ex'[OF n(1)] by auto

    obtain S1 S2 where S: "S = S1@S ! n#S2" "length S1 = n"
      using length_prefix_ex'[OF n(2)] by auto

    have "take n S@drop n T = S1@T ! n#T2" "take (Suc n) S@drop (Suc n) T = S1@S ! n#T2"
      using n T S append_eq_conv_conj
      by (metis, metis (no_types, opaque_lifting) Cons_nth_drop_Suc append.assoc append_Cons
                                                  append_Nil take_Suc_conv_app_nth)
    moreover have "(T ! n, S ! n) ∈ timpl_closure' c" using IH Suc.prems by simp
    ultimately show ?case
      using timpl_closure'_param IH'(1)
      by (metis (no_types, lifting) timpl_closure'_def rtrancl_trans)
  qed

  show ?thesis using aux[of "length T"] len by simp
qed

lemma timpl_closure_FunI':
  assumes IH: "⋀i. i < length T ⟹ (T ! i, S ! i) ∈ timpl_closure' c"
    and len: "length T = length S"
  shows "(Fun f T, Fun f S) ∈ timpl_closure' c"
using timpl_closure_FunI[OF IH len] by simp

lemma timpl_closure_FunI2:
  fixes f g::"('a, 'b, 'c, 'd) prot_fun"
  assumes IH: "⋀i. i < length T ⟹ ∃u. (T!i, u) ∈ timpl_closure' c ∧ (S!i, u) ∈ timpl_closure' c"
    and len: "length T = length S"
    and fg: "f = g ∨ (∃a b d. (a, d) ∈ c⁺ ∧ (b, d) ∈ c⁺ ∧ f = Abs a ∧ g = Abs b)"
  shows "∃h U. (Fun f T, Fun h U) ∈ timpl_closure' c ∧ (Fun g S, Fun h U) ∈ timpl_closure' c"
proof -
```

```
  let ?P = "λi u. (T ! i, u) ∈ timpl_closure' c ∧ (S ! i, u) ∈ timpl_closure' c"

  define U where "U ≡ map (λi. SOME u. ?P i u) [0..<length T]"

  have U1: "length U = length T"
    unfolding U_def by auto

  have U2: "(T ! i, U ! i) ∈ timpl_closure' c ∧ (S ! i, U ! i) ∈ timpl_closure' c"
    when i: "i < length U" for i
    using i someI_ex[of "?P i"] IH[of i] U1 len
    unfolding U_def by simp

  show ?thesis
  proof (cases "f = g")
    case False
    then obtain a b d where abd: "(a, d) ∈ c⁺" "(b, d) ∈ c⁺" "f = Abs a" "g = Abs b"
      using fg by blast

    define h::"('a, 'b, 'c, 'd) prot_fun" where "h = Abs d"

    have "f = h ∨ (∃a b. (a, b) ∈ c⁺ ∧ f = Abs a ∧ h = Abs b)"
         "g = h ∨ (∃a b. (a, b) ∈ c⁺ ∧ g = Abs a ∧ h = Abs b)"
      using abd unfolding h_def by blast+
    thus ?thesis by (metis timpl_closure_FunI len U1 U2)
  qed (metis timpl_closure_FunI' len U1 U2)
qed

lemma timpl_closure_FunI3:
  fixes f g::"('a, 'b, 'c, 'd) prot_fun"
  assumes IH: "⋀i. i < length T ⟹ ∃u. (T!i, u) ∈ timpl_closure' c ∧ (S!i, u) ∈ timpl_closure' c"
    and len: "length T = length S"
    and fg: "f = g ∨ (∃a b d. (a, d) ∈ c ∧ (b, d) ∈ c ∧ f = Abs a ∧ g = Abs b)"
  shows "∃h U. (Fun f T, Fun h U) ∈ timpl_closure' c ∧ (Fun g S, Fun h U) ∈ timpl_closure' c"
using timpl_closure_FunI2[OF IH len] fg unfolding timpl_closure'_timpls_trancl_eq by blast

lemma timpl_closure_fv_eq:
  assumes "s ∈ timpl_closure t T"
  shows "fv s = fv t"
using assms
by (induct rule: timpl_closure.induct)
   (metis, metis term_variants_pred_fv_eq)

lemma (in stateful_protocol_model) timpl_closure_subst:
  assumes t: "wf_trm t" "∀x ∈ fv t. ∃a. Γ_v x = TAtom (Atom a)"
    and δ: "wt_subst δ" "wf_trms (subst_range δ)"
  shows "timpl_closure (t · δ) T = timpl_closure t T ·_set δ"
proof
  have "s ∈ timpl_closure t T ·_set δ"
    when "s ∈ timpl_closure (t · δ) T" for s
    using that
  proof (induction s rule: timpl_closure.induct)
    case FP thus ?case using timpl_closure.FP[of t T] by simp
  next
    case (TI u a b s)
    then obtain u' where u': "u' ∈ timpl_closure t T" "u = u' · δ" by blast

    have u'_fv: "∀x ∈ fv u'. ∃a. Γ_v x = TAtom (Atom a)"
      using timpl_closure_fv_eq[OF u'(1)] t(2) by simp
    hence u_fv: "∀x ∈ fv u. ∃a. Γ_v x = TAtom (Atom a)"
      using u'(2) wt_subst_trm''[OF δ(1)] wt_subst_const_fv_type_eq[OF _ δ(1,2), of u']
      by fastforce

    have "∀x ∈ fv u' ∪ fv s. (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ Abs a ≠ f)"
```

```
    proof (intro ballI)
      fix x assume x: "x ∈ fv u' ∪ fv s"
      then obtain c where c: "Γv x = TAtom (Atom c)"
        using u'_fv u_fv term_variants_pred_fv_eq[OF TI.hyps(3)]
        by blast

      show "(∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ Abs a ≠ f)"
      proof (cases "δ x")
        case (Fun f T)
        hence **: "Γ (Fun f T) = TAtom (Atom c)" and "wftrm (Fun f T)"
          using c wt_subst_trm''[OF δ(1), of "Var x"] δ(2)
          by fastforce+
        hence "δ x = Fun f []" using Fun const_type_inv_wf by metis
        thus ?thesis using ** by force
      qed metis
    qed
    hence *: "∀x ∈ fv u' ∪ fv s.
                (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ ((λ_. [])(Abs a := [Abs b])) f = [])"
      by fastforce

    obtain s' where s': "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u' s'" "s = s' · δ"
      using term_variants_pred_subst'[OF _ *] u'(2) TI.hyps(3)
      by blast

    show ?case using timpl_closure.TI[OF u'(1) TI.hyps(2) s'(1)] s'(2) by blast
  qed
  thus "timpl_closure (t · δ) T ⊆ timpl_closure t T ·set δ" by fast

  have "s ∈ timpl_closure (t · δ) T"
    when s: "s ∈ timpl_closure t T ·set δ" for s
  proof -
    obtain s' where s': "s' ∈ timpl_closure t T" "s = s' · δ" using s by blast
    have "s' · δ ∈ timpl_closure (t · δ) T" using s'(1)
    proof (induction s' rule: timpl_closure.induct)
      case FP thus ?case using timpl_closure.FP[of "t · δ" T] by simp
    next
      case (TI u' a b s') show ?case
        using timpl_closure.TI[OF TI.IH TI.hyps(2)]
              term_variants_pred_subst[OF TI.hyps(3)]
        by blast
    qed
    thus ?thesis using s'(2) by metis
  qed
  thus "timpl_closure t T ·set δ ⊆ timpl_closure (t · δ) T" by fast
qed

lemma (in stateful_protocol_model) timpl_closure_subst_subset:
  assumes t: "t ∈ M"
    and M: "wftrms M" "∀x ∈ fvset M. ∃a. Γv x = TAtom (Atom a)"
    and δ: "wtsubst δ" "wftrms (subst_range δ)" "ground (subst_range δ)" "subst_domain δ ⊆ fvset M"
    and M_supset: "timpl_closure t T ⊆ M"
  shows "timpl_closure (t · δ) T ⊆ M ·set δ"
proof -
  have t': "wftrm t" "∀x ∈ fv t. ∃a. Γv x = TAtom (Atom a)" using t M by auto
  show ?thesis using timpl_closure_subst[OF t' δ(1,2), of T] M_supset by blast
qed

lemma (in stateful_protocol_model) timpl_closure_set_subst_subset:
  assumes M: "wftrms M" "∀x ∈ fvset M. ∃a. Γv x = TAtom (Atom a)"
    and δ: "wtsubst δ" "wftrms (subst_range δ)" "ground (subst_range δ)" "subst_domain δ ⊆ fvset M"
    and M_supset: "timpl_closure_set M T ⊆ M"
  shows "timpl_closure_set (M ·set δ) T ⊆ M ·set δ"
using timpl_closure_subst_subset[OF _ M δ, of _ T] M_supset
```

```
      timpl_closure_set_is_timpl_closure_union[of "M ·set δ" T]
      timpl_closure_set_is_timpl_closure_union[of M T]
by auto

lemma timpl_closure_set_Union:
  "timpl_closure_set (⋃Ms) T = (⋃M ∈ Ms. timpl_closure_set M T)"
using timpl_closure_set_is_timpl_closure_union[of "⋃Ms" T]
      timpl_closure_set_is_timpl_closure_union[of _ T]
by force

lemma timpl_closure_set_Union_subst_set:
  assumes "s ∈ timpl_closure_set (⋃{M ·set δ | δ. P δ}) T"
  shows "∃δ. P δ ∧ s ∈ timpl_closure_set (M ·set δ) T"
using assms timpl_closure_set_is_timpl_closure_union[of "(⋃{M ·set δ | δ. P δ})" T]
      timpl_closure_set_is_timpl_closure_union[of _ T]
by blast

lemma timpl_closure_set_Union_subst_singleton:
  assumes "s ∈ timpl_closure_set {t · δ | δ. P δ} T"
  shows "∃δ. P δ ∧ s ∈ timpl_closure_set {t · δ} T"
using assms timpl_closure_set_is_timpl_closure_union[of "{t · δ |δ. P δ}" T]
      timpl_closureton_is_timpl_closure[of _ T]
by fast

lemma timpl_closure'_inv:
  assumes "(s, t) ∈ timpl_closure' TI"
  shows "(∃x. s = Var x ∧ t = Var x) ∨ (∃f g S T. s = Fun f S ∧ t = Fun g T ∧ length S = length T)"
using assms unfolding timpl_closure'_def
proof (induction rule: rtrancl_induct)
  case base thus ?case by (cases s) auto
next
  case (step t u)
  obtain a b where ab: "(a, b) ∈ TI" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) t u"
    using timpl_closure'_step_inv[OF step.hyps(2)] by blast
  show ?case using step.IH
  proof
    assume "∃x. s = Var x ∧ t = Var x"
    thus ?case using step.hyps(2) term_variants_pred_inv_Var ab by fastforce
  next
    assume "∃f g S T. s = Fun f S ∧ t = Fun g T ∧ length S = length T"
    then obtain f g S T where st: "s = Fun f S" "t = Fun g T" "length S = length T" by blast
    thus ?case
      using ab step.hyps(2) term_variants_pred_inv'[of "(λ_. [])(Abs a := [Abs b])" g T u]
      by auto
  qed
qed

lemma timpl_closure'_inv':
  assumes "(s, t) ∈ timpl_closure' TI"
  shows "(∃x. s = Var x ∧ t = Var x) ∨
        (∃f g S T. s = Fun f S ∧ t = Fun g T ∧ length S = length T ∧
                   (∀i < length T. (S ! i, T ! i) ∈ timpl_closure' TI) ∧
                   (f ≠ g ⟶ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI⁺))"
    (is "?A s t ∨ ?B s t (timpl_closure' TI)")
using assms unfolding timpl_closure'_def
proof (induction rule: rtrancl_induct)
  case base thus ?case by (cases s) auto
next
  case (step t u)
  obtain a b where ab: "(a, b) ∈ TI" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) t u"
    using timpl_closure'_step_inv[OF step.hyps(2)] by blast
  show ?case using step.IH
  proof
```

```
    assume "?A s t"
    thus ?case using step.hyps(2) term_variants_pred_inv_Var ab by fastforce
  next
    assume "?B s t ((timpl_closure'_step TI)*)"
    then obtain f g S T where st:
        "s = Fun f S" "t = Fun g T" "length S = length T"
        "⋀i. i < length T ⟹ (S ! i, T ! i) ∈ (timpl_closure'_step TI)*"
        "f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI+"
      by blast
    obtain h U where u:
        "u = Fun h U" "length T = length U"
        "⋀i. i < length T ⟹ term_variants_pred ((λ_. [])(Abs a := [Abs b])) (T ! i) (U ! i)"
        "g ≠ h ⟹ is_Abs g ∧ is_Abs h ∧ (the_Abs g, the_Abs h) ∈ TI+"
      using ab(2) st(2) r_into_trancl[OF ab(1)]
            term_variants_pred_inv'(1,2,3,4)[of "(λ_. [])(Abs a := [Abs b])" g T u]
            term_variants_pred_inv'(5)[of "(λ_. [])(Abs a := [Abs b])" g T u "Abs a" "Abs b"]
      unfolding is_Abs_def the_Abs_def by force

    have "(S ! i, U ! i) ∈ (timpl_closure'_step TI)*" when i: "i < length U" for i
      using u(2) i rtrancl.rtrancl_into_rtrancl[OF
              st(4)[of i] timpl_closure'_step.intros[OF ab(1) u(3)[of i]]]
      by argo
    moreover have "length S = length U" using st u by argo
    moreover have "is_Abs f ∧ is_Abs h ∧ (the_Abs f, the_Abs h) ∈ TI+" when fh: "f ≠ h"
      using fh st u by fastforce
    ultimately show ?case using st(1) u(1) by blast
  qed
qed

lemma timpl_closure'_inv'':
  assumes "(Fun f S, Fun g T) ∈ timpl_closure' TI"
  shows "length S = length T"
    and "⋀i. i < length T ⟹ (S ! i, T ! i) ∈ timpl_closure' TI"
    and "f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI+"
using assms timpl_closure'_inv' by auto

lemma timpl_closure_Fun_inv:
  assumes "s ∈ timpl_closure (Fun f T) TI"
  shows "∃g S. s = Fun g S"
using assms timpl_closure_is_timpl_closure' timpl_closure'_inv
by fastforce

lemma timpl_closure_Fun_inv':
  assumes "Fun g S ∈ timpl_closure (Fun f T) TI"
  shows "length S = length T"
    and "⋀i. i < length S ⟹ S ! i ∈ timpl_closure (T ! i) TI"
    and "f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI+"
using assms timpl_closure_is_timpl_closure'
by (metis timpl_closure'_inv''(1), metis timpl_closure'_inv''(2), metis timpl_closure'_inv''(3))

lemma timpl_closure_Fun_not_Var[simp]:
  "Fun f T ∉ timpl_closure (Var x) TI"
using timpl_closure_Var_inv by fast

lemma timpl_closure_Var_not_Fun[simp]:
  "Var x ∉ timpl_closure (Fun f T) TI"
using timpl_closure_Fun_inv by fast

lemma (in stateful_protocol_model) timpl_closure_wf_trms:
  assumes m: "wf_trm m"
  shows "wf_trms (timpl_closure m TI)"
proof
  fix t assume "t ∈ timpl_closure m TI"
```

```
  thus "wf_trm t"
  proof (induction t rule: timpl_closure.induct)
    case TI thus ?case using term_variants_pred_wf_trms by force
  qed (rule m)
qed

lemma (in stateful_protocol_model) timpl_closure_set_wf_trms:
  assumes M: "wf_trms M"
  shows "wf_trms (timpl_closure_set M TI)"
proof
  fix t assume "t ∈ timpl_closure_set M TI"
  then obtain m where "t ∈ timpl_closure m TI" "m ∈ M" "wf_trm m"
    using M timpl_closure_set_is_timpl_closure_union by blast
  thus "wf_trm t" using timpl_closure_wf_trms by blast
qed

lemma timpl_closure_Fu_inv:
  assumes "t ∈ timpl_closure (Fun (Fu f) T) TI"
  shows "∃S. length S = length T ∧ t = Fun (Fu f) S"
using assms
proof (induction t rule: timpl_closure.induct)
  case (TI u a b s)
  then obtain U where U: "length U = length T" "u = Fun (Fu f) U"
    by blast
  hence *: "term_variants_pred ((λ_. [])(Abs a := [Abs b])) (Fun (Fu f) U) s"
    using TI.hyps(3) by meson

  show ?case
    using term_variants_pred_inv'(1,2,4)[OF *] U
    by force
qed simp

lemma timpl_closure_Fu_inv':
  assumes "Fun (Fu f) T ∈ timpl_closure t TI"
  shows "∃S. length S = length T ∧ t = Fun (Fu f) S"
using assms
proof (induction "Fun (Fu f) T" arbitrary: T rule: timpl_closure.induct)
  case (TI u a b)
  obtain g U where U:
      "u = Fun g U" "length U = length T"
      "Fu f ≠ g ⟹ Abs a = g ∧ Fu f = Abs b"
    using term_variants_pred_inv''[OF TI.hyps(4)] by fastforce

  have g: "g = Fu f" using U(3) by blast

  show ?case using TI.hyps(2)[OF U(1)[unfolded g]] U(2) by auto
qed simp

lemma timpl_closure_no_Abs_eq:
  assumes "t ∈ timpl_closure s TI"
    and "∀f ∈ funs_term t. ¬is_Abs f"
  shows "t = s"
using assms
proof (induction t rule: timpl_closure.induct)
  case (TI t a b s) thus ?case
    using term_variants_pred_eq_case_Abs[of a b t s]
    unfolding timpl_apply_term_def term_variants_pred_iff_in_term_variants[symmetric]
    by metis
qed simp

lemma timpl_closure_set_no_Abs_in_set:
  assumes "t ∈ timpl_closure_set FP TI"
    and "∀f ∈ funs_term t. ¬is_Abs f"
```

```
  shows "t ∈ FP"
using assms timpl_closure_no_Abs_eq unfolding timpl_closure_set_def by blast

lemma timpl_closure_funs_term_subset:
  "⋃(funs_term ` (timpl_closure t TI)) ⊆ funs_term t ∪ Abs ` snd ` TI"
  (is "?A ⊆ ?B ∪ ?C")
proof
  fix f assume "f ∈ ?A"
  then obtain s where "s ∈ timpl_closure t TI" "f ∈ funs_term s" by blast
  thus "f ∈ ?B ∪ ?C"
  proof (induction s rule: timpl_closure.induct)
    case (TI u a b s)
    have "Abs b ∈ Abs ` snd ` TI" using TI.hyps(2) by force
    thus ?case using term_variants_pred_funs_term[OF TI.hyps(3) TI.prems] TI.IH by force
  qed blast
qed

lemma timpl_closure_set_funs_term_subset:
  "⋃(funs_term ` (timpl_closure_set FP TI)) ⊆ ⋃(funs_term ` FP) ∪ Abs ` snd ` TI"
using timpl_closure_funs_term_subset[of _ TI]
      timpl_closure_set_is_timpl_closure_union[of FP TI]
by auto

lemma funs_term_OCC_TI_subset:
  defines "absc ≡ λa. Fun (Abs a) []"
  assumes OCC1: "∀t ∈ FP. ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` OCC"
    and OCC2: "snd ` TI ⊆ OCC"
  shows "∀t ∈ timpl_closure_set FP TI. ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` OCC" (is ?A)
    and "∀t ∈ absc ` OCC. ∀(a,b) ∈ TI. ∀s ∈ set ⟨a --» b⟩⟨t⟩. s ∈ absc ` OCC" (is ?B)
proof -
  let ?F = "⋃(funs_term ` FP)"
  let ?G = "Abs ` snd ` TI"

  show ?A
  proof (intro ballI impI)
    fix t f assume t: "t ∈ timpl_closure_set FP TI" and f: "f ∈ funs_term t" "is_Abs f"
    hence "f ∈ ?F ∨ f ∈ ?G" using timpl_closure_set_funs_term_subset[of FP TI] by auto
    thus "f ∈ Abs ` OCC"
    proof
      assume "f ∈ ?F" thus ?thesis using OCC1 f(2) by fast
    next
      assume "f ∈ ?G" thus ?thesis using OCC2 by auto
    qed
  qed

  { fix s t a b
    assume t: "t ∈ absc ` OCC"
      and ab: "(a, b) ∈ TI"
      and s: "s ∈ set ⟨a --» b⟩⟨t⟩"
    obtain c where c: "t = absc c" "c ∈ OCC" using t by blast
    hence "s = absc b ∨ s = absc c"
      using ab s timpl_apply_const'[of c a b] unfolding absc_def by auto
    moreover have "b ∈ OCC" using ab OCC2 by auto
    ultimately have "s ∈ absc ` OCC" using c(2) by blast
  } thus ?B by blast
qed

lemma (in stateful_protocol_model) intruder_synth_timpl_closure_set:
  fixes M::"('fun,'atom,'sets,'lbl) prot_terms" and t::"('fun,'atom,'sets,'lbl) prot_term"
  assumes "M ⊢c t"
    and "s ∈ timpl_closure t TI"
  shows "timpl_closure_set M TI ⊢c s"
using assms
```

```
proof (induction t arbitrary: s rule: intruder_synth_induct)
  case (AxiomC t)
  hence "s ∈ timpl_closure_set M TI"
    using timpl_closure_set_is_timpl_closure_union[of M TI]
    by blast
  thus ?case by simp
next
  case (ComposeC T f)
  obtain g S where s: "s = Fun g S"
    using timpl_closure_Fun_inv[OF ComposeC.prems] by blast
  hence s':
      "f = g" "length S = length T"
      "⋀i. i < length S ⟹ S ! i ∈ timpl_closure (T ! i) TI"
    using timpl_closure_Fun_inv'[of g S f T TI] ComposeC.prems ComposeC.hyps(2)
    unfolding is_Abs_def by fastforce+

  have "timpl_closure_set M TI ⊢c u" when u: "u ∈ set S" for u
    using ComposeC.IH u s'(2,3) in_set_conv_nth[of _ T] in_set_conv_nth[of u S] by auto
  thus ?case
    using s s'(1,2) ComposeC.hyps(1,2) intruder_synth.ComposeC[of S g "timpl_closure_set M TI"]
    by argo
qed

lemma (in stateful_protocol_model) intruder_synth_timpl_closure':
  fixes M::"('fun,'atom,'sets,'lbl) prot_terms" and t::"('fun,'atom,'sets,'lbl) prot_term"
  assumes "timpl_closure_set M TI ⊢c t"
    and "s ∈ timpl_closure t TI"
  shows "timpl_closure_set M TI ⊢c s"
by (metis intruder_synth_timpl_closure_set[OF assms] timpl_closure_set_idem)

lemma timpl_closure_set_absc_subset_in:
  defines "absc ≡ λa. Fun (Abs a) []"
  assumes A: "timpl_closure_set (absc ` A) TI ⊆ absc ` A"
    and a: "a ∈ A" "(a,b) ∈ TI⁺"
  shows "b ∈ A"
proof -
  have "timpl_closure (absc a) (TI⁺) ⊆ absc ` A"
    using a(1) A timpl_closure_timpls_trancl_eq
    unfolding timpl_closure_set_def by fast
  thus ?thesis
    using timpl_closure.TI[OF timpl_closure.FP[of "absc a"] a(2), of "absc b"]
          term_variants_P[of "[]" "[]" "(λ_. [])(Abs a := [Abs b])" "Abs b" "Abs a"]
    unfolding absc_def by auto
qed

lemma timpl_closure_Abs_ex:
  assumes t: "s ∈ timpl_closure t TI"
    and a: "Abs a ∈ funs_term t"
  shows "∃b ts. (a,b) ∈ TI* ∧ Fun (Abs b) ts ⊑ s"
using t
proof (induction rule: timpl_closure.induct)
  case (TI u b c s)
  obtain d ts where d: "(a,d) ∈ TI*" "Fun (Abs d) ts ⊑ u" using TI.IH by blast
  note 0 = TI.hyps(2) d(1) term_variants_pred_inv'(5)[OF term_variants_pred_const]
  show ?case using TI.hyps(3) d(2)
  proof (induction rule: term_variants_pred.induct)
    case (term_variants_P T S g f)
    note hyps = term_variants_P.hyps
    note prems = term_variants_P.prems
    note IH = term_variants_P.IH
    show ?case
    proof (cases "Fun (Abs d) ts = Fun f T")
      case False
```

```
      hence "∃t ∈ set T. Fun (Abs d) ts ⊑ t" using prems(1) by force
      then obtain i where i: "i < length T" "Fun (Abs d) ts ⊑ T ! i" by (metis in_set_conv_nth)
      show ?thesis by (metis IH[OF i] i(1) hyps(1) nth_mem subtermeqI'' term.order.trans)
    qed (metis hyps(3) 0 prot_fun.sel(4) r_into_rtrancl rtrancl_trans term.eq_refl term.sel(2))
  next
    case (term_variants_Fun T S f)
    note hyps = term_variants_Fun.hyps
    note prems = term_variants_Fun.prems
    note IH = term_variants_Fun.IH
    show ?case
    proof (cases "Fun (Abs d) ts = Fun f T")
      case False
      hence "∃t ∈ set T. Fun (Abs d) ts ⊑ t" using prems(1) by force
      then obtain i where i: "i < length T" "Fun (Abs d) ts ⊑ T ! i" by (metis in_set_conv_nth)
      show ?thesis by (metis IH[OF i] i(1) hyps(1) nth_mem subtermeqI'' term.order.trans)
    qed (metis 0(2) term.eq_refl term.sel(2))
  qed simp
qed (meson a funs_term_Fun_subterm rtrancl_eq_or_trancl)

lemma timpl_closure_trans:
  assumes "s ∈ timpl_closure t TI"
    and "u ∈ timpl_closure s TI"
  shows "u ∈ timpl_closure t TI"
using assms unfolding timpl_closure_is_timpl_closure' timpl_closure'_def by simp

lemma (in stateful_protocol_model) term_variants_pred_Ana_f_keys:
  assumes
    "length ss = length ts"
    "∀x ∈ fv k. x < length ss"
    "⋀i. i < length ss ⟹ term_variants_pred P (ss ! i) (ts ! i)"
  shows "term_variants_pred P (k · (!) ss) (k · (!) ts)"
using assms by (meson term_variants_pred_subst'')

lemma (in stateful_protocol_model) term_variants_pred_Ana_keys:
  fixes a b and s t::"('fun,'atom,'sets,'lbl) prot_term"
  defines "P ≡ ((λ_. [])(Abs a := [Abs b]))"
  assumes ab: "term_variants_pred P s t"
    and s: "Ana s = (Ks, Rs)"
    and t: "Ana t = (Kt, Rt)"
  shows "length Kt = length Ks" (is ?A)
    and "∀i < length Ks. term_variants_pred P (Ks ! i) (Kt ! i)" (is ?B)
proof -
  have ?A ?B when "s = Var x" for x
    using that ab s t iffD1[OF term_variants_pred_inv_Var(1) ab[unfolded that]] by auto
  moreover have ?A ?B when s': "s = Fun f ss" for f ss
  proof -
    obtain g ts where t':
        "t = Fun g ts" "length ss = length ts"
        "⋀i. i < length ss ⟹ term_variants_pred P (ss ! i) (ts ! i)"
        "f ≠ g ⟹ f = Abs a ∧ g = Abs b"
      using term_variants_pred_inv'[OF ab[unfolded s']]
      unfolding P_def by fastforce
    have ?A ?B when "f ≠ g" using that s s' t t'(1,2,4) by auto
    moreover have A: ?A when fg: "f = g"
      using s t Ana_Fu_keys_length_eq[OF t'(2)] Ana_nonempty_inv[of s] Ana_nonempty_inv[of t]
      unfolding fg s' t'(1) by (metis (no_types) fst_conv term.inject(2))
    moreover have ?B when fg: "f = g"
    proof (cases "Ana s = ([],[])")
      case True thus ?thesis using s t t'(2,3) A[OF fg] unfolding fg s' t'(1) by auto
    next
      case False
      then obtain h Kh Rh where h:
          "f = Fu h" "g = Fu h" "arity_f h = length ss" "arity_f h > 0" "Ana_f h = (Kh, Rh)"
```

```
          "Ana s = (Kh ·list (!) ss, map ((!) ss) Rh)" "Ana t = (Kh ·list (!) ts, map ((!) ts) Rh)"
        using A[OF fg] s t t'(2,3) Ana_nonempty_inv[of s] Ana_nonempty_inv[of t]
        unfolding fg s' t'(1) by fastforce
      show ?thesis
      proof (intro allI impI)
        fix i assume i: "i < length Ks"
        have Ks: "Ks = Kh ·list (!) ss" and Kt: "Kt = Kh ·list (!) ts"
          using h(6,7) s t by auto

        have 0: "Kh ! i ∈ set Kh" using Ks i by simp

        have 1: "∀x ∈ fv (Kh ! i). x < length ss"
          using 0 Ana_f_assm2_alt[OF h(5)]
          unfolding h(1-3) by fastforce

        have "term_variants_pred P (Kh ! i · (!) ss) (Kh ! i · (!) ts)"
          using term_variants_pred_Ana_f_keys[OF t'(2) 1 t'(3)]
          unfolding P_def by fast
        thus "term_variants_pred P (Ks ! i) (Kt ! i)"
          using i unfolding Ks Kt by simp
      qed
    qed
    ultimately show ?A ?B by fast+
  qed
  ultimately show ?A ?B by (cases s; simp_all)+
qed

lemma (in stateful_protocol_model) timpl_closure_Ana_keys:
  fixes s t::"('fun,'atom,'sets,'lbl) prot_term"
  assumes "t ∈ timpl_closure s TI"
    and "Ana s = (Ks, Rs)"
    and "Ana t = (Kt, Rt)"
  shows "length Kt = length Ks" (is ?A)
    and "∀i < length Ks. Kt ! i ∈ timpl_closure (Ks ! i) TI" (is ?B)
using assms
proof (induction arbitrary: Ks Rs Kt Rt rule: timpl_closure.induct)
  case FP
  { case 1 thus ?case by simp }
  { case 2 thus ?case using FP timpl_closure.FP by force }
next
  case (TI u a b t)
  obtain Ku Ru where u: "Ana u = (Ku, Ru)" by (metis surj_pair)
  note 0 = term_variants_pred_Ana_keys[OF TI.hyps(3) u]
  { case 1 thus ?case using 0(1) TI.IH(1) u by fastforce }
  { case 2 thus ?case by (metis 0(2)[OF 2(2)] TI.IH[OF 2(1) u] timpl_closure.TI[OF _ TI.hyps(2)]) }
qed

lemma (in stateful_protocol_model) timpl_closure_Ana_keys_length_eq:
  fixes s t::"('fun,'atom,'sets,'lbl) prot_term"
  assumes "t ∈ timpl_closure s TI"
    and "Ana s = (Ks, Rs)"
    and "Ana t = (Kt, Rt)"
  shows "length Kt = length Ks"
by (rule timpl_closure_Ana_keys(1,2)[OF assms])

lemma (in stateful_protocol_model) timpl_closure_Ana_keys_subset:
  fixes s t::"('fun,'atom,'sets,'lbl) prot_term"
  assumes "t ∈ timpl_closure s TI"
    and "Ana s = (Ks, Rs)"
    and "Ana t = (Kt, Rt)"
  shows "set Kt ⊆ timpl_closure_set (set Ks) TI"
proof -
  note 0 = timpl_closure_Ana_keys[OF assms]
```

```
  have "∀i < length Ks. Kt ! i ∈ timpl_closure_set (set Ks) TI"
    using in_set_conv_nth 0(2) unfolding timpl_closure_set_def by auto
  thus "set Kt ⊆ timpl_closure_set (set Ks) TI"
    using 0(1) by (metis subsetI in_set_conv_nth)
qed
```

3.5.3 Composition-only Intruder Deduction Modulo Term Implication Closure of the Intruder Knowledge

```
context stateful_protocol_model
begin

fun in_trancl where
  "in_trancl TI a b = (
    if (a,b) ∈ set TI then True
    else list_ex (λ(c,d). c = a ∧ in_trancl (removeAll (c,d) TI) d b) TI)"

definition in_rtrancl where
  "in_rtrancl TI a b ≡ a = b ∨ in_trancl TI a b"

declare in_trancl.simps[simp del]

fun timpls_transformable_to where
  "timpls_transformable_to TI (Var x) (Var y) = (x = y)"
| "timpls_transformable_to TI (Fun f T) (Fun g S) = (
    (f = g ∨ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)) ∧
    list_all2 (timpls_transformable_to TI) T S)"
| "timpls_transformable_to _ _ _ = False"

fun timpls_transformable_to' where
  "timpls_transformable_to' TI (Var x) (Var y) = (x = y)"
| "timpls_transformable_to' TI (Fun f T) (Fun g S) = (
    (f = g ∨ (is_Abs f ∧ is_Abs g ∧ in_trancl TI (the_Abs f) (the_Abs g))) ∧
    list_all2 (timpls_transformable_to' TI) T S)"
| "timpls_transformable_to' _ _ _ = False"

fun equal_mod_timpls where
  "equal_mod_timpls TI (Var x) (Var y) = (x = y)"
| "equal_mod_timpls TI (Fun f T) (Fun g S) = (
    (f = g ∨ (is_Abs f ∧ is_Abs g ∧
              ((the_Abs f, the_Abs g) ∈ set TI ∨
               (the_Abs g, the_Abs f) ∈ set TI ∨
               (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧ (the_Abs g, snd ti) ∈ set TI)))) ∧
    list_all2 (equal_mod_timpls TI) T S)"
| "equal_mod_timpls _ _ _ = False"

fun intruder_synth_mod_timpls where
  "intruder_synth_mod_timpls M TI (Var x) = List.member M (Var x)"
| "intruder_synth_mod_timpls M TI (Fun f T) = (
    (list_ex (λt. timpls_transformable_to TI t (Fun f T)) M) ∨
    (public f ∧ length T = arity f ∧ list_all (intruder_synth_mod_timpls M TI) T))"

fun intruder_synth_mod_timpls' where
  "intruder_synth_mod_timpls' M TI (Var x) = List.member M (Var x)"
| "intruder_synth_mod_timpls' M TI (Fun f T) = (
    (list_ex (λt. timpls_transformable_to' TI t (Fun f T)) M) ∨
    (public f ∧ length T = arity f ∧ list_all (intruder_synth_mod_timpls' M TI) T))"

fun intruder_synth_mod_eq_timpls where
  "intruder_synth_mod_eq_timpls M TI (Var x) = (Var x ∈ M)"
| "intruder_synth_mod_eq_timpls M TI (Fun f T) = (
    (∃t ∈ M. equal_mod_timpls TI t (Fun f T)) ∨
    (public f ∧ length T = arity f ∧ list_all (intruder_synth_mod_eq_timpls M TI) T))"
```

```
definition analyzed_closed_mod_timpls where
  "analyzed_closed_mod_timpls M TI ≡
    let ti = intruder_synth_mod_timpls M TI;
        cl = λts. comp_timpl_closure ts (set TI);
        f = list_all ti;
        g = λt. if f (fst (Ana t)) then f (snd (Ana t))
                else if list_all (λt. ∀f ∈ funs_term t. ¬is_Abs f) (fst (Ana t)) then True
                else if ∀s ∈ cl (set (fst (Ana t))). ¬ti s then True
                else ∀s ∈ cl {t}. case Ana s of (K,R) ⇒ f K ⟶ f R
    in list_all g M"

definition analyzed_closed_mod_timpls' where
  "analyzed_closed_mod_timpls' M TI ≡
    let f = list_all (intruder_synth_mod_timpls' M TI);
        g = λt. if f (fst (Ana t)) then f (snd (Ana t))
                else if list_all (λt. ∀f ∈ funs_term t. ¬is_Abs f) (fst (Ana t)) then True
                else ∀s ∈ comp_timpl_closure {t} (set TI). case Ana s of (K,R) ⇒ f K ⟶ f R
    in list_all g M"

lemma term_variants_pred_Abs_Ana_keys:
  fixes a b
  defines "P ≡ ((λ_. [])(Abs a := [Abs b]))"
  assumes st: "term_variants_pred P s t"
  shows "length (fst (Ana s)) = length (fst (Ana t))" (is "?P s t")
    and "∀i < length (fst (Ana s)). term_variants_pred P (fst (Ana s) ! i) (fst (Ana t) ! i)"
        (is "?Q s t")
proof -
  show "?P s t" using st
  proof (induction s t rule: term_variants_pred.induct)
    case (term_variants_Fun T S f) show ?case
    proof (cases f)
      case (Fu g) thus ?thesis using term_variants_Fun Ana_Fu_keys_length_eq by blast
    qed simp_all
  qed (simp_all add: P_def)

  show "?Q s t" using st
  proof (induction s t rule: term_variants_pred.induct)
    case (term_variants_Fun T S f)
    note hyps = term_variants_Fun.hyps
    let ?K = "λU. fst (Ana (Fun f U))"

    show ?case
    proof (cases f)
      case (Fu g) show ?thesis
      proof (cases "arity_f g = length T ∧ arity_f g > 0")
        case True
        hence *: "?K T = fst (Ana_f g) ·_list (!) T"
                 "?K S = fst (Ana_f g) ·_list (!) S"
          using Fu Ana_Fu_intro fst_conv prod.collapse
          by (metis (mono_tags, lifting), metis (mono_tags, lifting) hyps(1))

        have K: "j < length T" when j: "j ∈ fv_set (set (fst (Ana_f g)))" for j
          using True Ana_f_assm2_alt[of g "fst (Ana_f g)" _ j ]
          by (metis UnI1 prod.collapse that)

        show ?thesis
        proof (intro allI impI)
          fix i assume i: "i < length (?K T)"
          let ?u = "fst (Ana_f g) ! i"

          have **: "?K T ! i = ?u · (!) T" "?K S ! i = ?u · (!) S"
            using * i by simp_all
```

```
          have ***: "x < length T" when "x ∈ fv (fst (Ana_f g) ! i)" for x
            using that K Ana_f_assm2_alt[of g "fst (Ana_f g)" _ x] i hyps(1)
            unfolding * by force

          show "term_variants_pred P (?K T ! i) (?K S ! i)"
            using i hyps K *** term_variants_pred_subst''[of ?u P "(!) T" "(!) S"]
            unfolding * by auto
        qed
      qed (auto simp add: Fu)
    qed simp_all
  qed (simp_all add: P_def)
qed

lemma term_variants_pred_Abs_eq_case:
  assumes t: "term_variants_pred ((λ_. [])(Abs a := [Abs b])) s t" (is "?R s t")
    and s: "∀f ∈ funs_term s. ¬is_Abs f" (is "?P s")
  shows "s = t"
using s term_variants_pred_eq_case[OF t] by fastforce

lemma term_variants_Ana_keys_no_Abs_eq_case:
  fixes s t::"(('fun,'atom,'sets,'lbl) prot_fun,'v) term"
  assumes t: "term_variants_pred ((λ_. [])(Abs a := [Abs b])) s t" (is "?R s t")
    and s: "∀t ∈ set (fst (Ana s)). ∀f ∈ funs_term t. ¬is_Abs f" (is "?P s")
  shows "fst (Ana t) = fst (Ana s)" (is "?Q t s")
using s term_variants_pred_Abs_Ana_keys[OF t] term_variants_pred_Abs_eq_case[of a b]
by (metis nth_equalityI nth_mem)

lemma timpl_closure_Ana_keys_no_Abs_eq_case:
  assumes t: "t ∈ timpl_closure s TI"
    and s: "∀t ∈ set (fst (Ana s)). ∀f ∈ funs_term t. ¬is_Abs f" (is "?P s")
  shows "fst (Ana t) = fst (Ana s)"
using t
proof (induction t rule: timpl_closure.induct)
  case (TI u a b t)
  thus ?case using s term_variants_Ana_keys_no_Abs_eq_case by fastforce
qed simp

lemma in_trancl_closure_iff_in_trancl_fun:
  "(a,b) ∈ (set TI)⁺ ⟷ in_trancl TI a b" (is "?A TI a b ⟷ ?B TI a b")
proof
  show "?A TI a b ⟹ ?B TI a b"
  proof (induction rule: trancl_induct)
    case (step c d)
    show ?case using step.IH step.hyps(2)
    proof (induction TI a c rule: in_trancl.induct)
      case (1 TI a b)
      have "(a,b) ∈ set TI ∨ (∃e. (a,e) ∈ set TI ∧ in_trancl (removeAll (a,e) TI) e b)"
        using "1.prems"(1) in_trancl.simps[of TI a b]
        unfolding list_ex_iff by (cases "(a,b) ∈ set TI") auto
      show ?case
      proof (cases "(a,b) ∈ set TI")
        case True thus ?thesis
          by (metis (mono_tags, lifting) in_trancl.simps "1.prems"(2) list_ex_iff case_prodI
                member_remove prod.inject remove_code(1))
      next
        case F: False
        then obtain e where e: "(a,e) ∈ set TI" "in_trancl (removeAll (a,e) TI) e b"
          using "1.prems"(1) in_trancl.simps[of TI a b]
          unfolding list_ex_iff by (cases "(a,b) ∈ set TI") auto
        show ?thesis
        proof (cases "(b, d) ∈ set (removeAll (a, e) TI)")
          case True thus ?thesis
```

```
          using in_trancl.simps[of TI a d] e(1) "1.prems"(2) "1.IH"[OF F e(1) _ e(2)]
          unfolding list_ex_iff by auto
      next
        case False thus ?thesis using in_trancl.simps[of TI a d] "1.prems"(2) by simp
      qed
    qed
  qed
qed (metis in_trancl.simps)

show "?B TI a b ⟹ ?A TI a b"
proof (induction TI a b rule: in_trancl.induct)
  case (1 TI a b)
  let ?P = "λTI a b c d. in_trancl (List.removeAll (c,d) TI) d b"
  have *: "∃(c,d) ∈ set TI. c = a ∧ ?P TI a b c d" when "(a,b) ∉ set TI"
    using that "1.prems" list_ex_iff[of _ TI] in_trancl.simps[of TI a b]
    by auto
  show ?case
  proof (cases "(a,b) ∈ set TI")
    case False
    hence "∃(c,d) ∈ set TI. c = a ∧ ?P TI a b c d" using * by blast
    then obtain d where d: "(a,d) ∈ set TI" "?P TI a b a d" by blast
    have "(d,b) ∈ (set (removeAll (a,d) TI))⁺" using "1.IH"[OF False d(1)] d(2) by blast
    moreover have "set (removeAll (a,d) TI) ⊆ set TI" by simp
    ultimately have "(d,b) ∈ (set TI)⁺" using trancl_mono by blast
    thus ?thesis using d(1) by fastforce
  qed simp
qed
qed

lemma in_rtrancl_closure_iff_in_rtrancl_fun:
  "(a,b) ∈ (set TI)* ⟷ in_rtrancl TI a b"
by (metis rtrancl_eq_or_trancl in_trancl_closure_iff_in_trancl_fun in_rtrancl_def)

lemma in_trancl_mono:
  assumes "set TI ⊆ set TI'"
    and "in_trancl TI a b"
  shows "in_trancl TI' a b"
by (metis assms in_trancl_closure_iff_in_trancl_fun trancl_mono)

lemma equal_mod_timpls_refl:
  "equal_mod_timpls TI t t"
proof (induction t)
  case (Fun f T) thus ?case
    using list_all2_conv_all_nth[of "equal_mod_timpls TI" T T] by force
qed simp

lemma equal_mod_timpls_inv_Var:
  "equal_mod_timpls TI (Var x) t ⟹ t = Var x" (is "?A ⟹ ?C")
  "equal_mod_timpls TI t (Var x) ⟹ t = Var x" (is "?B ⟹ ?C")
proof -
  show "?A ⟹ ?C" by (cases t) auto
  show "?B ⟹ ?C" by (cases t) auto
qed

lemma equal_mod_timpls_inv:
  assumes "equal_mod_timpls TI (Fun f T) (Fun g S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ equal_mod_timpls TI (T ! i) (S ! i)"
    and "f ≠ g ⟹ (is_Abs f ∧ is_Abs g ∧ (
                  (the_Abs f, the_Abs g) ∈ set TI ∨ (the_Abs g, the_Abs f) ∈ set TI ∨
                  (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧
                                 (the_Abs g, snd ti) ∈ set TI)))"
using assms list_all2_conv_all_nth[of "equal_mod_timpls TI" T S]
```

```
by (auto elim: equal_mod_timpls.cases)

lemma equal_mod_timpls_inv':
  assumes "equal_mod_timpls TI (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ equal_mod_timpls TI (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ (is_Abs f ∧ is_Abs (the_Fun t) ∧ (
                  (the_Abs f, the_Abs (the_Fun t)) ∈ set TI ∨
                  (the_Abs (the_Fun t), the_Abs f) ∈ set TI ∨
                  (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧
                                 (the_Abs (the_Fun t), snd ti) ∈ set TI)))"
    and "¬is_Abs f ⟹ f = the_Fun t"
using assms list_all2_conv_all_nth[of "equal_mod_timpls TI" T]
by (cases t; auto)+

lemma equal_mod_timpls_if_term_variants:
  fixes s t::"(('a, 'b, 'c, 'd) prot_fun, 'e) term" and a b::"'c set"
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes st: "term_variants_pred P s t"
    and ab: "(a,b) ∈ set TI"
  shows "equal_mod_timpls TI s t"
using st P_def
proof (induction rule: term_variants_pred.induct)
  case (term_variants_P T S f) thus ?case
    using ab list_all2_conv_all_nth[of "equal_mod_timpls TI" T S]
          in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    by auto
next
  case (term_variants_Fun T S f) thus ?case
    using ab list_all2_conv_all_nth[of "equal_mod_timpls TI" T S]
          in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    by auto
qed simp

lemma equal_mod_timpls_mono:
  assumes "set TI ⊆ set TI'"
    and "equal_mod_timpls TI s t"
  shows "equal_mod_timpls TI' s t"
  using assms
proof (induction TI s t rule: equal_mod_timpls.induct)
  case (2 TI f T g S)
  have *: "f = g ∨ (is_Abs f ∧ is_Abs g ∧ ((the_Abs f, the_Abs g) ∈ set TI ∨
                (the_Abs g, the_Abs f) ∈ set TI ∨
                (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧
                               (the_Abs g, snd ti) ∈ set TI)))"
          "list_all2 (equal_mod_timpls TI) T S"
    using "2.prems" by simp_all

  show ?case
    using "2.IH"[OF _ _ "2.prems"(1)] list.rel_mono_strong[OF *(2), of "equal_mod_timpls TI'"]
          *(1) in_trancl_mono[OF "2.prems"(1)] set_rev_mp[OF _ "2.prems"(1)]
          equal_mod_timpls.simps(2)[of TI' f T g S]
    by metis
qed auto

lemma equal_mod_timpls_refl_minus_eq:
  "equal_mod_timpls TI s t ⟷ equal_mod_timpls (filter (λ(a,b). a ≠ b) TI) s t"
  (is "?A ⟷ ?B")
proof
  show ?A when ?B using that equal_mod_timpls_mono[of "filter (λ(a,b). a ≠ b) TI" TI] by auto

  show ?B when ?A using that
```

```
  proof (induction TI s t rule: equal_mod_timpls.induct)
    case (2 TI f T g S)
    define TI' where "TI' ≡ filter (λ(a,b). a ≠ b) TI"

    let ?P = "λX Y. f = g ∨ (is_Abs f ∧ is_Abs g ∧ ((the_Abs f, the_Abs g) ∈ set X ∨
                (the_Abs g, the_Abs f) ∈ set X ∨ (∃ti ∈ set Y.
                (the_Abs f, snd ti) ∈ set X ∧ (the_Abs g, snd ti) ∈ set X)))"

    have *: "?P TI TI" "list_all2 (equal_mod_timpls TI) T S"
      using "2.prems" by simp_all

    have "?P TI' TI"
      using *(1) unfolding TI'_def is_Abs_def by auto
    hence "?P TI' TI'"
      by (metis (no_types, lifting) snd_conv)
    moreover have "list_all2 (equal_mod_timpls TI') T S"
      using *(2) "2.IH" list.rel_mono_strong unfolding TI'_def by blast
    ultimately show ?case unfolding TI'_def by force
  qed auto
qed

lemma timpls_transformable_to_refl:
  "timpls_transformable_to TI t t" (is ?A)
  "timpls_transformable_to' TI t t" (is ?B)
by (induct t) (auto simp add: list_all2_conv_all_nth)

lemma timpls_transformable_to_inv_Var:
  "timpls_transformable_to TI (Var x) t ⟹ t = Var x" (is "?A ⟹ ?C")
  "timpls_transformable_to TI t (Var x) ⟹ t = Var x" (is "?B ⟹ ?C")
  "timpls_transformable_to' TI (Var x) t ⟹ t = Var x" (is "?A' ⟹ ?C")
  "timpls_transformable_to' TI t (Var x) ⟹ t = Var x" (is "?B' ⟹ ?C")
by (cases t; auto)+

lemma timpls_transformable_to_inv:
  assumes "timpls_transformable_to TI (Fun f T) (Fun g S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ timpls_transformable_to TI (T ! i) (S ! i)"
    and "f ≠ g ⟹ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)"
using assms list_all2_conv_all_nth[of "timpls_transformable_to TI" T S] by auto

lemma timpls_transformable_to'_inv:
  assumes "timpls_transformable_to' TI (Fun f T) (Fun g S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ timpls_transformable_to' TI (T ! i) (S ! i)"
    and "f ≠ g ⟹ (is_Abs f ∧ is_Abs g ∧ in_trancl TI (the_Abs f) (the_Abs g))"
using assms list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S] by auto

lemma timpls_transformable_to_inv':
  assumes "timpls_transformable_to TI (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ timpls_transformable_to TI (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ (
          is_Abs f ∧ is_Abs (the_Fun t) ∧ (the_Abs f, the_Abs (the_Fun t)) ∈ set TI)"
    and "¬is_Abs f ⟹ f = the_Fun t"
using assms list_all2_conv_all_nth[of "timpls_transformable_to TI" T]
by (cases t; auto)+

lemma timpls_transformable_to'_inv':
  assumes "timpls_transformable_to' TI (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ timpls_transformable_to' TI (T ! i) (args t ! i)"
```

```
    and "f ≠ the_Fun t ⟹ (
          is_Abs f ∧ is_Abs (the_Fun t) ∧ in_trancl TI (the_Abs f) (the_Abs (the_Fun t)))"
    and "¬is_Abs f ⟹ f = the_Fun t"
using assms list_all2_conv_all_nth[of "timpls_transformable_to' TI" T]
by (cases t; auto)+

lemma timpls_transformable_to_size_eq:
  fixes s t::"(('a, 'b, 'c, 'd) prot_fun, 'e) term"
  shows "timpls_transformable_to TI s t ⟹ size s = size t" (is "?A ⟹ ?C")
    and "timpls_transformable_to' TI s t ⟹ size s = size t" (is "?B ⟹ ?C")
proof -
  have *: "size_list size T = size_list size S"
    when "length T = length S" "⋀i. i < length T ⟹ size (T ! i) = size (S ! i)"
    for S T::"(('a, 'b, 'c, 'd) prot_fun, 'e) term list"
    using that
  proof (induction T arbitrary: S)
    case (Cons x T')
    then obtain y S' where y: "S = y#S'" by (cases S) auto
    hence "size_list size T' = size_list size S'" "size x = size y"
      using Cons.prems Cons.IH[of S'] by force+
    thus ?case using y by simp
  qed simp

  show ?C when ?A using that
  proof (induction rule: timpls_transformable_to.induct)
    case (2 TI f T g S)
    hence "length T = length S" "⋀i. i < length T ⟹ size (T ! i) = size (S ! i)"
      using timpls_transformable_to_inv(1,2)[of TI f T g S] by auto
    thus ?case using *[of S T] by simp
  qed simp_all

  show ?C when ?B using that
  proof (induction rule: timpls_transformable_to.induct)
    case (2 TI f T g S)
    hence "length T = length S" "⋀i. i < length T ⟹ size (T ! i) = size (S ! i)"
      using timpls_transformable_to'_inv(1,2)[of TI f T g S] by auto
    thus ?case using *[of S T] by simp
  qed simp_all
qed

lemma timpls_transformable_to_if_term_variants:
  fixes s t::"(('a, 'b, 'c, 'd) prot_fun, 'e) term" and a b::"'c set"
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes st: "term_variants_pred P s t"
    and ab: "(a,b) ∈ set TI"
  shows "timpls_transformable_to TI s t"
using st P_def
proof (induction rule: term_variants_pred.induct)
  case (term_variants_P T S f) thus ?case
    using ab list_all2_conv_all_nth[of "timpls_transformable_to TI" T S]
    by auto
next
  case (term_variants_Fun T S f) thus ?case
    using ab list_all2_conv_all_nth[of "timpls_transformable_to TI" T S]
    by auto
qed simp

lemma timpls_transformable_to'_if_term_variants:
  fixes s t::"(('a, 'b, 'c, 'd) prot_fun, 'e) term" and a b::"'c set"
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes st: "term_variants_pred P s t"
    and ab: "(a,b) ∈ (set TI)⁺"
  shows "timpls_transformable_to' TI s t"
```

```
using st P_def
proof (induction rule: term_variants_pred.induct)
  case (term_variants_P T S f) thus ?case
    using ab list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S]
          in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    by auto
next
  case (term_variants_Fun T S f) thus ?case
    using ab list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S]
          in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    by auto
qed simp

lemma timpls_transformable_to_trans:
  assumes TI_trancl: "∀(a,b) ∈ (set TI)⁺. a ≠ b ⟶ (a,b) ∈ set TI"
    and st: "timpls_transformable_to TI s t"
    and tu: "timpls_transformable_to TI t u"
  shows "timpls_transformable_to TI s u"
using st tu
proof (induction s arbitrary: t u)
  case (Var x) thus ?case using tu timpls_transformable_to_inv_Var(1) by fast
next
  case (Fun f T)
  obtain g S where t:
      "t = Fun g S" "length T = length S"
      "⋀i. i < length T ⟹ timpls_transformable_to TI (T ! i) (S ! i)"
      "f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI"
    using timpls_transformable_to_inv'[OF Fun.prems(1)] TI_trancl by auto

  obtain h U where u:
      "u = Fun h U" "length S = length U"
      "⋀i. i < length S ⟹ timpls_transformable_to TI (S ! i) (U ! i)"
      "g ≠ h ⟹ is_Abs g ∧ is_Abs h ∧ (the_Abs g, the_Abs h) ∈ set TI"
    using timpls_transformable_to_inv'[OF Fun.prems(2)[unfolded t(1)]] TI_trancl by auto

  have "list_all2 (timpls_transformable_to TI) T U"
    using t(1,2,3) u(1,2,3) Fun.IH
          list_all2_conv_all_nth[of "timpls_transformable_to TI" T S]
          list_all2_conv_all_nth[of "timpls_transformable_to TI" S U]
          list_all2_conv_all_nth[of "timpls_transformable_to TI" T U]
    by force
  moreover have "(the_Abs f, the_Abs h) ∈ set TI"
    when "(the_Abs f, the_Abs g) ∈ set TI" "(the_Abs g, the_Abs h) ∈ set TI"
         "f ≠ h" "is_Abs f" "is_Abs h"
    using that(3,4,5) TI_trancl trancl_into_trancl[OF r_into_trancl[OF that(1)] that(2)]
    unfolding is_Abs_def the_Abs_def
    by force
  hence "is_Abs f ∧ is_Abs h ∧ (the_Abs f, the_Abs h) ∈ set TI"
    when "f ≠ h"
    using that TI_trancl t(4) u(4) by fast
  ultimately show ?case using t(1) u(1) by force
qed

lemma timpls_transformable_to'_trans:
  assumes st: "timpls_transformable_to' TI s t"
    and tu: "timpls_transformable_to' TI t u"
  shows "timpls_transformable_to' TI s u"
using st tu
proof (induction s arbitrary: t u)
  case (Var x) thus ?case using tu timpls_transformable_to_inv_Var(3) by fast
next
  case (Fun f T)
  note 0 = in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
```

```
  obtain g S where t:
      "t = Fun g S" "length T = length S"
      "⋀i. i < length T ⟹ timpls_transformable_to' TI (T ! i) (S ! i)"
      "f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ (set TI)⁺"
    using timpls_transformable_to'_inv'[OF Fun.prems(1)] 0 by auto

  obtain h U where u:
      "u = Fun h U" "length S = length U"
      "⋀i. i < length S ⟹ timpls_transformable_to' TI (S ! i) (U ! i)"
      "g ≠ h ⟹ is_Abs g ∧ is_Abs h ∧ (the_Abs g, the_Abs h) ∈ (set TI)⁺"
    using timpls_transformable_to'_inv'[OF Fun.prems(2)[unfolded t(1)]] 0 by auto

  have "list_all2 (timpls_transformable_to' TI) T U"
    using t(1,2,3) u(1,2,3) Fun.IH
          list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S]
          list_all2_conv_all_nth[of "timpls_transformable_to' TI" S U]
          list_all2_conv_all_nth[of "timpls_transformable_to' TI" T U]
    by force
  moreover have "(the_Abs f, the_Abs h) ∈ (set TI)⁺"
    when "(the_Abs f, the_Abs g) ∈ (set TI)⁺" "(the_Abs g, the_Abs h) ∈ (set TI)⁺"
    using that by simp
  hence "is_Abs f ∧ is_Abs h ∧ (the_Abs f, the_Abs h) ∈ (set TI)⁺"
    when "f ≠ h"
    by (metis that t(4) u(4))
  ultimately show ?case using t(1) u(1) 0 by force
qed

lemma timpls_transformable_to_mono:
  assumes "set TI ⊆ set TI'"
    and "timpls_transformable_to TI s t"
  shows "timpls_transformable_to TI' s t"
  using assms
proof (induction TI s t rule: timpls_transformable_to.induct)
  case (2 TI f T g S)
  have *: "f = g ∨ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)"
          "list_all2 (timpls_transformable_to TI) T S"
    using "2.prems" by simp_all

  show ?case
    using "2.IH" "2.prems"(1) list.rel_mono_strong[OF *(2)] *(1) in_trancl_mono[of TI TI']
    by (metis (no_types, lifting) timpls_transformable_to.simps(2) set_rev_mp)
qed auto

lemma timpls_transformable_to'_mono:
  assumes "set TI ⊆ set TI'"
    and "timpls_transformable_to' TI s t"
  shows "timpls_transformable_to' TI' s t"
  using assms
proof (induction TI s t rule: timpls_transformable_to'.induct)
  case (2 TI f T g S)
  have *: "f = g ∨ (is_Abs f ∧ is_Abs g ∧ in_trancl TI (the_Abs f) (the_Abs g))"
          "list_all2 (timpls_transformable_to' TI) T S"
    using "2.prems" by simp_all

  show ?case
    using "2.IH" "2.prems"(1) list.rel_mono_strong[OF *(2)] *(1) in_trancl_mono[of TI TI']
    by (metis (no_types, lifting) timpls_transformable_to'.simps(2))
qed auto

lemma timpls_transformable_to_refl_minus_eq:
  "timpls_transformable_to TI s t ⟷ timpls_transformable_to (filter (λ(a,b). a ≠ b) TI) s t"
  (is "?A ⟷ ?B")
```

```
proof
  let ?TI' = "λTI. filter (λ(a,b). a ≠ b) TI"

  show ?A when ?B using that timpls_transformable_to_mono[of "?TI' TI" TI] by auto

  show ?B when ?A using that
  proof (induction TI s t rule: timpls_transformable_to.induct)
    case (2 TI f T g S)
    have *: "f = g ∨ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)"
            "list_all2 (timpls_transformable_to TI) T S"
      using "2.prems" by simp_all

    have "f = g ∨ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set (?TI' TI))"
      using *(1) unfolding is_Abs_def by auto
    moreover have "list_all2 (timpls_transformable_to (?TI' TI)) T S"
      using *(2) "2.IH" list.rel_mono_strong by blast
    ultimately show ?case by force
  qed auto
qed

lemma timpls_transformable_to_iff_in_timpl_closure:
  assumes "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "timpls_transformable_to TI' s t ⟷ t ∈ timpl_closure s (set TI)" (is "?A s t ⟷ ?B s t")
proof
  show "?A s t ⟹ ?B s t" using assms
  proof (induction s t rule: timpls_transformable_to.induct)
    case (2 TI f T g S)
    note prems = "2.prems"
    note IH = "2.IH"

    have 1: "length T = length S" "∀i<length T. timpls_transformable_to TI' (T ! i) (S ! i)"
      using prems(1) list_all2_conv_all_nth[of "timpls_transformable_to TI'" T S] by simp_all

    note 2 = timpl_closure_is_timpl_closure'
    note 3 = in_set_conv_nth[of _ T] in_set_conv_nth[of _ S]

    have 4: "timpl_closure' (set TI') = timpl_closure' (set TI)"
      using timpl_closure'_timpls_trancl_eq'[of "set TI"] prems(2) by simp

    have IH': "(T ! i, S ! i) ∈ timpl_closure' (set TI')" when i: "i < length S" for i
    proof -
      have "timpls_transformable_to TI' (T ! i) (S ! i)" using i 1 by presburger
      hence "S ! i ∈ timpl_closure (T ! i) (set TI)"
        using IH[of "T ! i" "S ! i"] i 1(1) prems(2) by force
      thus ?thesis using 2[of "S ! i" "T ! i" "set TI"] 4 by blast
    qed

    have 5: "f = g ∨ (∃a b. (a, b) ∈ (set TI')⁺ ∧ f = Abs a ∧ g = Abs b)"
      using prems(1) the_Abs_def[of f] the_Abs_def[of g] is_Abs_def[of f] is_Abs_def[of g]
      by fastforce

    show ?case using 2 4 timpl_closure_FunI[OF IH' 1(1) 5] 1(1) by auto
  qed (simp_all add: timpl_closure.FP)

  show "?B s t ⟹ ?A s t"
  proof (induction t rule: timpl_closure.induct)
    case (TI u a b v) show ?case
    proof (cases "a = b")
      case True thus ?thesis using TI.hyps(3) TI.IH term_variants_pred_refl_inv by fastforce
    next
      case False
      hence 1: "timpls_transformable_to TI' u v"
        using TI.hyps(2) assms timpls_transformable_to_if_term_variants[OF TI.hyps(3), of TI']
```

```
        by blast
      have 2: "(c,d) ∈ set TI'" when cd: "(c,d) ∈ (set TI')⁺" "c ≠ d" for c d
      proof -
        let ?cl = "λX. {(a,b) ∈ X⁺. a ≠ b}"
        have "?cl (set TI') = ?cl (?cl (set TI))" using assms by presburger
        hence "set TI' = ?cl (set TI')" using assms trancl_minus_refl_idem[of "set TI"] by argo
        thus ?thesis using cd by blast
      qed
      show ?thesis using timpls_transformable_to_trans[OF _ TI.IH 1] 2 by blast
    qed
  qed (use timpls_transformable_to_refl in fast)
qed

lemma timpls_transformable_to'_iff_in_timpl_closure:
  "timpls_transformable_to' TI s t ⟷ t ∈ timpl_closure s (set TI)" (is "?A s t ⟷ ?B s t")
proof
  show "?A s t ⟹ ?B s t"
  proof (induction s t rule: timpls_transformable_to'.induct)
    case (2 TI f T g S)
    note prems = "2.prems"
    note IH = "2.IH"

    have 1: "length T = length S" "∀i<length T. timpls_transformable_to' TI (T ! i) (S ! i)"
      using prems list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S] by simp_all

    note 2 = timpl_closure_is_timpl_closure'
    note 3 = in_set_conv_nth[of _ T] in_set_conv_nth[of _ S]

    have IH': "(T ! i, S ! i) ∈ timpl_closure' (set TI)" when i: "i < length S" for i
    proof -
      have "timpls_transformable_to' TI (T ! i) (S ! i)" using i 1 by presburger
      hence "S ! i ∈ timpl_closure (T ! i) (set TI)" using IH[of "T ! i" "S ! i"] i 1(1) by force
      thus ?thesis using 2[of "S ! i" "T ! i" "set TI"] by blast
    qed

    have 4: "f = g ∨ (∃a b. (a, b) ∈ (set TI)⁺ ∧ f = Abs a ∧ g = Abs b)"
      using prems the_Abs_def[of f] the_Abs_def[of g] is_Abs_def[of f] is_Abs_def[of g]
            in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
      by auto

    show ?case using 2 timpl_closure_FunI[OF IH' 1(1) 4] 1(1) by auto
  qed (simp_all add: timpl_closure.FP)

  show "?B s t ⟹ ?A s t"
  proof (induction t rule: timpl_closure.induct)
    case (TI u a b v) thus ?case
      using timpls_transformable_to'_trans
            timpls_transformable_to'_if_term_variants
      by blast
  qed (use timpls_transformable_to_refl(2) in fast)
qed

lemma equal_mod_timpls_iff_ex_in_timpl_closure:
  assumes "set TI' = {(a,b) ∈ TI⁺. a ≠ b}"
  shows "equal_mod_timpls TI' s t ⟷ (∃u. u ∈ timpl_closure s TI ∧ u ∈ timpl_closure t TI)"
    (is "?A s t ⟷ ?B s t")
proof
  show "?A s t ⟹ ?B s t" using assms
  proof (induction s t rule: equal_mod_timpls.induct)
    case (2 TI' f T g S)
    note prems = "2.prems"
    note IH = "2.IH"
```

```
  have 1: "length T = length S" "∀i<length T. equal_mod_timpls (TI') (T ! i) (S ! i)"
    using prems list_all2_conv_all_nth[of "equal_mod_timpls TI'" T S] by simp_all

  note 2 = timpl_closure_is_timpl_closure'
  note 3 = in_set_conv_nth[of _ T] in_set_conv_nth[of _ S]

  have 4: "timpl_closure' (set TI') = timpl_closure' TI"
    using timpl_closure'_timpls_trancl_eq'[of TI] prems
    by simp

  have IH: "∃u. (T ! i, u) ∈ timpl_closure' TI ∧ (S ! i, u) ∈ timpl_closure' TI"
    when i: "i < length S" for i
  proof -
    have "equal_mod_timpls TI' (T ! i) (S ! i)" using i 1 by presburger
    hence "∃u. u ∈ timpl_closure (T ! i) TI ∧ u ∈ timpl_closure (S ! i) TI"
      using IH[of "T ! i" "S ! i"] i 1(1) prems by force
    thus ?thesis using 4 unfolding 2 by blast
  qed

  let ?P = "λG. f = g ∨ (∃a b. (a, b) ∈ G ∧ f = Abs a ∧ g = Abs b) ∨
                (∃a b. (a, b) ∈ G ∧ f = Abs b ∧ g = Abs a) ∨
                (∃a b c. (a, c) ∈ G ∧ (b, c) ∈ G ∧ f = Abs a ∧ g = Abs b)"

  have "?P (set TI')"
    using prems the_Abs_def[of f] the_Abs_def[of g] is_Abs_def[of f] is_Abs_def[of g]
    by fastforce
  hence "?P (TI⁺)" unfolding prems by blast
  hence "?P (rtrancl TI)" by (metis (no_types, lifting) trancl_into_rtrancl)
  hence 5: "f = g ∨ (∃a b c. (a, c) ∈ TI* ∧ (b, c) ∈ TI* ∧ f = Abs a ∧ g = Abs b)" by blast

  show ?case
    using timpl_closure_FunI3[OF _ 1(1) 5]  IH 1(1)
    unfolding timpl_closure'_timpls_rtrancl_eq 2
    by auto
qed (use timpl_closure.FP in auto)

show "?A s t" when B: "?B s t"
proof -
  obtain u where u: "u ∈ timpl_closure s TI" "u ∈ timpl_closure t TI"
    using B by blast
  thus ?thesis using assms
  proof (induction u arbitrary: s t rule: term.induct)
    case (Var x s t) thus ?case
      using timpl_closure_Var_in_iff[of x s TI]
            timpl_closure_Var_in_iff[of x t TI]
            equal_mod_timpls.simps(1)[of TI' x x]
      by blast
  next
    case (Fun f U s t)
    obtain g S where s:
        "s = Fun g S" "length U = length S"
        "⋀i. i < length U ⟹ U ! i ∈ timpl_closure (S ! i) TI"
        "g ≠ f ⟹ is_Abs g ∧ is_Abs f ∧ (the_Abs g, the_Abs f) ∈ TI⁺"
      using Fun.prems(1) timpl_closure_Fun_inv'[of f U _ _ TI]
      by (cases s) auto

    obtain h T where t:
        "t = Fun h T" "length U = length T"
        "⋀i. i < length U ⟹ U ! i ∈ timpl_closure (T ! i) TI"
        "h ≠ f ⟹ is_Abs h ∧ is_Abs f ∧ (the_Abs h, the_Abs f) ∈ TI⁺"
      using Fun.prems(2) timpl_closure_Fun_inv'[of f U _ _ TI]
      by (cases t) auto
```

```
      have g: "(the_Abs g, the_Abs f) ∈ set TI'" "is_Abs f" "is_Abs g" when neq_f: "g ≠ f"
      proof -
        obtain ga fa where a: "g = Abs ga" "f = Abs fa"
          using s(4)[OF neq_f] unfolding is_Abs_def by presburger
        hence "the_Abs g ≠ the_Abs f" using neq_f by simp
        thus "(the_Abs g, the_Abs f) ∈ set TI'" "is_Abs f" "is_Abs g"
          using s(4)[OF neq_f] Fun.prems by blast+
      qed

      have h: "(the_Abs h, the_Abs f) ∈ set TI'" "is_Abs f" "is_Abs h" when neq_f: "h ≠ f"
      proof -
        obtain ha fa where a: "h = Abs ha" "f = Abs fa"
          using t(4)[OF neq_f] unfolding is_Abs_def by presburger
        hence "the_Abs h ≠ the_Abs f" using neq_f by simp
        thus "(the_Abs h, the_Abs f) ∈ set TI'" "is_Abs f" "is_Abs h"
          using t(4)[OF neq_f] Fun.prems by blast+
      qed

      have "equal_mod_timpls TI' (S ! i) (T ! i)"
        when i: "i < length U" for i
        using i Fun.IH s(1,2,3) t(1,2,3) nth_mem[OF i] Fun.prems by meson
      hence "list_all2 (equal_mod_timpls TI') S T"
        using list_all2_conv_all_nth[of "equal_mod_timpls TI'" S T] s(2) t(2) by presburger
      thus ?case using s(1) t(1) g h by fastforce
    qed
  qed
qed



context
begin
private inductive timpls_transformable_to_pred where
  Var: "timpls_transformable_to_pred A (Var x) (Var x)"
| Fun: "⟦¬is_Abs f; length T = length S;
         ⋀i. i < length T ⟹ timpls_transformable_to_pred A (T ! i) (S ! i)⟧
        ⟹ timpls_transformable_to_pred A (Fun f T) (Fun f S)"
| Abs: "b ∈ A ⟹ timpls_transformable_to_pred A (Fun (Abs a) []) (Fun (Abs b) [])"

private lemma timpls_transformable_to_pred_inv_Var:
  assumes "timpls_transformable_to_pred A (Var x) t"
  shows "t = Var x"
using assms by (auto elim: timpls_transformable_to_pred.cases)

private lemma timpls_transformable_to_pred_inv:
  assumes "timpls_transformable_to_pred A (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ timpls_transformable_to_pred A (T ! i) (args t ! i)"
    and "¬is_Abs f ⟹ f = the_Fun t"
    and "is_Abs f ⟹ (is_Abs (the_Fun t) ∧ the_Abs (the_Fun t) ∈ A)"
using assms by (auto elim!: timpls_transformable_to_pred.cases[of A])

private lemma timpls_transformable_to_pred_finite_aux1:
  assumes f: "¬is_Abs f"
  shows "{s. timpls_transformable_to_pred A (Fun f T) s} ⊆
          (λS. Fun f S) ` {S. length T = length S ∧
                              (∀s ∈ set S. ∃t ∈ set T. timpls_transformable_to_pred A t s)}"
    (is "?B ⊆ ?C")
proof
  fix s assume s: "s ∈ ?B"
  hence *: "timpls_transformable_to_pred A (Fun f T) s" by blast
```

```
  obtain S where S:
      "s = Fun f S" "length T = length S" "⋀i. i < length T ⟹ timpls_transformable_to_pred A (T ! i) (S ! i)"
    using f timpls_transformable_to_pred_inv[OF *] unfolding the_Abs_def is_Abs_def by auto

  have "∀s∈set S. ∃t∈set T. timpls_transformable_to_pred A t s" using S(2,3) in_set_conv_nth by metis
  thus "s ∈ ?C" using S(1,2) by blast
qed

private lemma timpls_transformable_to_pred_finite_aux2:
  "{s. timpls_transformable_to_pred A (Fun (Abs a) []) s} ⊆ (λb. Fun (Abs b) []) ` A" (is "?B ⊆ ?C")
proof
  fix s assume s: "s ∈ ?B"
  hence *: "timpls_transformable_to_pred A (Fun (Abs a) []) s" by blast

  obtain b where b: "s = Fun (Abs b) []" "b ∈ A"
    using timpls_transformable_to_pred_inv[OF *] unfolding the_Abs_def is_Abs_def by auto
  thus "s ∈ ?C" by blast
qed

private lemma timpls_transformable_to_pred_finite:
  fixes t::"(('fun,'atom,'sets,'lbl) prot_fun, 'a) term"
  assumes A: "finite A"
    and t: "wf_trm t"
  shows "finite {s. timpls_transformable_to_pred A t s}"
using t
proof (induction t)
  case (Var x)
  have "{s::(('fun,'atom,'sets,'lbl) prot_fun, 'a) term. timpls_transformable_to_pred A (Var x) s} = {Var x}"
    by (auto intro: timpls_transformable_to_pred.Var elim: timpls_transformable_to_pred_inv_Var)
  thus ?case by simp
next
  case (Fun f T)
  have IH: "finite {s. timpls_transformable_to_pred A t s}" when t: "t ∈ set T" for t
    using Fun.IH[OF t] wf_trm_param[OF Fun.prems t] by blast

  show ?case
  proof (cases "is_Abs f")
    case True
    then obtain a where a: "f = Abs a" unfolding is_Abs_def by presburger
    hence "T = []" using wf_trm_arity[OF Fun.prems] by simp_all
    hence "{a. timpls_transformable_to_pred A (Fun f T) a} ⊆ (λb. Fun (Abs b) []) ` A"
      using timpls_transformable_to_pred_finite_aux2[of A a] a by auto
    thus ?thesis using A finite_subset by fast
  next
    case False thus ?thesis
      using IH finite_lists_length_eq' timpls_transformable_to_pred_finite_aux1[of f A T] finite_subset
      by blast
  qed
qed

private lemma timpls_transformable_to_pred_if_timpls_transformable_to:
  assumes s: "timpls_transformable_to TI t s"
    and t: "wf_trm t" "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A"
  shows "timpls_transformable_to_pred (A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺) t s"
using s t
proof (induction rule: timpls_transformable_to.induct)
  case (2 TI f T g S)
  let ?A = "A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"
```

```
  note prems = "2.prems"
  note IH = "2.IH"

  note 0 = timpls_transformable_to_inv[OF prems(1)]

  have 1: "T = []" "S = []" when f: "f = Abs a" for a
    using f wf_trm_arity[OF prems(2)] 0(1) by simp_all

  have "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A" when t: "t ∈ set T" for t
    using t prems(3) funs_term_subterms_eq(1)[of "Fun f T"] by blast
  hence 2: "timpls_transformable_to_pred ?A (T ! i) (S ! i)"
    when i: "i < length T" for i
    using i IH 0(1,2) wf_trm_param[OF prems(2)]
    by (metis (no_types) in_set_conv_nth)

  have 3: "the_Abs f ∈ ?A" when f: "is_Abs f" using prems(3) f by force

  show ?case
  proof (cases "f = g")
    case True
    note fg = True
    show ?thesis
    proof (cases "is_Abs f")
      case True
      then obtain a where a: "f = Abs a" unfolding is_Abs_def by blast
      thus ?thesis using fg 1[OF a] timpls_transformable_to_pred.Abs[of a ?A a] 3 by simp
    qed (use fg timpls_transformable_to_pred.Fun[OF _ 0(1) 2, of f] in blast)
  next
    case False
    then obtain a b where ab: "f = Abs a" "g = Abs b" "(a, b) ∈ (set TI)⁺"
      using 0(3) in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
      unfolding is_Abs_def the_Abs_def by fastforce
    hence "a ∈ ?A" "b ∈ ?A" by force+
    thus ?thesis using timpls_transformable_to_pred.Abs ab(1,2) 1[OF ab(1)] by metis
  qed
qed (simp_all add: timpls_transformable_to_pred.Var)

private lemma timpls_transformable_to_pred_if_timpls_transformable_to':
  assumes s: "timpls_transformable_to' TI t s"
    and t: "wf_trm t" "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A"
  shows "timpls_transformable_to_pred (A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺) t s"
using s t
proof (induction rule: timpls_transformable_to.induct)
  case (2 TI f T g S)
  let ?A = "A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"

  note prems = "2.prems"
  note IH = "2.IH"

  note 0 = timpls_transformable_to'_inv[OF prems(1)]

  have 1: "T = []" "S = []" when f: "f = Abs a" for a
    using f wf_trm_arity[OF prems(2)] 0(1) by simp_all

  have "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A" when t: "t ∈ set T" for t
    using t prems(3) funs_term_subterms_eq(1)[of "Fun f T"] by blast
  hence 2: "timpls_transformable_to_pred ?A (T ! i) (S ! i)"
    when i: "i < length T" for i
    using i IH 0(1,2) wf_trm_param[OF prems(2)]
    by (metis (no_types) in_set_conv_nth)

  have 3: "the_Abs f ∈ ?A" when f: "is_Abs f" using prems(3) f by force
```

```
  show ?case
  proof (cases "f = g")
    case True
    note fg = True
    show ?thesis
    proof (cases "is_Abs f")
      case True
      then obtain a where a: "f = Abs a" unfolding is_Abs_def by blast
      thus ?thesis using fg 1[OF a] timpls_transformable_to_pred.Abs[of a ?A a] 3 by simp
    qed (use fg timpls_transformable_to_pred.Fun[OF _ 0(1) 2, of f] in blast)
  next
    case False
    then obtain a b where ab: "f = Abs a" "g = Abs b" "(a, b) ∈ (set TI)⁺"
      using 0(3) in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
      unfolding is_Abs_def the_Abs_def by fastforce
    hence "a ∈ ?A" "b ∈ ?A" by force+
    thus ?thesis using timpls_transformable_to_pred.Abs ab(1,2) 1[OF ab(1)] by metis
  qed
qed (simp_all add: timpls_transformable_to_pred.Var)

private lemma timpls_transformable_to_pred_if_equal_mod_timpls:
  assumes s: "equal_mod_timpls TI t s"
    and t: "wf_trm t" "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A"
  shows "timpls_transformable_to_pred (A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺) t s"
using s t
proof (induction rule: equal_mod_timpls.induct)
  case (2 TI f T g S)
  let ?A = "A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"

  note prems = "2.prems"
  note IH = "2.IH"

  note 0 = equal_mod_timpls_inv[OF prems(1)]

  have 1: "T = []" "S = []" when f: "f = Abs a" for a
    using f wf_trm_arity[OF prems(2)] 0(1) by simp_all

  have "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A" when t: "t ∈ set T" for t
    using t prems(3) funs_term_subterms_eq(1)[of "Fun f T"] by blast
  hence 2: "timpls_transformable_to_pred ?A (T ! i) (S ! i)"
    when i: "i < length T" for i
    using i IH 0(1,2) wf_trm_param[OF prems(2)]
    by (metis (no_types) in_set_conv_nth)

  have 3: "the_Abs f ∈ ?A" when f: "is_Abs f" using prems(3) f by force

  show ?case
  proof (cases "f = g")
    case True
    note fg = True
    show ?thesis
    proof (cases "is_Abs f")
      case True
      then obtain a where a: "f = Abs a" unfolding is_Abs_def by blast
      thus ?thesis using fg 1[OF a] timpls_transformable_to_pred.Abs[of a ?A a] 3 by simp
    qed (use fg timpls_transformable_to_pred.Fun[OF _ 0(1) 2, of f] in blast)
  next
    case False
    then obtain a b where ab: "f = Abs a" "g = Abs b"
        "(a, b) ∈ (set TI)⁺ ∨ (b, a) ∈ (set TI)⁺ ∨
         (∃ti ∈ set TI. (a, snd ti) ∈ (set TI)⁺ ∧ (b, snd ti) ∈ (set TI)⁺)"
      using 0(3) in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
      unfolding is_Abs_def the_Abs_def by fastforce
```

```
    hence "a ∈ ?A" "b ∈ ?A" by force+
    thus ?thesis using timpls_transformable_to_pred.Abs ab(1,2) 1[OF ab(1)] by metis
  qed
qed (simp_all add: timpls_transformable_to_pred.Var)

lemma timpls_transformable_to_finite:
  assumes t: "wf_trm t"
  shows "finite {s. timpls_transformable_to TI t s}" (is ?P)
    and "finite {s. timpls_transformable_to' TI t s}" (is ?Q)
proof -
  let ?A = "the_Abs ` {f ∈ funs_term t. is_Abs f} ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"

  have 0: "finite ?A" by auto

  have 1: "{s. timpls_transformable_to TI t s} ⊆ {s. timpls_transformable_to_pred ?A t s}"
    using timpls_transformable_to_pred_if_timpls_transformable_to[OF _ t] by auto

  have 2: "{s. timpls_transformable_to' TI t s} ⊆ {s. timpls_transformable_to_pred ?A t s}"
    using timpls_transformable_to_pred_if_timpls_transformable_to'[OF _ t] by auto

  show ?P using timpls_transformable_to_pred_finite[OF 0 t] finite_subset[OF 1] by blast
  show ?Q using timpls_transformable_to_pred_finite[OF 0 t] finite_subset[OF 2] by blast
qed

lemma equal_mod_timpls_finite:
  assumes t: "wf_trm t"
  shows "finite {s. equal_mod_timpls TI t s}"
proof -
  let ?A = "the_Abs ` {f ∈ funs_term t. is_Abs f} ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"

  have 0: "finite ?A" by auto

  have 1: "{s. equal_mod_timpls TI t s} ⊆ {s. timpls_transformable_to_pred ?A t s}"
    using timpls_transformable_to_pred_if_equal_mod_timpls[OF _ t] by auto

  show ?thesis using timpls_transformable_to_pred_finite[OF 0 t] finite_subset[OF 1] by blast
qed

end

lemma intruder_synth_mod_timpls_is_synth_timpl_closure_set:
  fixes t::"(('fun,'atom,'sets,'lbl) prot_fun, 'a) term" and TI TI'
  assumes "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "intruder_synth_mod_timpls M TI' t ⟷ timpl_closure_set (set M) (set TI) ⊢c t"
      (is "?C t ⟷ ?D t")
proof -
  have *: "(∃m ∈ M. timpls_transformable_to TI' m t) ⟷ t ∈ timpl_closure_set M (set TI)"
    when "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    for M TI TI' and t::"(('fun,'atom,'sets,'lbl) prot_fun, 'a) term"
    using timpls_transformable_to_iff_in_timpl_closure[OF that]
          timpl_closure_set_is_timpl_closure_union[of M "set TI"]
          timpl_closure_set_timpls_trancl_eq[of M "set TI"]
          timpl_closure_set_timpls_trancl_eq'[of M "set TI"]
    by auto

  show "?C t ⟷ ?D t"
  proof
    show "?C t ⟹ ?D t" using assms
    proof (induction t arbitrary: M TI TI' rule: intruder_synth_mod_timpls.induct)
      case (1 M TI' x)
      hence "Var x ∈ timpl_closure_set (set M) (set TI)"
        using timpl_closure.FP member_def unfolding timpl_closure_set_def by force
      thus ?case by simp
```

```
    next
      case (2 M TI f T)
      show ?case
      proof (cases "∃m ∈ set M. timpls_transformable_to TI' m (Fun f T)")
        case True thus ?thesis
          using "2.prems" *[of TI' TI "set M" "Fun f T"]
                intruder_synth.AxiomC[of "Fun f T" "timpl_closure_set (set M) (set TI)"]
          by blast
      next
        case False
        hence "¬(list_ex (λt. timpls_transformable_to TI' t (Fun f T)) M)"
          unfolding list_ex_iff by blast
        hence "public f" "length T = arity f" "list_all (intruder_synth_mod_timpls M TI') T"
          using "2.prems"(1) by force+
        thus ?thesis using "2.IH"[OF _ _ "2.prems"(2)] unfolding list_all_iff by force
      qed
    qed

    show "?D t ⟹ ?C t"
    proof (induction t rule: intruder_synth_induct)
      case (AxiomC t) thus ?case
        using timpl_closure_set_Var_in_iff[of _ "set M" "set TI"] *[OF assms, of "set M" t]
        by (cases t rule: term.exhaust) (force simp add: member_def list_ex_iff)+
    next
      case (ComposeC T f) thus ?case
        using list_all_iff[of "intruder_synth_mod_timpls M TI'" T]
              intruder_synth_mod_timpls.simps(2)[of M TI' f T]
        by blast
    qed
  qed
qed

lemma intruder_synth_mod_timpls'_is_synth_timpl_closure_set:
  fixes t::"(('fun,'atom,'sets,'lbl) prot_fun, 'a) term" and TI
  shows "intruder_synth_mod_timpls' M TI t ⟷ timpl_closure_set (set M) (set TI) ⊢c t"
      (is "?A t ⟷ ?B t")
proof -
  have *: "(∃m ∈ M. timpls_transformable_to' TI m t) ⟷ t ∈ timpl_closure_set M (set TI)"
    for M TI and t::"(('fun,'atom,'sets,'lbl) prot_fun, 'a) term"
    using timpls_transformable_to'_iff_in_timpl_closure[of TI _ t]
          timpl_closure_set_is_timpl_closure_union[of M "set TI"]
    by blast+

  show "?A t ⟷ ?B t"
  proof
    show "?A t ⟹ ?B t"
    proof (induction t arbitrary: M TI rule: intruder_synth_mod_timpls'.induct)
      case (1 M TI x)
      hence "Var x ∈ timpl_closure_set (set M) (set TI)"
        using timpl_closure.FP List.member_def[of M] unfolding timpl_closure_set_def by auto
      thus ?case by simp
    next
      case (2 M TI f T)
      show ?case
      proof (cases "∃m ∈ set M. timpls_transformable_to' TI m (Fun f T)")
        case True thus ?thesis
          using "2.prems" *[of "set M" TI "Fun f T"]
                intruder_synth.AxiomC[of "Fun f T" "timpl_closure_set (set M) (set TI)"]
          by blast
      next
        case False
        hence "public f" "length T = arity f" "list_all (intruder_synth_mod_timpls' M TI) T"
          using "2.prems" list_ex_iff[of _ M] by force+
```

```
        thus ?thesis
          using "2.IH"[of _ M TI] list_all_iff[of "intruder_synth_mod_timpls' M TI" T]
          by force
      qed
    qed

    show "?B t ⟹ ?A t"
    proof (induction t rule: intruder_synth_induct)
      case (AxiomC t) thus ?case
        using AxiomC timpl_closure_set_Var_in_iff[of _ "set M" "set TI"] *[of "set M" TI t]
              list_ex_iff[of _ M] List.member_def[of M]
        by (cases t rule: term.exhaust) force+
    next
      case (ComposeC T f) thus ?case
        using list_all_iff[of "intruder_synth_mod_timpls' M TI" T]
              intruder_synth_mod_timpls'.simps(2)[of M TI f T]
        by blast
    qed
  qed
qed

lemma intruder_synth_mod_eq_timpls_is_synth_timpl_closure_set:
  fixes t::"(('fun,'atom,'sets,'lbl) prot_fun, 'a) term" and TI
  defines "cl ≡ λTI. {(a,b) ∈ TI⁺. a ≠ b}"
  shows  "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b} ⟹
         intruder_synth_mod_eq_timpls M TI' t ⟷
         (∃s ∈ timpl_closure t (set TI). timpl_closure_set M (set TI) ⊢c s)"
      (is "?Q TI TI' ⟹ ?C t ⟷ ?D t")
proof -


  have **: "(∃m ∈ M. equal_mod_timpls TI' m t) ⟷
            (∃s ∈ timpl_closure t (set TI). s ∈ timpl_closure_set M (set TI))"
    when Q: "?Q TI TI'"
    for M TI TI' and t::"(('fun,'atom,'sets,'lbl) prot_fun, 'a) term"
    using equal_mod_timpls_iff_ex_in_timpl_closure[OF Q]
          timpl_closure_set_is_timpl_closure_union[of M "set TI"]
          timpl_closure_set_timpls_trancl_eq'[of M "set TI"]
    by fastforce



  show "?C t ⟷ ?D t" when Q: "?Q TI TI'"
  proof
    show "?C t ⟹ ?D t" using Q
    proof (induction t arbitrary: M TI rule: intruder_synth_mod_eq_timpls.induct)
      case (1 M TI' x M TI)
      hence "Var x ∈ timpl_closure_set M (set TI)" "Var x ∈ timpl_closure (Var x) (set TI)"
        using timpl_closure.FP unfolding timpl_closure_set_def by auto
      thus ?case by force
    next
      case (2 M TI' f T M TI)
      show ?case
      proof (cases "∃m ∈ M. equal_mod_timpls TI' m (Fun f T)")
        case True thus ?thesis
          using **[OF "2.prems"(2), of M "Fun f T"]
                intruder_synth.AxiomC[of _ "timpl_closure_set M (set TI)"]
          by blast
      next
        case False
        hence f: "public f" "length T = arity f" "list_all (intruder_synth_mod_eq_timpls M TI') T"
          using "2.prems" by force+
```

```
      let ?sy = "intruder_synth (timpl_closure_set M (set TI))"

      have IH: "∃u ∈ timpl_closure (T ! i) (set TI). ?sy u"
        when i: "i < length T" for i
        using "2.IH"[of _ M TI] f(3) nth_mem[OF i] "2.prems"(2)
        unfolding list_all_iff by blast

      define S where "S ≡ map (λu. SOME v. v ∈ timpl_closure u (set TI) ∧ ?sy v) T"

      have S1: "length T = length S"
        unfolding S_def by simp

      have S2: "S ! i ∈ timpl_closure (T ! i) (set TI)"
               "timpl_closure_set M (set TI) ⊢c S ! i"
        when i: "i < length S" for i
        using i IH someI_ex[of "λv. v ∈ timpl_closure (T ! i) (set TI) ∧ ?sy v"]
        unfolding S_def by auto

      have "Fun f S ∈ timpl_closure (Fun f T) (set TI)"
        using timpl_closure_FunI[of T S "set TI" f f] S1 S2(1)
        unfolding timpl_closure_is_timpl_closure' by presburger
      thus ?thesis
        by (metis intruder_synth.ComposeC[of S f] f(1,2) S1 S2(2) in_set_conv_nth[of _ S])
    qed
  qed

  show "?C t" when D: "?D t"
  proof -
    obtain s where "timpl_closure_set M (set TI) ⊢c s" "s ∈ timpl_closure t (set TI)"
      using D by blast
    thus ?thesis
    proof (induction s arbitrary: t rule: intruder_synth_induct)
      case (AxiomC s t)
      note 1 = timpl_closure_set_Var_in_iff[of _ M "set TI"] timpl_closure_Var_inv[of s _ "set TI"]
      note 2 = **[OF Q, of M]
      show ?case
      proof (cases t)
        case Var thus ?thesis using 1 AxiomC by auto
      next
        case Fun thus ?thesis using 2 AxiomC by auto
      qed
    next
      case (ComposeC T f t)
      obtain g S where gS:
          "t = Fun g S" "length S = length T"
          "∀i < length T. T ! i ∈ timpl_closure (S ! i) (set TI)"
          "g ≠ f ⟹ is_Abs g ∧ is_Abs f ∧ (the_Abs g, the_Abs f) ∈ (set TI)⁺"
        using ComposeC.prems(1) timpl_closure'_inv'[of t "Fun f T" "set TI"]
              timpl_closure_is_timpl_closure'[of _ _ "set TI"]
        by fastforce

      have IH: "intruder_synth_mod_eq_timpls M TI' u" when u: "u ∈ set S" for u
        by (metis u gS(2,3) ComposeC.IH in_set_conv_nth)

      note 0 = list_all_iff[of "intruder_synth_mod_eq_timpls M TI'" S]
               intruder_synth_mod_eq_timpls.simps(2)[of M TI' g S]

      have "f = g" using ComposeC.hyps gS(4) unfolding is_Abs_def by fastforce
      thus ?case by (metis ComposeC.hyps(1,2) gS(1,2) IH 0)
    qed
  qed
 qed
qed
```

```
lemma timpl_closure_finite:
  assumes t: "wf_trm t"
  shows "finite (timpl_closure t (set TI))"
using timpls_transformable_to'_iff_in_timpl_closure[of TI t]
      timpls_transformable_to_finite[OF t, of TI]
by auto

lemma timpl_closure_set_finite:
  fixes TI::"('sets set × 'sets set) list"
  assumes M_finite: "finite M"
    and M_wf: "wf_trms M"
  shows "finite (timpl_closure_set M (set TI))"
using timpl_closure_set_is_timpl_closure_union[of M "set TI"]
      timpl_closure_finite[of _ TI] M_finite M_wf finite
by auto

lemma comp_timpl_closure_is_timpl_closure_set:
  fixes M and TI::"('sets set × 'sets set) list"
  assumes M_finite: "finite M"
    and M_wf: "wf_trms M"
  shows "comp_timpl_closure M (set TI) = timpl_closure_set M (set TI)"
using lfp_while''[OF timpls_Un_mono[of M "set TI"]]
      timpl_closure_set_finite[OF M_finite M_wf]
      timpl_closure_set_lfp[of M "set TI"]
unfolding comp_timpl_closure_def Let_def by presburger

context
begin

private lemma analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux1:
  fixes M::"('fun,'atom,'sets,'lbl) prot_terms"
  assumes f: "arity_f f = length T" "arity_f f > 0" "Ana_f f = (K, R)"
    and i: "i < length R"
    and M: "timpl_closure_set M TI ⊢_c T ! (R ! i)"
    and m: "Fun (Fu f) T ∈ M"
    and t: "Fun (Fu f) S ∈ timpl_closure (Fun (Fu f) T) TI"
  shows "timpl_closure_set M TI ⊢_c S ! (R ! i)"
proof -
  have "R ! i < length T" using i Ana_f_assm2_alt[OF f(3)] f(1) by simp
  thus ?thesis
    using timpl_closure_Fun_inv'(1,2)[OF t] intruder_synth_timpl_closure'[OF M]
    by presburger
qed

private lemma analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux2:
  fixes M::"('fun,'atom,'sets,'lbl) prot_terms"
  assumes M: "∀s ∈ set (snd (Ana m)). timpl_closure_set M TI ⊢_c s"
    and m: "m ∈ M"
    and t: "t ∈ timpl_closure m TI"
    and s: "s ∈ set (snd (Ana t))"
  shows "timpl_closure_set M TI ⊢_c s"
proof -
  obtain f S K N where fS: "t = Fun (Fu f) S" "arity_f f = length S" "0 < arity_f f"
      and Ana_f: "Ana_f f = (K, N)"
      and Ana_t: "Ana t = (K ·_list (!) S, map ((!) S) N)"
    using Ana_nonempty_inv[of t] s by fastforce
  then obtain T where T: "m = Fun (Fu f) T" "length T = length S"
    using t timpl_closure_Fu_inv'[of f S m TI]
    by blast
  hence Ana_m: "Ana m = (K ·_list (!) T, map ((!) T) N)"
    using fS(2,3) Ana_f by auto
```

```
  obtain i where i: "i < length N" "s = S ! (N ! i)"
    using s[unfolded fS(1)] Ana_t[unfolded fS(1)] T(2)
          in_set_conv_nth[of s "map (λi. S ! i) N"]
    by auto
  hence "timpl_closure_set M TI ⊢c T ! (N ! i)"
    using M[unfolded T(1)] Ana_m[unfolded T(1)] T(2)
    by simp
  thus ?thesis
    using analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux1[
            OF fS(2)[unfolded T(2)[symmetric]] fS(3) Ana_f
               i(1) _ m[unfolded T(1)] t[unfolded fS(1) T(1)]]
          i(2)
    by argo
qed

lemma analyzed_closed_mod_timpls_is_analyzed_timpl_closure_set:
  fixes M::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes TI': "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and M_wf: "wf_trms (set M)"
  shows "analyzed_closed_mod_timpls M TI' ⟷ analyzed (timpl_closure_set (set M) (set TI))"
    (is "?A ⟷ ?B")
proof
  let ?C = "∀t ∈ timpl_closure_set (set M) (set TI).
              analyzed_in t (timpl_closure_set (set M) (set TI))"

  let ?P = "λT. ∀t ∈ set T. timpl_closure_set (set M) (set TI) ⊢c t"
  let ?Q = "λt. ∀s ∈ comp_timpl_closure {t} (set TI'). case Ana s of (K, R) ⇒ ?P K ⟶ ?P R"
  let ?W = "λt. ∀t ∈ set (fst (Ana t)). ∀f ∈ funs_term t. ¬is_Abs f"
  let ?V = "λt. ∀s ∈ comp_timpl_closure (set (fst (Ana t))) (set TI').
                  ¬timpl_closure_set (set M) (set TI) ⊢c s"

  note defs = analyzed_closed_mod_timpls_def analyzed_in_code
  note 0 = intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI', of M]
  note 1 = timpl_closure_set_is_timpl_closure_union[of _ "set TI"]

  have 2: "comp_timpl_closure N (set TI') = timpl_closure_set N (set TI)"
    when "wf_trms N" "finite N" for N::"('fun,'atom,'sets,'lbl) prot_terms"
    using that timpl_closure_set_timpls_trancl_eq'[of N "set TI"]
          comp_timpl_closure_is_timpl_closure_set[of N TI']
    unfolding TI'[symmetric] by presburger
  hence 3: "comp_timpl_closure {t} (set TI') ⊆ timpl_closure_set (set M) (set TI)"
    when t: "t ∈ set M" "wf_trm t" for t
    using t timpl_closure_set_mono[of "{t}" "set M"] by simp

  have ?A when C: ?C
    unfolding analyzed_closed_mod_timpls_def
              intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI']
              list_all_iff Let_def
  proof (intro ballI)
    fix t assume t: "t ∈ set M"
    show "if ?P (fst (Ana t)) then ?P (snd (Ana t))
          else if ?W t then True
          else if ?V t then True
          else ?Q t" (is ?R)
    proof (cases "?P (fst (Ana t))")
      case True
      hence "?P (snd (Ana t))"
        using C timpl_closure_setI[OF t, of "set TI"] prod.exhaust_sel
        unfolding analyzed_in_def by blast
      thus ?thesis using True by simp
    next
      case False
      have "?Q t" using 3[OF t] C M_wf t unfolding analyzed_in_def by auto
```

```
      thus ?thesis using False by argo
    qed
  qed
  thus ?A when B: ?B using B analyzed_is_all_analyzed_in by metis

  have ?C when A: ?A unfolding analyzed_in_def Let_def
  proof (intro ballI allI impI; elim conjE)
    fix t K T s
    assume t: "t ∈ timpl_closure_set (set M) (set TI)"
      and s: "s ∈ set T"
      and Ana_t: "Ana t = (K, T)"
      and K: "∀k ∈ set K. timpl_closure_set (set M) (set TI) ⊢c k"

    obtain m where m: "m ∈ set M" "t ∈ timpl_closure m (set TI)"
      using timpl_closure_set_is_timpl_closure_union t by blast

    show "timpl_closure_set (set M) (set TI) ⊢c s"
    proof (cases "∀k ∈ set (fst (Ana m)). timpl_closure_set (set M) (set TI) ⊢c k")
      case True
      hence *: "∀r ∈ set (snd (Ana m)). timpl_closure_set (set M) (set TI) ⊢c r"
        using m(1) A
        unfolding analyzed_closed_mod_timpls_def
                  intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI']
                  list_all_iff Let_def
        by simp

      show ?thesis
        using K s Ana_t A
              analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux2[OF * m]
        by simp
    next
      case False
      note F = this

      have *: "comp_timpl_closure {m} (set TI') = timpl_closure m (set TI)"
        using 2[of "{m}"] timpl_closureton_is_timpl_closure M_wf m(1)
        by blast

      have "wf_trms (set (fst (Ana m)))"
        using Ana_keys_wf'[of m "fst (Ana m)"] M_wf m(1) surj_pair[of "Ana m"] by fastforce
      hence **: "comp_timpl_closure (set (fst (Ana m))) (set TI') =
                 timpl_closure_set (set (fst (Ana m))) (set TI)"
        using 2[of "set (fst (Ana m))"] by blast

      have ***: "set K ⊆ timpl_closure_set (set (fst (Ana m))) (set TI)"
                "length K = length (fst (Ana m))"
        using timpl_closure_Ana_keys_subset[OF m(2) _ Ana_t]
              timpl_closure_Ana_keys_length_eq[OF m(2) _ Ana_t]
              surj_pair[of "Ana m"]
        by fastforce+

      show ?thesis
      proof (cases "?W m")
        case True
        hence "fst (Ana t) = fst (Ana m)" using m timpl_closure_Ana_keys_no_Abs_eq_case by fast
        thus ?thesis using F K Ana_t by simp
      next
        case False
        note F' = this

        show ?thesis
        proof (cases "?V m")
          case True
```

```
        hence "∀k ∈ set K. ¬timpl_closure_set (set M) (set TI) ⊢c k"
          using F K Ana_t m s *** unfolding ** by blast
        thus ?thesis using K F' *** by simp
      next
        case False
        hence "?Q m"
          using m(1) A F F'
          unfolding analyzed_closed_mod_timpls_def
                    intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI']
                    list_all_iff Let_def
          by auto
        thus ?thesis
          using * m(2) K s Ana_t
          unfolding Let_def by auto
      qed
    qed
  qed
 qed
 thus ?B when A: ?A using A analyzed_is_all_analyzed_in by metis
qed

lemma analyzed_closed_mod_timpls'_is_analyzed_timpl_closure_set:
 fixes M::"('fun,'atom,'sets,'lbl) prot_term list"
 assumes M_wf: "wftrms (set M)"
 shows "analyzed_closed_mod_timpls' M TI ⟷ analyzed (timpl_closure_set (set M) (set TI))"
   (is "?A ⟷ ?B")
proof
 let ?C = "∀t ∈ timpl_closure_set (set M) (set TI). analyzed_in t (timpl_closure_set (set M) (set
TI))"

 let ?P = "λT. ∀t ∈ set T. timpl_closure_set (set M) (set TI) ⊢c t"
 let ?Q = "λt. ∀s ∈ comp_timpl_closure {t} (set TI). case Ana s of (K, R) ⇒ ?P K ⟶ ?P R"
 let ?W = "λt. ∀t ∈ set (fst (Ana t)). ∀f ∈ funs_term t. ¬is_Abs f"

 note defs = analyzed_closed_mod_timpls'_def analyzed_in_code
 note 0 = intruder_synth_mod_timpls'_is_synth_timpl_closure_set[of M TI]
 note 1 = timpl_closure_set_is_timpl_closure_union[of _ "set TI"]

 have 2: "comp_timpl_closure {t} (set TI) = timpl_closure_set {t} (set TI)"
   when t: "t ∈ set M" "wftrm t" for t
   using t timpl_closure_set_timpls_trancl_eq[of "{t}" "set TI"]
         comp_timpl_closure_is_timpl_closure_set[of "{t}"]
   by blast
 hence 3: "comp_timpl_closure {t} (set TI) ⊆ timpl_closure_set (set M) (set TI)"
   when t: "t ∈ set M" "wftrm t" for t
   using t timpl_closure_set_mono[of "{t}" "set M"]
   by fast

 have ?A when C: ?C
   unfolding analyzed_closed_mod_timpls'_def
             intruder_synth_mod_timpls'_is_synth_timpl_closure_set
             list_all_iff Let_def
 proof (intro ballI)
   fix t assume t: "t ∈ set M"
   show "if ?P (fst (Ana t)) then ?P (snd (Ana t)) else if ?W t then True else ?Q t" (is ?R)
   proof (cases "?P (fst (Ana t))")
     case True
     hence "?P (snd (Ana t))"
       using C timpl_closure_setI[OF t, of "set TI"] prod.exhaust_sel
       unfolding analyzed_in_def by blast
     thus ?thesis using True by simp
   next
     case False
```

```
      have "?Q t" using 3[OF t] C M_wf t unfolding analyzed_in_def by auto
      thus ?thesis using False by argo
    qed
  qed
  thus ?A when B: ?B using B analyzed_is_all_analyzed_in by metis

  have ?C when A: ?A unfolding analyzed_in_def Let_def
  proof (intro ballI allI impI; elim conjE)
    fix t K T s
    assume t: "t ∈ timpl_closure_set (set M) (set TI)"
      and s: "s ∈ set T"
      and Ana_t: "Ana t = (K, T)"
      and K: "∀k ∈ set K. timpl_closure_set (set M) (set TI) ⊢c k"

    obtain m where m: "m ∈ set M" "t ∈ timpl_closure m (set TI)"
      using timpl_closure_set_is_timpl_closure_union t by blast

    show "timpl_closure_set (set M) (set TI) ⊢c s"
    proof (cases "∀k ∈ set (fst (Ana m)). timpl_closure_set (set M) (set TI) ⊢c k")
      case True
      hence *: "∀r ∈ set (snd (Ana m)). timpl_closure_set (set M) (set TI) ⊢c r"
        using m(1) A
        unfolding analyzed_closed_mod_timpls'_def
                  intruder_synth_mod_timpls'_is_synth_timpl_closure_set
                  list_all_iff
        by simp

      show ?thesis
        using K s Ana_t A
              analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux2[OF * m]
        by simp
    next
      case False
      note F = this

      have *: "comp_timpl_closure {m} (set TI) = timpl_closure m (set TI)"
        using 2[OF m(1)] timpl_closureton_is_timpl_closure M_wf m(1)
        by blast

      show ?thesis
      proof (cases "?W m")
        case True
        hence "fst (Ana t) = fst (Ana m)" using m timpl_closure_Ana_keys_no_Abs_eq_case by fast
        thus ?thesis using F K Ana_t by simp
      next
        case False
        hence "?Q m"
          using m(1) A F
          unfolding analyzed_closed_mod_timpls'_def
                    intruder_synth_mod_timpls'_is_synth_timpl_closure_set
                    list_all_iff Let_def
          by auto
        thus ?thesis
          using * m(2) K s Ana_t
          unfolding Let_def by auto
      qed
    qed
  qed
  thus ?B when A: ?A using A analyzed_is_all_analyzed_in by metis
qed

end
```

```
end

end
```

3.6 Stateful Protocol Verification

```
theory Stateful_Protocol_Verification
imports Stateful_Protocol_Model Term_Implication
begin
```

3.6.1 Fixed-Point Intruder Deduction Lemma

```
context stateful_protocol_model
begin

abbreviation pubval_terms::"('fun,'atom,'sets,'lbl) prot_terms" where
  "pubval_terms ≡ {t. ∃f ∈ funs_term t. is_PubConstValue f}"

abbreviation abs_terms::"('fun,'atom,'sets,'lbl) prot_terms" where
  "abs_terms ≡ {t. ∃f ∈ funs_term t. is_Abs f}"

definition intruder_deduct_GSMP::
  "[('fun,'atom,'sets,'lbl) prot_terms,
    ('fun,'atom,'sets,'lbl) prot_terms,
    ('fun,'atom,'sets,'lbl) prot_term]
    ⇒ bool" (‹⟨_;_⟩ ⊢GSMP _› 50)
where
  "⟨M; T⟩ ⊢GSMP t ≡ intruder_deduct_restricted M (λt. t ∈ GSMP T - (pubval_terms ∪ abs_terms)) t"

lemma intruder_deduct_GSMP_induct[consumes 1, case_names AxiomH ComposeH DecomposeH]:
  assumes "⟨M; T⟩ ⊢GSMP t" "⋀t. t ∈ M ⟹ P M t"
          "⋀S f. ⟦length S = arity f; public f;
                  ⋀s. s ∈ set S ⟹ ⟨M; T⟩ ⊢GSMP s;
                  ⋀s. s ∈ set S ⟹ P M s;
                  Fun f S ∈ GSMP T - (pubval_terms ∪ abs_terms)
                  ⟧ ⟹ P M (Fun f S)"
          "⋀t K T' tᵢ. ⟦⟨M; T⟩ ⊢GSMP t; P M t; Ana t = (K, T'); ⋀k. k ∈ set K ⟹ ⟨M; T⟩ ⊢GSMP k;
                        ⋀k. k ∈ set K ⟹ P M k; tᵢ ∈ set T'⟧ ⟹ P M tᵢ"
  shows "P M t"
proof -
  let ?Q = "λt. t ∈ GSMP T - (pubval_terms ∪ abs_terms)"
  show ?thesis
    using intruder_deduct_restricted_induct[of M ?Q t "λM Q t. P M t"] assms
    unfolding intruder_deduct_GSMP_def
    by blast
qed

lemma pubval_terms_subst:
  assumes "t · ϑ ∈ pubval_terms" "ϑ ` fv t ∩ pubval_terms = {}"
  shows "t ∈ pubval_terms"
using assms(1,2)
proof (induction t)
  case (Fun f T)
  let ?P = "λf. is_PubConstValue f"
  from Fun show ?case
  proof (cases "?P f")
    case False
    then obtain t where t: "t ∈ set T" "t · ϑ ∈ pubval_terms"
      using Fun.prems by auto
    hence "ϑ ` fv t ∩ pubval_terms = {}" using Fun.prems(2) by auto
    thus ?thesis using Fun.IH[OF t] t(1) by auto
  qed force
```

```
qed simp

lemma abs_terms_subst:
  assumes "t · ϑ ∈ abs_terms" "ϑ ` fv t ∩ abs_terms = {}"
  shows "t ∈ abs_terms"
using assms(1,2)
proof (induction t)
  case (Fun f T)
  let ?P = "λf. is_Abs f"
  from Fun show ?case
  proof (cases "?P f")
    case False
    then obtain t where t: "t ∈ set T" "t · ϑ ∈ abs_terms"
      using Fun.prems by auto
    hence "ϑ ` fv t ∩ abs_terms = {}" using Fun.prems(2) by auto
    thus ?thesis using Fun.IH[OF t] t(1) by auto
  qed force
qed simp

lemma pubval_terms_subst':
  assumes "t · ϑ ∈ pubval_terms" "∀n. PubConst Value n ∉ ⋃(funs_term ` (ϑ ` fv t))"
  shows "t ∈ pubval_terms"
proof -
  have False
    when fs: "f ∈ funs_term s" "s ∈ subterms_set (ϑ ` fv t)" "is_PubConstValue f"
    for f s
  proof -
    obtain T where T: "Fun f T ∈ subterms s" using funs_term_Fun_subterm[OF fs(1)] by force
    hence "Fun f T ∈ subterms_set (ϑ ` fv t)" using fs(2) in_subterms_subset_Union by blast
    thus ?thesis
      using assms(2) funs_term_Fun_subterm'[of f T] fs(3)
      unfolding is_PubConstValue_def
      by (cases f) force+
  qed
  thus ?thesis using pubval_terms_subst[OF assms(1)] by auto
qed

lemma abs_terms_subst':
  assumes "t · ϑ ∈ abs_terms" "∀n. Abs n ∉ ⋃(funs_term ` (ϑ ` fv t))"
  shows "t ∈ abs_terms"
proof -
  have "¬is_Abs f" when fs: "f ∈ funs_term s" "s ∈ subterms_set (ϑ ` fv t)" for f s
  proof -
    obtain T where T: "Fun f T ∈ subterms s" using funs_term_Fun_subterm[OF fs(1)] by force
    hence "Fun f T ∈ subterms_set (ϑ ` fv t)" using fs(2) in_subterms_subset_Union by blast
    thus ?thesis using assms(2) funs_term_Fun_subterm'[of f T] by (cases f) auto
  qed
  thus ?thesis using abs_terms_subst[OF assms(1)] by force
qed

lemma pubval_terms_subst_range_disj:
  "subst_range ϑ ∩ pubval_terms = {} ⟹ ϑ ` fv t ∩ pubval_terms = {}"
proof (induction t)
  case (Var x) thus ?case by (cases "x ∈ subst_domain ϑ") auto
qed auto

lemma abs_terms_subst_range_disj:
  "subst_range ϑ ∩ abs_terms = {} ⟹ ϑ ` fv t ∩ abs_terms = {}"
proof (induction t)
  case (Var x) thus ?case by (cases "x ∈ subst_domain ϑ") auto
qed auto

lemma pubval_terms_subst_range_comp:
```

```
  assumes "subst_range ϑ ∩ pubval_terms = {}" "subst_range δ ∩ pubval_terms = {}"
  shows "subst_range (ϑ ∘s δ) ∩ pubval_terms = {}"
proof -
  { fix t f assume t:
      "t ∈ subst_range (ϑ ∘s δ)" "f ∈ funs_term t" "is_PubConstValue f"
    then obtain x where x: "(ϑ ∘s δ) x = t" by auto
    have "ϑ x ∉ pubval_terms" using assms(1) by (cases "ϑ x ∈ subst_range ϑ") force+
    hence "(ϑ ∘s δ) x ∉ pubval_terms"
      using assms(2) pubval_terms_subst[of "ϑ x" δ] pubval_terms_subst_range_disj
      by (metis (mono_tags, lifting) subst_compose_def)
    hence False using t(2,3) x by blast
  } thus ?thesis by fast
qed

lemma pubval_terms_subst_range_comp':
  assumes "(ϑ ` X) ∩ pubval_terms = {}" "(δ ` fvset (ϑ ` X)) ∩ pubval_terms = {}"
  shows "((ϑ ∘s δ) ` X) ∩ pubval_terms = {}"
proof -
  { fix t f assume t:
      "t ∈ (ϑ ∘s δ) ` X" "f ∈ funs_term t" "is_PubConstValue f"
    then obtain x where x: "(ϑ ∘s δ) x = t" "x ∈ X" by auto
    have "ϑ x ∉ pubval_terms" using assms(1) x(2) by force
    moreover have "fv (ϑ x) ⊆ fvset (ϑ ` X)" using x(2) by (auto simp add: fv_subset)
    hence "δ ` fv (ϑ x) ∩ pubval_terms = {}" using assms(2) by auto
    ultimately have "(ϑ ∘s δ) x ∉ pubval_terms"
      using pubval_terms_subst[of "ϑ x" δ]
      by (metis (mono_tags, lifting) subst_compose_def)
    hence False using t(2,3) x by blast
  } thus ?thesis by fast
qed

lemma abs_terms_subst_range_comp:
  assumes "subst_range ϑ ∩ abs_terms = {}" "subst_range δ ∩ abs_terms = {}"
  shows "subst_range (ϑ ∘s δ) ∩ abs_terms = {}"
proof -
  { fix t f assume t: "t ∈ subst_range (ϑ ∘s δ)" "f ∈ funs_term t" "is_Abs f"
    then obtain x where x: "(ϑ ∘s δ) x = t" by auto
    have "ϑ x ∉ abs_terms" using assms(1) by (cases "ϑ x ∈ subst_range ϑ") force+
    hence "(ϑ ∘s δ) x ∉ abs_terms"
      using assms(2) abs_terms_subst[of "ϑ x" δ] abs_terms_subst_range_disj
      by (metis (mono_tags, lifting) subst_compose_def)
    hence False using t(2,3) x by blast
  } thus ?thesis by fast
qed

lemma abs_terms_subst_range_comp':
  assumes "(ϑ ` X) ∩ abs_terms = {}" "(δ ` fvset (ϑ ` X)) ∩ abs_terms = {}"
  shows "((ϑ ∘s δ) ` X) ∩ abs_terms = {}"
proof -
  { fix t f assume t:
      "t ∈ (ϑ ∘s δ) ` X" "f ∈ funs_term t" "is_Abs f"
    then obtain x where x: "(ϑ ∘s δ) x = t" "x ∈ X" by auto
    have "ϑ x ∉ abs_terms" using assms(1) x(2) by force
    moreover have "fv (ϑ x) ⊆ fvset (ϑ ` X)" using x(2) by (auto simp add: fv_subset)
    hence "δ ` fv (ϑ x) ∩ abs_terms = {}" using assms(2) by auto
    ultimately have "(ϑ ∘s δ) x ∉ abs_terms"
      using abs_terms_subst[of "ϑ x" δ]
      by (metis (mono_tags, lifting) subst_compose_def)
    hence False using t(2,3) x by blast
  } thus ?thesis by fast
qed

context
```

```
begin
private lemma Ana_abs_aux1:
  fixes δ::"(('fun,'atom,'sets,'lbl) prot_fun, nat, ('fun,'atom,'sets,'lbl) prot_var) gsubst"
    and α::"nat ⇒ 'sets set"
  assumes "Ana_f f = (K,T)"
  shows "(K ·_list δ) ·_αlist α = K ·_list (λn. δ n ·_α α)"
proof -
  { fix k assume "k ∈ set K"
    hence "k ∈ subterms_set (set K)" by force
    hence "k · δ ·_α α = k · (λn. δ n ·_α α)"
    proof (induction k)
      case (Fun g S)
      have "⋀s. s ∈ set S ⟹ s · δ ·_α α = s · (λn. δ n ·_α α)"
        using Fun.IH in_subterms_subset_Union[OF Fun.prems] Fun_param_in_subterms[of _ S g]
        by (meson contra_subsetD)
      thus ?case using Ana_f_assm1_alt[OF assms Fun.prems] by (cases g) auto
    qed simp
  } thus ?thesis unfolding abs_apply_list_def by force
qed

private lemma Ana_abs_aux2:
  fixes α::"nat ⇒ 'sets set"
    and K::"(('fun,'atom,'sets,'lbl) prot_fun, nat) term list"
    and M::"nat list"
    and T::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes "∀i ∈ fv_set (set K) ∪ set M. i < length T"
    and "(K ·_list (!) T) ·_αlist α = K ·_list (λn. T ! n ·_α α)"
  shows "(K ·_list (!) T) ·_αlist α = K ·_list (!) (map (λs. s ·_α α) T)" (is "?A1 = ?A2")
    and "(map ((!) T) M) ·_αlist α = map ((!) (map (λs. s ·_α α) T)) M" (is "?B1 = ?B2")
proof -
  have "T ! i ·_α α = (map (λs. s ·_α α) T) ! i" when "i ∈ fv_set (set K)" for i
    using that assms(1) by auto
  hence "k · (λi. T ! i ·_α α) = k · (λi. (map (λs. s ·_α α) T) ! i)" when "k ∈ set K" for k
    using that term_subst_eq_conv[of k "λi. T ! i ·_α α" "λi. (map (λs. s ·_α α) T) ! i"]
    by auto
  thus "?A1 = ?A2" using assms(2) by (force simp add: abs_apply_terms_def)

  have "T ! i ·_α α = map (λs. s ·_α α) T ! i" when "i ∈ set M" for i
    using that assms(1) by auto
  thus "?B1 = ?B2" by (force simp add: abs_apply_list_def)
qed

private lemma Ana_abs_aux1_set:
  fixes δ::"(('fun,'atom,'sets,'lbl) prot_fun, nat, ('fun,'atom,'sets,'lbl) prot_var) gsubst"
    and α::"nat ⇒ 'sets set"
  assumes "Ana_f f = (K,T)"
  shows "(set K ·_set δ) ·_αset α = set K ·_set (λn. δ n ·_α α)"
proof -
  { fix k assume "k ∈ set K"
    hence "k ∈ subterms_set (set K)" by force
    hence "k · δ ·_α α = k · (λn. δ n ·_α α)"
    proof (induction k)
      case (Fun g S)
      have "⋀s. s ∈ set S ⟹ s · δ ·_α α = s · (λn. δ n ·_α α)"
        using Fun.IH in_subterms_subset_Union[OF Fun.prems] Fun_param_in_subterms[of _ S g]
        by (meson contra_subsetD)
      thus ?case using Ana_f_assm1_alt[OF assms Fun.prems] by (cases g) auto
    qed simp
  } thus ?thesis unfolding abs_apply_terms_def by force
qed

private lemma Ana_abs_aux2_set:
  fixes α::"nat ⇒ 'sets set"
```

```
    and K::"(('fun,'atom,'sets,'lbl) prot_fun, nat) terms"
    and M::"nat set"
    and T::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes "∀i ∈ fv_set K ∪ M. i < length T"
    and "(K ·_set (!) T) ·_αset α = K ·_set (λn. T ! n ·_α α)"
  shows "(K ·_set (!) T) ·_αset α = K ·_set (!) (map (λs. s ·_α α) T)" (is "?A1 = ?A2")
    and "((!) T ` M) ·_αset α = (!) (map (λs. s ·_α α) T) ` M" (is "?B1 = ?B2")
proof -
  have "T ! i ·_α α = (map (λs. s ·_α α) T) ! i" when "i ∈ fv_set K" for i
    using that assms(1) by auto
  hence "k · (λi. T ! i ·_α α) = k · (λi. (map (λs. s ·_α α) T) ! i)" when "k ∈ K" for k
    using that term_subst_eq_conv[of k "λi. T ! i ·_α α" "λi. (map (λs. s ·_α α) T) ! i"]
    by auto
  thus "?A1 = ?A2" using assms(2) by (force simp add: abs_apply_terms_def)

  have "T ! i ·_α α = map (λs. s ·_α α) T ! i" when "i ∈ M" for i
    using that assms(1) by auto
  thus "?B1 = ?B2" by (force simp add: abs_apply_terms_def)
qed

lemma Ana_abs:
  fixes t::"('fun,'atom,'sets,'lbl) prot_term"
  assumes "Ana t = (K, T)"
  shows "Ana (t ·_α α) = (K ·_αlist α, T ·_αlist α)"
  using assms
proof (induction t rule: Ana.induct)
  case (1 f S)
  obtain K' T' where *: "Ana_f f = (K',T')" by force
  show ?case using 1
  proof (cases "arity_f f = length S ∧ arity_f f > 0")
    case True
    hence "K = K' ·_list (!) S" "T = map ((!) S) T'"
        and **: "arity_f f = length (map (λs. s ·_α α) S)" "arity_f f > 0"
      using 1 * by auto
    hence "K ·_αlist α = K' ·_list (!) (map (λs. s ·_α α) S)"
          "T ·_αlist α = map ((!) (map (λs. s ·_α α) S)) T'"
      using Ana_f_assm2_alt[OF *] Ana_abs_aux2[OF _ Ana_abs_aux1[OF *], of T' S α]
      unfolding abs_apply_list_def
      by auto
    moreover have "Fun (Fu f) S ·_α α = Fun (Fu f) (map (λs. s ·_α α) S)" by simp
    ultimately show ?thesis using Ana_Fu_intro[OF ** *] by metis
  qed (auto simp add: abs_apply_list_def)
qed (simp_all add: abs_apply_list_def)
end

lemma deduct_FP_if_deduct:
  fixes M IK FP::"('fun,'atom,'sets,'lbl) prot_terms"
  assumes IK: "IK ⊆ GSMP M - (pubval_terms ∪ abs_terms)" "∀t ∈ IK ·_αset α. FP ⊢_c t"
    and t: "IK ⊢ t" "t ∈ GSMP M - (pubval_terms ∪ abs_terms)"
  shows "FP ⊢ t ·_α α"
proof -
  let ?P = "λf. ¬is_PubConstValue f"
  let ?GSMP = "GSMP M - (pubval_terms ∪ abs_terms)"

  have 1: "∀m ∈ IK. m ∈ ?GSMP"
    using IK(1) by blast

  have 2: "∀t t'. t ∈ ?GSMP ⟶ t' ⊑ t ⟶ t' ∈ ?GSMP"
  proof (intro allI impI)
    fix t t' assume t: "t ∈ ?GSMP" "t' ⊑ t"
    hence "t' ∈ GSMP M" using ground_subterm unfolding GSMP_def by auto
    moreover have "¬is_PubConstValue f"
      when "f ∈ funs_term t" for f
```

```
      using t(1) that by auto
    hence "¬is_PubConstValue f"
      when "f ∈ funs_term t'" for f
      using that subtermeq_imp_funs_term_subset[OF t(2)] by auto
    moreover have "¬is_Abs f" when "f ∈ funs_term t" for f using t(1) that by auto
    hence "¬is_Abs f" when "f ∈ funs_term t'" for f
      using that subtermeq_imp_funs_term_subset[OF t(2)] by auto
    ultimately show "t' ∈ ?GSMP" by simp
  qed

  have 3: "∀t K T k. t ∈ ?GSMP ⟶ Ana t = (K, T) ⟶ k ∈ set K ⟶ k ∈ ?GSMP"
  proof (intro allI impI)
    fix t K T k assume t: "t ∈ ?GSMP" "Ana t = (K, T)" "k ∈ set K"
    hence "k ∈ GSMP M" using GSMP_Ana_key by blast
    moreover have "∀f ∈ funs_term t. ?P f" using t(1) by auto
    with t(2,3) have "∀f ∈ funs_term k. ?P f"
    proof (induction t arbitrary: k rule: Ana.induct)
      case 1 thus ?case by (metis Ana_Fu_keys_not_pubval_terms surj_pair)
    qed auto
    moreover have "∀f ∈ funs_term t. ¬is_Abs f" using t(1) by auto
    with t(2,3) have "∀f ∈ funs_term k. ¬is_Abs f"
    proof (induction t arbitrary: k rule: Ana.induct)
      case 1 thus ?case by (metis Ana_Fu_keys_not_abs_terms surj_pair)
    qed auto
    ultimately show "k ∈ ?GSMP" by simp
  qed

  have "⟨IK; M⟩ ⊢_GSMP t"
    unfolding intruder_deduct_GSMP_def
    by (rule restricted_deduct_if_deduct'[OF 1 2 3 t])
  thus ?thesis
  proof (induction t rule: intruder_deduct_GSMP_induct)
    case (AxiomH t)
    show ?case using IK(2) abs_in[OF AxiomH.hyps] by force
  next
    case (ComposeH T f)
    have *: "Fun f T ·α α = Fun f (map (λt. t ·α α) T)"
      using ComposeH.hyps(2,4)
      by (cases f) auto

    have **: "length (map (λt. t ·α α) T) = arity f"
      using ComposeH.hyps(1)
      by auto

    show ?case
      using intruder_deduct.Compose[OF ** ComposeH.hyps(2)] ComposeH.IH(1) *
      by auto
  next
    case (DecomposeH t K T' t_i)
    have *: "Ana (t ·α α) = (K ·αlist α, T' ·αlist α)"
      using Ana_abs[OF DecomposeH.hyps(2)]
      by metis

    have **: "t_i ·α α ∈ set (T' ·αlist α)"
      using DecomposeH.hyps(4) abs_in abs_list_set_is_set_abs_set[of T']
      by auto

    have ***: "FP ⊢ k"
      when k: "k ∈ set (K ·αlist α)" for k
    proof -
      obtain k' where k': "k' ∈ set K" "k = k' ·α α"
        by (metis (no_types) k abs_apply_terms_def imageE abs_list_set_is_set_abs_set)
```

```
      show "FP ⊢ k"
        using DecomposeH.IH k' by blast
    qed

    show ?case
      using intruder_deduct.Decompose[OF _ * _ **]
            DecomposeH.IH(1) ***(1)
      by blast
  qed
qed

end
```

3.6.2 Computing and Checking Term Implications and Messages

```
context stateful_protocol_model
begin

abbreviation (input) "absc s ≡ (Fun (Abs s) []::('fun,'atom,'sets,'lbl) prot_term)"

fun absdbupd where
  "absdbupd [] _ a = a"
| "absdbupd (insert⟨Var y, Fun (Set s) T⟩#D) x a = (
    if x = y then absdbupd D x (insert s a) else absdbupd D x a)"
| "absdbupd (delete⟨Var y, Fun (Set s) T⟩#D) x a = (
    if x = y then absdbupd D x (a - {s}) else absdbupd D x a)"
| "absdbupd (_#D) x a = absdbupd D x a"

lemma absdbupd_cons_cases:
  "absdbupd (insert⟨Var x, Fun (Set s) T⟩#D) x d = absdbupd D x (insert s d)"
  "absdbupd (delete⟨Var x, Fun (Set s) T⟩#D) x d = absdbupd D x (d - {s})"
  "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T) ⟹ absdbupd (insert⟨t,u⟩#D) x d = absdbupd D x d"
  "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T) ⟹ absdbupd (delete⟨t,u⟩#D) x d = absdbupd D x d"
proof -
  assume *: "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T)"
  let ?P = "absdbupd (insert⟨t,u⟩#D) x d = absdbupd D x d"
  let ?Q = "absdbupd (delete⟨t,u⟩#D) x d = absdbupd D x d"
  { fix y f T assume "t = Fun f T ∨ u = Var y" hence ?P ?Q by auto
  } moreover {
    fix y f T assume "t = Var y" "u = Fun f T" hence ?P using * by (cases f) auto
  } moreover {
    fix y f T assume "t = Var y" "u = Fun f T" hence ?Q using * by (cases f) auto
  } ultimately show ?P ?Q by (metis term.exhaust)+
qed simp_all

lemma absdbupd_filter: "absdbupd S x d = absdbupd (filter is_Update S) x d"
by (induction S x d rule: absdbupd.induct) simp_all

lemma absdbupd_append:
  "absdbupd (A@B) x d = absdbupd B x (absdbupd A x d)"
proof (induction A arbitrary: d)
  case (Cons a A) thus ?case
  proof (cases a)
    case (Insert t u) thus ?thesis
    proof (cases "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T)")
      case False
      then obtain s T where "t = Var x" "u = Fun (Set s) T" by force
      thus ?thesis by (simp add: Insert Cons.IH absdbupd_cons_cases(1))
    qed (simp_all add: Cons.IH absdbupd_cons_cases(3))
  next
    case (Delete t u) thus ?thesis
    proof (cases "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T)")
      case False
```

```
      then obtain s T where "t = Var x" "u = Fun (Set s) T" by force
      thus ?thesis by (simp add: Delete Cons.IH absdbupd_cons_cases(2))
    qed (simp_all add: Cons.IH absdbupd_cons_cases(4))
  qed simp_all
qed simp

lemma absdbupd_wellformed_transaction:
  assumes T: "wellformed_transaction T"
  shows "absdbupd (unlabel (transaction_strand T)) = absdbupd (unlabel (transaction_updates T))"
proof -
  define S0 where "S0 ≡ unlabel (transaction_strand T)"
  define S1 where "S1 ≡ unlabel (transaction_receive T)"
  define S2 where "S2 ≡ unlabel (transaction_checks T)"
  define S3 where "S3 ≡ unlabel (transaction_updates T)"
  define S4 where "S4 ≡ unlabel (transaction_send T)"

  note S_defs = S0_def S1_def S2_def S3_def S4_def

  have 0: "list_all is_Receive S1"
          "list_all is_Check_or_Assignment S2"
          "list_all is_Update S3"
          "list_all is_Send S4"
    using T unfolding wellformed_transaction_def S_defs by metis+

  have "filter is_Update S1 = []"
       "filter is_Update S2 = []"
       "filter is_Update S3 = S3"
       "filter is_Update S4 = []"
    using list_all_filter_nil[OF 0(1), of is_Update]
          list_all_filter_nil[OF 0(2), of is_Update]
          list_all_filter_eq[OF 0(3)]
          list_all_filter_nil[OF 0(4), of is_Update]
    by blast+
  moreover have "S0 = S1@S2@S3@S4"
    unfolding S_defs transaction_strand_def unlabel_def by auto
  ultimately have "filter is_Update S0 = S3"
    using filter_append[of is_Update] list_all_append[of is_Update]
    by simp
  thus ?thesis
    using absdbupd_filter[of S0]
    unfolding S_defs by presburger
qed

fun abs_substs_set::
  "[('fun,'atom,'sets,'lbl) prot_var list,
    'sets set list,
    ('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set,
    ('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set,
    ('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set ⇒ bool]
  ⇒ ((('fun,'atom,'sets,'lbl) prot_var × 'sets set) list) list"
where
  "abs_substs_set [] _ _ _ _ = [[]]"
| "abs_substs_set (x#xs) as posconstrs negconstrs msgconstrs = (
    let bs = filter (λa. posconstrs x ⊆ a ∧ a ∩ negconstrs x = {} ∧ msgconstrs x a) as;
        Δ = abs_substs_set xs as posconstrs negconstrs msgconstrs
    in concat (map (λb. map (λδ. (x, b)#δ) Δ) bs))"

definition abs_substs_fun::
  "[(('fun,'atom,'sets,'lbl) prot_var × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_var]
  ⇒ 'sets set"
where
  "abs_substs_fun δ x = (case find (λb. fst b = x) δ of Some (_,a) ⇒ a | None ⇒ {})"
```

```
lemmas abs_substs_set_induct = abs_substs_set.induct[case_names Nil Cons]

fun transaction_poschecks_comp::
  "(('fun,'atom,'sets,'lbl) prot_fun, ('fun,'atom,'sets,'lbl) prot_var) stateful_strand
  ⇒ (('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set)"
where
  "transaction_poschecks_comp [] = (λ_. {})"
| "transaction_poschecks_comp (⟨_: Var x ∈ Fun (Set s) []⟩#T) = (
    let f = transaction_poschecks_comp T in f(x := insert s (f x)))"
| "transaction_poschecks_comp (_#T) = transaction_poschecks_comp T"

fun transaction_negchecks_comp::
  "(('fun,'atom,'sets,'lbl) prot_fun, ('fun,'atom,'sets,'lbl) prot_var) stateful_strand
  ⇒ (('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set)"
where
  "transaction_negchecks_comp [] = (λ_. {})"
| "transaction_negchecks_comp (⟨Var x not in Fun (Set s) []⟩#T) = (
    let f = transaction_negchecks_comp T in f(x := insert s (f x)))"
| "transaction_negchecks_comp (_#T) = transaction_negchecks_comp T"

definition transaction_check_pre where
  "transaction_check_pre FPT T δ ≡
    let (FP, _, TI) = FPT;
        C = set (unlabel (transaction_checks T));
        xs = fv_list_sst (unlabel (transaction_strand T));
        ϑ = λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x
    in (∀x ∈ set (transaction_fresh T). δ x = {}) ∧
       (∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)) ∧
       (∀u ∈ C.
          (is_InSet u ⟶ (
            let x = the_elem_term u; s = the_set_term u
            in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∈ δ (the_Var x))) ∧
          ((is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1) ⟶ (
            let x = fst (hd (the_ins u)); s = snd (hd (the_ins u))
            in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∉ δ (the_Var x))))"

definition transaction_check_post where
  "transaction_check_post FPT T δ ≡
    let (FP, _, TI) = FPT;
        xs = fv_list_sst (unlabel (transaction_strand T));
        ϑ = λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x;
        u = λδ x. absdbupd (unlabel (transaction_updates T)) x (δ x)
    in (∀x ∈ set xs - set (transaction_fresh T). δ x ≠ u δ x ⟶ List.member TI (δ x, u δ x)) ∧
       (∀t ∈ trms_lsst (transaction_send T). intruder_synth_mod_timpls FP TI (t · ϑ (u δ)))"

definition fun_point_inter where "fun_point_inter f g ≡ λx. f x ∩ g x"
definition fun_point_union where "fun_point_union f g ≡ λx. f x ∪ g x"
definition fun_point_Inter where "fun_point_Inter fs ≡ λx. ⋂f ∈ fs. f x"
definition fun_point_Union where "fun_point_Union fs ≡ λx. ⋃f ∈ fs. f x"
definition fun_point_Inter_list where "fun_point_Inter_list fs ≡ λx. ⋂(set (map (λf. f x) fs))"
definition fun_point_Union_list where "fun_point_Union_list fs ≡ λx. ⋃(set (map (λf. f x) fs))"
definition ticl_abs where "ticl_abs TI a ≡ set (a#map snd (filter (λp. fst p = a) TI))"
definition ticl_abss where "ticl_abss TI as ≡ ⋃a ∈ as. ticl_abs TI a"

lemma fun_point_Inter_set_eq:
  "fun_point_Inter (set fs) = fun_point_Inter_list fs"
unfolding fun_point_Inter_def fun_point_Inter_list_def by simp

lemma fun_point_Union_set_eq:
  "fun_point_Union (set fs) = fun_point_Union_list fs"
unfolding fun_point_Union_def fun_point_Union_list_def by simp
```

```
lemma ticl_abs_refl_in: "x ∈ ticl_abs TI x"
unfolding ticl_abs_def by simp

lemma ticl_abs_iff:
  assumes TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "ticl_abs TI a = {b. (a,b) ∈ (set TI)*}"
proof (intro order_antisym subsetI)
  fix x assume x: "x ∈ {b. (a, b) ∈ (set TI)*}"
  hence "x = a ∨ (x ≠ a ∧ (a,x) ∈ (set TI)⁺)" by (metis mem_Collect_eq rtranclD)
  moreover have "ticl_abs TI a = {a} ∪ {b. (a,b) ∈ set TI}" unfolding ticl_abs_def by force
  ultimately show "x ∈ ticl_abs TI a" using TI by blast
qed (fastforce simp add: ticl_abs_def)

lemma ticl_abs_Inter:
  assumes xs: "⋂(ticl_abs TI ` xs) ≠ {}"
    and TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "⋂(ticl_abs TI ` ⋂(ticl_abs TI ` xs)) ⊆ ⋂(ticl_abs TI ` xs)"
proof
  fix x assume x: "x ∈ ⋂(ticl_abs TI ` ⋂(ticl_abs TI ` xs))"
  have *: "⋂(ticl_abs TI ` xs) = {b. ∀a ∈ xs. (a,b) ∈ (set TI)*}"
    unfolding ticl_abs_iff[OF TI] by blast

  have "(b,x) ∈ (set TI)*" when b: "∀a ∈ xs. (a,b) ∈ (set TI)*" for b
    using x b unfolding ticl_abs_iff[OF TI] by blast
  hence "(a,x) ∈ (set TI)*" when "a ∈ xs" for a
    using that xs rtrancl.rtrancl_into_rtrancl[of a _ "(set TI)*" x]
    unfolding * rtrancl_idemp[of "set TI"] by blast
  thus "x ∈ ⋂(ticl_abs TI ` xs)" unfolding * by blast
qed

function (sequential) match_abss'
::"(('a,'b,'c,'d) prot_fun, 'e) term ⇒
   (('a,'b,'c,'d) prot_fun, 'e) term ⇒
   ('e ⇒ 'c set set) option"
where
  "match_abss' (Var x) (Fun (Abs a) _) = Some ((λ_. {})(x := {a}))"
| "match_abss' (Fun f ts) (Fun g ss) = (
    if f = g ∧ length ts = length ss
    then map_option fun_point_Union_list (those (map2 match_abss' ts ss))
    else None)"
| "match_abss' _ _ = None"
by pat_completeness auto
termination
proof -
  let ?m = "measures [size ∘ fst]"

  have 0: "wf ?m" by simp

  show ?thesis
    apply (standard, use 0 in fast)
    by (metis (no_types) comp_def fst_conv measures_less Fun_zip_size_lt(1))
qed

definition match_abss where
  "match_abss OCC TI t s ≡ (
    let xs = fv t;
        OCC' = set OCC;
        f = λδ x. if x ∈ xs then δ x else OCC';
        g = λδ x. ⋂(ticl_abs TI ` δ x)
    in case match_abss' t s of
      Some δ ⇒
        let δ' = g δ
        in if ∀x ∈ xs. δ' x ≠ {} then Some (f δ') else None
```

```
    | None ⇒ None)"

lemma match_abss'_Var_inv:
  assumes δ: "match_abss' (Var x) t = Some δ"
  shows "∃a ts. t = Fun (Abs a) ts ∧ δ = (λ_. {})(x := {a})"
proof -
  obtain f ts where t: "t = Fun f ts" using δ by (cases t) auto
  then obtain a where a: "f = Abs a" using δ by (cases f) auto
  show ?thesis using δ unfolding t a by simp
qed

lemma match_abss'_Fun_inv:
  assumes "match_abss' (Fun f ts) (Fun g ss) = Some δ"
  shows "f = g" (is ?A)
    and "length ts = length ss" (is ?B)
    and "∃ϑ. Some ϑ = those (map2 match_abss' ts ss) ∧ δ = fun_point_Union_list ϑ" (is ?C)
    and "∀(t,s) ∈ set (zip ts ss). ∃σ. match_abss' t s = Some σ" (is ?D)
proof -
  note 0 = assms match_abss'.simps(2)[of f ts g ss] option.distinct(1)
  show ?A by (metis 0)
  show ?B by (metis 0)
  show ?C by (metis (no_types, opaque_lifting) 0 map_option_eq_Some)
  thus ?D using map2_those_Some_case[of match_abss' ts ss] by fastforce
qed

lemma match_abss'_FunI:
  assumes Δ: "⋀i. i < length T ⟹ match_abss' (U ! i) (T ! i) = Some (Δ i)"
    and T: "length T = length U"
  shows "match_abss' (Fun f U) (Fun f T) = Some (fun_point_Union_list (map Δ [0..<length T]))"
proof -
  have "match_abss' (Fun f U) (Fun f T) =
          map_option fun_point_Union_list (those (map2 match_abss' U T))"
    using T match_abss'.simps(2)[of f U f T] by presburger
  moreover have "those (map2 match_abss' U T) = Some (map Δ [0..<length T])"
    using Δ T those_map2_SomeI by metis
  ultimately show ?thesis by simp
qed

lemma match_abss'_Fun_param_subset:
  assumes "match_abss' (Fun f ts) (Fun g ss)  = Some δ"
    and "(t,s) ∈ set (zip ts ss)"
    and "match_abss' t s = Some σ"
  shows "σ x ⊆ δ x"
proof -
  obtain ϑ where ϑ:
      "those (map2 match_abss' ts ss) = Some ϑ"
      "δ = fun_point_Union_list ϑ"
    using match_abss'_Fun_inv[OF assms(1)] by metis

  have "σ ∈ set ϑ" using ϑ(1) assms(2-) those_Some_iff[of "map2 match_abss' ts ss" ϑ] by force
  thus ?thesis using ϑ(2) unfolding fun_point_Union_list_def by auto
qed

lemma match_abss'_fv_is_nonempty:
  assumes "match_abss' t s = Some δ"
    and "x ∈ fv t"
  shows "δ x ≠ {}" (is "?P δ")
using assms
proof (induction t s arbitrary: δ rule: match_abss'.induct)
  case (2 f ts g ss)
  note prems = "2.prems"
  note IH = "2.IH"
```

```
  have 0: "∀(t,s) ∈ set (zip ts ss). ∃σ. match_abss' t s = Some σ" "f = g" "length ts = length ss"
    using match_abss'_Fun_inv[OF prems(1)] by simp_all

  obtain t where t: "t ∈ set ts" "x ∈ fv t" using prems(2) by auto
  then obtain s where s: "s ∈ set ss" "(t,s) ∈ set (zip ts ss)"
    by (meson 0(3) in_set_impl_in_set_zip1 in_set_zipE)
  then obtain σ where σ: "match_abss' t s = Some σ" using 0(1) by fast

  show ?case
    using IH[OF conjI[OF 0(2,3)] s(2) _ σ] t(2) match_abss'_Fun_param_subset[OF prems(1) s(2) σ]
    by auto
qed auto

lemma match_abss'_nonempty_is_fv:
  fixes s t::"(('a,'b,'c,'d) prot_fun, 'v) term"
  assumes "match_abss' s t = Some δ"
    and "δ x ≠ {}"
  shows "x ∈ fv s"
using assms
proof (induction s t arbitrary: δ rule: match_abss'.induct)
  case (2 f ts g ss)
  note prems = "2.prems"
  note IH = "2.IH"

  obtain ϑ where ϑ: "Some ϑ = those (map2 match_abss' ts ss)" "δ = fun_point_Union_list ϑ"
      and fg: "f = g" "length ts = length ss"
    using match_abss'_Fun_inv[OF prems(1)] by fast

  have "∃σ ∈ set ϑ. σ x ≠ {}"
    using fg(2) prems ϑ unfolding fun_point_Union_list_def by auto
  then obtain t' s' σ where ts':
      "(t',s') ∈ set (zip ts ss)" "match_abss' t' s' = Some σ" "σ x ≠ {}"
    using those_map2_SomeD[OF ϑ(1)[symmetric]] by blast

  show ?case
    using ts'(3) IH[OF conjI[OF fg] ts'(1) _ ts'(2)] set_zip_leftD[OF ts'(1)] by force
qed auto

lemma match_abss'_Abs_in_funs_term:
  fixes s t::"(('a,'b,'c,'d) prot_fun, 'v) term"
  assumes "match_abss' s t = Some δ"
    and "a ∈ δ x"
  shows "Abs a ∈ funs_term t"
using assms
proof (induction s t arbitrary: a δ rule: match_abss'.induct)
  case (1 y b ts) show ?case
    using match_abss'_Var_inv[OF "1.prems"(1)]  "1.prems"(2)
    by (cases "x = y") simp_all
next
  case (2 f ts g ss)
  note prems = "2.prems"
  note IH = "2.IH"

  obtain ϑ where ϑ: "Some ϑ = those (map2 match_abss' ts ss)" "δ = fun_point_Union_list ϑ"
      and fg: "f = g" "length ts = length ss"
    using match_abss'_Fun_inv[OF prems(1)] by fast

  obtain t' s' σ where ts': "(t',s') ∈ set (zip ts ss)" "match_abss' t' s' = Some σ" "a ∈ σ x"
    using fg(2) prems ϑ those_map2_SomeD[OF ϑ(1)[symmetric]]
    unfolding fun_point_Union_list_def by fastforce

  show ?case
    using ts'(1) IH[OF conjI[OF fg] ts'(1) _ ts'(2,3)]
```

```
    by (meson set_zip_rightD term.set_intros(2))
qed auto

lemma match_abss'_subst_fv_ex_abs:
  assumes "match_abss' s (s · δ) = Some σ"
    and TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "∀x ∈ fv s. ∃a ts. δ x = Fun (Abs a) ts ∧ σ x = {a}" (is "?P s σ")
using assms(1)
proof (induction s "s · δ" arbitrary: σ rule: match_abss'.induct)
  case (2 f ts g ss)
  note prems = "2.prems"
  note hyps = "2.hyps"

  obtain ϑ where ϑ: "Some ϑ = those (map2 match_abss' ts ss)" "σ = fun_point_Union_list ϑ"
      and fg: "f = g" "length ts = length ss" "ss = ts ·list δ"
      and ts: "∀(t,s) ∈ set (zip ts ss). ∃σ. match_abss' t s = Some σ"
    using match_abss'_Fun_inv[OF prems(1)[unfolded hyps(2)[symmetric]]] hyps(2) by fastforce

  have 0: "those (map (λt. match_abss' t (t · δ)) ts) = Some ϑ"
    using ϑ(1) map2_map_subst unfolding fg(3) by metis

  have 1: "∀t ∈ set ts. ∃σ. match_abss' t (t · δ) = Some σ"
    using ts zip_map_subst[of ts δ] unfolding fg(3) by simp

  have 2: "σ' ∈ set ϑ"
    when t: "t ∈ set ts" "match_abss' t (t · δ) = Some σ'" for t σ'
    using t 0 those_Some_iff[of "map (λt. match_abss' t (t · δ)) ts" ϑ] by force

  have 3: "?P t σ'" "σ' x ≠ {}"
    when t: "t ∈ set ts" "x ∈ fv t" "match_abss' t (t · δ) = Some σ'" for t σ' x
    using t hyps(1)[OF conjI[OF fg(1,2)], of "(t, t · δ)" t σ'] zip_map_subst[of ts δ]
          match_abss'_fv_is_nonempty[of t "t · δ" σ' x]
    unfolding fg(3) by auto

  have 4: "σ' x = {}"
    when t: "x ∉ fv t" "match_abss' t (t · δ) = Some σ'" for t σ' x
    by (meson t match_abss'_nonempty_is_fv)

  show ?case
  proof
    fix x assume "x ∈ fv (Fun f ts)"
    then obtain t σ' where t: "t ∈ set ts" "x ∈ fv t" and σ': "match_abss' t (t · δ) = Some σ'"
      using 1 by auto
    then obtain a tsa where a: "δ x = Fun (Abs a) tsa"
      using 3[OF t σ'] by fast

    have "σ'' x = {a} ∨ σ'' x = {}"
      when "σ'' ∈ set ϑ" for σ''
      using that a 0 3[of _ x] 4[of x]
      unfolding those_Some_iff by fastforce
    thus "∃a ts. δ x = Fun (Abs a) ts ∧ σ x = {a}"
      using a 2[OF t(1) σ'] 3[OF t σ'] unfolding ϑ(2) fun_point_Union_list_def by auto
  qed
qed auto

lemma match_abss'_subst_disj_nonempty:
  assumes TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and "match_abss' s (s · δ) = Some σ"
    and "x ∈ fv s"
  shows "⋂(ticl_abs TI ` σ x) ≠ {} ∧ (∃a tsa. δ x = Fun (Abs a) tsa ∧ σ x = {a})" (is "?P σ")
using assms(2,3)
proof (induction s "s · δ" arbitrary: σ rule: match_abss'.induct)
  case (1 x a ts) thus ?case unfolding ticl_abs_def by force
```

```
next
  case (2 f ts g ss)
  note prems = "2.prems"
  note hyps = "2.hyps"

  obtain ϑ where ϑ: "Some ϑ = those (map2 match_abss' ts ss)" "σ = fun_point_Union_list ϑ"
      and fg: "f = g" "length ts = length ss" "ss = ts ·list δ"
      and ts: "∀(t,s) ∈ set (zip ts ss). ∃σ. match_abss' t s = Some σ"
    using match_abss'_Fun_inv[OF prems(1)[unfolded hyps(2)[symmetric]]] hyps(2) by fastforce

  define ts' where "ts' ≡ filter (λt. x ∈ fv t) ts"
  define ϑ' where "ϑ' ≡ map (λt. (t, the (match_abss' t (t · δ)))) ts"
  define ϑ'' where "ϑ'' ≡ map (λt. the (match_abss' t (t · δ))) ts'"

  have 0: "those (map (λt. match_abss' t (t · δ)) ts) = Some ϑ"
    using ϑ(1) map2_map_subst unfolding fg(3) by metis

  have 1: "∀t ∈ set ts. ∃σ. match_abss' t (t · δ) = Some σ"
    using ts zip_map_subst[of ts δ] unfolding fg(3) by simp

  have ts_not_nil: "ts ≠ []"
    using prems(2) by fastforce
  hence "∃t ∈ set ts. x ∈ fv t" using prems(2) by simp
  then obtain a tsa where a: "δ x = Fun (Abs a) tsa"
    using 1 match_abss'_subst_fv_ex_abs[OF _ TI, of _ δ]
    by metis
  hence a': "σ' x = {a}"
    when "t ∈ set ts" "x ∈ fv t" "match_abss' t (t · δ) = Some σ'"
    for t σ'
    using that match_abss'_subst_fv_ex_abs[OF _ TI, of _ δ]
    by fastforce

  have "ts' ≠ []" using prems(2) unfolding ts'_def by (simp add: filter_empty_conv)
  hence ϑ''_not_nil: "ϑ'' ≠ []" unfolding ϑ''_def by simp

  have 2: "σ' ∈ set ϑ"
    when t: "t ∈ set ts" "match_abss' t (t · δ) = Some σ'" for t σ'
    using t 0 those_Some_iff[of "map (λt. match_abss' t (t · δ)) ts" ϑ] by force

  have 3: "?P σ'" "σ' x ≠ {}"
    when t: "t ∈ set ts'" "match_abss' t (t · δ) = Some σ'" for t σ'
    using t hyps(1)[OF conjI[OF fg(1,2)], of "(t, t · δ)" t σ'] zip_map_subst[of ts δ]
          match_abss'_fv_is_nonempty[of t "t · δ" σ' x]
    unfolding fg(3) ts'_def by (force, force)

  have 4: "σ' x = {}"
    when t: "x ∉ fv t" "match_abss' t (t · δ) = Some σ'" for t σ'
    by (meson t match_abss'_nonempty_is_fv)

  have 5: "ϑ = map snd ϑ'"
    using 0 1 unfolding ϑ'_def by (induct ts arbitrary: ϑ) auto

  have "fun_point_Union_list (map snd ϑ') x =
        fun_point_Union_list (map snd (filter (λ(t,_). x ∈ fv t) ϑ')) x"
    using 1 4 unfolding ϑ'_def fun_point_Union_list_def by fastforce
  hence 6: "fun_point_Union_list ϑ x = fun_point_Union_list ϑ'' x"
    using 0 1 4 unfolding 5 ϑ'_def ϑ''_def fun_point_Union_list_def ts'_def by auto

  have 7: "?P σ'" "σ' x ≠ {}"
    when σ': "σ' ∈ set ϑ''" for σ'
    using that 1 3 unfolding ϑ''_def ts'_def by auto

  have "σ' x = {a}"
```

```
    when σ': "σ' ∈ set ϑ''" for σ'
    using σ' a' 1 unfolding ϑ''_def ts'_def by fastforce
  hence "fun_point_Union_list ϑ'' x = {b | b σ'. σ' ∈ set ϑ'' ∧ b ∈ {a}}"
    using ϑ''_not_nil unfolding fun_point_Union_list_def by auto
  hence 8: "fun_point_Union_list ϑ'' x = {a}"
    using ϑ''_not_nil by auto

  show ?case
    using 8 a
    unfolding ϑ(2) 6 ticl_abs_iff[OF TI] by auto
qed simp_all

lemma match_abssD:
  fixes OCC TI s
  defines "f ≡ (λδ x. if x ∈ fv s then δ x else set OCC)"
    and "g ≡ (λδ x. ⋂(ticl_abs TI ` δ x))"
  assumes δ': "match_abss OCC TI s t = Some δ'"
  shows "∃δ. match_abss' s t = Some δ ∧ δ' = f (g δ) ∧ (∀x ∈ fv s. δ x ≠ {} ∧ f (g δ) x ≠ {}) ∧
           (set OCC ≠ {} ⟶ (∀x. f (g δ) x ≠ {}))"
proof -
  obtain δ where δ: "match_abss' s t = Some δ"
    using δ' unfolding match_abss_def by force
  hence "Some δ' = (if ∀x ∈ fv s. g δ x ≠ {} then Some (f (g δ)) else None)"
    using δ' unfolding match_abss_def f_def g_def Let_def by simp
  hence "δ' = f (g δ)" "∀x ∈ fv s. δ x ≠ {} ∧ f (g δ) x ≠ {}"
    by (metis (no_types, lifting) option.inject option.distinct(1),
        metis (no_types, lifting) f_def option.distinct(1) match_abss'_fv_is_nonempty[OF δ])
  thus ?thesis using δ unfolding f_def by force
qed

lemma match_abss_ticl_abs_Inter_subset:
  assumes TI: "set TI = {(a,b). (a,b) ∈ (set TI)⁺ ∧ a ≠ b}"
    and δ: "match_abss OCC TI s t = Some δ"
    and x: "x ∈ fv s"
  shows "⋂(ticl_abs TI ` δ x) ⊆ δ x"
proof -
  let ?h1 = "λδ x. if x ∈ fv s then δ x else set OCC"
  let ?h2 = "λδ x. ⋂(ticl_abs TI ` δ x)"

  obtain δ' where δ':
      "match_abss' s t = Some δ'" "δ = ?h1 (?h2 δ')"
      "∀x ∈ fv s. δ' x ≠ {} ∧ δ x ≠ {}"
    using match_abssD[OF δ] by blast

  have "δ x = ⋂(ticl_abs TI ` δ' x)" "δ' x ≠ {}" "δ x ≠ {}"
    using x δ'(2,3) by auto
  thus ?thesis
    using ticl_abs_Inter TI by simp
qed

lemma match_abss_fv_has_abs:
  assumes "match_abss OCC TI s t = Some δ"
    and "x ∈ fv s"
  shows "δ x ≠ {}"
using assms match_abssD by fast

lemma match_abss_OCC_if_not_fv:
  fixes s t::"(('a,'b,'c,'d) prot_fun, 'v) term"
  assumes δ': "match_abss OCC TI s t = Some δ'"
    and x: "x ∉ fv s"
  shows "δ' x = set OCC"
proof -
  define f where "f ≡ λs::(('a,'b,'c,'d) prot_fun, 'v) term. λδ x. if x ∈ fv s then δ x else set
```

```
OCC"
  define g where "g ≡ λδ. λx::'v. ⋂(ticl_abs TI ` δ x)"

  obtain δ where δ: "match_abss' s t = Some δ" "δ' = f s (g δ)"
    using match_abssD[OF δ'] unfolding f_def g_def by blast

  show ?thesis
    using x δ(2) unfolding f_def by presburger
qed

inductive synth_abs_substs_constrs_rel for FP OCC TI where
  SolveNil:
    "synth_abs_substs_constrs_rel FP OCC TI [] (λ_. set OCC)"
| SolveCons:
    "ts ≠ [] ⟹ ∀t ∈ set ts. synth_abs_substs_constrs_rel FP OCC TI [t] (ϑ t)
      ⟹ synth_abs_substs_constrs_rel FP OCC TI ts (fun_point_Inter (ϑ ` set ts))"
| SolvePubConst:
    "arity c = 0 ⟹ public c
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Fun c []] (λ_. set OCC)"
| SolvePrivConstIn:
    "arity c = 0 ⟹ ¬public c ⟹ Fun c [] ∈ set FP
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Fun c []] (λ_. set OCC)"
| SolvePrivConstNotin:
    "arity c = 0 ⟹ ¬public c ⟹ Fun c [] ∉ set FP
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Fun c []] (λ_. {})"
| SolveValueVar:
    "ϑ = ((λ_. set OCC)(x := ticl_abss TI {a ∈ set OCC. ⟨a⟩_abs ∈ set FP}))
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Var x] ϑ"
| SolvePubComposed:
    "arity f > 0 ⟹ public f ⟹ length ts = arity f
      ⟹ ∀δ. δ ∈ Δ ⟷ (∃s ∈ set FP. match_abss OCC TI (Fun f ts) s = Some δ)
      ⟹ ϑ1 = fun_point_Union Δ
      ⟹ synth_abs_substs_constrs_rel FP OCC TI ts ϑ2
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Fun f ts] (fun_point_union ϑ1 ϑ2)"
| SolvePrivComposed:
    "arity f > 0 ⟹ ¬public f ⟹ length ts = arity f
      ⟹ ∀δ. δ ∈ Δ ⟷ (∃s ∈ set FP. match_abss OCC TI (Fun f ts) s = Some δ)
      ⟹ ϑ = fun_point_Union Δ
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Fun f ts] ϑ"

fun synth_abs_substs_constrs_aux where
  "synth_abs_substs_constrs_aux FP OCC TI (Var x) = (
    (λ_. set OCC)(x := ticl_abss TI (set (filter (λa. ⟨a⟩_abs ∈ set FP) OCC))))"
| "synth_abs_substs_constrs_aux FP OCC TI (Fun f ts) = (
    if ts = []
    then (if ¬public f ∧ Fun f ts ∉ set FP then (λ_. {}) else (λ_. set OCC))
    else (let Δ = map the (filter (λδ. δ ≠ None) (map (match_abss OCC TI (Fun f ts)) FP));
              ϑ1 = fun_point_Union_list Δ;
              ϑ2 = fun_point_Inter_list (
                    case ts of t#ts' ⇒
                      if ¬is_Var t ∧ args t = [] ∧ ¬public (the_Fun t)
                      then (if t ∉ set FP then [λ_. {}]
                            else (λ_. set OCC)#map (synth_abs_substs_constrs_aux FP OCC TI) ts')
                      else map (synth_abs_substs_constrs_aux FP OCC TI) ts
                  )
          in fun_point_union ϑ1 ϑ2))"

lemma synth_abs_substs_constrs_aux_fun_case:
  assumes ts: "ts ≠ []"
  shows "synth_abs_substs_constrs_aux FP OCC TI (Fun f ts) = (
    let Δ = map the (filter (λδ. δ ≠ None) (map (match_abss OCC TI (Fun f ts)) FP));
        ϑ1 = fun_point_Union_list Δ;
        ϑ2 = fun_point_Inter_list (map (synth_abs_substs_constrs_aux FP OCC TI) ts)
```

```
    in fun_point_union ϑ1 ϑ2)"
proof -
  let ?s = "synth_abs_substs_constrs_aux FP OCC TI"
  let ?P = "λt. ¬is_Var t ∧ args t = [] ∧ ¬public (the_Fun t)"

  obtain t ts' where ts': "ts = t#ts'" using ts by (cases ts) auto

  have "fun_point_Inter_list ((λ_. {})#map ?s ts') = fun_point_Inter_list [λ_. {}]"
    unfolding fun_point_Inter_list_def by simp
  thus ?thesis unfolding ts' by (cases "?P t") auto
qed

definition synth_abs_substs_constrs where
  "synth_abs_substs_constrs FPT T ≡
    let (FP,OCC,TI) = FPT;
        ts = trms_list_sst (unlabel (transaction_receive T));
        f = fun_point_Inter_list ∘ map (synth_abs_substs_constrs_aux FP OCC TI)
    in if ts = [] then (λ_. set OCC) else f ts"



definition transaction_check_comp::
  "[('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set ⇒ bool,
    ('fun,'atom,'sets,'lbl) prot_term list ×
    'sets set list ×
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ ((('fun,'atom,'sets,'lbl) prot_var × 'sets set) list) list"
where
  "transaction_check_comp msgcs FPT T ≡
    let (_, OCC, _) = FPT;
        S = unlabel (transaction_strand T);
        C = unlabel (transaction_checks T);
        xs = filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S);
        posconstrs = transaction_poschecks_comp C;
        negconstrs = transaction_negchecks_comp C;
        pre_check = transaction_check_pre FPT T;
        Δ = abs_substs_set xs OCC posconstrs negconstrs msgcs
    in filter (λδ. pre_check (abs_substs_fun δ)) Δ"

definition transaction_check'::
  "[('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set ⇒ bool,
    ('fun,'atom,'sets,'lbl) prot_term list ×
    'sets set list ×
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ bool"
where
  "transaction_check' msgcs FPT T ≡
    list_all (λδ. transaction_check_post FPT T (abs_substs_fun δ))
             (transaction_check_comp msgcs FPT T)"

definition transaction_check::
  "[('fun,'atom,'sets,'lbl) prot_term list ×
    'sets set list ×
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ bool"
where
  "transaction_check ≡ transaction_check' (λ_ _. True)"

definition transaction_check_coverage_rcv::
  "[('fun,'atom,'sets,'lbl) prot_term list ×
```

```
    'sets set list ×
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ bool"
where
  "transaction_check_coverage_rcv FPT T ≡
    let msgcs = synth_abs_substs_constrs FPT T
    in transaction_check' (λx a. a ∈ msgcs x) FPT T"

lemma abs_subst_fun_cons:
  "abs_substs_fun ((x,b)#δ) = (abs_substs_fun δ)(x := b)"
unfolding abs_substs_fun_def by fastforce

lemma abs_substs_cons:
  assumes "δ ∈ set (abs_substs_set xs as poss negs msgcs)"
          "b ∈ set as" "poss x ⊆ b" "b ∩ negs x = {}" "msgcs x b"
  shows "(x,b)#δ ∈ set (abs_substs_set (x#xs) as poss negs msgcs)"
using assms by auto

lemma abs_substs_cons':
  assumes δ: "δ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)"
    and b: "b ∈ set as" "poss x ⊆ b" "b ∩ negs x = {}" "msgcs x b"
  shows "δ(x := b) ∈ abs_substs_fun ` set (abs_substs_set (x#xs) as poss negs msgcs)"
proof -
  obtain ϑ where ϑ: "δ = abs_substs_fun ϑ" "ϑ ∈ set (abs_substs_set xs as poss negs msgcs)"
    using δ by force
  have "abs_substs_fun ((x, b)#ϑ) ∈ abs_substs_fun ` set (abs_substs_set (x#xs) as poss negs msgcs)"
    using abs_substs_cons[OF ϑ(2) b] by blast
  thus ?thesis
    using ϑ(1) abs_subst_fun_cons[of x b ϑ] by argo
qed

lemma abs_substs_has_abs:
  assumes "∀x. x ∈ set xs ⟶ δ x ∈ set as"
    and "∀x. x ∈ set xs ⟶ poss x ⊆ δ x"
    and "∀x. x ∈ set xs ⟶ δ x ∩ negs x = {}"
    and "∀x. x ∈ set xs ⟶ msgcs x (δ x)"
    and "∀x. x ∉ set xs ⟶ δ x = {}"
  shows "δ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)"
using assms
proof (induction xs arbitrary: δ)
  case (Cons x xs)
  define ϑ where "ϑ ≡ λy. if y ∈ set xs then δ y else {}"

  have "ϑ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)"
    using Cons.prems Cons.IH by (simp add: ϑ_def)
  moreover have "δ x ∈ set as" "poss x ⊆ δ x" "δ x ∩ negs x = {}" "msgcs x (δ x)"
    by (simp_all add: Cons.prems(1,2,3,4))
  ultimately have 0: "ϑ(x := δ x) ∈ abs_substs_fun ` set (abs_substs_set (x#xs) as poss negs msgcs)"
    by (metis abs_substs_cons')

  have "δ = ϑ(x := δ x)"
  proof
    fix y show "δ y = (ϑ(x := δ x)) y"
    proof (cases "y ∈ set (x#xs)")
      case False thus ?thesis using Cons.prems(5) by (fastforce simp add: ϑ_def)
    qed (auto simp add: ϑ_def)
  qed
  thus ?case by (metis 0)
qed (auto simp add: abs_substs_fun_def)

lemma abs_substs_abss_bounded:
  assumes "δ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)"
```

```
    and "x ∈ set xs"
  shows "δ x ∈ set as"
    and "poss x ⊆ δ x"
    and "δ x ∩ negs x = {}"
    and "msgcs x (δ x)"
using assms
proof (induct xs as poss negs msgcs arbitrary: δ rule: abs_substs_set_induct)
  case (Cons y xs as poss negs msgcs)
  { case 1 thus ?case using Cons.hyps(1) unfolding abs_substs_fun_def by fastforce }

  { case 2 thus ?case
    proof (cases "x = y")
      case False
      then obtain δ' where δ':
          "δ' ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)" "δ' x = δ x"
        using 2 unfolding abs_substs_fun_def by force
      moreover have "x ∈ set xs" using 2(2) False by simp
      moreover have "∃b. b ∈ set as ∧ poss y ⊆ b ∧ b ∩ negs y = {}"
        using 2 False by auto
      ultimately show ?thesis using Cons.hyps(2) by fastforce
    qed (auto simp add: abs_substs_fun_def)
  }

  { case 3 thus ?case
    proof (cases "x = y")
      case False
      then obtain δ' where δ':
          "δ' ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)" "δ' x = δ x"
        using 3 unfolding abs_substs_fun_def by force
      moreover have "x ∈ set xs" using 3(2) False by simp
      moreover have "∃b. b ∈ set as ∧ poss y ⊆ b ∧ b ∩ negs y = {}"
        using 3 False by auto
      ultimately show ?thesis using Cons.hyps(3) by fastforce
    qed (auto simp add: abs_substs_fun_def)
  }

  { case 4 thus ?case
    proof (cases "x = y")
      case False
      then obtain δ' where δ':
          "δ' ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)" "δ' x = δ x"
        using 4 unfolding abs_substs_fun_def by force
      moreover have "x ∈ set xs" using 4(2) False by simp
      moreover have "∃b. b ∈ set as ∧ poss y ⊆ b ∧ b ∩ negs y = {}"
        using 4 False by auto
      ultimately show ?thesis using Cons.hyps(4) by fastforce
    qed (auto simp add: abs_substs_fun_def)
  }
qed (simp_all add: abs_substs_fun_def)

lemma abs_substs_abss_bounded':
  assumes "δ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)"
    and "x ∉ set xs"
  shows "δ x = {}"
using assms unfolding abs_substs_fun_def
by (induct xs as poss negs msgcs arbitrary: δ rule: abs_substs_set_induct) (force, fastforce)

lemma transaction_poschecks_comp_unfold:
  "transaction_poschecks_comp C x = {s. ∃a. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ set C}"
proof (induction C)
  case (Cons c C) thus ?case
  proof (cases "∃a y s. c = ⟨a: Var y ∈ Fun (Set s) []⟩")
    case True
```

```
    then obtain a y s where c: "c = ⟨a: Var y ∈ Fun (Set s) []⟩" by force

    define f where "f ≡ transaction_poschecks_comp C"

    have "transaction_poschecks_comp (c#C) = f(y := insert s (f y))"
      using c by (simp add: f_def Let_def)
    moreover have "f x = {s. ∃a. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ set C}"
      using Cons.IH unfolding f_def by blast
    ultimately show ?thesis using c by auto
  next
    case False
    hence "transaction_poschecks_comp (c#C) = transaction_poschecks_comp C" (is ?P)
      using transaction_poschecks_comp.cases[of "c#C" ?P] by force
    thus ?thesis using False Cons.IH by auto
  qed
qed simp

lemma transaction_poschecks_comp_notin_fv_empty:
  assumes "x ∉ fv_sst C"
  shows "transaction_poschecks_comp C x = {}"
using assms transaction_poschecks_comp_unfold[of C x] by fastforce

lemma transaction_negchecks_comp_unfold:
  "transaction_negchecks_comp C x = {s. ⟨Var x not in Fun (Set s) []⟩ ∈ set C}"
proof (induction C)
  case (Cons c C) thus ?case
  proof (cases "∃y s. c = ⟨Var y not in Fun (Set s) []⟩")
    case True
    then obtain y s where c: "c = ⟨Var y not in Fun (Set s) []⟩" by force

    define f where "f ≡ transaction_negchecks_comp C"

    have "transaction_negchecks_comp (c#C) = f(y := insert s (f y))"
      using c by (simp add: f_def Let_def)
    moreover have "f x = {s. ⟨Var x not in Fun (Set s) []⟩ ∈ set C}"
      using Cons.IH unfolding f_def by blast
    ultimately show ?thesis using c by auto
  next
    case False
    hence "transaction_negchecks_comp (c#C) = transaction_negchecks_comp C" (is ?P)
      using transaction_negchecks_comp.cases[of "c#C" ?P]
      by force
    thus ?thesis using False Cons.IH by fastforce
  qed
qed simp

lemma transaction_negchecks_comp_notin_fv_empty:
  assumes "x ∉ fv_sst C"
  shows "transaction_negchecks_comp C x = {}"
using assms transaction_negchecks_comp_unfold[of C x] by fastforce

lemma transaction_check_preI[intro]:
  fixes T
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "C ≡ set (unlabel (transaction_checks T))"
  assumes a0: "∀x ∈ set (transaction_fresh T). δ x = {}"
    and a1: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ δ x ∈ set
OCC"
    and a2: "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)"
    and a3: "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ C ⟶ s ∈ δ x"
    and a4: "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ δ x"
  shows "transaction_check_pre (FP, OCC, TI) T δ"
proof -
```

```
  let ?P = "λu. is_InSet u ⟶ (
    let x = the_elem_term u; s = the_set_term u
    in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∈ δ (the_Var x))"

  let ?Q = "λu. (is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1) ⟶ (
    let x = fst (hd (the_ins u)); s = snd (hd (the_ins u))
    in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∉ δ (the_Var x))"

  have 1: "?P u" when u: "u ∈ C" for u
    apply (unfold Let_def, intro impI, elim conjE)
    using u a3 Fun_Set_InSet_iff[of u] by metis

  have 2: "?Q u" when u: "u ∈ C" for u
    apply (unfold Let_def, intro impI, elim conjE)
    using u a4 Fun_Set_NotInSet_iff[of u] by metis

  show ?thesis
    using a0 a1 a2 1 2 fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"]
    unfolding transaction_check_pre_def ϑ_def C_def Let_def
    by blast
qed

lemma transaction_check_pre_InSetE:
  assumes T: "transaction_check_pre FPT T δ"
    and u: "u = ⟨a: Var x ∈ Fun (Set s) []⟩"
           "u ∈ set (unlabel (transaction_checks T))"
  shows "s ∈ δ x"
proof -
  have "is_InSet u ⟶ is_Var (the_elem_term u) ∧ is_Fun_Set (the_set_term u) ⟶
        the_Set (the_Fun (the_set_term u)) ∈ δ (the_Var (the_elem_term u))"
    using T u unfolding transaction_check_pre_def Let_def by blast
  thus ?thesis using Fun_Set_InSet_iff[of u a x s] u by argo
qed

lemma transaction_check_pre_NotInSetE:
  assumes T: "transaction_check_pre FPT T δ"
    and u: "u = ⟨Var x not in Fun (Set s) []⟩"
           "u ∈ set (unlabel (transaction_checks T))"
  shows "s ∉ δ x"
proof -
  have "is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1 ⟶
         is_Var (fst (hd (the_ins u))) ∧ is_Fun_Set (snd (hd (the_ins u))) ⟶
         the_Set (the_Fun (snd (hd (the_ins u)))) ∉ δ (the_Var (fst (hd (the_ins u))))"
    using T u unfolding transaction_check_pre_def Let_def by blast
  thus ?thesis using Fun_Set_NotInSet_iff[of u  x s] u by argo
qed

lemma transaction_check_pre_ReceiveE:
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
  assumes T: "transaction_check_pre (FP, OCC, TI) T δ"
    and t: "t ∈ trms_lsst (transaction_receive T)"
  shows "intruder_synth_mod_timpls FP TI (t · ϑ δ)"
using T t unfolding transaction_check_pre_def Let_def ϑ_def by blast

lemma transaction_check_compI[intro]:
  assumes T: "transaction_check_pre (FP, OCC, TI) T δ"
    and T_adm: "admissible_transaction' T"
    and x1: "∀x. (x ∈ fv_transaction T - set (transaction_fresh T) ∧ fst x = TAtom Value)
                 ⟶ δ x ∈ set OCC ∧ msgcs x (δ x)"
    and x2: "∀x. (x ∉ fv_transaction T - set (transaction_fresh T) ∨ fst x ≠ TAtom Value)
                 ⟶ δ x = {}"
  shows "δ ∈ abs_substs_fun ` set (transaction_check_comp msgcs (FP, OCC, TI) T)"
proof -
```

```
  define S where "S ≡ unlabel (transaction_strand T)"
  define C where "C ≡ unlabel (transaction_checks T)"

  let ?xs = "fv_list_sst S"

  define poss where "poss ≡ transaction_poschecks_comp C"
  define negs where "negs ≡ transaction_negchecks_comp C"
  define ys where "ys ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) ?xs"

  have ys: "{x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value} = set ys"
    using fv_list_sst_is_fv_sst[of S]
    unfolding ys_def S_def by force

  have "δ x ∈ set OCC" "msgcs x (δ x)"
    when x: "x ∈ set ys" for x
    using x1 x ys by (blast, blast)
  moreover have "δ x = {}"
    when x: "x ∉ set ys" for x
    using x2 x ys by blast
  moreover have "poss x ⊆ δ x" when x: "x ∈ set ys" for x
  proof -
    have "s ∈ δ x" when u: "u = ⟨a: Var x ∈ Fun (Set s) []⟩" "u ∈ set C" for u a s
      using T u transaction_check_pre_InSetE[of "(FP, OCC, TI)" T δ]
      unfolding C_def by blast
    thus ?thesis
      using transaction_poschecks_comp_unfold[of C x]
      unfolding poss_def by blast
  qed
  moreover have "δ x ∩ negs x = {}" when x: "x ∈ set ys" for x
  proof (cases "x ∈ fv_sst C")
    case True
    hence "s ∉ δ x" when u: "u = ⟨Var x not in Fun (Set s) []⟩" "u ∈ set C" for u s
      using T u transaction_check_pre_NotInSetE[of "(FP, OCC, TI)" T δ]
      unfolding C_def by blast
    thus ?thesis
      using transaction_negchecks_comp_unfold[of C x]
      unfolding negs_def by blast
  next
    case False
    hence "negs x = {}"
      using x transaction_negchecks_comp_notin_fv_empty
      unfolding negs_def by blast
    thus ?thesis by blast
  qed
  ultimately have "δ ∈ abs_substs_fun ` set (abs_substs_set ys OCC poss negs msgcs)"
    using abs_substs_has_abs[of ys δ OCC poss negs msgcs]
    by fast
  thus ?thesis
    using T
    unfolding transaction_check_comp_def Let_def S_def C_def ys_def poss_def negs_def
    by fastforce
qed

context
begin
private lemma transaction_check_comp_in_aux:
  fixes T
  defines "C ≡ set (unlabel (transaction_checks T))"
  assumes T_adm: "admissible_transaction' T"
    and a1: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
          select⟨Var x, Fun (Set s) []⟩ ∈ C ⟶ s ∈ α x)"
    and a2: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
          ⟨Var x in Fun (Set s) []⟩ ∈ C ⟶ s ∈ α x)"
```

```
    and a3: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
          ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α x)"
  shows "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ C ⟶ s ∈ α x" (is ?A)
    and "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α x" (is ?B)
proof -
  note * = admissible_transaction_strand_step_cases(2,3)[OF T_adm]

  have 1: "fst x = TAtom Value" "x ∈ fv_transaction T - set (transaction_fresh T)"
    when x: "⟨a: Var x ∈ Fun (Set s) []⟩ ∈ C" for a x s
    using * x unfolding C_def by fast+

  have 2: "fst x = TAtom Value" "x ∈ fv_transaction T - set (transaction_fresh T)"
    when x: "⟨Var x not in Fun (Set s) []⟩ ∈ C" for x s
    using * x unfolding C_def by fast+

  show ?A
  proof (intro allI impI)
    fix a x s assume u: "⟨a: Var x ∈ Fun (Set s) []⟩ ∈ C"
    thus "s ∈ α x" using 1 a1 a2 by (cases a) metis+
  qed

  show ?B
  proof (intro allI impI)
    fix x s assume u: "⟨Var x not in Fun (Set s) []⟩ ∈ C"
    thus "s ∉ α x" using 2 a3 by meson
  qed
qed

lemma transaction_check_comp_in:
  fixes T
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "C ≡ set (unlabel (transaction_checks T))"
  assumes T_adm: "admissible_transaction' T"
    and a1: "∀x ∈ set (transaction_fresh T). α x = {}"
    and a2: "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ α)"
    and a3: "∀x ∈ fv_transaction T - set (transaction_fresh T). ∀s.
          select⟨Var x, Fun (Set s) []⟩ ∈ C ⟶ s ∈ α x"
    and a4: "∀x ∈ fv_transaction T - set (transaction_fresh T). ∀s.
          ⟨Var x in Fun (Set s) []⟩ ∈ C ⟶ s ∈ α x"
    and a5: "∀x ∈ fv_transaction T - set (transaction_fresh T). ∀s.
          ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α x"
    and a6: "∀x ∈ fv_transaction T - set (transaction_fresh T).
          fst x = TAtom Value ⟶ α x ∈ set OCC"
    and a7: "∀x ∈ fv_transaction T - set (transaction_fresh T).
          fst x = TAtom Value ⟶ msgcs x (α x)"
  shows "∃δ ∈ abs_substs_fun ` set (transaction_check_comp msgcs (FP, OCC, TI) T).
          ∀x ∈ fv_transaction T. fst x = TAtom Value ⟶ α x = δ x"
proof -
  let ?xs = "fv_list_sst (unlabel (transaction_strand T))"
  let ?ys = "filter (λx. x ∉ set (transaction_fresh T)) ?xs"

  define α' where "α' ≡ λx.
    if x ∈ fv_transaction T - set (transaction_fresh T) ∧ fst x = TAtom Value
    then α x
    else {}"

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

  have ϑα_Fun: "is_Fun (t · ϑ α) ⟷ is_Fun (t · ϑ α')" for t
    unfolding α'_def ϑ_def
    by (induct t) auto

  have "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ α')"
```

```
  proof (intro ballI impI)
    fix t assume t: "t ∈ trms_lsst (transaction_receive T)"

    have 1: "intruder_synth_mod_timpls FP TI (t · ϑ α)"
      using t a2
      by auto

    obtain r where r:
        "r ∈ set (unlabel (transaction_receive T))"
        "t ∈ trms_sstp r"
      using t by auto
    hence "∃ts. r = receive⟨ts⟩ ∧ t ∈ set ts"
      using wellformed_transaction_unlabel_cases(1)[OF T_wf]
      by fastforce
    hence 2: "fv t ⊆ fv_lsst (transaction_receive T)" using r by force

    have "fv t ⊆ fv_transaction T"
      by (metis (no_types, lifting) 2 transaction_strand_def sst_vars_append_subset(1)
                unlabel_append subset_Un_eq sup.bounded_iff)
    moreover have "fv t ∩ set (transaction_fresh T) = {}"
      using 2 T_wf vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_receive T)"]
      unfolding wellformed_transaction_def
      by fast
    ultimately have "ϑ α x = ϑ α' x" when "x ∈ fv t" for x
      using that unfolding α'_def ϑ_def by fastforce
    hence 3: "t · ϑ α = t · ϑ α'"
      using term_subst_eq by blast

    show "intruder_synth_mod_timpls FP TI (t · ϑ α')" using 1 3 by simp
  qed
  moreover have
      "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
          select⟨Var x, Fun (Set s) []⟩ ∈ C ⟶ s ∈ α' x)"
      "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
          ⟨Var x in Fun (Set s) []⟩ ∈ C ⟶ s ∈ α' x)"
      "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
          ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α' x)"
    using a3 a4 a5
    unfolding α'_def ϑ_def C_def
    by meson+
  hence "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ C ⟶ s ∈ α' x"
        "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α' x"
    using transaction_check_comp_in_aux[OF T_adm, of α']
    unfolding C_def
    by (fast, fast)
  ultimately have 4: "transaction_check_pre (FP, OCC, TI) T α'"
    using a6 transaction_check_preI[of T α' OCC FP TI]
    unfolding α'_def ϑ_def C_def
    by simp

  have 5: "∀x ∈ fv_transaction T. fst x = TAtom Value ⟶ α x = α' x"
    using a1 by (auto simp add: α'_def)

  have 6: "α' ∈ abs_substs_fun ` set (transaction_check_comp msgcs (FP, OCC, TI) T)"
    using transaction_check_compI[OF 4 T_adm, of msgcs] a6 a7
    unfolding α'_def
    by auto

  show ?thesis using 5 6 by blast
qed
end

lemma transaction_check_trivial_case:
```

```
  assumes "transaction_updates T = []"
    and "transaction_send T = []"
  shows "transaction_check FPT T"
using assms
by (simp add: list_all_iff transaction_check_def transaction_check'_def transaction_check_post_def)

end
```

3.6.3 Soundness of the Occurs-Message Transformation

```
context stateful_protocol_model
begin

context
begin
```

The occurs-message transformation, *add_occurs_msgs*, extends a transaction *T* with additional message-transmission steps such that the following holds: 1. for each fresh variable *x* of *T* the message *occurs (Var x)* now occurs in a send-step, 2. for each of the remaining free variables *x* of *T* the message *occurs (Var x)* now occurs in a receive-step.

```
definition add_occurs_msgs where
  "add_occurs_msgs T ≡
    let frsh = transaction_fresh T;
        xs = filter (λx. x ∉ set frsh) (fv_list_sst (unlabel (transaction_strand T)));
        f = map (λx. occurs (Var x));
        g = λC. if xs = [] then C else ⟨⋆, receive⟨f xs⟩⟩#C;
        h = λF. if frsh = [] then F
                else if F ≠ [] ∧ fst (hd F) = ⋆ ∧ is_Send (snd (hd F))
                then ⟨⋆,send⟨f frsh@the_msgs (snd (hd F))⟩⟩#tl F
                else ⟨⋆,send⟨f frsh⟩⟩#F
    in case T of Transaction A B C D E F ⇒ Transaction A B (g C) D E (h F)"

private fun rm_occurs_msgs_constr where
  "rm_occurs_msgs_constr [] = []"
| "rm_occurs_msgs_constr ((l,receive⟨ts⟩)#A) = (
    if ∃t. occurs t ∈ set ts
    then if ∃t ∈ set ts. ∀s. t ≠ occurs s
         then (l,receive⟨filter (λt. ∀s. t ≠ occurs s) ts⟩)#rm_occurs_msgs_constr A
         else rm_occurs_msgs_constr A
    else (l,receive⟨ts⟩)#rm_occurs_msgs_constr A)"
| "rm_occurs_msgs_constr ((l,send⟨ts⟩)#A) = (
    if ∃t. occurs t ∈ set ts
    then if ∃t ∈ set ts. ∀s. t ≠ occurs s
         then (l,send⟨filter (λt. ∀s. t ≠ occurs s) ts⟩)#rm_occurs_msgs_constr A
         else rm_occurs_msgs_constr A
    else (l,send⟨ts⟩)#rm_occurs_msgs_constr A)"
| "rm_occurs_msgs_constr (a#A) = a#rm_occurs_msgs_constr A"

private lemma add_occurs_msgs_cases:
  fixes T C frsh xs f
  defines "T' ≡ add_occurs_msgs T"
    and "frsh ≡ transaction_fresh T"
    and "xs ≡ filter (λx. x ∉ set frsh) (fv_list_sst (unlabel (transaction_strand T)))"
    and "xs' ≡ fv_transaction T - set frsh"
    and "f ≡ map (λx. occurs (Var x))"
    and "C' ≡ if xs = [] then C else ⟨⋆, receive⟨f xs⟩⟩#C"
    and "ts' ≡ f frsh"
  assumes T: "T = Transaction A B C D E F"
  shows "F = ⟨⋆,send⟨ts⟩⟩#F' ⟹ T' = Transaction A B C' D E (⟨⋆,send⟨ts'@ts⟩⟩#F')"
    (is "?A ts F' ⟹ ?A' ts F'")
  and "∄ts' F'. F = ⟨⋆,send⟨ts'⟩⟩#F' ⟹ frsh ≠ [] ⟹ T' = Transaction A B C' D E (⟨⋆,send⟨ts'⟩⟩#F)"
    (is "?B ⟹ ?B' ⟹ ?B''")
  and "frsh = [] ⟹ T' = Transaction A B C' D E F" (is "?C ⟹ ?C'")
```

```
  and "transaction_decl T' = transaction_decl T"
  and "transaction_fresh T' = transaction_fresh T"
  and "xs = [] ⟹ transaction_receive T' = transaction_receive T"
  and "xs ≠ [] ⟹ transaction_receive T' = ⟨⋆,receive⟨f xs⟩⟩#transaction_receive T"
  and "transaction_checks T' = transaction_checks T"
  and "transaction_updates T' = transaction_updates T"
  and "transaction_send T = ⟨⋆,send⟨ts⟩⟩#F' ⟹
        transaction_send T' = ⟨⋆,send⟨ts'@ts⟩⟩#F'" (is "?D ts F' ⟹ ?D' ts F'")
  and "∄ts' F'. transaction_send T = ⟨⋆,send⟨ts'⟩⟩#F' ⟹ frsh ≠ [] ⟹
        transaction_send T' = ⟨⋆,send⟨ts'⟩⟩#transaction_send T" (is "?E ⟹ ?E' ⟹ ?E''")
  and "frsh = [] ⟹ transaction_send T' = transaction_send T" (is "?F ⟹ ?F'")
  and "(xs' ≠ {} ∧ transaction_receive T' = ⟨⋆, receive⟨f xs⟩⟩#transaction_receive T) ∨
       (xs' = {} ∧ transaction_receive T' = transaction_receive T)" (is ?G)
  and "(frsh ≠ [] ∧ (∃ts F'.
          transaction_send T = ⟨⋆,send⟨ts⟩⟩#F' ∧ transaction_send T' = ⟨⋆,send⟨ts'@ts⟩⟩#F')) ∨
       (frsh ≠ [] ∧ transaction_send T' = ⟨⋆,send⟨ts'⟩⟩#transaction_send T) ∨
       (frsh = [] ∧ transaction_send T' = transaction_send T)" (is ?H)
proof -
  note defs = T'_def T frsh_def xs_def xs'_def f_def C'_def ts'_def add_occurs_msgs_def Let_def

  show 0: "?A ts F' ⟹ ?A' ts F'" for ts F' unfolding defs by simp

  have "F = [] ∨ fst (hd F) ≠ ⋆ ∨ ¬is_Send (snd (hd F))" when ?B
    using that unfolding is_Send_def by (cases F) auto
  thus 1: "?B ⟹ ?B' ⟹ ?B''" unfolding defs by force

  show "?C ⟹ ?C'" unfolding defs by auto

  show "transaction_decl T' = transaction_decl T"
       "transaction_fresh T' = transaction_fresh T"
       "transaction_checks T' = transaction_checks T"
       "transaction_updates T' = transaction_updates T"
    unfolding defs by simp_all

  show "xs = [] ⟹ transaction_receive T' = transaction_receive T"
       "xs ≠ [] ⟹ transaction_receive T' = ⟨⋆, receive⟨f xs⟩⟩#transaction_receive T"
    unfolding defs by simp_all
  moreover have "xs = [] ⟷ xs' = {}"
    using filter_empty_conv[of "λx. x ∉ set frsh"]
          fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"]
    unfolding xs_def xs'_def by blast
  ultimately show ?G by blast

  show 2: "?D ts F' ⟹ ?D' ts F'" for ts F' using 0 unfolding T by simp
  show 3: "?E ⟹ ?E' ⟹ ?E''" using 1 unfolding T by force
  show 4: "?F ⟹ ?F'" unfolding defs by simp

  show ?H
  proof (cases "frsh = []")
    case False thus ?thesis
      using 2 3[OF _ False] by (cases "∃ts F'. transaction_send T = ⟨⋆,send⟨ts⟩⟩#F'") (blast,blast)
  qed (simp add: 4)
qed

private lemma add_occurs_msgs_transaction_strand_set:
  fixes T C frsh xs f
  defines "frsh ≡ transaction_fresh T"
    and "xs ≡ filter (λx. x ∉ set frsh) (fv_list_sst (unlabel (transaction_strand T)))"
    and "f ≡ map (λx. occurs (Var x))"
  assumes T: "T = Transaction A B C D E F"
  shows "F = ⟨⋆,send⟨ts⟩⟩#F' ⟹
          set (transaction_strand (add_occurs_msgs T)) ⊆
          set (transaction_strand T) ∪ {⟨⋆,receive⟨f xs⟩⟩,⟨⋆,send⟨f frsh@ts⟩⟩}"
```

```
    (is "?A ⟹ ?A'")
  and "F = ⟨⋆,send⟨ts⟩⟩#F' ⟹
         set (unlabel (transaction_strand (add_occurs_msgs T))) ⊆
         set (unlabel (transaction_strand T)) ∪ {receive⟨f xs⟩,send⟨f frsh@ts⟩}"
    (is "?B ⟹ ?B'")
  and "∄ts' F'. F = ⟨⋆,send⟨ts'⟩⟩#F' ⟹
         set (transaction_strand (add_occurs_msgs T)) ⊆
         set (transaction_strand T) ∪ {⟨⋆,receive⟨f xs⟩⟩,⟨⋆,send⟨f frsh⟩⟩}"
    (is "?C ⟹ ?C'")
  and "∄ts' F'. F = ⟨⋆,send⟨ts'⟩⟩#F' ⟹
         set (unlabel (transaction_strand (add_occurs_msgs T))) ⊆
         set (unlabel (transaction_strand T)) ∪ {receive⟨f xs⟩,send⟨f frsh⟩}"
    (is "?D ⟹ ?D'")
proof -
  note 0 = add_occurs_msgs_cases[
           OF T, unfolded frsh_def[symmetric] xs_def[symmetric] f_def[symmetric]]

  show "?A ⟹ ?A'" using 0(1,3) unfolding T transaction_strand_def by (cases "frsh = []") auto
  thus "?B ⟹ ?B'" unfolding unlabel_def by force

  show "?C ⟹ ?C'" using 0(2,3) unfolding T transaction_strand_def by (cases "frsh = []") auto
  thus "?D ⟹ ?D'" unfolding unlabel_def by auto
qed

private lemma add_occurs_msgs_transaction_strand_cases:
  fixes T T'::"('a,'b,'c,'d) prot_transaction" and C frsh xs f ϑ
  defines "T' ≡ add_occurs_msgs T"
    and "S ≡ transaction_strand T"
    and "S' ≡ transaction_strand T'"
    and "frsh ≡ transaction_fresh T"
    and "xs ≡ filter (λx. x ∉ set frsh) (fv_list_sst (unlabel (transaction_strand T)))"
    and "f ≡ map (λx. occurs (Var x))"
    and "C ≡ transaction_receive T"
    and "D ≡ transaction_checks T"
    and "E ≡ transaction_updates T"
    and "F ≡ transaction_send T"
    and "C' ≡ if xs = [] then C else ⟨⋆,receive⟨f xs⟩⟩#C"
    and "C'' ≡ if xs = [] then dual_lsst C else ⟨⋆,send⟨f xs⟩⟩#dual_lsst C"
    and "C''' ≡ if xs = [] then dual_lsst (C ·_lsst ϑ) else ⟨⋆,send⟨f xs ·_list ϑ⟩⟩#dual_lsst (C ·_lsst ϑ)"
  shows "frsh = [] ⟹ S' = C'@D@E@F"
      (is "?A ⟹ ?A'")
    and "frsh ≠ [] ⟹ ∄ts F'. F = ⟨⋆,send⟨ts⟩⟩#F' ⟹ S' = C'@D@E@(⟨⋆,send⟨f frsh⟩⟩#F)"
      (is "?B ⟹ ?B' ⟹ ?B''")
    and "frsh ≠ [] ⟹ ∃ts F'. F = ⟨⋆,send⟨ts⟩⟩#F' ⟹
         ∃ts F'. F = ⟨⋆,send⟨ts⟩⟩#F' ∧ S' = C'@D@E@(⟨⋆,send⟨f frsh@ts⟩⟩#F')"
      (is "?C ⟹ ?C' ⟹ ?C''")
    and "frsh = [] ⟹ dual_lsst S' = C''@dual_lsst D@dual_lsst E@dual_lsst F"
      (is "?D ⟹ ?D'")
    and "frsh ≠ [] ⟹ ∄ts F'. F = ⟨⋆,send⟨ts⟩⟩#F' ⟹
         dual_lsst S' = C''@dual_lsst D@dual_lsst E@(⟨⋆,receive⟨f frsh⟩⟩#dual_lsst F)"
      (is "?E ⟹ ?E' ⟹ ?E''")
    and "frsh ≠ [] ⟹ ∃ts F'. F = ⟨⋆,send⟨ts⟩⟩#F' ⟹
         ∃ts F'. F = ⟨⋆,send⟨ts⟩⟩#F' ∧
                 dual_lsst S' = C''@dual_lsst D@dual_lsst E@(⟨⋆,receive⟨f frsh@ts⟩⟩#dual_lsst F')"
      (is "?F ⟹ ?F' ⟹ ?F''")
    and "frsh = [] ⟹
         dual_lsst (S' ·_lsst ϑ) = C'''@dual_lsst (D ·_lsst ϑ)@dual_lsst (E ·_lsst ϑ)@dual_lsst (F ·_lsst ϑ)"
      (is "?G ⟹ ?G'")
    and "frsh ≠ [] ⟹ ∄ts F'. F = ⟨⋆,send⟨ts⟩⟩#F' ⟹
         dual_lsst (S' ·_lsst ϑ) = C'''@dual_lsst (D ·_lsst ϑ)@dual_lsst (E ·_lsst ϑ)@
                                  (⟨⋆,receive⟨f frsh ·_list ϑ⟩⟩#dual_lsst (F ·_lsst ϑ))"
      (is "?H ⟹ ?H' ⟹ ?H''")
    and "frsh ≠ [] ⟹ ∃ts F'. F = ⟨⋆,send⟨ts⟩⟩#F' ⟹
```

```
        ∃ts F'. F = ⟨⋆,send⟨ts⟩⟩#F' ∧
              dual_lsst (S' ·_lsst ϑ) = C'''@dual_lsst (D ·_lsst ϑ)@dual_lsst (E ·_lsst ϑ)@
                                   (⟨⋆,receive⟨f frsh@ts ·_list ϑ⟩⟩#dual_lsst (F' ·_lsst ϑ))"
     (is "?I ⟹ ?I' ⟹ ?I''")
proof -
  obtain A' B' CC' D' E' F' where T: "T = Transaction A' B' CC' D' E' F'" by (cases T) simp

  note 0 = add_occurs_msgs_cases[
          OF T, unfolded frsh_def[symmetric] xs_def[symmetric] f_def[symmetric] T'_def[symmetric]]

  note defs = S'_def C_def D_def E_def F_def C'_def C''_def T transaction_strand_def

  show A: "?A ⟹ ?A'" using 0(3) unfolding defs by simp
  show B: "?B ⟹ ?B' ⟹ ?B''" using 0(2) unfolding defs by simp
  show C: "?C ⟹ ?C' ⟹ ?C''" using 0(1) unfolding defs by force

  have 1: "C''' = C'' ·_lsst ϑ"
    using subst_lsst_cons[of "⟨⋆, send⟨f xs⟩⟩" "dual_lsst C" ϑ] dual_lsst_subst[of C ϑ]
    unfolding C'''_def C''_def by (cases "xs = []") auto

  have 2: "(⟨⋆, receive⟨ts⟩⟩#dual_lsst G) ·_lsst ϑ = ⟨⋆, receive⟨ts ·_list ϑ⟩⟩#dual_lsst (G ·_lsst ϑ)"
    for ts and G::"('a,'b,'c,'d) prot_strand"
    using dual_lsst_subst[of G ϑ] subst_lsst_cons[of "⟨⋆, receive⟨ts⟩⟩" "dual_lsst G" ϑ]
    by simp

  note 3 = subst_lsst_append[of _ _ ϑ] dual_lsst_subst[of _ ϑ]

  show "?D ⟹ ?D'" using A unfolding defs by fastforce
  thus "?G ⟹ ?G'" unfolding 1 by (metis 3)

  show "?E ⟹ ?E' ⟹ ?E''" using B unfolding defs by fastforce
  thus "?H ⟹ ?H' ⟹ ?H''" unfolding 1 by (metis 2 3)

  show "?F ⟹ ?F' ⟹ ?F''" using C unfolding defs by fastforce
  thus "?I ⟹ ?I' ⟹ ?I''" unfolding 1 by (metis 2 3)
qed

private lemma add_occurs_msgs_trms_transaction:
  fixes T::"('a,'b,'c,'d) prot_transaction"
  shows "trms_transaction (add_occurs_msgs T) =
         trms_transaction T ∪ (λx. occurs (Var x))`(fv_transaction T ∪ set (transaction_fresh T))"
    (is "?A = ?B")
proof
  let ?occs = "(λx. occurs (Var x)) ` (fv_transaction T ∪ set (transaction_fresh T))"

  define frsh where "frsh ≡ transaction_fresh T"
  define xs where "xs ≡ filter (λx. x ∉ set frsh) (fv_list_sst (unlabel (transaction_strand T)))"
  define f where "f ≡ map (λx. occurs (Var x)::('a,'b,'c,'d) prot_term)"

  obtain A B C D E F where T: "T = Transaction A B C D E F" by (cases T) simp

  note 0 = add_occurs_msgs_transaction_strand_set(2,4)[
          OF T, unfolded f_def[symmetric] frsh_def[symmetric] xs_def[symmetric]]

  note 1 = add_occurs_msgs_transaction_strand_cases(1,2,3)[
          of T, unfolded f_def[symmetric] frsh_def[symmetric] xs_def[symmetric]]

  have 2: "set (f xs) ∪ set (f frsh) = ?occs"
  proof -
    define ys where "ys ≡ fv_list_sst (unlabel (transaction_strand T))"
    let ?ys' = "fv_transaction T - set frsh"
    define g where "g ≡ filter (λx. x ∉ set frsh)"
```

```
    have "set (g ys) = ?ys'"
      using fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"] unfolding ys_def g_def by auto
    hence "set (f (g ys)) = (λx. occurs (Var x)) ` ?ys'" unfolding f_def by force
    moreover have "set (f frsh) = (λx. occurs (Var x)) ` set frsh" unfolding f_def by force
    ultimately show ?thesis
      unfolding xs_def frsh_def[symmetric] ys_def[symmetric] g_def[symmetric] by blast
  qed

  have 3: "set (f []) = {}" unfolding f_def by blast

  have "trms_transaction (add_occurs_msgs T) ⊆ trms_transaction T ∪ set (f xs) ∪ set (f frsh)"
  proof (cases "∃ts F'. F = ⟨⋆, send⟨ts⟩⟩#F'")
    case True
    then obtain ts F' where F: "F = ⟨⋆, send⟨ts⟩⟩#F'" by blast
    have "set ts ⊆ trms_transaction T" unfolding T F trms_transaction_unfold by auto
    thus ?thesis using 0(1)[OF F] by force
  next
    case False show ?thesis using 0(2)[OF False] by force
  qed
  thus "?A ⊆ ?B" using 2 by blast

  have "trms_transaction T ∪ set (f xs) ∪ set (f frsh) ⊆ trms_transaction (add_occurs_msgs T)"
  proof (cases "frsh = []")
    case True show ?thesis using 1(1)[OF True] 3 unfolding True by (cases xs) (fastforce,force)
  next
    case False
    note * = 1(2-)[OF False]
    show ?thesis
    proof (cases "∃ts F'. transaction_send T = ⟨⋆, send⟨ts⟩⟩#F'")
      case True show ?thesis using *(2)[OF True] 3 by force
    next
      case False show ?thesis using *(1)[OF False] 3 by force
    qed
  qed
  thus "?B ⊆ ?A" using 2 by blast
qed

private lemma add_occurs_msgs_vars_eq:
  fixes T::"('fun,'var,'sets,'lbl) prot_transaction"
  assumes T_adm: "admissible_transaction' T"
  shows "fv_lsst (transaction_receive (add_occurs_msgs T)) =
         fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_checks T)" (is ?A)
    and "fv_lsst (transaction_send (add_occurs_msgs T)) =
         fv_lsst (transaction_send T) ∪ set (transaction_fresh T)" (is ?B)
    and "fv_transaction (add_occurs_msgs T) = fv_transaction T" (is ?C)
    and "bvars_transaction (add_occurs_msgs T) = bvars_transaction T" (is ?D)
    and "vars_transaction (add_occurs_msgs T) = vars_transaction T" (is ?E)
    and "fv_lsst (transaction_strand (add_occurs_msgs T) ·lsst ϑ) =
         fv_lsst (transaction_strand T ·lsst ϑ)" (is ?F)
    and "bvars_lsst (transaction_strand (add_occurs_msgs T) ·lsst ϑ) =
         bvars_lsst (transaction_strand T ·lsst ϑ)" (is ?G)
    and "vars_lsst (transaction_strand (add_occurs_msgs T) ·lsst ϑ) =
         vars_lsst (transaction_strand T ·lsst ϑ)" (is ?H)
    and "set (transaction_fresh (add_occurs_msgs T)) = set (transaction_fresh T)" (is ?I)
proof -
  obtain A B C D E F where T: "T = Transaction A B C D E F" by (cases T) simp

  have T_fresh: "set (transaction_fresh T) ⊆ fv_transaction T"
    using admissible_transactionE(7)[OF T_adm] unfolding fv_transaction_unfold by blast

  note 0 = add_occurs_msgs_cases[OF T]

  define xs where "xs ≡
```

```
    filter (λx. x ∉ set (transaction_fresh T)) (fv_list_sst (unlabel (transaction_strand T)))"

  show D: ?D
  proof -
    have "bvars_lsst (transaction_receive (add_occurs_msgs T)) = bvars_lsst (transaction_receive T)"
      using 0(6,7) by (cases "xs = []") (auto simp add: xs_def)
    moreover have "bvars_lsst (transaction_send (add_occurs_msgs T)) = bvars_lsst (transaction_send T)"
    proof (cases "∃ts' F'. F = ⟨⋆, send⟨ts'⟩⟩#F'")
      case True thus ?thesis using 0(1) unfolding T by force
    next
      case False show ?thesis using 0(2)[OF False] 0(3) unfolding T by (cases "B = []") auto
    qed
    ultimately show ?thesis using 0(8,9) unfolding bvars_transaction_unfold by argo
  qed

  have T_no_bvars:
      "bvars_transaction T = {}"
      "bvars_lsst (transaction_receive T) = {}"
      "bvars_lsst (transaction_checks T) = {}"
      "bvars_lsst (transaction_send T) = {}"
      "bvars_transaction (add_occurs_msgs T) = {}"
    using admissible_transactionE(4)[OF T_adm] D
    unfolding bvars_transaction_unfold by (blast,blast,blast,blast,blast)

  have T_fv_subst:
      "fv_lsst (transaction_strand T ·_lsst δ) = fv_set (δ ` fv_transaction T)" (is ?Q1)
      "fv_lsst (transaction_receive T ·_lsst δ) = fv_set (δ ` fv_lsst (transaction_receive T))" (is ?Q2)
      "fv_lsst (transaction_checks T ·_lsst δ) = fv_set (δ ` fv_lsst (transaction_checks T))" (is ?Q3)
      "fv_lsst (transaction_send T ·_lsst δ) = fv_set (δ ` fv_lsst (transaction_send T))" (is ?Q4)
      "fv_lsst (transaction_strand (add_occurs_msgs T) ·_lsst δ) =
       fv_set (δ ` fv_lsst (transaction_strand (add_occurs_msgs T)))" (is ?Q5)
      "fv_lsst (transaction_receive (add_occurs_msgs T) ·_lsst δ) =
       fv_set (δ ` fv_lsst (transaction_receive (add_occurs_msgs T)))" (is ?Q6)
    for δ
  proof -
    note * = fv_sst_subst_if_no_bvars

    have **: "bvars_lsst (transaction_receive (add_occurs_msgs T)) = {}"
      using T_no_bvars(5) unfolding bvars_transaction_unfold by fast

    show ?Q1 using *[OF T_no_bvars(1)] unfolding unlabel_subst by blast
    show ?Q2 using *[OF T_no_bvars(2)] unfolding unlabel_subst by blast
    show ?Q3 using *[OF T_no_bvars(3)] unfolding unlabel_subst by blast
    show ?Q4 using *[OF T_no_bvars(4)] unfolding unlabel_subst by blast
    show ?Q5 using *[OF T_no_bvars(5)] unfolding unlabel_subst by blast
    show ?Q6 using *[OF **] unfolding unlabel_subst by blast
  qed

  have A: "fv_lsst (transaction_receive (add_occurs_msgs T) ·_lsst δ) =
          fv_lsst (transaction_receive T ·_lsst δ) ∪ fv_lsst (transaction_checks T ·_lsst δ)"
    for δ
  proof -
    define rcv_trms where
      "rcv_trms ≡ map (λx. occurs (Var x)::('fun,'var,'sets,'lbl) prot_term) xs"

    have "fv_set (set rcv_trms) = fv_transaction T - set (transaction_fresh T)"
         "rcv_trms = [] ⟷ xs = []"
      using fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"]
      unfolding rcv_trms_def xs_def by auto
    hence 1: "fv_lsst (transaction_receive (add_occurs_msgs T)) =
             (fv_transaction T - set (transaction_fresh T)) ∪ fv_lsst (transaction_receive T)"
      using 0(6,7)[unfolded rcv_trms_def[symmetric] xs_def[symmetric]] by (cases "xs = []") auto
```

```
    have 2: "fv_lsst (transaction_receive T) ⊆ fv_transaction T - set (transaction_fresh T)"
      using admissible_transactionE(12)[OF T_adm] unfolding fv_transaction_unfold by fast

    have 3: "fv_transaction T - set (transaction_fresh T) =
             fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_checks T)"
      using admissible_transactionE(7,10,12,13)[OF T_adm]
      unfolding fv_transaction_unfold by blast

    show ?thesis using 1 2 3 T_fv_subst(2,3,6)[of δ] by force
  qed

  show ?A using A[of Var] unfolding subst_lsst_id_subst by blast

  show B: ?B using 0(14) by fastforce

  have B': "fv_lsst (transaction_send (add_occurs_msgs T) ·lsst δ) =
            fv_lsst (transaction_send T ·lsst δ) ∪ fv_set (δ ` set (transaction_fresh T))"
    for δ
  proof -
    note * = fv_sst_subst_if_no_bvars[of _ δ]

    have **: "bvars_lsst (transaction_send (add_occurs_msgs T)) = {}"
      using T_no_bvars(5) unfolding bvars_transaction_unfold by fast

    show ?thesis
      using B *[OF T_no_bvars(4)] *[OF **]
      unfolding unlabel_subst by simp
  qed

  show C: ?C
    using A[of Var] B T_fresh
    unfolding fv_transaction_unfold 0(8,9) subst_lsst_id_subst by blast

  show ?E using C D vars_sst_is_fv_sst_bvars_sst by metis

  have "fv_set (ϑ ` set (transaction_fresh T)) ⊆ fv_lsst (transaction_strand T ·lsst ϑ)"
    using T_fresh
    unfolding fv_sst_subst_if_no_bvars[OF T_no_bvars(1), of ϑ, unfolded unlabel_subst]
    by auto
  thus F: ?F
    using A[of ϑ] B'[of ϑ] fv_sst_append
          fv_sst_subst_if_no_bvars[OF T_no_bvars(1), of ϑ, unfolded unlabel_subst]
          fv_sst_subst_if_no_bvars[OF T_no_bvars(5), of ϑ, unfolded unlabel_subst C]
    unfolding transaction_strand_def by argo

  show G: ?G using D bvars_sst_subst unlabel_subst by metis

  show ?H using F G vars_sst_is_fv_sst_bvars_sst by metis

  show ?I using 0(5) by argo
qed

private lemma add_occurs_msgs_trms:
  "trms_transaction (add_occurs_msgs T) =
    trms_transaction T ∪ (λx. occurs (Var x)) ` (set (transaction_fresh T) ∪ fv_transaction T)"
proof -
  let ?f = "λx. occurs (Var x)"
  let ?xs = "filter (λx. x ∉ set (transaction_fresh T))
                    (fv_list_sst (unlabel (transaction_strand T)))"

  obtain A B C D E F where T: "T = Transaction A B C D E F" by (cases T) simp

  note 0 = add_occurs_msgs_cases[OF T]
```

```
  have "set ?xs = fv_transaction T - set (transaction_fresh T)"
    using fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"] by auto
  hence 1: "trms_lsst (transaction_receive (add_occurs_msgs T)) =
          trms_lsst (transaction_receive T) ∪ ?f ` (fv_transaction T - set (transaction_fresh T))"
    using 0(6,7) by (cases "?xs = []") auto

  have 2: "trms_lsst (transaction_send (add_occurs_msgs T)) =
          trms_lsst (transaction_send T) ∪ ?f ` set (transaction_fresh T)"
    using 0(10,11,12) by (cases "transaction_fresh T = []") (simp,fastforce)

  have 3: "trms_lsst (transaction_receive (add_occurs_msgs T)) ∪
          trms_lsst (transaction_send (add_occurs_msgs T)) =
          trms_lsst (transaction_receive T) ∪ trms_lsst (transaction_send T) ∪
          ?f ` (set (transaction_fresh T) ∪ fv_transaction T)"
    using 1 2 by blast

  show ?thesis using 3 unfolding trms_transaction_unfold 0(8,9) by blast
qed

lemma add_occurs_msgs_admissible_occurs_checks:
  fixes T::"('fun,'atom,'sets,'lbl) prot_transaction"
  assumes T_adm: "admissible_transaction' T"
  shows "admissible_transaction' (add_occurs_msgs T)" (is ?A)
    and "admissible_transaction_occurs_checks (add_occurs_msgs T)" (is ?B)
proof -
  let ?T' = "add_occurs_msgs T"

  obtain A B C D E F where T: "T = Transaction A B C D E F" by (cases T) simp

  note defs = T add_occurs_msgs_def Let_def admissible_transaction'_def
            admissible_transaction_occurs_checks_def

  note defs' = admissible_transaction_terms_def wf_trms_code[symmetric]

  note 1 = add_occurs_msgs_cases[OF T]
  note 2 = add_occurs_msgs_vars_eq[OF T_adm]
  note 3 = add_occurs_msgs_trms[of T]
  note 4 = add_occurs_msgs_transaction_strand_set[OF T]

  have occurs_wf: "wf_trm (occurs (Var x))" for x::"('fun,'atom,'sets,'lbl) prot_var" by fastforce

  have occurs_funs: "funs_term (occurs (Var x)) = {OccursFact, OccursSec}"
    for x::"('fun,'atom,'sets,'lbl) prot_var"
    by force

  have occurs_funs_not_attack: "¬(∃f ∈ ⋃(funs_term ` trms_sstp r). is_Attack f)"
    when "r = receive⟨map (λx. occurs (Var x)) xs⟩ ∨ r = send⟨map (λx. occurs (Var x)) ys⟩"
    for r::
      "(('fun,'atom,'sets,'lbl) prot_fun, ('fun,'atom,'sets,'lbl) prot_var) stateful_strand_step"
    and xs ys::"('fun,'atom,'sets,'lbl) prot_var list"
    using that by fastforce

  have occurs_funs_not_attack': "¬(∃f ∈ ⋃(funs_term ` trms_sstp r). is_Attack f)"
    when "r = send⟨map (λx. occurs (Var x)) xs@ts⟩"
    and "¬(∃f ∈ ⋃(funs_term ` trms_sstp (send⟨ts⟩)). is_Attack f)"
    for r::
      "(('fun,'atom,'sets,'lbl) prot_fun, ('fun,'atom,'sets,'lbl) prot_var) stateful_strand_step"
    and xs::"('fun,'atom,'sets,'lbl) prot_var list"
    and ts
    using that by fastforce

  let ?P1 = "λT. wellformed_transaction T"
```

```
let ?P2 = "λT. transaction_decl T () = []"
let ?P3 = "λT. list_all (λx. fst x = TAtom Value) (transaction_fresh T)"
let ?P4 = "λT. ∀x ∈ vars_transaction T. is_Var (fst x) ∧ (the_Var (fst x) = Value)"
let ?P5 = "λT. bvars_lsst (transaction_strand T) = {}"
let ?P6 = "λT. set (transaction_fresh T) ⊆
                fv_lsst (filter (is_Insert ∘ snd) (transaction_updates T)) ∪
                fv_lsst (transaction_send T)"
let ?P7 = "λT. ∀x ∈ fv_transaction T - set (transaction_fresh T).
               ∀y ∈ fv_transaction T - set (transaction_fresh T).
                 x ≠ y ⟶ ⟨Var x != Var y⟩ ∈ set (unlabel (transaction_checks T)) ∨
                          ⟨Var y != Var x⟩ ∈ set (unlabel (transaction_checks T))"
let ?P8 = "λT. fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)
                - set (transaction_fresh T)
                ⊆ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_checks T)"
let ?P9 = "λT. ∀r ∈ set (unlabel (transaction_checks T)).
                is_Equality r ⟶ fv (the_rhs r) ⊆ fv_lsst (transaction_receive T)"
let ?P10 = "λT. fv_lsst (transaction_checks T) ⊆
                  fv_lsst (transaction_receive T) ∪
                  fv_lsst (filter (λs. is_InSet (snd s) ∧ the_check (snd s) = Assign)
                                 (transaction_checks T))"
let ?P11 = "λT. admissible_transaction_checks T"
let ?P12 = "λT. admissible_transaction_updates T"
let ?P13 = "λT. admissible_transaction_terms T"
let ?P14 = "λT. admissible_transaction_send_occurs_form T"
let ?P15 = "λT. list_all (λa. is_Receive (snd a) ⟶ the_msgs (snd a) ≠ [])
                         (transaction_receive T)"
let ?P16 = "λT. list_all (λa. is_Send (snd a) ⟶ the_msgs (snd a) ≠ []) (transaction_send T)"

have T_props:
    "?P1 T" "?P2 T" "?P3 T" "?P4 T" "?P5 T" "?P6 T" "?P7 T" "?P8 T" "?P9 T" "?P10 T" "?P11 T"
    "?P12 T" "?P13 T" "?P14 T" "?P15 T" "?P16 T"
  using T_adm unfolding defs by meson+

have 5: "wf'_sst (X ∪ Y) (unlabel (dual_lsst (transaction_strand (add_occurs_msgs T))))"
  when X: "X = fst ` set (transaction_decl T ())"
    and Y: "Y = set (transaction_fresh T)"
    and T_wf: "wf'_sst (X ∪ Y) (unlabel (dual_lsst (transaction_strand T)))"
  for X Y
proof -
  define frsh where "frsh ≡ transaction_fresh T"
  define xs where "xs ≡ fv_list_sst (unlabel (transaction_strand T))"
  define ys where "ys ≡ filter (λx. x ∉ set frsh) xs"

  let ?snds = "unlabel (dual_lsst (transaction_receive T))"
  let ?snds' = "unlabel (dual_lsst (transaction_receive (add_occurs_msgs T)))"
  let ?chks = "unlabel (dual_lsst (transaction_checks T))"
  let ?chks' = "unlabel (dual_lsst (transaction_checks (add_occurs_msgs T)))"
  let ?upds = "unlabel (dual_lsst (transaction_updates T))"
  let ?upds' = "unlabel (dual_lsst (transaction_updates (add_occurs_msgs T)))"
  let ?rcvs = "unlabel (dual_lsst (transaction_send T))"
  let ?rcvs' = "unlabel (dual_lsst (transaction_send (add_occurs_msgs T)))"

  have p0: "set ?snds ⊆ set ?snds'" using 1(13) by auto

  have p1: "?chks = ?chks'" "?upds = ?upds'" using 1(8,9) by (argo,argo)

  have p2: "wfvarsoccs_sst ?snds ⊆ wfvarsoccs_sst ?snds'"
           "wfvarsoccs_sst (?snds@?chks@?upds) ⊆ wfvarsoccs_sst (?snds'@?chks'@?upds')"
           "X ∪ Y ∪ wfvarsoccs_sst (?snds@?chks@?upds) ⊆
            X ∪ Y ∪  wfvarsoccs_sst (?snds'@?chks'@?upds')"
    using p0 p1 unfolding wfvarsoccs_sst_def by auto

  have "wf'_sst (X ∪ Y ∪ wfvarsoccs_sst (?snds@?chks@?upds)) ?rcvs"
```

```
      using T_wf wf_sst_append_exec[of "X ∪ Y" "?snds@?chks@?upds" ?rcvs]
      unfolding transaction_strand_unlabel_dual_unfold by simp
    hence r0: "wf'_sst (X ∪ Y ∪ wfvarsoccs_sst (?snds'@?chks'@?upds')) ?rcvs"
      using wf_sst_vars_mono'[OF _ p2(3)] by blast

    have "list_all is_Send (unlabel (transaction_send T))"
      using admissible_transaction_is_wellformed_transaction(1)[OF T_adm]
      unfolding wellformed_transaction_def by blast
    hence "list_all is_Receive ?rcvs" by (metis dual_lsst_list_all(2))
    hence r1: "wfrestrictedvars_sst ?rcvs ⊆ X ∪ Y ∪ wfvarsoccs_sst (?snds'@?chks'@?upds')"
      using wfrestrictedvars_sst_receives_only_eq wf_sst_receives_only_fv_subset[OF r0] by blast

    have "fv_set ((λx. occurs (Var x)) ` set (transaction_fresh T)) ⊆ Y"
      unfolding Y by auto
    hence r2: "wfrestrictedvars_sst ?rcvs' ⊆ X ∪ Y ∪ wfvarsoccs_sst (?snds'@?chks'@?upds')"
      using 1(14) r1 unfolding wfrestrictedvars_sst_def by fastforce

    have r3: "wf'_sst (X ∪ Y) (?snds'@?chks'@?upds')"
    proof -
      have *: "wf'_sst (X ∪ Y) (?snds@?chks'@?upds')"
        using T_wf wf_sst_prefix[of "X ∪ Y" "?snds@?chks@?upds" ?rcvs] p1
        unfolding transaction_strand_unlabel_dual_unfold by simp

      have "?snds' = ?snds ∨ (∃ts. ?snds' = send⟨ts⟩#?snds)" using 1(13) by auto
      thus ?thesis
      proof
        assume "?snds' = ?snds" thus ?thesis using * by simp
      next
        assume "∃ts. ?snds' = send⟨ts⟩#?snds"
        then obtain ts where "?snds' = send⟨ts⟩#?snds" by blast
        thus ?thesis using wf_sst_sends_only_prepend[OF *, of "[send⟨ts⟩]"] by simp
      qed
    qed

    have "wf'_sst (X ∪ Y) (?snds'@?chks'@?upds'@?rcvs')"
      using wf_sst_append_suffix''[OF r3] r2 by auto
    thus ?thesis
      using unlabel_append dual_lsst_append
      unfolding transaction_strand_def by auto
  qed

  have T'_props_1: "?P1 ?T'"
    unfolding wellformed_transaction_def
    apply (intro conjI)
    subgoal using 1(13) T_props(1) unfolding wellformed_transaction_def by force
    subgoal using 1(8) T_props(1) unfolding wellformed_transaction_def by simp
    subgoal using 1(9) T_props(1) unfolding wellformed_transaction_def by simp
    subgoal using 1(14) T_props(1) unfolding wellformed_transaction_def by force
    subgoal using 1(4) T_props(1) unfolding wellformed_transaction_def by simp
    subgoal using 1(5) T_props(1) unfolding wellformed_transaction_def by simp
    subgoal using 1(4,5) T_props(1) unfolding wellformed_transaction_def by simp
    subgoal using T_props(1) unfolding 2(1) 1(5) wellformed_transaction_def by blast
    subgoal using 1(5,8) T_props(1) unfolding wellformed_transaction_def by simp
    subgoal using 1(5) 2(4) T_props(1) unfolding wellformed_transaction_def by simp
    subgoal using 2(3,4) T_props(1) unfolding wellformed_transaction_def by simp
    subgoal using 1(4,5) 5 T_props(1) unfolding wellformed_transaction_def by simp
    done

  have T'_props_2_12:
      "?P2 ?T'" "?P3 ?T'" "?P4 ?T'" "?P5 ?T'" "?P6 ?T'" "?P7 ?T'" "?P8 ?T'" "?P9 ?T'" "?P10 ?T'"
      "?P11 ?T'" "?P12 ?T'"
    subgoal using T_props(2) unfolding defs by force
    subgoal using T_props(3) unfolding defs by force
```

```
  subgoal using T_props(4) 2(5) by argo
  subgoal using T_props(5) 2(4) by argo
  subgoal using T_props(6) 1(5,8) 2(2) by auto
  subgoal using T_props(7) 1(5,8) 2(3) by presburger
  subgoal using T_props(8) 1(5,9) 2(1,2) by auto
  subgoal using T_props(9) 1(8) 2(1) by auto
  subgoal using T_props(10) 1(8) 2(1) by auto
  subgoal using T_props(11) 1(8) unfolding admissible_transaction_checks_def by argo
  subgoal using T_props(12) 1(9) unfolding admissible_transaction_updates_def by argo
  done


have T'_props_13_aux:
    "transaction_fresh ?T' = []" (is ?Q1)
    "is_Send r" (is ?Q2)
    "length (the_msgs r) = 1" (is ?Q3)
    "is_Fun_Attack (hd (the_msgs r))" (is ?Q4)
  when r: "r ∈ set (unlabel (transaction_strand (add_occurs_msgs T)))"
          "∃f ∈ ⋃(funs_term ` trms_sstp r). is_Attack f" (is "?Q' (trms_sstp r)")
  for r
proof -
  note q0 = conjunct2[OF conjunct2[OF T_props(13)[unfolded defs']]]

  let ?Q'' = "λts' F'. F = ⟨⋆, send⟨ts'⟩⟩#F'"
  let ?f = "map (λx. occurs (Var x))"
  let ?frsh = "transaction_fresh T"
  let ?xs = "fv_list_sst (unlabel (transaction_strand T))"

  have q1: "r ≠ send⟨?f ?frsh⟩" "r ≠ receive⟨?f (filter (λx. x ∉ set ?frsh) ?xs)⟩"
           "∀f ∈ ⋃(funs_term ` set (?f ?frsh)). ¬is_Attack f"
    using r(2) by (fastforce,fastforce,simp)

  have q2: "send⟨ts'⟩ ∈ set (unlabel (transaction_strand T))"
           "r = send⟨?f ?frsh@ts'⟩ ∨ r ∈ set (unlabel (transaction_strand T))"
    when "?Q'' ts' F'" for ts' F'
    subgoal using that unfolding T transaction_strand_def by force
    subgoal using that r(1) 4(2)[OF that] q1 unfolding T transaction_strand_def by fast
    done

  have q3: "?Q' (set ts')"
    when r': "?Q'' ts' F'" "r ∉ set (unlabel (transaction_strand T))" for ts' F'
  proof -
    have "r = send⟨?f ?frsh@ts'⟩" using q2(2)[OF r'(1)] r'(2) by argo
    thus ?thesis using r(2) by fastforce
  qed

  have q4: "r ∈ set (unlabel (transaction_strand T))" when "∄ts' F'. ?Q'' ts' F'"
    using 4(4)[OF that] r(1) q1(1,2) by blast

  have "∃r' ∈ set (unlabel (transaction_strand T)). ?Q' (trms_sstp r')"
    when "?Q'' ts' F'" for ts' F'
    apply (cases "r ∈ set (unlabel (transaction_strand T))")
    subgoal using q2(2)[OF that] r(2) by metis
    subgoal using q2(1)[OF that] q3[OF that] trms_sstp.simps(1)[of ts'] by metis
    done
  hence "?frsh = []" when "?Q'' ts' F'" for ts' F' using q0 that by blast
  hence "r = send⟨ts'⟩ ∨ r ∈ set (unlabel (transaction_strand T))" when "?Q'' ts' F'" for ts' F'
    using q2(2)[OF that] that by blast
  hence "r ∈ set (unlabel (transaction_strand T))" using q2(1) q4 by fast
  thus ?Q1 ?Q2 ?Q3 ?Q4 using r(2) q0 unfolding 1(5) by auto
qed

have T'_props_13: "?P13 ?T'"
```

```
    unfolding defs' 3
    apply (intro conjI)
    subgoal using conjunct1[OF T_props(13)[unfolded defs']] occurs_wf by fast
    subgoal using conjunct1[OF conjunct2[OF T_props(13)[unfolded defs']]] occurs_funs by auto
    subgoal using T'_props_13_aux by meson
    done

  have T'_props_14: "?P14 ?T'"
  proof (cases "∃ts' F'. transaction_send T = ⟨⋆,send⟨ts'⟩⟩#F'")
    case True
    then obtain ts' F' where F': "transaction_send T = ⟨⋆,send⟨ts'⟩⟩#F'" by meson
    show ?thesis
      using T_props(14) 1(10)[OF F'] F' 1(5,12)
      unfolding admissible_transaction_send_occurs_form_def Let_def
      by (cases "transaction_fresh T = []") auto
  next
    case False show ?thesis
      using T_props(14) 1(11)[OF False] 1(5,12)
      unfolding admissible_transaction_send_occurs_form_def Let_def
      by (cases "transaction_fresh T = []") auto
  qed

  let ?xs = "fv_list_sst (unlabel (transaction_strand T))"

  have T'_props_15: "?P15 ?T'"
    using T_props(15) 1(6,7) unfolding Let_def
    by (cases "filter (λx. x ∉ set (transaction_fresh T)) ?xs = []") (simp,fastforce)

  have T'_props_16: "?P16 ?T'"
  proof (cases "∃ts' F'. transaction_send T = ⟨⋆,send⟨ts'⟩⟩#F'")
    case True
    then obtain ts' F' where F': "transaction_send T = ⟨⋆,send⟨ts'⟩⟩#F'" by meson
    show ?thesis
      using T_props(16) 1(10)[OF F'] F' 1(5,12)
      unfolding Let_def by (cases "transaction_fresh T = []") auto
  next
    case False show ?thesis
      using T_props(16) 1(11)[OF False] 1(5,12)
      unfolding Let_def by (cases "transaction_fresh T = []") auto
  qed

  note T'_props = T'_props_1 T'_props_2_12 T'_props_13 T'_props_14 T'_props_15 T'_props_16

  show ?A using T'_props unfolding admissible_transaction'_def by meson

  have 5: "set (filter (λx. x ∉ set (transaction_fresh T))
                     (fv_list_sst (unlabel (transaction_strand T)))) =
          fv_transaction T - set (transaction_fresh T)"
    using fv_list_sst_is_fv_sst by fastforce

  have "transaction_receive ?T' ≠ []"
    and "is_Receive (hd (unlabel (transaction_receive ?T')))"
    and "∀x ∈ fv_transaction ?T' - set (transaction_fresh ?T'). fst x = TAtom Value ⟶
           occurs (Var x) ∈ set (the_msgs (hd (unlabel (transaction_receive ?T'))))"
    when x: "x ∈ fv_transaction ?T' - set (transaction_fresh ?T')" "fst x = TAtom Value" for x
    using 1(13) 5 x unfolding 1(5) 2(3) by (force,force,force)
  moreover have "transaction_send ?T' ≠ []" (is ?C)
    and "is_Send (hd (unlabel (transaction_send ?T')))" (is ?D)
    and "∀x ∈ set (transaction_fresh ?T').
          occurs (Var x) ∈ set (the_msgs (hd (unlabel (transaction_send ?T'))))" (is ?E)
    when T'_frsh: "transaction_fresh ?T' ≠ []"
    using 1(14) T'_frsh unfolding 1(5) by auto
  ultimately show ?B
```

```
    using T'_props_14 unfolding admissible_transaction_occurs_checks_def Let_def by blast
qed

private lemma add_occurs_msgs_in_trms_subst_cases:
  fixes T::"('fun,'atom,'sets,'lbl) prot_transaction"
  assumes T_adm: "admissible_transaction' T"
    and t: "t ∈ trms_lsst (transaction_strand (add_occurs_msgs T) ·_lsst ϑ)"
  shows "t ∈ trms_lsst (transaction_strand T ·_lsst ϑ) ∨
         (∃x ∈ fv_transaction T. t = occurs (ϑ x))"
proof -
  define frsh where "frsh ≡ transaction_fresh T"
  define xs where "xs ≡ filter (λx. x ∉ set frsh) (fv_list_sst (unlabel (transaction_strand T)))"
  define f where "f ≡ map (λx. occurs (Var x)::('fun,'atom,'sets,'lbl) prot_term)"

  obtain A B C D E F where T: "T = Transaction A B C D E F" by (cases T) simp

  note T'_adm = add_occurs_msgs_admissible_occurs_checks(1)[OF T_adm]

  have 0: "set (transaction_fresh T) ⊆ fv_transaction T"
    using admissible_transactionE(7)[OF T_adm]
    unfolding fv_transaction_unfold by blast
  hence 00: "set (f xs) ∪ set (f frsh) = (λx. occurs (Var x)) ` fv_transaction T"
    using fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"]
    unfolding f_def xs_def frsh_def by auto

  note 1 = add_occurs_msgs_transaction_strand_set[OF T]

  have 2: "set (transaction_strand (add_occurs_msgs T)) ⊆
           set (transaction_strand T) ∪ {⟨⋆,receive⟨f xs⟩⟩,⟨⋆,send⟨f frsh⟩⟩}"
    when "∄ts F'. F = ⟨⋆,send⟨ts⟩⟩#F'"
    using 1(3,4)[OF that] unfolding f_def[symmetric] frsh_def[symmetric] xs_def[symmetric] by blast

  have 3: "trms_transaction (add_occurs_msgs T) =
           trms_transaction T ∪ (λx. occurs (Var x)) ` fv_transaction T"
    using 0 add_occurs_msgs_trms_transaction[of T] by blast

  have 4: "bvars_transaction T ∩ subst_domain ϑ = {}"
          "bvars_transaction (add_occurs_msgs T) ∩ subst_domain ϑ = {}"
    using admissible_transactionE(4)[OF T_adm] admissible_transactionE(4)[OF T'_adm]
    by (blast,blast)

  note 5 = trms_sst_subst[OF 4(1), unfolded unlabel_subst]
           trms_sst_subst[OF 4(2), unfolded unlabel_subst]

  note 6 = fv_sst_is_subterm_trms_sst_subst[
            OF _ 4(1), unfolded add_occurs_msgs_admissible_occurs_checks(1)[OF T_adm] unlabel_subst]

  show ?thesis
    using t 6 unfolding 3 5 by fastforce
qed

private lemma add_occurs_msgs_updates_send_filter_iff:
  fixes f
  defines "f ≡ λT. list_ex (λa. is_Send (snd a) ∨ is_Update (snd a)) (transaction_strand T)"
    and "g ≡ λT. transaction_fresh T = [] ⟶ f T"
  shows "map add_occurs_msgs (filter g P) = filter g (map add_occurs_msgs P)"
proof -
  have "g T ⟷ g (add_occurs_msgs T)" for T
  proof -
    obtain A B C D E F where T: "T = Transaction A B C D E F" by (cases T) simp_all
    note 0 = add_occurs_msgs_cases[OF T]
    show ?thesis using 0(6,7,12) unfolding g_def f_def transaction_strand_def 0(5,8,9) by fastforce
  qed
```

```
  thus ?thesis by (induct P) simp_all
qed

lemma add_occurs_msgs_updates_send_filter_iff':
  fixes f
  defines "f ≡ λT. list_ex (λa. is_Send (snd a) ∨ is_Update (snd a)) (transaction_strand T)"
    and "g ≡ λT. transaction_fresh T = [] ⟶ transaction_updates T ≠ [] ∨ transaction_send T ≠ []"
  shows "map add_occurs_msgs (filter g P) = filter g (map add_occurs_msgs P)"
proof -
  have "g T ⟷ g (add_occurs_msgs T)" for T
  proof -
    obtain A B C D E F where T: "T = Transaction A B C D E F" by (cases T) simp_all
    note 0 = add_occurs_msgs_cases[OF T]
    show ?thesis using 0(6,7,12) unfolding g_def f_def transaction_strand_def 0(5,8,9) by argo
  qed
  thus ?thesis by (induct P) simp_all
qed

private lemma rm_occurs_msgs_constr_Cons:
  defines "f ≡ rm_occurs_msgs_constr"
  shows
    "¬is_Receive a ⟹ ¬is_Send a ⟹ f ((l,a)#A) = (l,a)#f A"
    "is_Receive a ⟹ ∄t. occurs t ∈ set (the_msgs a) ⟹ f ((l,a)#A) = (l,a)#f A"
    "is_Receive a ⟹ ∃t. occurs t ∈ set (the_msgs a) ⟹
      ∃t ∈ set (the_msgs a). ∀s. t ≠ occurs s ⟹
      f ((l,a)#A) = (l,receive⟨filter (λt. ∀s. t ≠ occurs s) (the_msgs a)⟩)#f A"
    "is_Receive a ⟹ ∃t. occurs t ∈ set (the_msgs a) ⟹
      ∀t ∈ set (the_msgs a). ∃s. t = occurs s ⟹ f ((l,a)#A) = f A"
    "is_Send a ⟹ ∄t. occurs t ∈ set (the_msgs a) ⟹ f ((l,a)#A) = (l,a)#f A"
    "is_Send a ⟹ ∃t. occurs t ∈ set (the_msgs a) ⟹
      ∃t ∈ set (the_msgs a). ∀s. t ≠ occurs s ⟹
      f ((l,a)#A) = (l,send⟨filter (λt. ∀s. t ≠ occurs s) (the_msgs a)⟩)#f A"
    "is_Send a ⟹ ∃t. occurs t ∈ set (the_msgs a) ⟹
      ∀t ∈ set (the_msgs a). ∃s. t = occurs s ⟹ f ((l,a)#A) = f A"
unfolding f_def by (cases a; auto)+

private lemma rm_occurs_msgs_constr_Cons':
  defines "f ≡ rm_occurs_msgs_constr"
    and "g ≡ filter (λt. ∀s. t ≠ occurs s)"
  assumes a: "is_Receive a ∨ is_Send a"
  shows
    "∄t. occurs t ∈ set (the_msgs a) ⟹ f ((l,a)#A) = (l,a)#f A"
    "∃t. occurs t ∈ set (the_msgs a) ⟹
      ∃t ∈ set (the_msgs a). ∀s. t ≠ occurs s ⟹
      is_Send a ⟹ f ((l,a)#A) = (l,send⟨g (the_msgs a)⟩)#f A"
    "∃t. occurs t ∈ set (the_msgs a) ⟹
      ∃t ∈ set (the_msgs a). ∀s. t ≠ occurs s ⟹
      is_Receive a ⟹ f ((l,a)#A) = (l,receive⟨g (the_msgs a)⟩)#f A"
    "∃t. occurs t ∈ set (the_msgs a) ⟹
      ∀t ∈ set (the_msgs a). ∃s. t = occurs s ⟹ f ((l,a)#A) = f A"
using a unfolding f_def g_def by (cases a; auto)+

private lemma rm_occurs_msgs_constr_Cons'':
  defines "f ≡ rm_occurs_msgs_constr"
    and "g ≡ filter (λt. ∀s. t ≠ occurs s)"
  assumes a: "a = receive⟨ts⟩ ∨ a = send⟨ts⟩"
  shows "f ((l,a)#A) = (l,a)#f A ∨ f ((l,a)#A) = (l,receive⟨g ts⟩)#f A ∨
         f ((l,a)#A) = (l,send⟨g ts⟩)#f A ∨ f ((l,a)#A) = f A"
using rm_occurs_msgs_constr_Cons(2-)[of a l A] a unfolding f_def g_def by (cases a) auto

private lemma rm_occurs_msgs_constr_ik_subset:
  "ik_lsst (rm_occurs_msgs_constr A) ⊆ ik_lsst A"
proof (induction A)
```

```
case (Cons a A)
let ?f = "filter (λt. ∀s. t ≠ occurs s)"

note IH = Cons.IH

obtain l b where a: "a = (l,b)" by (metis surj_pair)

have 0: "set (unlabel A) ⊆ set (unlabel (a#A))" by auto

note 1 = rm_occurs_msgs_constr_Cons[of b l A]
note 2 = in_ik_lsst_iff
note 3 = ik_sst_set_subset[OF 0]
note 4 = ik_sst_append
note 5 = 4[of "unlabel [a]" "unlabel A"] 4[of "unlabel [a]" "unlabel (rm_occurs_msgs_constr A)"]

show ?case
proof (cases "is_Send b ∨ is_Receive b")
  case True
  note b_cases = this

  define ts where "ts ≡ the_msgs b"

  have ts_cases: "is_Send b ⟹ b = send⟨ts⟩" "is_Receive b ⟹ b = receive⟨ts⟩"
    unfolding ts_def by simp_all

  have 6:
      "is_Send b ⟹ ik_lsst [(l,b)] = {}"
      "is_Send b ⟹ ik_lsst [(l,send⟨the_msgs b⟩)] = {}"
      "is_Send b ⟹ ik_lsst [(l,send⟨?f (the_msgs b)⟩)] = {}"
      "is_Receive b ⟹ ik_lsst [(l,b)] = set ts"
      "is_Receive b ⟹ ik_lsst [(l,receive⟨the_msgs b⟩)] = set ts"
      "is_Receive b ⟹ ik_lsst [(l,receive⟨?f (the_msgs b)⟩)] = set (?f ts)"
    using 2[of _ "[(l, send⟨the_msgs b⟩)]"]
          2[of _ "[(l, send⟨?f (the_msgs b)⟩)]"]
          2[of _ "[(l, receive⟨the_msgs b⟩)]"]
          2[of _ "[(l, receive⟨?f (the_msgs b)⟩)]"]
          b_cases ts_cases
    by auto

  have "ik_lsst (rm_occurs_msgs_constr (a#A)) = ik_lsst (rm_occurs_msgs_constr A)"
    when b: "is_Send b"
  proof (cases "∃t. occurs t ∈ set (the_msgs b)")
    case True
    note 7 = 1(6,7)[OF b True]

    show ?thesis
    proof (cases "∃t ∈ set (the_msgs b). ∀s. t ≠ occurs s")
      case True show ?thesis
        using 4[of "unlabel [(l,send⟨?f (the_msgs b)⟩)]"
                   "unlabel (rm_occurs_msgs_constr A)"]
        unfolding a 7(1)[OF True] 6(3)[OF b] by simp
    next
      case False
      hence F: "∀t ∈ set (the_msgs b). ∃s. t = occurs s" by simp
      show ?thesis
        using 4[of "unlabel [(l,send⟨the_msgs b⟩)]" "unlabel (rm_occurs_msgs_constr A)"]
        unfolding a 7(2)[OF F] 6(2)[OF b] by simp
    qed
  next
    case False show ?thesis
      using 4[of "unlabel [(l,b)]" "unlabel (rm_occurs_msgs_constr A)"]
      unfolding a 1(5)[OF b False] 6(1)[OF b] by auto
  qed
```

```
    moreover have "ik_lsst (rm_occurs_msgs_constr (a#A)) ⊆ set ts ∪ ik_lsst (rm_occurs_msgs_constr A)"
      when b: "is_Receive b"
    proof (cases "∃t. occurs t ∈ set (the_msgs b)")
      case True
      note 8 = 1(3,4)[OF b True]

      show ?thesis
      proof (cases "∃t ∈ set (the_msgs b). ∀s. t ≠ occurs s")
        case True show ?thesis
          using 4[of "unlabel [(l,receive⟨?f (the_msgs b)⟩)]"
                     "unlabel (rm_occurs_msgs_constr A)"]
          unfolding a 8(1)[OF True] 6(6)[OF b] by auto
      next
        case False
        hence F: "∀t ∈ set (the_msgs b). ∃s. t = occurs s" by simp
        show ?thesis
          using 4[of "unlabel [(l,receive⟨the_msgs b⟩)]" "unlabel (rm_occurs_msgs_constr A)"]
          unfolding a 8(2)[OF F] 6(5)[OF b] by simp
      qed
    next
      case False show ?thesis
        using 4[of "unlabel [(l,b)]" "unlabel (rm_occurs_msgs_constr A)"]
        unfolding a 1(2)[OF b False] 6(4)[OF b] by auto
    qed
    moreover have "ik_lsst (a#A) = set ts ∪ ik_lsst A" when b: "is_Receive b"
      using ik_lsst_Cons(2)[of l ts A] unfolding a ts_cases(2)[OF b] by blast
    ultimately show ?thesis using IH 3 b_cases by blast
  qed (use 1(1) IH 5 a in auto)
qed simp

private lemma rm_occurs_msgs_constr_append:
  "rm_occurs_msgs_constr (A@B) = rm_occurs_msgs_constr A@rm_occurs_msgs_constr B"
by (induction A rule: rm_occurs_msgs_constr.induct) auto

private lemma rm_occurs_msgs_constr_dual_lsst:
  "rm_occurs_msgs_constr (dual_lsst A) = dual_lsst (rm_occurs_msgs_constr A)"
proof (induction A)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  show ?case using Cons.IH unfolding a by (cases b) auto
qed simp

private lemma rm_occurs_msgs_constr_dbupd_sst_eq:
  "dbupd_sst (unlabel (rm_occurs_msgs_constr A)) I D = dbupd_sst (unlabel A) I D"
proof (induction A arbitrary: I D)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  show ?case
  proof (cases "is_Receive b ∨ is_Send b")
    case True
    then obtain ts where b: "b = receive⟨ts⟩ ∨ b = send⟨ts⟩" by (cases b) simp_all
    show ?thesis using rm_occurs_msgs_constr_Cons''[OF b, of l A] Cons.IH b unfolding a by fastforce
  next
    case False thus ?thesis using Cons.IH unfolding a by (cases b) auto
  qed
qed simp

private lemma rm_occurs_msgs_constr_subst:
  fixes A::"('a,'b,'c,'d) prot_strand" and ϑ::"('a,'b,'c,'d) prot_subst"
  assumes "∀x ∈ fv_lsst A. ∄t. ϑ x = occurs t" "∀x ∈ fv_lsst A. ϑ x ≠ Fun OccursSec []"
  shows "rm_occurs_msgs_constr (A ·_lsst ϑ) = (rm_occurs_msgs_constr A) ·_lsst ϑ"
    (is "?f (A ·_lsst ϑ) = (?f A) ·_lsst ϑ")
using assms
```

```
proof (induction A)
  case (Cons a A)
  note 0 = rm_occurs_msgs_constr_Cons
  note 1 = rm_occurs_msgs_constr_Cons'

  define f where "f ≡ ?f"
  define not_occ where "not_occ ≡ λt::('a,'b,'c,'d) prot_term. ∀s. t ≠ occurs s"
  define flt where "flt ≡ filter not_occ"

  obtain l b where a: "a = (l,b)" by (metis surj_pair)

  have 2: "∄t. ϑ x = occurs t" "ϑ x ≠ Fun OccursSec []"
    when b: "is_Receive b ∨ is_Send b" and t: "t ∈ set (the_msgs b)" and x: "x ∈ fv t" for x t
    using Cons.prems x t b unfolding a by (cases b; auto)+

  have IH: "f (A ·lsst ϑ) = (f A) ·lsst ϑ"
    using Cons.prems Cons.IH unfolding f_def by simp

  show ?case
  proof (cases "is_Receive b ∨ is_Send b")
    case True
    note T = this
    then obtain ts where ts: "b = receive⟨ts⟩ ∨ b = send⟨ts⟩" by (cases b) simp_all
    hence ts': "b ·sstp ϑ = receive⟨ts ·list ϑ⟩ ∨ b ·sstp ϑ = send⟨ts ·list ϑ⟩" by auto

    have the_msgs_b: "the_msgs b = ts" "the_msgs (b ·sstp ϑ) = ts ·list ϑ"
      using ts ts' by auto

    have 4: "is_Receive (b ·sstp ϑ) ∨ is_Send (b ·sstp ϑ)"
      using T by (cases b) simp_all

    note 6 = 1[OF T, of l A, unfolded f_def[symmetric]]
    note 7 = 1[OF 4, of l "A ·lsst ϑ", unfolded f_def[symmetric]]
    note 8 = ts IH subst_lsst_cons[of _ _ ϑ]

    have 9: "t · ϑ ∈ set (the_msgs (b ·sstp ϑ))" "not_occ (t · ϑ)"
      when t: "t ∈ set (the_msgs b)" "not_occ t" for t
    proof -
      show "t · ϑ ∈ set (the_msgs (b ·sstp ϑ))" using t ts ts' by auto
      moreover have "not_occ (t · ϑ)" when "t = Var x" for x
        using 2[OF T t(1)] t(2) unfolding that not_occ_def by simp
      moreover have "not_occ (t · ϑ)" when "t = Fun g ss" "g ≠ OccursFact" for g ss
        using 2[OF T t(1)] t(2) that(2) unfolding that(1) not_occ_def by simp
      moreover have "not_occ (t · ϑ)"
        when "t = Fun OccursFact ss" "∄s1 s2. ss = [s1,s2]" for ss
        using 2[OF T t(1)] t(2) that(2) unfolding that(1) not_occ_def by auto
      moreover have "not_occ (t · ϑ)"
        when "t = Fun OccursFact [s1,s2]" for s1 s2
        using 2[OF T t(1)] t(2) unfolding that not_occ_def by (cases s1) auto
      ultimately show "not_occ (t · ϑ)" by (cases t) (metis, metis)
    qed

    have 10: "not_occ t"
      when t: "t ∈ set (the_msgs b)" "not_occ (t · ϑ)" for t
    proof -
      have "t · ϑ ∈ set (the_msgs (b ·sstp ϑ))" using t ts ts' by auto
      moreover have "not_occ t" when "t = Var x" for x
        using 2[OF T t(1)] t(2) unfolding that not_occ_def by simp
      moreover have "not_occ t" when "t = Fun g ss" "g ≠ OccursFact" for g ss
        using 2[OF T t(1)] t(2) that(2) unfolding that(1) not_occ_def by simp
      moreover have "not_occ t"
        when "t = Fun OccursFact ss" "∄s1 s2. ss = [s1,s2]" for ss
        using 2[OF T t(1)] t(2) that(2) unfolding that(1) not_occ_def by auto
```

```
    moreover have "not_occ t"
      when "t = Fun OccursFact [s1,s2]" for s1 s2
      using 2[OF T t(1)] t(2) unfolding that not_occ_def by (cases s1) auto
    ultimately show "not_occ t" unfolding not_occ_def by force
  qed

  have 11: "not_occ (t · ϑ) ⟷ not_occ t" when "t ∈ set ts" for t
    using that 9 10 unfolding the_msgs_b by blast

  have 5: "(∃t. occurs t ∈ set ts) ⟷ (∃t. occurs t ∈ set ts ·set ϑ)"
    using 11 image_iff unfolding not_occ_def by fastforce

  have 12: "flt (ts ·list ϑ) = (flt ts) ·list ϑ" using 11
  proof (induction ts)
    case (Cons t ts)
    hence "not_occ (t · ϑ) = not_occ t" "flt (ts ·list ϑ) = (flt ts) ·list ϑ" by auto
    thus ?case unfolding flt_def by auto
  qed (metis flt_def filter.simps(1) map_is_Nil_conv)

  show ?thesis
  proof (cases "∃t. occurs t ∈ set (the_msgs b)")
    case True
    note T1 = this
    hence T2: "∃t. occurs t ∈ set (the_msgs (b ·sstp ϑ))" using 5 unfolding the_msgs_b by simp

    show ?thesis
    proof (cases "∃t ∈ set (the_msgs b). ∀s. t ≠ occurs s")
      case True
      note T1' = this
      have T2': "∃t ∈ set (the_msgs (b ·sstp ϑ)). ∀s. t ≠ occurs s"
        using T1' 11 unfolding the_msgs_b not_occ_def by auto

      show ?thesis using T
      proof
        assume b: "is_Receive b"
        hence bϑ: "is_Receive (b ·sstp ϑ)" using ts by fastforce

        show ?thesis
          using 6(3)[OF T1 T1' b] 7(3)[OF T2 T2' bϑ] IH 12
          unfolding f_def[symmetric] a flt_def[symmetric] not_occ_def[symmetric] the_msgs_b
          by (simp add: subst_lsst_cons)
      next
        assume b: "is_Send b"
        hence bϑ: "is_Send (b ·sstp ϑ)" using ts by fastforce

        show ?thesis
          using 6(2)[OF T1 T1' b] 7(2)[OF T2 T2' bϑ] IH 12
          unfolding f_def[symmetric] a flt_def[symmetric] not_occ_def[symmetric] the_msgs_b
          by (simp add: subst_lsst_cons)
      qed
    next
      case False
      hence F: "∀t ∈ set (the_msgs b). ∃s. t = occurs s" by blast
      hence F': "∀t ∈ set (the_msgs (b ·sstp ϑ)). ∃s. t = occurs s" unfolding the_msgs_b by auto

      have *: "∃t. occurs t ∈ set (the_msgs b)" when "the_msgs b ≠ []"
        using that F by (cases "the_msgs b") auto
      hence **: "∃t. occurs t ∈ set (the_msgs (b ·sstp ϑ))" when "the_msgs b ≠ []"
        using that 5 unfolding the_msgs_b by simp

      show ?thesis
      proof (cases "ts = []")
        case True
```

```
          hence ***: "∄t. occurs t ∈ set (the_msgs b)" "∄t. occurs t ∈ set (the_msgs (b ·sstp ϑ))"
            unfolding the_msgs_b by simp_all

          show ?thesis
            using IH 6(1)[OF ***(1)] 7(1)[OF ***(2)]
            unfolding a f_def[symmetric] True
            by (simp add: subst_lsst_cons)
        next
          case False thus ?thesis
            using IH 6(4)[OF * F] 7(4)[OF ** F']
            unfolding f_def[symmetric] not_occ_def[symmetric] a the_msgs_b
            by (simp add: subst_lsst_cons)
        qed
      qed
    next
      case False
      note F = this
      have F': "∄t. occurs t ∈ set (the_msgs (b ·sstp ϑ))"
        using F 11 unfolding not_occ_def the_msgs_b by fastforce

      show ?thesis
        using IH 6(1)[OF F] 7(1)[OF F']
        unfolding a f_def[symmetric] True
        by (simp add: subst_lsst_cons)
    qed
  next
    case False
    hence *: "¬is_Receive b" "¬is_Send b" "¬is_Receive (b ·sstp ϑ)" "¬is_Send (b ·sstp ϑ)"
      by (cases b; auto)+

    show ?thesis
      using IH 0(1)[OF *(1,2), of 1 A] 0(1)[OF *(3,4), of 1 "A ·lsst ϑ"] subst_lsst_cons[of a _ ϑ]
      unfolding a f_def by simp
  qed
qed simp

private lemma rm_occurs_msgs_constr_transaction_strand:
  assumes T_adm: "admissible_transaction' T"
  shows "rm_occurs_msgs_constr (transaction_checks T) = transaction_checks T" (is ?A)
    and "rm_occurs_msgs_constr (transaction_updates T) = transaction_updates T" (is ?B)
    and "admissible_transaction_no_occurs_msgs T ⟹
          rm_occurs_msgs_constr (transaction_receive T) = transaction_receive T" (is "?C ⟹ ?C'")
    and "admissible_transaction_no_occurs_msgs T ⟹
          rm_occurs_msgs_constr (transaction_send T) = transaction_send T" (is "?D ⟹ ?D'")
proof -
  note 0 = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]
  note 1 = wellformed_transaction_cases[OF 0]

  have 2: "∃ts. b = receive⟨ts⟩ ∧ (∄t. occurs t ∈ set ts)"
    when "admissible_transaction_no_occurs_msgs T" "(l,b) ∈ set (transaction_receive T)" for l b
    using that 1(1)[OF that(2)]
    unfolding admissible_transaction_no_occurs_msgs_def Let_def list_all_iff by fastforce

  have 3: "∃ts. b = send⟨ts⟩ ∧ (∄t. occurs t ∈ set ts)"
    when "admissible_transaction_no_occurs_msgs T" "(l,b) ∈ set (transaction_send T)" for l b
    using that 1(4)[OF that(2)]
    unfolding admissible_transaction_no_occurs_msgs_def Let_def list_all_iff by fastforce

  define A where "A ≡ transaction_receive T"
  define B where "B ≡ transaction_checks T"
  define C where "C ≡ transaction_updates T"
  define D where "D ≡ transaction_send T"
```

```
  show ?A using 1(2) unfolding B_def[symmetric]
  proof (induction B)
    case (Cons a A)
    hence IH: "rm_occurs_msgs_constr A = A" by (meson list.set_intros(2))
    obtain l b where a: "a = (l,b)" by (metis surj_pair)
    show ?case using Cons.prems IH unfolding a by (cases b) auto
  qed simp

  show ?B using 1(3) unfolding C_def[symmetric]
  proof (induction C)
    case (Cons a A)
    hence IH: "rm_occurs_msgs_constr A = A" by (meson list.set_intros(2))
    obtain l b where a: "a = (l,b)" by (metis surj_pair)
    show ?case using Cons.prems IH unfolding a by (cases b) auto
  qed simp

  show ?C' when ?C using 2[OF that] unfolding A_def[symmetric]
  proof (induction A)
    case (Cons a A)
    hence IH: "rm_occurs_msgs_constr A = A" by (meson list.set_intros(2))
    obtain l b where a: "a = (l,b)" by (metis surj_pair)
    obtain ts where b: "b = receive⟨ts⟩" using Cons.prems unfolding a by auto
    show ?case using Cons.prems IH unfolding a b by fastforce
  qed simp


  show ?D' when ?D using 3[OF that] unfolding D_def[symmetric]
  proof (induction D)
    case (Cons a A)
    hence IH: "rm_occurs_msgs_constr A = A" by (meson list.set_intros(2))
    obtain l b where a: "a = (l,b)" by (metis surj_pair)
    obtain ts where b: "b = send⟨ts⟩" using Cons.prems unfolding a by auto
    show ?case using Cons.prems IH unfolding a b by fastforce
  qed simp
qed

private lemma rm_occurs_msgs_constr_transaction_strand':
  fixes T::"('fun,'atom,'sets,'lbl) prot_transaction"
  assumes T_adm: "admissible_transaction' T"
    and T_no_occ: "admissible_transaction_no_occurs_msgs T"
  shows "rm_occurs_msgs_constr (transaction_strand (add_occurs_msgs T)) = transaction_strand T"
    (is "?f (?g (?h T)) = ?g T")
proof -
  obtain A B C D E F where T: "T = Transaction A B C D E F" by (cases T) simp

  have B: "B = transaction_fresh T" unfolding T by simp
  have F: "F = transaction_send T" unfolding T by simp

  define xs where "xs ≡ filter (λx. x ∉ set B) (fv_list_sst (unlabel (transaction_strand T)))"

  note 0 = rm_occurs_msgs_constr_transaction_strand
  note 1 = add_occurs_msgs_admissible_occurs_checks[OF T_adm]
  note 2 = 0(3,4)[OF T_adm T_no_occ]
  note 3 = add_occurs_msgs_cases[OF T]
  note 4 = 0(1,2)[OF 1(1)]

  have 5: "?f (transaction_checks (?h T)) = transaction_checks T"
          "?f (transaction_updates (?h T)) = transaction_updates T"
    using 4 3(8,9) by (argo, argo)

  have 6: "?f (transaction_receive (?h T)) = transaction_receive T"
  proof (cases "xs = []")
    case True show ?thesis using 3(6)[OF True[unfolded xs_def B]] 2(1) by simp
```

```
  next
    case False show ?thesis
      using False 3(7)[OF False[unfolded xs_def B]] 2(1)
            rm_occurs_msgs_constr_Cons(4)[
              of "receive⟨map (λx. occurs (Var x)) xs⟩" ⋆ "transaction_receive T"]
      unfolding B[symmetric] xs_def[symmetric]
      by (cases xs) (blast, auto)
  qed

  have 7: "?f (transaction_send (?h T)) = transaction_send T"
  proof (cases "∃ts' F'. F = ⟨⋆, send⟨ts'⟩⟩#F'")
    case True
    then obtain ts' F' where F': "F = ⟨⋆, send⟨ts'⟩⟩#F'" by blast

    have *: "transaction_send (?h T) = ⟨⋆, send⟨map (λx. occurs (Var x)) B@ts'⟩⟩#F'"
      using 3(1)[OF F'] unfolding T by fastforce

    have **: "ts' ≠ []" using admissible_transactionE(17)[OF T_adm] unfolding T F' by auto

    have ***: "∀s. t ≠ occurs s" when t: "t ∈ set ts'" for t
      using that T_no_occ
      unfolding T F' admissible_transaction_no_occurs_msgs_def Let_def list_all_iff by auto

    let ?ts = "map (λx. occurs (Var x)) B@ts'"

    have "∃t ∈ set ?ts. ∀s. t ≠ occurs s" using ** *** by (cases ts') auto
    moreover have "filter (λt. ∀s. t ≠ occurs s) ?ts = ts'" using *** by simp
    moreover have "∃t. occurs t ∈ set ?ts" when "B ≠ []" using that by (cases B) auto
    moreover have "?f [⟨⋆, send⟨ts'⟩⟩] = [⟨⋆, send⟨ts'⟩⟩]"
      using 2(2) ** *** unfolding F[symmetric] F' by force
    hence "?f F' = F'"
      using 2(2) rm_occurs_msgs_constr_append[of "[⟨⋆, send⟨ts'⟩⟩]" F']
      unfolding F[symmetric] F' by fastforce
    ultimately have "?f (⟨⋆, send⟨?ts⟩⟩#F') = ⟨⋆, send⟨ts'⟩⟩#F'"
      using 2(2) 3(10)[OF F'[unfolded F]] 3(12)
            rm_occurs_msgs_constr.simps(3)[of ⋆ ts' F']
            rm_occurs_msgs_constr_append[of "[⟨⋆, send⟨ts'⟩⟩]" F']
      unfolding F[symmetric] F' B[symmetric] by auto
    thus ?thesis using F * unfolding F' by argo
  next
    case False show ?thesis
      using 3(2)[OF False] 3(3) 2(2)
      unfolding B[symmetric] xs_def[symmetric] F[symmetric]
      by (cases B) auto
  qed

  show ?thesis
    using 5 6 7 rm_occurs_msgs_constr_append
    unfolding transaction_strand_def by metis
qed

private lemma rm_occurs_msgs_constr_transaction_strand'':
  fixes T::"('fun,'atom,'sets,'lbl) prot_transaction"
  assumes T_adm: "admissible_transaction' T"
    and T_no_occ: "admissible_transaction_no_occurs_msgs T"
    and ϑ: "∀x ∈ fv_transaction (add_occurs_msgs T). ∄t. ϑ x = occurs t"
           "∀x ∈ fv_transaction (add_occurs_msgs T). ϑ x ≠ Fun OccursSec []"
  shows "rm_occurs_msgs_constr (dual_lsst (transaction_strand (add_occurs_msgs T) ·_lsst ϑ)) =
         dual_lsst (transaction_strand T ·_lsst ϑ)"
using rm_occurs_msgs_constr_dual_lsst[of "transaction_strand (add_occurs_msgs T) ·_lsst ϑ"]
      rm_occurs_msgs_constr_subst[OF ϑ] rm_occurs_msgs_constr_transaction_strand'[OF T_adm T_no_occ]
by argo
```

```
private lemma rm_occurs_msgs_constr_bvars_subst_eq:
  "bvars_lsst (rm_occurs_msgs_constr A ·_lsst ϑ) = bvars_lsst (A ·_lsst ϑ)"
proof -
  have "bvars_lsst (rm_occurs_msgs_constr A) = bvars_lsst A"
  proof (induction A)
    case (Cons a A)
    obtain l b where a: "a = (l,b)" by (metis surj_pair)
    show ?case using Cons.IH unfolding a by (cases b) auto
  qed simp
  thus ?thesis by (metis bvars_sst_subst unlabel_subst)
qed

private lemma rm_occurs_msgs_constr_reachable_constraints_fv_eq:
  assumes P: "∀T ∈ set P. admissible_transaction' T"
             "∀T ∈ set P. admissible_transaction_no_occurs_msgs T"
    and A: "A ∈ reachable_constraints (map add_occurs_msgs P)"
  shows "fv_lsst (rm_occurs_msgs_constr A) = fv_lsst A"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  let ?f = rm_occurs_msgs_constr
  let ?B = "dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"

  define ϑ where "ϑ ≡ ξ ∘_s σ ∘_s α"

  obtain T' where T': "T' ∈ set P" "T = add_occurs_msgs T'"
    using step.hyps(2) by auto

  have T_adm: "admissible_transaction' T"
    using add_occurs_msgs_admissible_occurs_checks(1) step.hyps(2) P by auto

  have T'_adm: "admissible_transaction' T'"
    and T'_no_occ: "admissible_transaction_no_occurs_msgs T'"
    using T'(1) P by (blast,blast)

  have "?f (dual_lsst (transaction_strand T ·_lsst ϑ)) = dual_lsst (transaction_strand T' ·_lsst ϑ)"
    using rm_occurs_msgs_constr_transaction_strand''[OF T'_adm T'_no_occ, of ϑ]
          admissible_transaction_decl_fresh_renaming_subst_not_occurs[OF T_adm step.hyps(3,4,5)]
    unfolding T'(2) ϑ_def[symmetric] by blast
  moreover have "fv_lsst (transaction_strand T ·_lsst ϑ) = fv_lsst (transaction_strand T' ·_lsst ϑ)"
    using add_occurs_msgs_vars_eq(6)[OF T'_adm, of ϑ] unfolding T'(2) by blast
  ultimately have "fv_lsst (?f ?B) = fv_lsst ?B"
    using fv_sst_unlabel_dual_lsst_eq unfolding T'(2) ϑ_def[symmetric] by metis
  thus ?case
    using step.IH fv_sst_append[of "unlabel 𝒜" "unlabel ?B"]
          rm_occurs_msgs_constr_append[of 𝒜 ?B]
    by force
qed simp

private lemma rm_occurs_msgs_constr_reachable_constraints_vars_eq:
  assumes P: "∀T ∈ set P. admissible_transaction' T"
             "∀T ∈ set P. admissible_transaction_no_occurs_msgs T"
    and A: "A ∈ reachable_constraints (map add_occurs_msgs P)"
  shows "vars_lsst (rm_occurs_msgs_constr A) = vars_lsst A"
using rm_occurs_msgs_constr_bvars_subst_eq[of _ Var]
      rm_occurs_msgs_constr_reachable_constraints_fv_eq[OF P A]
by (metis vars_sst_is_fv_sst_bvars_sst subst_lsst_id_subst)

private lemma rm_occurs_msgs_constr_reachable_constraints_trms_cases_aux:
  assumes A: "x ∈ fv_sst A" "bvars_sst A = {}"
    and t: "t = occurs (ϑ x)"
    and ϑ: "(∃y. ϑ x = Var y) ∨ (∃c. ϑ x = Fun c [])"
  shows "(∃x ∈ fv_sst (A ·_sst ϑ). t = occurs (Var x)) ∨
```

```
        (∃c. Fun c [] ⊑set trmssst (A ·sst ϑ) ∧ t = occurs (Fun c []))"
using A
proof (induction A)
  case (Cons a A)
  have 0: "bvarssst A = {}" "set (bvarssstp a) = {}" "set (bvarssstp a) ∩ subst_domain ϑ = {}"
    using Cons.prems(2) by auto

  note 1 = fvsst_Cons[of a A] trmssst_cons[of a A] subst_sst_cons[of a A ϑ]

  show ?case
  proof (cases "x ∈ fvsst A")
    case False
    hence x: "x ∈ fvsstp a" using Cons.prems(1) by simp

    note 2 = x t ϑ

    have 3: "ϑ x ⊑set trmssstp (a ·sstp ϑ)"
      using subst_subterms[OF fvsstp_is_subterm_trmssstp[OF x]] trmssstp_subst[OF 0(3)] by auto

    have "Fun c [] ⊑set trmssstp (a ·sstp ϑ)" when "ϑ x = Fun c []" for c
      using that 3 t by argo
    moreover have "y ∈ fvsstp (a ·sstp ϑ)" when "ϑ x = Var y" for y
      using that 3 var_subterm_trmssstp_is_varssstp[of y "a ·sstp ϑ"] 0(2)
      unfolding varssstp_is_fvsstp_bvarssstp bvarssstp_subst by simp
    ultimately have
        "(∃x ∈ fvsstp (a ·sstp ϑ). t = occurs (Var x)) ∨
         (∃c. Fun c [] ⊑set trmssstp (a ·sstp ϑ) ∧ t = occurs (Fun c []))"
      using t ϑ by fast
    thus ?thesis using 1 by auto
  qed (use Cons.IH[OF _ 0(1)] 1 in force)
qed simp

private lemma rm_occurs_msgs_constr_reachable_constraints_trms_cases:
  assumes P: "∀T ∈ set P. admissible_transaction' T"
             "∀T ∈ set P. admissible_transaction_no_occurs_msgs T"
    and A: "A = rm_occurs_msgs_constr B"
    and B: "B ∈ reachable_constraints (map add_occurs_msgs P)"
    and t: "t ∈ trmslsst B"
  shows "t ∈ trmslsst A ∨ (∃x ∈ fvlsst A. t = occurs (Var x)) ∨
         (∃c. Fun c [] ⊑set (trmslsst A) ∧ t = occurs (Fun c []))"
    (is "?A A ∨ ?B A ∨ ?C A")
proof -
  define rm_occs where
    "rm_occs ≡ λA::('fun,'atom,'sets,'lbl) prot_strand. rm_occurs_msgs_constr A"
  define Q where "Q ≡ λA. ?A A ∨ ?B A ∨ ?C A"

  have 0: "Q B" when "Q A" "set A ⊆ set B" for A B
    using that unfolding Q_def fvsst_def trmssst_def unlabel_def by auto

  have "Q A" using B t unfolding A
  proof (induction rule: reachable_constraints.induct)
    case (step 𝒜 T ξ σ α)
    define ϑ where "ϑ ≡ ξ ∘s σ ∘s α"
    define ℬ where "ℬ ≡ duallsst (transaction_strand T ·lsst ϑ)"

    obtain T' where T': "T' ∈ set P" "T = add_occurs_msgs T'"
      using step.hyps(2) by auto

    note T'_adm = bspec[OF P(1) T'(1)] bspec[OF P(2) T'(1)]
    note T_adm = add_occurs_msgs_admissible_occurs_checks[OF T'_adm(1), unfolded T'(2)[symmetric]]

    note 1 = ϑ_def[symmetric] ℬ_def[symmetric] rm_occs_def[symmetric]
    note 2 = rm_occurs_msgs_constr_append[of 𝒜 ℬ, unfolded rm_occs_def[symmetric]]
```

```
    note 3 = admissible_transaction_decl_fresh_renaming_subst_not_occurs[
              OF T_adm(1) step.hyps(3,4,5)]

    have 4: "rm_occs (dual_lsst (transaction_strand T ·_lsst ϑ)) =
             dual_lsst (transaction_strand T' ·_lsst ϑ)"
      using 3 rm_occurs_msgs_constr_transaction_strand''[OF T'_adm]
      unfolding T'(2) 1 by blast

    have 5: "(∃y. ϑ x = Var y) ∨ (∃c. ϑ x = Fun c [])" for x
      using transaction_decl_fresh_renaming_substs_range'(1)[OF step.hyps(3,4,5)]
      unfolding ϑ_def[symmetric] by blast

    show ?case
    proof (cases "t ∈ trms_lsst 𝒜")
      case True show ?thesis using 0[OF step.IH[OF True]] unfolding 1 2 by simp
    next
      case False
      hence "t ∈ trms_lsst ℬ" using step.prems unfolding ℬ_def ϑ_def by simp
      hence "t ∈ trms_lsst (transaction_strand T' ·_lsst ϑ) ∨
             (∃x ∈ fv_transaction T'. t = occurs (ϑ x))"
        using add_occurs_msgs_in_trms_subst_cases[OF T'_adm(1), of t ϑ]
        unfolding ℬ_def trms_sst_unlabel_dual_lsst_eq T'(2) by blast
      moreover have "(∃y. ϑ x = Var y) ∨ (∃c. ϑ x = Fun c [])" for x
        using transaction_decl_fresh_renaming_substs_range'(1)[OF step.hyps(3,4,5)]
        unfolding ϑ_def[symmetric] by blast
      ultimately have "Q (rm_occs ℬ)"
        using rm_occurs_msgs_constr_reachable_constraints_trms_cases_aux[
              of _ "unlabel (transaction_strand T')" t ϑ]
            admissible_transactionE(4)[OF T'_adm(1)]
        unfolding Q_def ℬ_def 4 trms_sst_unlabel_dual_lsst_eq fv_sst_unlabel_dual_lsst_eq unlabel_subst
        by fast
      thus ?thesis using 0[of "rm_occs ℬ"] unfolding 1 2 by auto
    qed
  qed simp
  thus ?thesis unfolding Q_def by blast
qed

private lemma rm_occurs_msgs_constr_receive_attack_iff:
  fixes A::"('a,'b,'c,'d) prot_strand"
  shows "(∃ts. attack⟨n⟩ ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel A)) ⟷
         (∃ts. attack⟨n⟩ ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel (rm_occurs_msgs_constr A)))"
  (is "(∃ts. attack⟨n⟩ ∈ set ts ∧ ?A A ts) ⟷ (∃ts. attack⟨n⟩ ∈ set ts ∧ ?B A ts)")
proof
  let ?att = "λts. attack⟨n⟩ ∈ set ts"

  define f where "f ≡ λts::('a,'b,'c,'d) prot_term list. filter (λt. ∀s. t ≠ occurs s) ts"

  have 0: "?att ts ⟷ ?att (f ts)"
          "?att ts ⟹ ∃t. occurs t ∈ set ts ⟹ ∃t ∈ set ts. ∀s. t ≠ occurs s"
          "∄t. occurs t ∈ set ts ⟹ f ts = ts"
    for ts::"('a,'b,'c,'d) prot_term list"
    unfolding f_def
    subgoal by simp
    subgoal by auto
    subgoal by (induct ts) auto
    done

  have "?B A (f ts)" when A: "?A A ts" and ts: "?att ts" for ts using A
  proof (induction A)
    case (Cons a A)
    obtain l b where a: "a = (l,b)" by (metis surj_pair)
```

```
    show ?case
    proof (cases "?A A ts")
      case True thus ?thesis using Cons.IH unfolding a by (cases b) simp_all
    next
      case False
      hence b: "b = receive⟨ts⟩" using Cons.prems unfolding a by simp
      show ?thesis using 0(2)[OF ts] 0(3) unfolding a b f_def by simp
    qed
  qed simp
  thus "(∃ts. ?att ts ∧ ?A A ts) ⟹ (∃ts. ?att ts ∧ ?B A ts)" using 0(1) by fast

  have "∃ts'. ts = f ts' ∧ ?A A ts'" when B: "?B A ts" and ts: "?att ts" for ts using B
  proof (induction A)
    case (Cons a A)
    obtain l b where a: "a = (l,b)" by (metis surj_pair)

    note 1 = rm_occurs_msgs_constr_Cons

    have 2: "receive⟨ts⟩ ∈ set (unlabel (rm_occurs_msgs_constr A))"
      when rcv_ts: "receive⟨ts⟩ ∈ set (unlabel (rm_occurs_msgs_constr ((l,send⟨ts'⟩)#A)))"
      for l ts ts' and A::"('a,'b,'c,'d) prot_strand"
    proof -
      have *: "is_Send (send⟨ts'⟩)" by simp

      have "set (unlabel (rm_occurs_msgs_constr [(l, send⟨ts'⟩)])) ⊆ {send⟨ts'⟩, send⟨f ts'⟩}"
        using 1(5-7)[OF *, of l "[]"] unfolding f_def by auto
      thus ?thesis using rcv_ts rm_occurs_msgs_constr_append[of "[(l,send⟨ts'⟩)]" A] by auto
    qed

    show ?case
    proof (cases "?B A ts")
      case True thus ?thesis using Cons.IH by auto
    next
      case False
      hence 3: "receive⟨ts⟩ ∈ set (unlabel (rm_occurs_msgs_constr [a]))"
        using rm_occurs_msgs_constr_append[of "[a]" A] Cons.prems by simp

      obtain ts' where b: "b = receive⟨ts'⟩" and b': "is_Receive b"
        using 2[of ts l _ A] Cons.prems False
        unfolding a by (cases b) auto

      have ts': "the_msgs (receive⟨ts'⟩) = ts'" by simp

      have "∃t ∈ set (the_msgs b). ∀s. t ≠ occurs s" when "∃t. occurs t ∈ set (the_msgs b)"
        using that 3 1(4)[OF b' that, of l "[]"] unfolding a by force
      hence "ts = f ts'"
        using 3 0(3)[of ts'] 1(3)[OF b', of l "[]", unfolded rm_occurs_msgs_constr.simps(1)]
        unfolding a b ts' f_def[symmetric] by fastforce
      thus ?thesis unfolding a b by auto
    qed
  qed simp
  thus "(∃ts. ?att ts ∧ ?B A ts) ⟹ (∃ts. ?att ts ∧ ?A A ts)" using 0 by fast
qed

private lemma add_occurs_msgs_soundness_aux1:
  fixes P::"('fun,'atom,'sets,'lbl) prot"
  defines "wt_attack ≡ λℐ 𝒜 l n. welltyped_constraint_model ℐ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
  assumes P: "∀T ∈ set P. admissible_transaction' T"
    and P_val: "has_initial_value_producing_transaction P"
    and A: "𝒜 ∈ reachable_constraints P" "wt_attack ℐ 𝒜 l n"
  shows "∃ℬ ∈ reachable_constraints P. ∃𝒥.
          wt_attack 𝒥 ℬ l n ∧ (∀x ∈ fv_lsst ℬ. ∃n. 𝒥 x = Fun (Val n) [])"
proof -
```

```
let ?f = "λ(T,ξ,σ,α). dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
let ?g = "concat ∘ map ?f"
let ?rcv_att = "λA n. receive⟨[attack⟨n⟩]⟩ ∈ set (unlabel A)"
let ?wt_model = welltyped_constraint_model

define valconst_cases where "valconst_cases ≡
  λI::('fun,'atom,'sets,'lbl) prot_subst. λx.
    (∃n. I x = Fun (Val n) []) ∨ (∃n. I x = Fun (PubConst Value n) [])"

define valconsts_only where "valconsts_only ≡
  λX. λI::('fun,'atom,'sets,'lbl) prot_subst. ∀x ∈ X. ∃n. I x = Fun (Val n) []"

define db_eq where "db_eq ≡
  λA B::('fun,'atom,'sets,'lbl) prot_constr. λs. λupds::('fun,'atom,'sets,'lbl) prot_constr.
    let f = filter is_Update ∘ unlabel;
        g = filter (λa. ∄l t ts. a = (l,insert⟨t,Fun (Set s) ts⟩))
    in (upds = [] ∧ f A = f B) ∨ (upds ≠ [] ∧ f (g A) = f (g B))"

define db_upds_consts_fresh where "db_upds_consts_fresh ≡
  λA::('fun,'atom,'sets,'lbl) prot_constr. λX. λJ::('fun,'atom,'sets,'lbl) prot_subst.
    ∀x ∈ X. (∃n. ℐ x = Fun (PubConst Value n) []) ⟶ (∀n s.
      insert⟨Fun (Val n) [],s⟩ ∈ set (unlabel A) ∨
      delete⟨Fun (Val n) [],s⟩ ∈ set (unlabel A) ⟶
        J x ≠ Fun (Val n) [])"

define subst_eq_on_privvals where "subst_eq_on_privvals ≡ λX 𝒥.
  ∀x ∈ X. (∃n. ℐ x = Fun (Val n) []) ⟶ ℐ x = 𝒥 x"

define subst_in_ik_if_subst_pubval where "subst_in_ik_if_subst_pubval ≡
  λX 𝒥. λℬ::('fun,'atom,'sets,'lbl) prot_constr.
    ∀x ∈ X. (∃n. ℐ x = Fun (PubConst Value n) []) ⟶ 𝒥 x ∈ ik_lsst ℬ"

define subst_eq_iff where "subst_eq_iff ≡ λX. λ𝒥::('fun,'atom,'sets,'lbl) prot_subst.
  ∀x ∈ X. ∀y ∈ X. ℐ x = ℐ y ⟷ 𝒥 x = 𝒥 y"

obtain x_val T_val T_upds s_val ts_val l1_val l2_val where x_val:
    "T_val ∈ set P" "Var x_val ∈ set ts_val" "Γv x_val = TAtom Value"
    "fv_set (set ts_val) = {x_val}" "∀n. ¬(Fun (Val n) [] ⊑set set ts_val)"
    "T_val = Transaction (λ(). []) [x_val] [] [] T_upds [(l1_val,send⟨ts_val⟩)]"
    "T_upds = [] ∨
     (T_upds = [(l2_val,insert⟨Var x_val,⟨s_val⟩s⟩)] ∧
      (∀T ∈ set P. ∀(l,a) ∈ set (transaction_strand T). ∀t.
        a ≠ select⟨t,⟨s_val⟩s⟩ ∧ a ≠ ⟨t in ⟨s_val⟩s⟩ ∧ a ≠ ⟨t not in ⟨s_val⟩s⟩ ∧
        a ≠ delete⟨t,⟨s_val⟩s⟩))"
  using has_initial_value_producing_transactionE[OF P_val P, of thesis]
  by (auto simp add: disj_commute)

have 0: "∄n. Fun (PubConst Value n) [] ⊑set trms_lsst ℬ" when "ℬ ∈ reachable_constraints P" for ℬ
  using reachable_constraints_val_funs_private(1)[OF that P(1)] funs_term_Fun_subterm'
  unfolding is_PubConstValue_def by fastforce

have I: "?wt_model ℐ 𝒜" "interpretation_subst ℐ" "wf_trms (subst_range ℐ)" "wt_subst ℐ"
  using welltyped_constraint_model_prefix[OF A(2)[unfolded wt_attack_def]]
        A(2)[unfolded wt_attack_def welltyped_constraint_model_def constraint_model_def]
  by blast+

have 1: "∀x ∈ fv_lsst 𝒜. valconst_cases ℐ x"
  using reachable_constraints_fv_Value_const_cases[OF P(1) A(1) I(1)]
  unfolding valconst_cases_def by blast

have 2: "?rcv_att 𝒜 n"
  using A(2) strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "[send⟨[attack⟨n⟩]⟩]" ℐ]
        reachable_constraints_receive_attack_if_attack'(2)[OF A(1) P(1) I(1)]
```

```
  unfolding wt_attack_def welltyped_constraint_model_def constraint_model_def by simp

note ξ_empty = admissible_transaction_decl_subst_empty[OF bspec[OF P(1)]]

have lmm:
    "∃ℬ ∈ reachable_constraints P. ∃𝒥.
        ?wt_model 𝒥 ℬ ∧ valconsts_only (fv_lsst 𝒜 ∪ X) 𝒥 ∧ (?rcv_att 𝒜 n ⟶ ?rcv_att ℬ n) ∧
        subst_eq_on_privvals (fv_lsst 𝒜 ∪ X) 𝒥 ∧
        subst_in_ik_if_subst_pubval (fv_lsst 𝒜 ∪ X) 𝒥 ℬ ∧
        subst_eq_iff (fv_lsst 𝒜 ∪ X) 𝒥 ∧
        vars_lsst 𝒜 = vars_lsst ℬ ∧ fv_lsst 𝒜 = fv_lsst ℬ ∧
        (∀n ∈ N. ¬(Fun (Val n) [] ⊑_set trms_lsst ℬ)) ∧
        ik_lsst 𝒜 ⊆ ik_lsst ℬ ∧ trms_lsst 𝒜 ⊆ trms_lsst ℬ ∧
        db_eq 𝒜 ℬ s_val T_upds ∧
        db_upds_consts_fresh 𝒜 (fv_lsst 𝒜 ∪ X) 𝒥"
  when "finite N" "∀n ∈ N. ¬(Fun (Val n) [] ⊑_set trms_lsst 𝒜)" "X ∩ fv_lsst 𝒜 = {}"
      "finite X" "∀x ∈ X. valconst_cases ℐ x" "∀x ∈ X. Γ_v x = TAtom Value"
  for N X
  using A(1) I(1) 1 that
proof (induction arbitrary: N X rule: reachable_constraints.induct)
  case init
  define pubvals where "pubvals ≡ {n | n x. x ∈ X ∧ ℐ x = Fun (PubConst Value n) []}"
  define X_vals where "X_vals ≡ {n | n x. x ∈ X ∧ ℐ x = Fun (Val n) []}"

  have X_vals_finite: "finite X_vals"
    using finite_surj[OF init.prems(6),
                      of X_vals "λx. THE n. ℐ x = Fun (Val n) []"]
    unfolding X_vals_def by force

  have pubvals_finite: "finite pubvals"
    using finite_surj[OF init.prems(6),
                      of pubvals "λx. THE n. ℐ x = Fun (PubConst Value n) []"]
    unfolding pubvals_def by force

  obtain T_val_fresh_vals and δ::"nat ⇒ nat"
    where T_val_fresh_vals: "T_val_fresh_vals ∩ (N ∪ X_vals) = {}"
      and δ: "inj δ" "δ ` pubvals = T_val_fresh_vals"
    using ex_finite_disj_nat_inj[OF pubvals_finite finite_UnI[OF init.prems(3) X_vals_finite]]
    by blast

  have T_val_fresh_vals_finite: "finite T_val_fresh_vals"
    using pubvals_finite δ(2) by blast

  obtain ℬ::"('fun,'atom,'sets,'lbl) prot_constr"
    where B:
        "ℬ ∈ reachable_constraints P"
        "T_upds = [] ⟹ list_all is_Receive (unlabel ℬ)"
        "T_upds ≠ [] ⟹ list_all (λa. is_Insert a ∨ is_Receive a) (unlabel ℬ)"
        "vars_lsst ℬ = {}"
        "∀n. Fun (Val n) [] ⊑_set trms_lsst ℬ ⟶ Fun (Val n) [] ∈ ik_lsst ℬ"
        "T_val_fresh_vals = {n. Fun (Val n) [] ∈ ik_lsst ℬ}"
        "∀l a. (l,a) ∈ set ℬ ∧ is_Insert a ⟶
                (l = l2_val ∧ (∃n. a = insert⟨Fun (Val n) [],⟨s_val⟩_s⟩))"
    using reachable_constraints_initial_value_transaction[
            OF P reachable_constraints.init T_val_fresh_vals_finite _ x_val]
    by auto

  define 𝒥 where "𝒥 ≡ λx.
    if x ∈ X ∧ (∃n. ℐ x = Fun (PubConst Value n) [])
    then Fun (Val (δ (THE n. ℐ x = Fun (PubConst Value n) []))) []
    else ℐ x"

  have 0: "ik_lsst [] ⊆ ik_lsst ℬ" "trms_lsst [] ⊆ trms_lsst ℬ" "?rcv_att [] n ⟶ ?rcv_att ℬ n"
```

```
      "vars_lsst [] = vars_lsst ℬ" "fv_lsst [] = fv_lsst ℬ"
  using B(4) vars_sst_is_fv_sst_bvars_sst[of "unlabel ℬ"] by auto

have 1: "db_eq [] ℬ s_val T_upds" using B(2,3,7)
proof (induction ℬ)
  case (Cons a B)
  then obtain l b where a: "a = (l,b)" by (metis surj_pair)

  have IH: "db_eq [] B s_val T_upds" using Cons.prems Cons.IH by auto

  show ?case
  proof (cases "T_upds = []")
    case True
    hence "is_Receive b" using a Cons.prems(1) by simp
    thus ?thesis using IH unfolding a db_eq_def Let_def by auto
  next
    case False
    hence "is_Insert b ∨ is_Receive b" using a Cons.prems(2) by simp
    hence "∃t. a = (l2_val,insert⟨t,⟨s_val⟩s⟩)" when b: "¬is_Receive b"
      using b Cons.prems(3) unfolding a by (metis list.set_intros(1))
    thus ?thesis using IH False unfolding a db_eq_def Let_def by auto
  qed
qed (simp add: db_eq_def)

have 2: "?wt_model 𝒥 ℬ"
  unfolding welltyped_constraint_model_def constraint_model_def
proof (intro conjI)
  show "wt_subst 𝒥" using I(4) init.prems(8) unfolding 𝒥_def wt_subst_def by fastforce

  show "strand_sem_stateful {} {} (unlabel ℬ) 𝒥"
    using B(2,3) strand_sem_stateful_if_no_send_or_check[of "unlabel ℬ" "{}" "{}" 𝒥]
    unfolding list_all_iff by blast

  show "subst_domain 𝒥 = UNIV" "ground (subst_range 𝒥)"
    using I(2) unfolding 𝒥_def subst_domain_def by auto

  show "wf_trms (subst_range 𝒥)"
    using I(3) unfolding 𝒥_def by fastforce
qed

have 3: "valconsts_only (fv_lsst [] ∪ X) 𝒥"
  using init.prems(7) unfolding 𝒥_def valconsts_only_def valconst_cases_def by fastforce

have 4: "subst_eq_on_privvals (fv_lsst [] ∪ X) 𝒥"
  unfolding subst_eq_on_privvals_def 𝒥_def by force

have 5: "subst_in_ik_if_subst_pubval (fv_lsst [] ∪ X) 𝒥 ℬ"
proof (unfold subst_in_ik_if_subst_pubval_def; intro ballI impI)
  fix x assume x: "x ∈ fv_lsst [] ∪ X" and "∃n. ℐ x = Fun (PubConst Value n) []"
  then obtain n where n: "ℐ x = Fun (PubConst Value n) []" by blast

  have "n ∈ pubvals" using x n unfolding pubvals_def by fastforce
  hence "δ n ∈ T_val_fresh_vals" using δ(2) by fast
  hence "Fun (Val (δ n)) [] ∈ ik_lsst ℬ" using B(6) by fast
  thus "𝒥 x ∈ ik_lsst ℬ" using x n unfolding 𝒥_def by simp
qed

have 6: "subst_eq_iff (fv_lsst [] ∪ X) 𝒥"
proof (unfold subst_eq_iff_def; intro ballI)
  fix x y assume "x ∈ fv_lsst [] ∪ X" "y ∈ fv_lsst [] ∪ X"
  hence x: "x ∈ X" and y: "y ∈ X" by auto

  show "ℐ x = ℐ y ⟷ 𝒥 x = 𝒥 y"
```

```
      proof
        show "ℐ x = ℐ y ⟹ 𝒥 x = 𝒥 y" using x y unfolding 𝒥_def by presburger
      next
        assume J_eq: "𝒥 x = 𝒥 y" show "ℐ x = ℐ y"
        proof (cases "∃n. ℐ x = Fun (PubConst Value n) []")
          case True
          then obtain n where n: "ℐ x = Fun (PubConst Value n) []" by blast
          hence J_x: "𝒥 x = Fun (Val (δ n)) []" using x unfolding 𝒥_def by simp

          show ?thesis
          proof (cases "∃m. ℐ y = Fun (PubConst Value m) []")
            case True
            then obtain m where m: "ℐ y = Fun (PubConst Value m) []" by blast
            have J_y: "𝒥 y = Fun (Val (δ m)) []" using y m unfolding 𝒥_def by simp
            show ?thesis using J_eq J_x J_y injD[OF δ(1), of n m] n m by auto
          next
            case False
            then obtain m where m: "ℐ y = Fun (Val m) []"
              using init.prems(7) y unfolding valconst_cases_def by blast
            moreover have "δ n ∈ T_val_fresh_vals" using δ(2) x n unfolding pubvals_def by blast
            moreover have "m ∈ X_vals" using y m unfolding X_vals_def by blast
            ultimately have "𝒥 x ≠ ℐ y" using m J_x T_val_fresh_vals by auto
            moreover have "𝒥 y = ℐ y" using m unfolding 𝒥_def by simp
            ultimately show ?thesis using J_eq by argo
          qed
        next
          case False
          then obtain n where n: "ℐ x = Fun (Val n) []"
            using init.prems(7) x unfolding valconst_cases_def by blast
          hence J_x: "𝒥 x = ℐ x" unfolding 𝒥_def by auto

          show ?thesis
          proof (cases "∃m. ℐ y = Fun (PubConst Value m) []")
            case False
            then obtain m where m: "ℐ y = Fun (Val m) []"
              using init.prems(7) y unfolding valconst_cases_def by blast
            have J_y: "𝒥 y = ℐ y" using y m unfolding 𝒥_def by simp
            show ?thesis using J_x J_y J_eq by presburger
          next
            case True
            then obtain m where m: "ℐ y = Fun (PubConst Value m) []" by blast
            hence "𝒥 y = Fun (Val (δ m)) []" using y unfolding 𝒥_def by fastforce
            moreover have "δ m ∈ T_val_fresh_vals" using δ(2) y m unfolding pubvals_def by blast
            moreover have "n ∈ X_vals" using x n unfolding X_vals_def by blast
            ultimately have "𝒥 y ≠ ℐ x" using n J_x T_val_fresh_vals by auto
            thus ?thesis using J_x J_eq by argo
          qed
        qed
      qed
    qed

    have 7: "∀n ∈ N. Fun (Val n) [] ∉ subterms_set (trms_lsst ℬ)"
      using B(5,6) T_val_fresh_vals by blast

    have 8: "db_upds_consts_fresh [] (fv_lsst [] ∪ X) 𝒥" unfolding db_upds_consts_fresh_def by simp

    show ?case using B(1) 0 1 2 3 4 5 6 7 8 by blast
  next
    case (step 𝒜 T ξ σ α N X')
    define ϑ where "ϑ ≡ ξ ∘s σ ∘s α"
    define T' where "T' ≡ dual_lsst (transaction_strand T ·lsst ϑ)"
    define T'_pubvals where "T'_pubvals ≡ {n. ∃x ∈ fv_lsst T'. ℐ x = Fun (PubConst Value n) []}"
    define 𝒜_vals where "𝒜_vals ≡ {n. Fun (Val n) [] ⊑set trms_lsst 𝒜}"
```

```
define ℐ_vals where "ℐ_vals ≡ {n. ∃x ∈ fv_lsst 𝒜 ∪ X' ∪ fv_lsst T'. ℐ x = Fun (Val n) []}"
define σ_vals where "σ_vals ≡ {n. Fun (Val n) [] ∈ subst_range σ}"

have 3: "welltyped_constraint_model ℐ 𝒜"
        "∀x ∈ fv_lsst 𝒜. valconst_cases ℐ x"
        "∀x ∈ fv_lsst T'. valconst_cases ℐ x"
        "strand_sem_stateful (ik_lsst 𝒜 ·_set ℐ) (dbupd_sst (unlabel 𝒜) ℐ {}) (unlabel T') ℐ"
        "∀x ∈ fv_lsst 𝒜 ∪ X'. valconst_cases ℐ x"
  using step.prems(2) welltyped_constraint_model_prefix[OF step.prems(1)]
        step.prems(1)[unfolded welltyped_constraint_model_def constraint_model_def]
        strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "unlabel T'" ℐ]
        step.prems(7)
  unfolding ϑ_def[symmetric] T'_def[symmetric] unlabel_append fv_sst_append
  by (blast,blast,blast,simp,blast)

note T_adm = bspec[OF P step.hyps(2)]
note T_wf = admissible_transaction_is_wellformed_transaction[OF T_adm]
note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm step.hyps(3)]

note 4 = admissible_transaction_sem_iff
note 5 = iffD1[OF 4[OF T_adm I(2,3), of "ik_lsst 𝒜 ·_set ℐ" "dbupd_sst (unlabel 𝒜) ℐ {}" ϑ,
                    unfolded T'_def[symmetric]]
                  3(4)]

note σ_dom = transaction_fresh_subst_domain[OF step.hyps(4)]

have σ_ran: "∃n. t = Fun (Val n) []" when t: "t ∈ subst_range σ" for t
proof -
  obtain x where x: "x ∈ set (transaction_fresh T)" "t = σ x"
    using σ_dom t by auto
  show ?thesis
    using x(1) admissible_transactionE(2)[OF T_adm]
          transaction_fresh_subst_sends_to_val[OF step.hyps(4) x(1)]
    unfolding x(2) by meson
qed

have T'_vals_in_σ: "Fun (Val k) [] ∈ subst_range σ"
  when k: "Fun (Val k) [] ⊑_set trms_lsst T'" for k
proof -
  have "Fun (Val k) [] ∈ (subterms_set (trms_transaction T)) ·_set ϑ"
    using k admissible_transactionE(4)[OF T_adm]
          transaction_decl_fresh_renaming_substs_trms[
            OF step.hyps(3,4,5), of "transaction_strand T"]
    unfolding T'_def ϑ_def[symmetric] trms_sst_unlabel_dual_lsst_eq by fast
  then obtain t where t: "t ⊑_set trms_transaction T" "t · ϑ = Fun (Val k) []" by force
  hence "Fun (Val k) [] ∈ subst_range ϑ"
    using admissible_transactions_no_Value_consts(1)[OF T_adm] by (cases t) force+
  thus ?thesis
    using transaction_decl_fresh_renaming_substs_range'(4)[OF step.hyps(3,4,5)] ξ_empty
    unfolding ϑ_def[symmetric] by blast
qed

have σ_vals_is_T'_vals: "k ∈ σ_vals ⟷ Fun (Val k) [] ⊑_set trms_lsst T'" for k
proof
  show "k ∈ σ_vals" when "Fun (Val k) [] ⊑_set trms_lsst T'"
    using that T'_vals_in_σ unfolding σ_vals_def by blast

  show "Fun (Val k) [] ⊑_set trms_lsst T'" when k: "k ∈ σ_vals"
  proof -
    have "Fun (Val k) [] ∈ subst_range σ" using k unfolding σ_vals_def by fast
    then obtain x where x: "x ∈ fv_transaction T" "σ x = Fun (Val k) []"
      using admissible_transactionE(7)[OF T_adm]
            transaction_fresh_subst_domain[OF step.hyps(4)]
```

```
        unfolding fv_transaction_unfold by fastforce

      have "ϑ x = Fun (Val k) []" using x(2) unfolding ϑ_def ξ_empty subst_compose_def by auto
      thus ?thesis
        using fv_sst_is_subterm_trms_sst_subst[OF x(1), of ϑ]
              admissible_transactionE(4)[OF T_adm]
        unfolding T'_def trms_sst_unlabel_dual_lsst_eq unlabel_subst by simp
    qed
  qed

  have σ_vals_N_disj: "N ∩ σ_vals = {}"
    using step.prems(4) σ_vals_is_T'_vals
    unfolding ϑ_def[symmetric] T'_def[symmetric] unlabel_append trms_sst_append by blast

  have T'_pubvals_finite: "finite T'_pubvals"
    using finite_surj[OF fv_sst_finite[of "unlabel T'"],
                      of T'_pubvals "λx. THE n. I x = Fun (PubConst Value n) []"]
    unfolding T'_pubvals_def by force

  have σ_vals_finite: "finite σ_vals"
  proof -
    have *: "finite (subst_range σ)" using transaction_fresh_subst_domain[OF step.hyps(4)] by simp
    show ?thesis
      using finite_surj[OF *, of σ_vals "λt. THE n. t = Fun (Val n) []"]
      unfolding σ_vals_def by force
  qed

  have A_vals_finite: "finite A_vals"
  proof -
    have *: "A_vals ⊆ (λt. THE n. t = Fun (Val n) []) ` subterms_set (trms_lsst A)"
      unfolding A_vals_def by force
    show ?thesis
      by (rule finite_surj[OF subterms_union_finite[OF trms_sst_finite] *])
  qed

  have I_vals_finite: "finite I_vals"
  proof -
    define X where "X ≡ fv_lsst A ∪ X' ∪ fv_lsst T'"
    have *: "finite X" using fv_sst_finite step.prems(6) unfolding X_def by blast
    show ?thesis
      using finite_surj[OF *, of I_vals "λx. THE n. I x = Fun (Val n) []"]
      unfolding I_vals_def X_def[symmetric] by force
  qed

  obtain T_val_fresh_vals and δ::"nat ⇒ nat"
    where T_val_fresh_vals: "T_val_fresh_vals ∩ (N ∪ σ_vals ∪ A_vals ∪ I_vals) = {}"
      and δ: "inj δ" "δ ` T'_pubvals = T_val_fresh_vals"
    using step.prems(3) T'_pubvals_finite σ_vals_finite A_vals_finite I_vals_finite
    by (metis finite_UnI ex_finite_disj_nat_inj)

  define N' where "N' ≡ N ∪ σ_vals ∪ T_val_fresh_vals"

  have T_val_fresh_vals_finite: "finite T_val_fresh_vals"
    using T'_pubvals_finite δ(2) by blast

  have N'_finite: "finite N'"
    using step.prems(3) T'_pubvals_finite T_val_fresh_vals_finite σ_vals_finite
    unfolding N'_def by auto

  have A_vals_trms_in: "n ∈ A_vals" when "Fun (Val n) [] ⊑_set trms_lsst A" for n
    using that unfolding A_vals_def by blast

  have N'_notin_A: "¬(Fun (Val n) [] ⊑_set trms_lsst A)" when n: "n ∈ N'" for n
```

```
proof -
  have ?thesis when n': "n ∈ N"
    using n' step.prems(4) unfolding N'_def unlabel_append trms_sst_append by blast
  moreover have ?thesis when n': "n ∈ σ_vals"
    using n' step.hyps(4) unfolding σ_vals_def transaction_fresh_subst_def by blast
  moreover have ?thesis when n': "n ∈ T_val_fresh_vals"
    using n' T_val_fresh_vals 𝒜_vals_trms_in by blast
  ultimately show ?thesis using n unfolding N'_def by blast
qed

have T'_fv_𝒜_disj: "fv_lsst 𝒜 ∩ fv_lsst T' = {}"
  using transaction_decl_fresh_renaming_substs_vars_disj(8)[OF step.hyps(3,4,5)]
        transaction_decl_fresh_renaming_substs_vars_subset(4)[OF step.hyps(3,4,5,2)]
  unfolding ϑ_def[symmetric] T'_def fv_sst_unlabel_dual_lsst_eq by blast

have X'_disj: "X' ∩ fv_lsst 𝒜 = {}" "X' ∩ fv_lsst T' = {}"
  using step.prems(5)
  unfolding ϑ_def[symmetric] T'_def[symmetric] unlabel_append fv_sst_append
  by (blast, blast)

have X'_disj': "(X' ∪ fv_lsst T') ∩ fv_lsst 𝒜 = {}"
  using X'_disj(1) T'_fv_𝒜_disj by blast

have X'_finite: "finite (X' ∪ fv_lsst T')"
  using step.prems(6) fv_sst_finite by blast

have 𝒜_X'_valconstcases: "∀x ∈ X' ∪ fv_lsst T'. valconst_cases ℐ x"
  using 3(3,5) by blast

have T'_value_vars: "Γ_v x = TAtom Value" when x: "x ∈ fv_lsst T'" for x
  using x reachable_constraints_fv_Value_typed[
          OF P reachable_constraints.step[OF step.hyps]]
  unfolding ϑ_def[symmetric] T'_def[symmetric] unlabel_append fv_sst_append by blast

have X'_T'_value_vars: "∀x ∈ X' ∪ fv_lsst T'. Γ_v x = TAtom Value"
  using step.prems(8) T'_value_vars by blast

have N'_not_subterms_𝒜: "∀n ∈ N'. ¬(Fun (Val n) [] ⊑_set trms_lsst 𝒜)"
  using N'_notin_𝒜 by blast

obtain ℬ 𝒥 where B:
    "ℬ ∈ reachable_constraints P" "?wt_model 𝒥 ℬ"
    "valconsts_only (fv_lsst 𝒜 ∪ X' ∪ fv_lsst T') 𝒥" "?rcv_att 𝒜 n ⟶ ?rcv_att ℬ n"
    "subst_eq_on_privvals (fv_lsst 𝒜 ∪ X' ∪ fv_lsst T') 𝒥"
    "subst_in_ik_if_subst_pubval (fv_lsst 𝒜 ∪ X' ∪ fv_lsst T') 𝒥 ℬ"
    "subst_eq_iff (fv_lsst 𝒜 ∪ X' ∪ fv_lsst T') 𝒥"
    "vars_lsst 𝒜 = vars_lsst ℬ" "fv_lsst 𝒜 = fv_lsst ℬ" "ik_lsst 𝒜 ⊆ ik_lsst ℬ" "trms_lsst 𝒜 ⊆ trms_lsst ℬ"
    "∀n ∈ N'. ¬(Fun (Val n) [] ⊑_set trms_lsst ℬ)"
    "db_eq 𝒜 ℬ s_val T_upds"
    "db_upds_consts_fresh 𝒜 (fv_lsst 𝒜 ∪ X' ∪ fv_lsst T') 𝒥"
  using step.IH[OF 3(1,2) N'_finite N'_not_subterms_𝒜 X'_disj' X'_finite
                𝒜_X'_valconstcases X'_T'_value_vars]
  unfolding Un_assoc by fast

have J:
    "wt_subst 𝒥" "constr_sem_stateful 𝒥 (unlabel ℬ)"
    "interpretation_subst 𝒥" "wf_trms (subst_range 𝒥)"
  using B(2) unfolding welltyped_constraint_model_def constraint_model_def by blast+

have T_val_fresh_vals_notin_ℬ: "¬(Fun (Val n) [] ⊑_set trms_lsst ℬ)"
    when "n ∈ T_val_fresh_vals" for n
  using that B(12) unfolding N'_def by blast
hence "∀n ∈ T_val_fresh_vals. ¬(Fun (Val n) [] ⊑_set trms_lsst ℬ)" by blast
```

```
then obtain T_val_constr::"('fun,'atom,'sets,'lbl) prot_constr"
  where T_val_constr:
      "ℬ@T_val_constr ∈ reachable_constraints P"
      "T_val_constr ∈ reachable_constraints P"
      "T_upds = [] ⟹ list_all is_Receive (unlabel T_val_constr)"
      "T_upds ≠ [] ⟹ list_all (λa. is_Insert a ∨ is_Receive a) (unlabel T_val_constr)"
      "vars_lsst T_val_constr = {}"
      "∀n. Fun (Val n) [] ⊑_set trms_lsst ℬ ⟶ Fun (Val n) [] ∉ ik_lsst T_val_constr"
      "∀n. Fun (Val n) [] ⊑_set trms_lsst T_val_constr ⟶ Fun (Val n) [] ∈ ik_lsst T_val_constr"
      "T_val_fresh_vals = {n. Fun (Val n) [] ∈ ik_lsst T_val_constr}"
      "∀l a. (l,a) ∈ set T_val_constr ∧ is_Insert a ⟶
              (l = l2_val ∧ (∃n. a = insert⟨Fun (Val n) [],⟨s_val⟩_s⟩))"
  using reachable_constraints_initial_value_transaction[
          OF P B(1) T_val_fresh_vals_finite _ x_val]
  by blast

have T_val_constr_no_upds_if_no_T_upds:
    "filter is_Update (unlabel T_val_constr) = []"
  when "T_upds = []"
  using T_val_constr(3)[OF that] by (induct T_val_constr) auto

have T_val_fresh_vals_is_T_val_constr_vals:
    "k ∈ T_val_fresh_vals ⟷ Fun (Val k) [] ⊑_set trms_lsst T_val_constr"
  for k
  using that T_val_constr(7,8) ik_sst_trms_sst_subset by fast

have T_val_constr_no_fv: "fv_lsst T_val_constr = {}"
  using T_val_constr(5) vars_sst_is_fv_sst_bvars_sst by fast

have T_val_σ: "transaction_fresh_subst σ T (trms_lsst (ℬ@T_val_constr))"
proof -
  have "¬(t ⊑_set trms_lsst (ℬ@T_val_constr))" when t: "t ∈ subst_range σ" for t
  proof -
    obtain k where k: "t = Fun (Val k) []" using t σ_ran by fast
    have "k ∈ σ_vals" using t unfolding k σ_vals_def by blast
    thus ?thesis
      using B(12) T_val_fresh_vals T_val_constr(7,8)
      unfolding N'_def k unlabel_append trms_sst_append by blast
  qed
  thus ?thesis using step.hyps(4) unfolding transaction_fresh_subst_def by fast
qed

have T_val_α: "transaction_renaming_subst α P (vars_lsst (ℬ@T_val_constr))"
  using step.hyps B(8) T_val_constr(5) by auto

define ℬ' where "ℬ' ≡ ℬ@T_val_constr@T'"

define 𝒦 where "𝒦 ≡ λx.
  if x ∈ fv_lsst T'
  then if ∃n. ℐ x = Fun (PubConst Value n) []
       then if ∃y ∈ fv_lsst ℬ ∪ X'. ℐ y = ℐ x
            then 𝒥 (SOME y. y ∈ fv_lsst ℬ ∪ X' ∧ ℐ y = ℐ x)
            else Fun (Val (δ (THE n. ℐ x = Fun (PubConst Value n) []))) []
       else ℐ x
  else 𝒥 x"

have σ_ground_ran: "ground (subst_range σ)" "range_vars σ = {}"
  and ξ_ran_bvars_disj: "range_vars ξ ∩ bvars_transaction T = {}"
  using transaction_fresh_subst_domain[OF step.hyps(4)]
        transaction_fresh_subst_range_vars_empty[OF step.hyps(4)]
        transaction_decl_subst_range_vars_empty[OF step.hyps(3)]
  by (metis range_vars_alt_def, argo, blast)
```

```
have B_T'_fv_disj: "fv_lsst B ∩ fv_lsst T' = {}"
  using T'_fv_A_disj unfolding B(9) by argo
hence J_K_fv_B_eq: "J x = K x" when x: "x ∈ fv_lsst B ∪ X'" for x
  using x X'_disj unfolding K_def by auto

have B'1: "B' ∈ reachable_constraints P"
  using reachable_constraints.step[OF T_val_constr(1) step.hyps(2,3) T_val_σ T_val_α]
  unfolding B'_def T'_def ϑ_def by simp

have "∃n. K x = Fun (Val n) []" when x: "x ∈ fv_lsst (A@T') ∪ X'" for x
proof (cases "x ∈ fv_lsst T'")
  case True
  note T = this
  show ?thesis
  proof (cases "∃n. I x = Fun (PubConst Value n) []")
    case True thus ?thesis
      using T B(3,9) someI_ex[of "λy. y ∈ fv_lsst B ∪ X' ∧ I y = I x"]
      unfolding K_def valconsts_only_def
      by (cases "∃y ∈ fv_lsst B ∪ X'. I y = I x") (meson, auto)
  next
    case False thus ?thesis
      using T 3(3) unfolding K_def valconst_cases_def by fastforce
  qed
next
  case False thus ?thesis using x B(3) unfolding K_def valconsts_only_def by auto
qed
hence B'3: "valconsts_only (fv_lsst (A@T') ∪ X') K" unfolding valconsts_only_def by blast

have B'4: "?rcv_att B' n" when "?rcv_att (A@T') n"
  using that B(4) unfolding B'_def by auto

have "I x = K x" when x: "x ∈ fv_lsst (A@T') ∪ X'" "I x = Fun (Val n) []" for x n
proof -
  have "K x = J x" when "x ∉ fv_lsst T'" using that unfolding K_def by meson
  moreover have "K x = I x" when "x ∈ fv_lsst T'" using that x unfolding K_def by simp
  ultimately show ?thesis
    using B(5) x
    unfolding subst_eq_on_privvals_def unlabel_append fv_sst_append
    by (cases "x ∈ fv_lsst T'") auto
qed
hence B'5: "subst_eq_on_privvals (fv_lsst (A@T') ∪ X') K"
  unfolding subst_eq_on_privvals_def by blast

have A_fv_K_eq_J: "K x = J x" when x: "x ∈ fv_lsst A ∪ X'" for x
proof -
  have "x ∉ fv_lsst T'" using x T'_fv_A_disj X'_disj by blast
  thus ?thesis unfolding K_def by argo
qed

have T'_fv_I_val_K_eq_J: "K x = I x"
  when x: "x ∈ fv_lsst T'" "∄n. I x = Fun (PubConst Value n) []" for x
  using x B'5 3(3) unfolding unlabel_append fv_sst_append valconst_cases_def K_def by meson

have T'_fv_I_pubval_K_eq_δ_fresh_val:
    "K x = Fun (Val (δ n)) []" "δ n ∈ T_val_fresh_vals"
  when x: "x ∈ fv_lsst T'" "I x = Fun (PubConst Value n) []" "∀y ∈ fv_lsst B ∪ X'. I y ≠ I x"
  for x n
proof -
  show "K x = Fun (Val (δ n)) []" using x unfolding K_def by auto
  show "δ n ∈ T_val_fresh_vals" using δ(2) x unfolding T'_pubvals_def by blast
qed

have T'_fv_I_pubval_K_eq_J_val:
```

```
      "∃y ∈ fv_lsst ℬ ∪ X'. ∃m. ℐ y = ℐ x ∧ 𝒦 x = 𝒥 y ∧ 𝒦 x = Fun (Val m) []"
    when x: "x ∈ fv_lsst T'" "ℐ x = Fun (PubConst Value n) []" "∃y ∈ fv_lsst ℬ ∪ X'. ℐ y = ℐ x"
    for x n
  proof -
    have "𝒦 x = 𝒥 (SOME y. y ∈ fv_lsst ℬ ∪ X' ∧ ℐ y = ℐ x)" using x unfolding 𝒦_def by meson
    then obtain y where y: "y ∈ fv_lsst ℬ ∪ X'" "ℐ y = ℐ x" "𝒦 x = 𝒥 y"
      using x(3) someI_ex[of "λy. y ∈ fv_lsst ℬ ∪ X' ∧ ℐ y = ℐ x"] by blast
    thus ?thesis using B(3,9) unfolding valconsts_only_def by auto
  qed

  have T'_fv_ℐ_pubval_𝒦_eq_val: "∃n. 𝒦 x = Fun (Val n) []"
    when x: "x ∈ fv_lsst T'" "ℐ x = Fun (PubConst Value n) []" for x n
    using T'_fv_ℐ_pubval_𝒦_eq_δ_fresh_val[OF x] T'_fv_ℐ_pubval_𝒦_eq_𝒥_val[OF x] by auto

  have B'6': "𝒦 x ∈ ik_lsst ℬ"
    when x: "x ∈ fv_lsst 𝒜 ∪ X'" "ℐ x = Fun (PubConst Value n) []" for x n
    using x B(6) 𝒜_fv_𝒦_eq_𝒥 x(2) unfolding B(8) subst_in_ik_if_subst_pubval_def by simp

  have B'6'': "𝒦 x ∈ ik_lsst ℬ ∪ ik_lsst T_val_constr"
    when x: "x ∈ fv_lsst T'" "ℐ x = Fun (PubConst Value n) []" for x n
  proof (cases "∃y ∈ fv_lsst ℬ ∪ X'. ℐ y = ℐ x")
    case True thus ?thesis
      using B(6) x(2) T'_fv_ℐ_pubval_𝒦_eq_𝒥_val[OF x True]
      unfolding B(9) subst_in_ik_if_subst_pubval_def by force
  next
    case False thus ?thesis
      using T_val_constr(8) T'_fv_ℐ_pubval_𝒦_eq_δ_fresh_val[OF x] by force
  qed

  have "𝒦 x ∈ ik_lsst ℬ'"
    when x: "x ∈ fv_lsst (𝒜@T') ∪ X'" "ℐ x = Fun (PubConst Value n) []" for x n
  proof (cases "x ∈ fv_lsst T'")
    case True thus ?thesis using B'6''[OF _ x(2)] unfolding ℬ'_def by auto
  next
    case False
    hence "x ∈ fv_lsst 𝒜 ∪ X'"
      using x(1) unfolding unlabel_append fv_sst_append by blast
    thus ?thesis using B'6' x(2) unfolding ℬ'_def by simp
  qed
  hence B'6: "subst_in_ik_if_subst_pubval (fv_lsst (𝒜@T') ∪ X') 𝒦 ℬ'"
    unfolding subst_in_ik_if_subst_pubval_def by blast

  have B'7: "subst_eq_iff (fv_lsst (𝒜@T') ∪ X') 𝒦"
  proof (unfold subst_eq_iff_def; intro ballI)
    fix x y assume xy: "x ∈ fv_lsst (𝒜@T') ∪ X'" "y ∈ fv_lsst (𝒜@T') ∪ X'"

    let ?Q = "λx y. ℐ x = ℐ y ⟷ 𝒦 x = 𝒦 y"

    have *: "?Q x y"
      when xy: "x ∈ fv_lsst 𝒜 ∪ X'" "x ∉ fv_lsst T'" "y ∈ fv_lsst 𝒜 ∪ X'" "y ∉ fv_lsst T'" for x y
      using B(7) xy unfolding 𝒦_def subst_eq_iff_def by force

    have **: "?Q x y" when x: "x ∈ fv_lsst 𝒜 ∪ X'" and y: "y ∈ fv_lsst T'" for x y
    proof -
      have xy_neq: "x ≠ y" using x y T'_fv_𝒜_disj X'_disj by blast

      have x_eq: "𝒦 x = 𝒥 x"
        using 𝒜_fv_𝒦_eq_𝒥 x by blast

      have x_eq_if_val: "ℐ x = 𝒥 x" when "ℐ x = Fun (Val n) []" for n
        using that x B(5) unfolding subst_eq_on_privvals_def by blast

      have x_neq_if_neq_val: "ℐ x ≠ 𝒥 x" when "ℐ x = Fun (PubConst Value n) []" for n
```

```
        by (metis that B(3) x UnI1 prot_fun.distinct(37) term.inject(2) valconsts_only_def)

      have y_eq_if_val: "ℐ y = 𝒦 y" when "ℐ y = Fun (Val n) []" for n
        using that y B'5 unfolding subst_eq_on_privvals_def by simp

      have y_eq: "𝒦 y = Fun (Val (δ n)) []"
        when "ℐ y = Fun (PubConst Value n) []" "∀z ∈ fv_lsst ℬ ∪ X'. ℐ z ≠ ℐ y" for n
        by (rule T'_fv_ℐ_pubval_𝒦_eq_δ_fresh_val(1)[OF y that])

      have y_eq': "∃z ∈ fv_lsst ℬ ∪ X'. ∃m. ℐ z = ℐ y ∧ 𝒦 y = 𝒥 z ∧ 𝒦 y = Fun (Val m) []"
        when "ℐ y = Fun (PubConst Value n) []" "∃z ∈ fv_lsst ℬ ∪ X'. ℐ z = ℐ y" for n
        by (rule T'_fv_ℐ_pubval_𝒦_eq_𝒥_val[OF y that])

      have K_eq_if_I_eq: "ℐ x = ℐ y ⟹ 𝒦 x = 𝒦 y"
        apply (cases "∃n. ℐ x = Fun (PubConst Value n) []")
        subgoal using B(7,9) unfolding subst_eq_iff_def by (metis UnI1 x x_eq y_eq')
        subgoal by (metis x_eq x T'_fv_ℐ_val_𝒦_eq_𝒥[OF y] 3(5) valconst_cases_def x_eq_if_val)
        done

      have K_neq_if_I_neq_val: "𝒦 x ≠ 𝒦 y"
        when n: "ℐ y = Fun (Val n) []"
          and m: "ℐ x = Fun (PubConst Value m) []"
        for n m
      proof -
        have I_neq: "ℐ x ≠ ℐ y" using n m by simp

        note y_eq'' = y_eq_if_val[OF n]
        note x_neq = x_neq_if_neq_val[OF m]

        have x_ex: "∃z ∈ fv_lsst ℬ ∪ X'. ℐ z = ℐ x" using x unfolding B(9) by blast
        have J1: "𝒥 x ∈ ik_lsst ℬ" using B(6) x m unfolding subst_in_ik_if_subst_pubval_def by fast
        have J2: "ℐ x ≠ 𝒥 x"
          by (metis m B(3) x UnI1 prot_fun.distinct(37) term.inject(2) valconsts_only_def)
        have J3: "ℐ y = 𝒥 y" using B(5) n y unfolding subst_eq_on_privvals_def by blast
        have K_x: "𝒦 x ≠ ℐ x" using J2 x_eq by presburger
        have x_notin: "x ∉ fv_lsst T'" using x T'_fv_𝒜_disj X'_disj by blast
        have K_x': "𝒦 x = 𝒥 x" using x_notin unfolding 𝒦_def by argo
        have K_y: "𝒦 y = 𝒥 y" using y_eq'' J3 by argo

        have "𝒥 x ≠ 𝒥 y" using I_neq x y B(7) unfolding subst_eq_iff_def by blast
        thus ?thesis using K_x' K_y by argo
      qed

      show ?thesis
      proof
        show "ℐ x = ℐ y ⟹ 𝒦 x = 𝒦 y" by (rule K_eq_if_I_eq)
      next
        assume xy_eq: "𝒦 x = 𝒦 y" show "ℐ x = ℐ y"
        proof (cases "∃n. ℐ y = Fun (PubConst Value n) []")
          case True
          then obtain n where n: "ℐ y = Fun (PubConst Value n) []" by blast

          show ?thesis
          proof (cases "∃z ∈ fv_lsst ℬ ∪ X'. ℐ z = ℐ y")
            case True thus ?thesis
              using B(7,9) unfolding subst_eq_iff_def by (metis xy_eq UnI1 x x_eq y_eq'[OF n])
          next
            case False
            hence F: "∀z ∈ fv_lsst ℬ ∪ X'. ℐ z ≠ ℐ y" by blast
            note y_eq'' = y_eq[OF n F]

            have n_in: "δ n ∈ T_val_fresh_vals"
              using  δ(2) x_eq xy_eq T_val_fresh_vals_notin_ℬ y n
```

```
          unfolding T'_pubvals_def by blast
        hence y_notin: "¬(𝒦 y ⊑set iklsst ℬ)"
          using T_val_fresh_vals_notin_ℬ y_eq'' iksst_trmssst_subset[of "unlabel ℬ"]
          by auto

        show ?thesis
        proof (cases "∃m. ℐ x = Fun (PubConst Value m) []")
          case True thus ?thesis
            using y_notin B(6) x xy_eq x_eq
            unfolding B(9) subst_in_ik_if_subst_pubval_def
            by fastforce
        next
          case False
          then obtain m where m: "ℐ x = Fun (Val m) []"
            using 3(5) x unfolding valconst_cases_def by fast
          hence "ℐ x = 𝒥 x" using x B(5) unfolding subst_eq_on_privvals_def by blast
          hence "𝒦 x = Fun (Val m) []" using m x_eq by argo
          moreover have "m ∉ T_val_fresh_vals"
            using m T_val_fresh_vals x unfolding ℐ_vals_def by blast
          hence "m ≠ δ n" using n_in by blast
          ultimately have False using xy_eq y_eq'' by force
          thus ?thesis by simp
        qed
      qed
    next
      case False
      then obtain n where n: "ℐ y = Fun (Val n) []"
        using 3(3) y unfolding valconst_cases_def by fast

      note y_eq'' = y_eq_if_val[OF n]

      show ?thesis
      proof (cases "∃m. ℐ x = Fun (Val m) []")
        case True thus ?thesis by (metis xy_eq x_eq y_eq'' x_eq_if_val)
      next
        case False
        then obtain m where m: "ℐ x = Fun (PubConst Value m) []"
          using 3(5) x unfolding valconst_cases_def by blast

        show ?thesis using K_neq_if_I_neq_val[OF n m] xy_eq by blast
      qed
    qed
  qed
qed

have ***: "?Q x y" when x: "x ∈ fvlsst T'" and y: "y ∈ fvlsst T'" for x y
proof
  assume xy_eq: "ℐ x = ℐ y" show "𝒦 x = 𝒦 y"
  proof (cases "∃n. ℐ x = Fun (PubConst Value n) []")
    case True thus ?thesis
      using xy_eq x y B(7) T'_fv_ℐ_pubval_𝒦_eq_δ_fresh_val(1) T'_fv_ℐ_pubval_𝒦_eq_𝒥_val
      unfolding B(9) subst_eq_iff_def by (metis (no_types) UnI1)
  qed (metis xy_eq x y T'_fv_ℐ_val_𝒦_eq_𝒥)
next
  assume xy_eq: "𝒦 x = 𝒦 y"

  have case1: False
    when x': "x ∈ fvlsst T'"
      and y': "y ∈ fvlsst T'"
      and xy_eq': "𝒦 x = 𝒦 y"
      and m: "ℐ x = Fun (PubConst Value m) []"
      and n: "ℐ y = Fun (Val n) []"
    for x y n m
```

```
    proof -
      have F: "∄n. 𝓘 y = Fun (PubConst Value n) []" using n by auto

      note x_eq = T'_fv_𝓘_pubval_𝓚_eq_δ_fresh_val[OF x' m]
      note y_eq = T'_fv_𝓘_val_𝓚_eq_𝓙[OF y' F]

      have "∃z ∈ fv_lsst 𝓑 ∪ X'. 𝓘 z = 𝓘 x"
      proof (rule ccontr)
        assume no_z: "¬(∃z ∈ fv_lsst 𝓑 ∪ X'. 𝓘 z = 𝓘 x)"
        hence "n ≠ δ m" using n y' x_eq(2) T_val_fresh_vals unfolding 𝓘_vals_def by blast
        thus False using xy_eq' x_eq y_eq(1) n no_z by auto
      qed
      then obtain z k where z:
          "z ∈ fv_lsst 𝓑 ∪ X'" "𝓘 z = 𝓘 x" "𝓚 x = 𝓙 z" "𝓚 x = Fun (Val k) []"
        using T'_fv_𝓘_pubval_𝓚_eq_𝓙_val[OF x' m] by blast

      have "𝓘 y = 𝓙 z" using z(2,3) y_eq xy_eq' by presburger
      hence "𝓘 x = 𝓘 y" using z(1,2) ** B(9) 𝓙_𝓚_fv_𝓑_eq y' y_eq by metis
      thus False using n m by simp
    qed

    have case2: "m = n"
      when x': "x ∈ fv_lsst T'"
        and y': "y ∈ fv_lsst T'"
        and xy_eq': "𝓚 x = 𝓚 y"
        and m: "𝓘 x = Fun (PubConst Value m) []"
        and n: "𝓘 y = Fun (PubConst Value n) []"
      for x y n m
    proof (cases "∀z ∈ fv_lsst 𝓑 ∪ X'. 𝓘 z ≠ 𝓘 x")
      case True show ?thesis
        apply (cases "∀z ∈ fv_lsst 𝓑 ∪ X'. 𝓘 z ≠ 𝓘 y")
        subgoal
          using xy_eq' m n T'_fv_𝓘_pubval_𝓚_eq_δ_fresh_val[OF x' m True]
                T'_fv_𝓘_pubval_𝓚_eq_δ_fresh_val[OF y' n] injD[OF δ(1), of m n]
          by fastforce
        subgoal by (metis x' y' xy_eq' ** B(9) True)
        done
    qed (metis x' y' xy_eq' m n ** B(9) prot_fun.inject(6) term.inject(2))

    have case3: "m = n"
      when x': "x ∈ fv_lsst T'"
        and y': "y ∈ fv_lsst T'"
        and xy_eq': "𝓚 x = 𝓚 y"
        and m: "𝓘 x = Fun (Val m) []"
        and n: "𝓘 y = Fun (Val n) []"
      for x y n m
      using x' y' xy_eq' m n T'_fv_𝓘_val_𝓚_eq_𝓙 by fastforce

    show "𝓘 x = 𝓘 y"
      using x y xy_eq case1 case2 case3 3(3)
      unfolding valconst_cases_def by metis
  qed

  have ****: "?Q x y" when xy: "x ∈ X'" "x ∉ fv_lsst T'" "y ∈ fv_lsst T'" for x y
  proof -
    have xy': "y ∉ X'" "y ∉ fv_lsst 𝓐" "x ∉ fv_lsst 𝓐"
      using xy X'_disj T'_fv_𝓐_disj by (blast, blast, blast)

    have K_x: "𝓚 x = 𝓙 x" using xy(2) unfolding 𝓚_def by argo

    have I_iff_J: "𝓘 x = 𝓘 y ⟷ 𝓙 x = 𝓙 y" using xy B(7) unfolding subst_eq_iff_def by fast

    show ?thesis using K_x I_iff_J K_x injD[OF δ(1)] xy B(7,9) ** by (meson UnCI)
```

```
    qed

    show "?Q x y"
      using xy * **[of x y] **[of y x] ***[of x y] ****[of x y] ****[of y x]
      unfolding unlabel_append fv_sst_append by (metis Un_iff)
  qed

  have B'8_9: "vars_lsst (𝒜@T') = vars_lsst ℬ'" "fv_lsst (𝒜@T') = fv_lsst ℬ'"
    using B(8,9) T_val_constr(5) vars_sst_is_fv_sst_bvars_sst[of "unlabel T_val_constr"]
    unfolding ℬ'_def unlabel_append vars_sst_append fv_sst_append by simp_all

  have I': "∃n. ℐ x = Fun (PubConst Value n) []"
    when x: "x ∈ fv_lsst T'" "∄n. ℐ x = Fun (Val n) []" for x
    using x 3(3) unfolding valconst_cases_def by fast

  have B'10_11: "ik_lsst (𝒜@T') ⊆ ik_lsst ℬ'" "trms_lsst (𝒜@T') ⊆ trms_lsst ℬ'"
    using B(10,11) unfolding N'_def ℬ'_def unlabel_append trms_sst_append ik_sst_append
    by (blast, blast)

  have B'12: "∀n ∈ N. ¬(Fun (Val n) [] ⊑_set trms_lsst ℬ')"
    using B(12) σ_vals_is_T'_vals σ_vals_N_disj T_val_fresh_vals
          T_val_fresh_vals_is_T_val_constr_vals
    unfolding N'_def ℬ'_def unfolding unlabel_append trms_sst_append ik_sst_append by fast

  have B'13: "db_eq (𝒜@T') ℬ' s_val T_upds"
  proof -
    let ?f = "filter is_Update ∘ unlabel"
    let ?g = "filter (λa. ∄l t ts. a = (l, insert⟨t,Fun (Set s_val) ts⟩))"

    have "?f (?g T_val_constr) = []" using T_val_constr(3,4,9)
    proof (induction T_val_constr)
      case (Cons a B)
      then obtain l b where a: "a = (l,b)" by (metis surj_pair)

      have IH: "?f (?g B) = []" using Cons.prems Cons.IH by auto

      show ?case
      proof (cases "T_upds = []")
        case True
        hence "is_Receive b" using a Cons.prems(1,2) by simp
        thus ?thesis using IH unfolding a Let_def by auto
      next
        case False
        hence "is_Insert b ∨ is_Receive b" using a Cons.prems(1,2) by simp
        hence "∃t. a = (l2_val, insert⟨t,⟨s_val⟩_s⟩)" when b: "¬is_Receive b"
          using b Cons.prems(3) unfolding a by (metis list.set_intros(1))
        thus ?thesis using IH unfolding a Let_def by auto
      qed
    qed simp
    hence "?f (?g (𝒜@T')) = ?f (?g 𝒜)@?f (?g T')"
          "?f (?g (ℬ@T_val_constr@T')) = ?f (?g ℬ)@?f (?g T')"
      when "T_upds ≠ []"
      by simp_all
    moreover have "?f T_val_constr = []" when "T_upds = []"
      using T_val_constr_no_upds_if_no_T_upds[OF that] by force
    hence "?f (𝒜@T') = ?f 𝒜@?f T'"
          "?f (ℬ@T_val_constr@T') = ?f ℬ@?f T'"
      when "T_upds = []"
      using that by auto
    ultimately show ?thesis using B(13) unfolding ℬ'_def db_eq_def Let_def by presburger
  qed

  have B'14: "db_upds_consts_fresh (𝒜@T') (fv_lsst (𝒜@T') ∪ X') 𝒦"
```

```
proof (unfold db_upds_consts_fresh_def; intro ballI allI impI; elim exE)
  fix x s n m
  assume x: "x ∈ fv_lsst (𝒜@T') ∪ X'"
    and n: "insert⟨Fun (Val n) [],s⟩ ∈ set (unlabel (𝒜@T')) ∨
            delete⟨Fun (Val n) [],s⟩ ∈ set (unlabel (𝒜@T'))"
          (is "?A (𝒜@T')")
    and m: "ℐ x = Fun (PubConst Value m) []"

  have A_cases: "?A 𝒜 ∨ ?A T'" using n by force

  have n_in_case: "n ∈ σ_vals" when A: "?A T'"
  proof -
    obtain t s' where t:
      "insert⟨t,s'⟩ ∈ set (unlabel (transaction_strand T)) ∨
       delete⟨t,s'⟩ ∈ set (unlabel (transaction_strand T))"
      "Fun (Val n) [] = t · ϑ"
    using A dual_lsst_unlabel_steps_iff(4,5)
          stateful_strand_step_mem_substD(4,5)[of _ _ _ ϑ]
          subst_lsst_unlabel[of _ ϑ]
    unfolding T'_def by (metis (no_types, opaque_lifting))

    have "Fun (Val n) [] ∈ subst_range ϑ"
      using t transaction_inserts_are_Value_vars(1)[OF T_wf(1,3), of t s']
            transaction_deletes_are_Value_vars(1)[OF T_wf(1,3), of t s']
      by force
    hence "Fun (Val n) [] ∈ subst_range σ"
      using transaction_decl_fresh_renaming_substs_range'(4)[
              OF step.hyps(3,4,5) _ ξ_empty]
      unfolding ϑ_def by blast
    thus ?thesis unfolding σ_vals_def by fast
  qed

  have in_A_case: "𝒦 x ≠ Fun (Val n) []"
    when y: "y ∈ fv_lsst 𝒜 ∪ X'" "ℐ x = ℐ y" "𝒦 x = 𝒥 y" for y
    using A_cases
  proof
    assume "?A 𝒜" thus ?thesis
      using B(14) m y(1,3) unfolding db_upds_consts_fresh_def y(2) by auto
  next
    assume "?A T'"
    hence "n ∈ N'" using n_in_case unfolding N'_def by blast
    moreover have "𝒥 y ∈ trms_lsst ℬ"
      using B(6) y(1) m ik_sst_trms_sst_subset
      unfolding y(2) subst_in_ik_if_subst_pubval_def by blast
    ultimately show ?thesis using B(12) y(3) by fastforce
  qed

  show "𝒦 x ≠ Fun (Val n) []"
  proof (cases "x ∈ fv_lsst T'")
    case True
    note 0 = T'_fv_ℐ_pubval_𝒦_eq_δ_fresh_val[OF True m, unfolded B(9)[symmetric]]
    note 1 = T'_fv_ℐ_pubval_𝒦_eq_𝒥_val[OF True m, unfolded B(9)[symmetric]]

    show ?thesis
    proof (cases "∀y∈ fv_lsst 𝒜 ∪ X'. ℐ y ≠ ℐ x")
      case True show ?thesis
        using A_cases 0[OF True] T_val_fresh_vals n_in_case
        unfolding 𝒜_vals_def by force
    next
      case False
      then obtain y where "y ∈ fv_lsst 𝒜 ∪ X'" "ℐ y = ℐ x" "𝒦 x = 𝒥 y" using 1 by blast
      thus ?thesis using in_A_case by auto
    qed
```

```
  next
    case False
    hence x_in: "x ∈ fv_lsst 𝒜 ∪ X'" using x unfolding unlabel_append fv_sst_append by fast
    hence x_eq: "𝒦 x = 𝒥 x" using 𝒜_fv_𝒦_eq_𝒥 by blast

    show ?thesis using in_A_case[OF x_in _ x_eq] by blast
  qed
qed

have B'2: "?wt_model 𝒦 ℬ'"
proof (unfold welltyped_constraint_model_def; intro conjI)
  have "Γ (𝒦 x) = Γ_v x" for x
  proof -
    have "wt_subst 𝒥" "wt_subst ℐ"
      using B(2) 3(1) unfolding welltyped_constraint_model_def by (blast,blast)
    hence *: "⋀y. Γ (𝒥 y) = Γ_v y" "⋀y. Γ (ℐ y) = Γ_v y" unfolding wt_subst_def by auto

    show ?thesis
    proof (cases "x ∈ fv_lsst T'")
      case True
      note x = this
      show ?thesis
      proof (cases "∃n. ℐ x = Fun (PubConst Value n) []")
        case True thus ?thesis using T'_fv_ℐ_pubval_𝒦_eq_val[OF x] T'_value_vars[OF x] by force
      next
        case False thus ?thesis using x * unfolding 𝒦_def by presburger
      qed
    next
      case False thus ?thesis using *(1) unfolding 𝒦_def by presburger
    qed
  qed
  thus "wt_subst 𝒦" unfolding 𝒦_def wt_subst_def by force

  show "constraint_model 𝒦 ℬ'"
  proof (unfold constraint_model_def; intro conjI)
    have *: "strand_sem_stateful {} {} (unlabel 𝒜) ℐ"
            "strand_sem_stateful {} {} (unlabel ℬ) 𝒥"
            "interpretation_subst ℐ" "interpretation_subst 𝒥"
            "wf_trms (subst_range ℐ)" "wf_trms (subst_range 𝒥)"
      using B(2) 3(1) unfolding welltyped_constraint_model_def constraint_model_def by fast+

    show K0: "subst_domain 𝒦 = UNIV"
    proof -
      have "x ∈ subst_domain 𝒦" for x
      proof (cases "x ∈ fv_lsst T'")
        case True thus ?thesis
          using T'_fv_ℐ_pubval_𝒦_eq_val[OF True] T'_fv_ℐ_val_𝒦_eq_𝒥[OF True] *(3)
          unfolding subst_domain_def by (cases "∃n. ℐ x = Fun (PubConst Value n) []") auto
      next
        case False thus ?thesis using *(4) unfolding 𝒦_def subst_domain_def by auto
      qed
      thus ?thesis by blast
    qed

    have "fv (𝒦 x) = {}" for x
      using interpretation_grounds_all[OF *(3)]
            interpretation_grounds_all[OF *(4)]
      unfolding 𝒦_def by simp
    thus K1: "ground (subst_range 𝒦)" by simp

    have "wf_trm (Fun (Val n) [])" for n by fastforce
    moreover have "wf_trm (ℐ x)" "wf_trm (𝒥 x)" for x using *(5,6) by (fastforce,fastforce)
    ultimately have "wf_trm (𝒦 x)" for x unfolding 𝒦_def by auto
```

```
thus K2: "wf_trms (subst_range 𝒦)" by simp

show "strand_sem_stateful {} {} (unlabel ℬ') 𝒦"
proof (unfold ℬ'_def unlabel_append strand_sem_append_stateful Un_empty_left; intro conjI)
  let ?sem = "λM D A. strand_sem_stateful M D (unlabel A) 𝒦"
  let ?M1 = "ik_lsst ℬ ·_set 𝒦"
  let ?M2 = "?M1 ∪ (ik_lsst T_val_constr ·_set 𝒦)"
  let ?D1 = "dbupd_sst (unlabel ℬ) 𝒦 {}"
  let ?D2 = "dbupd_sst (unlabel T_val_constr) 𝒦 ?D1"

  show "?sem {} {} ℬ"
    using 𝒥_𝒦_fv_ℬ_eq strand_sem_model_swap[OF _ *(2)] by blast

  show "?sem ?M1 ?D1 T_val_constr"
    using T_val_constr(3,4) strand_sem_stateful_if_no_send_or_check
    unfolding list_all_iff by blast

  have D2: "?D2 = ?D1 ∪ {(t · 𝒦, s · 𝒦) | t s. insert⟨t,s⟩ ∈ set (unlabel T_val_constr)}"
    using T_val_constr(3,4) dbupd_sst_no_deletes
    unfolding list_all_iff by blast

  have K3: "interpretation_subst 𝒦"
    using K0 K1 by argo

  have rcv_ϑ_is_α: "t · ϑ = t · α"
    when t: "(l,receive⟨ts⟩) ∈ set (transaction_receive T)" "t ∈ set ts" for l ts t
  proof -
    have "fv t ⊆ fv_lsst (transaction_receive T)"
      using t(2) stateful_strand_step_fv_subset_cases(2)[OF unlabel_in[OF t(1)]] by auto
    hence "t · σ = t" using t σ_dom σ_ran admissible_transactionE(12,13)[OF T_adm] by blast
    thus ?thesis unfolding ϑ_def ξ_empty by simp
  qed

  have eq_ϑ_is_α: "t · ϑ = t · α" "s · ϑ = s · α"
    when t: "(l,⟨ac: t ≐ s⟩) ∈ set (transaction_checks T)" for l ac t s
  proof -
    have "fv t ∪ fv s ⊆ fv_lsst (transaction_checks T)"
      using stateful_strand_step_fv_subset_cases(3)[OF unlabel_in[OF t]] by auto
    hence "t · σ = t" "s · σ = s"
      using t σ_dom σ_ran admissible_transactionE(12,13)[OF T_adm] by (blast, blast)
    thus "t · ϑ = t · α" "s · ϑ = s · α" unfolding ϑ_def ξ_empty by simp_all
  qed

  have noteq_ϑ_is_α: "t · ϑ = t · α" "s · ϑ = s · α"
    when t: "(l,⟨t != s⟩) ∈ set (transaction_checks T)" for l t s
  proof -
    have "fv t ∪ fv s ⊆ fv_lsst (transaction_checks T)"
      using stateful_strand_step_fv_subset_cases(8)[OF unlabel_in[OF t]] by auto
    hence "t · σ = t" "s · σ = s"
      using t σ_dom σ_ran admissible_transactionE(12,13)[OF T_adm] by (blast, blast)
    thus "t · ϑ = t · α" "s · ϑ = s · α" unfolding ϑ_def ξ_empty by simp_all
  qed

  have in_ϑ_is_α: "t · ϑ = t · α" "s · ϑ = s · α"
    when t: "(l,⟨ac: t ∈ s⟩) ∈ set (transaction_checks T)" for l ac t s
  proof -
    have "fv t ∪ fv s ⊆ fv_lsst (transaction_checks T)"
      using stateful_strand_step_fv_subset_cases(6)[OF unlabel_in[OF t]] by auto
    hence "t · σ = t" "s · σ = s"
      using t σ_dom σ_ran admissible_transactionE(12,13)[OF T_adm] by (blast, blast)
    thus "t · ϑ = t · α" "s · ϑ = s · α" unfolding ϑ_def ξ_empty by simp_all
  qed
```

```
have notin_ϑ_is_α: "t · ϑ = t · α" "s · ϑ = s · α"
  when t: "(l,⟨t not in s⟩) ∈ set (transaction_checks T)" for l t s
proof -
  have "fv t ∪ fv s ⊆ fv_lsst (transaction_checks T)"
    using stateful_strand_step_fv_subset_cases(9)[OF unlabel_in[OF t]] by auto
  hence "t · σ = t" "s · σ = s"
    using t σ_dom σ_ran admissible_transactionE(12,13)[OF T_adm] by (blast, blast)
  thus "t · ϑ = t · α" "s · ϑ = s · α" unfolding ϑ_def ξ_empty by simp_all
qed

have T'_trm_no_val: "∄n. s = Fun (Val n) [] ∨ s = Fun (PubConst Value n) []"
  when t: "t ∈ trms_transaction T" "s ⊑ t · α" for t s
proof -
  have ?thesis when "s ⊑ t"
    using that t admissible_transactions_no_Value_consts'[OF T_adm]
          admissible_transactions_no_PubConsts[OF T_adm]
    by blast
  moreover have "Fun k [] ⊑ u" when "Fun k [] ⊑ u · α" for k u using that
  proof (induction u)
    case (Var x) thus ?case
      using transaction_renaming_subst_is_renaming(2)[OF step.hyps(5), of x] by fastforce
  qed auto
  ultimately show ?thesis using t by blast
qed

define flt1 where "flt1 ≡ λA::('fun,'atom,'sets,'lbl) prot_constr.
                           filter is_Update (unlabel A)"
define flt2 where "flt2 ≡ λA::('fun,'atom,'sets,'lbl) prot_constr.
                           filter (λa. ∄l t ts. a = (l, insert⟨t,⟨s_val⟨ts⟩⟩_s⟩)) A"
define flt3 where "flt3 ≡ λA::(('fun,'atom,'sets,'lbl) prot_fun,
                               ('fun,'atom,'sets,'lbl) prot_var) stateful_strand.
                           filter (λa. ∄t ts. a = insert⟨t,⟨s_val⟨ts⟩⟩_s⟩) A"

have flt2_subset: "set (unlabel (flt2 A)) ⊆ set (unlabel A)" for A
  unfolding flt2_def unlabel_def by auto

have flt2_unlabel: "unlabel (flt2 A) = flt3 (unlabel A)" for A
  unfolding flt2_def flt3_def by (induct A) auto

have flt2_suffix:
    "suffix (filter (λa. ∄t ts. a = insert⟨t,⟨s_val⟨ts⟩⟩_s⟩) A) (unlabel (flt2 B))"
  when "suffix A (unlabel B)" for A B
  using that unfolding flt2_def by (induct B arbitrary: A rule: List.rev_induct) auto

have flt_AB: "flt1 (flt2 𝒜) = flt1 (flt2 ℬ)"
proof -
  have *: "flt1 (flt2 𝒜) = filter is_Update (flt3 (unlabel 𝒜))"
          "flt1 (flt2 ℬ) = filter is_Update (flt3 (unlabel ℬ))"
    using flt2_unlabel unfolding flt1_def by presburger+

  have **: "filter is_Update (flt3 C) = flt3 (filter is_Update C)" for C
  proof (induction C)
    case Nil thus ?case unfolding flt3_def by force
  next
    case (Cons c C) thus ?case unfolding flt3_def by (cases c) auto
  qed

  show ?thesis
  proof (cases "T_upds = []")
    case True
    hence "filter is_Update (unlabel 𝒜) = filter is_Update (unlabel ℬ)"
      using B(13) unfolding db_eq_def by fastforce
    thus ?thesis using ** unfolding * by presburger
```

```
    next
      case False thus ?thesis
        using B(13) unfolding flt1_def flt2_def db_eq_def Let_def by force
    qed
  qed

  have A_setops_Fun: "∀t s. insert⟨t,s⟩ ∈ set (unlabel 𝒜) ⟶ (∃g ts. s = Fun g ts)"
    using reachable_constraints_setops_form[OF step.hyps(1) P]
    unfolding setops_sst_def by fastforce

  have A_insert_delete_not_subterm:
      "ℐ x = 𝒦 x ∨ (¬(ℐ x ⊑ t) ∧ ¬(ℐ x ⊑ s) ∧ ¬(𝒦 x ⊑ t) ∧ ¬(𝒦 x ⊑ s))"
    when x: "x ∈ fv_lsst 𝒜 ∪ fv_lsst T' ∪ fv t ∪ fv s"
      and x_neq: "ℐ x ≠ 𝒦 x"
      and ts: "insert⟨t,s⟩ ∈ set (unlabel 𝒜) ∨ delete⟨t,s⟩ ∈ set (unlabel 𝒜)"
    for x t s
  proof -
    have x_in: "x ∈ fv_lsst 𝒜 ∪ fv_lsst T'"
      using ts x stateful_strand_step_fv_subset_cases(4,5) by blast

    note ts' = reachable_constraints_insert_delete_form[OF step.hyps(1) P ts]

    have *: "ℐ x = 𝒦 x" when n: "ℐ x = Fun (Val n) []" for n
      using n B'5 x_in
      unfolding subst_eq_on_privvals_def unlabel_append fv_sst_append
      by blast

    have **: "¬(ℐ x ⊑ t)" "¬(ℐ x ⊑ s)" "¬(𝒦 x ⊑ t)" "¬(𝒦 x ⊑ s)"
      when n: "ℐ x = Fun (PubConst Value n) []" for n
    proof -
      show "¬(ℐ x ⊑ s)"
        using ts'(1) x_in 3(2,3) unfolding valconst_cases_def by fastforce

      show "¬(𝒦 x ⊑ s)"
        using ts'(1) x_in B'3
        unfolding valconsts_only_def unlabel_append fv_sst_append by force

      show "¬(ℐ x ⊑ t)" using n ts'(3) by fastforce

      from ts'(3) have "𝒦 x ≠ t"
      proof
        assume "∃y. t = Var y" thus ?thesis
          using B'3 x_in unfolding valconsts_only_def by force
      next
        assume "∃k. t = Fun (Val k) []" thus ?thesis
          using B'14 n x_in ts unfolding db_upds_consts_fresh_def by auto
      qed
      thus "¬(𝒦 x ⊑ t)" using ts'(3) by auto
    qed

    show ?thesis using * ** 3(2,3) x_in unfolding valconst_cases_def by fast
  qed

  have flt2_insert_in_iff:
      "insert⟨u,v⟩ ∈ set (unlabel A) ⟷ insert⟨u,v⟩ ∈ set (unlabel (flt2 A))"
    (is "?A A ⟷ ?B A")
    when h: "s = ⟨h⟩_s" "h ≠ s_val" and t: "(t · I,s · I) = (u,v) ·_p I"
    for t s h u v A and I::"('fun,'atom,'sets,'lbl) prot_subst"
  proof
    show "?B A ⟹ ?A A" using flt2_subset by fast
    show "?A A ⟹ ?B A"
    proof (induction A)
      case (Cons a A)
```

```
    obtain l b where a: "a = (l,b)" by (metis surj_pair)
    show ?case
    proof (cases "b = insert⟨u,v⟩")
      case True thus ?thesis using h t unfolding a flt2_def by force
    next
      case False thus ?thesis using Cons.prems Cons.IH unfolding a flt2_def by auto
    qed
  qed simp
qed

have flt2_inset_iff:
    "(t · 𝒦, s · 𝒦) ∈ dbupd_sst (unlabel (flt2 ℬ)) 𝒦 {} ⟷
     (t · 𝒦, s · 𝒦) ∈ dbupd_sst (unlabel ℬ) 𝒦 {}"
  (is "?A ⟷ ?B")
  when h: "s = ⟨h⟩_s" "h ≠ s_val"
  for t s h
proof
  let ?C1 = "λu v B C. suffix (delete⟨u,v⟩#B) (unlabel C)"
  let ?C2 = "λt s u v. (t,s) = (u,v) ·_p 𝒦"
  let ?C3 = "λt s C. ∃u v. ?C2 t s u v ∧ insert⟨u,v⟩ ∈ set C"
  let ?D = "λt s C. ∀u v B. ?C1 u v B C ∧ ?C2 t s u v ⟶ ?C3 t s B"

  let ?db = "λC D. dbupd_sst C 𝒦 D"

  have "?C3 t s B"
    when "?D t s (flt2 ℬ)" "?C1 u v B ℬ" "?C2 t s u v" for u v B t s
    using that flt2_suffix flt2_subset by fastforce
  thus "?A ⟹ ?B" using flt2_subset unfolding dbupd_sst_in_iff by blast

  show ?A when ?B using that
  proof (induction ℬ rule: List.rev_induct)
    case (snoc a A)
    obtain l b where a: "a = (l,b)" by (metis surj_pair)

    have *:
        "?db (unlabel (A@[a])) {} = ?db [b] (?db (unlabel A) {})"
        "?db (unlabel (flt2 (A@[a]))) {} =
         ?db (unlabel (flt2 [a])) (?db (unlabel (flt2 A)) {})"
      using dbupd_sst_append[of _ _ 𝒦 "{}"] unfolding a flt2_def by auto

    show ?case
    proof (cases "∃u v. b = insert⟨u,v⟩ ∧ (t · 𝒦, s · 𝒦) = (u,v) ·_p 𝒦")
      case True
      then obtain u v where "b = insert⟨u,v⟩" "(t · 𝒦, s · 𝒦) = (u, v) ·_p 𝒦" by force
      thus ?thesis using h *(2) unfolding a flt2_def by auto
    next
      case False
      hence IH: "(t · 𝒦, s · 𝒦) ∈ dbupd_sst (unlabel (flt2 A)) 𝒦 {}"
        using snoc.prems snoc.IH unfolding *(1) by (cases b) auto

      show ?thesis
      proof (cases "is_Delete b")
        case True
        then obtain u v where b: "b = delete⟨u,v⟩" by (cases b) auto

        have b': "unlabel (flt2 [a]) = [b]"
                 "unlabel (flt2 (A@[a])) = unlabel (flt2 A)@[b]"
          unfolding a flt2_def b by (fastforce,fastforce)

        have "(t · 𝒦, s · 𝒦) ≠ (u,v) ·_p 𝒦" using *(1) snoc.prems unfolding b' b by simp
        thus ?thesis using *(2) IH unfolding b' b by simp
      next
        case False thus ?thesis using *(2) IH unfolding a flt2_def by (cases b) auto
```

```
        qed
      qed
    qed simp
  qed

  have inset_model_swap:
      "(t · I, s · I) ∈ dbupd_sst (unlabel A) I {} ⟷
        (t · K, s · K) ∈ dbupd_sst (unlabel B) K {}"
    (is "?in I (unlabel A) ⟷ ?in K (unlabel B)")
    when h: "s = ⟨h⟩_s"
            "h ≠ s_val ∨ filter is_Update (unlabel A) = filter is_Update (unlabel B)"
      and t: "t = Var tx"
      and t_s_fv: "fv t ∪ fv s ⊆ fv_lsst T'"
      and q: "∀x ∈ fv t ∪ fv s.
                I x = K x ∨ (¬(I x ⊑ t) ∧ ¬(I x ⊑ s) ∧ ¬(K x ⊑ t) ∧ ¬(K x ⊑ s))"
             "∀x ∈ fv_lsst A ∪ fv t ∪ fv s. ∃c. I x = Fun c []"
             "∀x ∈ fv_lsst A ∪ fv t ∪ fv s. ∃c. K x = Fun c []"
             "∀x ∈ fv_lsst A ∪ fv t ∪ fv s. ∀y ∈ fv_lsst A ∪ fv t ∪ fv s.
                I x = I y ⟷ K x = K y"
    for t s h tx
  proof -
    let ?upds = "λA. filter is_Update (unlabel A)"

    have flt2_fv: "fv_lsst (flt2 A) ⊆ fv_lsst A"
      using fv_sst_mono[OF flt2_subset[of A]] by blast

    have upds_fv: "fv_sst (?upds A) ⊆ fv_lsst A" by auto

    have flt2_upds_fv: "fv_sst (?upds (flt2 A)) ⊆ fv_sst (?upds A)"
      using flt2_subset[of A] by auto

    have h_neq: "Set h ≠ (Set s_val::('fun,'atom,'sets,'lbl) prot_fun)"
      when "h ≠ s_val"
      using that by simp

    have *: "⋃(fv_pair ` {}) = {}" "{} ·_pset I = {}" "{} ·_pset K = {}" by blast+

    have "?in I (?upds (flt2 A)) ⟷ ?in K (?upds (flt2 A))"
    proof
      let ?X = "fv_sst (?upds (flt2 A)) ∪ fv t ∪ fv s ∪
                ⋃(fv_pair ` ({}::(('fun,'atom,'sets,'lbl) prot_term ×
                                 ('fun,'atom,'sets,'lbl) prot_term) set))"

      let ?q0 = "λδ ϑ.
             ∀x ∈ ?X.
               δ x = ϑ x ∨
               (¬(δ x ⊑ t) ∧ ¬(δ x ⊑ s) ∧¬(ϑ x ⊑ t) ∧ ¬(ϑ x ⊑ s) ∧
                (∀(u,v) ∈ {}. ¬(δ x ⊑ u) ∧ ¬(δ x ⊑ v) ∧ ¬(ϑ x ⊑ u) ∧ ¬(ϑ x ⊑ v)) ∧
                (∀u v. insert⟨u,v⟩ ∈ set (?upds (flt2 A)) ∨
                       delete⟨u,v⟩ ∈ set (?upds (flt2 A)) ⟶
                         ¬(δ x ⊑ u) ∧ ¬(δ x ⊑ v) ∧ ¬(ϑ x ⊑ u) ∧ ¬(ϑ x ⊑ v)))"

      let ?q1 = "λδ. ∀x ∈ ?X. ∃c. δ x = Fun c []"

      let ?q2 = "λδ ϑ. ∀x ∈ ?X. ∀y ∈ ?X. δ x = δ y ⟷ ϑ x = ϑ y"

      have q0: "?q0 I K" "?q0 K I"
      proof -
        have upd_ex:
            "∃u v. x ∈ fv u ∪ fv v ∧
                   (insert⟨u,v⟩ ∈ set (?upds A) ∨ delete⟨u,v⟩ ∈ set (?upds A))"
          when "x ∈ fv_sst (?upds A)" for x and A::"('fun,'atom,'sets,'lbl) prot_constr"
          using that
```

```
    proof (induction A)
      case (Cons a A)
      obtain l b where a: "a = (l,b)" by (metis surj_pair)
      show ?case using Cons.IH Cons.prems unfolding a by (cases b) auto
    qed simp

    have "¬(ℐ x ⊑ t)" "¬(ℐ x ⊑ s)" "¬(𝒦 x ⊑ t)" "¬(𝒦 x ⊑ s)"
      when x: "x ∈ fv_sst (?upds (flt2 𝒜)) ∪ fv t ∪ fv s"
        and x_neq: "ℐ x ≠ 𝒦 x"
      for x
    proof -
      have "¬(ℐ x ⊑ t) ∧ ¬(ℐ x ⊑ s) ∧ ¬(𝒦 x ⊑ t) ∧ ¬(𝒦 x ⊑ s)"
      proof (cases "x ∈ fv t ∪ fv s")
        case True thus ?thesis using q(1) x_neq by blast
      next
        case False
        hence "x ∈ fv_lsst 𝒜" using x flt2_upds_fv upds_fv by blast
        hence "∃n. 𝒦 x = Fun (Val n) []"
              "∃n. ℐ x = Fun (Val n) [] ∨ ℐ x = Fun (PubConst Value n) []"
          using B'3 3(2)
          unfolding valconst_cases_def valconsts_only_def unlabel_append fv_sst_append
          by (blast, blast)
        thus ?thesis unfolding t h(1) by auto
      qed
      thus "¬(ℐ x ⊑ t)" "¬(ℐ x ⊑ s)" "¬(𝒦 x ⊑ t)" "¬(𝒦 x ⊑ s)" by simp_all
    qed
    moreover have "¬(ℐ x ⊑ u)" "¬(ℐ x ⊑ v)" "¬(𝒦 x ⊑ u)" "¬(𝒦 x ⊑ v)"
      when x: "x ∈ fv_sst (?upds (flt2 𝒜)) ∪ fv t ∪ fv s"
        and x_neq: "ℐ x ≠ 𝒦 x"
        and uv: "insert⟨u,v⟩ ∈ set (?upds (flt2 𝒜)) ∨
                 delete⟨u,v⟩ ∈ set (?upds (flt2 𝒜))"
      for x u v
    proof -
      have uv': "insert⟨u,v⟩ ∈ set (unlabel 𝒜) ∨ delete⟨u,v⟩ ∈ set (unlabel 𝒜)"
        using uv flt2_subset by auto

      have x_in: "x ∈ fv_lsst 𝒜 ∪ fv_lsst T' ∪ fv u ∪ fv v"
        using t_s_fv x flt2_upds_fv upds_fv by blast

      show "¬(ℐ x ⊑ u)" "¬(ℐ x ⊑ v)" "¬(𝒦 x ⊑ u)" "¬(𝒦 x ⊑ v)"
        using x_neq A_insert_delete_not_subterm[OF x_in x_neq uv'] by simp_all
    qed
    ultimately show "?q0 ℐ 𝒦" unfolding upd_ex unfolding *
      by (metis (no_types, lifting) empty_iff sup_bot_right)
    thus "?q0 𝒦 ℐ" by (metis (lifting) empty_iff)
  qed

  have q1: "?q1 ℐ" "?q1 𝒦"
    using q(2,3) flt2_upds_fv upds_fv by (blast,blast)

  have q2: "?q2 ℐ 𝒦" "?q2 𝒦 ℐ"
    using q(4) flt2_upds_fv upds_fv unfolding * by (blast,blast)

  show "?in ℐ (?upds (flt2 𝒜)) ⟹ ?in 𝒦 (?upds (flt2 𝒜))"
    using dbupd_sst_subst_const_swap[OF _ q0(1) q1(1,2) q2(1)] by force

  show "?in 𝒦 (?upds (flt2 𝒜)) ⟹ ?in ℐ (?upds (flt2 𝒜))"
    using dbupd_sst_subst_const_swap[OF _ q0(2) q1(2,1) q2(2)] by force
qed
hence flt2_subst_swap: "?in ℐ (unlabel (flt2 𝒜)) ⟷ ?in 𝒦 (unlabel (flt2 𝒜))"
  using dbupd_sst_filter by blast
```

```
have "?in 𝓘 (?upds 𝓐) ⟷ ?in 𝓚 (?upds 𝓐)"
proof
  let ?X = "fv_sst (?upds 𝓐) ∪ fv t ∪ fv s ∪
          ⋃(fv_pair ` ({}::(('fun,'atom,'sets,'lbl) prot_term ×
                          ('fun,'atom,'sets,'lbl) prot_term) set))"

  let ?q0 = "λδ ϑ.
        ∀x ∈ ?X.
          δ x = ϑ x ∨
          (¬(δ x ⊑ t) ∧ ¬(δ x ⊑ s) ∧¬(ϑ x ⊑ t) ∧ ¬(ϑ x ⊑ s) ∧
           (∀(u,v) ∈ {}. ¬(δ x ⊑ u) ∧ ¬(δ x ⊑ v) ∧ ¬(ϑ x ⊑ u) ∧ ¬(ϑ x ⊑ v)) ∧
           (∀u v. insert⟨u,v⟩ ∈ set (?upds 𝓐) ∨
                  delete⟨u,v⟩ ∈ set (?upds 𝓐) ⟶
                    ¬(δ x ⊑ u) ∧ ¬(δ x ⊑ v) ∧ ¬(ϑ x ⊑ u) ∧ ¬(ϑ x ⊑ v)))"

  let ?q1 = "λδ. ∀x ∈ ?X. ∃c. δ x = Fun c []"

  let ?q2 = "λδ ϑ. ∀x ∈ ?X. ∀y ∈ ?X. δ x = δ y ⟷ ϑ x = ϑ y"

  have q0: "?q0 𝓘 𝓚" "?q0 𝓚 𝓘"
  proof -
    have upd_ex:
        "∃u v. x ∈ fv u ∪ fv v ∧
               (insert⟨u,v⟩ ∈ set (?upds A) ∨ delete⟨u,v⟩ ∈ set (?upds A))"
      when "x ∈ fv_sst (?upds A)" for x and A::"('fun,'atom,'sets,'lbl) prot_constr"
      using that
    proof (induction A)
      case (Cons a A)
      obtain l b where a: "a = (l,b)" by (metis surj_pair)
      show ?case using Cons.IH Cons.prems unfolding a by (cases b) auto
    qed simp

    have "¬(𝓘 x ⊑ t)" "¬(𝓘 x ⊑ s)" "¬(𝓚 x ⊑ t)" "¬(𝓚 x ⊑ s)"
      when x: "x ∈ fv_sst (?upds 𝓐) ∪ fv t ∪ fv s"
        and x_neq: "𝓘 x ≠ 𝓚 x"
      for x
    proof -
      have "¬(𝓘 x ⊑ t) ∧ ¬(𝓘 x ⊑ s) ∧ ¬(𝓚 x ⊑ t) ∧ ¬(𝓚 x ⊑ s)"
      proof (cases "x ∈ fv t ∪ fv s")
        case True thus ?thesis using q(1) x_neq by blast
      next
        case False
        hence "x ∈ fv_lsst 𝓐" using x flt2_upds_fv upds_fv by blast
        hence "∃n. 𝓚 x = Fun (Val n) []"
              "∃n. 𝓘 x = Fun (Val n) [] ∨ 𝓘 x = Fun (PubConst Value n) []"
          using B'3 3(2)
          unfolding valconst_cases_def valconsts_only_def unlabel_append fv_sst_append
          by (blast, blast)
        thus ?thesis unfolding t h(1) by auto
      qed
      thus "¬(𝓘 x ⊑ t)" "¬(𝓘 x ⊑ s)" "¬(𝓚 x ⊑ t)" "¬(𝓚 x ⊑ s)" by simp_all
    qed
    moreover have "¬(𝓘 x ⊑ u)" "¬(𝓘 x ⊑ v)" "¬(𝓚 x ⊑ u)" "¬(𝓚 x ⊑ v)"
      when x: "x ∈ fv_sst (?upds 𝓐) ∪ fv t ∪ fv s"
        and x_neq: "𝓘 x ≠ 𝓚 x"
        and uv: "insert⟨u,v⟩ ∈ set (?upds 𝓐) ∨
                 delete⟨u,v⟩ ∈ set (?upds 𝓐)"
      for x u v
    proof -
      have uv': "insert⟨u,v⟩ ∈ set (unlabel 𝓐) ∨ delete⟨u,v⟩ ∈ set (unlabel 𝓐)"
        using uv flt2_subset by auto

      have x_in: "x ∈ fv_lsst 𝓐 ∪ fv_lsst T' ∪ fv u ∪ fv v"
```

```
        using t_s_fv x flt2_upds_fv upds_fv by blast

      show "¬(ℐ x ⊑ u)" "¬(ℐ x ⊑ v)" "¬(𝒦 x ⊑ u)" "¬(𝒦 x ⊑ v)"
        using x_neq A_insert_delete_not_subterm[OF x_in x_neq uv'] by simp_all
    qed
    ultimately show "?q0 ℐ 𝒦" unfolding upd_ex unfolding *
      by (metis (no_types, lifting) empty_iff sup_bot_right)
    thus "?q0 𝒦 ℐ" by (metis (lifting) empty_iff)
  qed

  have q1: "?q1 ℐ" "?q1 𝒦"
    using q(2,3) flt2_upds_fv upds_fv by (blast,blast)

  have q2: "?q2 ℐ 𝒦" "?q2 𝒦 ℐ"
    using q(4) flt2_upds_fv upds_fv unfolding * by (blast,blast)

  show "?in ℐ (?upds 𝒜) ⟹ ?in 𝒦 (?upds 𝒜)"
    using dbupd_sst_subst_const_swap[OF _ q0(1) q1(1,2) q2(1)] by force

  show "?in 𝒦 (?upds 𝒜) ⟹ ?in ℐ (?upds 𝒜)"
    using dbupd_sst_subst_const_swap[OF _ q0(2) q1(2,1) q2(2)] by force
qed
hence db_subst_swap:
    "?in ℐ (unlabel 𝒜) ⟷ ?in 𝒦 (unlabel 𝒜)"
  using dbupd_sst_filter by blast

have "?in 𝒦 (unlabel ℬ)" when A: "?in ℐ (unlabel 𝒜)" using h(2)
proof
  assume h': "h ≠ s_val"
  have "?in ℐ (unlabel (flt2 𝒜))"
    using A flt2_unlabel dbupd_sst_set_term_neq_in_iff[OF h_neq[OF h'] A_setops_Fun]
    unfolding h(1) flt3_def by simp
  hence "?in 𝒦 (unlabel (flt2 𝒜))" using flt2_subst_swap by blast
  hence "?in 𝒦 (flt1 (flt2 𝒜))" using dbupd_sst_filter unfolding flt1_def by blast
  hence "?in 𝒦 (flt1 (flt2 ℬ))" using flt_AB by simp
  hence "?in 𝒦 (unlabel (flt2 ℬ))" using dbupd_sst_filter unfolding flt1_def by blast
  thus ?thesis using flt2_inset_iff[OF h(1) h'] by fast
next
  assume h': "filter is_Update (unlabel 𝒜) = filter is_Update (unlabel ℬ)"
  have "?in 𝒦 (unlabel 𝒜)" using A db_subst_swap by blast
  hence "?in 𝒦 (flt1 𝒜)" using dbupd_sst_filter unfolding flt1_def by blast
  hence "?in 𝒦 (flt1 ℬ)" using h' unfolding flt1_def by simp
  thus ?thesis using dbupd_sst_filter unfolding flt1_def by blast
qed
moreover have "¬?in 𝒦 (unlabel ℬ)" when A: "¬?in ℐ (unlabel 𝒜)" using h(2)
proof
  assume h': "h ≠ s_val"
  have "¬?in ℐ (unlabel (flt2 𝒜))"
    using A flt2_unlabel dbupd_sst_set_term_neq_in_iff[OF h_neq[OF h'] A_setops_Fun]
    unfolding h(1) flt3_def by simp
  hence "¬?in 𝒦 (unlabel (flt2 𝒜))" using flt2_subst_swap by blast
  hence "¬?in 𝒦 (flt1 (flt2 𝒜))" using dbupd_sst_filter unfolding flt1_def by blast
  hence "¬?in 𝒦 (flt1 (flt2 ℬ))" using flt_AB by simp
  hence "¬?in 𝒦 (unlabel (flt2 ℬ))" using dbupd_sst_filter unfolding flt1_def by blast
  thus ?thesis using flt2_inset_iff[OF h(1) h'] by fast
next
  assume h': "filter is_Update (unlabel 𝒜) = filter is_Update (unlabel ℬ)"
  have "¬?in 𝒦 (unlabel 𝒜)" using A db_subst_swap by blast
  hence "¬?in 𝒦 (flt1 𝒜)" using dbupd_sst_filter unfolding flt1_def by blast
  hence "¬?in 𝒦 (flt1 ℬ)" using h' unfolding flt1_def by simp
  thus ?thesis using dbupd_sst_filter unfolding flt1_def by blast
qed
ultimately show ?thesis by blast
```

```
qed

have "?M2 ⊢ t · ϑ · 𝒦"
  when ts: "(l, receive⟨ts⟩) ∈ set (transaction_receive T)" "t ∈ set ts" for l t ts
proof -
  have *: "ik_lsst 𝒜 ·_set ℐ ⊢ t · ϑ · ℐ" using 5 ts by blast

  note tϑα = rcv_ϑ_is_α[OF ts]

  have t_T'_trm: "t ∈ trms_transaction T"
    using trms_sst_memI(2)[OF unlabel_in[OF ts(1)] ts(2)]
    unfolding trms_transaction_unfold by blast

  have t_T'_trm': "t · ϑ ∈ trms_lsst T'"
    using trms_sst_memI(2)[
            OF stateful_strand_step_subst_inI(2)[
                OF unlabel_in[OF ts(1)], unfolded unlabel_subst]]
          ts(2)
    unfolding T'_def trms_sst_unlabel_dual_lsst_eq trms_transaction_subst_unfold by auto

  note t_no_val = T'_trm_no_val[OF t_T'_trm, unfolded tϑα[symmetric]]

  have t_fv_T': "fv (t · ϑ) ⊆ fv_lsst T'"
    using ts(2) stateful_strand_step_fv_subset_cases(2)[
            OF stateful_strand_step_subst_inI(2)[OF unlabel_in[OF ts(1)], of ϑ]]
    unfolding T'_def unlabel_subst fv_sst_unlabel_dual_lsst_eq fv_transaction_subst_unfold
    by auto

  have ik_B_fv_subset: "fv_set (ik_lsst ℬ) ⊆ fv_lsst ℬ"
    by (meson UnE fv_ik_sst_is_fv_sst subset_iff)

  let ?fresh_vals = "(λn. Fun (Val n) []) ` T_val_fresh_vals"

  have q0: "ik_lsst ℬ ·_set ℐ ⊢ t · ϑ · ℐ" using * B(10) by (blast intro: ideduct_mono)

  have q1: "∀x ∈ fv_set (ik_lsst ℬ) ∪ fv (t · ϑ). valconst_cases ℐ x"
    using 3(2,3) t_fv_T' ik_B_fv_subset unfolding B(9) by blast

  have q2: "∀x ∈ fv_set (ik_lsst ℬ) ∪ fv (t · ϑ). ∃n. 𝒦 x = Fun (Val n) []"
    using B'3 t_fv_T' ik_B_fv_subset
    unfolding valconsts_only_def unlabel_append fv_sst_append B(9)
    by blast

  have T_val_constr_ik:
    "∃M. ik_lsst T_val_constr = M ∪ ?fresh_vals"
    "∃M. ik_lsst T_val_constr ·_set 𝒦 = (M ·_set 𝒦) ∪ ?fresh_vals"
  proof -
    obtain M where M: "ik_lsst T_val_constr = M ∪ ?fresh_vals"
      using T_val_constr(8) by blast
    have "?fresh_vals ·_set 𝒦 = ?fresh_vals" by fastforce
    thus "∃M. ik_lsst T_val_constr = M ∪ ?fresh_vals"
         "∃M. ik_lsst T_val_constr ·_set 𝒦 = (M ·_set 𝒦) ∪ ?fresh_vals"
      using M by (fastforce, fastforce)
  qed

  have "𝒦 x ∈ ik_lsst ℬ ∪ ik_lsst T_val_constr"
    when x: "x ∈ fv_set (ik_lsst ℬ) ∪ fv (t · ϑ)" "ℐ x = Fun (PubConst Value n) []" for x n
    using x(1) B'6'[OF _ x(2)] B'6''[OF _ x(2)] t_fv_T' ik_B_fv_subset
    unfolding B(9) unlabel_append fv_sst_append by blast
  hence q3: "∀x ∈ fv_set (ik_lsst ℬ) ∪ fv (t · ϑ).
              (∃n. ℐ x = Fun (PubConst Value n) []) ⟶ 𝒦 x ∈ ik_lsst ℬ ∪ ?fresh_vals"
    using T_val_constr_ik(1) T_val_constr(8) q2
    unfolding B(9) ℬ'_def unlabel_append ik_sst_append fv_sst_append
```

```
      by (metis (no_types, lifting) UnE UnI1 UnI2 image_iff mem_Collect_eq)

    have q4: "∀x ∈ fv_set (ik_lsst ℬ) ∪ fv (t · ϑ). (∃n. ℐ x = Fun (Val n) []) ⟶ ℐ x = 𝒦 x"
      using B'5 t_fv_T' ik_B_fv_subset
      unfolding subst_eq_on_privvals_def B(9) unlabel_append fv_sst_append
      by blast

    have q5: "∀x ∈ fv_set (ik_lsst ℬ) ∪ fv (t · ϑ). ∀y ∈ fv_set (ik_lsst ℬ) ∪ fv (t · ϑ).
                ℐ x = ℐ y ⟷ 𝒦 x = 𝒦 y"
      using B'7 t_fv_T' ik_B_fv_subset
      unfolding subst_eq_iff_def B(9) unlabel_append fv_sst_append
      by blast

    have q6: "∀n. ¬(Fun (PubConst Value n) [] ⊑_set insert (t · ϑ) (ik_lsst ℬ))"
    proof -
      have "∄n. s = Fun (PubConst Value n) []" when s: "s ⊑_set trms_lsst ℬ'" for s
      proof -
        have "f ≠ PubConst Value n" when f: "f ∈ funs_term s" for f n
          using f s reachable_constraints_val_funs_private(1)[OF B'1 P, of f]
          unfolding is_PubConstValue_def is_PubConst_def the_PubConst_type_def
          by (metis (mono_tags, lifting) UN_I funs_term_subterms_eq(2) prot_fun.simps(85))
        thus ?thesis by fastforce
      qed
      moreover have "ik_lsst ℬ ⊆ trms_lsst ℬ'"
        using ik_sst_trms_sst_subset unfolding ℬ'_def unlabel_append trms_sst_append by blast
      ultimately show ?thesis
        using t_no_val by blast
    qed

    show ?thesis
      using deduct_val_const_swap[OF q0 q1[unfolded valconst_cases_def] q2 q3 q4 q5 q6]
            T_val_constr_ik(2)
      by (blast intro: ideduct_mono)
  qed
  moreover have "t · ϑ · 𝒦 = s · ϑ · 𝒦"
    when ts: "(l, ⟨ac: t ≐ s⟩) ∈ set (transaction_checks T)" for l ac t s
  proof -
    have q0: "t · ϑ · ℐ = s · ϑ · ℐ" using 5 ts by blast

    have "fv_sstp (⟨ac: (t · ϑ) ≐ (s · ϑ)⟩) ⊆ fv_lsst (transaction_checks T ·_lsst ϑ)"
      using stateful_strand_step_fv_subset_cases(3)[
              OF stateful_strand_step_subst_inI(3)[OF unlabel_in[OF ts], of ϑ]]
      unfolding unlabel_subst by simp
    hence t_s_fv: "fv (t · ϑ) ⊆ fv_lsst T'" "fv (s · ϑ) ⊆ fv_lsst T'"
      unfolding T'_def fv_sst_unlabel_dual_lsst_eq fv_transaction_subst_unfold[of T ϑ]
      by (fastforce, fastforce)

    have "t ∈ trms_transaction T" "s ∈ trms_transaction T"
      using trms_sst_memI(3,4)[OF unlabel_in[OF ts]]
      unfolding trms_transaction_unfold by (blast, blast)
    hence "∄n. u = Fun (Val n) [] ∨ u = Fun (PubConst Value n) []"
      when u: "u ⊑ t · ϑ ∨ u ⊑ s · ϑ" for u
      using u T'_trm_no_val unfolding eq_ϑ_is_α[OF ts] by blast
    hence "¬(ℐ x ⊑ t · ϑ)" "¬(ℐ x ⊑ s · ϑ)"
      when x: "x ∈ fv (t · ϑ) ∪ fv (s · ϑ)" for x
      using x t_s_fv I' by (fast, fast)
    hence q1:
        "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ). ℐ x = 𝒦 x ∨ (¬(ℐ x ⊑ t · ϑ) ∧ ¬(ℐ x ⊑ s · ϑ))"
      by blast

    have q2: "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ). ∃c. ℐ x = Fun c []"
      using t_s_fv 3(3) unfolding valconst_cases_def by blast
```

```
      have q3: "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ). ∃c. 𝒦 x = Fun c []"
        using t_s_fv B'3 unfolding valconsts_only_def unlabel_append fv_sst_append by blast

      have q4: "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ). ∀y ∈ fv (t · ϑ) ∪ fv (s · ϑ).
                ℐ x = ℐ y ⟷ 𝒦 x = 𝒦 y"
        using B'7 t_s_fv unfolding subst_eq_iff_def B(9) unlabel_append fv_sst_append by blast

      show ?thesis by (rule subst_const_swap_eq'[OF q0 q1 q2 q3 q4])
    qed
    moreover have "t · ϑ · 𝒦 ≠ s · ϑ · 𝒦"
      when ts: "(l, ⟨t != s⟩) ∈ set (transaction_checks T)" for l t s
    proof -
      have q0: "t · ϑ · ℐ ≠ s · ϑ · ℐ" using 5 ts by blast

      have "fv_sstp (⟨(t · ϑ) != (s · ϑ)⟩) ⊆ fv_lsst (transaction_checks T ·_lsst ϑ)"
        using stateful_strand_step_fv_subset_cases(8)[
                OF stateful_strand_step_subst_inI(8)[OF unlabel_in[OF ts], of ϑ]]
        unfolding unlabel_subst by simp
      hence t_s_fv: "fv (t · ϑ) ⊆ fv_lsst T'" "fv (s · ϑ) ⊆ fv_lsst T'"
        unfolding T'_def fv_sst_unlabel_dual_lsst_eq fv_transaction_subst_unfold[of T ϑ]
        by (fastforce, fastforce)

      have "t ∈ trms_transaction T" "s ∈ trms_transaction T"
        using trms_sst_memI(9)[OF unlabel_in[OF ts]]
        unfolding trms_transaction_unfold by auto
      hence "∄n. u = Fun (Val n) []" when u: "u ⊑ t · ϑ ∨ u ⊑ s · ϑ" for u
        using u T'_trm_no_val unfolding noteq_ϑ_is_α[OF ts] by blast
      hence "¬(𝒦 x ⊑ t · ϑ)" "¬(𝒦 x ⊑ s · ϑ)"
        when x: "x ∈ fv (t · ϑ) ∪ fv (s · ϑ)" for x
        using x t_s_fv B'3
        unfolding valconsts_only_def unlabel_append fv_sst_append
        by (fast, fast)
      hence q1: "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ). 𝒦 x = ℐ x ∨ (¬(𝒦 x ⊑ t · ϑ) ∧ ¬(𝒦 x ⊑ s ·
ϑ))"
        by blast

      have q2: "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ). ∃c. 𝒦 x = Fun c []"
        using t_s_fv B'3 unfolding valconsts_only_def unlabel_append fv_sst_append by blast

      have q3: "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ). ∃c. ℐ x = Fun c []"
        using t_s_fv 3(3) unfolding valconst_cases_def by blast

      have q4: "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ). ∀y ∈ fv (t · ϑ) ∪ fv (s · ϑ).
                𝒦 x = 𝒦 y ⟷ ℐ x = ℐ y"
        using B'7 t_s_fv unfolding subst_eq_iff_def B(9) unlabel_append fv_sst_append by blast

      show ?thesis using q0 subst_const_swap_eq'[OF _ q1 q2 q3 q4] by fast
    qed
    moreover have "(t · ϑ · 𝒦, s · ϑ · 𝒦) ∈ ?D2"
      when ts: "(l, ⟨ac: t ∈ s⟩) ∈ set (transaction_checks T)" for l ac t s
    proof -
      have s_neq_s_val:
          "s ≠ ⟨s_val⟩_s ∨ filter is_Update (unlabel 𝒜) = filter is_Update (unlabel ℬ)"
      proof (cases "T_upds = []")
        case False thus ?thesis
          using step.hyps(2) ts x_val(7)
          unfolding transaction_strand_def
          by (cases ac) fastforce+
      qed (use B(13)[unfolded db_eq_def] in simp)

      have ts': "⟨ac: t ∈ s⟩ ∈ set (unlabel (transaction_strand T))"
        using ts unlabel_in[OF ts] unfolding transaction_strand_def by fastforce
```

```
    have "fv_sstp (⟨ac: (t · ϑ) ∈ (s · ϑ)⟩) ⊆ fv_lsst (transaction_checks T ·_lsst ϑ)"
      using stateful_strand_step_fv_subset_cases(6)[
              OF stateful_strand_step_subst_inI(6)[OF unlabel_in[OF ts], of ϑ]]
      unfolding unlabel_subst by simp
    hence t_s_fv: "fv (t · ϑ) ⊆ fv_lsst T'" "fv (s · ϑ) ⊆ fv_lsst T'"
      unfolding T'_def fv_sst_unlabel_dual_lsst_eq fv_transaction_subst_unfold[of T ϑ]
      by (fastforce, fastforce)

    have "t ∈ trms_transaction T" "s ∈ trms_transaction T"
      using ts' unfolding trms_sst_def by (force, force)
    hence "∄n. u = Fun (Val n) [] ∨ u = Fun (PubConst Value n) []"
      when u: "u ⊑ t · ϑ ∨ u ⊑ s · ϑ" for u
      using u T'_trm_no_val unfolding in_ϑ_is_α[OF ts] by blast
    hence "¬(𝒦 x ⊑ t · ϑ)" "¬(𝒦 x ⊑ s · ϑ)" "¬(ℐ x ⊑ t · ϑ)" "¬(ℐ x ⊑ s · ϑ)"
      when x: "x ∈ fv (t · ϑ) ∪ fv (s · ϑ)" for x
      using x t_s_fv B'3 I'
      unfolding valconsts_only_def unlabel_append fv_sst_append
      by (fast,fast,fast,fast)
    hence q1: "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ).
                ℐ x = 𝒦 x ∨
                (¬(ℐ x ⊑ t · ϑ) ∧ ¬(ℐ x ⊑ s · ϑ) ∧ ¬(𝒦 x ⊑ t · ϑ) ∧ ¬(𝒦 x ⊑ s · ϑ))"
      by blast

    have q2: "∀x ∈ fv_lsst 𝒜 ∪ fv (t · ϑ) ∪ fv (s · ϑ). ∃c. ℐ x = Fun c []"
      using t_s_fv 3(2,3) unfolding valconst_cases_def by blast

    have q3: "∀x ∈ fv_lsst 𝒜 ∪ fv (t · ϑ) ∪ fv (s · ϑ). ∃c. 𝒦 x = Fun c []"
      using t_s_fv B'3 unfolding valconsts_only_def unlabel_append fv_sst_append by blast

    have q4: "∀x ∈ fv_lsst 𝒜 ∪ fv (t · ϑ) ∪ fv (s · ϑ).
              ∀y ∈ fv_lsst 𝒜 ∪ fv (t · ϑ) ∪ fv (s · ϑ).
                ℐ x = ℐ y ⟷ 𝒦 x = 𝒦 y"
      using B'7 t_s_fv unfolding subst_eq_iff_def B(9) unlabel_append fv_sst_append by blast

    obtain h tx where s: "s = ⟨h⟩_s" and tx: "t = Var tx"
      using ts' transaction_selects_are_Value_vars[OF T_wf(1,2), of t s]
            transaction_inset_checks_are_Value_vars[OF T_adm, of t s]
      by (cases ac) auto

    have h:
        "s · ϑ = ⟨h⟩_s"
        "h ≠ s_val ∨ filter is_Update (unlabel 𝒜) = filter is_Update (unlabel ℬ)"
      using s s_neq_s_val by (simp,blast)

    obtain ty where ty: "t · ϑ = Var ty"
      using tx transaction_renaming_subst_is_renaming(2)[OF step.hyps(5), of tx]
      unfolding in_ϑ_is_α[OF ts] by force

    have "(t · ϑ · ℐ, s · ϑ · ℐ) ∈ dbupd_sst (unlabel 𝒜) ℐ {}" using 5 ts by blast
    hence "(t · ϑ · 𝒦, s · ϑ · 𝒦) ∈ dbupd_sst (unlabel ℬ) 𝒦 {}"
      using inset_model_swap[OF h ty _ q1 q2 q3 q4] t_s_fv by simp
    thus ?thesis unfolding D2 by blast
  qed
  moreover have "(t · ϑ · 𝒦, s · ϑ · 𝒦) ∉ ?D2"
    when ts: "(l, ⟨t not in s⟩) ∈ set (transaction_checks T)" for l t s
  proof -
    have s_neq_s_val:
        "(T_upds ≠ [] ∧ s ≠ ⟨s_val⟩_s) ∨
         (T_upds = [] ∧ filter is_Update (unlabel 𝒜) = filter is_Update (unlabel ℬ))"
    proof (cases "T_upds = []")
      case False thus ?thesis
        using step.hyps(2) ts x_val(7) unfolding transaction_strand_def by force
    qed (use B(13)[unfolded db_eq_def] in simp)
```

```
have ts': "⟨t not in s⟩ ∈ set (unlabel (transaction_strand T))"
  using ts unlabel_in[OF ts] unfolding transaction_strand_def by fastforce

have "fv_sstp (⟨(t · ϑ) not in (s · ϑ)⟩) ⊆ fv_lsst (transaction_checks T ·_lsst ϑ)"
  using stateful_strand_step_fv_subset_cases(9)[
          OF stateful_strand_step_subst_inI(9)[OF unlabel_in[OF ts], of ϑ]]
  unfolding unlabel_subst by simp
hence t_s_fv: "fv (t · ϑ) ⊆ fv_lsst T'" "fv (s · ϑ) ⊆ fv_lsst T'"
  unfolding T'_def fv_sst_unlabel_dual_lsst_eq fv_transaction_subst_unfold[of T ϑ]
  by (fastforce, fastforce)

have "t ∈ trms_transaction T" "s ∈ trms_transaction T"
  using ts' unfolding trms_sst_def by (force, force)
hence "∄n. u = Fun (Val n) [] ∨ u = Fun (PubConst Value n) []"
  when u: "u ⊑ t · ϑ ∨ u ⊑ s · ϑ" for u
  using u T'_trm_no_val unfolding notin_ϑ_is_α[OF ts] by blast
hence "¬(𝒦 x ⊑ t · ϑ)" "¬(𝒦 x ⊑ s · ϑ)" "¬(ℐ x ⊑ t · ϑ)" "¬(ℐ x ⊑ s · ϑ)"
  when x: "x ∈ fv (t · ϑ) ∪ fv (s · ϑ)" for x
  using x t_s_fv B'3 I'
  unfolding valconsts_only_def unlabel_append fv_sst_append
  by (fast,fast,fast,fast)
hence q1: "∀x ∈ fv (t · ϑ) ∪ fv (s · ϑ).
            ℐ x = 𝒦 x ∨
            (¬(ℐ x ⊑ t · ϑ) ∧ ¬(ℐ x ⊑ s · ϑ) ∧ ¬(𝒦 x ⊑ t · ϑ) ∧ ¬(𝒦 x ⊑ s · ϑ))"
  by blast

have q2: "∀x ∈ fv_lsst 𝒜 ∪ fv (t · ϑ) ∪ fv (s · ϑ). ∃c. ℐ x = Fun c []"
  using t_s_fv 3(2,3) unfolding valconst_cases_def by blast

have q3: "∀x ∈ fv_lsst 𝒜 ∪ fv (t · ϑ) ∪ fv (s · ϑ). ∃c. 𝒦 x = Fun c []"
  using t_s_fv B'3 unfolding valconsts_only_def unlabel_append fv_sst_append by blast

have q4: "∀x ∈ fv_lsst 𝒜 ∪ fv (t · ϑ) ∪ fv (s · ϑ).
          ∀y ∈ fv_lsst 𝒜 ∪ fv (t · ϑ) ∪ fv (s · ϑ).
            ℐ x = ℐ y ⟷ 𝒦 x = 𝒦 y"
  using B'7 t_s_fv unfolding subst_eq_iff_def B(9) unlabel_append fv_sst_append by blast

obtain h tx where s: "s = ⟨h⟩_s" and tx: "t = Var tx"
  using transaction_notinset_checks_are_Value_vars(1,2)[OF T_adm ts', of t s] by auto

have h:
    "s · ϑ = ⟨h⟩_s"
    "h ≠ s_val ∨ filter is_Update (unlabel 𝒜) = filter is_Update (unlabel ℬ)"
    "T_upds ≠ [] ⟹ h ≠ s_val"
  using s s_neq_s_val by (simp,blast,blast)

obtain ty where ty: "t · ϑ = Var ty"
  using tx transaction_renaming_subst_is_renaming(2)[OF step.hyps(5), of tx]
  unfolding notin_ϑ_is_α[OF ts] by force

have *: "(t · ϑ · 𝒦, s · ϑ · 𝒦) ≠ (u · 𝒦, v · 𝒦)"
  when u: "insert⟨u,v⟩ ∈ set (unlabel T_val_constr)"
    and h': "h ≠ s_val"
  for u v
proof -
  have "v = ⟨s_val⟩_s" using T_val_constr(9) unlabel_mem_has_label[OF u] by force
  thus ?thesis using h(1) h' by simp
qed

have "(t · ϑ · ℐ, s · ϑ · ℐ) ∉ dbupd_sst (unlabel 𝒜) ℐ {}" using 5 ts by blast
hence **: "(t · ϑ · 𝒦, s · ϑ · 𝒦) ∉ dbupd_sst (unlabel ℬ) 𝒦 {}"
  using inset_model_swap[OF h(1,2) ty _ q1 q2 q3 q4] t_s_fv by simp
```

```
            show ?thesis
            proof (cases "T_upds = []")
              case True
              have "dbupd_sst (unlabel T_val_constr) I D = D" for I D
                using T_val_constr_no_upds_if_no_T_upds[OF True]
                      dbupd_sst_filter[of "unlabel T_val_constr"]
                by force
              thus ?thesis using ** by simp
            next
              case False thus ?thesis
                using ** * h(3) T_val_constr_no_upds_if_no_T_upds unfolding D2 by blast
            qed
          qed
          ultimately show "?sem ?M2 ?D2 T'"
            unfolding T'_def 4[OF T_adm K3 K2] by blast
        qed
      qed
    qed

    show ?case
      using B'1 B'2 B'3 B'4 B'5 B'6 B'7 B'8_9 B'10_11 B'12 B'13 B'14
      unfolding ϑ_def[symmetric] T'_def[symmetric] by blast
  qed

  obtain ℬ 𝒥 where B:
      "ℬ ∈ reachable_constraints P" "?wt_model 𝒥 ℬ"
      "∀x ∈ fv_lsst 𝒜. ∃n. 𝒥 x = Fun (Val n) []" "?rcv_att 𝒜 n ⟶ ?rcv_att ℬ n" "fv_lsst 𝒜 = fv_lsst
ℬ"
    using lmm[OF finite.emptyI _ _ finite.emptyI] unfolding valconsts_only_def by auto

  show ?thesis
    using B(1,3) welltyped_constraint_model_attack_if_receive_attack[OF B(2)] B(4) 2
    unfolding wt_attack_def B(5) by (meson list.set_intros(1))
qed

private lemma add_occurs_msgs_soundness_aux2:
  assumes P: "∀T ∈ set P. admissible_transaction T"
    and A: "𝒜 ∈ reachable_constraints P"
  shows "∃ℬ ∈ reachable_constraints (map add_occurs_msgs P). 𝒜 = rm_occurs_msgs_constr ℬ"
using A
proof (induction rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  define ϑ where "ϑ ≡ ξ ∘s σ ∘s α"

  let ?A' = "dual_lsst (transaction_strand T ·lsst ϑ)"
  let ?B' = "dual_lsst (transaction_strand (add_occurs_msgs T) ·lsst ϑ)"

  obtain A B C D E F where T: "T = Transaction A B C D E F" by (cases T) simp

  have P': "∀T ∈ set P. admissible_transaction' T"
           "∀T ∈ set P. admissible_transaction_no_occurs_msgs T"
    using P admissible_transactionE'(1,2) by (blast,blast)

  note T_adm' = bspec[OF P step.hyps(2)]
  note T_adm = bspec[OF P'(1) step.hyps(2)]
  note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm step.hyps(3)]
  note T_fresh_val = admissible_transactionE(2)[OF T_adm]

  note T_no_occ = admissible_transactionE'(2)[OF T_adm']

  obtain ℬ where B:
      "ℬ ∈ reachable_constraints (map add_occurs_msgs P)" "𝒜 = rm_occurs_msgs_constr ℬ"
```

```
    using step.IH by blast

  note 0 = add_occurs_msgs_cases[OF T]
  note 1 = add_occurs_msgs_vars_eq[OF bspec[OF P'(1)]]
  note 2 = add_occurs_msgs_trms[of T]
  note 3 = add_occurs_msgs_transaction_strand_set[OF T]

  have 4: "add_occurs_msgs T ∈ set (map add_occurs_msgs P)"
    using step.hyps(2) by simp

  have 5: "transaction_decl_subst ξ (add_occurs_msgs T)"
    using step.hyps(3) 0(4) unfolding transaction_decl_subst_def by argo

  have "t ∉ subterms_set (trms_lsst ℬ)"
       "t ∉ subterms_set (trms_transaction (add_occurs_msgs T))"
    when t: "t ∈ subst_range σ" for t
  proof -
    obtain c where c: "t = Fun (Val c) []"
      using t T_fresh_val transaction_fresh_subst_domain[OF step.hyps(4)]
            transaction_fresh_subst_sends_to_val[OF step.hyps(4), of _ thesis]
      by fastforce

    have *: "t ∉ subterms_set (trms_lsst 𝒜)" "t ∉ subterms_set (trms_transaction T)"
      using t step.hyps(4) unfolding transaction_fresh_subst_def by (fast,fast)

    have "t ⊑_set trms_lsst 𝒜 ∨ (∃x ∈ fv_lsst 𝒜. t ⊑ occurs (Var x)) ∨
          (∃c. Fun c [] ⊑_set trms_lsst 𝒜 ∧ t ⊑ occurs (Fun c []))"
      when t: "t ⊑_set trms_lsst ℬ"
      using t rm_occurs_msgs_constr_reachable_constraints_trms_cases[OF P' B(2,1)] by fast
    thus "t ∉ subterms_set (trms_lsst ℬ)"
      using *(1) unfolding c by fastforce

    show "t ∉ subterms_set (trms_transaction (add_occurs_msgs T))"
      using *(2) unfolding 2 c by force
  qed
  hence 6: "transaction_fresh_subst σ (add_occurs_msgs T) (trms_lsst ℬ)"
    using step.hyps(4) unfolding transaction_fresh_subst_def 0(5) 2 by fast

  have 7: "transaction_renaming_subst α (map add_occurs_msgs P) (vars_lsst ℬ)"
    using step.hyps(5) rm_occurs_msgs_constr_reachable_constraints_vars_eq[OF P' B(1)] B(2) 1(5)
    unfolding transaction_renaming_subst_def by simp

  have "?A' = rm_occurs_msgs_constr ?B'"
    using admissible_transaction_decl_fresh_renaming_subst_not_occurs[OF T_adm step.hyps(3,4,5)]
          rm_occurs_msgs_constr_transaction_strand''[OF T_adm T_no_occ]
    unfolding ϑ_def[symmetric] by metis
  hence 8: "𝒜@?A' = rm_occurs_msgs_constr (ℬ@?B')"
    by (metis rm_occurs_msgs_constr_append B(2))

  show ?case using reachable_constraints.step[OF B(1) 4 5 6 7] 8 unfolding ϑ_def by blast
qed (metis reachable_constraints.init rm_occurs_msgs_constr.simps(1))

private lemma add_occurs_msgs_soundness_aux3:
  assumes P: "∀T ∈ set P. admissible_transaction T"
    and A: "𝒜 ∈ reachable_constraints (map add_occurs_msgs P)"
           "welltyped_constraint_model ℐ (rm_occurs_msgs_constr 𝒜)"
    and I: "∀x ∈ fv_lsst 𝒜. ∃n. ℐ x = Fun (Val n) []" (is "?I 𝒜")
  shows "welltyped_constraint_model ℐ 𝒜" (is "?Q ℐ 𝒜")
using A I
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  let ?f = rm_occurs_msgs_constr
  let ?sem = "λB. strand_sem_stateful (ik_lsst 𝒜 ·_set ℐ) (dbupd_sst (unlabel 𝒜) ℐ {}) (unlabel B) ℐ"
```

```
define ϑ where "ϑ ≡ ξ ∘s σ ∘s α"
define ℬ where "ℬ ≡ dual_lsst (transaction_strand T ·lsst ϑ)"

obtain T' where T': "T' ∈ set P" "T = add_occurs_msgs T'"
  using step.hyps(2) by fastforce
then obtain A' B' C' D' E' F' where T'': "T' = Transaction A' B' C' D' E' F'"
  using prot_transaction.exhaust by blast

have P': "∀T ∈ set (map add_occurs_msgs P). admissible_transaction' T"
         "∀T ∈ set (map add_occurs_msgs P). admissible_transaction_occurs_checks T"
         "∀T ∈ set P. admissible_transaction' T"
         "∀T ∈ set P. admissible_transaction_no_occurs_msgs T"
  using P admissible_transactionE' add_occurs_msgs_admissible_occurs_checks
  by (fastforce,fastforce,fastforce,fastforce)

note T_adm = bspec[OF P'(1) step.hyps(2)]
note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]
note T'_adm = bspec[OF P'(3) T'(1)]
note T'_no_occ = bspec[OF P'(4) T'(1)]
note T'_wf = admissible_transaction_is_wellformed_transaction(1)[OF T'_adm]
note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm step.hyps(3)]
note T_fresh_val = admissible_transactionE(2)[OF T_adm]

have 0: "?Q ℐ (?f 𝒜)" "?I 𝒜" "?I ℬ"
  by (metis step.prems(1) welltyped_constraint_model_prefix rm_occurs_msgs_constr_append,
      simp_all add: step.prems(2) ϑ_def ℬ_def)

note IH = step.IH[OF 0(1,2)]

have I': "wt_subst ℐ" "interpretation_subst ℐ" "wf_trms (subst_range ℐ)"
  using step.prems(1) unfolding welltyped_constraint_model_def constraint_model_def by blast+

have 1: "∀x ∈ fv_transaction T. ∄t. ϑ x = occurs t"
        "∀x ∈ fv_transaction T. ϑ x ≠ Fun OccursSec []"
  using admissible_transaction_decl_fresh_renaming_subst_not_occurs[OF T_adm step.hyps(3,4,5)]
  unfolding ϑ_def[symmetric] by simp_all

have "(ik_lsst (?f 𝒜) ·set ℐ) ∪ (ik_lsst 𝒜 ·set ℐ) = ik_lsst 𝒜 ·set ℐ"
  using rm_occurs_msgs_constr_ik_subset by fast
hence 2: "?sem (?f ℬ)"
  using step.prems(1) strand_sem_append_stateful[of "{}" "{}" "unlabel (?f 𝒜)" "unlabel (?f ℬ)"]
        rm_occurs_msgs_constr_dbupd_sst_eq[of 𝒜 ℐ "{}"] rm_occurs_msgs_constr_append[of 𝒜 ℬ]
        strand_sem_ik_mono_stateful[of "ik_lsst (?f 𝒜) ·set ℐ" _ _ _ "ik_lsst 𝒜 ·set ℐ"]
  unfolding welltyped_constraint_model_def constraint_model_def ϑ_def[symmetric] ℬ_def[symmetric]
  by auto

note 3 = rm_occurs_msgs_constr_transaction_strand''[
          OF T'_adm T'_no_occ 1[unfolded T'(2)], unfolded ℬ_def[symmetric] T'(2)[symmetric]]

note 4 = add_occurs_msgs_cases[OF T'', unfolded T'(2)[symmetric]]


define xs where "xs ≡ fv_list_sst (unlabel (transaction_strand T'))"
define flt where "flt ≡ filter (λx. x ∉ set (transaction_fresh T'))"
define occs where "occs ≡ map (λx. occurs (Var x)::('fun,'atom,'sets,'lbl) prot_term)"

note 6 = add_occurs_msgs_transaction_strand_cases(7,8,9)[
          of T' ϑ, unfolded xs_def[symmetric] flt_def[symmetric] occs_def[symmetric]
                       T'(2)[symmetric]]

have 7: "x ∈ fv_transaction T - set (transaction_fresh T)"
  when x: "x ∈ set (flt xs)" for x
```

```
    using that fv_list_sst_is_fv_sst add_occurs_msgs_vars_eq(3,9)[OF T'_adm]
    unfolding xs_def flt_def T'(2) by force

  have 9: "∃y. ϑ x = Var y"
    when x: "x ∈ set (flt xs)" for x
  proof -
    have *: "x ∉ fst ` set (transaction_decl T ())"
      using admissible_transactionE(1)[OF T_adm] by simp

    have **: "x ∉ set (transaction_fresh T)" using 7[OF x] by simp

    show ?thesis
      using transaction_decl_fresh_renaming_substs_range(4)[OF step.hyps(3,4,5) * **]
      unfolding ϑ_def by blast
  qed

  have 8: "∃y ∈ fv_lsst ℬ. ϑ x = Var y"
    when x: "x ∈ set (flt xs)" for x
  proof -
    note * = 7[OF x]

    obtain y where y: "ϑ x = Var y" using 9[OF x] by blast

    have "x ∈ fv_lsst (dual_lsst (transaction_strand T))" by (metis * Diff_iff fv_sst_unlabel_dual_lsst_eq)
    have "ϑ x ∈ ϑ ` fv_lsst (transaction_strand T)" using * by fast
    hence "fv (ϑ x) ⊆ fv_set (ϑ ` fv_transaction T)" by force
    hence "fv (ϑ x) ⊆ fv_lsst (transaction_strand T ·_lsst ϑ)"
      using fv_sst_subst_if_no_bvars[OF admissible_transactionE(4)[OF T_adm], of ϑ]
      by (metis unlabel_subst)
    hence "fv (ϑ x) ⊆ fv_lsst ℬ" by (metis fv_sst_unlabel_dual_lsst_eq ℬ_def)
    thus ?thesis using y by simp
  qed

  have ℬ_var_is_ℐ_val: "∃n. ℐ x = Fun (Val n) []" when x: "x ∈ fv_lsst ℬ" for x
    using step.prems(2) x unfolding ℬ_def[symmetric] ϑ_def[symmetric] by auto

  have T'_var_is_ϑℐ_val: "∃n. ϑ x · ℐ = Fun (Val n) []" when x: "x ∈ set (flt xs)" for x
    using 8[OF x] ℬ_var_is_ℐ_val by force



  have poschecks_has_occ: "occurs (Fun (Val n) []) ∈ ik_lsst 𝒜"
    when x: "⟨ac: t ∈ s⟩ ∈ set (unlabel ℬ)"
      and n: "t · ℐ = Fun (Val n) []"
    for ac t s n
  proof -
    have *: "(t · ℐ, s · ℐ) ∈ dbupd_sst (unlabel 𝒜) ℐ {}"
    proof -
      obtain t' s' where t':
          "⟨ac: t' ∈ s'⟩ ∈ set (unlabel (transaction_checks T'))" "t = t' · ϑ" "s = s' · ϑ"
        using 4(8) x stateful_strand_step_mem_substD(6)
              wellformed_transaction_strand_unlabel_memberD(10)[OF T_wf(1)]
              dual_lsst_unlabel_steps_iff(6)
        unfolding ℬ_def by (metis (no_types) unlabel_subst dual_lsst_subst)

      have "(t' · ϑ · ℐ, s' · ϑ · ℐ) ∈ dbupd_sst (unlabel 𝒜) ℐ {}"
        using t'(1) 2
              wellformed_transaction_unlabel_sem_iff[
                OF T'_wf(1) I'(2,3), of "ik_lsst 𝒜 ·_set ℐ" "dbupd_sst (unlabel 𝒜) ℐ {}" ϑ]
        unfolding 3 by blast
      thus ?thesis using t'(2,3) by simp
    qed
```

```
  have "t' ∈ trms_lsst 𝒜"
    when "insert⟨t',s'⟩ ∈ set (unlabel 𝒜)" for t' s'
    using that by force

  have "t · ℐ ⊑_set trms_lsst 𝒜"
  proof -
    obtain t' s' where t': "insert⟨t',s'⟩ ∈ set (unlabel 𝒜)" "t · ℐ = t' · ℐ" "s · ℐ = s' · ℐ"
      using * db_sst_in_cases[of "t · ℐ" "s · ℐ" "unlabel 𝒜" ℐ "[]"]
            db_sst_set_is_dbupd_sst[of "unlabel 𝒜" ℐ "[]"]
      by auto

    have t'': "t' = t · ℐ ∨ (∃y ∈ fv_lsst 𝒜. t' = Var y ∧ ℐ y = t · ℐ)"
      using t'(1,2) stateful_strand_step_fv_subset_cases(4)
      unfolding n by (cases t') (force,force)
    thus ?thesis
    proof
      assume "t' = t · ℐ" thus ?thesis using t'(1) by force
    next
      assume "∃y ∈ fv_lsst 𝒜. t' = Var y ∧ ℐ y = t · ℐ"
      then obtain y where y: "y ∈ fv_lsst 𝒜" "ℐ y = t · ℐ" by blast

      have "Γ_v y = TAtom Value"
        using y(2) wt_subst_trm''[OF I'(1), of "Var y"] unfolding n by simp
      hence "∃B. prefix B 𝒜 ∧ t · ℐ ⊑_set trms_lsst B"
        by (metis y constraint_model_Value_var_in_constr_prefix[OF step.hyps(1) IH P'(1,2)])
      thus ?thesis unfolding prefix_def by auto
    qed
  qed
  thus ?thesis
    using reachable_constraints_occurs_fact_ik_case'[OF step.hyps(1) P'(1,2)]
    unfolding n by blast
qed


have snds_has_occ: "occurs (Fun (Val n) []) ∈ ik_lsst 𝒜"
  when ts: "send⟨ts⟩ ∈ set (unlabel ℬ)"
    and n: "Fun (Val n) [] ⊑_set set ts ·_set ℐ"
  for ts n
proof -
  have "receive⟨ts⟩ ∈ set (unlabel (transaction_strand T ·_lsst ϑ))"
    using ts dual_lsst_unlabel_steps_iff(2) unfolding ℬ_def by metis
  then obtain ts' where ts':
      "receive⟨ts'⟩ ∈ set (unlabel (transaction_strand T))" "ts = ts' ·_list ϑ"
    by (metis subst_lsst_memD(1) unlabel_in unlabel_mem_has_label)


  have "?sem (dual_lsst (transaction_receive T' ·_lsst ϑ))"
    using 2 strand_sem_append_stateful[of "ik_lsst 𝒜 ·_set ℐ" "dbupd_sst (unlabel 𝒜) ℐ {}"]
    unfolding 3 transaction_dual_subst_unlabel_unfold by blast
  moreover have "list_all is_Receive (unlabel (transaction_receive T'))"
    using T'_wf unfolding wellformed_transaction_def by blast
  hence "list_all is_Send (unlabel (dual_lsst (transaction_receive T' ·_lsst ϑ)))"
    by (metis subst_lsst_unlabel subst_sst_list_all(2) dual_lsst_list_all(1))
  hence "ik_lsst (dual_lsst (transaction_receive T' ·_lsst ϑ)) = {}"
    using in_ik_sst_iff unfolding list_all_iff is_Send_def by fast
  ultimately have *: "ik_lsst 𝒜 ·_set ℐ ⊢ t ·  ℐ"
    when "send⟨ts⟩ ∈ set (unlabel (dual_lsst (transaction_receive T' ·_lsst ϑ)))" "t ∈ set ts"
    for t ts
    using strand_sem_stateful_sends_deduct[OF _ that] by simp
  hence *: "ik_lsst 𝒜 ·_set ℐ ⊢ t · ϑ · ℐ"
    when ts: "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T'))" "t ∈ set ts" for t ts
    using ts(2) dual_lsst_unlabel_steps_iff(2)[of "ts ·_list ϑ" "transaction_receive T' ·_lsst ϑ"]
          stateful_strand_step_subst_inI(2)[OF ts(1), of ϑ, unfolded unlabel_subst]
```

```
      by auto

    have **: "set (flt xs) = fv_transaction T' - set (transaction_fresh T')"
      using fv_list_sst_is_fv_sst unfolding flt_def xs_def by fastforce

    have rcv_case: ?thesis
      when "ts = ts' ·_list ϑ" "Fun (Val n) [] ⊑_set set ts ·_set ℐ"
           "receive⟨ts'⟩ ∈ set (unlabel (transaction_receive T'))"
      for ts ts'
      using that * reachable_constraints_occurs_fact_ik_case''[OF step.hyps(1) IH P'(1,2)] by auto

    have "receive⟨ts'⟩ ∈ set (unlabel (transaction_receive T))"
      using wellformed_transaction_strand_unlabel_memberD(1)[OF T_wf] ts'(1) by blast
    hence "(ts' = map (λx. occurs (Var x)) (flt xs) ∧ ts' ≠ []) ∨
           receive⟨ts'⟩ ∈ set (unlabel (transaction_receive T'))"
      (is "?A ∨ ?B")
      using ** ts'(1) add_occurs_msgs_cases(13)[OF T'']
      unfolding T'(2)[symmetric] xs_def[symmetric] flt_def[symmetric] by force
    thus ?thesis
    proof
      assume ?A
      then obtain x where x: "x ∈ set (flt xs)" "Fun (Val n) [] ⊑ ϑ x · ℐ"
        using ts' n by fastforce

      have x': "ϑ x · ℐ = Fun (Val n) []" "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
               "x ∈ fv_transaction T'" "x ∉ set (transaction_fresh T')"
        using x(2) T'_var_is_ϑℐ_val[OF x(1)] 7[OF x(1)] ** x(1) by fastforce+

      let ?snds = "unlabel (dual_lsst (transaction_receive T' ·_lsst ϑ))"
      let ?chks = "unlabel (dual_lsst (transaction_checks T' ·_lsst ϑ))"

      have B_subsets: "set ?chks ⊆ set (unlabel ℬ)"
        unfolding ℬ_def transaction_dual_subst_unlabel_unfold 4(8) by fastforce

      from admissible_transaction_fv_in_receives_or_selects_dual_subst[OF T'_adm x'(4,5), of ϑ]
      show ?thesis
      proof
        assume "∃ts. send⟨ts⟩ ∈ set ?snds ∧ ϑ x ⊑_set set ts"
        then obtain ss where ss: "send⟨ss⟩ ∈ set ?snds" "ϑ x ⊑_set set ss" by blast

        obtain ss' where ss':
            "ss = ss' ·_list ϑ" "receive⟨ss'⟩ ∈ set (unlabel (transaction_receive T'))"
          by (metis ss(1) dual_lsst_unlabel_steps_iff(2) subst_lsst_memD(1)
                    unlabel_in unlabel_mem_has_label)

        show ?thesis
          using rcv_case[OF ss'(1) _ ss'(2)] subst_subterms[OF ss(2), of ℐ] x'(1) by argo
      qed (use B_subsets poschecks_has_occ[OF _ x'(1)] in blast)
    qed (metis rcv_case[OF ts'(2) n])
  qed


  have "occurs (ϑ x · ℐ) ∈ ik_lsst 𝒜" when x: "x ∈ set (flt xs)" for x
  proof -
    have "(∃ac s. ⟨ac: ϑ x ∈ s⟩ ∈ set (unlabel ℬ)) ∨
          (∃ts. send⟨ts⟩ ∈ set (unlabel ℬ) ∧ ϑ x ⊑_set set ts)"
      (is "(∃ac s. ?A ac s) ∨ (∃ts. ?B1 ts ∧ ?B2 ts)")
      using 7[OF x] admissible_transaction_fv_in_receives_or_selects_dual_subst[OF T_adm, of x ϑ]
      unfolding ℬ_def transaction_dual_subst_unlabel_unfold by auto
    thus ?thesis
    proof
      assume "∃ac s. ?A ac s"
      then obtain ac s where s: "?A ac s" by blast
```

```
    show ?thesis using poschecks_has_occ[OF s] T'_var_is_ϑℐ_val[OF x] by force
  next
    assume "∃ts. ?B1 ts ∧ ?B2 ts"
    then obtain ts where ts: "?B1 ts" "?B2 ts" by meson
    have ts': "ϑ x · ℐ ⊑set set ts ·set ℐ" by (metis ts(2) subst_subterms)
    show ?thesis using snds_has_occ[OF ts(1)] ts' T'_var_is_ϑℐ_val[OF x] by force
  qed
qed
hence "occurs (ϑ x · ℐ) · ℐ ∈ iklsst 𝒜 ·set ℐ" when "x ∈ set (flt xs)" for x using that by fast
moreover have "occurs (ϑ x · ℐ) · ℐ = occurs (ϑ x · ℐ)" for x
  using subst_ground_ident[OF interpretation_grounds[OF I'(2), of "ϑ x"], of ℐ] by simp
ultimately have "occurs (ϑ x · ℐ) ∈ iklsst 𝒜 ·set ℐ" when "x ∈ set (flt xs)" for x
  using that by auto
hence "iklsst 𝒜 ·set ℐ ⊢ t · ℐ" when "t ∈ set (occs (flt xs) ·list ϑ)" for t
  using that unfolding occs_def by auto
hence occs_sem: "?sem [⟨⋆, send⟨occs (flt xs) ·list ϑ⟩⟩]"
  by auto



have "?sem ℬ"
proof -
  let ?IK = "iklsst 𝒜 ·set ℐ"
  let ?DB = "dbupdsst (unlabel 𝒜) ℐ {}"
  let ?snds = "duallsst (transaction_receive T' ·lsst ϑ)"
  let ?snds_occs = "(⟨⋆, send⟨occs (flt xs) ·list ϑ⟩⟩)#?snds"
  let ?chks = "duallsst (transaction_checks T' ·lsst ϑ)"
  let ?upds = "duallsst (transaction_updates T' ·lsst ϑ)"
  let ?rcvs = "duallsst (transaction_send T' ·lsst ϑ)"

  note * = strand_sem_append_stateful[of _ _ _ _ ℐ]
  note ** = transaction_dual_subst_unlabel_unfold
  have ***: "⋀M. M ∪ (iksst [] ·set ℐ) = M"
            "⋀D. dbupdsst [] ℐ D = D"
    by simp_all

  have snds_sem:
      "?sem ?snds"
      "?sem ?snds_occs"
    using 2 occs_sem *[of ?IK ?DB]
    unfolding 3 ** by (blast, fastforce)

  have "list_all is_Receive (unlabel (transaction_receive T'))"
    using T'_wf unfolding wellformed_transaction_def by blast
  hence "list_all is_Send (unlabel ?snds)" "list_all is_Send (unlabel ?snds_occs)"
    using subst_sst_list_all(2) unlabel_subst duallsst_list_all(1)
    by (metis, metis (no_types) list.pred_inject(2) stateful_strand_step.disc(1) unlabel_Cons(1))
  hence "∀a ∈ set (unlabel ?snds). ¬is_Receive a ∧ ¬is_Insert a ∧ ¬is_Delete a"
        "∀a ∈ set (unlabel ?snds_occs). ¬is_Receive a ∧ ¬is_Insert a ∧ ¬is_Delete a"
    unfolding list_all_iff by (blast,blast)
  hence snds_no_upds:
      "iklsst ?snds ·set ℐ = {}"
      "dbupdsst (unlabel ?snds) ℐ ?DB = ?DB"
      "iklsst (?snds_occs) ·set ℐ = {}"
      "dbupdsst (unlabel ?snds_occs) ℐ ?DB = ?DB"
    by (metis iksst_snoc_no_receive_empty, metis dbupdsst_no_upd,
        metis iksst_snoc_no_receive_empty, metis dbupdsst_no_upd)

  have chks_sem:
      "?sem ?chks"
    using 2 snds_no_upds *
    unfolding 3 ** by auto
```

```
have "list_all is_Check_or_Assignment (unlabel (transaction_checks T'))"
  using T'_wf unfolding wellformed_transaction_def by blast
hence "list_all is_Check_or_Assignment (unlabel ?chks)"
  by (metis (no_types) subst_sst_list_all(11) unlabel_subst dual_lsst_list_all(11))
hence "∀a ∈ set (unlabel ?chks). ¬is_Receive a ∧ ¬is_Insert a ∧ ¬is_Delete a"
  unfolding list_all_iff by blast
hence chks_no_upds:
    "ik_lsst ?chks ·_set ℐ = {}"
    "dbupd_sst (unlabel ?chks) ℐ ?DB = ?DB"
  by (metis ik_sst_snoc_no_receive_empty, metis dbupd_sst_no_upd)

have upds_sem:
    "?sem ?upds"
  using 2 snds_no_upds chks_no_upds *
  unfolding 3 ** by auto

have "list_all is_Send (unlabel (transaction_send T'))"
  using T'_wf unfolding wellformed_transaction_def by fast
hence "list_all is_Send (unlabel (transaction_send T' ·_lsst ϑ))"
  by (metis (no_types, opaque_lifting) subst_sst_list_all(1) unlabel_subst)
hence rcvs_is_rcvs: "list_all is_Receive (unlabel ?rcvs)"
  using dual_lsst_list_all(2) by blast

have rcvs_sem: "strand_sem_stateful M D (unlabel rcvs) ℐ"
  when "list_all is_Receive (unlabel rcvs)"
  for M D and rcvs::"('fun, 'atom, 'sets, 'lbl) prot_strand"
  using rcvs_is_rcvs strand_sem_receive_prepend_stateful[of M D "[]" ℐ, OF _ that] by auto

have B_sem: "?sem (?snds@?chks@?upds@rcvs)"
            "?sem (?snds_occs@?chks@?upds@rcvs)"
  when "list_all is_Receive (unlabel rcvs)" for rcvs
  using strand_sem_append_stateful[of _ _ _ _ ℐ]
        snds_sem snds_no_upds chks_sem chks_no_upds
        upds_sem rcvs_sem[OF that]
  by (force, force)

show ?thesis
proof (cases "transaction_fresh T' = []")
  case True
  show ?thesis using B_sem[OF rcvs_is_rcvs] unfolding ℬ_def 6(1)[OF True] by force
next
  case False
  note F = this
  show ?thesis
  proof (cases "∃ts F'. transaction_send T' = ⟨⋆, send⟨ts⟩⟩#F'")
    case True
    obtain ts F' rcvs' where F':
        "transaction_send T' = ⟨⋆, send⟨ts⟩⟩#F'"
        "ℬ = (if flt xs = [] then ?snds else ?snds_occs)@?chks@?upds@rcvs'"
        "rcvs' = ⟨⋆, receive⟨occs (transaction_fresh T')@ts ·_list ϑ⟩⟩#dual_lsst (F' ·_lsst ϑ)"
      using 6(3)[OF F True] unfolding ℬ_def by blast

    have *: "list_all is_Receive (unlabel rcvs')"
      using rcvs_is_rcvs dual_lsst_Cons(1)[of ⋆ ts "F' ·_lsst ϑ"]
            subst_lsst_cons[of "⟨⋆, send⟨ts⟩⟩" F' ϑ]
      unfolding F'(1,3) list_all_iff by auto

    show ?thesis using B_sem[OF *] unfolding F'(2) by fastforce
  next
    case False
    have *:
        "list_all is_Receive (unlabel (⟨⋆, receive⟨occs (transaction_fresh T') ·_list ϑ⟩⟩#?rcvs))"
      using rcvs_is_rcvs by auto
```

```
        show ?thesis using B_sem[OF *] unfolding ℬ_def 6(2)[OF F False] by fastforce
      qed
    qed
  qed
  thus ?case
    using IH strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "unlabel ℬ" ℐ]
    unfolding welltyped_constraint_model_def constraint_model_def ϑ_def[symmetric] ℬ_def[symmetric]
    by simp
qed simp

theorem add_occurs_msgs_soundness:
  defines "wt_attack ≡ λℐ 𝒜 l n. welltyped_constraint_model ℐ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
  assumes P: "∀T ∈ set P. admissible_transaction T"
             "has_initial_value_producing_transaction P"
    and A: "𝒜 ∈ reachable_constraints P" "wt_attack ℐ 𝒜 l n"
  shows "∃ℬ ∈ reachable_constraints (map add_occurs_msgs P). ∃𝒥. wt_attack 𝒥 ℬ l n"
proof -
  have P': "∀T ∈ set (map add_occurs_msgs P). admissible_transaction' T"
           "∀T ∈ set (map add_occurs_msgs P). admissible_transaction_occurs_checks T"
           "∀T ∈ set P. admissible_transaction' T"
           "∀T ∈ set P. admissible_transaction_no_occurs_msgs T"
    using P admissible_transactionE' add_occurs_msgs_admissible_occurs_checks
    by (fastforce,fastforce,fastforce,fastforce)

  obtain 𝒜' 𝒥 where A':
      "𝒜' ∈ reachable_constraints P" "wt_attack 𝒥 𝒜' l n" "∀x∈fv_lsst 𝒜'. ∃n. 𝒥 x = Fun (Val n) []"
    using add_occurs_msgs_soundness_aux1[OF P'(3) P(2) A[unfolded wt_attack_def]]
    unfolding wt_attack_def by blast

  have J: "welltyped_constraint_model 𝒥 𝒜'"
    using A'(2) welltyped_constraint_model_prefix
    unfolding wt_attack_def by blast

  obtain ℬ where B:
      "ℬ ∈ reachable_constraints (map add_occurs_msgs P)" "𝒜' = rm_occurs_msgs_constr ℬ"
    using add_occurs_msgs_soundness_aux2[OF P(1) A'(1)] by blast

  have J': "welltyped_constraint_model 𝒥 ℬ"
    using add_occurs_msgs_soundness_aux3[OF P(1) B(1) J[unfolded B(2)]]
          A'(3) rm_occurs_msgs_constr_reachable_constraints_fv_eq[OF P'(3,4) B(1)]
    unfolding wt_attack_def B(2) by blast

  obtain ts where ts: "receive⟨ts⟩ ∈ set (unlabel ℬ)" "attack⟨n⟩ ∈ set ts"
    using reachable_constraints_receive_attack_if_attack''[OF P'(3) A'(1,2)[unfolded wt_attack_def]]
          rm_occurs_msgs_constr_receive_attack_iff[of n ℬ]
    unfolding B(2)[symmetric] by auto

  have J'': "wt_attack 𝒥 ℬ l n"
    using welltyped_constraint_model_attack_if_receive_attack[OF J' ts]
    unfolding wt_attack_def by fast

  show ?thesis
    using B(1) J'' by blast
qed

end

end
```

3.6.4 Automatically Checking Protocol Security in a Typed Model

```
context stateful_protocol_model
```

```
begin

definition abs_intruder_knowledge (‹α_ik›) where
  "α_ik S ℐ ≡ (ik_lsst S ·_set ℐ) ·_αset α₀ (db_lsst S ℐ)"

definition abs_value_constants (‹α_vals›) where
  "α_vals S ℐ ≡ {t ∈ subterms_set (trms_lsst S) ·_set ℐ. ∃n. t = Fun (Val n) []} ·_αset α₀ (db_lsst S ℐ)"

definition abs_term_implications (‹α_ti›) where
  "α_ti 𝒜 T ϑ ℐ ≡ {(s,t) | s t x.
    s ≠ t ∧ x ∈ fv_transaction T ∧ x ∉ set (transaction_fresh T) ∧
    Fun (Abs s) [] = ϑ x · ℐ ·_α α₀ (db_lsst 𝒜 ℐ) ∧
    Fun (Abs t) [] = ϑ x · ℐ ·_α α₀ (db_lsst (𝒜@dual_lsst (transaction_strand T ·_lsst ϑ)) ℐ)}"

lemma abs_intruder_knowledge_append:
  "α_ik (A@B) ℐ =
    (ik_lsst A ·_set ℐ) ·_αset α₀ (db_lsst (A@B) ℐ) ∪
    (ik_lsst B ·_set ℐ) ·_αset α₀ (db_lsst (A@B) ℐ)"
by (metis unlabel_append abs_set_union image_Un ik_sst_append abs_intruder_knowledge_def)

lemma abs_value_constants_append:
  fixes A B::"('a,'b,'c,'d) prot_strand"
  shows "α_vals (A@B) ℐ =
      {t ∈ subterms_set (trms_lsst A) ·_set ℐ. ∃n. t = Fun (Val n) []} ·_αset α₀ (db_lsst (A@B) ℐ) ∪
      {t ∈ subterms_set (trms_lsst B) ·_set ℐ. ∃n. t = Fun (Val n) []} ·_αset α₀ (db_lsst (A@B) ℐ)"
proof -
  define a0 where "a0 ≡ α₀ (db_sst (unlabel (A@B)) ℐ)"
  define M where "M ≡ λa::('a,'b,'c,'d) prot_strand.
                          {t ∈ subterms_set (trms_lsst a) ·_set ℐ. ∃n. t = Fun (Val n) []}"

  have "M (A@B) = M A ∪ M B"
    using image_Un[of "λx. x · ℐ" "subterms_set (trms_lsst A)" "subterms_set (trms_lsst B)"]
    unfolding M_def unlabel_append[of A B] trms_sst_append[of "unlabel A" "unlabel B"] by blast
  hence "M (A@B) ·_αset a0 = (M A ·_αset a0) ∪ (M B ·_αset a0)" by (simp add: abs_set_union)
  thus ?thesis unfolding abs_value_constants_def a0_def M_def by force
qed

lemma transaction_renaming_subst_has_no_pubconsts_abss:
  fixes α::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "subst_range α ∩ pubval_terms = {}" (is ?A)
    and "subst_range α ∩ abs_terms = {}" (is ?B)
proof -
  { fix t assume "t ∈ subst_range α"
    then obtain x where "t = Var x"
      using transaction_renaming_subst_is_renaming(1)[OF assms]
      by force
    hence "t ∉ pubval_terms" "t ∉ abs_terms" by simp_all
  } thus ?A ?B by auto
qed

lemma transaction_fresh_subst_has_no_pubconsts_abss:
  fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜" "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
  shows "subst_range σ ∩ pubval_terms = {}" (is ?A)
    and "subst_range σ ∩ abs_terms = {}" (is ?B)
proof -
  { fix t assume "t ∈ subst_range σ"
    then obtain x where "x ∈ set (transaction_fresh T)" "σ x = t"
      using assms(1) unfolding transaction_fresh_subst_def by auto
    then obtain n where "t = Fun (Val n) []"
      using transaction_fresh_subst_sends_to_val[OF assms(1)] assms(2)
      by meson
```

```
    hence "t ∉ pubval_terms" "t ∉ abs_terms" unfolding is_PubConstValue_def by simp_all
  } thus ?A ?B by auto
qed

lemma reachable_constraints_GSMP_no_pubvals_abss:
  assumes "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and ℐ: "interpretation_subst ℐ" "wt_subst ℐ" "wf_trms (subst_range ℐ)"
           "∀n. PubConst Value n ∉ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
           "∀n. Abs n ∉ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
  shows "trms_lsst 𝒜 ·_set ℐ ⊆ GSMP (⋃T ∈ set P. trms_transaction T) - (pubval_terms ∪ abs_terms)"
    (is "?A ⊆ ?B")
using assms(1) ℐ(4,5)
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T ξ σ α)
  define trms_P where "trms_P ≡ (⋃T ∈ set P. trms_transaction T)"
  define T' where "T' ≡ transaction_strand T ·_lsst ξ ∘_s σ ∘_s α"

  have ξ_elim: "ξ ∘_s σ ∘_s α = σ ∘_s α"
    using admissible_transaction_decl_subst_empty[OF bspec[OF P step.hyps(2)] step.hyps(3)]
    by simp

  note T_fresh = admissible_transactionE(2)[OF bspec[OF P step.hyps(2)]]
  note T_no_bvars = admissible_transactionE(4)[OF bspec[OF P step.hyps(2)]]

  have T_no_PubVal: "∀T ∈ set P. ∀n. PubConst Value n ∉ ⋃(funs_term ` trms_transaction T)"
    and T_no_Abs: "∀T ∈ set P. ∀n. Abs n ∉ ⋃(funs_term ` trms_transaction T)"
    using admissible_transactions_no_Value_consts''[OF bspec[OF P]] by metis+

  have ℐ': "∀n. PubConst Value n ∉ ⋃ (funs_term ` (ℐ ` fv_lsst 𝒜))"
           "∀n. Abs n ∉ ⋃ (funs_term ` (ℐ ` fv_lsst 𝒜))"
    using step.prems fv_sst_append[of "unlabel 𝒜"] unlabel_append[of 𝒜]
    by auto

  have "wt_subst (rm_vars (set X) (ξ ∘_s σ ∘_s α))" for X
    using wt_subst_rm_vars[of "ξ ∘_s σ ∘_s α" "set X"]
          transaction_decl_fresh_renaming_substs_wt[OF step.hyps(3-5)]
    by metis
  hence wt: "wt_subst ((rm_vars (set X) (ξ ∘_s σ ∘_s α)) ∘_s ℐ)" for X
    using ℐ(2) wt_subst_compose by fast

  have wftrms: "wf_trms (subst_range ((rm_vars (set X) (ξ ∘_s σ ∘_s α)) ∘_s ℐ))" for X
    using wf_trms_subst_compose[OF wf_trms_subst_rm_vars' ℐ(3)]
          transaction_decl_fresh_renaming_substs_range_wf_trms[OF step.hyps(3-5)]
    by fast

  have "trms_lsst (dual_lsst T') ·_set ℐ ⊆ ?B"
  proof
    fix t assume "t ∈ trms_lsst (dual_lsst T') ·_set ℐ"
    hence "t ∈ trms_lsst T' ·_set ℐ" using trms_sst_unlabel_dual_lsst_eq by blast
    then obtain s X where s:
        "s ∈ trms_transaction T"
        "t = s · rm_vars (set X) (ξ ∘_s σ ∘_s α) ∘_s ℐ"
        "set X ⊆ bvars_transaction T"
      using trms_sst_unlabel_subst'' unfolding T'_def by blast

    define ϑ where "ϑ ≡ rm_vars (set X) (ξ ∘_s σ ∘_s α)"

    note 0 = pubval_terms_subst_range_comp[OF
              transaction_fresh_subst_has_no_pubconsts_abss(1)[OF step.hyps(4) T_fresh]
              transaction_renaming_subst_has_no_pubconsts_abss(1)[OF step.hyps(5)]]
             abs_terms_subst_range_comp[OF
              transaction_fresh_subst_has_no_pubconsts_abss(2)[OF step.hyps(4) T_fresh]
```

```
          transaction_renaming_subst_has_no_pubconsts_abss(2)[OF step.hyps(5)]]

  have 1: "s ∈ trms_P" using step.hyps(2) s(1) unfolding trms_P_def by auto

  have s_nin: "s ∉ pubval_terms" "s ∉ abs_terms"
    using 1 T_no_PubVal T_no_Abs funs_term_Fun_subterm
    unfolding trms_P_def is_PubConstValue_def is_Abs_def is_PubConst_def
    by (fastforce, blast)

  have 2: "(ℐ ` fv_lsst (𝒜@dual_lsst T')) ∩ pubval_terms = {}"
          "(ℐ ` fv_lsst (𝒜@dual_lsst T')) ∩ abs_terms = {}"
          "subst_range (ξ ∘s σ ∘s α) ∩ pubval_terms = {}"
          "subst_range (ξ ∘s σ ∘s α) ∩ abs_terms = {}"
    using 0 step.prems funs_term_Fun_subterm
    unfolding T'_def ϑ_def ξ_elim
    by (fastforce simp add: is_PubConstValue_def is_PubConst_def,
        fastforce simp add: is_Abs_def,
        argo, argo)

  have "subst_range ϑ ⊆ subst_range (ξ ∘s σ ∘s α)"
    using rm_vars_img_subset unfolding ϑ_def ξ_elim by blast
  hence 3: "subst_range ϑ ∩ pubval_terms = {}"
           "subst_range ϑ ∩ abs_terms = {}"
    using 2(3,4) step.prems funs_term_Fun_subterm
    unfolding T'_def ϑ_def ξ_elim by (blast,blast)

  have "(ℐ ` fv (s · ϑ)) ∩ pubval_terms = {}"
       "(ℐ ` fv (s · ϑ)) ∩ abs_terms = {}"
  proof -
    have "ϑ = ξ ∘s σ ∘s α" "bvars_transaction T = {}" "vars_lsst T' = fv_lsst T'"
      using s(3) T_no_bvars step.hyps(2) rm_vars_empty
            vars_sst_is_fv_sst_bvars_sst[of "unlabel T'"]
            bvars_sst_subst[of "unlabel (transaction_strand T)" "ξ ∘s σ ∘s α"]
            unlabel_subst[of "transaction_strand T" "ξ ∘s σ ∘s α"]
      unfolding ϑ_def T'_def by simp_all
    hence "fv (s · ϑ) ⊆ fv_lsst T'"
      using trms_sst_fv_subst_subset[OF s(1), of ϑ] unlabel_subst[of "transaction_strand T" ϑ]
      unfolding T'_def by auto
    moreover have "fv_lsst T' ⊆ fv_lsst (𝒜@dual_lsst T')"
      using fv_sst_append[of "unlabel 𝒜" "unlabel (dual_lsst T')"]
            unlabel_append[of 𝒜 "dual_lsst T'"]
            fv_sst_unlabel_dual_lsst_eq[of T']
      by simp_all
    hence "ℐ ` fv_lsst T' ∩ pubval_terms = {}" "ℐ ` fv_lsst T' ∩ abs_terms = {}"
      using 2(1,2) by blast+
    ultimately show "(ℐ ` fv (s · ϑ)) ∩ pubval_terms = {}" "(ℐ ` fv (s · ϑ)) ∩ abs_terms = {}"
      by blast+
  qed
  hence σαℐ_disj: "((ϑ ∘s ℐ) ` fv s) ∩ pubval_terms = {}"
                  "((ϑ ∘s ℐ) ` fv s) ∩ abs_terms = {}"
    using pubval_terms_subst_range_comp'[of ϑ "fv s" ℐ]
          abs_terms_subst_range_comp'[of ϑ "fv s" ℐ]
          pubval_terms_subst_range_disj[OF 3(1), of s]
          abs_terms_subst_range_disj[OF 3(2), of s]
    by (simp_all add: subst_apply_fv_unfold)

  have 4: "t ∉ pubval_terms" "t ∉ abs_terms"
    using s(2) s_nin σαℐ_disj
          pubval_terms_subst[of s "rm_vars (set X) (ξ ∘s σ ∘s α) ∘s ℐ"]
          pubval_terms_subst_range_disj[of "rm_vars (set X) (ξ ∘s σ ∘s α) ∘s ℐ" s]
          abs_terms_subst[of s "rm_vars (set X) (ξ ∘s σ ∘s α) ∘s ℐ"]
          abs_terms_subst_range_disj[of "rm_vars (set X) (ξ ∘s σ ∘s α) ∘s ℐ" s]
    unfolding ϑ_def
```

```
    by blast+

  have "t ∈ SMP trms_P" "fv t = {}"
    by (metis s(2) SMP.Substitution[OF SMP.MP[OF 1] wt wftrms, of X],
        metis s(2) subst_subst_compose[of s "rm_vars (set X) (ξ ∘s σ ∘s α)" ℐ]
                   interpretation_grounds[OF ℐ(1), of "s · rm_vars (set X) (ξ ∘s σ ∘s α)"])
  hence 5: "t ∈ GSMP trms_P" unfolding GSMP_def by simp

  show "t ∈ ?B" using 4 5 by (auto simp add: trms_P_def)
 qed
 thus ?case
  using step.IH[OF ℐ'] trms_sst_append[of "unlabel 𝒜"] unlabel_append[of 𝒜]
        image_Un[of "λx. x · ℐ" "trms_lsst 𝒜"]
  by (simp add: T'_def)
qed simp

lemma α_ti_covers_α0_aux:
 assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
  and T: "T ∈ set P"
  and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘s σ ∘s α))"
  and ξ: "transaction_decl_subst ξ T"
  and σ: "transaction_fresh_subst σ T (trms_lsst 𝒜)"
  and α: "transaction_renaming_subst α P (vars_lsst 𝒜)"
  and P: "∀T ∈ set P. admissible_transaction' T"
  and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
  and t: "t ∈ subterms_set (trms_lsst 𝒜)"
         "t = Fun (Val n) [] ∨ t = Var x"
  and neq:
   "t · ℐ ·α α0 (db_lsst 𝒜 ℐ) ≠
    t · ℐ ·α α0 (db_lsst (𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘s σ ∘s α)) ℐ)"
 shows "∃y ∈ fv_transaction T - set (transaction_fresh T).
        t · ℐ = (ξ ∘s σ ∘s α) y · ℐ ∧ Γv y = TAtom Value"
proof -
 let ?𝒜' = "𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘s σ ∘s α)"
 let ?ℬ = "unlabel (dual_lsst (transaction_strand T))"
 let ?ℬ' = "?ℬ ·_sst ξ ∘s σ ∘s α"
 let ?ℬ'' = "unlabel (dual_lsst (transaction_strand T ·_lsst ξ ∘s σ ∘s α))"

 have ℐ_interp: "interpretation_subst ℐ"
  and ℐ_wt: "wt_subst ℐ"
  and ℐ_wf: "wf_trms (subst_range ℐ)"
  by (metis ℐ welltyped_constraint_model_def constraint_model_def,
      metis ℐ welltyped_constraint_model_def,
      metis ℐ welltyped_constraint_model_def constraint_model_def)

 note T_adm = bspec[OF P(1) T]
 note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]
 note T_adm_upds = admissible_transaction_is_wellformed_transaction(3)[OF T_adm]

 have T_fresh_vars_value_typed: "∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
  using T P(1) protocol_transaction_vars_TAtom_typed(3)[of T] P(1) by simp

 note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm ξ]

 note ξσα_wt = transaction_decl_fresh_renaming_substs_wt[OF ξ σ α]

 have 𝒜_wf_trms: "wf_trms (trms_lsst 𝒜)"
  by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P(1) 𝒜_reach)
 hence t_wf: "wf_trm t" using t by auto

 have 𝒜_no_val_bvars: "¬TAtom Value ⊑ Γv x"
  when "x ∈ bvars_lsst 𝒜" for x
  using P(1) reachable_constraints_no_bvars 𝒜_reach
```

```
        varssst_is_fvsst_bvarssst[of "unlabel 𝒜"] that
        admissible_transactionE(4)
    by fast

  have x': "x ∈ varslsst 𝒜" when "t = Var x"
    using that t by (simp add: var_subterm_trmssst_is_varssst)

  have "∃f ∈ funs_term (t · ℐ). is_Val f"
    using abs_eq_if_no_Val neq by metis
  hence "∃n T. Fun (Val n) T ⊑ t · ℐ"
    using funs_term_Fun_subterm
    unfolding is_Val_def by fast
  hence "TAtom Value ⊑ Γ (Var x)" when "t = Var x"
    using wt_subst_trm''[OF ℐ_wt, of "Var x"] that
        subtermeq_imp_subtermtypeeq[of "t · ℐ"] wf_trm_subst[OF ℐ_wf, of t] t_wf
    by fastforce
  hence x_val: "Γv x = TAtom Value" when "t = Var x"
    using reachable_constraints_vars_TAtom_typed[OF 𝒜_reach P x'] that
    by fastforce
  hence x_fv: "x ∈ fvlsst 𝒜" when "t = Var x" using x'
    using reachable_constraints_Value_vars_are_fv[OF 𝒜_reach P x'] that
    by blast
  then obtain m where m: "t · ℐ = Fun (Val m) []"
    using constraint_model_Value_term_is_Val[
          OF 𝒜_reach welltyped_constraint_model_prefix[OF ℐ] P P_occ, of x]
        t(2) x_val
    by force
  hence 0: "α0 (dblsst 𝒜 ℐ) m ≠ α0 (dbsst (unlabel 𝒜@?ℬ'') ℐ) m"
    using neq by (simp add: unlabel_def)

  have t_val: "Γ t = TAtom Value" using x_val t by force

  obtain u s where s: "t · ℐ = u · ℐ" "insert⟨u,s⟩ ∈ set ?ℬ' ∨ delete⟨u,s⟩ ∈ set ?ℬ'"
    using to_abs_neq_imp_db_update[OF 0] m
    by (metis (no_types, lifting) duallsst_subst subst_lsst_unlabel)
  then obtain u' s' where s':
      "u = u' · ξ ∘s σ ∘s α" "s = s' · ξ ∘s σ ∘s α"
      "insert⟨u',s'⟩ ∈ set ?ℬ ∨ delete⟨u',s'⟩ ∈ set ?ℬ"
    using stateful_strand_step_mem_substD(4,5)
    by blast
  hence s'': "insert⟨u',s'⟩ ∈ set (unlabel (transaction_strand T)) ∨
              delete⟨u',s'⟩ ∈ set (unlabel (transaction_strand T))"
    using duallsst_unlabel_steps_iff(4,5)[of u' s' "transaction_strand T"]
    by simp_all
  then obtain y where y: "y ∈ fv_transaction T" "u' = Var y"
    using transaction_inserts_are_Value_vars[OF T_wf T_adm_upds, of u' s']
        transaction_deletes_are_Value_vars[OF T_wf T_adm_upds, of u' s']
        stateful_strand_step_fv_subset_cases(4,5)[of u' s' "unlabel (transaction_strand T)"]
    by auto
  hence 1: "t · ℐ = (ξ ∘s σ ∘s α) y · ℐ" using y s(1) s'(1) by (metis eval_term.simps(1))

  have 2: "y ∉ set (transaction_fresh T)" when "(ξ ∘s σ ∘s α) y · ℐ ≠ σ y"
    using transaction_fresh_subst_grounds_domain[OF σ, of y] subst_compose[of σ α y] that ξ_empty
    by (auto simp add: subst_ground_ident)

  have 3: "y ∉ set (transaction_fresh T)" when "(ξ ∘s σ ∘s α) y · ℐ ∈ subtermsset (trmslsst 𝒜)"
    using 2 that σ unfolding transaction_fresh_subst_def by fastforce

  have 4: "∀x ∈ fvlsst 𝒜. Γv x = TAtom Value ⟶
            (∃B. prefix B 𝒜 ∧ x ∉ fvlsst B ∧ ℐ x ∈ subtermsset (trmslsst B))"
    by (metis welltyped_constraint_model_prefix[OF ℐ]
              constraint_model_Value_var_in_constr_prefix[OF 𝒜_reach _ P P_occ])
```

```
    have 5: "Γv y = TAtom Value"
      using 1 t_val
            wt_subst_trm''[OF ξσα_wt, of "Var y"]
            wt_subst_trm''[OF I_wt, of t]
            wt_subst_trm''[OF I_wt, of "(ξ ∘s σ ∘s α) y"]
      by (auto simp del: subst_subst_compose)

    have "y ∉ set (transaction_fresh T)"
    proof (cases "t = Var x")
      case True
      hence *: "I x = Fun (Val m) []" "x ∈ fv_lsst A" "I x = (ξ ∘s σ ∘s α) y · I"
        using m t(1) 1 x_fv x' by (force, blast, force)

      obtain B where B: "prefix B A" "I x ∈ subterms_set (trms_lsst B)"
        using *(2) 4 x_val[OF True] by fastforce
      hence "∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst B)"
        using transaction_fresh_subst_range_fresh(1)[OF σ] trms_sst_unlabel_prefix_subset(1)[of B]
        unfolding prefix_def by fast
      thus ?thesis using *(1,3) B(2) 2 by (metis subst_imgI term.distinct(1))
    next
      case False
      hence "t · I ∈ subterms_set (trms_lsst A)" using t by simp
      thus ?thesis using 1 3 by argo
    qed
    thus ?thesis using 1 5 y(1) by fast
qed

lemma α_ti_covers_α0_Var:
  assumes A_reach: "A ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and I: "welltyped_constraint_model I (A@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trms_lsst A)"
    and α: "transaction_renaming_subst α P (vars_lsst A)"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and x: "x ∈ fv_lsst A"
  shows "I x ·α α0 (db_lsst (A@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) I) ∈
           timpl_closure_set {I x ·α α0 (db_lsst A I)} (α_ti A T (ξ ∘s σ ∘s α) I)"
proof -
  define a0 where "a0 ≡ α0 (db_lsst A I)"
  define a0' where "a0' ≡ α0 (db_lsst (A@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) I)"
  define a3 where "a3 ≡ α_ti A T (ξ ∘s σ ∘s α) I"

  have A_wf_trms: "wf_trms (trms_lsst A)"
    by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P(1) A_reach)

  note T_adm = bspec[OF P(1) T]
  note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm ξ]
  note ξσα_wt = transaction_decl_fresh_renaming_substs_wt[OF ξ σ α]

  have I_interp: "interpretation_subst I"
    and I_wt: "wt_subst I"
    and I_wf_trms: "wf_trms (subst_range I)"
    by (metis I welltyped_constraint_model_def constraint_model_def,
        metis I welltyped_constraint_model_def,
        metis I welltyped_constraint_model_def constraint_model_def)

  have "Γv x = Var Value ∨ (∃a. Γv x = Var (prot_atom.Atom a))"
    using reachable_constraints_vars_TAtom_typed[OF A_reach P, of x]
          x vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
    by auto
```

```
hence "ℐ x ·α a0' ∈ timpl_closure_set {ℐ x ·α a0} a3"
proof
  assume x_val: "Γv x = TAtom Value"
  show "ℐ x ·α a0' ∈ timpl_closure_set {ℐ x ·α a0} a3"
  proof (cases "ℐ x ·α a0 = ℐ x ·α a0'")
    case False
    hence "∃y ∈ fv_transaction T - set (transaction_fresh T).
            ℐ x = (ξ ∘s σ ∘s α) y · ℐ ∧ Γv y = TAtom Value"
      using αti_covers_α0_aux[OF 𝒜_reach T ℐ ξ σ α P P_occ fvsst_is_subterm_trmssst[OF x], of _ x]
      unfolding a0_def a0'_def
      by fastforce
    then obtain y where y:
        "y ∈ fv_transaction T - set (transaction_fresh T)"
        "ℐ x = (ξ ∘s σ ∘s α) y · ℐ"
        "ℐ x ·α a0 = (ξ ∘s σ ∘s α) y · ℐ ·α a0"
        "ℐ x ·α a0' = (ξ ∘s σ ∘s α) y · ℐ ·α a0'"
        "Γv y = TAtom Value"
      by metis
    then obtain n where n: "(ξ ∘s σ ∘s α) y · ℐ = Fun (Val n) []"
      using Γv_TAtom''(2)[of y] x x_val
            transaction_var_becomes_Val[
              OF reachable_constraints.step[OF 𝒜_reach T ξ σ α] ℐ ξ σ α P P_occ T, of y]
      by force

    have "a0 n ≠ a0' n"
         "y ∈ fv_transaction T"
         "y ∉ set (transaction_fresh T)"
         "absc (a0 n) = (ξ ∘s σ ∘s α) y · ℐ ·α a0"
         "absc (a0' n) = (ξ ∘s σ ∘s α) y · ℐ ·α a0'"
      using y n False by force+
    hence 1: "(a0 n, a0' n) ∈ a3"
      unfolding a0_def a0'_def a3_def abs_term_implications_def
      by blast

    have 2: "ℐ x ·α a0' ∈ set ⟨a0 n --» a0' n⟩⟨ℐ x ·α a0⟩"
      using y n timpl_apply_const by auto

    show ?thesis
      using timpl_closure.TI[OF timpl_closure.FP 1] 2
            term_variants_pred_iff_in_term_variants[
              of "(λ_. [])(Abs (a0 n) := [Abs (a0' n)])"]
      unfolding timpl_closure_set_def timpl_apply_term_def
      by auto
  qed (auto intro: timpl_closure_setI)
next
  assume "∃a. Γv x = TAtom (Atom a)"
  then obtain a where x_atom: "Γv x = TAtom (Atom a)" by force

  obtain f T where fT: "ℐ x = Fun f T"
    using interpretation_grounds[OF ℐ_interp, of "Var x"]
    by (cases "ℐ x") auto

  have fT_atom: "Γ (Fun f T) = TAtom (Atom a)"
    using wt_subst_trm''[OF ℐ_wt, of "Var x"] x_atom fT
    by simp

  have T: "T = []"
    using fT wf_trm_subst[OF ℐ_wftrms, of "Var x"] const_type_inv_wf[OF fT_atom]
    by fastforce

  have f: "¬is_Val f" using fT_atom unfolding is_Val_def by auto

  have "ℐ x ·α b = ℐ x" for b
```

```
      using T fT abs_term_apply_const(2)[OF f]
      by auto
    thus "ℐ x ·α a0' ∈ timpl_closure_set {ℐ x ·α a0} a3"
      by (auto intro: timpl_closure_setI)
  qed
  thus ?thesis by (metis a0_def a0'_def a3_def)
qed

lemma αti_covers_α0_Val:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trmslsst 𝒜)"
    and α: "transaction_renaming_subst α P (varslsst 𝒜)"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and n: "Fun (Val n) [] ∈ subtermsset (trmslsst 𝒜)"
  shows "Fun (Val n) [] ·α α0 (dblsst (𝒜@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℐ) ∈
          timpl_closure_set {Fun (Val n) [] ·α α0 (dblsst 𝒜 ℐ)} (αti 𝒜 T (ξ ∘s σ ∘s α) ℐ)"
proof -
  define T' where "T' ≡ duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
  define a0 where "a0 ≡ α0 (dblsst 𝒜 ℐ)"
  define a0' where "a0' ≡ α0 (dblsst (𝒜@T') ℐ)"
  define a3 where "a3 ≡ αti 𝒜 T (ξ ∘s σ ∘s α) ℐ"

  have 𝒜_wftrms: "wftrms (trmslsst 𝒜)"
    by (metis reachable_constraints_wftrms admissible_transactions_wftrms P(1) 𝒜_reach)

  note T_adm = bspec[OF P(1) T]

  have "Fun (Abs (a0' n)) [] ∈ timpl_closure_set {Fun (Abs (a0 n)) []} a3"
  proof (cases "a0 n = a0' n")
    case False
    then obtain x where x:
        "x ∈ fv_transaction T - set (transaction_fresh T)" "Fun (Val n) [] = (ξ ∘s σ ∘s α) x · ℐ"
      using αti_covers_α0_aux[OF 𝒜_reach T ℐ ξ σ α P P_occ n]
      by (fastforce simp add: a0_def a0'_def T'_def)
    hence "absc (a0 n) = (ξ ∘s σ ∘s α) x · ℐ ·α a0" "absc (a0' n) = (ξ ∘s σ ∘s α) x · ℐ ·α a0'"
      by simp_all
    hence 1: "(a0 n, a0' n) ∈ a3"
      using False x(1)
      unfolding a0_def a0'_def a3_def abs_term_implications_def T'_def
      by blast
    show ?thesis
      using timpl_apply_Abs[of "[]" "[]" "a0 n" "a0' n"]
            timpl_closure.TI[OF timpl_closure.FP[of "Fun (Abs (a0 n)) []" a3] 1]
            term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs (a0 n) := [Abs (a0' n)])"]
      unfolding timpl_closure_set_def timpl_apply_term_def
      by force
  qed (auto intro: timpl_closure_setI)
  thus ?thesis by (simp add: a0_def a0'_def a3_def T'_def)
qed

lemma αti_covers_α0_ik:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trmslsst 𝒜)"
    and α: "transaction_renaming_subst α P (varslsst 𝒜)"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
```

```
    and t: "t ∈ ik_lsst 𝒜"
  shows "t · ℐ ·_α α₀ (db_lsst (𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) ℐ) ∈
            timpl_closure_set {t · ℐ ·_α α₀ (db_lsst 𝒜 ℐ)} (α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ)"
proof -
  define a0 where "a0 ≡ α₀ (db_lsst 𝒜 ℐ)"
  define a0' where "a0' ≡ α₀ (db_lsst (𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) ℐ)"
  define a3 where "a3 ≡ α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ"

  let ?U = "λT a. map (λs. s · ℐ ·_α a) T"

  have 𝒜_wf_trms: "wf_trms (trms_lsst 𝒜)"
    by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P(1) 𝒜_reach)

  note T_adm = bspec[OF P(1) T]

  have "t ∈ subterms_set (ik_lsst 𝒜)" "wf_trm t" using 𝒜_wf_trms t ik_sst_trms_sst_subset by force+
  hence "∀t0 ∈ subterms t. t0 · ℐ ·_α a0' ∈ timpl_closure_set {t0 · ℐ ·_α a0} a3"
  proof (induction t)
    case (Var x) thus ?case
      using α_ti_covers_α₀_Var[OF 𝒜_reach T ℐ ξ σ α P P_occ, of x]
            ik_sst_var_is_fv[of x "unlabel 𝒜"] vars_sst_is_fv_sst_bvars_sst[of "unlabel 𝒜"]
      by (simp add: a0_def a0'_def a3_def)
  next
    case (Fun f S)
    have IH: "∀t0 ∈ subterms t. t0 · ℐ ·_α a0' ∈ timpl_closure_set {t0 · ℐ ·_α a0} a3"
      when "t ∈ set S" for t
      using that Fun.prems(1) wf_trm_param[OF Fun.prems(2)] Fun.IH
      by (meson in_subterms_subset_Union params_subterms subsetCE)
    hence "t ·_α a0' ∈ timpl_closure_set {t ·_α a0} a3"
      when "t ∈ set (map (λs. s · ℐ) S)" for t
      using that by auto
    hence "t ·_α a0' ∈ timpl_closure (t ·_α a0) a3"
      when "t ∈ set (map (λs. s · ℐ) S)" for t
      using that timpl_closureton_is_timpl_closure by auto
    hence "(t ·_α a0, t ·_α a0') ∈ timpl_closure' a3"
      when "t ∈ set (map (λs. s · ℐ) S)" for t
      using that timpl_closure_is_timpl_closure' by auto
    hence IH': "((?U S a0) ! i, (?U S a0') ! i) ∈ timpl_closure' a3"
      when "i < length (map (λs. s · ℐ ·_α a0) S)" for i
      using that by auto

    show ?case
    proof (cases "∃n. f = Val n")
      case True
      then obtain n where "Fun f S = Fun (Val n) []"
        using Fun.prems(2) unfolding wf_trm_def by force
      moreover have "Fun f S ∈ subterms_set (trms_lsst 𝒜)"
        using ik_sst_trms_sst_subset Fun.prems(1) by blast
      ultimately show ?thesis
        using α_ti_covers_α₀_Val[OF 𝒜_reach T ℐ ξ σ α P P_occ]
        by (simp add: a0_def a0'_def a3_def)
    next
      case False
      hence "Fun f S · ℐ ·_α a = Fun f (map (λt. t · ℐ ·_α a) S)" for a by (cases f) simp_all
      hence "(Fun f S · ℐ ·_α a0, Fun f S · ℐ ·_α a0') ∈ timpl_closure' a3"
        using timpl_closure_FunI[OF IH']
        by simp
      hence "Fun f S · ℐ ·_α a0' ∈ timpl_closure_set {Fun f S · ℐ ·_α a0} a3"
        using timpl_closureton_is_timpl_closure
              timpl_closure_is_timpl_closure'
        by metis
      thus ?thesis using IH by simp
    qed
```

```
  qed
  thus ?thesis by (simp add: a0_def a0'_def a3_def)
qed

lemma transaction_prop1:
  assumes "δ ∈ abs_substs_fun ` set (transaction_check_comp msgcs (FP, OCC, TI) T)"
    and "x ∈ fv_transaction T"
    and "x ∉ set (transaction_fresh T)"
    and "δ x ≠ absdbupd (unlabel (transaction_updates T)) x (δ x)"
    and "transaction_check' msgcs (FP, OCC, TI) T"
    and TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "(δ x, absdbupd (unlabel (transaction_updates T)) x (δ x)) ∈ (set TI)⁺"
proof -
  let ?upd = "λx. absdbupd (unlabel (transaction_updates T)) x (δ x)"

  have 0: "fv_transaction T = set (fv_list_sst (unlabel (transaction_strand T)))"
    by (metis fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"])

  have 1: "transaction_check_post (FP, OCC, TI) T δ"
    using assms(1,5)
    unfolding transaction_check_def transaction_check'_def list_all_iff
    by blast

  have "(δ x, ?upd x) ∈ set TI ⟷ (δ x, ?upd x) ∈ (set TI)⁺"
    using TI using assms(4) by blast
  thus ?thesis
    using assms(2,3,4) 0 1 in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    unfolding transaction_check_post_def List.member_def Let_def by blast
qed

lemma transaction_prop2:
  assumes δ: "δ ∈ abs_substs_fun ` set (transaction_check_comp msgcs (FP, OCC, TI) T)"
    and x: "x ∈ fv_transaction T" "fst x = TAtom Value"
    and T_check: "transaction_check' msgcs (FP, OCC, TI) T"
    and T_adm: "admissible_transaction' T"
    and T_occ: "admissible_transaction_occurs_checks T"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "x ∉ set (transaction_fresh T) ⟹ δ x ∈ set OCC" (is "?A' ⟹ ?A")
    and "absdbupd (unlabel (transaction_updates T)) x (δ x) ∈ set OCC" (is ?B)
proof -
  let ?xs = "fv_list_sst (unlabel (transaction_strand T))"
  let ?ys = "filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) ?xs"
  let ?C = "unlabel (transaction_checks T)"
  let ?poss = "transaction_poschecks_comp ?C"
  let ?negs = "transaction_negchecks_comp ?C"
  let ?δupd = "λy. absdbupd (unlabel (transaction_updates T)) y (δ y)"

  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

  have 0: "{x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value} = set ?ys"
    using fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"]
    by force

  have 1: "transaction_check_pre (FP, OCC, TI) T δ"
    using δ unfolding transaction_check_comp_def Let_def by fastforce
```

```
  have 2: "transaction_check_post (FP, OCC, TI) T δ"
    using δ T_check unfolding transaction_check_def transaction_check'_def list_all_iff by auto

  have 3: "δ ∈ abs_substs_fun ` set (abs_substs_set ?ys OCC ?poss ?negs msgcs)"
    using δ unfolding transaction_check_comp_def Let_def by force

  show A: ?A when ?A' using that 0 3 x abs_substs_abss_bounded by blast

  have 4: "x ∈ fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)"
    when x': "x ∈ set (transaction_fresh T)"
    using x' admissible_transactionE(7)[OF T_adm] by blast

  have "intruder_synth_mod_timpls FP TI (occurs (absc (?δupd x)))"
    when x': "x ∈ set (transaction_fresh T)"
  proof -
    obtain l ts S where ts:
        "transaction_send T = (l,send⟨ts⟩)#S" "occurs (Var x) ∈ set ts"
      using admissible_transaction_occurs_checksE2[OF T_occ x'] by blast
    hence "occurs (Var x) ∈ set ts" "send⟨ts⟩ ∈ set (unlabel (transaction_send T))"
      using x' unfolding suffix_def by (fastforce, fastforce)
    thus ?thesis using 2 x unfolding transaction_check_post_def by fastforce
  qed
  hence "timpl_closure_set (set FP) (set TI) ⊢_c occurs (absc (?δupd x))"
    when x': "x ∈ set (transaction_fresh T)"
    using x' intruder_synth_mod_timpls_is_synth_timpl_closure_set[
            OF TI, of FP "occurs (absc (?δupd x))"]
    by argo
  hence "Abs (?δupd x) ∈ ⋃(funs_term ` timpl_closure_set (set FP) (set TI))"
    when x': "x ∈ set (transaction_fresh T)"
    using x' ideduct_synth_priv_fun_in_ik[
            of "timpl_closure_set (set FP) (set TI)" "occurs (absc (?δupd x))"]
    by simp
  hence "∃t ∈ timpl_closure_set (set FP) (set TI). Abs (?δupd x) ∈ funs_term t"
    when x': "x ∈ set (transaction_fresh T)"
    using x' by force
  hence 5: "?δupd x ∈ set OCC" when x': "x ∈ set (transaction_fresh T)"
    using x' OCC by fastforce

  have 6: "?δupd x ∈ set OCC" when x': "x ∉ set (transaction_fresh T)"
  proof (cases "δ x = ?δupd x")
    case False
    hence "(δ x, ?δupd x) ∈ (set TI)⁺" "δ x ∈ set OCC"
      using A 2 x' x TI
      unfolding transaction_check_post_def fv_list_sst_is_fv_sst Let_def
                in_trancl_closure_iff_in_trancl_fun[symmetric]
                List.member_def
      by blast+
    thus ?thesis using timpl_closure_set_absc_subset_in[OF OCC(2)] by blast
  qed (simp add: A x' x(1))

  show ?B by (metis 5 6)
qed

lemma transaction_prop3:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trms_lsst 𝒜)"
    and α: "transaction_renaming_subst α P (vars_lsst 𝒜)"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
```

```
      "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢_c t"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
      "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
  shows "∀x ∈ set (transaction_fresh T). (ξ ∘_s σ ∘_s α) x · ℐ ·_α α_0 (db_lsst 𝒜 ℐ) = absc {}" (is ?A)
    and "∀t ∈ trms_lsst (transaction_receive T).
            intruder_synth_mod_timpls FP TI (t · (ξ ∘_s σ ∘_s α) · ℐ ·_α α_0 (db_lsst 𝒜 ℐ))" (is ?B)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀s. select⟨Var x,Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
                ⟶ (∃ss. (ξ ∘_s σ ∘_s α) x · ℐ ·_α α_0 (db_lsst 𝒜 ℐ) = absc ss ∧ s ∈ ss)" (is ?C)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀s. ⟨Var x in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
                ⟶ (∃ss. (ξ ∘_s σ ∘_s α) x · ℐ ·_α α_0 (db_lsst 𝒜 ℐ) = absc ss ∧ s ∈ ss)" (is ?D)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀s. ⟨Var x not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
                ⟶ (∃ss. (ξ ∘_s σ ∘_s α) x · ℐ ·_α α_0 (db_lsst 𝒜 ℐ) = absc ss ∧ s ∉ ss)" (is ?E)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T). Γ_v x = TAtom Value ⟶
         (ξ ∘_s σ ∘_s α) x · ℐ ·_α α_0 (db_lsst 𝒜 ℐ) ∈ absc ` set OCC" (is ?F)
proof -
  let ?T' = "dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
  let ?ϑ = "ξ ∘_s σ ∘_s α"

  define a0 where "a0 ≡ α_0 (db_lsst 𝒜 ℐ)"
  define a0' where "a0' ≡ α_0 (db_lsst (𝒜@?T') ℐ)"
  define fv_AT' where "fv_AT' ≡ fv_lsst (𝒜@?T')"

  note T_adm = bspec[OF P(1) T]
  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]
  note T_adm' = admissible_transaction_is_wellformed_transaction(2-4)[OF T_adm]

  note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm ξ]
  hence ξ_elim: "?ϑ = σ ∘_s α" by simp

  have ℐ': "interpretation_subst ℐ" "wt_subst ℐ" "wf_trms (subst_range ℐ)"
           "∀n. PubConst Value n ∉ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
           "∀n. Abs n ∉ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
           "∀n. PubConst Value n ∉ ⋃(funs_term ` (ℐ ` fv_AT'))"
           "∀n. Abs n ∉ ⋃(funs_term ` (ℐ ` fv_AT'))"
    using ℐ admissible_transaction_occurs_checks_prop'[
            OF 𝒜_reach welltyped_constraint_model_prefix[OF ℐ] P P_occ]
          admissible_transaction_occurs_checks_prop'[
            OF reachable_constraints.step[OF 𝒜_reach T ξ σ α] ℐ P P_occ]
    unfolding welltyped_constraint_model_def constraint_model_def is_Val_def is_Abs_def fv_AT'_def
    by (meson,meson,meson,metis,metis,metis,metis)

  have T_no_pubconsts: "∀n. PubConst Value n ∉ ⋃(funs_term ` trms_transaction T)"
    and T_no_abss: "∀n. Abs n ∉ ⋃(funs_term ` trms_transaction T)"
    and T_fresh_vars_value_typed: "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
    and T_fv_const_typed: "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    using protocol_transaction_vars_TAtom_typed
          protocol_transactions_no_pubconsts
          protocol_transactions_no_abss
          funs_term_Fun_subterm P T
    by (fast, fast, fast, fast)

  have wt_σαℐ: "wt_subst (σ ∘_s α ∘_s ℐ)"
    using ℐ'(2) wt_subst_compose transaction_fresh_subst_wt[OF σ]
```

```
      transaction_renaming_subst_wt[OF α]
    by blast

  have 1: "?ϑ y · ℐ = σ y" when "y ∈ set (transaction_fresh T)" for y
    using transaction_fresh_subst_grounds_domain[OF σ that] subst_compose[of σ α y]
    unfolding ξ_elim by (simp add: subst_ground_ident)

  have 2: "?ϑ y · ℐ ∉ subterms_set (trms_lsst 𝒜)" when "y ∈ set (transaction_fresh T)" for y
    using 1[OF that] that σ unfolding transaction_fresh_subst_def by auto

  have 3: "∀x ∈ fv_lsst 𝒜. Γ_v x = TAtom Value ⟶
          (∃B. prefix B 𝒜 ∧ x ∉ fv_lsst B ∧ ℐ x ∈ subterms_set (trms_lsst B))"
    by (metis welltyped_constraint_model_prefix[OF ℐ]
            constraint_model_Value_var_in_constr_prefix[OF 𝒜_reach _ P P_occ])

  have 4: "∃n. ?ϑ y · ℐ = Fun (Val n) []"
    when "y ∈ fv_transaction T" "Γ_v y = TAtom Value" for y
    using transaction_var_becomes_Val[
          OF reachable_constraints.step[OF 𝒜_reach T ξ σ α] ℐ ξ σ α P P_occ T]
        that T_fv_const_typed Γ_v_TAtom''[of y]
    by metis

  have ℐ_is_T_model: "strand_sem_stateful (ik_lsst 𝒜 ·_set ℐ) (set (db_lsst 𝒜 ℐ)) (unlabel ?T') ℐ"
    using ℐ unlabel_append[of 𝒜 ?T'] db_sst_set_is_dbupd_sst[of "unlabel 𝒜" ℐ "[]"]
        strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "unlabel ?T'" ℐ]
    by (simp add: welltyped_constraint_model_def constraint_model_def db_sst_def)

  have T_rcv_no_val_bvars: "bvars_lsst (transaction_receive T) ∩ subst_domain ?ϑ = {}"
    using admissible_transaction_no_bvars[OF T_adm] bvars_transaction_unfold[of T] by blast

  show ?A
  proof
    fix y assume y: "y ∈ set (transaction_fresh T)"
    then obtain yn where yn: "(ξ ∘_s σ ∘_s α) y · ℐ = Fun (Val yn) []" "Fun (Val yn) [] ∈ subst_range
σ"
      by (metis ξ_elim T_fresh_vars_value_typed transaction_fresh_subst_sends_to_val'[OF σ])

    { — since y is fresh (ξ ∘_s σ ∘_s α) y · ℐ cannot be part of the database state of ℐ 𝒜
      fix t' s assume t': "insert⟨t',s⟩ ∈ set (unlabel 𝒜)" "t' · ℐ = Fun (Val yn) []"
      then obtain z where t'_z: "t' = Var z" using 2[OF y] yn(1) by (cases t') auto
      hence z: "z ∈ fv_lsst 𝒜" "ℐ z = (ξ ∘_s σ ∘_s α) y · ℐ" using t' yn(1) by force+
      hence z': "Γ_v z = TAtom Value"
        by (metis Γ.simps(1) Γ_consts_simps(2) t'(2) t'_z wt_subst_trm'' ℐ'(2))

      obtain B where B: "prefix B 𝒜" "ℐ z ∈ subterms_set (trms_lsst B)" using z z' 3 by fastforce
      hence "∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst B)"
        using transaction_fresh_subst_range_fresh(1)[OF σ] trms_sst_unlabel_prefix_subset(1)[of B]
        unfolding prefix_def by fast
      hence False using B(2) 1[OF y] z yn(1) by (metis subst_imgI term.distinct(1))
    } hence "∄s. (?ϑ y · ℐ, s) ∈ set (db_lsst 𝒜 ℐ)"
      using db_sst_in_cases[of "?ϑ y · ℐ" _ "unlabel 𝒜" ℐ "[]"] yn(1)
      by (force simp add: db_sst_def)
    thus "?ϑ y · ℐ ·_α α₀ (db_lsst 𝒜 ℐ) = absc {}"
      using to_abs_empty_iff_notin_db[of yn "db'_lsst 𝒜 ℐ []"] yn(1)
      by (simp add: db_sst_def)
  qed

  show receives_covered: ?B
  proof
    fix t assume t: "t ∈ trms_lsst (transaction_receive T)"
    hence t_in_T: "t ∈ trms_transaction T"
      using trms_sst_unlabel_prefix_subset(1)[of "transaction_receive T"]
      unfolding transaction_strand_def by fast
```

```
obtain ts where ts: "t ∈ set ts" "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T))"
  using t wellformed_transaction_send_receive_trm_cases(1)[OF T_wf] by blast

have t_rcv: "receive⟨ts ·list ?ϑ⟩ ∈ set (unlabel (transaction_receive T ·lsst ?ϑ))"
  using subst_lsst_unlabel_member[of "receive⟨ts⟩" "transaction_receive T" ?ϑ]
        trmssst_in[OF t] ts
  by fastforce

have "list_all (λt. iklsst 𝒜 ·set ℐ ⊢ t · ?ϑ · ℐ) ts"
  using wellformed_transaction_sem_receives[OF T_wf ℐ_is_T_model t_rcv]
  unfolding list_all_iff by fastforce
hence *: "iklsst 𝒜 ·set ℐ ⊢ t · ?ϑ · ℐ" using ts(1) unfolding list_all_iff by fast

have t_fv: "fv (t · ?ϑ) ⊆ fv_AT'"
  using fvsst_append[of "unlabel 𝒜"] unlabel_append[of 𝒜]
        fvsst_unlabel_duallsst_eq[of "transaction_strand T ·lsst ?ϑ"]
        ts(1) t_rcv fv_transaction_subst_unfold[of T ?ϑ]
  unfolding fv_AT'_def by force

have **: "∀t ∈ (iklsst 𝒜 ·set ℐ) ·αset a0. timpl_closure_set (set FP) (set TI) ⊢c t"
  using FP(3) by (auto simp add: a0_def abs_intruder_knowledge_def)

note lms1 = pubval_terms_subst[OF _ pubval_terms_subst_range_disj[
              OF transaction_fresh_subst_has_no_pubconsts_abss(1)[OF σ], of t]]
            pubval_terms_subst[OF _ pubval_terms_subst_range_disj[
              OF transaction_renaming_subst_has_no_pubconsts_abss(1)[OF α], of "t · σ"]]

note lms2 = abs_terms_subst[OF _ abs_terms_subst_range_disj[
              OF transaction_fresh_subst_has_no_pubconsts_abss(2)[OF σ], of t]]
            abs_terms_subst[OF _ abs_terms_subst_range_disj[
              OF transaction_renaming_subst_has_no_pubconsts_abss(2)[OF α], of "t · σ"]]

have "t ∈ (⋃T∈set P. trms_transaction T)" "fv (t · σ ∘s α · ℐ) = {}"
  using t_in_T T interpretation_grounds[OF ℐ'(1)] by fast+
moreover have "wftrms (subst_range (σ ∘s α ∘s ℐ))"
  using wf_trm_subst_rangeI[of σ, OF transaction_fresh_subst_is_wf_trm[OF σ]]
        wf_trm_subst_rangeI[of α, OF transaction_renaming_subst_is_wf_trm[OF α]]
        wf_trms_subst_compose[of σ α, THEN wf_trms_subst_compose[OF _ ℐ'(3)]]
  by blast
moreover
have "t ∉ pubval_terms"
  using t_in_T T_no_pubconsts funs_term_Fun_subterm
  unfolding is_PubConstValue_def is_PubConst_def by fastforce
hence "t · ?ϑ ∉ pubval_terms"
  using lms1 T_fresh_vars_value_typed
  unfolding ξ_elim by auto
hence "t · ?ϑ · ℐ ∉ pubval_terms"
  using ℐ'(6) t_fv pubval_terms_subst'[of "t · ?ϑ" ℐ]
  by auto
moreover have "t ∉ abs_terms"
  using t_in_T T_no_abss funs_term_Fun_subterm
  unfolding is_Abs_def by force
hence "t · ?ϑ ∉ abs_terms"
  using lms2 T_fresh_vars_value_typed
  unfolding ξ_elim by auto
hence "t · ?ϑ · ℐ ∉ abs_terms"
  using ℐ'(7) t_fv abs_terms_subst'[of "t · ?ϑ" ℐ]
  by auto
ultimately have ***:
    "t · ξ ∘s σ ∘s α · ℐ ∈ GSMP (⋃T∈set P. trms_transaction T) - (pubval_terms ∪ abs_terms)"
  using SMP.Substitution[OF SMP.MP[of t "⋃T∈set P. trms_transaction T"], of "σ ∘s α ∘s ℐ"]
        subst_subst_compose[of t ?ϑ ℐ] wt_σαℐ
```

```
      unfolding GSMP_def ξ_elim by fastforce

    have ****:
        "ik_lsst 𝒜 ·set ℐ ⊆ GSMP (⋃T∈set P. trms_transaction T) - (pubval_terms ∪ abs_terms)"
      using reachable_constraints_GSMP_no_pubvals_abss[OF 𝒜_reach P ℐ'(1-5)]
            ik_sst_trms_sst_subset[of "unlabel 𝒜"]
      by blast

    show "intruder_synth_mod_timpls FP TI (t · ?ϑ · ℐ ·α α0 (db_lsst 𝒜 ℐ))"
      using deduct_FP_if_deduct[OF **** ** * ***] deducts_eq_if_analyzed[OF FP(1)]
            intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI, of FP]
      unfolding a0_def by force
  qed

  show ?C
  proof (intro ballI allI impI)
    fix y s
    assume y: "y ∈ fv_transaction T - set (transaction_fresh T)"
       and s: "select⟨Var y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))"
    hence "select⟨Var y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_strand T))"
      unfolding transaction_strand_def unlabel_def by auto
    hence y_val: "Γv y = TAtom Value"
      using transaction_selects_are_Value_vars[OF T_wf T_adm'(1)]
      by fastforce

    have "select⟨?ϑ y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T ·lsst ?ϑ))"
      using subst_lsst_unlabel_member[OF s]
      by fastforce
    hence "((ξ ∘s σ ∘s α) y · ℐ, Fun (Set s) []) ∈ set (db_lsst 𝒜 ℐ)"
      using wellformed_transaction_sem_pos_checks[
              OF T_wf ℐ_is_T_model,
              of Assign "(ξ ∘s σ ∘s α) y" "Fun (Set s) []"]
      by simp
    thus "∃ss. (ξ ∘s σ ∘s α) y · ℐ ·α α0 (db_lsst 𝒜 ℐ) = absc ss ∧ s ∈ ss"
      using to_abs_alt_def[of "db_lsst 𝒜 ℐ"] 4[of y] y y_val by auto
  qed

  show ?D
  proof (intro ballI allI impI)
    fix y s
    assume y: "y ∈ fv_transaction T - set (transaction_fresh T)"
       and s: "⟨Var y in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))"
    hence "⟨Var y in Fun (Set s) []⟩ ∈ set (unlabel (transaction_strand T))"
      unfolding transaction_strand_def unlabel_def by auto
    hence y_val: "Γv y = TAtom Value"
      using transaction_inset_checks_are_Value_vars[OF T_adm]
      by fastforce

    have "⟨?ϑ y in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T ·lsst ?ϑ))"
      using subst_lsst_unlabel_member[OF s]
      by fastforce
    hence "(?ϑ y · ℐ, Fun (Set s) []) ∈ set (db_lsst 𝒜 ℐ)"
      using wellformed_transaction_sem_pos_checks[
              OF T_wf ℐ_is_T_model,
              of Check "?ϑ y" "Fun (Set s) []"]
      by simp
    thus "∃ss. ?ϑ y · ℐ ·α α0 (db_lsst 𝒜 ℐ) = absc ss ∧ s ∈ ss"
      using to_abs_alt_def[of "db_lsst 𝒜 ℐ"] 4[of y] y y_val by auto
  qed

  show ?E
  proof (intro ballI allI impI)
    fix y s
```

```
    assume y: "y ∈ fv_transaction T - set (transaction_fresh T)"
       and s: "⟨Var y not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))"
    hence "⟨Var y not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_strand T))"
      unfolding transaction_strand_def unlabel_def by auto
    hence y_val: "Γv y = TAtom Value"
      using transaction_notinset_checks_are_Value_vars(1)[
              OF T_adm, of "[]" "[]" "[(Var y, Fun (Set s) [])]" "Var y" "Fun (Set s) []"]
      by force

    have "⟨?ϑ y not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T ·lsst ?ϑ))"
      using subst_lsst_unlabel_member[OF s]
      by fastforce
    hence "(?ϑ y · ℐ, Fun (Set s) []) ∉ set (dblsst 𝒜 ℐ)"
      using wellformed_transaction_sem_neg_checks'(2)[
              OF T_wf ℐ_is_T_model,
              of "[]" "?ϑ y" "Fun (Set s) []"]
      by simp
    moreover have "list_all admissible_transaction_updates P"
      using Ball_set[of P "admissible_transaction"] P(1)
            Ball_set[of P admissible_transaction_updates]
            admissible_transaction_is_wellformed_transaction(3)
      by fast
    moreover have "list_all wellformed_transaction P"
      using P(1) Ball_set[of P "admissible_transaction"] Ball_set[of P wellformed_transaction]
            admissible_transaction_is_wellformed_transaction(1)
      by blast
    ultimately have "((ξ ∘s σ ∘s α) y · ℐ, Fun (Set s) S) ∉ set (dblsst 𝒜 ℐ)" for S
      using reachable_constraints_dblsst_set_args_empty[OF 𝒜_reach]
      unfolding admissible_transaction_updates_def
      by auto
    thus "∃ss. (ξ ∘s σ ∘s α) y · ℐ ·α α0 (dblsst 𝒜 ℐ) = absc ss ∧ s ∉ ss"
      using to_abs_alt_def[of "dblsst 𝒜 ℐ"] 4[of y] y y_val by auto
  qed

  show ?F
  proof (intro ballI impI)
    fix y assume y: "y ∈ fv_transaction T - set (transaction_fresh T)" "Γv y = TAtom Value"
    then obtain yn where yn: "?ϑ y · ℐ = Fun (Val yn) []" using 4 by blast
    hence y_abs: "?ϑ y · ℐ ·α α0 (dblsst 𝒜 ℐ) = Fun (Abs (α0 (dblsst 𝒜 ℐ) yn)) []" by simp

    have "y ∈ fvlsst (transaction_receive T) ∨ (y ∈ fvlsst (transaction_checks T) ∧
          (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ y ∈ fv t ∪ fv s))"
      using admissible_transaction_fv_in_receives_or_selects[OF T_adm] y by blast
    thus "?ϑ y · ℐ ·α α0 (dblsst 𝒜 ℐ) ∈ absc ` set OCC"
    proof
      assume "y ∈ fvlsst (transaction_receive T)"
      then obtain ts where ts:
          "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T))" "y ∈ fvset (set ts)"
        using wellformed_transaction_unlabel_cases(1)[OF T_wf]
        by (force simp add: unlabel_def)

      have *: "?ϑ y · ℐ ∈ subtermsset (set ts ·set ?ϑ ∘s ℐ)"
              "list_all (λt. timpl_closure_set (set FP) (set TI) ⊢c t · ?ϑ · ℐ ·α α0 (dblsst 𝒜 ℐ)) ts"
        using ts fv_subterms_substI[of y _ "?ϑ ∘s ℐ"] subst_compose[of ?ϑ ℐ y]
              subterms_subst_subset[of "?ϑ ∘s ℐ"] receives_covered
        unfolding intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI, symmetric] list_all_iff
        by fastforce+

      have "Abs (α0 (dblsst 𝒜 ℐ) yn) ∈ ⋃(funs_term ` (timpl_closure_set (set FP) (set TI)))"
        using * y_abs abs_subterms_in[of "?ϑ y · ℐ" _ "α0 (dblsst 𝒜 ℐ)"]
              ideduct_synth_priv_fun_in_ik[
                OF _ funs_term_Fun_subterm'[of "Abs (α0 (dblsst 𝒜 ℐ) yn)" "[]"]]
        unfolding list_all_iff by fastforce
```

```
      hence "?ϑ y · ℐ ·α α0 (dblsst 𝒜 ℐ) ∈ subtermsset (timpl_closure_set (set FP) (set TI))"
        using y_abs wf_trms_subterms[OF timpl_closure_set_wf_trms[OF FP(2), of "set TI"]]
              funs_term_Fun_subterm[of "Abs (α0 (dblsst 𝒜 ℐ) yn)"]
        unfolding wftrm_def by fastforce
      hence "funs_term (?ϑ y · ℐ ·α α0 (dblsst 𝒜 ℐ))
              ⊆ (⋃t ∈ timpl_closure_set (set FP) (set TI). funs_term t)"
        using funs_term_subterms_eq(2)[of "timpl_closure_set (set FP) (set TI)"] by blast
      thus ?thesis using y_abs OCC(1) by fastforce
    next
      assume "y ∈ fvlsst (transaction_checks T) ∧
          (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ y ∈ fv t ∪ fv s)"
      then obtain t t' where
          "select⟨t,t'⟩ ∈ set (unlabel (transaction_checks T))" "y ∈ fv t ∪ fv t'"
        by blast
      then obtain l s where "(l,select⟨Var y, Fun (Set s) []⟩) ∈ set (transaction_checks T)"
        using admissible_transaction_strand_step_cases(2)[OF T_adm]
        unfolding unlabel_def by fastforce
      then obtain U where U:
          "prefix (U@[(l,select⟨Var y, Fun (Set s) []⟩)]) (transaction_checks T)"
        using in_set_conv_decomp[of "(l, select⟨Var y,Fun (Set s) []⟩)" "transaction_checks T"]
        by (auto simp add: prefix_def)
      hence "select⟨Var y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))"
        by (force simp add: prefix_def unlabel_def)
      hence "select⟨?ϑ y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T ·lsst ?ϑ))"
        using subst_lsst_unlabel_member
        by fastforce
      hence "(Fun (Val yn) [], Fun (Set s) []) ∈ set (dblsst 𝒜 ℐ)"
        using yn wellformed_transaction_sem_pos_checks[
                OF T_wf ℐ_is_T_model, of Assign "?ϑ y" "Fun (Set s) []"]
        by fastforce
      hence "Fun (Val yn) [] ∈ subtermsset (trmslsst 𝒜) ·set ℐ"
        using dbsst_in_cases[of "Fun (Val yn) []"]
        by (fastforce simp add: dbsst_def)
      thus ?thesis
        using OCC(3) yn abs_in[of "Fun (Val yn) []" _ "α0 (dblsst 𝒜 ℐ)"]
        unfolding abs_value_constants_def
        by (metis (mono_tags, lifting) mem_Collect_eq subsetCE)
    qed
  qed
qed

lemma transaction_prop4:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trmslsst 𝒜)"
    and α: "transaction_renaming_subst α P (varslsst 𝒜)"
    and P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and x: "x ∈ set (transaction_fresh T)"
    and y: "y ∈ fv_transaction T - set (transaction_fresh T)" "Γv y = TAtom Value"
  shows "(ξ ∘s σ ∘s α) x · ℐ ∉ subtermsset (trmslsst (𝒜 ·lsst ℐ))" (is ?A)
    and "(ξ ∘s σ ∘s α) y · ℐ ∈ subtermsset (trmslsst (𝒜 ·lsst ℐ))" (is ?B)
proof -
  let ?T' = "duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"

  from ℐ have ℐ': "welltyped_constraint_model ℐ 𝒜"
    using welltyped_constraint_model_prefix by auto

  note T_P_adm = bspec[OF P(1)]
  note T_adm = bspec[OF P(1) T]
```

```
note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

have be: "bvars_lsst 𝒜 = {}"
  using T_P_adm 𝒜_reach reachable_constraints_no_bvars admissible_transaction_no_bvars(2)
  by blast

have T_no_bvars: "fv_transaction T = vars_transaction T"
  using admissible_transaction_no_bvars[OF T_adm] by simp

note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm ξ]

have ℐ_wt: "wt_subst ℐ" by (metis ℐ welltyped_constraint_model_def)

have x_val: "fst x = TAtom Value"
  using x admissible_transactionE(14)[OF T_adm] unfolding list_all_iff by blast

obtain xn where xn: "σ x = Fun (Val xn) []"
  using x_val transaction_fresh_subst_sends_to_val[OF σ x] Γ_v_TAtom''(2)[of x] by meson
hence xnxn: "(ξ ∘_s σ ∘_s α) x = Fun (Val xn) []"
  unfolding subst_compose_def ξ_empty by auto

from xn xnxn have a0: "(ξ ∘_s σ ∘_s α) x · ℐ = Fun (Val xn) []"
  by auto

have b0: "Γ_v x = TAtom Value"
  using P x T protocol_transaction_vars_TAtom_typed(3)
  by metis

note 0 = a0 b0

have σ_x_nin_A: "σ x ∉ subterms_set (trms_lsst 𝒜)"
proof -
  have "σ x ∈ subst_range σ"
    by (metis transaction_fresh_subst_sends_to_val[OF σ x b0])
  moreover
  have "∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst 𝒜)"
    using σ transaction_fresh_subst_def[of σ T "trms_lsst 𝒜"] by blast
  ultimately
  show ?thesis
    by auto
qed

have *: "y ∉ set (transaction_fresh T)"
   using assms by auto

have **: "y ∈ fv_lsst (transaction_receive T) ∨ (y ∈ fv_lsst (transaction_checks T) ∧
         (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ y ∈ fv t ∪ fv s))"
  using * y admissible_transaction_fv_in_receives_or_selects[OF T_adm]
  by blast

have y_fv: "y ∈ fv_transaction T" using y fv_transaction_unfold by blast

have y_val: "fst y = TAtom Value" using y(2) Γ_v_TAtom''(2) by blast

have "σ x · ℐ ∉ subterms_set (trms_lsst (𝒜 ·_lsst ℐ))"
proof (rule ccontr)
  assume "¬σ x · ℐ ∉ subterms_set (trms_lsst (𝒜 ·_lsst ℐ))"
  then have a: "σ x · ℐ ∈ subterms_set (trms_lsst (𝒜 ·_lsst ℐ))"
    by auto

  then have σ_x_I_in_A: "σ x · ℐ ∈ subterms_set (trms_lsst 𝒜) ·_set ℐ"
    using reachable_constraints_subterms_subst[OF 𝒜_reach ℐ' P] by blast
```

```
    have "∃u. u ∈ fv_lsst 𝒜 ∧ ℐ u = σ x"
    proof -
      from σ_x_I_in_A have "∃tu. tu ∈ ⋃ (subterms ` (trms_lsst 𝒜)) ∧ tu · ℐ = σ x · ℐ"
        by force
      then obtain tu where tu: "tu ∈ ⋃ (subterms ` (trms_lsst 𝒜)) ∧ tu · ℐ = σ x · ℐ"
        by auto
      then have "tu ≠ σ x"
        using σ_x_nin_A by auto
      moreover
      have "tu · ℐ = σ x"
        using tu by (simp add: xn)
      ultimately
      have "∃u. tu = Var u"
        unfolding xn by (cases tu) auto
      then obtain u where "tu = Var u"
        by auto
      have "u ∈ fv_lsst 𝒜 ∧ ℐ u = σ x"
      proof -
        have "u ∈ vars_lsst 𝒜"
          using ‹tu = Var u› tu var_subterm_trms_sst_is_vars_sst by fastforce
        then have "u ∈ fv_lsst 𝒜"
          using be vars_sst_is_fv_sst_bvars_sst[of "unlabel 𝒜"] by blast
        moreover
        have "ℐ u = σ x"
          using ‹tu = Var u› ‹tu · ℐ = σ x› by auto
        ultimately
        show ?thesis
          by auto
      qed
      then show "∃u. u ∈ fv_lsst 𝒜 ∧ ℐ u = σ x"
        by metis
    qed
    then obtain u where u:
      "u ∈ fv_lsst 𝒜" "ℐ u = σ x"
      by auto
    then have u_TA: "Γ_v u = TAtom Value"
      using P(1) T x_val Γ_v_TAtom''(2)[of x]
            wt_subst_trm''[OF ℐ_wt, of "Var u"] wt_subst_trm''[of σ "Var x"]
            transaction_fresh_subst_wt[OF σ] protocol_transaction_vars_TAtom_typed(3)
      by force
    have "∃B. prefix B 𝒜 ∧ u ∉ fv_lsst B ∧ ℐ u ∈ subterms_set (trms_lsst B)"
      using u u_TA
      by (metis welltyped_constraint_model_prefix[OF ℐ]
                constraint_model_Value_var_in_constr_prefix[OF 𝒜_reach _ P P_occ])
    then obtain B where "prefix B 𝒜 ∧ u ∉ fv_lsst B ∧ ℐ u ∈ subterms_set (trms_lsst B)"
      by blast
    moreover have "⋃(subterms ` trms_lsst xs) ⊆ ⋃(subterms ` trms_lsst ys)"
      when "prefix xs ys"
      for xs ys::"('fun,'atom,'sets,'lbl) prot_strand"
      using that subterms_set_mono trms_sst_mono unlabel_mono set_mono_prefix by metis
    ultimately have "ℐ u ∈ subterms_set (trms_lsst 𝒜)"
      by blast
    then have "σ x ∈ subterms_set (trms_lsst 𝒜)"
      using u by auto
    then show "False"
      using σ_x_nin_A by auto
  qed
  then show ?A
    using eval_term.simps(1)[of _ x σ]
    unfolding subst_compose_def xn ξ_empty by auto

  from ** show ?B
  proof
```

```
  define T' where "T' ≡ transaction_receive T"
  define ϑ where "ϑ ≡ ξ ∘s σ ∘s α"

  assume y: "y ∈ fv_lsst (transaction_receive T)"
  hence "Var y ∈ subterms_set (trms_lsst T')" by (metis T'_def fv_sst_is_subterm_trms_sst)
  then obtain z where z: "z ∈ set (unlabel T')" "Var y ∈ subterms_set (trms_sstp z)"
    by (induct T') auto

  have "is_Receive z"
    using Ball_set[of "unlabel T'" is_Receive] z(1)
          admissible_transaction_is_wellformed_transaction(1)[OF T_adm]
    unfolding wellformed_transaction_def T'_def
    by blast
  then obtain tys where "z = receive⟨tys⟩" by (cases z) auto
  hence tys: "receive⟨tys ·list ϑ⟩ ∈ set (unlabel (T' ·lsst ϑ))" "ϑ y ∈ subterms_set (set tys ·set ϑ)"
    using z subst_mono unfolding subst_apply_labeled_stateful_strand_def unlabel_def by force+
  hence y_deduct: "list_all (λt. ik_lsst 𝒜 ·set ℐ ⊢ t · ϑ · ℐ) tys"
    using transaction_receive_deduct[OF T_wf _ ξ σ α] ℐ
    unfolding T'_def ϑ_def welltyped_constraint_model_def list_all_iff by auto

  obtain ty where ty: "ty ∈ set tys" "ϑ y ⊑ ty · ϑ" "ik_lsst 𝒜 ·set ℐ ⊢ ty · ϑ · ℐ"
    using tys y_deduct unfolding list_all_iff by blast

  obtain zn where zn: "(ξ ∘s σ ∘s α) y · ℐ = Fun (Val zn) []"
    using transaction_var_becomes_Val[
            OF reachable_constraints.step[OF 𝒜_reach T ξ σ α] ℐ ξ σ α P P_occ T y_fv y_val]
    by metis

  have "(ξ ∘s σ ∘s α) y · ℐ ∈ subterms_set (ik_lsst 𝒜 ·set ℐ)"
    using ty tys(2) y_deduct private_fun_deduct_in_ik[of _ _ "Val zn"]
    by (metis ϑ_def zn subst_mono public.simps(3))
  thus ?B
    using ik_sst_subst[of "unlabel 𝒜" ℐ] unlabel_subst[of 𝒜 ℐ]
          subterms_set_mono[OF ik_sst_trms_sst_subset[of "unlabel (𝒜 ·lsst ℐ)"]]
    by fastforce
next
  assume y': "y ∈ fv_lsst (transaction_checks T) ∧
              (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ y ∈ fv t ∪ fv s)"
  then obtain s where s: "select⟨Var y,s⟩ ∈ set (unlabel (transaction_checks T))"
                         "fst y = TAtom Value"
    using admissible_transaction_strand_step_cases(1,2)[OF T_adm] by fastforce

  obtain z zn where zn: "(ξ ∘s σ ∘s α) y = Var z" "ℐ z = Fun (Val zn) []"
    using transaction_var_becomes_Val[
            OF reachable_constraints.step[OF 𝒜_reach T ξ σ α] ℐ ξ σ α P P_occ T y_fv s(2)]
          transaction_decl_fresh_renaming_substs_range(4)[OF ξ σ α _ *]
          transaction_decl_subst_empty_inv[OF ξ[unfolded ξ_empty]]
    by auto

  have transaction_selects_db_here:
      "⋀n s. select⟨Var (TAtom Value, n), Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
              ⟹ (α (TAtom Value, n) · ℐ, Fun (Set s) []) ∈ set (db_lsst 𝒜 ℐ)"
    using transaction_selects_db[OF T_adm _ ξ σ α] ℐ
    unfolding welltyped_constraint_model_def by auto

  have "∃n. y = (Var Value, n)"
    using T Γv_TAtom_inv(2) y_fv y(2)
    by blast
  moreover
  have "admissible_transaction_checks T"
    using admissible_transaction_is_wellformed_transaction(2)[OF T_adm]
    by blast
  then have "is_Fun_Set (the_set_term (select⟨Var y,s⟩))"
```

```
      using s unfolding admissible_transaction_checks_def
      by auto
    then have "∃ss. s = Fun (Set ss) []"
      using is_Fun_Set_exi
      by auto
    ultimately
    obtain n ss where nss: "y = (TAtom Value, n)" "s = Fun (Set ss) []"
      by auto
    then have "select⟨Var (TAtom Value, n), Fun (Set ss) []⟩ ∈ set (unlabel (transaction_checks T))"
      using s by auto
    then have in_db: "(α (TAtom Value, n) · ℐ, Fun (Set ss) []) ∈ set (db_lsst 𝒜 ℐ)"
      using transaction_selects_db_here[of n ss] by auto
    have "(ℐ z, s) ∈ set (db_lsst 𝒜 ℐ)"
    proof -
      have "(α y · ℐ, s) ∈ set (db_lsst 𝒜 ℐ)"
        using in_db nss by auto
      moreover
      have "α y = Var z"
        using zn ξ_empty * σ
        by (metis (no_types, opaque_lifting) subst_compose_def subst_imgI subst_to_var_is_var
                term.distinct(1) transaction_fresh_subst_def var_comp(2))
      then have "α y · ℐ = ℐ z"
        by auto
      ultimately
      show "(ℐ z, s) ∈ set (db_lsst 𝒜 ℐ)"
        by auto
    qed
    then have "∃t' s'. insert⟨t',s'⟩ ∈ set (unlabel 𝒜) ∧ ℐ z = t' · ℐ ∧ s = s' · ℐ"
      using db_sst_in_cases[of "ℐ z" s "unlabel 𝒜" ℐ "[]"] unfolding db_sst_def by auto
    then obtain t' s' where t's': "insert⟨t',s'⟩ ∈ set (unlabel 𝒜) ∧ ℐ z = t' · ℐ ∧ s = s' · ℐ"
      by auto
    then have "t' ∈ subterms_set (trms_lsst 𝒜)"
      by force
    then have "t' · ℐ ∈ (subterms_set (trms_lsst 𝒜)) ·_set ℐ"
      by auto
    then have "ℐ z ∈ (subterms_set (trms_lsst 𝒜)) ·_set ℐ"
      using t's' by auto
    then have "ℐ z ∈ subterms_set (trms_lsst (𝒜 ·_lsst ℐ))"
      using reachable_constraints_subterms_subst[
              OF 𝒜_reach welltyped_constraint_model_prefix[OF ℐ] P]
      by auto
    then show ?B
      using zn(1) by simp
  qed
qed

lemma transaction_prop5:
  fixes T ξ σ α 𝒜 ℐ T' a0 a0' ϑ
  defines "T' ≡ dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
    and "a0 ≡ α₀ (db_lsst 𝒜 ℐ)"
    and "a0' ≡ α₀ (db_lsst (𝒜@T') ℐ)"
    and "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@T')"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trms_lsst 𝒜)"
    and α: "transaction_renaming_subst α P (vars_lsst 𝒜)"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢_c t"
    and OCC:
```

```
    "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
    "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
    "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
  and TI:
    "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  and P:
    "∀T ∈ set P. admissible_transaction' T"
  and P_occ:
    "∀T ∈ set P. admissible_transaction_occurs_checks T"
  and step: "list_all (transaction_check (FP, OCC, TI)) P"
 shows "∃δ ∈ abs_substs_fun ` set (transaction_check_comp (λ_ _. True) (FP, OCC, TI) T).
        ∀x ∈ fv_transaction T. Γ_v x = TAtom Value ⟶
           (ξ ∘_s σ ∘_s α) x · ℐ ·_α a0 = absc (δ x) ∧
           (ξ ∘_s σ ∘_s α) x · ℐ ·_α a0' = absc (absdbupd (unlabel (transaction_updates T)) x (δ x))"
proof -
 define comp0 where
   "comp0 ≡ abs_substs_fun ` set (transaction_check_comp (λ_ _. True) (FP, OCC, TI) T)"
 define check0 where "check0 ≡ transaction_check (FP, OCC, TI) T"
 define upd where "upd ≡ λδ x. absdbupd (unlabel (transaction_updates T)) x (δ x)"
 define b0 where "b0 ≡ λx. THE b. absc b = (ξ ∘_s σ ∘_s α) x · ℐ ·_α a0"

 note all_defs = comp0_def check0_def a0_def a0'_def upd_def b0_def ϑ_def T'_def

 have 𝒜_wf_trms: "wf_trms (trms_lsst 𝒜)"
   by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P(1) 𝒜_reach)

 have ℐ_interp: "interpretation_subst ℐ"
   and ℐ_wt: "wt_subst ℐ"
   and ℐ_wf_trms: "wf_trms (subst_range ℐ)"
   by (metis ℐ welltyped_constraint_model_def constraint_model_def,
       metis ℐ welltyped_constraint_model_def,
       metis ℐ welltyped_constraint_model_def constraint_model_def)

 have ℐ_is_T_model: "strand_sem_stateful (ik_lsst 𝒜 ·_set ℐ) (set (db_lsst 𝒜 ℐ)) (unlabel T') ℐ"
   using ℐ unlabel_append[of 𝒜 T'] db_sst_set_is_dbupd_sst[of "unlabel 𝒜" ℐ "[]"]
         strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "unlabel T'" ℐ]
   by (simp add: welltyped_constraint_model_def constraint_model_def db_sst_def)

 note T_adm = bspec[OF P(1) T]
 note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

 have T_no_bvars: "fv_transaction T = vars_transaction T" "bvars_transaction T = {}"
   using admissible_transaction_no_bvars[OF T_adm] by simp_all

 have T_vars_const_typed: "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
   and T_fresh_vars_value_typed: "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
   using T P protocol_transaction_vars_TAtom_typed(2,3)[of T] by simp_all

 note ξ_empty = admissible_transaction_decl_subst_empty[OF T_adm ξ]

 have wt_σαℐ: "wt_subst (ξ ∘_s σ ∘_s α ∘_s ℐ)" and wt_σα: "wt_subst (ξ ∘_s σ ∘_s α)"
   using ℐ_wt wt_subst_compose transaction_decl_fresh_renaming_substs_wt[OF ξ σ α]
   by (blast, blast)

 have T_vars_vals: "∀x ∈ fv_transaction T. ∃n. (ξ ∘_s σ ∘_s α) x · ℐ = Fun (Val n) []"
 proof
   fix x assume x: "x ∈ fv_transaction T"
   have "∃n. (σ ∘_s α) x · ℐ = Fun (Val n) []"
   proof (cases "x ∈ subst_domain σ")
     case True
     then obtain n where "σ x = Fun (Val n) []"
       using transaction_fresh_subst_sends_to_val[OF σ]
             transaction_fresh_subst_domain[OF σ]
```

```
          T_fresh_vars_value_typed
      by metis
    thus ?thesis by (simp add: subst_compose_def)
  next
    case False
    hence *: "(σ ∘s α) x = α x" by (auto simp add: subst_compose_def)

    obtain y where y: "Γv x = Γv y" "α x = Var y"
      using transaction_renaming_subst_wt[OF α]
            transaction_renaming_subst_is_renaming(1)[OF α]
      by (metis Γ.simps(1) prod.exhaust wt_subst_def)
    hence "y ∈ fv_lsst (transaction_strand T ·lsst σ ∘s α)"
      using x * T_no_bvars(2) unlabel_subst[of "transaction_strand T" "σ ∘s α"]
            fv_sst_subst_fv_subset[of x "unlabel (transaction_strand T)" "σ ∘s α"]
      by auto
    hence "y ∈ fv_lsst (𝒜@dual_lsst (transaction_strand T ·lsst σ ∘s α))"
      using fv_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst σ ∘s α"]
            fv_sst_append[of "unlabel 𝒜"] unlabel_append[of 𝒜]
      by auto
    thus ?thesis
      using x y * T P
            constraint_model_Value_term_is_Val[
              OF reachable_constraints.step[OF 𝒜_reach T ξ σ α] ℐ[unfolded T'_def] P P_occ, of y]
            admissible_transaction_Value_vars[of T] ξ_empty
      by simp
  qed
  thus "∃n. (ξ ∘s σ ∘s α) x · ℐ = Fun (Val n) []" using ξ_empty by simp
qed

have T_vars_absc: "∀x ∈ fv_transaction T. ∃!n. (ξ ∘s σ ∘s α) x · ℐ ·α a0 = absc n"
  using T_vars_vals by fastforce
hence "(absc ∘ b0) x = (ξ ∘s σ ∘s α) x · ℐ ·α a0" when "x ∈ fv_transaction T" for x
  using that unfolding b0_def by fastforce
hence T_vars_absc': "t · (absc ∘ b0) = t · (ξ ∘s σ ∘s α) · ℐ ·α a0"
  when "fv t ⊆ fv_transaction T" "∄n T. Fun (Val n) T ∈ subterms t" for t
  using that(1) abs_term_subst_eq'[OF _ that(2), of "ξ ∘s σ ∘s α ∘s ℐ" a0 "absc ∘ b0"]
        subst_compose[of "ξ ∘s σ ∘s α" ℐ] subst_subst_compose[of t "ξ ∘s σ ∘s α" ℐ]
  by fastforce

have "∃δ ∈ comp0. ∀x ∈ fv_transaction T. fst x = TAtom Value ⟶ b0 x = δ x"
proof -
  let ?C = "set (unlabel (transaction_checks T))"
  let ?xs = "fv_transaction T - set (transaction_fresh T)"

  note * = transaction_prop3[OF 𝒜_reach T ℐ[unfolded T'_def] ξ σ α FP OCC TI P P_occ]

  have **:
      "∀x ∈ set (transaction_fresh T). b0 x = {}"
      "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ b0)"
        (is ?B)
  proof -
    show ?B
    proof (intro ballI impI)
      fix t assume t: "t ∈ trms_lsst (transaction_receive T)"
      hence t': "fv t ⊆ fv_transaction T" "∄n T. Fun (Val n) T ∈ subterms t"
        using trms_transaction_unfold[of T] vars_transaction_unfold[of T]
              trms_sst_fv_vars_sst_subset[of t "unlabel (transaction_strand T)"]
              admissible_transactions_no_Value_consts'[OF T_adm]
              wellformed_transaction_send_receive_fv_subset(1)[OF T_wf t(1)]
        by blast+

      have "intruder_synth_mod_timpls FP TI (t · (absc ∘ b0))"
        using t(1) t' *(2) T_vars_absc'
```

```
        by (metis a0_def)
      moreover have "(absc ∘ b0) x = (ϑ b0) x" when "x ∈ fv t" for x
        using that T P admissible_transaction_Value_vars[of T]
              ‹fv t ⊆ fv_transaction T› Γv_TAtom''(2)[of x]
        unfolding ϑ_def by fastforce
      hence "t · (absc ∘ b0) = t · ϑ b0"
        using term_subst_eq[of t "absc ∘ b0" "ϑ b0"] by argo
      ultimately show "intruder_synth_mod_timpls FP TI (t · ϑ b0)"
        using intruder_synth.simps[of "set FP"] by (cases "t · ϑ b0") metis+
    qed
  qed (simp add: *(1) a0_def b0_def)

  have ***: "∀x ∈ ?xs. ∀s. select⟨Var x,Fun (Set s) []⟩ ∈ ?C ⟶ s ∈ b0 x"
            "∀x ∈ ?xs. ∀s. ⟨Var x in Fun (Set s) []⟩ ∈ ?C ⟶ s ∈ b0 x"
            "∀x ∈ ?xs. ∀s. ⟨Var x not in Fun (Set s) []⟩ ∈ ?C ⟶ s ∉ b0 x"
            "∀x ∈ ?xs. fst x = TAtom Value ⟶ b0 x ∈ set OCC"
    unfolding a0_def b0_def
    using *(3,4) apply (force, force)
    using *(5) apply force
    using *(6) admissible_transaction_Value_vars[OF bspec[OF P T]] by force

  show ?thesis
    using transaction_check_comp_in[OF T_adm **[unfolded ϑ_def] ***]
    unfolding comp0_def
    by metis
qed
hence 1: "∃δ ∈ comp0. ∀x ∈ fv_transaction T.
            fst x = TAtom Value ⟶ (ξ ∘s σ ∘s α) x · ℐ ·α a0 = absc (δ x)"
  using T_vars_absc unfolding b0_def a0_def by fastforce

obtain δ where δ:
    "δ ∈ comp0"
    "∀x ∈ fv_transaction T. fst x = TAtom Value ⟶ (ξ ∘s σ ∘s α) x · ℐ ·α a0 = absc (δ x)"
  using 1 by force

have 2: "ϑ x · ℐ ·α α0 (db'lsst (duallsst (A ·lsst ϑ)) ℐ D) = absc (absdbupd (unlabel A) x d)"
  when "ϑ x · ℐ ·α α0 D = absc d"
  and "∀t u. insert⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃y s. t = Var y ∧ u = Fun (Set s) [])"
  and "∀t u. delete⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃y s. t = Var y ∧ u = Fun (Set s) [])"
  and "∀y ∈ fvlsst A. ϑ x · ℐ = ϑ y · ℐ ⟶ x = y"
  and "∀y ∈ fvlsst A. ∃n. ϑ y · ℐ = Fun (Val n) []"
  and x: "ϑ x · ℐ = Fun (Val n) []"
  and D: "∀d ∈ set D. ∃s. snd d = Fun (Set s) []"
  for A::"('fun,'atom,'sets,'lbl) prot_strand" and x ϑ D n d
  using that(2,3,4,5)
proof (induction A rule: List.rev_induct)
  case (snoc a A)
  then obtain l b where a: "a = (l,b)" by (metis surj_pair)

  have IH: "α0 (db'lsst (duallsst (A ·lsst ϑ)) ℐ D) n = absdbupd (unlabel A) x d"
    using snoc unlabel_append[of A "[a]"] a x
    by (simp del: unlabel_append)

  have b_prems: "∀y ∈ fvsstp b. ϑ x · ℐ = ϑ y · ℐ ⟶ x = y"
                "∀y ∈ fvsstp b. ∃n. ϑ y · ℐ = Fun (Val n) []"
    using snoc.prems(3,4) a by (simp_all add: unlabel_def)

  have *: "filter is_Update (unlabel (duallsst (A@[a] ·lsst ϑ))) =
           filter is_Update (unlabel (duallsst (A ·lsst ϑ)))"
          "filter is_Update (unlabel (A@[a])) = filter is_Update (unlabel A)"
    when "¬is_Update b"
    using that a
    by (cases b, simp_all add: duallsst_def unlabel_def subst_apply_labeled_stateful_strand_def)+
```

```
note ** = IH a dual_lsst_subst_append[of A "[a]" ϑ]

note *** = * absdbupd_filter[of "unlabel (A@[a])"]
           absdbupd_filter[of "unlabel A"]
           db_sst_filter[of "unlabel (dual_lsst (A@[a] ·_lsst ϑ))"]
           db_sst_filter[of "unlabel (dual_lsst (A ·_lsst ϑ))"]

note **** = **(2,3) dual_lsst_subst_snoc[of A a ϑ]
            unlabel_append[of "dual_lsst A ·_lsst ϑ" "[dual_lsstp a ·_lsstp ϑ]"]
            db_sst_append[of "unlabel (dual_lsst A ·_lsst ϑ)" "unlabel [dual_lsstp a ·_lsstp ϑ]" ℐ D]

have "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) ℐ D) n = absdbupd (unlabel (A@[a])) x d" using ** ***
proof (cases b)
  case (Insert t t')
  then obtain y s m where y: "t = Var y" "t' = Fun (Set s) []" "ϑ y · ℐ = Fun (Val m) []"
    using snoc.prems(1) b_prems(2) a by (fastforce simp add: unlabel_def)
  hence a': "db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) ℐ D =
             List.insert ((Fun (Val m) [], Fun (Set s) [])) (db'_lsst (dual_lsst A ·_lsst ϑ) ℐ D)"
            "unlabel [dual_lsstp a ·_lsstp ϑ] = [insert⟨ϑ y, Fun (Set s) []⟩]"
            "unlabel [a] = [insert⟨Var y, Fun (Set s) []⟩]"
    using **** Insert by simp_all

  show ?thesis
  proof (cases "x = y")
    case True
    hence "ϑ x · ℐ = ϑ y · ℐ" by simp
    hence "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) ℐ D) n =
           insert s (α₀ (db'_lsst (dual_lsst (A ·_lsst ϑ)) ℐ D) n)"
      by (metis (no_types, lifting) y(3) a'(1) x dual_lsst_subst to_abs_list_insert')
    thus ?thesis using True IH a'(3) absdbupd_append[of "unlabel A"] by (simp add: unlabel_def)
  next
    case False
    hence "ϑ x · ℐ ≠ ϑ y · ℐ" using b_prems(1) y Insert by simp
    hence "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) ℐ D) n = α₀ (db'_lsst (dual_lsst (A ·_lsst ϑ)) ℐ D) n"
      by (metis (no_types, lifting) y(3) a'(1) x dual_lsst_subst to_abs_list_insert)
    thus ?thesis using False IH a'(3) absdbupd_append[of "unlabel A"] by (simp add: unlabel_def)
  qed
next
  case (Delete t t')
  then obtain y s m where y: "t = Var y" "t' = Fun (Set s) []" "ϑ y · ℐ = Fun (Val m) []"
    using snoc.prems(2) b_prems(2) a by (fastforce simp add: unlabel_def)
  hence a': "db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) ℐ D =
             List.removeAll ((Fun (Val m) [], Fun (Set s) [])) (db'_lsst (dual_lsst A ·_lsst ϑ) ℐ D)"
            "unlabel [dual_lsstp a ·_lsstp ϑ] = [delete⟨ϑ y, Fun (Set s) []⟩]"
            "unlabel [a] = [delete⟨Var y, Fun (Set s) []⟩]"
    using **** Delete by simp_all

  have "∃s S. snd d = Fun (Set s) []" when "d ∈ set (db'_lsst (dual_lsst A ·_lsst ϑ) ℐ D)" for d
    using snoc.prems(1,2) db_lsst_dual_lsst_set_ex[OF that _ _ D] by (simp add: unlabel_def)
  moreover {
    fix t::"('fun,'atom,'sets,'lbl) prot_term"
      and D::"(('fun,'atom,'sets,'lbl) prot_term × ('fun,'atom,'sets,'lbl) prot_term) list"
    assume "∀d ∈ set D. ∃s. snd d = Fun (Set s) []"
    hence "removeAll (t, Fun (Set s) []) D = filter (λd. ∄S. d = (t, Fun (Set s) S)) D"
      by (induct D) auto
  } ultimately have a'':
      "List.removeAll ((Fun (Val m) [], Fun (Set s) [])) (db'_lsst (dual_lsst A ·_lsst ϑ) ℐ D) =
       filter (λd. ∄S. d = (Fun (Val m) [], Fun (Set s) S)) (db'_lsst (dual_lsst A ·_lsst ϑ) ℐ D)"
    by simp

  show ?thesis
  proof (cases "x = y")
```

```
        case True
        hence "ϑ x · 𝓘 = ϑ y · 𝓘" by simp
        hence "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) 𝓘 D) n =
              (α₀ (db'_lsst (dual_lsst (A ·_lsst ϑ)) 𝓘 D) n) - {s}"
          using y(3) a'' a'(1) x by (simp add: dual_lsst_subst to_abs_list_remove_all')
        thus ?thesis using True IH a'(3) absdbupd_append[of "unlabel A"] by (simp add: unlabel_def)
      next
        case False
        hence "ϑ x · 𝓘 ≠ ϑ y · 𝓘" using b_prems(1) y Delete by simp
        hence "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) 𝓘 D) n = α₀ (db'_lsst (dual_lsst (A ·_lsst ϑ)) 𝓘 D) n"
          by (metis (no_types, lifting) y(3) a'(1) x dual_lsst_subst to_abs_list_remove_all)
        thus ?thesis using False IH a'(3) absdbupd_append[of "unlabel A"] by (simp add: unlabel_def)
      qed
    qed simp_all
    thus ?case by (simp add: x)
  qed (simp add: that(1))

  have 3: "x = y"
    when xy: "(ξ ∘s σ ∘s α) x · 𝓘 = (ξ ∘s σ ∘s α) y · 𝓘" "x ∈ fv_transaction T" "y ∈ fv_transaction
T"
    for x y
  proof -
    have "x ∉ set (transaction_fresh T) ⟹ y ∉ set (transaction_fresh T) ⟹ ?thesis"
      using xy admissible_transaction_strand_sem_fv_ineq[OF T_adm 𝓘_is_T_model[unfolded T'_def]]
      by fast
    moreover {
      assume *: "x ∈ set (transaction_fresh T)" "y ∈ set (transaction_fresh T)"
      hence "Γv x = TAtom Value" "Γv y = TAtom Value"
        using T_fresh_vars_value_typed by (blast, blast)
      then obtain xn yn where "σ x = Fun (Val xn) []" "σ y = Fun (Val yn) []"
        using * transaction_fresh_subst_sends_to_val[OF σ] by meson
      hence "σ x = σ y" using that(1) ξ_empty by (simp add: subst_compose)
      moreover have "inj_on σ (subst_domain σ)" "x ∈ subst_domain σ" "y ∈ subst_domain σ"
        using * σ unfolding transaction_fresh_subst_def by auto
      ultimately have ?thesis unfolding inj_on_def by blast
    } moreover have False when "x ∈ set (transaction_fresh T)" "y ∉ set (transaction_fresh T)"
      using that(2) xy T_no_bvars admissible_transaction_Value_vars[OF bspec[OF P T], of y]
            transaction_prop4[OF 𝓐_reach T 𝓘[unfolded T'_def] ξ σ α P P_occ that(1), of y]
      by auto
    moreover have False when "x ∉ set (transaction_fresh T)" "y ∈ set (transaction_fresh T)"
      using that(1) xy T_no_bvars admissible_transaction_Value_vars[OF bspec[OF P T], of x]
            transaction_prop4[OF 𝓐_reach T 𝓘[unfolded T'_def] ξ σ α P P_occ that(2), of x]
      by fastforce
    ultimately show ?thesis by metis
  qed

  have 4: "∃y s. t = Var y ∧ u = Fun (Set s) []"
    when "insert⟨t,u⟩ ∈ set (unlabel (transaction_strand T))" for t u
    using that admissible_transaction_strand_step_cases(3)[OF T_adm] T_wf
    by blast

  have 5: "∃y s. t = Var y ∧ u = Fun (Set s) []"
    when "delete⟨t,u⟩ ∈ set (unlabel (transaction_strand T))" for t u
    using that admissible_transaction_strand_step_cases(3)[OF T_adm] T_wf
    by blast

  have 6: "∃n. (ξ ∘s σ ∘s α) y · 𝓘 = Fun (Val n) []" when "y ∈ fv_transaction T" for y
    using that by (simp add: T_vars_vals)

  have "list_all wellformed_transaction P" "list_all admissible_transaction_updates P"
    using P(1) Ball_set[of P admissible_transaction'] Ball_set[of P wellformed_transaction]
          Ball_set[of P admissible_transaction_updates]
          admissible_transaction_is_wellformed_transaction(1,3)
```

```
    by fastforce+
  hence 7: "∃s. snd d = Fun (Set s) []" when "d ∈ set (db_lsst 𝒜 ℐ)" for d
    using that reachable_constraints_db_lsst_set_args_empty[OF 𝒜_reach]
    unfolding admissible_transaction_updates_def by (cases d) simp

  have "(ξ ∘s σ ∘s α) x · ℐ ·α a0' = absc (upd δ x)"
    when x: "x ∈ fv_transaction T" "fst x = TAtom Value" for x
  proof -
    have "(ξ ∘s σ ∘s α) x · ℐ ·α α0 (db'_lsst (dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℐ (db_lsst
𝒜 ℐ))
           = absc (absdbupd (unlabel (transaction_strand T)) x (δ x))"
      using 2[of "ξ ∘s σ ∘s α" x "db_lsst 𝒜 ℐ" "δ x" "transaction_strand T"]
            3[OF _ x(1)] 4 5 6[OF that(1)] 6 7 x δ(2)
      unfolding all_defs by blast
    thus ?thesis
      using x db_sst_append[of "unlabel 𝒜"] absdbupd_wellformed_transaction[OF T_wf]
      unfolding all_defs db_sst_def by force
  qed
  thus ?thesis using δ Γv_TAtom''(2) unfolding all_defs by blast
qed

lemma transaction_prop6:
  fixes T ξ σ α 𝒜 ℐ T' a0 a0'
  defines "T' ≡ dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
    and "a0 ≡ α0 (db_lsst 𝒜 ℐ)"
    and "a0' ≡ α0 (db_lsst (𝒜@T') ℐ)"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@T')"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trms_lsst 𝒜)"
    and α: "transaction_renaming_subst α P (vars_lsst 𝒜)"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢c t"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
      "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction' T"
    and P_occ:
      "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and step: "list_all (transaction_check (FP, OCC, TI)) P"
  shows "∀t ∈ timpl_closure_set (α_ik 𝒜 ℐ) (α_ti 𝒜 T (ξ ∘s σ ∘s α) ℐ).
          timpl_closure_set (set FP) (set TI) ⊢c t" (is ?A)
    and "timpl_closure_set (α_vals 𝒜 ℐ) (α_ti 𝒜 T (ξ ∘s σ ∘s α) ℐ) ⊆ absc ` set OCC" (is ?B)
    and "∀t ∈ trms_lsst (transaction_send T). is_Fun (t · (ξ ∘s σ ∘s α) · ℐ ·α a0') ⟶
          timpl_closure_set (set FP) (set TI) ⊢c t · (ξ ∘s σ ∘s α) · ℐ ·α a0'" (is ?C)
    and "∀x ∈ fv_transaction T. Γv x = TAtom Value ⟶
          (ξ ∘s σ ∘s α) x · ℐ ·α a0' ∈ absc ` set OCC" (is ?D)
proof -
  define comp0 where
    "comp0 ≡ abs_substs_fun ` set (transaction_check_comp (λ_ _. True) (FP, OCC, TI) T)"
  define check0 where "check0 ≡ transaction_check (FP, OCC, TI) T"

  define upd where "upd ≡ λδ x. absdbupd (unlabel (transaction_updates T)) x (δ x)"

  define ϑ where "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
```

```
note T_adm = bspec[OF P T]
note T_occ = bspec[OF P_occ T]
note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

have ϑ_prop: "ϑ σ x = absc (σ x)" when "Γ_v x = TAtom Value" for σ x
  using that Γ_v_TAtom''(2)[of x] unfolding ϑ_def by simp

 — The set-membership status of all value constants in T under ℐ, σ, α are covered by the check
have 0: "∃δ ∈ comp0. ∀x ∈ fv_transaction T. Γ_v x = TAtom Value ⟶
          (ξ ∘_s σ ∘_s α) x · ℐ ·_α a0 = absc (δ x) ∧
          (ξ ∘_s σ ∘_s α) x · ℐ ·_α a0' = absc (upd δ x)"
  using transaction_prop5[OF 𝒜_reach T ℐ[unfolded T'_def] ξ σ α FP OCC TI P P_occ step]
  unfolding a0_def a0'_def T'_def upd_def comp0_def
  by blast

— All set-membership changes are covered by the term implication graph
have 1: "(δ x, upd δ x) ∈ (set TI)⁺"
  when "δ ∈ comp0" "δ x ≠ upd δ x" "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  for x δ
  using T that step Ball_set[of P "transaction_check (FP, OCC, TI)"]
        transaction_prop1[of δ "λ_ _. True" FP OCC TI T x] TI
  unfolding upd_def comp0_def transaction_check_def
  by blast

— All set-membership changes are covered by the fixed point
have 2:  "upd δ x ∈ set OCC"
  when "δ ∈ comp0" "x ∈ fv_transaction T" "fst x = TAtom Value" for x δ
  using T that step Ball_set[of P "transaction_check (FP, OCC, TI)"]
        T_adm T_occ FP OCC TI transaction_prop2[of δ "λ_ _. True" FP OCC TI T x]
  unfolding upd_def comp0_def transaction_check_def
  by blast

obtain δ where δ:
    "δ ∈ comp0"
    "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ⟶
      (ξ ∘_s σ ∘_s α) x · ℐ ·_α a0 = absc (δ x) ∧
      (ξ ∘_s σ ∘_s α) x · ℐ ·_α a0' = absc (upd δ x)"
  using 0 by force

have "∃x. ab = (δ x, upd δ x) ∧ x ∈ fv_transaction T - set (transaction_fresh T) ∧ δ x ≠ upd δ x"
  when ab: "ab ∈ α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ" for ab
proof -
  obtain a b where ab': "ab = (a,b)" by (metis surj_pair)
  then obtain x where x:
      "a ≠ b" "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
      "absc a = (ξ ∘_s σ ∘_s α) x · ℐ ·_α a0" "absc b = (ξ ∘_s σ ∘_s α) x · ℐ ·_α a0'"
    using ab unfolding abs_term_implications_def a0_def a0'_def T'_def by blast
  hence "absc a = absc (δ x)" "absc b = absc (upd δ x)"
    using δ(2) admissible_transaction_Value_vars[OF bspec[OF P T] x(2)]
    by metis+
  thus ?thesis using x ab' by blast
qed
hence α_ti_TI_subset: "α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ ⊆ {(a,b) ∈ (set TI)⁺. a ≠ b}" using 1[OF δ(1)] by
blast

have "timpl_closure_set (timpl_closure_set (set FP) (set TI)) (α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ) ⊢_c t"
  when t: "t ∈ timpl_closure_set (α_ik 𝒜 ℐ) (α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ)" for t
  using timpl_closure_set_is_timpl_closure_union[of "α_ik 𝒜 ℐ" "α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ"]
        intruder_synth_timpl_closure_set FP(3) t
  by blast
thus ?A
  using ideduct_synth_mono[OF _ timpl_closure_set_mono[OF
          subset_refl[of "timpl_closure_set (set FP) (set TI)"]
```

```
          αti_TI_subset]]
        timpl_closure_set_timpls_trancl_eq'[of "timpl_closure_set (set FP) (set TI)" "set TI"]
    unfolding timpl_closure_set_idem
    by force

  have "timpl_closure_set (αvals 𝒜 ℐ) (αti 𝒜 T (ξ ∘s σ ∘s α) ℐ) ⊆
        timpl_closure_set (absc ` set OCC) {(a,b) ∈ (set TI)⁺. a ≠ b}"
    using timpl_closure_set_mono[OF _ αti_TI_subset] OCC(3) by blast
  thus ?B using OCC(2) timpl_closure_set_timpls_trancl_subset' by blast

  have "transaction_check_post (FP, OCC, TI) T δ"
    using T δ(1) step
    unfolding transaction_check_def transaction_check'_def comp0_def list_all_iff
    by fastforce
  hence 3: "timpl_closure_set (set FP) (set TI) ⊢c t · ϑ (upd δ)"
    when "t ∈ trmslsst (transaction_send T)" "is_Fun (t · ϑ (upd δ))" for t
    using that
    unfolding transaction_check_post_def upd_def ϑ_def
              intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI, symmetric]
    by fastforce

  have 4: "∀x ∈ fv t. (ξ ∘s σ ∘s α ∘s ℐ) x ·α a0' = ϑ (upd δ) x"
    when "t ∈ trmslsst (transaction_send T)" for t
    using wellformed_transaction_send_receive_fv_subset(2)[OF T_wf that]
          δ(2) subst_compose[of "ξ ∘s σ ∘s α" ℐ] ϑ_prop
          admissible_transaction_Value_vars[OF bspec[OF P T]]
    by fastforce

  have 5: "∄n T. Fun (Val n) T ∈ subterms t" when "t ∈ trmslsst (transaction_send T)" for t
    using that admissible_transactions_no_Value_consts'[OF T_adm] trms_transaction_unfold[of T]
    by blast

  show ?D using 2[OF δ(1)] δ(2) Γv_TAtom''(2) unfolding a0'_def T'_def by blast

  show ?C using 3 abs_term_subst_eq'[OF 4 5] by simp
qed

lemma reachable_constraints_covered_step:
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trmslsst 𝒜)"
    and α: "transaction_renaming_subst α P (varslsst 𝒜)"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wftrms (set FP)"
      "∀t ∈ αik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢c t"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
      "αvals 𝒜 ℐ ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction' T"
    and P_occ:
      "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and transactions_covered: "list_all (transaction_check (FP, OCC, TI)) P"
  shows "∀t ∈ αik (𝒜@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℐ.
          timpl_closure_set (set FP) (set TI) ⊢c t" (is ?A)
```

```
    and "αvals (𝒜@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℐ ⊆ absc ` set OCC" (is ?B)
proof -
  note step_props =
    transaction_prop6[OF 𝒜_reach T ℐ ξ σ α FP(1,2,3) OCC TI P P_occ transactions_covered]

  define T' where "T' ≡ duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
  define a0 where "a0 ≡ α0 (dblsst 𝒜 ℐ)"
  define a0' where "a0' ≡ α0 (dblsst (𝒜@T') ℐ)"

  define vals where "vals ≡ λS::('fun,'atom,'sets,'lbl) prot_constr.
      {t ∈ subtermsset (trmslsst S) ·set ℐ. ∃n. t = Fun (Val n) []}"

  define vals_sym where "vals_sym ≡ λS::('fun,'atom,'sets,'lbl) prot_constr.
      {t ∈ subtermsset (trmslsst S). (∃n. t = Fun (Val n) []) ∨ (∃m. t = Var (TAtom Value,m))}"

  have ℐ_wt: "wtsubst ℐ" by (metis ℐ welltyped_constraint_model_def)

  have ℐ_grounds: "fv (t · ℐ) = {}" for t
    using ℐ interpretation_grounds[of ℐ]
    unfolding welltyped_constraint_model_def constraint_model_def by auto

  have wt_σαℐ: "wtsubst (ξ ∘s σ ∘s α ∘s ℐ)" and wt_σα: "wtsubst (ξ ∘s σ ∘s α)"
    using ℐ_wt wt_subst_compose transaction_decl_fresh_renaming_substs_wt[OF ξ σ α]
    by (blast, blast)

  have "∀T∈set P. bvars_transaction T = {}"
    using P admissible_transactionE(4) by metis
  hence 𝒜_no_bvars: "bvarslsst 𝒜 = {}"
    using reachable_constraints_no_bvars[OF 𝒜_reach] by metis

  have ℐ_vals: "∃n. ℐ (TAtom Value, m) = Fun (Val n) []"
    when "(TAtom Value, m) ∈ fvlsst 𝒜" for m
    using constraint_model_Value_term_is_Val'[
            OF 𝒜_reach welltyped_constraint_model_prefix[OF ℐ] P P_occ]
          𝒜_no_bvars varssst_is_fvsst_bvarssst[of "unlabel 𝒜"] that
    by blast

  have vals_sym_vals: "t · ℐ ∈ vals 𝒜" when t: "t ∈ vals_sym 𝒜" for t
  proof (cases t)
    case (Var x)
    then obtain m where *: "x = (TAtom Value,m)" using t unfolding vals_sym_def by blast
    moreover have "t ∈ subtermsset (trmslsst 𝒜)" using t unfolding vals_sym_def by blast
    hence "t · ℐ ∈ subtermsset (trmslsst 𝒜) ·set ℐ" "∃n. ℐ (Var Value, m) = Fun (Val n) []"
      using Var * ℐ_vals[of m] var_subterm_trmssst_is_varssst[of x "unlabel 𝒜"]
            Γv_TAtom[of Value m] reachable_constraints_Value_vars_are_fv[OF 𝒜_reach P(1), of x]
      by blast+
    ultimately show ?thesis using Var unfolding vals_def by auto
  next
    case (Fun f T)
    then obtain n where "f = Val n" "T = []" using t unfolding vals_sym_def by blast
    moreover have "t ∈ subtermsset (trmslsst 𝒜)" using t unfolding vals_sym_def by blast
    hence "t · ℐ ∈ subtermsset (trmslsst 𝒜) ·set ℐ" using Fun by blast
    ultimately show ?thesis using Fun unfolding vals_def by auto
  qed

  have vals_vals_sym: "∃s. s ∈ vals_sym 𝒜 ∧ t = s · ℐ" when "t ∈ vals 𝒜" for t
    using that constraint_model_Val_is_Value_term[OF ℐ]
    unfolding vals_def vals_sym_def by fast

  note T_adm = bspec[OF P T]
  note T_wf = admissible_transaction_is_wellformed_transaction(1)[OF T_adm]

  have 0:
```

```
      "αik (𝒜@T') ℐ = (iklsst 𝒜 ·set ℐ) ·αset a0' ∪ (iklsst T' ·set ℐ) ·αset a0'"
      "αvals (𝒜@T') ℐ = vals 𝒜 ·αset a0' ∪ vals T' ·αset a0'"
    by (metis abs_intruder_knowledge_append a0'_def,
        metis abs_value_constants_append[of 𝒜 T' ℐ] a0'_def vals_def)

  have 1: "(iklsst T' ·set ℐ) ·αset a0' =
           (trmslsst (transaction_send T) ·set (ξ ∘s σ ∘s α) ·set ℐ) ·αset a0'"
    by (metis T'_def dual_transaction_ik_is_transaction_send''[OF T_wf])

  have 2: "bvarslsst (transaction_strand T) ∩ subst_domain ξ = {}"
          "bvarslsst (transaction_strand T) ∩ subst_domain σ = {}"
          "bvarslsst (transaction_strand T) ∩ subst_domain α = {}"
    using admissible_transactionE(4)[OF T_adm] by blast+

  have "vals T' ⊆ (ξ ∘s σ ∘s α) ` fv_transaction T ·set ℐ"
  proof
    fix t assume "t ∈ vals T'"
    then obtain s n where s:
        "s ∈ subtermsset (trmslsst T')" "t = s · ℐ" "t = Fun (Val n) []"
      unfolding vals_def by fast
    then obtain u where u:
        "u ∈ subtermsset (trmslsst (transaction_strand T))"
        "s = u · (ξ ∘s σ ∘s α)"
      using transaction_decl_fresh_renaming_substs_trms[OF ξ σ α 2]
            trmssst_unlabel_duallsst_eq[of "transaction_strand T ·lsst ξ ∘s σ ∘s α"]
      unfolding T'_def by blast

    have *: "t = u · (ξ ∘s σ ∘s α ∘s ℐ)" using s(2) u(2) subst_subst_compose by simp
    then obtain x where x: "u = Var x"
      using s(3) admissible_transactions_no_Value_consts(1)[OF T_adm u(1)] by (cases u) force+
    hence **: "x ∈ vars_transaction T"
      by (metis u(1) var_subterm_trmssst_is_varssst)

    have "Γv x = TAtom Value"
      using * x s(3) wt_subst_trm''[OF wt_σαℐ, of u]
      by simp
    thus "t ∈ (ξ ∘s σ ∘s α) ` fv_transaction T ·set ℐ"
      using admissible_transaction_Value_vars_are_fv[OF T_adm **] x *
            eval_term.simps(1)[of _ x "ξ ∘s σ ∘s α ∘s ℐ"]
            subst_comp_set_image[of "ξ ∘s σ ∘s α" ℐ "fv_transaction T"]
      by blast
  qed
  hence 3: "vals T' ·αset a0' ⊆ ((ξ ∘s σ ∘s α) ` fv_transaction T ·set ℐ) ·αset a0'"
    by (simp add: abs_apply_terms_def image_mono)

  have "t · ℐ ·α a0' ∈ timpl_closure_set (αik 𝒜 ℐ) (αti 𝒜 T (ξ ∘s σ ∘s α) ℐ)"
    when "t ∈ iklsst 𝒜" for t
    using that abs_in[OF imageI[OF that]]
          αti_covers_α0_ik[OF 𝒜_reach T ℐ ξ σ α P P_occ]
          timpl_closure_set_mono[of "{t · ℐ ·α a0}" "αik 𝒜 ℐ" "αti 𝒜 T (ξ ∘s σ ∘s α) ℐ"
                                   "αti 𝒜 T (ξ ∘s σ ∘s α) ℐ"]
    unfolding a0_def a0'_def T'_def abs_intruder_knowledge_def by fast
  hence A: "αik (𝒜@T') ℐ ⊆
              timpl_closure_set (αik 𝒜 ℐ) (αti 𝒜 T (ξ ∘s σ ∘s α) ℐ) ∪
              (trmslsst (transaction_send T) ·set (ξ ∘s σ ∘s α) ·set ℐ) ·αset a0'"
    using 0(1) 1 by (auto simp add: abs_apply_terms_def)

  have "t · ℐ ·α a0' ∈ timpl_closure_set {t · ℐ ·α a0} (αti 𝒜 T (ξ ∘s σ ∘s α) ℐ)"
    when t: "t ∈ vals_sym 𝒜" for t
  proof -
    have "(∃n. t = Fun (Val n) [] ∧ t ∈ subtermsset (trmslsst 𝒜)) ∨
          (∃n. t = Var (TAtom Value,n) ∧ (TAtom Value,n) ∈ fvlsst 𝒜)"
      (is "?P ∨ ?Q")
```

```
      using t var_subterm_trms_sst_is_vars_sst[of _ "unlabel 𝒜"]
            Γ_v_TAtom[of Value] reachable_constraints_Value_vars_are_fv[OF 𝒜_reach P(1)]
      unfolding vals_sym_def by fast
    thus ?thesis
    proof
      assume ?P
      then obtain n where n: "t = Fun (Val n) []" "t ∈ subterms_set (trms_lsst 𝒜)" by force
      thus ?thesis
        using α_ti_covers_α_0_Val[OF 𝒜_reach T ℐ ξ σ α P P_occ, of n]
        unfolding a0_def a0'_def T'_def by fastforce
    next
      assume ?Q
      thus ?thesis
        using α_ti_covers_α_0_Var[OF 𝒜_reach T ℐ ξ σ α P P_occ]
        unfolding a0_def a0'_def T'_def by fastforce
    qed
  qed
  moreover have "t · ℐ ·_α a0 ∈ α_vals 𝒜 ℐ"
    when "t ∈ vals_sym 𝒜" for t
    using that abs_in vals_sym_vals
    unfolding a0_def abs_value_constants_def vals_sym_def vals_def
    by (metis (mono_tags, lifting))
  ultimately have "t · ℐ ·_α a0' ∈ timpl_closure_set (α_vals 𝒜 ℐ) (α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ)"
    when t: "t ∈ vals_sym 𝒜" for t
    using t timpl_closure_set_mono[of "{t · ℐ ·_α a0}" "α_vals 𝒜 ℐ" "α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ"
                                    "α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ"]
    by blast
  hence "t ·_α a0' ∈ timpl_closure_set (α_vals 𝒜 ℐ) (α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ)"
    when t: "t ∈ vals 𝒜" for t
    using vals_vals_sym[OF t] by blast
  hence B: "α_vals (𝒜@T') ℐ ⊆
            timpl_closure_set (α_vals 𝒜 ℐ) (α_ti 𝒜 T (ξ ∘_s σ ∘_s α) ℐ) ∪
            ((ξ ∘_s σ ∘_s α) ` fv_transaction T ·_set ℐ) ·_αset a0'"
    using 0(2) 3
    by (simp add: abs_apply_terms_def image_subset_iff)

  have 4: "fv (t · ξ ∘_s σ ∘_s α · ℐ ·_α a) = {}" for t a
    using ℐ_grounds[of "t · ξ ∘_s σ ∘_s α"] abs_fv[of "t · ξ ∘_s σ ∘_s α · ℐ" a]
    by argo

  have "is_Fun (t · ξ ∘_s σ ∘_s α · ℐ ·_α a0')" for t
    using 4[of t a0'] by force
  thus ?A
    using A step_props(1,3)
    unfolding T'_def a0_def a0'_def abs_apply_terms_def
    by blast

  show ?B
    using B step_props(2,4) admissible_transaction_Value_vars[OF bspec[OF P T]]
    by (auto simp add: T'_def a0_def a0'_def abs_apply_terms_def)
qed

lemma reachable_constraints_covered:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
    and TI:
```

```
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction' T"
    and P_occ:
      "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and transactions_covered: "list_all (transaction_check (FP, OCC, TI)) P"
  shows "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢c t"
    and "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
using 𝒜_reach ℐ
proof (induction rule: reachable_constraints.induct)
  case init
  { case 1 show ?case by (simp add: abs_intruder_knowledge_def) }
  { case 2 show ?case by (simp add: abs_value_constants_def) }
next
  case (step 𝒜 T ξ σ α)
  { case 1
    hence "welltyped_constraint_model ℐ 𝒜"
      by (metis welltyped_constraint_model_prefix)
    hence IH: "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢c t"
              "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
      using step.IH by metis+
    show ?case
      using reachable_constraints_covered_step[
              OF step.hyps(1,2) "1.prems" step.hyps(3-5) FP(1,2) IH(1)
                 FP(3) OCC IH(2) TI P P_occ transactions_covered]
      by metis
  }
  { case 2
    hence "welltyped_constraint_model ℐ 𝒜"
      by (metis welltyped_constraint_model_prefix)
    hence IH: "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢c t"
              "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
      using step.IH by metis+
    show ?case
      using reachable_constraints_covered_step[
              OF step.hyps(1,2) "2.prems" step.hyps(3-5) FP(1,2) IH(1)
                 FP(3) OCC IH(2) TI P P_occ transactions_covered]
      by metis
  }
qed

lemma attack_in_fixpoint_if_attack_in_ik:
  fixes FP::"('fun,'atom,'sets,'lbl) prot_terms"
  assumes "∀t ∈ IK ·α_set a. FP ⊢c t"
    and "attack⟨n⟩ ∈ IK"
  shows "attack⟨n⟩ ∈ FP"
proof -
  have "attack⟨n⟩ ·α a ∈ IK ·α_set a" by (rule abs_in[OF assms(2)])
  hence "FP ⊢c attack⟨n⟩ ·α a" using assms(1) by blast
  moreover have "attack⟨n⟩ ·α a = attack⟨n⟩" by simp
  ultimately have "FP ⊢c attack⟨n⟩" by metis
  thus ?thesis using ideduct_synth_priv_const_in_ik[of FP "Attack n"] by simp
qed

lemma attack_in_fixpoint_if_attack_in_timpl_closure_set:
  fixes FP::"('fun,'atom,'sets,'lbl) prot_terms"
  assumes "attack⟨n⟩ ∈ timpl_closure_set FP TI"
  shows "attack⟨n⟩ ∈ FP"
proof -
  have "∀f ∈ funs_term (attack⟨n⟩). ¬is_Abs f" by auto
  thus ?thesis using timpl_closure_set_no_Abs_in_set[OF assms] by blast
qed
```

```
theorem prot_secure_if_fixpoint_covered_typed:
  assumes FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
      "has_initial_value_producing_transaction P"
    and transactions_covered: "list_all (transaction_check (FP, OCC, TI)) (map add_occurs_msgs P)"
    and attack_notin_FP: "attack⟨n⟩ ∉ set FP"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "∄ℐ. welltyped_constraint_model ℐ (𝒜@[(1, send⟨[attack⟨n⟩]⟩)])" (is "∄ℐ. ?Q ℐ")
proof
  assume "∃ℐ. ?Q ℐ"
  then obtain ℐ ℬ where
      ℬ: "ℬ ∈ reachable_constraints (map add_occurs_msgs P)"
    and ℐ: "welltyped_constraint_model ℐ (ℬ@[(1, send⟨[attack⟨n⟩]⟩)])"
    using add_occurs_msgs_soundness[OF P 𝒜]
    unfolding list_all_iff by blast

  have P':
      "∀T ∈ set (map add_occurs_msgs P). admissible_transaction' T"
      "∀T ∈ set (map add_occurs_msgs P). admissible_transaction_occurs_checks T"
    using P add_occurs_msgs_admissible_occurs_checks[OF admissible_transactionE'(1)] by auto

  have 0: "attack⟨n⟩ ∉ ik_lsst ℬ ·_set ℐ"
    using welltyped_constraint_model_prefix[OF ℐ]
          reachable_constraints_covered(1)[OF ℬ _ FP OCC TI P' transactions_covered]
          attack_in_fixpoint_if_attack_in_ik[
            of "ik_lsst ℬ ·_set ℐ" "α₀ (db_lsst ℬ ℐ)" "timpl_closure_set (set FP) (set TI)" n]
          attack_in_fixpoint_if_attack_in_timpl_closure_set
          attack_notin_FP
    unfolding abs_intruder_knowledge_def by blast

  have 1: "ik_lsst ℬ ·_set ℐ ⊢ attack⟨n⟩"
    using ℐ strand_sem_append_stateful[of "{}" "{}" "unlabel ℬ" _ ℐ]
    unfolding welltyped_constraint_model_def constraint_model_def by force

  show False
    using 0 private_const_deduct[OF _ 1]
          reachable_constraints_receive_attack_if_attack'(1)[
            OF ℬ P'(1) welltyped_constraint_model_prefix[OF ℐ] 1]
    by simp
qed

end
```

3.6.5 Theorem: A Protocol is Secure if it is Covered by a Fixed-Point

```
context stateful_protocol_model
begin

theorem prot_secure_if_fixpoint_covered:
  fixes P
  assumes FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "ground (set FP)"
```

```
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and M:
      "has_all_wt_instances_of Γ (⋃T ∈ set P. trms_transaction T) N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. admissible_transaction T"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
      "has_initial_value_producing_transaction P"
    and transactions_covered: "list_all (transaction_check (FP, OCC, TI)) (map add_occurs_msgs P)"
    and attack_notin_FP: "attack⟨n⟩ ∉ set FP"
    and A: "𝒜 ∈ reachable_constraints P"
  shows "∄ℐ. constraint_model ℐ (𝒜@[(1, send⟨[attack⟨n⟩]⟩)])"
    (is "∄ℐ. constraint_model ℐ ?A")
proof
  assume "∃ℐ. constraint_model ℐ ?A"
  then obtain ℐ where "constraint_model ℐ ?A" by force
  then obtain ℐ_τ where I: "welltyped_constraint_model ℐ_τ ?A"
    using reachable_constraints_typing_result[OF M P(1,2) A] by blast

  note a = FP OCC TI P(1,3) transactions_covered attack_notin_FP A

  show False
    using prot_secure_if_fixpoint_covered_typed[OF a] I
    by force
qed

end
```

3.6.6 Alternative Protocol-Coverage Check

```
context stateful_protocol_model
begin

context
begin

private lemma transaction_check_variant_soundness_aux0:
  assumes S: "S ≡ unlabel (transaction_strand T)"
    and xs: "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S)"
    and x: "fst x = Var Value" "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "x ∈ set xs"
using x fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"]
unfolding xs S by auto

private lemma transaction_check_variant_soundness_aux1:
  fixes T FP S C xs OCC negs poss as
  assumes C: "C ≡ unlabel (transaction_checks T)"
    and S: "S ≡ unlabel (transaction_strand T)"
    and xs: "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S)"
    and poss: "poss ≡ transaction_poschecks_comp C"
    and negs: "negs ≡ transaction_negchecks_comp C"
    and as: "as ≡ map (λx. (x, set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC))) xs"
    and f: "f ≡ λx. case List.find (λp. fst p = x) as of Some p ⇒ snd p | None ⇒ {}"
    and x: "x ∈ set xs"
  shows "f x = set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC)"
proof -
  define g where "g ≡ λx. set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC)"
```

```
  define gs where "gs ≡ map (λx. (x, g x)) xs"

  have 1: "(x, g x) ∈ set gs" using x unfolding gs_def by simp

  have 2: "distinct xs" unfolding xs fv_list_sst_def by auto

  have "∃i < length xs. xs ! i = x ∧ (∀j < i. xs ! j ≠ x)" when x: "x ∈ set xs" for x
  proof (rule ex1E[OF distinct_Ex1[OF 2 x]])
    fix i assume i: "i < length xs ∧ xs ! i = x" and "∀j. j < length xs ∧ xs ! j = x ⟶ j = i"
    hence "∀j < length xs. xs ! j = x ⟶ j = i" by blast
    hence "∀j < i. xs ! j = x ⟶ j = i" using i by auto
    hence "∀j < i. xs ! j ≠ x" by blast
    thus ?thesis using i by blast
  qed
  hence "∃i < length gs. gs ! i = (x, g x) ∧ (∀j < i. gs ! j ≠ (x, g x))"
    using 1 unfolding gs_def by fastforce
  hence "∃i < length gs. fst (gs ! i) = x ∧ (x, g x) = gs ! i ∧ (∀j < i. fst (gs ! j) ≠ x)"
    using nth_map[of _ xs "λx. (x, g x)"] length_map[of "λx. (x, g x)" xs]
    unfolding gs_def by (metis (no_types, lifting) fstI min.strict_order_iff min_less_iff_conj)
  hence "List.find (λp. fst p = x) gs = Some (x, g x)"
    using find_Some_iff[of "λp. fst p = x" gs "(x, g x)"] by blast
  thus ?thesis
    unfolding f as gs_def g_def by force
qed

private lemma transaction_check_variant_soundness_aux2:
  fixes T FP S C xs OCC negs poss as
  assumes C: "C ≡ unlabel (transaction_checks T)"
    and S: "S ≡ unlabel (transaction_strand T)"
    and xs: "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S)"
    and poss: "poss ≡ transaction_poschecks_comp C"
    and negs: "negs ≡ transaction_negchecks_comp C"
    and as: "as ≡ map (λx. (x, set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC))) xs"
    and f: "f ≡ λx. case List.find (λp. fst p = x) as of Some p ⇒ snd p | None ⇒ {}"
    and x: "x ∉ set xs"
  shows "f x = {}"
proof -
  define g where "g ≡ λx. set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC)"
  define gs where "gs ≡ map (λx. (x, g x)) xs"

  have "(x, y) ∉ set gs" for y using x unfolding gs_def by force
  thus ?thesis
    using find_None_iff[of "λp. fst p = x" gs]
    unfolding f as gs_def g_def by fastforce
qed

private lemma synth_abs_substs_constrs_rel_if_synth_abs_substs_constrs:
  fixes T OCC negs poss
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "ts ≡ trms_list_sst (unlabel (transaction_receive T))"
  assumes ts_wf: "∀t ∈ set ts. wf_trm t"
    and FP_ground: "ground (set FP)"
    and FP_wf: "wf_trms (set FP)"
  shows "synth_abs_substs_constrs_rel FP OCC TI ts (synth_abs_substs_constrs (FP,OCC,TI) T)"
proof -
  let ?R = "synth_abs_substs_constrs_rel FP OCC TI"
  let ?D = "synth_abs_substs_constrs_aux FP OCC TI"

  have *: "?R [t] (?D t)" when "wf_trm t" for t using that
  proof (induction t)
    case (Var x) thus ?case
      using synth_abs_substs_constrs_rel.SolveValueVar[of "?D (Var x)" OCC x TI FP]
      by fastforce
```

```
next
  case (Fun f ss)
  let ?xs = "fv (Fun f ss)"
  let ?lst = "map (match_abss OCC TI (Fun f ss)) FP"

  define flt where
    "flt = (λδ::(('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set set) option. δ ≠ None)"
  define Δ where "Δ = map the (filter flt (map (match_abss OCC TI (Fun f ss)) FP))"
  define ϑ1 where "ϑ1 = fun_point_Union (set Δ)"
  define ϑ2 where "ϑ2 = fun_point_Inter (?D ` set ss)"

  have f: "arity f = length ss" using wf_trm_arity[OF Fun.prems] by simp

  have IH: "?R [s] (?D s)" when s: "s ∈ set ss" for s
    using Fun.IH[OF s wf_trm_subterm[OF Fun.prems Fun_param_is_subterm[OF s]]] s
    by force

  have Δ3: "∀δ. δ ∈ set Δ ⟷ (∃s ∈ set FP. match_abss OCC TI (Fun f ss) s = Some δ)"
      (is "∀δ. δ ∈ set Δ ⟷ ?P δ")
  proof (intro allI iffI)
    fix δ assume "δ ∈ set Δ"
    then obtain u where "u ∈ set FP" "match_abss OCC TI (Fun f ss) u = Some δ"
      unfolding Δ_def flt_def by fastforce
    thus "?P δ" by blast
  next
    fix δ assume "?P δ"
    then obtain u where u: "u ∈ set FP" "match_abss OCC TI (Fun f ss) u = Some δ" by blast

    have "Some δ ∈ set ?lst" using u unfolding flt_def by force
    hence "Some δ ∈ set (filter flt ?lst)" unfolding flt_def by force
    moreover have "∃ϑ. d = Some ϑ" when d: "d ∈ set (filter flt ?lst)" for d
      using d unfolding flt_def by simp
    ultimately have "δ ∈ set (map the (filter flt ?lst))" by force
    thus "δ ∈ set Δ" unfolding Δ_def by blast
  qed

  show ?case
  proof (cases "ss = []")
    case True
    note ss = this
    show ?thesis
    proof (cases "¬public f ∧ Fun f ss ∉ set FP")
      case True thus ?thesis
        using synth_abs_substs_constrs_rel.SolvePrivConstNotin[of f FP OCC TI]
        unfolding ss by force
    next
      case False thus ?thesis
        using f synth_abs_substs_constrs_rel.SolvePubConst[of f FP OCC TI]
              synth_abs_substs_constrs_rel.SolvePrivConstIn[of f FP OCC TI]
        unfolding ss by auto
    qed
  next
    case False
    note ss = this
    hence f': "arity f > 0" using f by auto
    have IH': "?R ss (fun_point_Inter (?D ` set ss))"
      using IH synth_abs_substs_constrs_rel.SolveCons[OF ss, of FP OCC TI ?D] by blast

    have "?D (Fun f ss) = (
            fun_point_union (fun_point_Union_list Δ) (fun_point_Inter_list (map ?D ss)))"
      using synth_abs_substs_constrs_aux_fun_case[OF ss, of FP OCC TI f]
      unfolding Let_def Δ_def flt_def by argo
    hence "?D (Fun f ss) = fun_point_union ϑ1 ϑ2"
```

```
        using fun_point_Inter_set_eq[of "map ?D ss"] fun_point_Union_set_eq[of Δ]
        unfolding ϑ1_def ϑ2_def by simp
      thus ?thesis
        using synth_abs_substs_constrs_rel.SolvePubComposed[
                OF f' no_private_funs[OF f'] f[symmetric] Δ3 ϑ1_def IH']
        unfolding ϑ2_def by argo
    qed
  qed

  note l0 = synth_abs_substs_constrs_rel.SolveNil[of FP OCC TI]
  note d0 = synth_abs_substs_constrs_def ts_def

  note l1 = * ts_wf synth_abs_substs_constrs_rel.SolveCons[of ts FP OCC TI ?D]
  note d1 = d0 Let_def fun_point_Inter_set_eq[symmetric] fun_point_Inter_def

  show ?thesis
  proof (cases "ts = []")
    case True thus ?thesis using l0 unfolding d0 by simp
  next
    case False thus ?thesis using l1 unfolding d1 by auto
  qed
qed

private function (sequential) match_abss'_timpls_transform
::"('c set × 'c set) list ⇒
   ('a,'b,'c,'d) prot_subst ⇒
   ('a,'b,'c,'d) prot_term ⇒
   ('a,'b,'c,'d) prot_term ⇒
   (('a,'b,'c,'d) prot_var ⇒ 'c set set) option"
where
  "match_abss'_timpls_transform TI δ (Var x) (Fun (Abs a) _) = (
    if ∃b ts. δ x = Fun (Abs b) ts ∧ (a = b ∨ (a,b) ∈ set TI)
    then Some ((λ_. {})(x := {a}))
    else None)"
| "match_abss'_timpls_transform TI δ (Fun f ts) (Fun g ss) = (
    if f = g ∧ length ts = length ss
    then map_option fun_point_Union_list (those (map2 (match_abss'_timpls_transform TI δ) ts ss))
    else None)"
| "match_abss'_timpls_transform _ _ _ _ = None"
by pat_completeness auto
termination
proof -
  let ?m = "measures [size ∘ fst ∘ snd ∘ snd]"

  have 0: "wf ?m" by simp

  show ?thesis
    apply (standard, use 0 in fast)
    by (metis (no_types) comp_def fst_conv snd_conv measures_less Fun_zip_size_lt(1))
qed

private lemma match_abss'_timpls_transform_Var_inv:
  assumes "match_abss'_timpls_transform TI δ (Var x) (Fun (Abs a) ts) = Some σ"
  shows "∃b ts. δ x = Fun (Abs b) ts ∧ (a = b ∨ (a, b) ∈ set TI)"
   and "σ = ((λ_. {})(x := {a}))"
using assms match_abss'_timpls_transform.simps(1)[of TI δ x a ts]
by (metis option.distinct(1), metis option.distinct(1) option.inject)

private lemma match_abss'_timpls_transform_Fun_inv:
  assumes "match_abss'_timpls_transform TI δ (Fun f ts) (Fun g ss) = Some σ"
  shows "f = g" (is ?A)
    and "length ts = length ss" (is ?B)
    and "∃ϑ. Some ϑ = those (map2 (match_abss'_timpls_transform TI δ) ts ss) ∧ σ =
```

```
fun_point_Union_list ϑ" (is ?C)
    and "∀(t,s) ∈ set (zip ts ss). ∃σ'. match_abss'_timpls_transform TI δ t s = Some σ'" (is ?D)
proof -
  note 0 = assms match_abss'_timpls_transform.simps(2)[of TI δ f ts g ss] option.distinct(1)
  show ?A by (metis 0)
  show ?B by (metis 0)
  show ?C by (metis (no_types, opaque_lifting) 0 map_option_eq_Some)
  thus ?D using map2_those_Some_case[of "match_abss'_timpls_transform TI δ" ts ss] by fastforce
qed

private lemma match_abss'_timpl_transform_nonempty_is_fv:
  assumes "match_abss'_timpls_transform TI δ s t = Some σ"
    and "σ x ≠ {}"
  shows "x ∈ fv s"
using assms
proof (induction s t arbitrary: TI δ σ rule: match_abss'_timpls_transform.induct)
  case (1 TI δ y a ts) show ?case
    using match_abss'_timpls_transform_Var_inv[OF "1.prems"(1)] "1.prems"(2)
    by fastforce
next
  case (2 TI δ f ts g ss)
  note prems = "2.prems"
  note IH = "2.IH"

  obtain ϑ where ϑ:
          "Some ϑ = those (map2 (match_abss'_timpls_transform TI δ) ts ss)"
          "σ = fun_point_Union_list ϑ"
      and fg: "f = g" "length ts = length ss"
    using match_abss'_timpls_transform_Fun_inv[OF prems(1)] by fast

  have "∃σ ∈ set ϑ. σ x ≠ {}"
    using fg(2) prems ϑ unfolding fun_point_Union_list_def by auto
  then obtain t' s' σ where ts':
      "(t',s') ∈ set (zip ts ss)" "match_abss'_timpls_transform TI δ t' s' = Some σ" "σ x ≠ {}"
    using those_map2_SomeD[OF ϑ(1)[symmetric]] by blast

  show ?case
    using ts'(3) IH[OF conjI[OF fg] ts'(1) _ ts'(2)] set_zip_leftD[OF ts'(1)] by force
qed auto

private lemma match_abss'_timpls_transformI:
  fixes s t::"('a,'b,'c,'d) prot_term"
    and δ::"('a,'b,'c,'d) prot_subst"
    and σ::"('a,'b,'c,'d) prot_var ⇒ 'c set set"
  assumes TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and δ: "timpls_transformable_to TI t (s · δ)"
    and σ: "match_abss' s t = Some σ"
    and t: "fv t = {}"
    and s: "∀f ∈ funs_term s. ¬is_Abs f"
           "∀x ∈ fv s. ∃a. δ x = ⟨a⟩_abs"
  shows "match_abss'_timpls_transform TI δ s t = Some σ"
using δ σ t s
proof (induction t arbitrary: s δ σ)
  case (Fun f ts)
  note prems = Fun.prems
  note IH = Fun.IH
  show ?case
  proof (cases s)
    case (Var x)
    obtain a b where a: "f = Abs a" "σ = (λ_. {})(x := {a})" and b: "δ x = ⟨b⟩_abs"
      using match_abss'_Var_inv[OF prems(2)[unfolded Var]] prems(5)[unfolded Var]
      by auto
    thus ?thesis
```

```
      using prems(1) timpls_transformable_to_inv(3)[of TI f ts "Abs b" "[]"]
      unfolding Var by auto
  next
    case (Fun g ss)
    note 0 = timpls_transformable_to_inv[OF prems(1)[unfolded Fun eval_term.simps(2)]]
    note 1 = match_abss'_Fun_inv[OF prems(2)[unfolded Fun]]

    obtain ϑ where ϑ: "those (map2 match_abss' ss ts) = Some ϑ" "σ = fun_point_Union_list ϑ"
      using 1(3) by force

    have "timpls_transformable_to TI t' (s' · δ)" "∃σ'. match_abss' s' t' = Some σ'"
      when "(t',s') ∈ set (zip ts ss)" for s' t'
      by (metis 0(2) nth_map[of _ ss] zip_arg_index[OF that],
          use that 1(4) in_set_zip_swap[of t' s' ts ss] in fast)
    hence IH': "match_abss'_timpls_transform TI δ s' t' = Some σ'"
      when "(t',s') ∈ set (zip ts ss)" "match_abss' s' t' = Some σ'" for s' t' σ'
      using that IH[of t' s' δ σ'] prems(3-) unfolding Fun
      by (metis (no_types, lifting) set_zip_leftD set_zip_rightD subsetI subset_empty
term.set_intros(2) term.set_intros(4))

    have "those (map2 (match_abss'_timpls_transform TI δ) ss ts) = Some ϑ"
      using IH' ϑ(1) 1(4) in_set_zip_swap[of _ _ ss ts]
            those_Some_iff[of "map2 match_abss' ss ts" ϑ]
            those_Some_iff[of "map2 (match_abss'_timpls_transform TI δ) ss ts" ϑ]
      by auto
    thus ?thesis using ϑ(2) 1(1,2) Fun by simp
  qed
qed simp

lemma timpls_transformable_to_match_abss'_nonempty_disj':
  fixes s t::"('a,'b,'c,'d) prot_term"
    and δ::"('a,'b,'c,'d) prot_subst"
    and σ::"('a,'b,'c,'d) prot_var ⇒ 'c set set"
  assumes TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and δ: "timpls_transformable_to TI t (s · δ)"
    and σ: "match_abss' s t = Some σ"
    and x: "x ∈ fv s"
    and t: "fv t = {}"
    and s: "∀f ∈ funs_term s. ¬is_Abs f"
           "∀x ∈ fv s. ∃a. δ x = ⟨a⟩_abs"
    and a: "δ x = ⟨a⟩_abs"
  shows "∀b ∈ σ x. (b,a) ∈ (set TI)*" (is "?P σ x")
proof -
  note 0 = match_abss'_subst_disj_nonempty[OF TI]

  have 1: "s · δ ∈ timpl_closure t (set TI)"
    using timpls_transformable_to_iff_in_timpl_closure[OF TI] δ by blast

  have 2: "match_abss'_timpls_transform TI δ s t = Some σ"
    using match_abss'_timpls_transformI[OF TI δ σ t s] by simp

  show "?P σ x" using 2 TI x t s a
  proof (induction TI δ s t arbitrary: σ rule: match_abss'_timpls_transform.induct)
    case (1 TI δ y c ts) thus ?case
      using match_abss'_timpls_transform_Var_inv[OF "1.prems"(1)] by auto
  next
    case (2 TI δ f ts g ss)
    obtain ϑ where fg: "f = g" "length ts = length ss"
        and ϑ: "Some ϑ = those (map2 (match_abss'_timpls_transform TI δ) ts ss)"
               "σ = fun_point_Union_list ϑ"
               "∀(t, s)∈set (zip ts ss). ∃σ'. match_abss'_timpls_transform TI δ t s = Some σ'"
      using match_abss'_timpls_transform_Fun_inv[OF "2.prems"(1)] by blast
```

```
    have "(b,a) ∈ (set TI)*" when ϑ': "ϑ' ∈ set ϑ" "b ∈ ϑ' x" for ϑ' b
    proof -
      obtain t' s' where t':
          "(t',s') ∈ set (zip ts ss)" "match_abss'_timpls_transform TI δ t' s' = Some ϑ'"
        using those_map2_SomeD[OF ϑ(1)[symmetric] ϑ'(1)] by blast

      have *: "fv s' = {}" "∀f ∈ funs_term t'. ¬is_Abs f" "∀x ∈ fv t'. ∃a. δ x = ⟨a⟩abs"
        using "2.prems"(4-6) set_zip_leftD[OF t'(1)] set_zip_rightD[OF t'(1)]
        by (fastforce, fastforce, fastforce)

      have **: "x ∈ fv t'"
        using ϑ'(2) match_abss'_timpl_transform_nonempty_is_fv[OF t'(2)] by blast

      show ?thesis
        using ϑ'(2) "2.IH"[OF conjI[OF fg] t'(1) _ t'(2) "2.prems"(2) ** * "2.prems"(7)] by blast
    qed
    thus ?case using ϑ(1) unfolding ϑ(2) fun_point_Union_list_def by simp
  qed auto
qed

lemma timpls_transformable_to_match_abss'_nonempty_disj:
  fixes s t::"('a,'b,'c,'d) prot_term"
    and δ::"('a,'b,'c,'d) prot_subst"
    and σ::"('a,'b,'c,'d) prot_var ⇒ 'c set set"
  assumes TI: "set TI = {(a,b) ∈ (set TI)+. a ≠ b}"
    and δ: "timpls_transformable_to TI t (s · δ)"
    and σ: "match_abss' s t = Some σ"
    and x: "x ∈ fv s"
    and t: "fv t = {}"
    and s: "∀f ∈ funs_term s. ¬is_Abs f"
           "∀x ∈ fv s. ∃a. δ x = ⟨a⟩abs"
  shows "⋂(ticl_abs TI ` σ x) ≠ {}"
proof -
  have 0: "(a,b) ∈ (set TI)*" when y: "y ∈ fv s" "a ∈ σ y" "δ y = ⟨b⟩abs" for a b y
    using timpls_transformable_to_match_abss'_nonempty_disj'[OF TI δ σ y(1) t s y(3)] y(2) by blast

  obtain b where b: "δ x = ⟨b⟩abs" using x s(2) by blast

  have "b ∈ ticl_abs TI a" when a: "a ∈ σ x" for a
    using 0[OF x a b] unfolding ticl_abs_iff[OF TI] by blast
  thus ?thesis by blast
qed

lemma timpls_transformable_to_subst_subterm:
  fixes s t::"(('a,'b,'c,'d) prot_fun, 'v) term"
    and δ σ::"(('a,'b,'c,'d) prot_fun, 'v) subst"
  assumes "timpls_transformable_to TI (t · δ) (t · σ)"
    and "s ⊑ t"
  shows "timpls_transformable_to TI (s · δ) (s · σ)"
using assms
proof (induction "t · δ" "t · σ" arbitrary: t rule: timpls_transformable_to.induct)
  case (1 TI x y) thus ?case by (cases t) auto
next
  case (2 TI f T g S)
  note prems = "2.prems"
  note hyps = "2.hyps"(2-)
  note IH = "2.hyps"(1)

  show ?case
  proof (cases "s = t")
    case False
    then obtain h U u where t: "t = Fun h U" "u ∈ set U" "s ⊑ u"
      using prems(2) by (cases t) auto
```

```
    then obtain i where i: "i < length U" "U ! i = u"
      by (metis in_set_conv_nth)

    have "timpls_transformable_to TI (u · δ) (u · σ)"
      using t i prems(1) timpls_transformable_to_inv(2)[of TI h "U ·list δ" h "U ·list σ" i] by simp
    thus ?thesis using IH hyps t by auto
  qed (use prems in auto)
qed simp_all

lemma timpls_transformable_to_subst_match_case:
  assumes "timpls_transformable_to TI s (t · ϑ)"
    and "fv s = {}"
    and "∀f ∈ funs_term t. ¬is_Abs f"
    and "distinct (fv_list t)"
    and "∀x ∈ fv t. ∃a. ϑ x = ⟨a⟩abs"
  shows "∃δ. s = t · δ"
using assms
proof (induction s "t · ϑ" arbitrary: t rule: timpls_transformable_to.induct)
  case (2 TI f T g S)
  note prems = "2.prems"
  note hyps = "2.hyps"(2-)
  note IH = "2.hyps"(1)

  show ?case
  proof (cases t)
    case (Var x)
    then obtain a where a: "t · ϑ = ⟨a⟩abs" using prems(5) by fastforce
    show ?thesis
      using hyps timpls_transformable_to_inv'[OF prems(1)[unfolded a]]
      unfolding Var by force
  next
    case (Fun h U)
    have g: "g = h" and S: "S = U ·list ϑ"
      using hyps unfolding Fun by simp_all

    note 0 = distinct_fv_list_Fun_param[OF prems(4)[unfolded Fun]]

    have 1: "∀f ∈ funs_term u. ¬is_Abs f" when u: "u ∈ set U" for u
      using prems(3) u unfolding Fun by fastforce

    have 2: "fv t' = {}" when t': "t' ∈ set T" for t'
      using t' prems(2) by simp

    have 3: "∀x ∈ fv u. ∃a. ϑ x = ⟨a⟩abs" when u: "u ∈ set U" for u
      using u prems(5) unfolding Fun by simp

    have "¬is_Abs f"
      using prems(3) timpls_transformable_to_inv(3)[OF prems(1)[unfolded hyps[symmetric] S g]]
      unfolding Fun by auto
    hence f: "f = h" and T: "length T = length U"
      using prems(1) timpls_transformable_to_inv(1,3)[of TI f T h "U ·list ϑ"]
      unfolding Fun by fastforce+

    define Δ where "Δ ≡ λi. if i < length T then SOME δ. T ! i = U ! i · δ else undefined"

    have "timpls_transformable_to TI (T ! i) (U ! i · ϑ)" when i: "i < length T" for i
      using prems(1)[unfolded Fun] T i timpls_transformable_to_inv(2)[of TI f T h "U ·list ϑ" i]
      by auto
    hence "∃δ. T ! i = U ! i · δ" when i: "i < length T" for i
      using i T IH[OF _ _ _ 2 1 0 3, of "T ! i" "U ! i"]
      unfolding Fun g S by simp
    hence Δ: "T ! i = U ! i · Δ i" when i: "i < length T" for i
      using i someI2[of "λδ. T ! i = U ! i · δ" _ "λδ. T ! i = U ! i · δ"]
```

```
      unfolding Δ_def by fastforce

    define δ where "δ ≡ λx. if ∃i < length T. x ∈ fv (U ! i)
                         then Δ (SOME i. i < length T ∧ x ∈ fv (U ! i)) x
                         else undefined"

    have "T ! i = U ! i · δ" when i: "i < length T" for i
    proof -
      have "j = i"
        when x: "x ∈ fv (U ! i)" and j: "j < length T" "x ∈ fv (U ! j)" for j x
        using x i j T distinct_fv_list_idx_fv_disjoint[OF prems(4)[unfolded Fun], of h U]
        by (metis (no_types, lifting) disjoint_iff_not_equal neqE term.dual_order.refl)
      hence "δ x = Δ i x" when x: "x ∈ fv (U ! i)" for x
        using x i some_equality[of "λi. i < length T ∧ x ∈ fv (U ! i)" i]
        unfolding δ_def by (metis (no_types, lifting))
      thus ?thesis by (metis Δ i term_subst_eq)
    qed
    hence "T = U ·list δ" by (metis (no_types, lifting) T length_map nth_equalityI nth_map)
    hence "Fun f T = Fun f U · δ" by simp
    thus ?thesis using Fun f by fast
  qed
qed simp_all

lemma timpls_transformable_to_match_abss'_case:
  assumes "timpls_transformable_to TI s (t · ϑ)"
    and "fv s = {}"
    and "∀f ∈ funs_term t. ¬is_Abs f"
    and "∀x ∈ fv t. ∃a. ϑ x = ⟨a⟩abs"
  shows "∃δ. match_abss' t s = Some δ"
using assms
proof (induction s "t · ϑ" arbitrary: t rule: timpls_transformable_to.induct)
  case (2 TI f T g S)
  note prems = "2.prems"
  note hyps = "2.hyps"(2-)
  note IH = "2.hyps"(1)

  show ?case
  proof (cases t)
    case (Var x)
    then obtain a where a: "t · ϑ = ⟨a⟩abs" using prems(4) by fastforce
    thus ?thesis
      using timpls_transformable_to_inv'(4)[OF prems(1)[unfolded a]]
      by (metis (no_types) Var is_Abs_def term.sel(2) match_abss'.simps(1))
  next
    case (Fun h U)
    have g: "g = h" and S: "S = U ·list ϑ"
      using hyps unfolding Fun by simp_all

    have 1: "∀f ∈ funs_term u. ¬is_Abs f" when u: "u ∈ set U" for u
      using prems(3) u unfolding Fun by fastforce

    have 2: "fv t' = {}" when t': "t' ∈ set T" for t'
      using t' prems(2) by simp

    have 3: "∀x ∈ fv u. ∃a. ϑ x = ⟨a⟩abs" when u: "u ∈ set U" for u
      using u prems(4) unfolding Fun by simp

    have "¬is_Abs f"
      using prems(3) timpls_transformable_to_inv(3)[OF prems(1)[unfolded hyps[symmetric] S g]]
      unfolding Fun by auto
    hence f: "f = h" and T: "length T = length U"
      using prems(1) timpls_transformable_to_inv(1,3)[of TI f T h "U ·list ϑ"]
      unfolding Fun by fastforce+
```

```
    define Δ where "Δ ≡ λi.
      if i < length T
      then SOME δ. match_abss' (U ! i) (T ! i) = Some δ
      else undefined"

    have "timpls_transformable_to TI (T ! i) (U ! i · ϑ)" when i: "i < length T" for i
      using prems(1)[unfolded Fun] T i timpls_transformable_to_inv(2)[of TI f T h "U ·list ϑ" i]
      by auto
    hence "∃δ. match_abss' (U ! i) (T ! i) = Some δ" when i: "i < length T" for i
      using i T IH[OF _ _ _ 2 1 3, of "T ! i" "U ! i"]
      unfolding Fun g S by simp
    hence "match_abss' (U ! i) (T ! i) = Some (Δ i)" when i: "i < length T" for i
      using i someI2[of "λδ. match_abss' (U ! i) (T ! i) = Some δ" _
                        "λδ. match_abss' (U ! i) (T ! i) = Some δ"]
      unfolding Δ_def by fastforce
    thus ?thesis
      using match_abss'_FunI[OF _ T] unfolding Fun f by auto
  qed
qed simp_all

lemma timpls_transformable_to_match_abss_case:
  assumes TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and "timpls_transformable_to TI s (t · ϑ)"
    and "fv s = {}"
    and "∀f ∈ funs_term t. ¬is_Abs f"
    and "∀x ∈ fv t. ∃a. ϑ x = ⟨a⟩abs"
  shows "∃δ. match_abss OCC TI t s = Some δ"
proof -
  obtain δ where δ: "match_abss' t s = Some δ"
    using timpls_transformable_to_match_abss'_case[OF assms(2-)] by blast

  show ?thesis
    using δ timpls_transformable_to_match_abss'_nonempty_disj[OF assms(1,2) δ _ assms(3-5)]
    unfolding match_abss_def by simp
qed

lemma timpls_transformable_to_match_abss_obtain:
  assumes TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and s_t_timpl: "timpls_transformable_to TI s (t · ϑ)"
    and s_ground: "fv s = {}"
    and t_no_abs: "∀f ∈ funs_term t. ¬is_Abs f"
    and t_ϑ_abs: "∀x ∈ fv t. ∃a. ϑ x = ⟨a⟩abs"
  obtains δ where "match_abss OCC TI t s = Some δ"
    and "∀x a. x ∈ fv t ∧ ϑ x = ⟨a⟩abs ⟶ a ∈ δ x"
    and "∀x. x ∉ fv t ⟶ δ x = set OCC"
proof -
  let ?f = "λδ x. if x ∈ fv t then δ x else set OCC"
  let ?g = "λδ x. ⋂(ticl_abs TI ` δ x)"

  obtain δ where δ: "match_abss OCC TI t s = Some δ"
    using timpls_transformable_to_match_abss_case[OF TI s_t_timpl s_ground t_no_abs t_ϑ_abs]
    by blast

  obtain σ where σ:
      "match_abss' t s = Some σ" "δ = ?f (?g σ)"
    using match_abssD[OF δ] by blast

  have "∀b ∈ σ x. a ∈ ticl_abs TI b" when x: "x ∈ fv t" and a: "ϑ x = ⟨a⟩abs" for x a
    using timpls_transformable_to_match_abss'_nonempty_disj'[
              OF TI s_t_timpl σ(1) x s_ground t_no_abs t_ϑ_abs a]
    unfolding ticl_abs_iff[OF TI]
    by simp
```

```
  hence 0: "a ∈ δ x" when x: "x ∈ fv t" and a: "ϑ x = ⟨a⟩abs" for x a
    using x a σ(2) by simp

  have 1: "δ x = set OCC" when x: "x ∉ fv t" for x
    using x match_abss_OCC_if_not_fv[OF δ x]
    by simp

  show ?thesis using δ σ 0 1 that by blast
qed

private lemma transaction_check_variant_soundness_aux3:
  fixes T FP S C xs OCC negs poss as
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "C ≡ unlabel (transaction_checks T)"
    and "S ≡ unlabel (transaction_strand T)"
    and "ts ≡ trms_listsst (unlabel (transaction_receive T))"
    and "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_listsst S)"
  assumes TI0: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
               "∀(a,b) ∈ set TI. a ≠ b"
    and OCC: "∀t ∈ set FP. ∀a. Abs a ∈ funs_term t ⟶ a ∈ set OCC"
    and FP_ground: "ground (set FP)"
    and x: "x ∈ set xs"
    and xs: "∀x. x ∈ set xs ⟶ δ x ∈ set OCC"
            "∀x. x ∈ set xs ⟶ poss x ⊆ δ x"
            "∀x. x ∈ set xs ⟶ δ x ∩ negs x = {}"
            "∀x. x ∉ set xs ⟶ δ x = {}"
    and ts: "∀t ∈ trmslsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)"
            "∀t ∈ trmslsst (transaction_receive T). ∀f ∈ funs_term t. ¬is_Abs f"
            "∀x ∈ fvset (trmslsst (transaction_receive T)). fst x = TAtom Value"
    and C: "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ set C ⟶ s ∈ δ x"
           "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ set C ⟶ s ∉ δ x"
    and σ: "synth_abs_substs_constrs_rel FP OCC TI ts σ"
  shows "δ x ∈ σ x"
proof -
  note defs = assms(1-5)

  note TI = trancl_eqI'[OF TI0]

  have δx0: "δ x ∈ set OCC" "poss x ⊆ δ x" "δ x ∩ negs x = {}" using x xs by (blast,blast,blast)

  have ts0: "∀t ∈ set ts. intruder_synth_mod_timpls FP TI (t · ϑ δ)"
    using ts(1) trms_listsst_is_trmssst unfolding ts_def by blast

  have ts1: "¬Fun (Abs n) S ⊑set set ts" for n S
    using ts(2) funs_term_Fun_subterm'
    unfolding ts_def trms_transaction_unfold trms_listsst_is_trmssst[symmetric] is_Abs_def
    by fastforce

  have ts2: "∀x ∈ fvset (set ts). fst x = TAtom Value"
    using ts(3)
    unfolding ts_def trms_transaction_unfold trms_listsst_is_trmssst[symmetric]
    by fastforce

  show ?thesis using σ ts0 ts1 ts2
  proof (induction rule: synth_abs_substs_constrs_rel.induct)
    case (SolvePrivConstNotin c)
    hence "intruder_synth_mod_timpls FP TI (Fun c [])" by force
    hence "list_ex (λt. timpls_transformable_to TI t (Fun c [])) FP"
      using SolvePrivConstNotin.hyps(1,2) by simp
    then obtain t where t: "t ∈ set FP" "timpls_transformable_to TI t (Fun c [])"
      unfolding list_ex_iff by blast

    have "¬is_Abs c"
```

```
      using SolvePrivConstNotin.prems(2)[of _ "[]"]
      by (metis in_subterms_Union is_Abs_def list.set_intros(1))
    hence "t = Fun c []"
      using t(2) timpls_transformable_to_inv[of TI] by (cases t) auto
    thus ?case using t(1) SolvePrivConstNotin.hyps(3) by fast
next
  case (SolveValueVar ϑ1 y)
  have "list_ex (λt. timpls_transformable_to TI t ⟨δ y⟩_abs) FP"
    using SolveValueVar.prems(1-3) unfolding ϑ_def by simp
  then obtain t where t: "t ∈ set FP" "timpls_transformable_to TI t ⟨δ y⟩_abs"
    unfolding list_ex_iff by blast

  obtain a where a: "t = ⟨a⟩_abs" "a = δ y ∨ (a, δ y) ∈ set TI"
  proof -
    obtain ft tst where ft: "t = Fun ft tst"
      using t(2) timpls_transformable_to_inv_Var(1)[of TI _ "⟨δ y⟩_abs"]
      by (cases t) auto

    have "tst = []" "is_Abs ft" "the_Abs ft = δ y ∨ (the_Abs ft, δ y) ∈ set TI"
      using timpls_transformable_to_inv'(2,4,5)[OF t(2)[unfolded ft]]
      by (simp, force, force)
    thus thesis using that[of "the_Abs ft"] ft by force
  qed

  have "a ∈ set OCC"
    using t(1)[unfolded a(1)] OCC by auto
  thus ?case
    using δx0(1) t(1)[unfolded a(1)] a(2)
    unfolding SolveValueVar.hyps(1) ticl_abss_def ticl_abs_def
    by force
next
  case (SolvePubComposed g us Δ ϑ1 ϑ2) show ?case
  proof (cases "∀t ∈ set us. intruder_synth_mod_timpls FP TI (t · ϑ δ)")
    case True
    hence "δ x ∈ ϑ2 x"
      using SolvePubComposed.IH SolvePubComposed.prems(2,3)
            distinct_fv_list_Fun_param[of g us]
      by auto
    thus ?thesis unfolding fun_point_union_def by simp
  next
    case False
    hence "list_ex (λt. timpls_transformable_to TI t (Fun g us · ϑ δ)) FP"
      using SolvePubComposed.prems(1) intruder_synth_mod_timpls.simps(2)[of FP TI g "us ·_list ϑ δ"]
      unfolding list_all_iff by auto
    then obtain t where t: "t ∈ set FP" "timpls_transformable_to TI t (Fun g us · ϑ δ)"
      unfolding list_ex_iff by blast

    have t_ground: "fv t = {}"
      using t(1) FP_ground by simp

    have g_no_abs: "¬is_Abs f" when f: "f ∈ funs_term (Fun g us)" for f
    proof -
      obtain fts where "Fun f fts ⊑ Fun g us" using funs_term_Fun_subterm[OF f] by blast
      thus ?thesis using SolvePubComposed.prems(2)[of _ fts] by (cases f) auto
    qed

    have g_ϑ_abs: "∃a. ϑ δ y = ⟨a⟩_abs" when y: "y ∈ fv (Fun g us)" for y
      using y SolvePubComposed.prems(3) unfolding ϑ_def by fastforce

    let ?h1 = "λδ x. if x ∈ fv (Fun g us) then δ x else set OCC"
    let ?h2 = "λδ x. ⋂(ticl_abs TI ` δ x)"

    obtain δ' where δ':
```

```
        "match_abss OCC TI (Fun g us) t = Some δ'"
        "∀x a. x ∈ fv (Fun g us) ∧ ϑ δ x = ⟨a⟩abs ⟶ a ∈ δ' x"
        "∀x. x ∉ fv (Fun g us) ⟶ δ' x = set OCC"
      using g_no_abs g_ϑ_abs
            timpls_transformable_to_match_abss_obtain[OF TI t(2) t_ground, of OCC thesis]
      by blast

    have δ'_Δ: "δ' ∈ Δ"
      using t(1) δ'(1) SolvePubComposed.hyps(4) by metis

    have "δ x ∈ δ' x" when x_in_g: "x ∈ fv (Fun g us)"
    proof -
      have "fst x = TAtom Value" using x_in_g SolvePubComposed.prems(3) by auto
      hence "ϑ δ x = ⟨δ x⟩abs" unfolding ϑ_def by simp
      thus "δ x ∈ δ' x" using δ'(2) x_in_g by blast
    qed
    hence "δ x ∈ δ' x" using δ'(3) δx0(1) by blast
    hence "δ x ∈ ϑ1 x"
      using δ'_Δ δx0(1) unfolding SolvePubComposed.hyps(5) fun_point_Union_def by auto
    thus ?thesis unfolding fun_point_union_def by simp
  qed
 qed (auto simp add: δx0 fun_point_Inter_def)
qed

private lemma transaction_check_variant_soundness_aux4:
 fixes T FP S C xs OCC negs poss as
 defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
   and "C ≡ unlabel (transaction_checks T)"
   and "S ≡ unlabel (transaction_strand T)"
   and "xas ≡ (the_Abs ∘ the_Fun) ` set (filter (λt. is_Fun t ∧ is_Abs (the_Fun t)) FP)"
   and "ts ≡ trms_listsst (unlabel (transaction_receive T))"
   and "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_listsst S)"
   and "poss ≡ transaction_poschecks_comp C"
   and "negs ≡ transaction_negchecks_comp C"
   and "as ≡ map (λx. (x, set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC))) xs"
   and "f ≡ λx. case List.find (λp. fst p = x) as of Some p ⇒ snd p | None ⇒ {}"
 assumes T_adm: "admissible_transaction' T"
   and TI0: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
            "∀(a,b) ∈ set TI. a ≠ b"
   and OCC: "∀t ∈ set FP. ∀a. Abs a ∈ funs_term t ⟶ a ∈ set OCC"
   and FP_ground: "ground (set FP)"
   and FP_wf: "wftrms (set FP)"
   and "x ∈ set xs"
   and "∀x. x ∈ set xs ⟶ δ x ∈ set OCC"
   and "∀x. x ∈ set xs ⟶ poss x ⊆ δ x"
   and "∀x. x ∈ set xs ⟶ δ x ∩ negs x = {}"
   and "∀x. x ∉ set xs ⟶ δ x = {}"
   and "∀t ∈ trmslsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)"
   and "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ set C ⟶ s ∈ δ x"
   and "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ set C ⟶ s ∉ δ x"
 shows "δ x ∈ synth_abs_substs_constrs (FP,OCC,TI) T x"
proof -
 let ?FPT = "(FP,OCC,TI)"
 let ?P = "λs u. let δ = mgu s u
                 in δ ≠ None ⟶ (∀x ∈ fv s. is_Fun (the δ x) ⟶ is_Abs (the_Fun (the δ x)))"

 define ϑ0 where "ϑ0 ≡ λx.
   if fst x = TAtom Value ∧ x ∈ fv_transaction T ∧ x ∉ set (transaction_fresh T)
   then {ab ∈ set OCC. poss x ⊆ ab ∧ negs x ∩ ab = {}} else {}"

 define g where "g ≡ λx. set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC)"
 define gs where "gs ≡ map (λx. (x, g x)) xs"
```

```
  note defs = assms(3-10) ϑ0_def
  note assm = assms(11-)[unfolded defs]

  have ts0: "∀t ∈ set ts. wf_trm t"
    using admissible_transaction_is_wellformed_transaction(4)[OF T_adm]
    unfolding admissible_transaction_terms_def wf_trms_code[symmetric]
              ts_def trms_list_sst_is_trms_sst[symmetric]
              trms_transaction_unfold
    by fast

  have ts1: "∀t ∈ set ts. ∀f ∈ funs_term t. ¬is_Abs f"
    using protocol_transactions_no_abss[OF T_adm] funs_term_Fun_subterm
          trms_sst_unlabel_prefix_subset(1)
    unfolding ts_def trms_list_sst_is_trms_sst[symmetric] is_Abs_def transaction_strand_def
    by (metis (no_types, opaque_lifting) in_subterms_Union in_subterms_subset_Union subset_iff)

  have ts2: "∀x ∈ fv_set (set ts). fst x = TAtom Value"
    using admissible_transaction_Value_vars[OF T_adm]
          wellformed_transaction_send_receive_fv_subset(1)[
            OF admissible_transaction_is_wellformed_transaction(1)[OF T_adm]]
    unfolding ts_def trms_transaction_unfold trms_list_sst_is_trms_sst[symmetric] Γ_v_TAtom''(2)
    by fastforce

  have "f x = ϑ0 x" for x
  proof (cases "fst x = Var Value ∧ x ∈ fv_transaction T ∧ x ∉ set (transaction_fresh T)")
    case True
    hence "ϑ0 x = {ab ∈ set OCC. poss x ⊆ ab ∧ negs x ∩ ab = {}}" unfolding ϑ0_def by argo
    thus ?thesis
      using True transaction_check_variant_soundness_aux0[OF S_def xs_def, of x]
            transaction_check_variant_soundness_aux1[
              OF C_def S_def xs_def poss_def negs_def as_def f_def, of x]
      by simp
  next
    case False
    hence 0: "ϑ0 x = {}" unfolding ϑ0_def by argo

    have "x ∉ set xs"
      using False fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"]
      unfolding xs_def S_def by fastforce
    hence "List.find (λp. fst p = x) gs = None"
      using find_None_iff[of "λp. fst p = x" gs] unfolding gs_def by simp
    hence "f x = {}"
      unfolding f_def as_def gs_def g_def by force
    thus ?thesis using 0 by simp
  qed
  thus ?thesis
    using synth_abs_substs_constrs_rel_if_synth_abs_substs_constrs[
            OF _ FP_ground FP_wf, of T, unfolded trms_list_sst_is_trms_sst ts_def[symmetric], OF ts0]
          transaction_check_variant_soundness_aux3[
            OF TI0 OCC FP_ground assm(7-11),
            of "synth_abs_substs_constrs ?FPT T",
            unfolded trms_list_sst_is_trms_sst ts_def[symmetric],
            OF assm(12)[unfolded ϑ_def trms_list_sst_is_trms_sst ts_def[symmetric]]
               ts1 ts2 assm(13-)[unfolded C_def]]
    unfolding defs synth_abs_substs_constrs_def Let_def by blast
qed

private lemma transaction_check_variant_soundness_aux5:
  fixes FP OCC TI T S C
  defines "msgcs ≡ λx a. a ∈ synth_abs_substs_constrs (FP,OCC,TI) T x"
    and "S ≡ unlabel (transaction_strand T)"
    and "C ≡ unlabel (transaction_checks T)"
    and "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S)"
```

```
    and "poss ≡ transaction_poschecks_comp C"
    and "negs ≡ transaction_negchecks_comp C"
  assumes T_adm: "admissible_transaction' T"
    and TI: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
            "∀(a,b) ∈ set TI. a ≠ b"
    and OCC: "∀t ∈ set FP. ∀a. Abs a ∈ funs_term t ⟶ a ∈ set OCC"
    and FP: "ground (set FP)"
            "wf_trms (set FP)"
    and δ: "δ ∈ abs_substs_fun ` set (abs_substs_set xs OCC poss negs (λ_ _. True))"
           "transaction_check_pre (FP,OCC,TI) T δ"
  shows "δ ∈ abs_substs_fun ` set (abs_substs_set xs OCC poss negs msgcs)"
proof -
  have 0: "δ x ∈ set OCC" "poss x ⊆ δ x" "δ x ∩ negs x = {}" when x: "x ∈ set xs" for x
    using abs_substs_abss_bounded[OF δ(1) x] by simp_all

  have 1: "δ x = {}" when x: "x ∉ set xs" for x
    by (rule abs_substs_abss_bounded'[OF δ(1) x])

  have 2: "msgcs x (δ x)" when x: "x ∈ set xs" for x
    using 0 1 x transaction_check_variant_soundness_aux4[OF T_adm TI OCC FP, of x δ]
          transaction_check_pre_ReceiveE[OF δ(2)] transaction_check_pre_InSetE[OF δ(2)]
          transaction_check_pre_NotInSetE[OF δ(2)]
    unfolding msgcs_def xs_def C_def S_def negs_def poss_def by fast

  show ?thesis
    using abs_substs_has_abs[of xs δ OCC poss negs msgcs] 0 1 2 by blast
qed

theorem transaction_check_variant_soundness:
  assumes P_adm: "∀T ∈ set P. admissible_transaction' T"
    and TI: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
            "∀(a,b) ∈ set TI. a ≠ b"
    and OCC: "∀t ∈ set FP. ∀a. Abs a ∈ funs_term t ⟶ a ∈ set OCC"
    and FP: "ground (set FP)"
            "wf_trms (set FP)"
    and T_in: "T ∈ set P"
    and T_check: "transaction_check_coverage_rcv (FP,OCC,TI) T"
  shows "transaction_check (FP,OCC,TI) T"
proof -
  have 0: "admissible_transaction' T"
    using P_adm T_in by blast

  show ?thesis
    using T_check transaction_check_variant_soundness_aux5[OF 0 TI OCC FP]
    unfolding transaction_check_def transaction_check'_def transaction_check_coverage_rcv_def
              transaction_check_comp_def list_all_iff Let_def
    by force
qed

end

end
```

3.6.7 Automatic Fixed-Point Computation

```
context stateful_protocol_model
begin

fun reduce_fixpoint' where
  "reduce_fixpoint' FP _ [] = FP"
| "reduce_fixpoint' FP TI (t#M) = (
  let FP' = List.removeAll t FP
  in if intruder_synth_mod_timpls FP' TI t then FP' else reduce_fixpoint' FP TI M)"
```

```
definition reduce_fixpoint where
  "reduce_fixpoint FP TI ≡
    let f = λFP. reduce_fixpoint' FP TI FP
    in while (λM. set (f M) ≠ set M) f FP"

definition compute_fixpoint_fun' where
  "compute_fixpoint_fun' P (n::nat option) enable_traces Δ S0 ≡
    let P' = map add_occurs_msgs P;

        sy = intruder_synth_mod_timpls;

        FP' = λS. fst (fst S);
        TI' = λS. snd (fst S);
        OCC' = λS. remdups (
          (map (λt. the_Abs (the_Fun (args t ! 1)))
               (filter (λt. is_Fun t ∧ the_Fun t = OccursFact) (FP' S)))@
          (map snd (TI' S)));

        equal_states = λS S'. set (FP' S) = set (FP' S') ∧ set (TI' S) = set (TI' S');

        trace' = λS. snd S;

        close  = λM f. let g = remdups ∘ f in while (λA. set (g A) ≠ set A) g M;
        close' = λM f. let g = remdups ∘ f in while (λA. set (g A) ≠ set A) g M;
        trancl_minus_refl = λTI.
          let aux = λts p. map (λq. (fst p,snd q)) (filter ((=) (snd p) ∘ fst) ts)
          in filter (λp. fst p ≠ snd p) (close' TI (λts. concat (map (aux ts) ts)@ts));
        snd_Ana = λN M TI. let N' = filter (λt. ∀k ∈ set (fst (Ana t)). sy M TI k) N in
          filter (λt. ¬sy M TI t)
                 (concat (map (λt. filter (λs. s ∈ set (snd (Ana t))) (args t)) N'));
        Ana_cl = λFP TI.
          close FP (λM. (M@snd_Ana M M TI));
        TI_cl = λFP TI.
          close FP (λM. (M@filter (λt. ¬sy M TI t)
                                  (concat (map (λm. concat (map (λ(a,b). ⟨a --» b⟩⟨m⟩) TI)) M))));
        Ana_cl' = λFP TI.
          let K = λt. set (fst (Ana t));
              flt = λM t. (∃k ∈ K t. ¬sy M TI k) ∧ (∃k ∈ K t. ∃f ∈ funs_term k. is_Abs f);
              N = λM. comp_timpl_closure_list (filter (flt M) M) TI
          in close FP (λM. M@snd_Ana (N M) M TI);

        Δ' = λS. Δ (FP' S, OCC' S, TI' S);
        result = λS T δ.
          let not_fresh = λx. x ∉ set (transaction_fresh T);
              xs = filter not_fresh (fv_list_sst (unlabel (transaction_strand T)));
              u = λδ x. absdbupd (unlabel (transaction_strand T)) x (δ x)
          in (remdups (filter (λt. ¬sy (FP' S) (TI' S) t)
                          (concat (map (λts. the_msgs ts ·_list (absc ∘ u δ))
                                       (filter is_Send (unlabel (transaction_send T)))))),
              remdups (filter (λs. fst s ≠ snd s) (map (λx. (δ x, u δ x)) xs)));
        result_tuple = λS T δ. (result S T (abs_substs_fun δ), if enable_traces then δ else []);
        update_state = λS. if list_ex (λt. is_Fun t ∧ is_Attack (the_Fun t)) (FP' S) then S
          else let results = map (λT. map (result_tuple S T) (Δ' S T)) P';
                   newtrace_flt = (λn. let x = map fst (results ! n); y = map fst x; z = map snd x
                    in set (concat y) - set (FP' S) ≠ {} ∨ set (concat z) - set (TI' S) ≠ {});
                   trace =
                    if enable_traces
                    then trace' S@[concat (map (λi. map (λa. (i,snd a)) (results ! i))
                                           (filter newtrace_flt [0..<length results]))]
                    else [];
                   U = map fst (concat results);
                   V = ((remdups (concat (map fst U)@FP' S),
```

```
                      remdups (filter (λx. fst x ≠ snd x) (concat (map snd U)@TI' S))),
                     trace);
                W = ((Ana_cl (TI_cl (FP' V) (TI' V)) (TI' V),
                      trancl_minus_refl (TI' V)),
                     trace' V)
        in if ¬equal_states W S then W
        else ((Ana_cl' (FP' W) (TI' W), TI' W), trace' W);

      S = ((λh. case n of None ⇒ while (λS. ¬equal_states S (h S)) h | Some m ⇒ h ^^ m)
           update_state S0)
    in ((reduce_fixpoint (FP' S) (TI' S), OCC' S, TI' S), trace' S)"
```

definition *compute_fixpoint_fun* **where**

```
  "compute_fixpoint_fun P ≡
    let P' = (filter (λT. transaction_updates T ≠ [] ∨ transaction_send T ≠ []) (remdups P));
        f = (λFPT T. let msgcs = synth_abs_substs_constrs FPT T
                     in transaction_check_comp (λx a. a ∈ msgcs x) FPT T)
    in fst (compute_fixpoint_fun' P' None False f (([],[]),[]))"
```

lemmas *compute_fixpoint_fun_code[code]*

```
        = compute_fixpoint_fun_def[simplified compute_fixpoint_fun'_def[of _ "None" "False" _
"(([],[]),[])"
                                  ,simplified reduce_fixpoint_def o_def Option.option.case if_False]]
```

definition *compute_fixpoint_with_trace* **where**

```
  "compute_fixpoint_with_trace P ≡
    compute_fixpoint_fun' P None True (transaction_check_comp (λ_ _. True)) (([],[]),[])"
```

definition *compute_fixpoint_from_trace* **where**

```
  "compute_fixpoint_from_trace P trace ≡
    let P' = map add_occurs_msgs P;
        Δ = λFPT T.
          let pre_check = transaction_check_pre FPT T;
              δs = map snd (filter (λ(i,as). P' ! i = T) (concat trace))
          in filter (λδ. pre_check (abs_substs_fun δ)) δs;
        f = compute_fixpoint_fun' ∘ map (nth P);
        g = λL FPT. fst ((f L (Some 1) False Δ ((fst FPT,snd (snd FPT)),[])))
    in fold g (map (map fst) trace) ([],[],[])"
```

definition *compute_reduced_attack_trace* **where**

```
  "compute_reduced_attack_trace P trace ≡
    let attack_in_fixpoint = list_ex (λt. ∃f ∈ funs_term t. is_Attack f) ∘ fst;
        is_attack_trace = attack_in_fixpoint ∘ compute_fixpoint_from_trace P;

        trace' =
          let is_attack_transaction =
                list_ex is_Fun_Attack ∘ concat ∘ map the_msgs ∘
                filter is_Send ∘ unlabel ∘ transaction_send;
              trace' =
                if trace = [] then []
                else butlast trace@[filter (is_attack_transaction ∘ nth P ∘ fst) (last trace)]
           in trace';

        iter = λtrace_prev trace_rest elem (prev,rest).
          let next =
            if is_attack_trace (trace_prev@(prev@rest)#trace_rest)
            then prev
            else prev@[elem]
          in (next, tl rest);
        iter' = λtrace_part (trace_prev,trace_rest).
          let updated = foldr (iter trace_prev (tl trace_rest)) trace_part ([],tl (rev trace_part))
          in (trace_prev@[rev (fst updated)], tl trace_rest);
```

```
      reduced_trace = fst (fold iter' trace' ([],trace'))
    in concat reduced_trace"
```

end

3.6.8 Locales for Protocols Proven Secure through Fixed-Point Coverage

```
type_synonym ('f,'a,'s,'l) fixpoint_triple =
  "('f,'a,'s,'l) prot_term list × 's set list × ('s set × 's set) list"

context stateful_protocol_model
begin

definition "attack_notin_fixpoint (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) ≡
  list_all (λt. ∀f ∈ funs_term t. ¬is_Attack f) (fst FPT)"

definition "protocol_covered_by_fixpoint (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) P ≡
  list_all (transaction_check FPT)
           (filter (λT. transaction_updates T ≠ [] ∨ transaction_send T ≠ [])
                   (map add_occurs_msgs P))"

definition "protocol_covered_by_fixpoint_coverage_rcv (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) P
≡
  list_all (transaction_check_coverage_rcv FPT)
           (filter (λT. transaction_updates T ≠ [] ∨ transaction_send T ≠ [])
                   (map add_occurs_msgs P))"

definition "analyzed_fixpoint (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) ≡
  let (FP, _, TI) = FPT
  in analyzed_closed_mod_timpls FP TI"

definition "wellformed_protocol_SMP_set (P::('fun,'atom,'sets,'lbl) prot) N ≡
  has_all_wt_instances_of Γ (⋃T ∈ set P. trms_transaction T) (set N) ∧
  comp_tfr_set arity Ana Γ (set N) ∧
  list_all (λT. list_all (comp_tfr_sstp Γ Pair) (unlabel (transaction_strand T))) P"


definition "wellformed_protocol'' (P::('fun,'atom,'sets,'lbl) prot) N ≡
  let f = λT. transaction_fresh T = [] ⟶ transaction_updates T ≠ [] ∨ transaction_send T ≠ []
  in list_all (λT. list_all is_Receive (unlabel (transaction_receive T)) ∧
                   list_all is_Check_or_Assignment (unlabel (transaction_checks T)) ∧
                   list_all is_Update (unlabel (transaction_updates T)) ∧
                   list_all is_Send (unlabel (transaction_send T)))
              P ∧
     list_all admissible_transaction (filter f P) ∧
     wellformed_protocol_SMP_set P N"

definition "wellformed_protocol' (P::('fun,'atom,'sets,'lbl) prot) N ≡
  wellformed_protocol'' P N ∧
  has_initial_value_producing_transaction P"

definition "wellformed_protocol (P::('fun,'atom,'sets,'lbl) prot) ≡
  let f = λM. remdups (concat (map subterms_list M@map (fst ∘ Ana) M));
      N0 = remdups (concat (map (trms_list_sst ∘ unlabel ∘ transaction_strand) P));
      N = while (λA. set (f A) ≠ set A) f N0
  in wellformed_protocol' P N"

definition "wellformed_fixpoint' (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) ≡
  let (FP, OCC, TI) = FPT; OCC' = set OCC
  in list_all (λt. wf_trm' arity t ∧ fv t = {}) FP ∧
     list_all (λa. a ∈ OCC') (map snd TI) ∧
     list_all (λt. ∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ OCC') FP"
```

```
definition "wellformed_term_implication_graph (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) ≡
  let (_, _, TI) = FPT
  in list_all (λ(a,b). list_all (λ(c,d). b = c ∧ a ≠ d ⟶ List.member TI (a,d)) TI) TI ∧
     list_all (λp. fst p ≠ snd p) TI"

definition "wellformed_fixpoint (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) ≡
  wellformed_fixpoint' FPT ∧ wellformed_term_implication_graph FPT"

lemma wellformed_protocol_SMP_set_mono:
  assumes "wellformed_protocol_SMP_set P S"
    and "set P' ⊆ set P"
  shows "wellformed_protocol_SMP_set P' S"
using assms
unfolding wellformed_protocol_SMP_set_def comp_tfr_set_def has_all_wt_instances_of_def
          wf_trms'_def list_all_iff
by fast

lemma wellformed_protocol''_mono:
  assumes "wellformed_protocol'' P S"
    and "set P' ⊆ set P"
  shows "wellformed_protocol'' P' S"
using assms wellformed_protocol_SMP_set_mono[of P S P']
unfolding wellformed_protocol''_def list_all_iff by auto

lemma wellformed_protocol'_mono:
  assumes "wellformed_protocol' P S"
    and "set P' ⊆ set P"
    and "has_initial_value_producing_transaction P'"
  shows "wellformed_protocol' P' S"
using assms wellformed_protocol_SMP_set_mono[of P S P'] wellformed_protocol''_mono
unfolding wellformed_protocol'_def by blast

lemma protocol_covered_by_fixpoint_if_protocol_covered_by_fixpoint_coverage_rcv:
  assumes P: "wellformed_protocol'' P P_SMP"
    and FPT: "wellformed_fixpoint FPT"
    and covered: "protocol_covered_by_fixpoint_coverage_rcv FPT P"
  shows "protocol_covered_by_fixpoint FPT P"
proof -
  obtain FP OCC TI where FPT': "FPT = (FP,OCC,TI)" by (metis surj_pair)

  note defs = FPT' wellformed_protocol''_def wellformed_fixpoint_def wellformed_fixpoint'_def
              wellformed_term_implication_graph_def Let_def
              wf_trms_code[symmetric] wf_trm_code[symmetric]
              member_def case_prod_unfold list_all_iff

  let ?f = "λT. transaction_fresh T = [] ⟶ transaction_updates T ≠ [] ∨ transaction_send T ≠ []"

  have TI: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
              "∀(a,b) ∈ set TI. a ≠ b"
      and OCC: "∀t ∈ set FP. ∀a. Abs a ∈ funs_term t ⟶ a ∈ set OCC"
      and FP: "ground (set FP)"
              "wf_trms (set FP)"
    using FPT unfolding defs by (simp, simp, fastforce, simp, simp)

  have P_adm: "∀T ∈ set (filter ?f (map add_occurs_msgs P)). admissible_transaction' T"
    using P add_occurs_msgs_admissible_occurs_checks(1)[OF admissible_transactionE'(1)]
    unfolding defs add_occurs_msgs_updates_send_filter_iff'[of P, symmetric] by auto

  show ?thesis
    using covered transaction_check_variant_soundness[OF P_adm TI OCC FP]
    unfolding protocol_covered_by_fixpoint_def protocol_covered_by_fixpoint_coverage_rcv_def
              FPT' list_all_iff
    by fastforce
```

```
qed

lemma protocol_covered_by_fixpoint_if_protocol_covered_by_fixpoint_coverage_rcv':
  assumes P: "wellformed_protocol'' P P_SMP"
    and P': "set P' ⊆ set P"
    and FPT: "wellformed_fixpoint FPT"
    and covered: "protocol_covered_by_fixpoint_coverage_rcv FPT P'"
  shows "protocol_covered_by_fixpoint FPT P'"
using protocol_covered_by_fixpoint_if_protocol_covered_by_fixpoint_coverage_rcv[OF _ FPT covered]
      wellformed_protocol''_mono[OF P P']
by simp

lemma protocol_covered_by_fixpoint_trivial_case:
  assumes "list_all (λT. transaction_updates T = [] ∧ transaction_send T = [])
                  (map add_occurs_msgs P)"
  shows "protocol_covered_by_fixpoint FPT P"
using assms
by (simp add: list_all_iff transaction_check_trivial_case protocol_covered_by_fixpoint_def)

lemma protocol_covered_by_fixpoint_empty[simp]:
  "protocol_covered_by_fixpoint FPT []"
by (simp add: protocol_covered_by_fixpoint_def)

lemma protocol_covered_by_fixpoint_Cons[simp]:
  "protocol_covered_by_fixpoint FPT (T#P) ⟷
    transaction_check FPT (add_occurs_msgs T) ∧ protocol_covered_by_fixpoint FPT P"
using transaction_check_trivial_case[of "add_occurs_msgs T"]
unfolding protocol_covered_by_fixpoint_def case_prod_unfold by simp

lemma protocol_covered_by_fixpoint_append[simp]:
  "protocol_covered_by_fixpoint FPT (P1@P2) ⟷
   protocol_covered_by_fixpoint FPT P1 ∧ protocol_covered_by_fixpoint FPT P2"
by (simp add: protocol_covered_by_fixpoint_def case_prod_unfold)

lemma protocol_covered_by_fixpoint_I1[intro]:
  assumes "list_all (protocol_covered_by_fixpoint FPT) P"
  shows "protocol_covered_by_fixpoint FPT (concat P)"
using assms by (auto simp add: protocol_covered_by_fixpoint_def list_all_iff)

lemma protocol_covered_by_fixpoint_I2[intro]:
  assumes "protocol_covered_by_fixpoint FPT P1"
    and "protocol_covered_by_fixpoint FPT P2"
  shows "protocol_covered_by_fixpoint FPT (P1@P2)"
using assms by (auto simp add: protocol_covered_by_fixpoint_def)

lemma protocol_covered_by_fixpoint_I3:
  assumes "∀T ∈ set P. ∀δ::('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set.
    transaction_check_pre FPT (add_occurs_msgs T) δ ⟶
    transaction_check_post FPT (add_occurs_msgs T) δ"
  shows "protocol_covered_by_fixpoint FPT P"
using assms
unfolding protocol_covered_by_fixpoint_def transaction_check_def transaction_check'_def
          transaction_check_comp_def list_all_iff Let_def case_prod_unfold
          Product_Type.fst_conv Product_Type.snd_conv
by fastforce

lemmas protocol_covered_by_fixpoint_intros =
  protocol_covered_by_fixpoint_I1
  protocol_covered_by_fixpoint_I2
  protocol_covered_by_fixpoint_I3

lemma prot_secure_if_prot_checks:
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
```

```
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
  assumes attack_notin_fixpoint: "attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered: "protocol_covered_by_fixpoint FP_OCC_TI P"
    and analyzed_fixpoint: "analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol: "wellformed_protocol' P N"
    and wellformed_fixpoint: "wellformed_fixpoint FP_OCC_TI"
  shows "∀𝒜 ∈ reachable_constraints P. ∄ℐ. constraint_model ℐ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
    (is "?secure P")
proof -
  define FP where "FP ≡ let (FP,_,_) = FP_OCC_TI in FP"
  define OCC where "OCC ≡ let (_,OCC,_) = FP_OCC_TI in OCC"
  define TI where "TI ≡ let (_,_,TI) = FP_OCC_TI in TI"

  define f where "f ≡ λT::('fun,'atom,'sets,'lbl) prot_transaction.
    transaction_fresh T = [] ⟶ transaction_updates T ≠ [] ∨ transaction_send T ≠ []"

  define g where "g ≡ λT::('fun,'atom,'sets,'lbl) prot_transaction.
    transaction_fresh T = [] ⟶
      list_ex (λa. is_Update (snd a) ∨ is_Send (snd a)) (transaction_strand T)"

  note wellformed_prot_defs =
    wellformed_protocol'_def wellformed_protocol''_def wellformed_protocol_SMP_set_def

  have attack_notin_FP: "attack⟨n⟩ ∉ set FP"
    using attack_notin_fixpoint[unfolded attack_notin_fixpoint_def]
    unfolding list_all_iff FP_def by force

  have 1: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
    using wellformed_fixpoint
    unfolding wellformed_fixpoint_def wf_trms_code[symmetric] Let_def TI_def
              list_all_iff member_def case_prod_unfold
              wellformed_term_implication_graph_def
    by auto

  have 0: "wf_trms (set FP)"
    and 2: "∀(a,b) ∈ set TI. a ≠ b"
    and 3: "snd ` set TI ⊆ set OCC"
    and 4: "∀t ∈ set FP. ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
    and 5: "ground (set FP)"
    using wellformed_fixpoint
    unfolding wellformed_fixpoint_def wf_trm_code[symmetric] is_Abs_def the_Abs_def
              list_all_iff Let_def case_prod_unfold set_map FP_def OCC_def TI_def
              wellformed_fixpoint'_def wellformed_term_implication_graph_def
    by (fast, fast, blast, fastforce, simp)

  have 8: "finite (set N)"
    and 9: "has_all_wt_instances_of Γ (⋃T ∈ set (filter g P). trms_transaction T) (set N)"
    and 10: "tfr_set (set N)"
    and 11: "∀T ∈ set (filter f P). list_all tfr_sstp (unlabel (transaction_strand T))"
    and 12: "∀T ∈ set (filter f P). admissible_transaction T"
    using wellformed_protocol[unfolded wellformed_prot_defs]
          tfr_set_if_comp_tfr_set[of "set N"]
    unfolding Let_def list_all_iff wf_trms_code[symmetric] tfr_sstp_is_comp_tfr_sstp[symmetric]
              has_all_wt_instances_of_def f_def[symmetric]
    by (fast, fastforce, fast, fastforce, fast)

  have 13: "wf_trms (set N)"
    using wellformed_protocol[unfolded wellformed_prot_defs]
          finite_SMP_representationD
    unfolding wf_trm_code[symmetric] wf_trms'_def comp_tfr_set_def list_all_iff Let_def by fast

  have 14: "has_initial_value_producing_transaction (filter g P)"
    using wellformed_protocol has_initial_value_producing_transaction_update_send_ex_filter
```

```
    unfolding wellformed_protocol'_def Let_def g_def by blast

  note TI0 = trancl_eqI'[OF 1 2]

  have "analyzed (timpl_closure_set (set FP) (set TI))"
    using analyzed_fixpoint[unfolded analyzed_fixpoint_def]
          analyzed_closed_mod_timpls_is_analyzed_timpl_closure_set[OF TI0 0]
    unfolding FP_def TI_def
    by force
  note FP0 = this 0 5

  note OCC0 = funs_term_OCC_TI_subset(1)[OF 4 3]
              timpl_closure_set_supset'[OF funs_term_OCC_TI_subset(2)[OF 4 3]]

  note M0 = 9 8 10 13

  have "f T ⟷ g T" when T: "T ∈ set P" for T
  proof -
    have *: "list_all stateful_strand_step.is_Receive (unlabel (transaction_receive T))"
            "list_all is_Check_or_Assignment (unlabel (transaction_checks T))"
            "list_all is_Update (unlabel (transaction_updates T))"
            "list_all stateful_strand_step.is_Send (unlabel (transaction_send T))"
      using T wellformed_protocol
      unfolding wellformed_protocol_def wellformed_prot_defs Let_def list_all_iff
      by (fast, fast, fast, fast)

    show ?thesis
      using transaction_updates_send_ex_iff[OF *]
      unfolding f_def g_def by (metis (no_types, lifting) list_ex_cong)
  qed
  hence 15: "∀T ∈ set (filter g P). list_all tfr_sstp (unlabel (transaction_strand T))"
    and 16: "∀T ∈ set (filter g P). admissible_transaction T"
    using 11 12 by auto

  have "list_all (transaction_check (FP, OCC, TI)) (map add_occurs_msgs (filter g P))"
    using transactions_covered[unfolded protocol_covered_by_fixpoint_def]
          transaction_check_trivial_case[of _ FP_OCC_TI]
    unfolding FP_def OCC_def TI_def list_all_iff Let_def case_prod_unfold
    by auto
  note P0 = 16 15 14 this attack_notin_FP

  show ?thesis
    using prot_secure_if_fixpoint_covered[OF FP0 OCC0 TI0 M0 P0]
          reachable_constraints_secure_if_filter_secure_case[unfolded g_def[symmetric]]
    by fast
qed

lemma prot_secure_if_prot_checks_coverage_rcv:
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
  assumes attack_notin_fixpoint: "attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered: "protocol_covered_by_fixpoint_coverage_rcv FP_OCC_TI P"
    and analyzed_fixpoint: "analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol: "wellformed_protocol' P N"
    and wellformed_fixpoint: "wellformed_fixpoint FP_OCC_TI"
  shows "∀𝒜 ∈ reachable_constraints P. ∄ℐ. constraint_model ℐ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
using prot_secure_if_prot_checks[
        OF attack_notin_fixpoint _
           analyzed_fixpoint wellformed_protocol wellformed_fixpoint]
      protocol_covered_by_fixpoint_if_protocol_covered_by_fixpoint_coverage_rcv[
        OF _ wellformed_fixpoint transactions_covered]
      wellformed_protocol[unfolded wellformed_protocol'_def]
by blast
```

```
end

locale secure_stateful_protocol =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
    and P_SMP::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes attack_notin_fixpoint: "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered: "pm.protocol_covered_by_fixpoint FP_OCC_TI P"
    and analyzed_fixpoint: "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol: "pm.wellformed_protocol' P P_SMP"
    and wellformed_fixpoint: "pm.wellformed_fixpoint FP_OCC_TI"
begin

theorem protocol_secure:
  "∀𝒜 ∈ pm.reachable_constraints P. ∄ℐ. pm.constraint_model ℐ (𝒜@[(l, send⟨[attack⟨n⟩]⟩)])"
by (rule pm.prot_secure_if_prot_checks[OF
           attack_notin_fixpoint transactions_covered
           analyzed_fixpoint wellformed_protocol wellformed_fixpoint])

corollary protocol_welltyped_secure:
  "∀𝒜 ∈ pm.reachable_constraints P. ∄ℐ. pm.welltyped_constraint_model ℐ (𝒜@[(l,
send⟨[attack⟨n⟩]⟩)])"
using protocol_secure unfolding pm.welltyped_constraint_model_def by fast

end

locale secure_stateful_protocol' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
  assumes attack_notin_fixpoint': "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered': "pm.protocol_covered_by_fixpoint FP_OCC_TI P"
    and analyzed_fixpoint': "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol': "pm.wellformed_protocol P"
    and wellformed_fixpoint': "pm.wellformed_fixpoint FP_OCC_TI"
begin

sublocale secure_stateful_protocol
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P
  FP_OCC_TI
  "let f = λM. remdups (concat (map subterms_list M@map (fst ∘ pm.Ana) M));
       N0 = remdups (concat (map (trms_list_sst ∘ unlabel ∘ transaction_strand) P))
   in while (λA. set (f A) ≠ set A) f N0"
apply unfold_locales
using attack_notin_fixpoint' transactions_covered' analyzed_fixpoint'
```

```
    wellformed_protocol'[unfolded pm.wellformed_protocol_def Let_def] wellformed_fixpoint'
unfolding Let_def by blast+

end

locale secure_stateful_protocol'' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
  assumes checks: "let FPT = pm.compute_fixpoint_fun P
    in pm.attack_notin_fixpoint FPT ∧ pm.protocol_covered_by_fixpoint FPT P ∧
       pm.analyzed_fixpoint FPT ∧ pm.wellformed_protocol P ∧ pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol'
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P "pm.compute_fixpoint_fun P"
using checks[unfolded Let_def case_prod_unfold] by unfold_locales meson+

end

locale secure_stateful_protocol''' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
    and P_SMP::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes checks': "let P' = P; FPT = FP_OCC_TI; P'_SMP = P_SMP
                    in pm.attack_notin_fixpoint FPT ∧
                       pm.protocol_covered_by_fixpoint FPT P' ∧
                       pm.analyzed_fixpoint FPT ∧
                       pm.wellformed_protocol' P' P'_SMP ∧
                       pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P FP_OCC_TI P_SMP
using checks'[unfolded Let_def case_prod_unfold] by unfold_locales meson+

end

locale secure_stateful_protocol'''' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
```

```
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
  assumes checks'': "let P' = P; FPT = FP_OCC_TI
                     in pm.attack_notin_fixpoint FPT ∧
                        pm.protocol_covered_by_fixpoint FPT P' ∧
                        pm.analyzed_fixpoint FPT ∧
                        pm.wellformed_protocol P' ∧
                        pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol'
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P FP_OCC_TI
using checks''[unfolded Let_def case_prod_unfold] by unfold_locales meson+

end

locale secure_stateful_protocol_coverage_rcv =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
    and P_SMP::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes attack_notin_fixpoint_coverage_rcv: "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered_coverage_rcv: "pm.protocol_covered_by_fixpoint_coverage_rcv FP_OCC_TI P"
    and analyzed_fixpoint_coverage_rcv: "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol_coverage_rcv: "pm.wellformed_protocol' P P_SMP"
    and wellformed_fixpoint_coverage_rcv: "pm.wellformed_fixpoint FP_OCC_TI"
begin

sublocale secure_stateful_protocol
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P
  FP_OCC_TI P_SMP
using pm.protocol_covered_by_fixpoint_if_protocol_covered_by_fixpoint_coverage_rcv[
        OF _ wellformed_fixpoint_coverage_rcv transactions_covered_coverage_rcv]
      attack_notin_fixpoint_coverage_rcv analyzed_fixpoint_coverage_rcv
      wellformed_protocol_coverage_rcv wellformed_fixpoint_coverage_rcv
      wellformed_protocol_coverage_rcv[unfolded pm.wellformed_protocol'_def]
by unfold_locales meson+

end

locale secure_stateful_protocol_coverage_rcv' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
  assumes attack_notin_fixpoint_coverage_rcv': "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered_coverage_rcv': "pm.protocol_covered_by_fixpoint_coverage_rcv FP_OCC_TI P"
```

```
    and analyzed_fixpoint_coverage_rcv': "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol_coverage_rcv': "pm.wellformed_protocol P"
    and wellformed_fixpoint_coverage_rcv': "pm.wellformed_fixpoint FP_OCC_TI"
begin

sublocale secure_stateful_protocol_coverage_rcv
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P
  FP_OCC_TI
  "let f = λM. remdups (concat (map subterms_list M@map (fst ∘ pm.Ana) M));
       N0 = remdups (concat (map (trms_list_sst ∘ unlabel ∘ transaction_strand) P))
   in while (λA. set (f A) ≠ set A) f N0"
apply unfold_locales
using attack_notin_fixpoint_coverage_rcv' transactions_covered_coverage_rcv'
analyzed_fixpoint_coverage_rcv'
      wellformed_protocol_coverage_rcv'[unfolded pm.wellformed_protocol_def Let_def]
wellformed_fixpoint_coverage_rcv'
unfolding Let_def by blast+

end

locale secure_stateful_protocol_coverage_rcv'' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
  assumes checks_coverage_rcv: "let FPT = pm.compute_fixpoint_fun P
    in pm.attack_notin_fixpoint FPT ∧ pm.protocol_covered_by_fixpoint_coverage_rcv FPT P ∧
       pm.analyzed_fixpoint FPT ∧ pm.wellformed_protocol P ∧ pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol_coverage_rcv'
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P "pm.compute_fixpoint_fun P"
using checks_coverage_rcv[unfolded Let_def case_prod_unfold] by unfold_locales meson+

end

locale secure_stateful_protocol_coverage_rcv''' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
    and P_SMP::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes checks_coverage_rcv': "let P' = P; FPT = FP_OCC_TI; P'_SMP = P_SMP
                        in pm.attack_notin_fixpoint FPT ∧
                           pm.protocol_covered_by_fixpoint_coverage_rcv FPT P' ∧
                           pm.analyzed_fixpoint FPT ∧
                           pm.wellformed_protocol' P' P'_SMP ∧
                           pm.wellformed_fixpoint FPT"
begin
```

```
sublocale secure_stateful_protocol_coverage_rcv
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P FP_OCC_TI P_SMP
using checks_coverage_rcv'[unfolded Let_def case_prod_unfold] by unfold_locales meson+

end

locale secure_stateful_protocol_coverage_rcv'''' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun,'atom,'sets,'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"
  assumes checks_coverage_rcv'': "let P' = P; FPT = FP_OCC_TI
                        in pm.attack_notin_fixpoint FPT ∧
                           pm.protocol_covered_by_fixpoint_coverage_rcv FPT P' ∧
                           pm.analyzed_fixpoint FPT ∧
                           pm.wellformed_protocol P' ∧
                           pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol_coverage_rcv'
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P FP_OCC_TI
using checks_coverage_rcv''[unfolded Let_def case_prod_unfold] by unfold_locales meson+

end
```

3.6.9 Automatic Protocol Composition

```
context stateful_protocol_model
begin

definition welltyped_leakage_free_protocol where
  "welltyped_leakage_free_protocol S P ≡
    let f = λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}};
        Sec = (f (set S)) - {m. {} ⊢_c m}
    in ∀𝒜 ∈ reachable_constraints P. ∄ℐ_τ s.
      (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜) ∧ s ∈ Sec - declassified_lsst 𝒜 ℐ_τ ∧
      welltyped_constraint_model ℐ_τ (𝒜@[⟨⋆, send⟨[s]⟩⟩])"

definition wellformed_composable_protocols where
  "wellformed_composable_protocols (P::('fun,'atom,'sets,'lbl) prot list) N ≡
    let
      Ts = concat P;
      steps = remdups (concat (map transaction_strand Ts));
      MP0 = ⋃T ∈ set Ts. trms_transaction T ∪ pair' Pair ` setops_transaction T
    in
      list_all (wf_trm' arity) N ∧
      has_all_wt_instances_of Γ MP0 (set N) ∧
      comp_tfr_set arity Ana Γ (set N) ∧
      list_all (comp_tfr_sstp Γ Pair ∘ snd) steps ∧
      list_all admissible_transaction_terms Ts ∧
      list_all (list_all (λx. Γ_v x = TAtom Value ∨ (is_Var (Γ_v x) ∧ is_Atom (the_Var (Γ_v x)))) ∘
                transaction_fresh)
               Ts ∧
      list_all (λT. ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γ_v x) Ts ∧
      list_all (λT. ∀x ∈ vars_transaction T. ∀f ∈ funs_term (Γ_v x). f ≠ Pair ∧ f ≠ OccursFact)
               Ts ∧
```

```
    list_all (list_all (λs. is_Send (snd s) ∧ length (the_msgs (snd s)) = 1 ∧
                        is_Fun_Attack (hd (the_msgs (snd s))) ⟶
                          the_Attack_label (the_Fun (hd (the_msgs (snd s)))) = fst s) ∘
              transaction_strand)
             Ts ∧
    list_all (λr. (∃f ∈ ⋃(funs_term ` (trms_sstp (snd r))). f = OccursFact ∨ f = OccursSec) ⟶
                  (is_Receive (snd r) ∨ is_Send (snd r)) ∧ fst r = ⋆ ∧
                  (∀t ∈ set (the_msgs (snd r)).
                    (OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t) ⟶
                      is_Fun t ∧ length (args t) = 2 ∧ t = occurs (args t ! 1) ∧
                      is_Var (args t ! 1) ∧ (Γ (args t ! 1) = TAtom Value)))
             steps"

definition wellformed_composable_protocols' where
  "wellformed_composable_protocols' (P::('fun,'atom,'sets,'lbl) prot list) ≡
    let
      Ts = concat P
    in
      list_all wellformed_transaction Ts ∧
      list_all (list_all
                 (λp. let (x,cs) = p in
                        is_Var (Γv x) ∧ is_Atom (the_Var (Γv x)) ∧
                        (∀c ∈ cs. Γv x = Γ (Fun (Fu c) []::('fun,'atom,'sets,'lbl) prot_term))) ∘
              (λT. transaction_decl T ()))
             Ts"

definition composable_protocols where
  "composable_protocols (P::('fun,'atom,'sets,'lbl) prot list) Ms S ≡
    let
      steps = concat (map transaction_strand (concat P));
      M_fun = (λl. case find ((=) l ∘ fst) Ms of Some M ⇒ set (snd M) | None ⇒ {})
    in comp_par_comp_lsst public arity Ana Γ Pair steps M_fun (set S)"

lemma composable_protocols_par_comp_constr:
  fixes S f
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Sec ≡ (f (set S)) - {m. intruder_synth {} m}"
  assumes Ps_pc: "wellformed_composable_protocols Ps N"
                 "wellformed_composable_protocols' Ps"
                 "composable_protocols Ps Ms S"
  shows "∀𝒜 ∈ reachable_constraints (concat Ps). ∀ℐ. constraint_model ℐ 𝒜 ⟶
           (∃ℐτ. welltyped_constraint_model ℐτ 𝒜 ∧
                 ((∀n. welltyped_constraint_model ℐτ (proj n 𝒜)) ∨
                  (∃𝒜' l t. prefix 𝒜' 𝒜 ∧ suffix [(l, receive⟨t⟩)] 𝒜' ∧
                            strand_leaks_lsst 𝒜' Sec ℐτ)))"
    (is "∀𝒜 ∈ _. ∀_. _ ⟶ ?Q 𝒜 ℐ")
proof (intro allI ballI impI)
  fix 𝒜 ℐ
  assume 𝒜: "𝒜 ∈ reachable_constraints (concat Ps)" and ℐ: "constraint_model ℐ 𝒜"

  let ?Ts = "concat Ps"
  let ?steps = "concat (map transaction_strand ?Ts)"
  let ?MP0 = "⋃T ∈ set ?Ts. trms_transaction T ∪ pair' Pair ` setops_transaction T"
  let ?M_fun = "λl. case find ((=) l ∘ fst) Ms of Some M ⇒ set (snd M) | None ⇒ {}"

  have M:
      "has_all_wt_instances_of Γ ?MP0 (set N)"
      "finite (set N)" "tfr_set (set N)" "wf_trms (set N)"
    using Ps_pc tfr_set_if_comp_tfr_set[of "set N"]
    unfolding composable_protocols_def wellformed_composable_protocols_def
              Let_def list_all_iff wf_trm_code[symmetric]
    by fast+
```

```
  have P:
      "∀T ∈ set ?Ts. wellformed_transaction T"
      "∀T ∈ set ?Ts. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set ?Ts. list_all tfr_sstp (unlabel (transaction_strand T))"
      "comp_par_comp_lsst public arity Ana Γ Pair ?steps ?M_fun (set S)"
    using Ps_pc tfr_sstp_is_comp_tfr_sstp
    unfolding wellformed_composable_protocols_def wellformed_composable_protocols'_def
              composable_protocols_def Let_def list_all_iff unlabel_def wf_trms_code[symmetric]
              admissible_transaction_terms_def
    by (meson, meson, fastforce, blast)

  show "?Q 𝒜 ℐ"
    using reachable_constraints_par_comp_constr[OF M P 𝒜 ℐ]
    unfolding Sec_def f_def by fast
qed

context
begin
private lemma reachable_constraints_no_leakage_alt_aux:
  fixes P lbls L
  defines "lbls ≡ λT. map (the_LabelN o fst) (filter (Not o has_LabelS) (transaction_strand T))"
    and "L ≡ set (remdups (concat (map lbls P)))"
  assumes l: "l ∉ L"
  shows "map (transaction_proj l) P = map transaction_star_proj P"
proof -
  have 0: "¬list_ex (has_LabelN l) (transaction_strand T)" when "T ∈ set P" for T
    using that l unfolding L_def lbls_def list_ex_iff by force

  have 1: "¬list_ex (has_LabelN l) (transaction_strand T)"
    when T: "T ∈ set (map (transaction_proj l) P)" for T
  proof -
    obtain T' where T': "T' ∈ set P" "T = transaction_proj l T'" using T by auto
    show ?thesis
      using T'(2) 0[OF T'(1)] proj_set_subset[of l "transaction_strand T'"]
            transaction_strand_proj[of l T']
      unfolding list_ex_iff by fastforce
  qed

  have "list_all has_LabelS (transaction_strand T)"
    when "T ∈ set (map (transaction_proj l) P)" for T
    using that 1[OF that] transaction_proj_idem[of l]
          transaction_strand_proj[of l "transaction_proj l T"]
          has_LabelS_proj_iff_not_has_LabelN[of l "transaction_strand (transaction_proj l T)"]
    by (metis (no_types) ex_map_conv)
  thus ?thesis
    using transaction_star_proj_ident_iff transaction_proj_member
          transaction_star_proj_negates_transaction_proj(1)
    by (metis (mono_tags, lifting) map_eq_conv)
qed

private lemma reachable_constraints_star_no_leakage:
  fixes Sec P lbls k
  defines "no_leakage ≡ λ𝒜. ∄ℐ_τ 𝒜' s.
      prefix 𝒜' 𝒜 ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜') ∧ s ∈ Sec - declassified_lsst 𝒜' ℐ_τ ∧
      welltyped_constraint_model ℐ_τ (𝒜'@[(k,send⟨[s]⟩)])"
  assumes Sec: "∀s ∈ Sec. ¬{} ⊢_c s" "ground Sec"
  shows "∀𝒜 ∈ reachable_constraints (map transaction_star_proj P). no_leakage 𝒜"
proof
  fix A assume A: "A ∈ reachable_constraints (map transaction_star_proj P)"

  have A': "∀(l,a) ∈ set A. l = ⋆"
    using reachable_constraints_preserves_labels[OF A] transaction_star_proj_has_star_labels
    unfolding list_all_iff by fastforce
```

```
  show "no_leakage A"
    using constr_sem_stateful_star_proj_no_leakage[OF Sec(2) A']
          unlabel_append[of A] singleton_lst_proj(4)[of k]
    unfolding no_leakage_def welltyped_constraint_model_def constraint_model_def by fastforce
qed

private lemma reachable_constraints_no_leakage_alt:
  fixes Sec P lbls k
  defines "no_leakage ≡ λ𝒜. ∄ℐτ 𝒜' s.
      prefix 𝒜' 𝒜 ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜') ∧ s ∈ Sec - declassified_lsst 𝒜' ℐτ ∧
      welltyped_constraint_model ℐτ (𝒜'@[(k,send⟨[s]⟩)])"
    and "lbls ≡ λT. map (the_LabelN o fst) (filter (Not o has_LabelS) (transaction_strand T))"
    and "L ≡ set (remdups (concat (map lbls P)))"
  assumes Sec: "∀s ∈ Sec. ¬{} ⊢c s" "ground Sec"
    and lbl: "∀l ∈ L. ∀𝒜 ∈ reachable_constraints (map (transaction_proj l) P). no_leakage 𝒜"
  shows "∀l. ∀𝒜 ∈ reachable_constraints (map (transaction_proj l) P). ∄ℐτ 𝒜'.
              interpretation_subst ℐτ ∧ wt_subst ℐτ ∧ wf_trms (subst_range ℐτ) ∧
              prefix 𝒜' 𝒜 ∧ (∃l' ts. suffix [(l', receive⟨ts⟩)] 𝒜') ∧ strand_leaks_lsst 𝒜' Sec ℐτ"
proof (intro allI ballI)
  fix l 𝒜
  assume 𝒜: "𝒜 ∈ reachable_constraints (map (transaction_proj l) P)"

  let ?Q = "λℐτ 𝒜'.
    interpretation_subst ℐτ ∧ wt_subst ℐτ ∧ wf_trms (subst_range ℐτ) ∧
    prefix 𝒜' 𝒜 ∧ (∃l' t. suffix [(l', receive⟨t⟩)] 𝒜') ∧ strand_leaks_lsst 𝒜' Sec ℐτ"

  show "∄ℐτ 𝒜'. ?Q ℐτ 𝒜'"
  proof
    assume "∃ℐτ 𝒜'. ?Q ℐτ 𝒜'"
    then obtain ℐτ 𝒜' t n l' ts' where
        ℐτ: "interpretation_subst ℐτ" "wt_subst ℐτ" "wf_trms (subst_range ℐτ)" and
        𝒜': "prefix 𝒜' 𝒜" "suffix [(l', receive⟨ts'⟩)] 𝒜'" and
        t: "t ∈ Sec - declassified_lsst 𝒜' ℐτ" and
        n: "constr_sem_stateful ℐτ (proj_unl n 𝒜'@[send⟨[t]⟩])"
      unfolding strand_leaks_lsst_def by blast
    hence 0: "welltyped_constraint_model ℐτ (proj n 𝒜'@[(m,send⟨[t]⟩)])" for m
      unfolding welltyped_constraint_model_def constraint_model_def by fastforce

    have t_Sec: "¬{} ⊢c t" "t · ℐτ = t"
      using t Sec subst_ground_ident[of t ℐτ] by auto

    obtain B k' s where B:
        "constr_sem_stateful ℐτ (proj_unl n B@[send⟨[t]⟩])"
        "prefix (proj n B) (proj n 𝒜)" "suffix [(k', receive⟨s⟩)] (proj n B)"
        "t ∈ Sec - declassified_lsst (proj n B) ℐτ"
      using constr_sem_stateful_proj_priv_term_prefix_obtain[OF 𝒜'(1) n t t_Sec]
      by metis
    hence 1: "welltyped_constraint_model ℐτ (proj n B@[(m,send⟨[t]⟩)])" for m
      using 0 unfolding welltyped_constraint_model_def constraint_model_def by fastforce

    note 2 = reachable_constraints_transaction_proj_proj_eq
    note 3 = reachable_constraints_transaction_proj_star_proj
    note 4 = reachable_constraints_no_leakage_alt_aux

    note star_case = 0 t t_Sec(1) reachable_constraints_star_no_leakage[OF Sec]
                  𝒜'(2) 3[OF 𝒜] prefix_proj(1)[OF 𝒜'(1)]
                  suffix_proj(1)[OF 𝒜'(2)] declassified_lsst_proj_eq

    note lbl_case = 0 t(1) 𝒜 𝒜' lbl 2(2)[OF 𝒜 𝒜'(1)]

    show False
    proof (cases "l = n")
```

```
      case True thus ?thesis
      proof (cases "l ∈ L")
        case False
        hence "map (transaction_proj l) P = map transaction_star_proj P"
          using 4 unfolding L_def lbls_def by fast
        thus ?thesis
          using lbl_case(1-4,7) star_case(4,5) True by metis
      qed (metis lbl_case no_leakage_def)
    next
      case False
      hence "no_leakage (proj n 𝒜)" using star_case(4,6) unfolding no_leakage_def by fast
      thus ?thesis by (metis B(2-4) 1 no_leakage_def)
    qed
  qed
qed

private lemma reachable_constraints_no_leakage_alt'_aux1:
  fixes P::"('a,'b,'c,'d) prot_transaction list"
  defines "f ≡ list_all ((list_all (Not ∘ has_LabelS)) ∘ tl ∘ transaction_send)"
  assumes P: "f P"
  shows "f (map (transaction_proj l) P)"
    and "f (map transaction_star_proj P)"
proof -
  let ?g = "λT. tl (transaction_send T)"
  have "set (?g (transaction_proj l T)) ⊆ set (?g T)" (is "?A ⊆ ?C")
    and "set (?g (transaction_star_proj T)) ⊆ set (?g T)" (is "?B ⊆ ?C")
    for T::"('a,'b,'c,'d) prot_transaction"
  proof -
    obtain T1 T2 T3 T4 T5 T6 where T: "T = Transaction T1 T2 T3 T4 T5 T6" by (cases T) simp
    have "transaction_send (transaction_proj l T) = proj l (transaction_send T)"
         "transaction_send (transaction_star_proj T) = filter has_LabelS (transaction_send T)"
      using transaction_proj.simps[of l T1 T2 T3 T4 T5 T6]
            transaction_star_proj.simps[of T1 T2 T3 T4 T5 T6]
      unfolding T proj_def Let_def by auto
    hence "set (?g (transaction_proj l T)) ⊆ set (proj l (?g T))"
          "set (?g (transaction_star_proj T)) ⊆ set (filter has_LabelS (?g T))"
      unfolding proj_def
      by (metis (no_types, lifting) filter.simps(2) list.collapse list.sel(2,3)
                                    list.set_sel(2) subsetI)+
    thus "?A ⊆ ?C" "?B ⊆ ?C" using T unfolding proj_def by auto
  qed
  thus "f (map (transaction_proj l) P)" "f (map transaction_star_proj P)"
    using P unfolding f_def list_all_iff by fastforce+
qed

private lemma reachable_constraints_no_leakage_alt'_aux2:
  fixes P
  defines "f ≡ λT.
    list_all is_Receive (unlabel (transaction_receive T)) ∧
    list_all is_Check_or_Assignment (unlabel (transaction_checks T)) ∧
    list_all is_Update (unlabel (transaction_updates T)) ∧
    list_all is_Send (unlabel (transaction_send T))"
  assumes P: "list_all f P"
  shows "list_all f (map (transaction_proj l) P)" (is ?A)
    and "list_all f (map transaction_star_proj P)" (is ?B)
proof -
  have "f (transaction_proj l T)" (is ?A')
    and "f (transaction_star_proj T)" (is ?B')
    when T_in: "T ∈ set P" for T
  proof -
    obtain T1 T2 T3 T4 T5 T6 where T: "T = Transaction T1 T2 T3 T4 T5 T6" by (cases T)
    have "f T" using P T_in unfolding list_all_iff by simp
    thus ?A' ?B'
```

```
      unfolding f_def T unlabel_def proj_def Let_def list_all_iff
                transaction_proj.simps[of l T1 T2 T3 T4 T5 T6]
                transaction_star_proj.simps[of T1 T2 T3 T4 T5 T6]
      by auto
  qed
  thus ?A ?B unfolding list_all_iff by auto
qed

private lemma reachable_constraints_no_leakage_alt':
  fixes Sec P lbls k
  defines "no_leakage ≡ λ𝒜. ∄ℐτ 𝒜' s.
      prefix 𝒜' 𝒜 ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜') ∧ s ∈ Sec - declassified_lsst 𝒜' ℐτ ∧
      welltyped_constraint_model ℐτ (𝒜'@[(k,send⟨[s]⟩)])"
    and "no_leakage' ≡ λ𝒜. ∄ℐτ s.
      (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜) ∧ s ∈ Sec - declassified_lsst 𝒜 ℐτ ∧
      welltyped_constraint_model ℐτ (𝒜@[(k,send⟨[s]⟩)])"
  assumes P: "list_all wellformed_transaction P"
           "list_all ((list_all (Not ∘ has_LabelS)) ∘ tl ∘ transaction_send) P"
    and Sec: "∀s ∈ Sec. ¬{} ⊢c s" "ground Sec"
    and lbl: "∀l ∈ L. ∀𝒜 ∈ reachable_constraints (map (transaction_proj l) P). no_leakage' 𝒜"
  shows "∀l ∈ L. ∀𝒜 ∈ reachable_constraints (map (transaction_proj l) P). no_leakage 𝒜" (is ?A)
    and "∀𝒜 ∈ reachable_constraints (map transaction_star_proj P). no_leakage 𝒜" (is ?B)
proof -
  define f where "f ≡ λT::('fun,'atom,'sets,'lbl) prot_transaction.
    list_all is_Receive (unlabel (transaction_receive T)) ∧
    list_all is_Check_or_Assignment (unlabel (transaction_checks T)) ∧
    list_all is_Update (unlabel (transaction_updates T)) ∧
    list_all is_Send (unlabel (transaction_send T))"

  define g where "(g::('fun,'atom,'sets,'lbl) prot_transaction ⇒ bool) ≡
    list_all (Not ∘ has_LabelS) ∘ tl ∘ transaction_send"

  have P': "list_all f P"
    using P(1) unfolding wellformed_transaction_def f_def list_all_iff by fastforce

  note 0 = reachable_constraints_no_leakage_alt'_aux1[OF P(2), unfolded g_def[symmetric]]

  note 1 = reachable_constraints_no_leakage_alt'_aux2[
          OF P'[unfolded f_def], unfolded f_def[symmetric]]

  note 2 = reachable_constraints_aligned_prefix_ex[unfolded f_def[symmetric] g_def[symmetric]]

  have 3: "∀𝒜 ∈ reachable_constraints (map transaction_star_proj P). no_leakage' 𝒜"
    using reachable_constraints_star_no_leakage[OF Sec] unfolding no_leakage'_def by blast

  show ?A
  proof (intro ballI)
    fix l 𝒜 assume l: "l ∈ L" and 𝒜: "𝒜 ∈ reachable_constraints (map (transaction_proj l) P)"
    show "no_leakage 𝒜"
    proof (rule ccontr)
      assume "¬no_leakage 𝒜"
      then obtain ℐτ 𝒜' s where 𝒜':
          "prefix 𝒜' 𝒜" "∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜'" "s ∈ Sec - declassified_lsst 𝒜' ℐτ"
          "welltyped_constraint_model ℐτ (𝒜'@[(k, send⟨[s]⟩)])"
        unfolding no_leakage_def by blast

      have s: "¬{} ⊢c s" "fv s = {}" using 𝒜'(3) Sec by auto

      have ℐτ: "constr_sem_stateful ℐτ (unlabel 𝒜'@[send⟨[s]⟩])"
            "wt_subst ℐτ" "interpretation_subst ℐτ" "wf_trms (subst_range ℐτ)"
        using 𝒜'(4) unfolding welltyped_constraint_model_def constraint_model_def by auto

      show False
```

```
        using 2[OF 1(1) 0(1) s 𝒜 𝒜'(1,2) ℐ_τ(1)] 1 lbl 𝒜'(3) ℐ_τ(2,3,4)
              singleton_lst_proj(4)[of k "send⟨[s]⟩"] unlabel_append[of _ "[(k, send⟨[s]⟩)]"]
        unfolding no_leakage'_def welltyped_constraint_model_def constraint_model_def by metis
    qed
  qed

  show ?B
  proof (intro ballI)
    fix 𝒜 assume 𝒜: "𝒜 ∈ reachable_constraints (map transaction_star_proj P)"
    show "no_leakage 𝒜"
    proof (rule ccontr)
      assume "¬no_leakage 𝒜"
      then obtain ℐ_τ 𝒜' s where 𝒜':
          "prefix 𝒜' 𝒜" "∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜'" "s ∈ Sec - declassified_lsst 𝒜' ℐ_τ"
          "welltyped_constraint_model ℐ_τ (𝒜'@[(k, send⟨[s]⟩)])"
        unfolding no_leakage_def by blast

      have s: "¬{} ⊢_c s" "fv s = {}" using 𝒜'(3) Sec by auto

      have ℐ_τ: "constr_sem_stateful ℐ_τ (unlabel 𝒜'@[send⟨[s]⟩])"
              "wt_subst ℐ_τ" "interpretation_subst ℐ_τ" "wf_trms (subst_range ℐ_τ)"
        using 𝒜'(4) unfolding welltyped_constraint_model_def constraint_model_def by auto

      show False
        using 2[OF 1(2) 0(2) s 𝒜 𝒜'(1,2) ℐ_τ(1)] 3 𝒜'(3) ℐ_τ(2,3,4)
              singleton_lst_proj(4)[of k "send⟨[s]⟩"] unlabel_append[of _ "[(k, send⟨[s]⟩)]"]
        unfolding no_leakage'_def welltyped_constraint_model_def constraint_model_def by metis
    qed
  qed
qed

lemma composable_protocols_par_comp_prot_alt:
  fixes S f Sec lbls Ps
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Sec ≡ (f (set S)) - {m. {} ⊢_c m}"
    and "lbls ≡ λT. map (the_LabelN o fst) (filter (Not o has_LabelS) (transaction_strand T))"
    and "L ≡ set (remdups (concat (map lbls (concat Ps))))"
    and "no_leakage ≡ λ𝒜. ∄ℐ_τ 𝒜' s.
      prefix 𝒜' 𝒜 ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜') ∧ s ∈ Sec - declassified_lsst 𝒜' ℐ_τ ∧
      welltyped_constraint_model ℐ_τ (𝒜'@[⟨⋆, send⟨[s]⟩⟩])"
  assumes Ps_pc: "wellformed_composable_protocols Ps N"
                 "wellformed_composable_protocols' Ps"
                 "composable_protocols Ps Ms S"
    and component_secure:
          "∀𝒜 ∈ reachable_constraints (map (transaction_proj l) (concat Ps)). ∄ℐ.
              welltyped_constraint_model ℐ (𝒜@[⟨l, send⟨[attack⟨ln l⟩]⟩⟩])"
    and no_leakage:
          "∀l ∈ L. ∀𝒜 ∈ reachable_constraints (map (transaction_proj l) (concat Ps)). no_leakage 𝒜"
  shows "∀𝒜 ∈ reachable_constraints (concat Ps). ∄ℐ.
            constraint_model ℐ (𝒜@[⟨l, send⟨[attack⟨ln l⟩]⟩⟩])"
proof
  fix 𝒜
  assume 𝒜: "𝒜 ∈ reachable_constraints (concat Ps)"
  let ?att = "[⟨l, send⟨[attack⟨ln l⟩]⟩⟩]"

  define Q where "Q ≡ λℐ_τ. interpretation_subst ℐ_τ ∧ wt_subst ℐ_τ ∧ wf_trms (subst_range ℐ_τ)"

  define R where "R ≡ λ𝒜 ℐ_τ.
    ∃𝒜' l t. prefix 𝒜' 𝒜 ∧ suffix [(l, receive⟨t⟩)] 𝒜' ∧ strand_leaks_lsst 𝒜' Sec ℐ_τ"

  define M where "M ≡ ⋃T∈set (concat Ps). trms_transaction T ∪ pair' Pair ` setops_transaction T"

  have Sec: "∀s ∈ Sec. ¬{} ⊢_c s" "ground Sec" unfolding Sec_def f_def by auto
```

```
have par_comp':
    "∀𝒜 ∈ reachable_constraints (concat Ps). ∀ℐ. constraint_model ℐ 𝒜 ⟶
       (∃ℐτ. welltyped_constraint_model ℐτ 𝒜 ∧
           ((∀n. welltyped_constraint_model ℐτ (proj n 𝒜)) ∨ R 𝒜 ℐτ))"
  using 𝒜 composable_protocols_par_comp_constr[OF Ps_pc] unfolding Sec_def f_def R_def by fast

have "∀l. ∀𝒜 ∈ reachable_constraints (map (transaction_proj l) (concat Ps)). ∄ℐτ. Q ℐτ ∧ R 𝒜 ℐτ"
  using reachable_constraints_no_leakage_alt[OF
          Sec no_leakage[unfolded no_leakage_def L_def lbls_def]]
  unfolding Q_def R_def by blast
hence no_leakage':
    "∀𝒜 ∈ reachable_constraints (concat Ps). ∄ℐτ. Q ℐτ ∧ R 𝒜 ℐτ"
  using reachable_constraints_component_leaks_if_composed_leaks[OF Sec, of "concat Ps"
          "λℐτ. interpretation_subst ℐτ ∧ wt_subst ℐτ ∧ wf_trms (subst_range ℐτ)"]
  unfolding Q_def R_def by blast

have M: "has_all_wt_instances_of Γ M (set N)" "finite (set N)" "tfr_set (set N)" "wf_trms (set N)"
 and P: "∀T ∈ set (concat Ps). wellformed_transaction T"
        "∀T ∈ set (concat Ps). admissible_transaction_terms T"
        "∀T ∈ set (concat Ps). ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γv x"
        "∀T ∈ set (concat Ps). ∀s ∈ set (transaction_strand T).
            is_Send (snd s) ∧ length (the_msgs (snd s)) = 1 ∧
            is_Fun_Attack (hd (the_msgs (snd s))) ⟶
              the_Attack_label (the_Fun (hd (the_msgs (snd s)))) = fst s"
        "∀T ∈ set (concat Ps). list_all tfr_sstp (unlabel (transaction_strand T))"
  using Ps_pc(1,2) tfr_set_if_comp_tfr_set tfr_sstp_is_comp_tfr_sstp
  unfolding wellformed_composable_protocols_def wellformed_composable_protocols'_def
            list_all_iff Let_def M_def wf_trms'_def wf_trms_code unlabel_def Γv_TAtom''(1,2)
  by (force, force, fast, fast, fast, fast, fast, simp, simp)

have P_fresh: "∀T ∈ set (concat Ps). ∀x ∈ set (transaction_fresh T).
                Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
    (is "∀T ∈ ?P. ∀x ∈ ?frsh T. ?Q x")
proof (intro ballI)
  fix T x assume T: "T ∈ ?P" "x ∈ ?frsh T"
  hence "Γv x = TAtom Value ∨ (is_Var (Γv x) ∧ is_Atom (the_Var (Γv x)))"
    using Ps_pc(1) unfolding wellformed_composable_protocols_def list_all_iff Let_def by fastforce
  thus "?Q x" by (metis prot_atom.is_Atom_def term.collapse(1))
qed

have P': "∀T ∈ set (concat Ps). wf_trms' arity (trms_transaction T)"
  using P(2) admissible_transaction_terms_def by fast

have "¬welltyped_constraint_model ℐ (𝒜@?att)" for ℐ
proof
  assume "welltyped_constraint_model ℐ (𝒜@?att)"
  hence ℐ: "welltyped_constraint_model ℐ 𝒜" "ik_lsst 𝒜 ·set ℐ ⊢ attack⟨ln 1⟩"
    using strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "unlabel ?att"]
          unlabel_append[of 𝒜 ?att]
    unfolding welltyped_constraint_model_def constraint_model_def by auto

  obtain ℐτ where ℐτ:
      "welltyped_constraint_model ℐτ 𝒜"
      "welltyped_constraint_model ℐτ (proj 1 𝒜)"
    using 𝒜 ℐ no_leakage' par_comp'
    unfolding Q_def welltyped_constraint_model_def constraint_model_def by metis

  have "⟨1, receive⟨[attack⟨ln 1⟩]⟩⟩ ∈ set 𝒜"
    using reachable_constraints_receive_attack_if_attack(3)[OF 𝒜 P(1-2) P_fresh P(3) ℐ P(4)]
    by auto
  hence "ik_lsst (proj 1 𝒜) ·set ℐτ ⊢ attack⟨ln 1⟩"
    using in_proj_set[of 1 "receive⟨[attack⟨ln 1⟩]⟩" 𝒜] in_ik_lsst_iff[of "attack⟨ln 1⟩" "proj 1 𝒜"]
```

```
          intruder_deduct.Axiom[of "attack⟨ln 1⟩" "ik_lsst (proj 1 𝒜) ·set ℐτ"]
      by fastforce
    hence "welltyped_constraint_model ℐτ (proj 1 𝒜@?att)"
      using ℐτ strand_sem_append_stateful[of "{}" "{}" "unlabel (proj 1 𝒜)" "unlabel ?att" ℐτ]
      unfolding welltyped_constraint_model_def constraint_model_def by auto
    thus False
      using component_secure reachable_constraints_transaction_proj[OF 𝒜, of 1] by simp
  qed
  thus "∄ℐ. constraint_model ℐ (𝒜@?att)"
    using reachable_constraints_typing_result'[OF M_def M P(1) P' P(5) 𝒜] by blast
qed

lemma composable_protocols_par_comp_prot:
  fixes S f Sec lbls Ps
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Sec ≡ (f (set S)) - {m. {} ⊢c m}"
    and "lbls ≡ λT. map (the_LabelN o fst) (filter (Not o has_LabelS) (transaction_strand T))"
    and "L ≡ set (remdups (concat (map lbls (concat Ps))))"
    and "no_leakage ≡ λ𝒜. ∄ℐτ s.
      (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜) ∧ s ∈ Sec - declassified_lsst 𝒜 ℐτ ∧
      welltyped_constraint_model ℐτ (𝒜@[⟨⋆, send⟨[s]⟩⟩])"
  assumes Ps_pc: "wellformed_composable_protocols Ps N"
                 "wellformed_composable_protocols' Ps"
                 "composable_protocols Ps Ms S"
                 "list_all ((list_all (Not ∘ has_LabelS)) ∘ tl ∘ transaction_send) (concat Ps)"
    and component_secure:
         "∀𝒜 ∈ reachable_constraints (map (transaction_proj l) (concat Ps)). ∄ℐ.
             welltyped_constraint_model ℐ (𝒜@[⟨l, send⟨[attack⟨ln l⟩]⟩⟩])"
    and no_leakage:
         "∀l ∈ L. ∀𝒜 ∈ reachable_constraints (map (transaction_proj l) (concat Ps)). no_leakage 𝒜"
  shows "∀𝒜 ∈ reachable_constraints (concat Ps). ∄ℐ.
          constraint_model ℐ (𝒜@[⟨1, send⟨[attack⟨ln 1⟩]⟩⟩])"
proof -
  have P': "list_all wellformed_transaction (concat Ps)"
    using Ps_pc(2) unfolding wellformed_composable_protocols'_def by meson

  have Sec: "∀s ∈ Sec. ¬{} ⊢c s" "ground Sec" unfolding Sec_def f_def by auto

  note 0 = composable_protocols_par_comp_prot_alt[
          OF Ps_pc(1-3) component_secure, unfolded lbls_def[symmetric] L_def[symmetric]]

  note 1 = reachable_constraints_no_leakage_alt'[
          OF P' Ps_pc(4) Sec no_leakage[unfolded no_leakage_def]]

  show ?thesis using 0 1 unfolding f_def Sec_def by argo
qed

lemma composable_protocols_par_comp_prot':
  assumes P_defs:
      "Pc = concat Ps"
      "set Ps_with_stars =
        (λn. map (transaction_proj n) Pc) `
          set (remdups (concat
                (map (λT. map (the_LabelN ∘ fst) (filter (Not ∘ has_LabelS) (transaction_strand T)))
                  Pc)))"
    and Ps_wellformed:
      "list_all (list_all (Not ∘ has_LabelS) ∘ tl ∘ transaction_send) Pc"
      "wellformed_composable_protocols Ps N"
      "wellformed_composable_protocols' Ps"
      "composable_protocols Ps Ms S"
    and Ps_no_leakage:
      "list_all (welltyped_leakage_free_protocol S) Ps_with_stars"
    and P_def:
```

```
      "P = map (transaction_proj n) Pc"
    and P_wt_secure:
      "∀𝒜 ∈ reachable_constraints P. ∄ℐ.
        welltyped_constraint_model ℐ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
  shows "∀𝒜 ∈ reachable_constraints Pc. ∄ℐ.
          constraint_model ℐ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
by (rule composable_protocols_par_comp_prot[
          OF Ps_wellformed(2,3,4,1)[unfolded P_defs(1)]
             P_wt_secure[unfolded P_def[unfolded P_defs(1)]]
             transaction_proj_ball_subst[
               OF P_defs(2)[unfolded P_defs(1)]
                  Ps_no_leakage(1)[
                    unfolded list_all_iff welltyped_leakage_free_protocol_def Let_def]],
          unfolded P_defs(1)[symmetric]])

end

context
begin

lemma welltyped_constraint_model_leakage_model_swap:
  fixes I α δ::"('fun,'atom,'sets,'lbl) prot_subst" and s
  assumes A: "welltyped_constraint_model I (A@[⟨⋆, send⟨[s · δ]⟩⟩])"
    and α: "transaction_renaming_subst α P (vars_lsst A)"
    and δ: "wt_subst δ" "wf_trms (subst_range δ)" "subst_domain δ = fv s" "ground (subst_range δ)"
  obtains J
    where "welltyped_constraint_model J (A@[⟨⋆, send⟨[s · δ]⟩⟩])"
    and "ik_lsst A ·set J ⊢ s · α · J"
proof
  note defs = welltyped_constraint_model_def constraint_model_def
  note δ_s = subst_fv_dom_ground_if_ground_img[OF equalityD2[OF δ(3)] δ(4)]
  note α' = transaction_renaming_subst_is_renaming(2)[OF α]
            inj_on_subset[OF transaction_renaming_subst_is_injective[OF α]
                             subset_UNIV[of "fv s"]]
            transaction_renaming_subst_var_obtain(2)[OF α, of _ s]
            transaction_renaming_subst_is_renaming(6)[OF α, of s]
            transaction_renaming_subst_vars_disj(8)[OF α]
            transaction_renaming_subst_wt[OF α]

  define αinv where "αinv ≡ subst_var_inv α (fv s)"
  define δ' where "δ' ≡ rm_vars (UNIV - fv (s · α)) (αinv ∘s δ)"
  define J where "J ≡ λx. if x ∈ fv (s · α) then δ' x else I x"

  have α_invertible: "s = s · α ∘s αinv"
    using α'(1) inj_var_ran_subst_is_invertible'[of α s] inj_on_subset[OF α'(2)]
    unfolding αinv_def by blast

  have δ'_domain: "subst_domain δ' = fv (s · α)"
  proof -
    have "x ∈ subst_domain (αinv ∘s δ)" when x: "x ∈ fv (s · α)" for x
    proof -
      obtain y where y: "y ∈ fv s" "α y = Var x"
        using α'(3)[OF x] by blast

      have "y ∈ subst_domain δ" using y(1) δ(3) by blast
      moreover have "(αinv ∘s δ) x = δ y"
        using y α'(3)[OF x] α_invertible
              vars_term_subset_subst_eq[of "Var y" s "α ∘s αinv" Var]
        unfolding δ'_def αinv_def
        by (metis (no_types, lifting) fv_subst_subset eval_term.simps(1)
                  subst_apply_term_empty subst_compose)
      ultimately show ?thesis using δ(4) by fastforce
    qed
```

```
    thus ?thesis using rm_vars_dom[of "UNIV - fv (s · α)" "αinv ∘s δ"] unfolding δ'_def by blast
  qed

  have δ'_range: "fv t = {}" when t: "t ∈ (subst_range δ')" for t
  proof -
    obtain x where "x ∈ fv (s · α)" "δ' x = t" using t δ'_domain by auto
    thus ?thesis
      by (metis (no_types, lifting) δ'_def subst_compose_def δ(3,4) α_invertible fv_subst_subset
            subst_fv_dom_ground_if_ground_img subst_subst_compose Diff_iff)
  qed

  have J0: "x ∈ fv (s · α) ⟹ J x = δ' x"
           "x ∉ fv (s · α) ⟹ J x = I x" for x
    unfolding J_def by (cases "x ∈ fv (s · α)") (simp_all add: subst_compose)

  have J1: "subst_range J ⊆ subst_range δ' ∪ subst_range I"
  proof
    fix t assume "t ∈ subst_range J"
    then obtain x where x: "x ∈ subst_domain J" "J x = t" by auto
    hence "t = δ' x ⟹ x ∈ subst_domain δ'" "t = I x ⟹ x ∈ subst_domain I"
      by (metis subst_domI subst_dom_vars_in_subst)+
    thus "t ∈ subst_range δ' ∪ subst_range I" using x(2) J0[of x] by auto
  qed

  have "x ∉ fv (s · α)" when x: "x ∈ fv_lsst (A@[⟨⋆, send⟨[s · δ]⟩⟩])" for x
    using x δ_s α'(4) α'(5) by auto
  hence "I x = J x" when x: "x ∈ fv_lsst (A@[⟨⋆, send⟨[s · δ]⟩⟩])" for x
    using x unfolding J_def δ'_def by auto
  hence "constr_sem_stateful J (unlabel (A@[⟨⋆, send⟨[s · δ]⟩⟩]))"
    using A strand_sem_model_swap[of "unlabel (A@[⟨⋆, send⟨[s · δ]⟩⟩])" I J "{}" "{}"]
    unfolding defs by blast
  moreover have "wt_subst J"
    using A subst_var_inv_wt[OF α'(6), of "fv s"]
          wt_subst_trm''[OF δ(1)] subst_compose[of "subst_var_inv α (fv s)" δ]
    unfolding defs J_def δ'_def αinv_def wt_subst_def by presburger
  moreover have "interpretation_subst J"
  proof -
    have "fv t = {}" when t: "t ∈ (subst_range J)" for t
      using t A J1 δ'_range unfolding defs by auto
    moreover have "x ∈ subst_domain J" for x
    proof (cases "x ∈ fv (s · α)")
      case True thus ?thesis using J0(1)[of x] δ'_domain unfolding subst_domain_def by auto
    next
      case False
      have "subst_domain I = UNIV" using A unfolding defs by fast
      thus ?thesis using J0(2)[OF False] unfolding subst_domain_def by auto
    qed
    ultimately show ?thesis by auto
  qed
  moreover have "wf_trms (subst_range δ')"
    using wf_trms_subst_compose[OF subst_var_inv_wf_trms[of α "fv s"] δ(2)]
    unfolding δ'_def αinv_def by force
  hence "wf_trms (subst_range J)" using A J1 unfolding defs by fast
  ultimately show "welltyped_constraint_model J (A@[⟨⋆, send⟨[s · δ]⟩⟩])" unfolding defs by blast
  hence "ik_lsst A ·set J ⊢ s · δ"
    using δ_s strand_sem_append_stateful[of "{}" "{}" "unlabel A" "[send⟨[s · δ]⟩]" J]
    unfolding defs by (simp add: subst_ground_ident)
  moreover have "s · α · J = s · δ"
  proof -
    have "J x = δ' x" when x: "x ∈ fv (s · α)" for x using x unfolding J_def by argo
    hence "s · α · J = s · α · δ'" using subst_agreement[of "s · α" J δ'] by force
    thus ?thesis
      using α_invertible unfolding δ'_def rm_vars_subst_eq'[symmetric] by (metis subst_subst_compose)
```

```
  qed
  hence "s · α · J = s · δ" by auto
  ultimately show "ik_lsst A ·set J ⊢ s · α · J" by argo
qed

lemma welltyped_leakage_free_protocol_pointwise:
  "welltyped_leakage_free_protocol S P ⟷ list_all (λs. welltyped_leakage_free_protocol [s] P) S"
unfolding welltyped_leakage_free_protocol_def list_all_iff Let_def by fastforce

lemma welltyped_leakage_free_no_deduct_constI:
  fixes c
  defines "s ≡ Fun c []::('fun,'atom,'sets,'lbl) prot_term"
  assumes s: "∀𝒜 ∈ reachable_constraints P. ∄ℐ_τ. welltyped_constraint_model ℐ_τ (𝒜@[⟨⋆, send⟨[s]⟩⟩])"
  shows "welltyped_leakage_free_protocol [s] P"
using s unfolding welltyped_leakage_free_protocol_def s_def by auto

lemma welltyped_leakage_free_pub_termI:
  assumes s: "{} ⊢_c s"
  shows "welltyped_leakage_free_protocol [s] P"
proof -
  define Q where "Q ≡ λM t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}"
  define f where "f ≡ λM. {t · δ | t δ. Q M t δ}"
  define Sec where "Sec ≡ f (set [s]) - {m. {} ⊢_c m}"

  have 0: "fv s = {}" using s pgwt_ground pgwt_is_empty_synth by blast
  have 1: "s · δ = s" for δ by (rule subst_ground_ident[OF 0])
  have 2: "wt_subst Var" "wf_trms (subst_range Var)"
    using wt_subst_Var wf_trm_subst_range_Var by (blast,blast)

  have "f (set [s]) = {s}"
  proof
    show "f (set [s]) ⊆ {s}" using 0 1 unfolding f_def Q_def by auto

    have "Q {s} s Var" using 0 2 unfolding Q_def by auto
    thus "{s} ⊆ f (set [s])" using 1[of Var] unfolding f_def by force
  qed
  hence "Sec = {}" using s unfolding Sec_def by simp
  thus ?thesis unfolding welltyped_leakage_free_protocol_def Let_def Sec_def f_def Q_def by blast
qed

lemma welltyped_leakage_free_pub_constI:
  assumes c: "public_f c" "arity_f c = 0"
  shows "welltyped_leakage_free_protocol [⟨c⟩_c] P"
using c welltyped_leakage_free_pub_termI[OF intruder_synth.ComposeC[of "[]" "Fu c" "{}"]] by simp

lemma welltyped_leakage_free_long_term_secretI:
  fixes n
  defines
      "Tatt ≡ λs'. Transaction (λ(). []) [] [⟨n, receive⟨[s']⟩⟩] [] [] [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]"
  assumes P_wt_secure:
      "∀𝒜 ∈ reachable_constraints P. ∄ℐ.
        welltyped_constraint_model ℐ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
    and s_long_term_secret:
      "∃ϑ. wt_subst ϑ ∧ inj_on ϑ (fv s) ∧ ϑ ` fv s ⊆ range Var ∧ Tatt (s · ϑ) ∈ set P"
  shows "welltyped_leakage_free_protocol [s] P"
proof (rule ccontr)
  define Q where "Q ≡ λM t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}"
  define f where "f ≡ λM. {t · δ | t δ. Q M t δ}"
  define Sec where "Sec ≡ f (set [s]) - {m. {} ⊢_c m}"

  note defs = Sec_def f_def Q_def
  note defs' = welltyped_constraint_model_def constraint_model_def
```

```
assume "¬welltyped_leakage_free_protocol [s] P"
then obtain A I s' where A:
    "A ∈ reachable_constraints P" "s' ∈ Sec - declassified_lsst A I"
    "welltyped_constraint_model I (A@[⟨⋆, send⟨[s']⟩⟩])"
  unfolding welltyped_leakage_free_protocol_def defs by fastforce

obtain ϑ where ϑ: "wt_subst ϑ" "ϑ ` fv s ⊆ range Var" "inj_on ϑ (fv s)" "Tatt (s · ϑ) ∈ set P"
  using s_long_term_secret by blast

obtain δ where δ:
  "wt_subst δ" "wf_trms (subst_range δ)" "subst_domain δ = fv (s · ϑ)" "ground (subst_range δ)"
  "s' = s · ϑ · δ"
proof -
  obtain δ where *: "wt_subst δ" "wf_trms (subst_range δ)" "fv s' = {}" "s' = s · δ"
    using A(2) unfolding defs by auto

  define σ where "σ ≡ subst_var_inv ϑ (fv s) ∘s δ"
  define δ' where "δ' ≡ rm_vars (UNIV - fv (s · ϑ)) σ"

  have **: "s' = s · ϑ · σ"
    using *(4) inj_var_ran_subst_is_invertible[OF ϑ(3,2)]
    unfolding σ_def by simp

  have "s' = s · ϑ · δ'"
    using ** rm_vars_subst_eq'[of "s · ϑ" σ]
    unfolding δ'_def by simp
  moreover have "wt_subst σ"
    using ϑ(1) *(1) subst_var_inv_wt wt_subst_compose
    unfolding σ_def by presburger
  hence "wt_subst δ'" using wt_subst_rm_vars unfolding δ'_def by blast
  moreover have "wf_trms (subst_range σ)"
    using wf_trms_subst_compose[OF subst_var_inv_wf_trms *(2)] unfolding σ_def by blast
  hence "wf_trms (subst_range δ')" using wf_trms_subst_rm_vars'[of σ] unfolding δ'_def by blast
  moreover have "fv (s · ϑ) ⊆ subst_domain σ"
    using *(3) ** ground_term_subst_domain_fv_subset unfolding σ_def by blast
  hence "subst_domain δ' = fv (s · ϑ)"
    using rm_vars_dom[of "UNIV - fv (s · ϑ)" σ] unfolding δ'_def by blast
  moreover have "ground (subst_range δ')"
  proof -
    { fix t assume "t ∈ subst_range δ'"
      then obtain x where "x ∈ fv (s · ϑ)" "δ' x = t"
        using ‹subst_domain δ' = fv (s · ϑ)› by auto
      hence "t ⊑ s · ϑ · δ'" by (meson subst_mono_fv)
      hence "fv t = {}" using ‹s' = s · ϑ · δ'› *(3) ground_subterm by blast
    } thus ?thesis by force
  qed
  ultimately show thesis using that[of δ'] by fast
qed

have ξ: "transaction_decl_subst Var (Tatt t)"
  and σ: "transaction_fresh_subst Var (Tatt t) (trms_lsst A)"
  for t
  unfolding transaction_decl_subst_def transaction_fresh_subst_def Tatt_def by simp_all

obtain α::"('fun,'atom,'sets,'lbl) prot_subst"
  where α: "transaction_renaming_subst α P (vars_lsst A)"
  unfolding transaction_renaming_subst_def by blast

obtain J where J:
    "welltyped_constraint_model J (A@[⟨⋆, send⟨[s · ϑ · δ]⟩⟩])" "ik_lsst A ·set J ⊢ s · ϑ · α · J"
  using welltyped_constraint_model_leakage_model_swap[OF A(3)[unfolded δ(5)] α δ(1-4)] by blast

define T where "T = dual_lsst (transaction_strand (Tatt (s · ϑ)) ·lsst α)"
```

```
  define B where "B ≡ A@T"

  have "transaction_receive (Tatt t) = [⟨n, receive⟨[t]⟩⟩]"
       "transaction_checks (Tatt t) = []"
       "transaction_updates (Tatt t) = []"
       "transaction_send (Tatt t) = [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]"
    for t
    unfolding Tatt_def by simp_all
  hence T_def': "T = [⟨n, send⟨[s · ϑ · α]⟩⟩, ⟨n, receive⟨[attack⟨ln n⟩]⟩⟩]"
    using subst_lsst_append[of "transaction_receive (Tatt (s · ϑ))" _ α]
          subst_lsst_singleton[of "ln n" "receive⟨[s · ϑ]⟩" α]
          subst_lsst_singleton[of "ln n" "send⟨[attack⟨ln n⟩]⟩" α]
    unfolding transaction_strand_def T_def by fastforce

  have B0: "ik_lsst B ·_set J ⊢ attack⟨ln n⟩"
    using in_ik_sst_iff[of "attack⟨ln n⟩" "unlabel T"]
    unfolding B_def T_def' by (force intro!: intruder_deduct.Axiom)

  have B1: "B ∈ reachable_constraints P"
    using reachable_constraints.step[OF A(1) ϑ(4) ξ σ α]
    unfolding B_def T_def by simp

  have "welltyped_constraint_model J B"
    using J strand_sem_append_stateful[of "{}" "{}" "unlabel A" _ J]
    unfolding defs' B_def T_def' by fastforce
  hence B2: "welltyped_constraint_model J (B@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
    using B0 strand_sem_append_stateful[of "{}" "{}" "unlabel B" "[send⟨[attack⟨ln n⟩]⟩]" J]
    unfolding defs' B_def by auto

  show False using P_wt_secure B1 B2 by blast
qed

lemma welltyped_leakage_free_value_constI:
  assumes P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. admissible_transaction_terms T"
      "∀T ∈ set P. transaction_decl T () = []"
      "∀T ∈ set P. bvars_transaction T = {}"
    and P_fresh_declass:
      "∀T ∈ set P. transaction_fresh T ≠ [] ⟶
        (transaction_send T ≠ [] ∧ (let (l,a) = hd (transaction_send T)
          in l = ⋆ ∧ is_Send a ∧ Var ` set (transaction_fresh T) ⊆ set (the_msgs a)))"
  shows "welltyped_leakage_free_protocol [⟨m: value⟩_v] P"
proof (rule ccontr)
  define Q where "Q ≡ λM t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}"
  define f where "f ≡ λM. {t · δ | t δ. Q M t δ}"
  define Sec where "Sec ≡ f (set [⟨m: value⟩_v]) - {m. {} ⊢_c m}"

  note defs = Sec_def f_def Q_def
  note defs' = welltyped_constraint_model_def constraint_model_def

  assume "¬welltyped_leakage_free_protocol [⟨m: value⟩_v] P"
  then obtain A I s where A:
      "A ∈ reachable_constraints P" "s ∈ Sec - declassified_lsst A I"
      "welltyped_constraint_model I (A@[⟨⋆, send⟨[s]⟩⟩])"
    unfolding welltyped_leakage_free_protocol_def defs by fastforce

  have "ik_lsst A ·_set I ⊢ s · I" using welltyped_constraint_model_deduct_split[OF A(3)] by simp
  moreover have "s · I = s" using A(2) unfolding defs by fast
  ultimately have s_deduct: "ik_lsst (A ·_lsst I) ⊢ s" by (metis ik_sst_subst unlabel_subst)

  note I0 = welltyped_constraint_model_prefix[OF A(3)]
```

```
have I1: "wt_subst I" using A(3) unfolding defs' by blast

obtain f ts δ where f: "s = Fun f ts" "s = ⟨m: value⟩_v ⋅ δ" "¬{} ⊢_c s" "s ∉ declassified_lsst A I"
    and δ: "wt_subst δ" "wf_trms (subst_range δ)" "fv s = {}"
  using A(2) unfolding defs by (cases s) auto

have s1: "Γ s = TAtom Value"
  by (metis Γ.simps(1) Γ_v_TAtom f(2) wt_subst_trm''[OF δ(1)])

have s2: "wf_trm s"
  using f(2) δ(2) by force

have s3: "ts = []"
  using f(1) s1 s2 const_type_inv_wf by blast

obtain sn where sn: "s = Fun (Val sn) []"
  using s1 f(3) Γ_Fu_simps(4) Γ_Set_simps(3) unfolding f(1) s3 by (cases f) auto

have "s ⊑_set ik_lsst A ⋅_set I"
  using private_fun_deduct_in_ik'[OF s_deduct[unfolded sn]]
  by (metis sn public.simps(3) ik_sst_subst unlabel_subst)
hence s4: "s ⊑_set trms_lsst A"
  using constraint_model_Val_const_in_constr_prefix[OF A(1) I0 P(1,2)]
  unfolding sn by presburger

obtain B T ξ σ α where B:
    "prefix (B@dual_lsst (transaction_strand T ⋅_lsst ξ ∘_s σ ∘_s α)) A"
    "B ∈ reachable_constraints P" "T ∈ set P" "transaction_decl_subst ξ T"
    "transaction_fresh_subst σ T (trms_lsst B)" "transaction_renaming_subst α P (vars_lsst B)"
    "s ∈ subst_range σ"
  using constraint_model_Value_in_constr_prefix_fresh_action'[OF A(1) P(2-) s4[unfolded sn]] sn
  by blast

obtain Tts Tsnds sx
  where T: "transaction_send T = ⟨⋆, send⟨Tts⟩⟩#Tsnds" "Var ` set (transaction_fresh T) ⊆ set Tts"
    and sx: "Var sx ∈ set Tts" "σ sx = s"
  using P_fresh_declass B(3,5,7)
  unfolding transaction_fresh_subst_def is_Send_def
  by (cases "transaction_send T") (fastforce,fastforce)

have ξ_elim: "ξ ∘_s σ ∘_s α = σ ∘_s α"
  using admissible_transaction_decl_subst_empty'[OF bspec[OF P(3) B(3)] B(4)]
  by simp

have s5: "s ∈ set (Tts ⋅_list ξ ∘_s σ ∘_s α ∘_s I)"
  using sx unfolding ξ_elim sn by force

have s6: "⟨⋆, receive⟨Tts ⋅_list ξ ∘_s σ ∘_s α ∘_s I⟩⟩ ∈ set (A ⋅_lsst I)"
proof -
  have "⟨⋆, send⟨Tts⟩⟩ ∈ set (transaction_send T)"
    using T(1) by simp
  hence "⟨⋆, send⟨Tts ⋅_list δ⟩⟩ ∈ set (transaction_send T ⋅_lsst δ)" for δ
    unfolding subst_apply_labeled_stateful_strand_def by force
  hence "⟨⋆, send⟨Tts ⋅_list δ⟩⟩ ∈ set (transaction_strand T ⋅_lsst δ)" for δ
    using transaction_strand_subst_subsets(4)[of T δ] by fast
  hence *: "⟨⋆, receive⟨Tts ⋅_list δ⟩⟩ ∈ set (dual_lsst (transaction_strand T ⋅_lsst δ))" for δ
    using dual_lsst_steps_iff(1)[of ⋆ "Tts ⋅_list δ"] by blast

  have "⟨⋆, receive⟨Tts ⋅_list ξ ∘_s σ ∘_s α⟩⟩ ∈ set A"
    using B(1) *[of "ξ ∘_s σ ∘_s α"] unfolding prefix_def by force
  thus ?thesis
    unfolding subst_apply_labeled_stateful_strand_def by force
qed
```

```
  show False
    using s6 f(4) ideduct_mono[OF Axiom[OF s5], of "⋃{set ts|ts. ⟨⋆,receive⟨ts⟩⟩ ∈ set (A ·lsst I)}"]
    unfolding declassifiedlsst_def by blast
qed

lemma welltyped_leakage_free_priv_constI:
  fixes c
  defines "s ≡ Fun c []::('fun,'atom,'sets,'lbl) prot_term"
  assumes c: "¬public c" "arity c = 0" "Γ s = TAtom ca" "ca ≠ Value"
    and P: "∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γv x)"
           "∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γ s ≠ Γv x"
           "∀T ∈ set P. ∀t ∈ subtermsset (trmslsst (transaction_send T)). s ∉ set (snd (Ana t))"
           "∀T ∈ set P. s ∉ trmslsst (transaction_send T)"
           "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value ∨ (∃a. Γv x = ⟨a⟩τa)"
           "∀T ∈ set P. wellformed_transaction T"
   shows "∀𝒜 ∈ reachable_constraints P. ∄ℐτ. welltyped_constraint_model ℐτ (𝒜@[⟨⋆, send⟨[s]⟩⟩])"
         (is "∀𝒜 ∈ ?R. ?P 𝒜")
     and "welltyped_leakage_free_protocol [s] P"
proof -
  show "∀𝒜 ∈ ?R. ?P 𝒜"
  proof
    fix A assume A: "A ∈ reachable_constraints P"

    define Q where "Q ≡ λM t δ. t ∈ M ∧ wtsubst δ ∧ wftrms (subst_range δ) ∧ fv (t · δ) = {}"
    define f where "f ≡ λM. {t · δ | t δ. Q M t δ}"
    define Sec where "Sec ≡ f (set [s]) - {m. {} ⊢c m}"

    define f' where "f' ≡ λ(T,ξ,σ::('fun,'atom,'sets,'lbl) prot_subst,α).
                              duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
    define g' where "g' ≡ concat ∘ map f'"

    let ?P_s_cases = "λM. s ∈ M ∨ (∃m ∈ subtermsset M. s ∈ set (snd (Ana m)))"
    let ?P_s_cases' = "λM δ. s ∈ M ·set δ ∨ (∃m ∈ subtermsset M ·set δ. s ∈ set (snd (Ana m)))"

    note defs = Sec_def f_def Q_def
    note defs' = welltyped_constraint_model_def constraint_model_def

    show "?P A"
    proof (rule ccontr)
      assume "¬?P A"
      then obtain I where I: "welltyped_constraint_model I (A@[⟨⋆, send⟨[s]⟩⟩])" by blast

      obtain Ts where Ts:
          "A = g' Ts" "∀B. prefix B Ts ⟶ g' B ∈ reachable_constraints P"
          "∀B T ξ σ α. prefix (B@[(T,ξ,σ,α)]) Ts ⟶
                T ∈ set P ∧ transaction_decl_subst ξ T ∧
                transaction_fresh_subst σ T (trmslsst (g' B)) ∧
                transaction_renaming_subst α P (varslsst (g' B))"
        using reachable_constraints_as_transaction_lists[OF A(1)] unfolding g'_def f'_def by blast

      have "iklsst A ·set I ⊢ s · I" and I_s: "s · I = s"
        using welltyped_constraint_model_deduct_split[OF I]
        unfolding s_def by simp_all
      hence s_deduct: "iklsst (A ·lsst I) ⊢ s" "iklsst A ·set I ⊢ s"
        by (metis iksst_subst unlabel_subst, metis)

      have I_wt: "wtsubst I"
          and I_wf: "wftrms (subst_range I)"
          and I_grounds: "ground (subst_range I)"
          and I_interp: "interpretationsubst I"
        using I unfolding defs' by (blast,blast,blast,blast)
```

```
have Sec_unfold: "Sec = {s}"
proof
  have "¬{} ⊢c s" using ideduct_synth_priv_const_in_ik[OF _ c(1)] unfolding s_def by blast
  thus "{s} ⊆ Sec" unfolding defs s_def by fastforce
qed (auto simp add: defs s_def)

have s2: "wftrm s"
  using c(1,2) unfolding s_def by fastforce

have A_ik_fv: "∃a. Γv x = TAtom a ∧ a ≠ ca" when x: "x ∈ fvset (iklsst A)" for x
proof -
  obtain T where T: "T ∈ set P" "Γv x ∈ Γv ` fv_transaction T"
    using fv_iksst_is_fvsst[OF x] reachable_constraints_var_types_in_transactions(1)[OF A P(5)]
    by fast
  then obtain y where y: "y ∈ vars_transaction T" "Γv y = Γv x"
    using varssst_is_fvsst_bvarssst[of "unlabel (transaction_strand T)"] by fastforce
  then obtain a where a: "Γv y = TAtom a" using P(1) T(1) by blast
  hence "Γv x = TAtom a" "Γ s ≠ Γv x" "Γ s = TAtom ca" using y P(2) T(1) c(3) by auto
  thus ?thesis by force
qed

have I_s_x: "¬s ⊑ I x" when x: "x ∈ fvset (iklsst A)" for x
proof -
  obtain a where a: "Γv x = TAtom a" "a ≠ ca" using A_ik_fv[OF x] by blast
  hence a': "Γ (I x) = TAtom a" using wt_subst_trm''[OF I_wt, of "Var x"] by simp

  obtain f ts where f: "I x = Fun f ts"
    by (meson empty_fv_exists_fun interpretation_grounds_all[OF I_interp])
  hence ts: "ts = []"
    using I_wf const_type_inv_wf[OF a'[unfolded f]] by fastforce

  have "c ≠ f" using f[unfolded ts] a a' c(3)[unfolded s_def] by force
  thus ?thesis using f ts unfolding s_def by simp
qed

have A_ik_I_const: "∃f. arity f = 0 ∧ I x = Fun f []" when x: "x ∈ fvset (iklsst A)" for x
  using x A_ik_fv I_wt empty_fv_exists_fun[OF interpretation_grounds_all[OF I_interp, of x]]
        wf_trm_subst_rangeD[OF I_wf, of x] const_type_inv const_type_inv_wf
  by (metis (no_types, lifting) Γ.simps(1) wtsubst_def)
hence A_ik_subst: "subtermsset (iklsst A ·set I) = subtermsset (iklsst A) ·set I"
  using subterms_subst''[of "iklsst A" I] by blast

have sublmm1: "s ∈ set (snd (Ana m))"
  when m: "m ⊑set M" "s ∈ set (snd (Ana (m · δ)))"
   and M: "⋀y. y ∈ fvset M ⟹ ¬s ⊑ δ y"
  for m M δ
proof -
  have m_fun: "is_Fun m"
    using m M Ana_subterm' vars_iff_subtermeq_set
    unfolding s_def is_Var_def by (metis eval_term.simps(1))

  obtain f K R ts i where f:
      "m = ⟨f ts⟩t" "arityf f = length ts" "arityf f > 0" "Anaf f = (K, R)"
    and i: "i < length R" "s = ts ! (R ! i) · δ"
    and R_i: "∀i < length R. map ((!) ts) R ! i = ts ! (R ! i) ∧ R ! i < length ts"
  proof -
    obtain f ts K R where f:
        "m · δ = ⟨f ts⟩t" "arityf f = length ts" "arityf f > 0"
        "Anaf f = (K, R)" "Ana (m · δ) = (K ·list (!) ts, map ((!) ts) R)"
      using m(2) Ana_nonempty_inv[of "m · δ"] by force

    obtain ts' where m': "m = ⟨f ts'⟩t" "ts = ts' ·list δ"
      using f(1) m_fun by auto
```

```
    have R_i: "map ((!) ts) R ! i = ts ! (R ! i)" "R ! i < length ts"
      when i: "i < length R" for i
      using i Ana_f_assm2_alt[OF f(4), of "R ! i"] f(2) by simp_all
    then obtain i where i: "s = ts ! (R ! i)" "i < length R"
      by (metis (no_types, lifting) m(2) f(5) in_set_conv_nth length_map snd_conv)

    have ts': "arity_f f = length ts'" "length ts = length ts'" using m'(2) f(2) by simp_all

    have s': "s = ts' ! (R ! i) · δ" using R_i(2)[OF i(2)] i(1) unfolding ts'(2) m'(2) by simp

    show thesis using that f m' R_i ts' s' i by auto
  qed

  have "s = ts ! (R ! i)"
  proof (cases "ts ! (R ! i)")
    case (Var x)
    hence "Var x ∈ set ts" using R_i i nth_mem by fastforce
    hence "x ∈ fv_set M" using m(1) f(1) fv_subterms_set by fastforce
    thus ?thesis using i M Var by fastforce
  qed (use i s_def in fastforce)
  thus "s ∈ set (snd (Ana m))" using f(1) Ana_Fu_intro[OF f(2-4)] i(1) by simp
qed

have "¬s ⊑ δ y"
  when m: "m ⊑_set trms_lsst (transaction_send T)" "s ∈ set (snd (Ana (m · δ)))"
    and T: "T ∈ set P" and δ_wt: "wt_subst δ"
    and δ_ran: "⋀t. t ∈ subst_range δ ⟹ (∃c. t = Fun c [] ∧ arity c = 0) ∨ (∃x. t = Var x)"
    and y: "y ∈ fv_set (trms_lsst (transaction_send T))"
  for m T δ y
proof
  assume "s ⊑ δ y"
  hence "Γ_v y = Γ s" using wt_subst_trm''[OF δ_wt, of "Var y"] δ_ran[of "δ y"] by fastforce
  moreover have "y ∈ vars_transaction T"
    using y trms_sst_fv_vars_sst_subset unfolding vars_transaction_unfold[of T] by fastforce
  ultimately show False using P(2) T by force
qed
hence sublmm2: "s ∈ set (snd (Ana m))"
  when m: "m ⊑_set trms_lsst (transaction_send T)" "s ∈ set (snd (Ana (m · δ)))"
    and T: "T ∈ set P" and δ_wt: "wt_subst δ"
    and δ_ran: "⋀t. t ∈ subst_range δ ⟹ (∃c. t = Fun c [] ∧ arity c = 0) ∨ (∃x. t = Var x)"
  for m T δ
  using sublmm1[OF m] m T δ_wt δ_ran by blast

have "s ∈ ik_lsst A ∨ (∃m ∈ subterms_set (ik_lsst A) ·_set I. s ∈ set (snd (Ana m)))"
  using private_const_deduct[OF c(1) s_deduct(2)[unfolded s_def]]
        I_s_x const_mem_subst_cases[of c] A_ik_subst
  unfolding s_def by blast
hence "?P_s_cases (ik_lsst A)" using sublmm1[of _ "ik_lsst A"] I_s_x by blast
then obtain T ξ σ α where T: "(T,ξ,σ,α) ∈ set Ts" "?P_s_cases (ik_lsst (f' (T,ξ,σ,α)))"
  using ik_lsst_concat[of "map f' Ts"] Ts(1)[unfolded g'_def] by fastforce

obtain B where "prefix (B@[(T, ξ, σ, α)]) Ts" by (metis T(1) prefix_snoc_in_iff)
hence T_in_P: "T ∈ set P"
    and T_wf: "wellformed_transaction T"
    and ξ: "transaction_decl_subst ξ T"
    and σ: "transaction_fresh_subst σ T (trms_lsst (concat (map f' B)))"
    and α: "transaction_renaming_subst α P (vars_lsst (concat (map f' B)))"
  using P(6) Ts(3)[unfolded g'_def] unfolding comp_def by (metis,metis,metis,metis,metis)

note ξσα_wt = transaction_decl_fresh_renaming_substs_wt[OF ξ σ α]
note ξσα_ran = transaction_decl_fresh_renaming_substs_range'(1)[OF ξ σ α]
```

```
      have "subterms_set (M ·_set ξ ∘_s σ ∘_s α) = subterms_set M ·_set ξ ∘_s σ ∘_s α" for M
        using ξσα_ran subterms_subst''[of _ "ξ ∘_s σ ∘_s α"] by (meson subst_imgI)
      hence s_cases: "?P_s_cases' (trms_lsst (transaction_send T)) (ξ ∘_s σ ∘_s α)"
        using T(2) dual_transaction_ik_is_transaction_send'[OF T_wf, of "ξ ∘_s σ ∘_s α"]
        unfolding f'_def by auto

      from s_cases show False
      proof
        assume "s ∈ trms_lsst (transaction_send T) ·_set ξ ∘_s σ ∘_s α"
        then obtain t where t: "t ∈ trms_lsst (transaction_send T)" "s = t · ξ ∘_s σ ∘_s α" by force
        have "s ≠ t" using P(4) T_in_P t(1) by blast
        then obtain x where x: "t = Var x" using t(2) unfolding s_def by (cases t) auto

        have "Γ_v x = Γ s" using x t(2) wt_subst_trm''[OF ξσα_wt, of "Var x"] by simp
        moreover have "x ∈ vars_transaction T"
          using t(1) trms_sst_fv_vars_sst_subset unfolding x vars_transaction_unfold[of T] by fastforce
        ultimately show False using P(2) T_in_P by force
      qed (use sublmm2[OF _ _ T_in_P ξσα_wt ξσα_ran] P(3) T_in_P in blast)
    qed
  qed
  thus "welltyped_leakage_free_protocol [s] P"
    using welltyped_leakage_free_no_deduct_constI[of P c]
    unfolding s_def by blast
qed

lemma welltyped_leakage_free_priv_constI':
  assumes c: "¬public_f c" "arity_f c = 0" "Γ_f c = Some ca"
    and P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γ ⟨c⟩_c ≠ Γ_v x"
      "∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γ_v x)"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = ⟨a⟩_τa)"
      "∀T ∈ set P. ∀t ∈ subterms_set (trms_lsst (transaction_send T)). ⟨c⟩_c ∉ set (snd (Ana t))"
      "∀T ∈ set P. ⟨c⟩_c ∉ trms_lsst (transaction_send T)"
   shows "∀A ∈ reachable_constraints P. ∄I_τ. welltyped_constraint_model I_τ (A@[⟨⋆, send⟨[⟨c⟩_c]⟩⟩])"
     and "welltyped_leakage_free_protocol [⟨c⟩_c] P"
using c welltyped_leakage_free_priv_constI[OF _ _ _ _ P(3,2,5,6,4,1), of "Atom ca"]
by (force, force)

lemma welltyped_leakage_free_set_constI:
  assumes P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. ∀f ∈ ⋃(funs_term ` (trms_lsst (transaction_send T))). ¬is_Set f"
      "∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γ_v x ≠ TAtom SetType"
      "∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γ_v x)"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = ⟨a⟩_τa)"
    and c: "arity_s c = 0"
  shows "∀A ∈ reachable_constraints P. ∄I_τ. welltyped_constraint_model I_τ (A@[⟨⋆, send⟨[⟨c⟩_s]⟩⟩])"
    and "welltyped_leakage_free_protocol [⟨c⟩_s] P"
proof -
  have c'': "⟨c⟩_s ∉ subterms t"
    when T: "T ∈ set P" and t: "t ⊑_set trms_lsst (transaction_send T)" for T t
    using t bspec[OF P(2) T] subtermeq_imp_funs_term_subset[of t]
          funs_term_Fun_subterm'[of "Set c" "[]::('fun,'atom,'sets,'lbl) prot_term list"]
    by fastforce

  have P':
      "∀T ∈ set P. ∀t ∈ subterms_set (trms_lsst (transaction_send T)). ⟨c⟩_s ∉ set (snd (Ana t))"
      "∀T ∈ set P. ⟨c⟩_s ∉ trms_lsst (transaction_send T)"
    subgoal using Ana_subterm' c'' by fast
    subgoal using c'' by fast
    done
```

```
  have P'':
      "∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γ ⟨c⟩s ≠ Γv x"
    using P(3) Γ_consts_simps(4)[OF c] by fastforce

  show "∀A ∈ reachable_constraints P. ∄Iτ. welltyped_constraint_model Iτ (A@[⟨⋆, send⟨[⟨c⟩s]⟩⟩])"
       "welltyped_leakage_free_protocol [⟨c⟩s] P"
    using c welltyped_leakage_free_priv_constI[OF _ _ _ _ P(4) P'' P' P(5,1), of SetType]
    by (force, force)
qed

lemma welltyped_leakage_free_occurssec_constI:
  defines "s ≡ Fun OccursSec []"
  assumes P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γv x ≠ TAtom OccursSecType"
      "∀T ∈ set P. ∀t ∈ subtermsset (trmslsst (transaction_send T)). Fun OccursSec [] ∉ set (snd (Ana
t))"
      "∀T ∈ set P. Fun OccursSec [] ∉ trmslsst (transaction_send T)"
      "∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γv x)"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value ∨ (∃a. Γv x = ⟨a⟩τa)"
  shows "∀A ∈ reachable_constraints P. ∄Iτ. welltyped_constraint_model Iτ (A@[⟨⋆, send⟨[s]⟩⟩])"
    and "welltyped_leakage_free_protocol [s] P"
proof -
  have P':
      "∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γ (Fun OccursSec []) ≠ Γv
x"
    using P(2) by auto

  show "∀A ∈ reachable_constraints P. ∄Iτ. welltyped_constraint_model Iτ (A@[⟨⋆, send⟨[s]⟩⟩])"
       "welltyped_leakage_free_protocol [s] P"
    using welltyped_leakage_free_priv_constI[OF _ _ _ _ P(5) P' P(3,4,6,1), of OccursSecType]
    unfolding s_def by auto
qed

lemma welltyped_leakage_free_occurs_factI:
  assumes P: "∀T ∈ set P. admissible_transaction' T"
    and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
    and P_occ_star:
      "∀T ∈ set P. ∀r ∈ set (transaction_send T).
        OccursFact ∈ ⋃(funs_term ` (trmssstp (snd r))) ⟶ fst r = ⋆"
  shows "welltyped_leakage_free_protocol [occurs x] P"
proof -
  define Q where "Q ≡ λM t δ. t ∈ M ∧ wtsubst δ ∧ wftrms (subst_range δ) ∧ fv (t · δ) = {}"
  define f where "f ≡ λM. {t · δ | t δ. Q M t δ}"
  define Sec where "Sec ≡ f (set [occurs x]) - {m. {} ⊢c m}"

  define f' where "f' ≡ λ(T,ξ,σ::('fun,'atom,'sets,'lbl) prot_subst,α).
                        duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
  define g' where "g' ≡ concat ∘ map f'"

  note defs = Sec_def f_def Q_def
  note defs' = welltyped_constraint_model_def constraint_model_def

  show ?thesis
  proof (rule ccontr)
    assume "¬welltyped_leakage_free_protocol [occurs x] P"
    then obtain A I k where A:
        "A ∈ reachable_constraints P" "occurs k ∈ Sec - declassifiedlsst A I"
        "welltyped_constraint_model I (A@[⟨⋆, send⟨[occurs k]⟩⟩])"
      unfolding welltyped_leakage_free_protocol_def defs by fastforce

    note A' = welltyped_constraint_model_prefix[OF A(3)]
```

```
    have occ_I: "occurs k · I = occurs k" using A(2) unfolding defs by auto
    hence occ_in_ik: "occurs k ∈ ik_lsst A" "occurs k ∈ ik_lsst A ·set I"
      using welltyped_constraint_model_deduct_split(2)[OF A(3)]
            reachable_constraints_occurs_fact_deduct_in_ik[OF A(1) A' P P_occ, of k]
      by (argo, argo)
    then obtain l ts where ts: "(l,receive⟨ts⟩) ∈ set A" "occurs k ∈ set ts"
      using in_ik_sst_iff[of "occurs k" "unlabel A"] unfolding unlabel_def by force

    obtain T a B α σ ξ
      where B: "prefix (B@f' (T,ξ,σ,α)) A"
        and T: "T ∈ set P" "transaction_decl_subst ξ T"
               "transaction_fresh_subst σ T (trms_lsst B)"
               "transaction_renaming_subst α P (vars_lsst B)"
        and a: "a ∈ set (transaction_strand T)" "fst (l,receive⟨ts⟩) = fst a"
               "(l,receive⟨ts⟩) = dual_lsstp a ·lsstp ξ ∘s σ ∘s α"
      using reachable_constraints_transaction_action_obtain[OF A(1) ts(1), of thesis]
      unfolding f'_def by simp

    obtain ts' where ts': "a = (l,send⟨ts'⟩)" "ts = ts' ·list ξ ∘s σ ∘s α"
      using surj_pair[of a] a(2,3) by (cases "snd a") force+

    obtain t where t: "t ∈ set ts'" "occurs k = t · ξ ∘s σ ∘s α"
      using ts(2) unfolding ts'(2) by force

    have occ_t: "OccursFact ∈ funs_term t"
    proof (cases t)
      case (Var y) thus ?thesis
        using t(2) transaction_decl_fresh_renaming_substs_range'(1)[OF T(2-), of "occurs k"]
        by fastforce
    qed (use t(2) in simp)

    have P_wf: "∀T ∈ set P. wellformed_transaction T"
      using P admissible_transaction_is_wellformed_transaction(1) by blast

    have l: "l = ⋆"
      using wellformed_transaction_strand_memberD(8)[OF bspec[OF P_wf T(1)] a(1)[unfolded ts'(1)]]
            t(1) T(1) P_occ_star occ_t
      unfolding ts'(1) by fastforce

    have "occurs k ∈ ⋃{set ts |ts. ⟨⋆, receive⟨ts⟩⟩ ∈ set (A ·lsst I)}"
      using subst_lsst_memI[OF ts(1), of I] subst_set_map[OF ts(2), of I]
      unfolding occ_I l by auto
    thus False using A(2) unfolding declassified_lsst_def by simp
  qed
qed

lemma welltyped_leakage_free_setop_pairI:
  assumes P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. ∀x ∈ vars_transaction T. Γv x = TAtom Value ∨ (∃a. Γv x = ⟨a⟩τa)"
      "∀T ∈ set P. ∀f ∈ ⋃(funs_term ` (trms_lsst (transaction_send T))). ¬is_Set f"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
      "∀T ∈ set P. transaction_decl T () = []"
      "∀T ∈ set P. admissible_transaction_terms T"
    and c: "arity_s c = 0"
  shows "welltyped_leakage_free_protocol [pair (x, ⟨c⟩s)] P"
proof -
  define Q where "Q ≡ λM t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}"
  define f where "f ≡ λM. {t · δ | t δ. Q M t δ}"
  define Sec where "Sec ≡ f (set [pair (x, ⟨c⟩s)]) - {m. {} ⊢c m}"

  define f' where "f' ≡ λ(T,ξ,σ::('fun,'atom,'sets,'lbl) prot_subst,α).
                          dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
```

```
  define g' where "g' ≡ concat ∘ map f'"

  note defs = Sec_def f_def Q_def
  note defs' = welltyped_constraint_model_def constraint_model_def

  have P':
      "∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γv x ≠ TAtom SetType"
      "∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γv x)"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value ∨ (∃a. Γv x = ⟨a⟩τa)"
    subgoal using P(2,4) by fastforce
    subgoal using P(2) by fastforce
    subgoal using P(4) by fast
    done

  note 0 = welltyped_leakage_free_set_constI(1)[OF P(1,3) P' c]

  show ?thesis
  proof (rule ccontr)
    assume "¬welltyped_leakage_free_protocol [pair (x, ⟨c⟩s)] P"
    then obtain A I k where A:
        "A ∈ reachable_constraints P" "pair (k, ⟨c⟩s) ∈ Sec - declassifiedlsst A I"
        "welltyped_constraint_model I (A@[⟨⋆, send⟨[pair (k, ⟨c⟩s)]⟩⟩])"
      unfolding welltyped_leakage_free_protocol_def defs pair_def by fastforce

    note A' = welltyped_constraint_model_prefix[OF A(3)]

    have "pair (k, ⟨c⟩s) · I = pair (k, ⟨c⟩s)" using A(2) unfolding defs by auto
    hence "iklsst A ·set I ⊢ pair (k, ⟨c⟩s)"
      using welltyped_constraint_model_deduct_split(2)[OF A(3)] by argo
    then obtain n where n: "intruder_deduct_num (iklsst A ·set I) n (pair (k, ⟨c⟩s))"
      using deduct_num_if_deduct by fast

    have "wtsubst I" "wftrms (subst_range I)" "iklsst A ⊆ trmslsst A"
      using A(3) iksst_trmssst_subset unfolding defs' by simp_all
    hence "iklsst A ·set I ⊆ SMP (trmslsst A)" by blast
    hence "Pair ∉ ⋃(funs_term ` (iklsst A ·set I))"
      using reachable_constraints_no_Pair_fun'[OF A(1) P(4-6)] P by blast
    hence 1: "¬pair (k, ⟨c⟩s) ⊑set iklsst A ·set I"
      using funs_term_Fun_subterm'[of Pair] unfolding pair_def by auto

    have 2: "pair (k, ⟨c⟩s) ∉ set (snd (Ana m))" when m: "m ⊑set iklsst A ·set I" for m
      using m 1 term.dual_order.trans Ana_subterm'[of "pair (k, ⟨c⟩s)" m] by auto

    have "¬iklsst A ·set I ⊢ ⟨c⟩s"
      using 0 A(1) A' welltyped_constraint_model_deduct_iff[of I A ⋆ "⟨c⟩s"] by force
    moreover have "iklsst A ·set I ⊢ ⟨c⟩s"
      using 1 2 deduct_inv[OF n] deduct_if_deduct_num[of "iklsst A ·set I" _ "⟨c⟩s"]
      unfolding pair_def by auto
    ultimately show False by blast
  qed
qed

lemma welltyped_leakage_free_short_term_secretI:
  fixes c x y f n d l l'
  defines "s ≡ ⟨f [⟨c⟩c, ⟨x: value⟩v]⟩t"
    and "Tatt ≡ Transaction (λ(). []) []
          [⟨⋆, receive⟨[occurs ⟨y: value⟩v]⟩⟩,
           ⟨l, receive⟨[⟨f [⟨c⟩c, ⟨y: value⟩v]⟩t]⟩⟩]
          [⟨l', ⟨⟨y: value⟩v not in ⟨d⟩s⟩⟩]
          []
          [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]"
  assumes P:
      "∀T ∈ set P. admissible_transaction' T"
```

```
      "∀T ∈ set P. admissible_transaction_occurs_checks T"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value ∨ (∃a. Γv x = ⟨a⟩τa)"
    and subterms_sec:
      "∀𝒜 ∈ reachable_constraints P. ∄ℐτ. welltyped_constraint_model ℐτ (𝒜@[⟨⋆, send⟨[⟨c⟩c]⟩⟩])"
    and P_sec:
      "∀𝒜 ∈ reachable_constraints P. ∄ℐτ.
        welltyped_constraint_model ℐτ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
    and P_Tatt: "Tatt ∈ set P"
    and P_d:
      "∀T ∈ set P. ∀a ∈ set (transaction_updates T).
        is_Insert (snd a) ∧ the_set_term (snd a) = ⟨d⟩s ⟶
          transaction_send T ≠ [] ∧ (let (l,b) = hd (transaction_send T)
            in l = ⋆ ∧ is_Send b ∧ ⟨f [⟨c⟩c, the_elem_term (snd a)]⟩t ∈ set (the_msgs b))"
  shows "welltyped_leakage_free_protocol [s] P"
proof -
  define Q where "Q ≡ λM t δ. t ∈ M ∧ wtsubst δ ∧ wftrms (subst_range δ) ∧ fv (t · δ) = {}"
  define Sec where "Sec ≡ {t · δ | t δ. Q (set [s]) t δ} - {m. {} ⊢c m}"

  define f' where "f' ≡ λ(T,ξ,σ::('fun,'atom,'sets,'lbl) prot_subst,α).
                          duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
  define g' where "g' ≡ concat ∘ map f'"

  note defs = Sec_def Q_def
  note defs' = welltyped_constraint_model_def constraint_model_def

  show ?thesis
  proof (rule ccontr)
    assume "¬welltyped_leakage_free_protocol [s] P"
    then obtain A I k where A:
        "A ∈ reachable_constraints P" "⟨f [⟨c⟩c, k]⟩t ∈ Sec - declassifiedlsst A I"
        "welltyped_constraint_model I (A@[⟨⋆, send⟨[⟨f [⟨c⟩c, k]⟩t]⟩⟩])"
      unfolding welltyped_leakage_free_protocol_def defs s_def by fastforce

    have I: "wtsubst I" "interpretationsubst I" "wftrms (subst_range I)"
      using A(3) unfolding defs' by (blast,blast,blast)

    note A' = welltyped_constraint_model_prefix[OF A(3)]

    have "strand_sem_stateful {} {} (unlabel A) I"
      using A' unfolding defs' by simp
    hence A'': "strand_sem_stateful {} {} (unlabel A) (I(z := k))"
      when z: "z ∉ fvlsst A" for z
      using z strand_sem_model_swap[of "unlabel A" I "I(z := k)" "{}" "{}"] by auto

    obtain δ where δ:
        "δ (the_Var ⟨x: value⟩v) = k" "wtsubst δ" "wftrms (subst_range δ)"
        "fv (δ (the_Var ⟨x: value⟩v)) = {}"
      using A(2) unfolding defs s_def by auto

    have k: "Γ k = TAtom Value" "fv k = {}" "wftrm k"
      subgoal using δ(1) wt_subst_trm''[OF δ(2), of "⟨x: value⟩v"] by simp
      subgoal using δ(1,4) by blast
      subgoal using δ(1,3) by force
      done
    then obtain fk where fk: "k = Fun fk []"
      using const_type_inv_wf by (cases k) auto

    have fk': "iklsst A ·set I ⊢ ⟨f [⟨c⟩c, k]⟩t"
      using fk welltyped_constraint_model_deduct_split(2)[OF A(3)] by auto

    have "¬welltyped_constraint_model I (A@[⟨⋆, send⟨[⟨c⟩c]⟩⟩])"
      using subterms_sec(1) A(1) by blast
    hence "¬iklsst A ·set I ⊢ ⟨c⟩c"
```

```
    using A' strand_sem_append_stateful[of "{}" "{}" "unlabel A" "unlabel [⟨⋆, send⟨[⟨c⟩_c]⟩⟩]" I]
    unfolding defs' by auto
  hence "⟨f [⟨c⟩_c, k]⟩_t ⊑_set ik_lsst A ·_set I"
    using fk' deduct_inv'[OF fk'] by force
  moreover have "k ⊑ ⟨f [⟨c⟩_c, k]⟩_t" by simp
  ultimately have k_in_ik: "k ⊑_set ik_lsst A ·_set I"
    using in_subterms_subset_Union by blast
  hence "k ⊑_set ik_lsst A ∨ (∃x ∈ fv_set (ik_lsst A). k ⊑ I x)"
    using const_subterms_subst_cases[of fk I "ik_lsst A"]
    unfolding fk by fast
  hence "fk ∈ ⋃(funs_term ` ik_lsst A) ∨ (∃x ∈ fv_lsst A. k ⊑ I x)"
    unfolding fk by (meson UN_iff funs_term_Fun_subterm' fv_ik_sst_is_fv_sst )
  hence "fk ∈ ⋃(funs_term ` trms_lsst A) ∨ (∃x ∈ fv_lsst A. k ⊑ I x)"
    using ik_sst_trms_sst_subset by blast
  moreover have "Γ_v x = TAtom Value" when x: "x ∈ fv_lsst A" for x
    using x reachable_constraints_var_types_in_transactions(1)[OF A(1) P(3)]
          P(1,2) admissible_transaction_Value_vars
    by force
  ultimately have "fk ∈ ⋃(funs_term ` trms_lsst A) ∨ (∃x ∈ fv_lsst A. Γ_v x = TAtom Value ∧ k ⊑ I x)"
    by blast
  hence "fk ∈ ⋃(funs_term ` trms_lsst A) ∨ (∃x ∈ fv_lsst A. Γ_v x = TAtom Value ∧ I x = k)"
    using I(1,3) wf_trm_TComp_subterm wf_trm_subst_rangeD
    unfolding fk wt_subst_def by (metis Γ.simps(1) term.distinct(1))
  then obtain kn where kn: "fk = Val kn"
    using reachable_constraints_val_funs_private[OF A(1) P(1), of fk]
          constraint_model_Value_term_is_Val[OF A(1) A' P(1,2)]
          Fun_Value_type_inv[OF k(1)[unfolded fk]]
    unfolding fk is_PubConstValue_def by (cases fk) force+

  obtain α::"('fun,'atom,'sets,'lbl) prot_subst"
    where α: "transaction_renaming_subst α P (vars_lsst A)"
    unfolding transaction_renaming_subst_def by blast

  obtain y' yn' where y':
      "α (the_Var ⟨y: value⟩_v) = Var y'" "y ≠ yn'" "Var y' = ⟨yn': value⟩_v"
    using transaction_renaming_subst_is_renaming(1,2)[OF α] by force

  define B where "B ≡ A@dual_lsst (transaction_strand Tatt ·_lsst α)"
  define J where "J ≡ I(y' := k)"

  have "y' ∈ range_vars α"
    using y'(1) transaction_renaming_subst_is_renaming(3)[OF α]
    by (metis (no_types, lifting) in_mono subst_fv_imgI term.set_intros(3))
  hence y'': "y' ∉ vars_lsst A"
    using transaction_renaming_subst_vars_disj(6)[OF α] by blast

  have 0: "(k,⟨d⟩_s) ∉ set (db_lsst A I)"
  proof
    assume a: "(k,⟨d⟩_s) ∈ set (db_lsst A I)"
    obtain l t t' where t: "(l,insert⟨t,t'⟩) ∈ set A" "t · I = k" "t' · I = ⟨d⟩_s"
      using db_sst_in_cases[OF a[unfolded db_sst_def]] unfolding unlabel_def by auto

    obtain T b B α σ ξ where T:
        "prefix (B@dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)) A"
        "T ∈ set P" "transaction_decl_subst ξ T"
        "transaction_fresh_subst σ T (trms_lsst B)" "transaction_renaming_subst α P (vars_lsst B)"
        "b ∈ set (transaction_strand T)" "(l, insert⟨t,t'⟩) = dual_lsstp b ·_lsstp ξ ∘_s σ ∘_s α"
        "fst (l, insert⟨t,t'⟩) = fst b"
      using reachable_constraints_transaction_action_obtain[OF A(1) t(1)] by metis

    define ϑ where "ϑ ≡ ξ ∘_s σ ∘_s α"

    obtain b' where "b = (l,b')"
```

```
      using T(8) by (cases b) simp
    then obtain tb tb'
        where b': "b = (l,insert⟨tb,tb'⟩)"
          and tb: "t = tb · ϑ"
          and tb': "t' = tb' · ϑ"
      using T(7) unfolding ϑ_def by (cases b') auto

    note T_adm = bspec[OF P(1) T(2)]
    note T_wf = admissible_transaction_is_wellformed_transaction(1,3)[OF T_adm]

    have b: "b ∈ set (transaction_updates T)"
      using transaction_strand_memberD[OF T(6)[unfolded b']]
            wellformed_transaction_cases[OF T_wf(1)]
      unfolding b' by blast

    have "∃n. tb = ⟨n: value⟩_v" and *: "tb' = ⟨d⟩_s"
      using tb tb' T(6) t(3) transaction_inserts_are_Value_vars[OF T_wf, of tb tb']
      unfolding b' unlabel_def by (force,force)

    have "is_Insert (snd b)" "the_set_term (snd b) = ⟨d⟩_s" "the_elem_term (snd b) = tb"
      unfolding b' * by simp_all
    hence "transaction_send T ≠ []"
          "let (l, a) = hd (transaction_send T)
           in l = ⋆ ∧ is_Send a ∧ ⟨f [⟨c⟩_c, tb]⟩_t ∈ set (the_msgs a)"
      using P_d T(2) b by (fast,fast)
    hence "∃ts. ⟨⋆,send⟨ts⟩⟩ ∈ set (transaction_send T) ∧ ⟨f [⟨c⟩_c, tb]⟩_t ∈ set ts"
      unfolding is_Send_def by (cases "transaction_send T") auto
    then obtain ts where ts: "⟨⋆,send⟨ts⟩⟩ ∈ set (transaction_strand T)" "⟨f [⟨c⟩_c, tb]⟩_t ∈ set ts"
      unfolding transaction_strand_def by auto

    have "⟨⋆,receive⟨ts ·_list ϑ⟩⟩ ∈ set A" "⟨f [⟨c⟩_c, t]⟩_t ∈ set (ts ·_list ϑ)"
      using subst_lsst_memI[OF ts(1), of ϑ] subst_set_map[OF ts(2), of ϑ]
            dual_lsst_steps_iff(1)[of ⋆ "ts ·_list ϑ" "transaction_strand T ·_lsst ϑ"]
            set_mono_prefix[OF T(1)[unfolded ϑ_def[symmetric]]]
      unfolding tb by auto
    hence "⟨f [⟨c⟩_c, k]⟩_t ∈ ⋃{set ts |ts. ⟨⋆, receive⟨ts⟩⟩ ∈ set (A ·_lsst I)}"
      using t(2) subst_lsst_memI[of "⟨⋆,receive⟨ts ·_list ϑ⟩⟩" A I] by force
    thus False
      using A(2) unfolding declassified_lsst_def by auto
  qed

  have "y' ∉ fv_set (ik_lsst A)"
    using y'' fv_ik_subset_vars_sst'[of "unlabel A"] by blast
  hence 1: "ik_lsst A ·_set I = ik_lsst A ·_set J"
    unfolding J_def by (metis (no_types, lifting) fv_subset image_cong in_mono repl_invariance)

  have "(k,⟨d⟩_s) ∉ dbupd_sst (unlabel A) I {}"
    using 0 db_sst_set_is_dbupd_sst[of "unlabel A" I "[]"]
    unfolding db_sst_def by force
  hence "(k,⟨d⟩_s) ∉ dbupd_sst (unlabel A) J {}"
    using y'' vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
          db_sst_subst_swap[of "unlabel A" I J "[]"]
          db_sst_set_is_dbupd_sst[of "unlabel A" _ "[]"]
    unfolding db_sst_def J_def by (metis (no_types, lifting) Un_iff empty_set fun_upd_other)
  hence "((Var y' · J, ⟨d⟩_s · J) ∉ dbupd_sst (unlabel A) J {})"
    unfolding J_def fk by simp
  hence "strand_sem_stateful (ik_lsst A ·_set J) (dbupd_sst (unlabel A) J {})
          (unlabel [⟨n, ⟨Var y' not in ⟨d⟩_s⟩⟩]) J"
    using stateful_strand_sem_NegChecks_no_bvars(1)[
            of "ik_lsst A ·_set J" "dbupd_sst (unlabel A) J {}" "Var y'" "⟨d⟩_s" J]
    by simp
  hence 2: "strand_sem_stateful {} {} (unlabel (A@[⟨n, ⟨Var y' not in ⟨d⟩_s⟩⟩])) J"
    using A'' y' y'' vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
```

```
        strand_sem_append_stateful[
          of "{}" "{}" "unlabel A" "unlabel [⟨n, ⟨Var y' not in ⟨d⟩s⟩⟩]" J]
      unfolding J_def by simp

    have B: "B ∈ reachable_constraints P"
      using reachable_constraints.step[OF A(1) P_Tatt _ _ α, of Var Var]
      unfolding B_def Tatt_def transaction_decl_subst_def transaction_fresh_subst_def by simp

    have Tatt': "dual_lsst (transaction_strand Tatt ·lsst α) =
                  [⟨⋆, send⟨[occurs (Var y')]⟩⟩,
                   ⟨1, send⟨[⟨f [⟨c⟩c, Var y']⟩t]⟩⟩,
                   ⟨1', ⟨Var y' not in ⟨d⟩s⟩⟩,
                   ⟨n, receive⟨[attack⟨ln n⟩]⟩⟩]"
      using y'
      unfolding Tatt_def transaction_strand_def dual_lsst_def subst_apply_labeled_stateful_strand_def
      by auto

    have J: "wt_subst J" "interpretation_subst J" "wf_trms (subst_range J)"
      unfolding J_def
      subgoal using wt_subst_subst_upd[OF I(1)] k(1) y'(3) by simp
      subgoal by (metis I(2) k(2) fun_upd_apply interpretation_grounds_all interpretation_substI)
      subgoal using I(3) k(3) by fastforce
      done

    have 3: "ik_lsst A ·set J ⊢ ⟨f [⟨c⟩c, Var y']⟩t · J"
      using 1 fk fk' unfolding J_def by auto

    have 4: "ik_lsst A ·set J ⊢ occurs (Var y') · J"
      using reachable_constraints_occurs_fact_ik_case'[
              OF A(1) P(1,2) constraint_model_Val_const_in_constr_prefix'[
                                OF A(1) A' P(1) k_in_ik[unfolded fk kn]]]
            intruder_deduct.Axiom[of "occurs k" "ik_lsst A ·set J"]
      unfolding J_def fk kn by fastforce

    have "strand_sem_stateful {} {} (unlabel A) J"
      using 2 strand_sem_append_stateful by force
    hence "strand_sem_stateful {} {}
            (unlabel (A@[⟨⋆, send⟨[occurs (Var y')]⟩⟩,
                         ⟨n, send⟨[⟨f [⟨c⟩c, Var y']⟩t]⟩⟩,
                         ⟨n, ⟨Var y' not in ⟨d⟩s⟩⟩])) J"
      using 2 3 4 strand_sem_append_stateful[of "{}" "{}" _ _ J]
      unfolding unlabel_def ik_sst_def by force
    hence "strand_sem_stateful {} {} (unlabel (B@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])) J"
      using strand_sem_receive_send_append[of "{}" "{}" _ J "attack⟨ln n⟩"]
            strand_sem_append_stateful[of "{}" "{}" _ _ J]
      unfolding B_def Tatt' by simp
    hence "welltyped_constraint_model J (B@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
      using B J unfolding defs' by blast
    thus False using B(1) P_sec by blast
  qed
qed

lemma welltyped_leakage_free_short_term_secretI':
  fixes c x y f n d l l' τ
  defines "s ≡ ⟨f [⟨c⟩c, Var (TAtom τ,x)]⟩t"
    and "Tatt ≡ Transaction (λ(). []) []
          [(l, receive⟨[⟨f [⟨c⟩c, Var (TAtom τ,y)]⟩t]⟩)]
          [(l', ⟨Var (TAtom τ,y) not in ⟨d⟩s⟩)]
          []
          [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]"
  assumes P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. ∀x ∈ set (unlabel (transaction_updates T)).
```

```
      is_Update x ⟶ is_Fun (the_set_term x)"
    and subterms_sec:
      "∀𝒜 ∈ reachable_constraints P. ∄ℐτ. welltyped_constraint_model ℐτ (𝒜@[⟨⋆, send⟨[⟨c⟩c]⟩⟩])"
    and P_sec:
      "∀𝒜 ∈ reachable_constraints P. ∄ℐτ.
        welltyped_constraint_model ℐτ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
    and P_Tatt: "Tatt ∈ set P"
    and P_d:
      "∀T ∈ set P. ∀a ∈ set (transaction_updates T).
        is_Insert (snd a) ∧ the_set_term (snd a) = ⟨d⟩s ⟶
          transaction_send T ≠ [] ∧ (let (l,b) = hd (transaction_send T)
            in l = ⋆ ∧ is_Send b ∧ ⟨f [⟨c⟩c, the_elem_term (snd a)]⟩t ∈ set (the_msgs b))"
  shows "welltyped_leakage_free_protocol [s] P"
proof -
  define Q where "Q ≡ λM t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}"
  define Sec where "Sec ≡ {t · δ | t δ. Q (set [s]) t δ} - {m. {} ⊢c m}"

  define f' where "f' ≡ λ(T,ξ,σ::('fun,'atom,'sets,'lbl) prot_subst,α).
                          dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
  define g' where "g' ≡ concat ∘ map f'"

  note defs = Sec_def Q_def
  note defs' = welltyped_constraint_model_def constraint_model_def

  show ?thesis
  proof (rule ccontr)
    assume "¬welltyped_leakage_free_protocol [s] P"
    then obtain A I k where A:
        "A ∈ reachable_constraints P" "⟨f [⟨c⟩c, k]⟩t ∈ Sec - declassified_lsst A I"
        "welltyped_constraint_model I (A@[⟨⋆, send⟨[⟨f [⟨c⟩c, k]⟩t]⟩⟩])"
      unfolding welltyped_leakage_free_protocol_def defs s_def by fastforce

    have I: "wt_subst I" "interpretation_subst I" "wf_trms (subst_range I)"
      using A(3) unfolding defs' by (blast,blast,blast)

    note A' = welltyped_constraint_model_prefix[OF A(3)]

    have "strand_sem_stateful {} {} (unlabel A) I"
      using A' unfolding defs' by simp
    hence A'': "strand_sem_stateful {} {} (unlabel A) (I(z := k))"
      when z: "z ∉ fv_lsst A" for z
      using z strand_sem_model_swap[of "unlabel A" I "I(z := k)" "{}" "{}"] by auto

    obtain δ where δ:
        "δ (TAtom τ,x) = k" "wt_subst δ" "wf_trms (subst_range δ)" "fv (δ (TAtom τ,x)) = {}"
      using A(2) unfolding defs s_def by auto

    have k: "Γ k = TAtom τ" "fv k = {}" "wf_trm k"
      subgoal using δ(1) wt_subst_trm''[OF δ(2), of "Var (TAtom τ,x)"] by simp
      subgoal using δ(1,4) by blast
      subgoal using δ(1,3) by force
      done
    then obtain fk where fk: "k = Fun fk []"
      using const_type_inv_wf by (cases k) auto

    have fk': "ik_lsst A ·set I ⊢ ⟨f [⟨c⟩c, k]⟩t"
      using fk welltyped_constraint_model_deduct_split(2)[OF A(3)] by auto

    obtain α::"('fun,'atom,'sets,'lbl) prot_subst"
      where α: "transaction_renaming_subst α P (vars_lsst A)"
      unfolding transaction_renaming_subst_def by blast

    obtain y' yn' where y': "α (TAtom τ,y) = Var y'" "y ≠ yn'" "y' = (TAtom τ,yn')"
```

```
    using transaction_renaming_subst_is_renaming(1,2)[OF α] by force

  define B where "B ≡ A@dual_{lsst} (transaction_strand Tatt ·_{lsst} α)"
  define J where "J ≡ I(y' := k)"

  have "y' ∈ range_vars α"
    using y'(1) transaction_renaming_subst_is_renaming(3)[OF α]
    by (metis (no_types, lifting) in_mono subst_fv_imgI term.set_intros(3))
  hence y'': "y' ∉ vars_{lsst} A"
    using transaction_renaming_subst_vars_disj(6)[OF α] by blast

  have 0: "(k,⟨d⟩_s) ∉ set (db_{lsst} A I)"
  proof
    assume a: "(k,⟨d⟩_s) ∈ set (db_{lsst} A I)"
    obtain l t t' where t: "(l,insert⟨t,t'⟩) ∈ set A" "t · I = k" "t' · I = ⟨d⟩_s"
      using db_{sst}_in_cases[OF a[unfolded db_{sst}_def]] unfolding unlabel_def by auto

    obtain T b B α σ ξ where T:
        "prefix (B@dual_{lsst} (transaction_strand T ·_{lsst} ξ ∘_s σ ∘_s α)) A"
        "T ∈ set P" "transaction_decl_subst ξ T"
        "transaction_fresh_subst σ T (trms_{lsst} B)" "transaction_renaming_subst α P (vars_{lsst} B)"
        "b ∈ set (transaction_strand T)" "(l, insert⟨t,t'⟩) = dual_{lsstp} b ·_{lsstp} ξ ∘_s σ ∘_s α"
        "fst (l, insert⟨t,t'⟩) = fst b"
      using reachable_constraints_transaction_action_obtain[OF A(1) t(1)] by metis

    define ϑ where "ϑ ≡ ξ ∘_s σ ∘_s α"

    obtain b' where "b = (l,b')"
      using T(8) by (cases b) simp
    then obtain tb tb'
        where b': "b = (l,insert⟨tb,tb'⟩)"
          and tb: "t = tb · ϑ"
          and tb': "t' = tb' · ϑ"
      using T(7) unfolding ϑ_def by (cases b') auto

    note T_wf = bspec[OF P(1) T(2)] bspec[OF P(2) T(2)]

    have b: "b ∈ set (transaction_updates T)"
      using transaction_strand_memberD[OF T(6)[unfolded b']]
            wellformed_transaction_cases[OF T_wf(1)]
      unfolding b' by blast

    have "is_Fun tb'"
      using bspec[OF P(2) T(2)]
            wellformed_transaction_strand_unlabel_memberD(6)[
                  OF T_wf(1) unlabel_in[OF T(6)[unfolded b']]]
      by fastforce
    hence *: "tb' = ⟨d⟩_s"
      using t(3) unfolding b' tb' by force

    have "is_Insert (snd b)" "the_set_term (snd b) = ⟨d⟩_s" "the_elem_term (snd b) = tb"
      unfolding b' * by simp_all
    hence "transaction_send T ≠ []"
          "let (l, a) = hd (transaction_send T)
           in l = ⋆ ∧ is_Send a ∧ ⟨f [⟨c⟩_c, tb]⟩_t ∈ set (the_msgs a)"
      using P_d T(2) b by (fast,fast)
    hence "∃ts. ⟨⋆,send⟨ts⟩⟩ ∈ set (transaction_send T) ∧ ⟨f [⟨c⟩_c, tb]⟩_t ∈ set ts"
      unfolding is_Send_def by (cases "transaction_send T") auto
    then obtain ts where ts: "⟨⋆,send⟨ts⟩⟩ ∈ set (transaction_strand T)" "⟨f [⟨c⟩_c, tb]⟩_t ∈ set ts"
      unfolding transaction_strand_def by auto

    have "⟨⋆,receive⟨ts ·_{list} ϑ⟩⟩ ∈ set A" "⟨f [⟨c⟩_c, t]⟩_t ∈ set (ts ·_{list} ϑ)"
      using subst_lsst_memI[OF ts(1), of ϑ] subst_set_map[OF ts(2), of ϑ]
```

```
          dual_lsst_steps_iff(1)[of ⋆ "ts ·_list ϑ" "transaction_strand T ·_lsst ϑ"]
          set_mono_prefix[OF T(1)[unfolded ϑ_def[symmetric]]]
    unfolding tb by auto
  hence "⟨f [⟨c⟩_c, k]⟩_t ∈ ⋃{set ts |ts. ⟨⋆, receive⟨ts⟩⟩ ∈ set (A ·_lsst I)}"
    using t(2) subst_lsst_memI[of "⟨⋆,receive⟨ts ·_list ϑ⟩⟩" A I] by force
  thus False
    using A(2) unfolding declassified_lsst_def by auto
qed

have "y' ∉ fv_set (ik_lsst A)"
  using y'' fv_ik_subset_vars_sst'[of "unlabel A"] by blast
hence 1: "ik_lsst A ·_set I = ik_lsst A ·_set J"
  unfolding J_def by (metis (no_types, lifting) fv_subset image_cong in_mono repl_invariance)

have "(k,⟨d⟩_s) ∉ dbupd_sst (unlabel A) I {}"
  using 0 db_sst_set_is_dbupd_sst[of "unlabel A" I "[]"]
  unfolding db_sst_def by force
hence "(k,⟨d⟩_s) ∉ dbupd_sst (unlabel A) J {}"
  using y'' vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
        db_sst_subst_swap[of "unlabel A" I J "[]"]
        db_sst_set_is_dbupd_sst[of "unlabel A" _ "[]"]
  unfolding db_sst_def J_def by (metis (no_types, lifting) Un_iff empty_set fun_upd_other)
hence "((Var y' · J, ⟨d⟩_s · J) ∉ dbupd_sst (unlabel A) J {})"
  unfolding J_def fk by simp
hence "strand_sem_stateful (ik_lsst A ·_set J) (dbupd_sst (unlabel A) J {})
        (unlabel [⟨n, ⟨Var y' not in ⟨d⟩_s⟩⟩]) J"
  using stateful_strand_sem_NegChecks_no_bvars(1)[
          of "ik_lsst A ·_set J" "dbupd_sst (unlabel A) J {}" "Var y'" "⟨d⟩_s" J]
  by simp
hence 2: "strand_sem_stateful {} {} (unlabel (A@[⟨n, ⟨Var y' not in ⟨d⟩_s⟩⟩])) J"
  using A'' y' y'' vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
        strand_sem_append_stateful[
          of "{}" "{}" "unlabel A" "unlabel [⟨n, ⟨Var y' not in ⟨d⟩_s⟩⟩]" J]
  unfolding J_def by simp

have B: "B ∈ reachable_constraints P"
  using reachable_constraints.step[OF A(1) P_Tatt _ _ α, of Var Var]
  unfolding B_def Tatt_def transaction_decl_subst_def transaction_fresh_subst_def by simp

have Tatt': "dual_lsst (transaction_strand Tatt ·_lsst α) =
              [(l, send⟨[⟨f [⟨c⟩_c, Var y']⟩_t]⟩),
               (l', ⟨Var y' not in ⟨d⟩_s⟩),
               ⟨n, receive⟨[attack⟨ln n⟩]⟩⟩]"
  using y'
  unfolding Tatt_def transaction_strand_def dual_lsst_def subst_apply_labeled_stateful_strand_def
  by auto

have J: "wt_subst J" "interpretation_subst J" "wf_trms (subst_range J)"
  unfolding J_def
  subgoal using wt_subst_subst_upd[OF I(1)] k(1) y'(3) by simp
  subgoal by (metis I(2) k(2) fun_upd_apply interpretation_grounds_all interpretation_substI)
  subgoal using I(3) k(3) by fastforce
  done

have 3: "ik_lsst A ·_set J ⊢ ⟨f [⟨c⟩_c, Var y']⟩_t · J"
  using 1 fk fk' unfolding J_def by auto

have "strand_sem_stateful {} {} (unlabel A) J"
  using 2 strand_sem_append_stateful by force
hence "strand_sem_stateful {} {}
        (unlabel (A@[⟨n, send⟨[⟨f [⟨c⟩_c, Var y']⟩_t]⟩⟩,
                      ⟨n, ⟨Var y' not in ⟨d⟩_s⟩⟩])) J"
  using 2 3 strand_sem_append_stateful[of "{}" "{}" _ _ J]
```

```
      unfolding unlabel_def ik_sst_def by force
    hence "strand_sem_stateful {} {} (unlabel (B@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])) J"
      using strand_sem_receive_send_append[of "{}" "{}" _ J "attack⟨ln n⟩"]
            strand_sem_append_stateful[of "{}" "{}" _ _ J]
      unfolding B_def Tatt' by simp
    hence "welltyped_constraint_model J (B@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
      using B J unfolding defs' by blast
    thus False using B(1) P_sec by blast
  qed
qed

definition welltyped_leakage_free_invkey_conditions' where
  "welltyped_leakage_free_invkey_conditions' invfun privfunsec declassifiedset S n P ≡
    let f = λs. is_Var s ∧ fst (the_Var s) = TAtom Value;
        g = λs. is_Fun s ∧ args s = [] ∧ is_Set (the_Fun s) ∧
                arity_s (the_Set (the_Fun s)) = 0;
        h = λs. is_Fun s ∧ args s = [] ∧ is_Fu (the_Fun s) ∧
                public_f (the_Fu (the_Fun s)) ∧ arity_f (the_Fu (the_Fun s)) = 0
    in (∀s∈set S.
          f s ∨
          (is_Fun s ∧ the_Fun s = Pair ∧ length (args s) = 2 ∧ g (args s ! 1)) ∨
          g s ∨ h s ∨ s = ⟨privfunsec⟩_c ∨ s = Fun OccursSec [] ∨
          (is_Fun s ∧ the_Fun s = OccursFact ∧ length (args s) = 2 ∧
           args s ! 0 = Fun OccursSec []) ∨
          (is_Fun s ∧ the_Fun s = Fu invfun ∧ length (args s) = 2 ∧
           args s ! 0 = ⟨privfunsec⟩_c ∧ f (args s ! 1)) ∨
          (is_Fun s ∧ is_Fu (the_Fun s) ∧ fv s = {} ∧
           Transaction (λ(). []) [] [⟨n, receive⟨[s]⟩⟩] [] [] [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]∈set P)) ∧
       (¬public_f privfunsec ∧ arity_f privfunsec = 0 ∧ Γ_f privfunsec ≠ None) ∧
       (∀T∈set P. transaction_fresh T ≠ [] ⟶
         transaction_send T ≠ [] ∧
         (let (l, a) = hd (transaction_send T)
          in l = ⋆ ∧ is_Send a ∧ Var ` set (transaction_fresh T) ⊆ set (the_msgs a))) ∧
       (∀T∈set P. ∀x∈vars_transaction T. is_Var (Γ_v x)) ∧
       (∀T∈set P. ∀x∈set (transaction_fresh T). Γ_v x = Var Value ∨ (∃a. Γ_v x = ⟨a⟩_τa)) ∧
       (∀T∈set P. ∀f∈⋃(funs_term ` trms_lsst (transaction_send T)). ¬is_Set f) ∧
       (∀T∈set P. ∀r∈set (transaction_send T).
          OccursFact ∈ ⋃(funs_term ` trms_sstp (snd r)) ⟶ has_LabelS r) ∧
       (∀T∈set P. ∀t∈subterms_set (trms_lsst (transaction_send T)).
          ⟨privfunsec⟩_c ∉ set (snd (Ana t))) ∧
       (∀T∈set P. ⟨privfunsec⟩_c ∉ trms_lsst (transaction_send T)) ∧
       (∀T∈set P. ∀a∈set (transaction_updates T).
            is_Insert (snd a) ∧ the_set_term (snd a) = ⟨declassifiedset⟩_s ⟶
              transaction_send T ≠ [] ∧
              (let (l, b) = hd (transaction_send T)
               in l = ⋆ ∧ is_Send b ∧
                  ⟨invfun [⟨privfunsec⟩_c, the_elem_term (snd a)]⟩_t ∈ set (the_msgs b)))"

definition welltyped_leakage_free_invkey_conditions where
  "welltyped_leakage_free_invkey_conditions invfun privfunsec declassifiedset S n P ≡
    let Tatt = λR. Transaction (λ(). []) []
                    (R@[⟨n, receive⟨[⟨invfun [⟨privfunsec⟩_c, ⟨0: value⟩_v]⟩_t]⟩⟩])
                    [⟨⋆, ⟨⟨0: value⟩_v not in ⟨declassifiedset⟩_s⟩⟩]
                    []
                    [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]
    in welltyped_leakage_free_invkey_conditions' invfun privfunsec declassifiedset S n P ∧
       has_initial_value_producing_transaction P ∧
       (if Tatt [⟨⋆, receive⟨[occurs ⟨0: value⟩_v]⟩⟩] ∈ set P
        then ∀T∈set P. admissible_transaction' T ∧ admissible_transaction_occurs_checks T
        else Tatt [] ∈ set P ∧
             (∀T∈set P. wellformed_transaction T) ∧
             (∀T∈set P. admissible_transaction_terms T) ∧
             (∀T∈set P. bvars_transaction T = {}) ∧
```

```
        (∀T∈set P. transaction_decl T () = []) ∧
        (∀T∈set P. ∀x∈set (transaction_fresh T). let τ = fst x
           in τ = TAtom Value ∧ τ ≠ Γ ⟨privfunsec⟩_c) ∧
        (∀T∈set P. ∀x∈vars_transaction T. let τ = fst x
           in is_Var τ ∧ (the_Var τ = Value ∨ is_Atom (the_Var τ)) ∧ τ ≠ Γ ⟨privfunsec⟩_c) ∧
        (∀T∈set P. ∀t∈subterms_set (trms_lsst (transaction_send T)).
           Fun OccursSec [] ∉ set (snd (Ana t))) ∧
        (∀T∈set P. Fun OccursSec [] ∉ trms_lsst (transaction_send T)) ∧
        (∀T∈set P. ∀x∈set (unlabel (transaction_updates T)).
           is_Update x ⟶ is_Fun (the_set_term x)) ∧
        (∀s∈set S. is_Fun s ⟶ the_Fun s ≠ OccursFact))"

lemma welltyped_leakage_free_invkeyI:
  assumes P_wt_secure: "∀𝒜 ∈ reachable_constraints P.
                     ∄ℐ. welltyped_constraint_model ℐ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
    and a: "welltyped_leakage_free_invkey_conditions invfun privsec declassset S n P"
  shows "welltyped_leakage_free_protocol S P"
proof -
  let ?Tatt' = "λR C. Transaction (λ(). []) [] R C [] [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]
                    ::('fun,'atom,'sets,'lbl) prot_transaction"
  let ?Tatt = "λR. ?Tatt' (R@[⟨n, receive⟨[⟨invfun [⟨privsec⟩_c, ⟨0: value⟩_v]⟩_t]⟩⟩])
                      [⟨⋆, ⟨⟨0: value⟩_v not in ⟨declassset⟩_s⟩⟩]"

  define Tatt1 where "Tatt1 ≡ ?Tatt [⟨⋆, receive⟨[occurs ⟨0: value⟩_v]⟩⟩]"
  define Tatt2 where "Tatt2 ≡ ?Tatt []"
  define Tatt_lts where "Tatt_lts ≡ λs. ?Tatt' [⟨n, receive⟨[s]⟩⟩] []"

  note defs = welltyped_leakage_free_invkey_conditions_def Let_def

  note defs' = defs welltyped_leakage_free_invkey_conditions'_def

  note Tatts = Tatt1_def Tatt2_def Tatt_lts_def

  obtain at where 0: "¬public_f privsec" "arity_f privsec = 0" "Γ_f privsec = Some at"
    using a unfolding defs' by fast

  have *: "∀T∈set P. admissible_transaction' T" "∀T∈set P. admissible_transaction_occurs_checks T"
    when "Tatt1 ∈ set P"
    using a that unfolding defs Tatts by (meson,meson)

  have **: "Tatt1 ∈ set P ∨ Tatt2 ∈ set P" using a unfolding defs Tatts by argo

  have ***:
    "∀T∈set P. ∀x∈set (transaction_fresh T). Γ_v x = TAtom Value ∧ Γ_v x ≠ Γ ⟨privsec⟩_c"
    "∀T∈set P. ∀x∈vars_transaction T. ∃a. Γ_v x = TAtom a ∧
          (a = Value ∨ (∃b. a = Atom b)) ∧ TAtom a ≠ Γ ⟨privsec⟩_c"
    when "Tatt1 ∉ set P"
    subgoal using a that Γ_v_TAtom''(2) unfolding defs Tatts by metis
    subgoal
      using a that Γ_v_TAtom''(1,2)
      unfolding defs Tatts[symmetric] is_Atom_def is_Var_def by fastforce
    done

  have ****: "s ≠ occurs x"
    when "Tatt1 ∉ set P" "s ∈ set S" for s x
    using a that ** unfolding defs Tatts the_Fun_def by fastforce

  have 1:
      "∀T∈set P. transaction_fresh T ≠ [] ⟶
        transaction_send T ≠ [] ∧
        (let (l, a) = hd (transaction_send T)
         in l = ⋆ ∧ is_Send a ∧ Var ` set (transaction_fresh T) ⊆ set (the_msgs a))"
      "∀T∈set P. ∀x∈vars_transaction T. is_Var (Γ_v x)"
```

```
    "∀T∈set P. ∀x∈set (transaction_fresh T). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = ⟨a⟩_τa)"
    "∀T∈set P. ∀f∈⋃(funs_term ` trms_lsst (transaction_send T)). ¬is_Set f"
    "∀T∈set P. ∀r∈set (transaction_send T).
      OccursFact ∈ ⋃(funs_term ` trms_sstp (snd r)) ⟶ has_LabelS r"
    "∀T∈set P. ∀t∈subterms_set (trms_lsst (transaction_send T)). ⟨privsec⟩_c ∉ set (snd (Ana t))"
    "∀T∈set P. ⟨privsec⟩_c ∉ trms_lsst (transaction_send T)"
    "∀T∈set P. ∀a∈set (transaction_updates T).
      is_Insert (snd a) ∧ the_set_term (snd a) = ⟨declassset⟩_s ⟶
      transaction_send T ≠ [] ∧
      (let (l, b) = hd (transaction_send T)
       in l = ⋆ ∧ is_Send b ∧
          ⟨invfun [⟨privsec⟩_c, the_elem_term (snd a)]⟩_t ∈ set (the_msgs b))"
  using a unfolding defs' by (meson,meson,meson,meson,meson,meson,meson,meson)

have 2:
    "∀T∈set P. wellformed_transaction T"
    "∀T∈set P. ∀x∈vars_transaction T ∪ set (transaction_fresh T). Γ ⟨privsec⟩_c ≠ Γ_v x"
    "∀T∈set P. admissible_transaction_terms T"
    "∀T∈set P. ∀x∈set (transaction_fresh T). Γ_v x = TAtom Value"
    "∀T∈set P. transaction_decl T () = []"
    "∀T∈set P. ∀x∈vars_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = ⟨a⟩_τa)"
    "∀T∈set P. ∀x∈vars_transaction T ∪ set (transaction_fresh T). Γ_v x ≠ TAtom SetType"
    "∀T∈set P. ∀x∈vars_transaction T ∪ set (transaction_fresh T). Γ_v x ≠ TAtom OccursSecType"
    "∀T∈set P. ∀t∈subterms_set (trms_lsst (transaction_send T)). Fun OccursSec [] ∉ set (snd (Ana t))"
    "∀T∈set P. Fun OccursSec [] ∉ trms_lsst (transaction_send T)"
    "∀T∈set P. bvars_transaction T = {}"
    "∀T∈set P. ∀x∈set (unlabel (transaction_updates T)). is_Update x ⟶ is_Fun (the_set_term x)"
  subgoal using a * unfolding defs by (metis admissible_transaction_is_wellformed_transaction(1))
  subgoal
    apply (cases "Tatt1 ∈ set P")
    subgoal using a * admissible_transactionE(2,3) Γ_Fu_simps(4) unfolding defs Tatts by force
    subgoal using a Γ_Fu_simps(4) unfolding defs Tatts by fastforce
    done
  subgoal using a * admissible_transaction_is_wellformed_transaction(4) unfolding defs by metis
  subgoal using a * admissible_transactionE(2) unfolding defs Tatts[symmetric] by fastforce
  subgoal using a * admissible_transactionE(1) unfolding defs Tatts[symmetric] by metis
  subgoal using * *** admissible_transactionE(3) by fast
  subgoal using * *** admissible_transactionE(2,3) by (cases "Tatt1 ∈ set P") (force, fastforce)
  subgoal using * *** admissible_transactionE(2,3) by (cases "Tatt1 ∈ set P") (force, fastforce)
  subgoal using a * admissible_transaction_occurs_checksE6 unfolding defs by metis
  subgoal using a * admissible_transaction_occurs_checksE5 unfolding defs by metis
  subgoal
    using a * admissible_transaction_no_bvars(2)
    unfolding defs Tatts[symmetric] by fastforce
  subgoal
    using a * admissible_transaction_is_wellformed_transaction(3)
    unfolding defs Tatts[symmetric] admissible_transaction_updates_def by fastforce
  done

have Tatt_lts_case:
    "∃ϑ. wt_subst ϑ ∧ inj_on ϑ (fv s) ∧ ϑ ` fv s ⊆ range Var ∧
         ?Tatt' [⟨n, receive⟨[s · ϑ]⟩⟩] [] ∈ set P"
  when s: "fv s = {}" "Tatt_lts s ∈ set P" for s
proof -
  have "wt_subst Var" "inj_on Var (fv s)" "Var ` fv s ⊆ range Var" "s · Var = s"
    using s(1) by simp_all
  thus ?thesis using s(2) unfolding Tatts by metis
qed

have Tatt1_case:
    "?Tatt' [⟨⋆, receive⟨[occurs ⟨0: value⟩_v]⟩⟩, ⟨n, receive⟨[⟨invfun [⟨privsec⟩_c, ⟨0: value⟩_v]⟩_t]⟩⟩]
            [⟨⋆, ⟨⟨0: value⟩_v not in ⟨declassset⟩_s⟩⟩] ∈ set P"
    when "Tatt1 ∈ set P"
```

```
  using that unfolding Tatts by auto

have Tatt2_case:
    "?Tatt' [⟨n, receive⟨[⟨invfun [⟨privsec⟩c, ⟨0: value⟩v]⟩t]⟩⟩]
            [⟨⋆, ⟨⟨0: value⟩v not in ⟨declassset⟩s⟩⟩] ∈ set P"
  when "Tatt2 ∈ set P"
  using that unfolding Tatts by auto

note 3 = pair_def case_prod_conv
note 4 = welltyped_leakage_free_priv_constI'[OF 0(1-3) 2(1,2) 1(2,3,6,7)]
note 5 = welltyped_leakage_free_setop_pairI[OF 2(1,6) 1(4) 2(4,5,3), unfolded 3]
         welltyped_leakage_free_set_constI(2)[OF 2(1) 1(4) 2(7) 1(2,3), unfolded 3]
         welltyped_leakage_free_pub_constI
         4(2)
         welltyped_leakage_free_occurssec_constI(2)[OF 2(1,8-10) 1(2,3)]
         welltyped_leakage_free_value_constI[OF 2(1,3,5,11) 1(1)]
         welltyped_leakage_free_short_term_secretI'[
           OF 2(1,12) 4(1) P_wt_secure Tatt2_case 1(8)]

         welltyped_leakage_free_long_term_secretI[OF P_wt_secure Tatt_lts_case]

         welltyped_leakage_free_short_term_secretI[
           OF * 1(3) 4(1) P_wt_secure Tatt1_case 1(8)]
         welltyped_leakage_free_occurs_factI[OF * 1(5)]

         ** ****

have 6: "is_Fun s ∧ length (args s) = 2 ⟷ (∃f t u. s = Fun f [t, u])"
  for s::"('fun,'atom,'sets,'lbl) prot_term"
  by auto

define pubconst_cond where
  "pubconst_cond ≡ λs::('fun,'atom,'sets,'lbl) prot_term.
    is_Fun s ∧ args s = [] ∧ is_Fu (the_Fun s) ∧
    publicf (the_Fu (the_Fun s)) ∧ arityf (the_Fu (the_Fun s)) = 0"

define valuevar_cond where
  "valuevar_cond ≡ λs::('fun,'atom,'sets,'lbl) prot_term.
    is_Var s ∧ fst (the_Var s) = TAtom Value"

define setconst_cond where
  "setconst_cond ≡ λs::('fun,'atom,'sets,'lbl) prot_term.
    is_Fun s ∧ args s = [] ∧ is_Set (the_Fun s) ∧ aritys (the_Set (the_Fun s)) = 0"

define setop_pair_cond where
  "setop_pair_cond ≡ λs.
    is_Fun s ∧ the_Fun s = Pair ∧ length (args s) = 2 ∧ setconst_cond (args s ! 1)"

define occursfact_cond where
  "occursfact_cond ≡ λs::('fun,'atom,'sets,'lbl) prot_term.
    is_Fun s ∧ the_Fun s = OccursFact ∧ length (args s) = 2 ∧
    args s ! 0 = Fun OccursSec []"

define invkey_cond where
  "invkey_cond ≡ λs.
    is_Fun s ∧ the_Fun s = Fu invfun ∧ length (args s) = 2 ∧
    args s ! 0 = ⟨privsec⟩c ∧ valuevar_cond (args s ! 1)"

define ground_lts_cond where
  "ground_lts_cond ≡ λs. is_Fun s ∧ is_Fu (the_Fun s) ∧ fv s = {} ∧ Tatt_lts s ∈ set P"

note cond_defs =
  pubconst_cond_def valuevar_cond_def setconst_cond_def setop_pair_cond_def
```

```
    occursfact_cond_def invkey_cond_def ground_lts_cond_def

  have "(∃m. s = ⟨m: value⟩_v) ⟷ valuevar_cond s"
       "(∃x c. arity_s c = 0 ∧ s = Fun Pair [x, ⟨c⟩_s]) ⟷ setop_pair_cond s"
       "(∃c. arity_s c = 0 ∧ s = ⟨c⟩_s) ⟷ setconst_cond s"
       "(∃c. public_f c ∧ arity_f c = 0 ∧ s = ⟨c⟩_c) ⟷ pubconst_cond s"
       "(∃x. s = occurs x) ⟷ occursfact_cond s"
       "(∃x. s = ⟨invfun [⟨privsec⟩_c, ⟨x: value⟩_v]⟩_t) ⟷ invkey_cond s"
       "(∃f ts. s = ⟨f ts⟩_t ∧ fv s = {} ∧ Tatt_lts s ∈ set P) ⟷ ground_lts_cond s"
    for s::"('fun,'atom,'sets,'lbl) prot_term"
    unfolding is_Set_def the_Set_def is_Fu_def cond_defs
    by (fastforce,use 6[of s] in fastforce,fastforce,force,fastforce,fastforce,fastforce)
  moreover have
      "(∀s ∈ set S. valuevar_cond s ∨ setop_pair_cond s ∨ setconst_cond s ∨ pubconst_cond s ∨
                    s = ⟨privsec⟩_c ∨ s = Fun OccursSec [] ∨ occursfact_cond s ∨ invkey_cond s ∨
                    ground_lts_cond s) ∧
       (¬public_f privsec ∧ arity_f privsec = 0 ∧ Γ_f privsec ≠ None)"
    using a unfolding defs' cond_defs Tatts by meson
  ultimately have 7:
      "∀s ∈ set S.
        (∃x c. arity_s c = 0 ∧ s = Fun Pair [x, ⟨c⟩_s]) ∨
        (∃c. arity_s c = 0 ∧ s = ⟨c⟩_s) ∨
        (∃c. public_f c ∧ arity_f c = 0 ∧ s = ⟨c⟩_c) ∨
        s = ⟨privsec⟩_c ∨ s = Fun OccursSec [] ∨
        (∃m. s = ⟨m: value⟩_v) ∨
        (∃x. s = occurs x) ∨
        (∃x. s = ⟨invfun [⟨privsec⟩_c, ⟨x: value⟩_v]⟩_t) ∨
        (∃f ts. s = ⟨f ts⟩_t ∧ fv s = {} ∧ Tatt_lts s ∈ set P)"
    unfolding Let_def by fastforce

  show ?thesis
    by (rule iffD2[OF welltyped_leakage_free_protocol_pointwise]; unfold list_all_iff; intro ballI)
       (use bspec[OF 7] 5 in blast)
qed

end

end

locale composable_stateful_protocols =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets,nat) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"nat"
    and label_witness2::"nat"
  +
  fixes Pc::"('fun,'atom,'sets,nat) prot_transaction list"
    and Ps Ps_with_star_projs::"('fun,'atom,'sets,nat) prot_transaction list list"
    and Pc_SMP Sec_symbolic::"('fun,'atom,'sets,nat) prot_term list"
    and Ps_GSMPs::"(nat × ('fun,'atom,'sets,nat) prot_term list) list"
  assumes Pc_def: "Pc = concat Ps"
    and Ps_with_star_projs_def: "let Pc' = Pc; L = [0..<length Ps] in
          Ps_with_star_projs = map (λn. (map (transaction_proj n) Pc')) L ∧
          set L = set (remdups (concat (
                          map (λT. map (the_LabelN ∘ fst)
                                       (filter (Not ∘ has_LabelS) (transaction_strand T)))
                              Pc')))"
    and Pc_wellformed_composable:
          "list_all (list_all (Not ∘ has_LabelS) ∘ tl ∘ transaction_send) Pc"
          "pm.wellformed_composable_protocols Ps Pc_SMP"
```

```
        "pm.wellformed_composable_protocols' Ps"
        "pm.composable_protocols Ps Ps_GSMPs Sec_symbolic"
begin

theorem composed_protocol_preserves_component_goals:
  assumes components_leakage_free:
      "list_all (pm.welltyped_leakage_free_protocol Sec_symbolic) Ps_with_star_projs"
    and n_def: "n < length Ps_with_star_projs"
    and P_def: "P = Ps_with_star_projs ! n"
    and P_welltyped_secure:
      "∀𝒜 ∈ pm.reachable_constraints P. ∄ℐ.
          pm.welltyped_constraint_model ℐ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
  shows "∀𝒜 ∈ pm.reachable_constraints Pc. ∄ℐ.
          pm.constraint_model ℐ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
proof -
  note 0 = Ps_with_star_projs_def[unfolded Let_def]

  have 1:
      "set Ps_with_star_projs =
          (λn. map (transaction_proj n) Pc) `
            set (remdups (concat (map (λT. map (the_LabelN ∘ fst)
                                          (filter (Not ∘ has_LabelS) (transaction_strand T)))
                                Pc)))"
    by (metis (mono_tags, lifting) 0 image_set)

  have 2: "Ps_with_star_projs ! n = map (transaction_proj n) Pc"
    using conjunct1[OF 0] n_def by fastforce

  show ?thesis
    by (rule pm.composable_protocols_par_comp_prot'[
                OF Pc_def 1 Pc_wellformed_composable
                   components_leakage_free 2 P_welltyped_secure[unfolded P_def]])
qed

end

end
```

4 Trac Support and Automation

4.1 Useful Eisbach Methods for Automating Protocol Verification

```
theory Eisbach_Protocol_Verification
  imports Stateful_Protocol_Composition_and_Typing.Stateful_Compositionality
          "HOL-Eisbach.Eisbach_Tools"
begin

ML‹fun code_simp_all_tac ctx = PARALLEL_ALLGOALS (fn i => Code_Simp.dynamic_tac ctx i)›
method_setup code_simp_all = ‹Scan.succeed (fn ctxt => SIMPLE_METHOD (code_simp_all_tac ctxt))›
                             ‹code_simp (all goals)›

ML‹fun normalization_all_tac ctx = PARALLEL_ALLGOALS (fn i => (CONVERSION(Nbe.dynamic_conv ctx)
                                                                THEN_ALL_NEW (TRY o resolve_tac ctx
[TrueI])) i)›
method_setup normalization_all = ‹Scan.succeed (fn ctxt => SIMPLE_METHOD (normalization_all_tac
ctxt))›
                             ‹normalization (all goals)›

ML‹fun eval_all_tac ctx =
  let
    fun eval_tac ctxt =
      let val conv = Code_Runtime.dynamic_holds_conv
      in
        CONVERSION (Conv.params_conv ~1 (Conv.concl_conv ~1 o conv) ctxt) THEN'
        resolve_tac ctxt [TrueI]
      end
  in
    PARALLEL_ALLGOALS (fn i => eval_tac ctx i)
  end ›
method_setup eval_all = ‹Scan.succeed (fn ctxt => SIMPLE_METHOD (eval_all_tac ctxt))›
                             ‹evaluation (all goals)›

named_theorems exhausts
named_theorems type_class_instance_lemmata
named_theorems protocol_checks
named_theorems protocol_checks'
named_theorems coverage_check_unfold_protocol_lemma
named_theorems coverage_check_unfold_transaction_lemma
named_theorems coverage_check_unfold_lemmata
named_theorems protocol_check_intro_lemmata
named_theorems transaction_coverage_lemmata
named_theorems protocol_def
named_theorems protocol_defs

method UNIV_lemma =
  (rule UNIV_eq_I; (subst insert_iff)+; subst empty_iff; smt exhausts)+

method type_class_instance =
  (intro_classes; auto simp add: type_class_instance_lemmata)

method protocol_model_subgoal =
  (((rule allI, case_tac f); (erule forw_subst)+)?, simp_all)

method protocol_model_interpretation =
  (unfold_locales; protocol_model_subgoal+)
```

```
method composable_protocols_intro =
  (unfold protocol_checks' Let_def;
   intro comp_par_complsstI';
   (simp only: list.map(1,2) prod.sel(1))?;
   (intro list_set_ballI)?;
   (simp only: if_P if_not_P)?)

method coverage_check_intro =
  (((unfold coverage_check_unfold_protocol_lemma)?;
    intro protocol_check_intro_lemmata;
    simp only: list_all_simps list_all_append list.map concat.simps map_append product_concat_map;
    intro conjI TrueI);
   clarsimp?;
   (intro conjI TrueI)?;
   (rule transaction_coverage_lemmata)?)

method coverage_check_unfold =
  (unfold coverage_check_unfold_lemmata
          Let_def case_prod_unfold Product_Type.fst_conv Product_Type.snd_conv;
   simp only: list_all_simps;
   intro conjI TrueI)

method coverage_check_intro' =
  (((unfold coverage_check_unfold_protocol_lemma coverage_check_unfold_transaction_lemma)?;
    intro protocol_check_intro_lemmata;
    simp only: list_all_simps list_all_append list.map concat.simps map_append product_concat_map;
    intro conjI TrueI);
   (clarsimp+)?;
   (intro conjI TrueI)?;
   ((rule transaction_coverage_lemmata)+)?;
   coverage_check_unfold)

method check_protocol_intro =
  (unfold_locales, unfold protocol_checks[symmetric])

method check_protocol_intro' =
  ((check_protocol_intro;
    coverage_check_intro?;
    (unfold protocol_checks' Let_def; intro conjI)?),
  tactic distinct_subgoals_tac)

method check_protocol_with methods meth =
  ((check_protocol_intro; coverage_check_intro?), meth)

method parallel_check_protocol_with methods meth =
  (check_protocol_with ‹((simp_all add: protocol_def protocol_defs Let_def, safe)?), tactic
‹distinct_subgoals_tac›, meth›)

method check_protocol =
  (parallel_check_protocol_with ‹code_simp_all›)

method check_protocol_nbe =
  (parallel_check_protocol_with ‹normalization_all›)

method check_protocol_eval =
  (parallel_check_protocol_with ‹eval_all›)

method check_protocol_compositionality =
  (check_protocol_with ‹code_simp_all›; fastforce?)

method check_protocol_compositionality_nbe =
  (check_protocol_with ‹normalization_all›; fastforce?)
```

method *check_protocol_compositionality_eval* =
  (check_protocol_with ‹(eval_all?, code_simp_all?)›; fastforce?)

end

4.2 ML Yacc Library

theory
  "ml_yacc_lib"
  imports
  Main
begin
ML_file *"ml-yacc-lib/base.sig"*
ML_file *"ml-yacc-lib/join.sml"*
ML_file *"ml-yacc-lib/lrtable.sml"*
ML_file *"ml-yacc-lib/stream.sml"*
ML_file *"ml-yacc-lib/parser2.sml"*

end

4.3 Abstract Syntax for Trac Terms

theory
  trac_term
  imports
    "First_Order_Terms.Term"
    "ml_yacc_lib"

begin
ML‹

```
structure Trac_Utils =
struct
  val valN = "val"
  val timpliesN = "timplies"
  val occursN = "occurs"
  val enumN = "enum"
  val enum_trac_typeN = "enum"
  val value_trac_typeN = "value"
  val priv_fun_secN = "PrivFunSec"
  val secret_typeN = "SecretType"
  val enum_typeN = "EnumType"
  val other_pubconsts_typeN = "PubConstType"

  val default_extra_types = [enum_typeN, secret_typeN]
  val extended_extra_types = default_extra_types@[other_pubconsts_typeN]
  val all_special_types = value_trac_typeN::enum_trac_typeN::extended_extra_types

  val special_funs = ["occurs", "zero", valN, priv_fun_secN]

  fun infenumN e = enumN^"_"^e

  fun list_find p ts =
    let
      fun aux _ [] = NONE
        | aux n (t::ts) =
            if p t
            then SOME (t,n)
            else aux (n+1) ts
    in
```

```
    aux 0 ts
  end

fun map_prod f (a,b) = (f a, f b)

fun list_product [] = [[]]
  | list_product (xs::xss) =
      List.concat (map (fn x => map (fn ys => x::ys) (list_product xss)) xs)

fun list_triangle_product _ [] = []
  | list_triangle_product f (x::xs) = map (f x) xs@list_triangle_product f xs

fun list_subseqs [] = [[]]
  | list_subseqs (x::xs) = let val xss = list_subseqs xs in map (cons x) xss@xss end

fun list_intersect xs ys =
    List.exists (fn x => member (op =) ys x) xs orelse
    List.exists (fn y => member (op =) xs y) ys

fun list_partitions xs constrs =
  let
    val peq = eq_set (op =)
    val pseq = eq_set peq
    val psseq = eq_set pseq

    fun illegal p q =
      let
        val pq = union (op =) p q
        fun f (a,b) = member (op =) pq a andalso member (op =) pq b
      in
        List.exists f constrs
      end

    fun merges _ [] = []
      | merges q (p::ps) =
          if illegal p q then map (cons p) (merges q ps)
          else (union (op =) p q::ps)::(map (cons p) (merges q ps))

    fun merges_all [] = []
      | merges_all (p::ps) = merges p ps@map (cons p) (merges_all ps)

    fun step pss = fold (union pseq) (map merges_all pss) []

    fun loop pss pssprev =
      let val pss' = step pss
      in if psseq (pss,pss') then pssprev else loop pss' (union pseq pss' pssprev)
      end

    val init = [map single xs]
  in
    loop init init
  end

fun list_rm_pair sel l x = filter (fn e => sel e <> x) l

fun list_minus list_rm l m = List.foldl (fn (a,b) => list_rm b a) l m

fun list_upto n =
  let
    fun aux m = if m >= n then [] else m::aux (m+1)
  in
    aux 0
  end
```

```
end
›
```

ML‹
```
structure Trac_Term (* : TRAC_TERM *) =
struct
open Trac_Utils
exception TypeError

type TypeDecl = string * string

datatype MsgType = TAtom of string
                 | TComp of string * MsgType list

type TypedVars = (string list * MsgType) list

datatype Msg = Var of string
             | Const of string
             | Fun of string * Msg list
             | Abbrev of string * Msg list
             | Attack

(* TODO: maybe add a set-type *)
datatype cType = Enumeration of string
               | InfiniteEnumeration of string
               | EnumType
               | ValueType
               | PrivFunSecType
               | AtomicType of string
               | ComposedType of string * cType list
               | Untyped

datatype cMsg = cVar of string * cType
              | cConst of string
              | cFun of string * cMsg list
              | cAttack
              | cSet of string * cMsg list
              | cAbs of (string * string list) list
              | cOccursFact of cMsg
              | cPrivFunSec
              | cEnum of string

fun MsgType_str (TAtom a) = a
  | MsgType_str (TComp (f,ts)) = f ^ "(" ^ String.concatWith "," (map MsgType_str ts) ^  ")"

fun Msg_str (Var x) = x
  | Msg_str (Const x) = x
  | Msg_str (Fun (f,ps)) =
      if ps = [] then f else f ^ "(" ^ String.concatWith "," (map Msg_str ps) ^ ")"
  | Msg_str (Abbrev (f,ps)) =
      if ps = [] then f else f ^ "![" ^ String.concatWith "," (map Msg_str ps) ^ "]"
  | Msg_str Attack = "attack"

fun msg_vars t =
  let fun f (Var x) = [x]
        | f (Fun (_,ps)) = List.concat (map f ps)
        | f (Abbrev (_,ps)) = List.concat (map f ps)
        | f (Const _) = []
        | f Attack = []
  in distinct (op =) (f t)
  end

fun cType_str (Enumeration e)         = e
```

```
  | cType_str (InfiniteEnumeration e) = e
  | cType_str EnumType                = enum_trac_typeN
  | cType_str ValueType               = value_trac_typeN
  | cType_str PrivFunSecType          = secret_typeN
  | cType_str (AtomicType a)          = a
  | cType_str (ComposedType (f,ts))   = f ^ "(" ^ String.concatWith "," (map cType_str ts) ^ ")"
  | cType_str Untyped                 = "untyped"

fun cMsg_str' notypes (cVar (x,tau)) = x ^ (if notypes then "" else ":" ^ cType_str tau)
  | cMsg_str' _       (cConst s) = s
  | cMsg_str' notypes (cFun (f,ts)) =
      f ^ "(" ^ String.concatWith "," (map (cMsg_str' notypes) ts) ^ ")"
  | cMsg_str' _       cAttack = "attack"
  | cMsg_str' notypes (cSet (s,ts)) =
      s ^ "(" ^ String.concatWith "," (map (cMsg_str' notypes) ts) ^ ")"
  | cMsg_str' _       (cAbs bs) =
      valN ^ "(" ^ String.concatWith ","
        (map (fn (c,cs) => c ^ "(" ^ String.concatWith "," cs  ^ ")") bs) ^
      ")"
  | cMsg_str' notypes (cOccursFact t) = occursN ^ "(" ^ cMsg_str' notypes t ^ ")"
  | cMsg_str' _       cPrivFunSec = priv_fun_secN
  | cMsg_str' _       (cEnum e) = e

val cMsg_str = cMsg_str' false

fun subst_apply_cMsg' (delta:(string * cType) -> cMsg) (t:cMsg) =
  case t of
    cVar x => delta x
  | cFun (f,ts) => cFun (f, map (subst_apply_cMsg' delta) ts)
  | cSet (s,ts) => cSet (s, map (subst_apply_cMsg' delta) ts)
  | cOccursFact bs => cOccursFact (subst_apply_cMsg' delta bs)
  | c => c

fun subst_apply_cMsg (delta:(string * cMsg) list) =
  subst_apply_cMsg' (fn (n,tau) => (
      case List.find (fn x => fst x = n) delta of
        SOME x => snd x
      | NONE => cVar (n,tau)))

fun subst_apply_Msg d (Var x) = (
      case List.find (fn (y,_) => x = y) d of
        SOME (_,t) => t
      | NONE => error ("Error: Cannot find variable " ^ x))
  | subst_apply_Msg _ (Const c) = Const c
  | subst_apply_Msg d (Fun (f,ts')) = Fun (f,map (subst_apply_Msg d) ts')
  | subst_apply_Msg d (Abbrev (f,ts')) = Abbrev (f,map (subst_apply_Msg d) ts')
  | subst_apply_Msg _ Attack = Attack

fun certifyMsgType' finite_enums infinite_enums (TAtom a) =
      if a = enum_trac_typeN then EnumType
      else if a = value_trac_typeN then ValueType
      else if List.exists (fn b => a = b) finite_enums then Enumeration a
      else if List.exists (fn b => a = b) infinite_enums then InfiniteEnumeration a
      else AtomicType a
  | certifyMsgType' finite_enums infinite_enums (TComp (f,ts)) =
      ComposedType (f,map (certifyMsgType' finite_enums infinite_enums) ts)

fun certifyMsgType ((finite_enums:string list)
                   ,(infinite_enums:string list)
                   ,(decls:TypedVars)
                   ,(fresh:TypedVars)) n =
  case List.find (fn (vs,_) => member (op =) vs n) decls of
    SOME (_,tau) => certifyMsgType' finite_enums infinite_enums tau
```

```
  | NONE => (
      case List.find (fn (vs,_) => member (op =) vs n) fresh of
        SOME (_,tau) => certifyMsgType' finite_enums infinite_enums tau
      | NONE => error ("Error: Missing or invalid type annotation for variable " ^ n))

fun certifyMsg' notypes params (Var n)         =
      if notypes then cVar (n, Untyped) else cVar (n, certifyMsgType params n)
  | certifyMsg' _       _      (Const c)       =
      cConst c
  | certifyMsg' notypes params (Fun (f,ts))    =
      cFun (f, map (certifyMsg' notypes params) ts)
  | certifyMsg' _       _      (Abbrev p) =
      error ("Error: Got an unexpected term abbreviation (they should have all been expanded " ^
             "and removed at this point): " ^ Msg_str (Abbrev p))
  | certifyMsg' _       _      Attack          =
      cAttack

val certifyMsg = certifyMsg' false
val certifyMsgUntyped = certifyMsg' true ([], [], [], [])

fun mk_Value_cVar x = cVar (x,ValueType)

val fv_Msg =
  let
    fun aux (Var x) = [x]
      | aux (Fun (_,ts)) = List.concat (map aux ts)
      | aux _ = []
  in
    distinct (op =) o aux
  end

val fv_cMsg =
  let
    fun aux (cVar x) = [x]
      | aux (cFun (_,ts)) = List.concat (map aux ts)
      | aux (cSet (_,ts)) = List.concat (map aux ts)
      | aux (cOccursFact bs) = aux bs
      | aux _ = []
  in
    distinct (op =) o aux
  end
end
›
```

ML‹

```
structure TracProtocol (* : TRAC_TERM *) =
struct
open Trac_Utils Trac_Term

datatype enum_spec_elem =
  Consts of string list
| Union of string list
| InfiniteSet

fun is_Consts t = case t of Consts _ => true | _ => false
fun the_Consts t = case t of Consts cs => cs | _ => error "Consts"

type type_spec = string list
type enum_spec = (string * enum_spec_elem) list
type set_spec_elem  = (string * int * bool)
type set_spec = set_spec_elem list

fun extract_Consts (spec:enum_spec) =
```

```
  (List.concat o map the_Consts o filter is_Consts o map snd) spec

type funT = (string * int * MsgType option)
type fun_spec = {private: funT list, public: funT list}

type ruleT = (string * string list) * Msg list * string list
type anaT = ruleT list

datatype prot_label = LabelN | LabelS

type Bvars = TypedVars

datatype Negcheck = INEQ of Msg * Msg
                  | NOTIN of Msg * (string * Msg list)

datatype action = RECEIVE of Msg list
                | SEND of Msg list
                | EQUATION of Msg * Msg
                | LETBINDING of Msg * Msg
                | IN of Msg * (string * Msg list)
                | NOTINANY of Msg * string
                | NEGCHECKS of Bvars * Negcheck list
                | INSERT of Msg * (string * Msg list)
                | DELETE of Msg * (string * Msg list)
                | NEW of TypedVars
                | ATTACK

datatype labeled_action =
  LABELED_ACTION of prot_label * action
| ABBREVIATION of string * Msg list

type transaction_name = string * (string list * MsgType) list * (string * string) list

type transaction={transaction:transaction_name,actions:labeled_action list}

fun typedvars_str xss =
  let fun f (xs,tau) = String.concatWith "," xs ^ ": " ^ MsgType_str tau
  in String.concatWith ", " (map f xss) end

val action_str =
  let
    fun set_action_str (t,(s,ps)) pre mid =
      pre ^ Msg_str t ^ mid ^ s ^ (
      if ps = [] then "" else "(" ^ String.concatWith "," (map Msg_str ps) ^  ")")
    fun negcheck_str (INEQ (t,t')) = Msg_str t ^ " != " ^ Msg_str t'
      | negcheck_str (NOTIN p)    = set_action_str p "" " notin "
    fun to_str (SEND ts)             = "send " ^ String.concatWith ", " (map Msg_str ts)
      | to_str (RECEIVE ts)          = "receive " ^ String.concatWith ", " (map Msg_str ts)
      | to_str (LETBINDING (t,t'))   = "let " ^ Msg_str t ^ " = " ^ Msg_str t'
      | to_str (EQUATION (t,t'))     = Msg_str t ^ " == " ^ Msg_str t'
      | to_str (IN p)                = set_action_str p "" " in "
      | to_str (NOTINANY (t,s))      = set_action_str (t,(s,[])) "" " notin " ^ "(_)"
      | to_str (NEGCHECKS (bvars,ns)) = String.concatWith " or " (map negcheck_str ns) ^
                                        (if null bvars then "" else " forall ") ^
                                        typedvars_str bvars
      | to_str (INSERT p)            = set_action_str p "insert " " "
      | to_str (DELETE p)            = set_action_str p "delete " " "
      | to_str (NEW xs)              = "new " ^ typedvars_str xs
      | to_str ATTACK                = "attack"
  in
    to_str
  end
```

```
fun labeled_action_str (LABELED_ACTION (lbl,act)) =
      (case lbl of LabelN => "  " | LabelS => "* ") ^ action_str act
  | labeled_action_str (ABBREVIATION (f,ts)) =
      f ^ "![" ^ String.concatWith "," (map Msg_str ts) ^ "]"

fun typedvars_flatten xss = List.concat (map (fn (xs,tau) => map (fn x => (x,tau)) xs) xss)
fun typedvars_fvs xss = map fst (typedvars_flatten xss)

fun action_fvs (RECEIVE ts)        = distinct (op =) (List.concat (map msg_vars ts))
  | action_fvs (LETBINDING (t,t')) = distinct (op =) (msg_vars t@msg_vars t')
  | action_fvs (EQUATION (t,t'))   = distinct (op =) (msg_vars t@msg_vars t')
  | action_fvs (IN (t,(_,p)))      = distinct (op =) (msg_vars t@List.concat (map msg_vars p))
  | action_fvs (NOTINANY (t,_))    = msg_vars t
  | action_fvs (NEGCHECKS (bvars,ns))  =
      let
        fun f (INEQ (t,t')) = msg_vars t@msg_vars t'
          | f (NOTIN (t,(_,p))) = msg_vars t@List.concat (map msg_vars p)
      in
        filter_out (member (op =) (typedvars_fvs bvars)) (distinct (op =) (List.concat (map f ns)))
      end
  | action_fvs (NEW xs)            = distinct (op =) (typedvars_fvs xs)
  | action_fvs (INSERT (t,(_,p)))  = distinct (op =) (msg_vars t@List.concat (map msg_vars p))
  | action_fvs (DELETE (t,(_,p)))  = distinct (op =) (msg_vars t@List.concat (map msg_vars p))
  | action_fvs (SEND ts)           = distinct (op =) (List.concat (map msg_vars ts))
  | action_fvs ATTACK              = []

fun mkTransaction transaction actions = {transaction=transaction,
                                         actions=actions}:transaction

fun is_RECEIVE a = case a of RECEIVE _ => true | _ => false
fun is_SEND a = case a of SEND _ => true | _ => false
fun is_LETBINDING a = case a of LETBINDING _ => true | _ => false
fun is_EQUATION a = case a of EQUATION _ => true | _ => false
fun is_IN a = case a of IN _ => true | _ => false
fun is_NEGCHECKS a = case a of NEGCHECKS _ => true | _ => false
fun is_NOTINANY a = case a of NOTINANY _ => true | _ => false
fun is_INSERT a = case a of INSERT _ => true | _ => false
fun is_DELETE a = case a of DELETE _ => true | _ => false
fun is_NEW a = case a of NEW _ => true | _ => false
fun is_ATTACK a = case a of ATTACK => true | _ => false

fun the_RECEIVE a = case a of RECEIVE t => t | _ => error "RECEIVE"
fun the_SEND a = case a of SEND t => t | _ => error "SEND"
fun the_LETBINDING a = case a of LETBINDING t => t | _ => error "LETBINDING"
fun the_EQUATION a = case a of EQUATION t => t | _ => error "EQUATION"
fun the_IN a = case a of IN t => t | _ => error "IN"
fun the_NEGCHECKS a = case a of NEGCHECKS t => t | _ => error "NEGCHECKS"
fun the_NOTINANY a = case a of NOTINANY t => t | _ => error "NOTINANY"
fun the_INSERT a = case a of INSERT t => t | _ => error "INSERT"
fun the_DELETE a = case a of DELETE t => t | _ => error "DELETE"
fun the_NEW a = case a of NEW t => t | _ => error "FRESH"

fun maybe_the_RECEIVE a = case a of RECEIVE t => SOME t | _ => NONE
fun maybe_the_SEND a = case a of SEND t => SOME t | _ => NONE
fun maybe_the_LETBINDING a = case a of LETBINDING t => SOME t | _ => NONE
fun maybe_the_EQUATION a = case a of EQUATION t => SOME t | _ => NONE
fun maybe_the_IN a = case a of IN t => SOME t | _ => NONE
fun maybe_the_NEGCHECKS a = case a of NEGCHECKS t => SOME t | _ => NONE
fun maybe_the_NOTINANY a = case a of NOTINANY t => SOME t | _ => NONE
fun maybe_the_INSERT a = case a of INSERT t => SOME t | _ => NONE
fun maybe_the_DELETE a = case a of DELETE t => SOME t | _ => NONE
fun maybe_the_NEW a = case a of NEW t => SOME t | _ => NONE
```

```
fun subst_apply_labeled_action d (LABELED_ACTION (lbl,a)) =
      let
        val ap = subst_apply_Msg d
        fun rm_vars_ap ys =
          let val zs = typedvars_fvs ys
          in subst_apply_Msg (filter (fn (x,_) => not (member (op =) zs x)) d) end
        fun ap_negcheck xs (INEQ (t,t')) =
              INEQ (rm_vars_ap xs t, rm_vars_ap xs t')
          | ap_negcheck xs (NOTIN (t,(f,ts))) =
              NOTIN (rm_vars_ap xs t,(f,map (rm_vars_ap xs) ts))
        fun aux (RECEIVE ts)        = RECEIVE (map ap ts)
          | aux (SEND ts)           = SEND (map ap ts)
          | aux (EQUATION (t,t'))   = EQUATION (ap t, ap t')
          | aux (LETBINDING (t,t')) = LETBINDING (ap t, ap t')
          | aux (IN (t,(f,ts)))     = IN (ap t,(f,map ap ts))
          | aux (NOTINANY (t,f))    = NOTINANY (ap t, f)
          | aux (NEGCHECKS (xs,ns)) = NEGCHECKS (xs,map (ap_negcheck xs) ns)
          | aux (INSERT (t,(f,ts))) = INSERT (ap t,(f,map ap ts))
          | aux (DELETE (t,(f,ts))) = DELETE (ap t,(f,map ap ts))
          | aux (NEW p)             = NEW p
          | aux ATTACK              = ATTACK
      in
        LABELED_ACTION (lbl,aux a)
      end
  | subst_apply_labeled_action d (ABBREVIATION (f,ts')) =
      ABBREVIATION (f,map (subst_apply_Msg d) ts')

fun expand_term_abbreviations _       (Var x) = Var x
  | expand_term_abbreviations _       (Const c) = Const c
  | expand_term_abbreviations abbrevs (Fun (f,ts)) =
      Fun (f,map (expand_term_abbreviations abbrevs) ts)
  | expand_term_abbreviations abbrevs (Abbrev (f,ts)) = (
      case List.find (fn ((g,_),_) => f = g) abbrevs of
        SOME ((_,xs),t) =>
          if length xs <> length ts
          then error ("Error: The number of parameters given to the term abbreviation " ^
                      Msg_str (Abbrev(f,ts)) ^ " does not match the number of parameters " ^
                      "in its declaration")
          else
            let val delta = xs ~~ ts
            in expand_term_abbreviations abbrevs (subst_apply_Msg delta t)
            end
      | NONE => error ("Error: Cannot find term abbreviation " ^ f))
  | expand_term_abbreviations _       Attack = Attack

fun expand_term_abbreviations_in_action abbrevs ac =
  let
    val exp = expand_term_abbreviations abbrevs
    fun exp_n (INEQ (t,t')) = INEQ (exp t,exp t')
      | exp_n (NOTIN (t,(s,ts))) = NOTIN (exp t,(s,map exp ts))
  in case ac of
        RECEIVE ts => RECEIVE (map exp ts)
      | SEND ts => SEND (map exp ts)
      | EQUATION (t,t') => EQUATION (exp t, exp t')
      | LETBINDING (t,t') => LETBINDING (exp t, exp t')
      | IN (t,(s,ts)) => IN (exp t,(s,map exp ts))
      | NOTINANY (t,s) => NOTINANY (exp t,s)
      | NEGCHECKS (xs,ns) => NEGCHECKS (xs,map exp_n ns)
      | INSERT (t,(s,ts)) => INSERT (exp t,(s,map exp ts))
      | DELETE (t,(s,ts)) => DELETE (exp t,(s,map exp ts))
      | NEW xs => NEW xs
      | ATTACK => ATTACK
  end
```

```
fun expand_action_abbreviations (abbrevs:((string * string list) * labeled_action list) list) =
  let
    fun get abbr = case List.find (fn ((a,_),_) => abbr = a) abbrevs of
        SOME ((_,xs),acs) => (xs,acs)
      | NONE => error ("Error: Action sequence abbreviation " ^ abbr ^ " has not been declared")

    fun expand (abbr,ts) =
      let
        val (xs,acs) = get abbr

        val _ = if length xs <> length ts
                then error ("Error: Action sequence abbreviation " ^ abbr ^ " has been applied " ^
                            "with the wrong number of parameters: Expected " ^
                            Int.toString (length xs) ^ " parameters but got " ^
                            Int.toString (length ts))
                else ()

        val delta = xs ~~ ts
      in expand_action_abbreviations abbrevs (map (subst_apply_labeled_action delta) acs) end
  in
    List.concat o
    map (fn a => case a of LABELED_ACTION p => [p]
                         | ABBREVIATION p => expand p)
  end


datatype abbreviation =
  TermAbbreviation of (string * string list) * Msg
| ActionsAbbreviation of (string * string list) * labeled_action list

type abbreviation_spec = abbreviation list

type protocol = {
  name:string
 ,type_spec:type_spec
 ,enum_spec:enum_spec
 ,set_spec:set_spec
 ,function_spec:fun_spec option
 ,analysis_spec:anaT
 ,abbreviation_spec:abbreviation_spec
 ,transaction_spec:(string option * transaction list) list
 ,fixed_point: (Msg * (string * string) list) list option
}

exception TypeError

val fun_empty = {
                  public=[]
                 ,private=[]
                }:fun_spec

fun update_fun_public (fun_spec:fun_spec) public =
    ({public = public
     ,private = #private fun_spec
    }):fun_spec

fun update_fun_private (fun_spec:fun_spec) private =
    ({public = #public fun_spec
     ,private = private
    }):fun_spec
```

```
val empty={
           name=""
          ,type_spec=[]
          ,enum_spec=[]
          ,set_spec=[]
          ,function_spec=NONE
          ,analysis_spec=[]
          ,abbreviation_spec=[]
          ,transaction_spec=[]
          ,fixed_point=NONE
         }:protocol

fun update_name (protocol_spec:protocol) name =
    ({name = name
     ,type_spec = #type_spec protocol_spec
     ,enum_spec = #enum_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,abbreviation_spec = #abbreviation_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_sets (protocol_spec:protocol) set_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,enum_spec = #enum_spec protocol_spec
     ,set_spec = set_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,abbreviation_spec = #abbreviation_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_type_spec (protocol_spec:protocol) type_spec =
    ({name = #name protocol_spec
     ,type_spec = type_spec
     ,enum_spec = #enum_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,abbreviation_spec = #abbreviation_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_enum_spec (protocol_spec:protocol) enum_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,enum_spec = enum_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,abbreviation_spec = #abbreviation_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_functions (protocol_spec:protocol) function_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,enum_spec = #enum_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = SOME function_spec
     ,analysis_spec = #analysis_spec protocol_spec
```

```
     ,abbreviation_spec = #abbreviation_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_analysis (protocol_spec:protocol) analysis_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,enum_spec = #enum_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = analysis_spec
     ,abbreviation_spec = #abbreviation_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_abbreviations (protocol_spec:protocol) abbreviation_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,enum_spec = #enum_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,abbreviation_spec = abbreviation_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_transactions (prot_name:string option) (protocol_spec:protocol) transaction_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,enum_spec = #enum_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,abbreviation_spec = #abbreviation_spec protocol_spec
     ,transaction_spec = (prot_name,transaction_spec)::(#transaction_spec protocol_spec)
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_fixed_point (protocol_spec:protocol) fixed_point =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,enum_spec = #enum_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,abbreviation_spec = #abbreviation_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = fixed_point
    }):protocol

end
›
```

ML‹

```
structure TracProtocolCert (* : TRAC_TERM *) =
struct
open Trac_Utils Trac_Term TracProtocol

type cBvars = (string * cType) list

datatype cNegCheckVariant = cInequality of cMsg * cMsg
                          | cNotInSet of cMsg * cMsg
```

```
datatype cPosCheckVariant = cCheck
                          | cAssignment

datatype cAction = cReceive of cMsg list
                 | cSend of cMsg list
                 | cEquality of cPosCheckVariant * (cMsg * cMsg)
                 | cInSet of cPosCheckVariant * (cMsg * cMsg)
                 | cNotInAny of cMsg * string
                 | cNegChecks of cBvars * cNegCheckVariant list
                 | cInsert of cMsg * cMsg
                 | cDelete of cMsg * cMsg
                 | cNew of (string * cType) list
                 | cAssertAttack

type flat_enum_spec = (string * string list * string list) list
type cFunT = (string * int)
type cConstsT = (string * string option)
type cFunSpec = {public_funs: cFunT list, public_consts: cConstsT list,
                 private_consts: cConstsT list}
type cAnaRule = {head: (string * string list), keys: cMsg list,
                 results: string list, is_priv_fun: bool}
type cAnaSpec = cAnaRule list

type cTransaction_name = string * (string * cType) list * (string * string) list

type cTransaction={
  transaction:cTransaction_name
 ,receive_actions:(prot_label * cAction) list
 ,checksingle_actions:(prot_label * cAction) list
 ,checkall_actions:(prot_label * cAction) list
 ,fresh_actions:(prot_label * cAction) list
 ,update_actions:(prot_label * cAction) list
 ,send_actions:(prot_label * cAction) list
 ,attack_actions:(prot_label * cAction) list}

type cProtocol = {
  name:string
 ,type_spec:type_spec
 ,enum_spec:flat_enum_spec
 ,set_spec:set_spec
 ,function_spec:cFunSpec option
 ,analysis_spec:cAnaSpec
 ,transaction_spec:(string option * cTransaction list) list
 ,fixed_point: (cMsg list * (string * string list) list list *
                ((string * string list) list * (string * string list) list) list) option
}

fun is_Receive a = case a of cReceive _ => true | _ => false
fun is_Send a = case a of cSend _ => true | _ => false
fun is_Equality a = case a of cEquality _ => true | _ => false
fun is_InSet a = case a of cInSet _ => true | _ => false
fun is_NegChecks a = case a of cNegChecks _ => true | _ => false
fun is_NotInAny a = case a of cNotInAny _ => true | _ => false
fun is_Insert a = case a of cInsert _ => true | _ => false
fun is_Delete a = case a of cDelete _ => true | _ => false
fun is_Fresh a = case a of cNew _ => true | _ => false
fun is_Attack a = case a of cAssertAttack => true | _ => false
fun is_Inequality a = case a of cInequality _ => true | _ => false
fun is_NotInSet a = case a of cNotInSet _ => true | _ => false

fun the_Receive a = case a of cReceive t => t | _ => error "Receive"
fun the_Send a = case a of cSend t => t | _ => error "Send"
fun the_Equality a = case a of cEquality t => t | _ => error "Equality"
```

```
fun the_InSet a = case a of cInSet t => t | _ => error "InSet"
fun the_NegChecks a = case a of cNegChecks t => t | _ => error "NegChecks"
fun the_NotInAny a = case a of cNotInAny t => t | _ => error "NotInAny"
fun the_Insert a = case a of cInsert t => t | _ => error "Insert"
fun the_Delete a = case a of cDelete t => t | _ => error "Delete"
fun the_Fresh a = case a of cNew ts => ts | _ => error "New"
fun the_Inequality a = case a of cInequality p => p | _ => error "Inequality"
fun the_NotInSet a = case a of cNotInSet p => p | _ => error "NotInSet"

fun maybe_the_Receive a = case a of cReceive t => SOME t | _ => NONE
fun maybe_the_Send a = case a of cSend t => SOME t | _ => NONE
fun maybe_the_Equality a = case a of cEquality (_,t) => SOME t | _ => NONE
fun maybe_the_InSet a = case a of cInSet (_,t) => SOME t | _ => NONE
fun maybe_the_NegChecks a = case a of cNegChecks t => SOME t | _ => NONE
fun maybe_the_NotInAny a = case a of cNotInAny t => SOME t | _ => NONE
fun maybe_the_Insert a = case a of cInsert t => SOME t | _ => NONE
fun maybe_the_Delete a = case a of cDelete t => SOME t | _ => NONE
fun maybe_the_Fresh a = case a of cNew ts => SOME ts | _ => NONE
fun maybe_the_Inequality a = case a of cInequality p => SOME p | _ => NONE
fun maybe_the_NotInSet a = case a of cNotInSet p => SOME p | _ => NONE

fun subst_apply_cAction (delta:(string * cMsg) list) (lbl:prot_label,a:cAction) =
  let
    val ap = subst_apply_cMsg
    val apply = ap delta
    fun rm_vars_apply ys = ap (filter (fn (x,_) => List.all (fn (y,_) => x <> y) ys) delta)
    fun rm_vars_apply_pair xs (t,t') = (rm_vars_apply xs t, rm_vars_apply xs t')
    fun apply_negcheck xs (cInequality p) = cInequality (rm_vars_apply_pair xs p)
      | apply_negcheck xs (cNotInSet p) = cNotInSet (rm_vars_apply_pair xs p)
  in
    case a of
      cReceive ts => (lbl,cReceive (map apply ts))
    | cSend ts => (lbl,cSend (map apply ts))
    | cEquality (v,(t,t')) => (lbl,cEquality (v,(apply t, apply t')))
    | cInSet (v,(x,s)) => (lbl,cInSet (v,(apply x, apply s)))
    | cNotInAny (x,s) => (lbl,cNotInAny (apply x, s))
    | cNegChecks (bvars,ps) => (lbl,cNegChecks (bvars,map (apply_negcheck bvars) ps))
    | cInsert (x,s) => (lbl,cInsert (apply x, apply s))
    | cDelete (x,s) => (lbl,cDelete (apply x, apply s))
    | cNew xs => (lbl,cNew xs)
    | cAssertAttack => (lbl,cAssertAttack)
  end

fun subst_apply_cActions delta =
  map (subst_apply_cAction delta)

val cAction_str =
  let
    val cmsg_str = cMsg_str' true
    fun var_str (x,tau) = x ^ ": " ^ cType_str tau
    fun set_action_str (t,s) pre mid = pre ^ cmsg_str t ^ mid ^ cmsg_str s
    fun negcheck_str (cInequality (t,t')) = cmsg_str t ^ " != " ^ cmsg_str t'
      | negcheck_str (cNotInSet p)        = set_action_str p "" " notin "
    fun to_str (cSend ts)               = "send " ^ String.concatWith ", " (map cmsg_str ts)
      | to_str (cReceive ts)            = "receive " ^ String.concatWith ", " (map cmsg_str ts)
      | to_str (cEquality (psv,(t,t'))) = (
          case psv of
            cCheck => cmsg_str t ^ " == " ^ cmsg_str t'
          | cAssignment => "let " ^ cmsg_str t ^ " = " ^ cmsg_str t')
      | to_str (cInSet (psv,p))    = (
          case psv of
            cCheck => set_action_str p "" " in "
          | cAssignment => set_action_str p "select " " ")
```

```
      | to_str (cNotInAny (t,s))        = cmsg_str t ^ " notin " ^ s ^ "(_)"
      | to_str (cNegChecks (bvars,ns))  = String.concatWith " or " (map negcheck_str ns) ^
                                          (if null bvars then "" else " forall ") ^
                                          String.concatWith ", " (map var_str bvars)
      | to_str (cInsert p)              = set_action_str p "insert " " "
      | to_str (cDelete p)              = set_action_str p "delete " " "
      | to_str (cNew xs)                = "new " ^ String.concatWith ", " (map var_str xs)
      | to_str cAssertAttack            = "attack"
  in
    to_str
  end

fun cTransaction_str (tr:cTransaction) =
  let
    fun lbl_act_str (lbl,act) = (case lbl of LabelN => "  " | LabelS => "* ") ^ cAction_str act
    fun name_str (name, decls, ineqs) =
      name ^ "(" ^ String.concatWith ", " (map (fn (x,t) => x ^ ": " ^ cType_str t) decls) ^ ")" ^
      (if null ineqs then "" else " where ") ^
      String.concatWith ", " (map (fn (a,b) => a ^ " != " ^ b) ineqs)
  in
    name_str (#transaction tr) ^ "\n" ^
    String.concatWith "\n" (
      map lbl_act_str
        ((#receive_actions tr)
        @(#checksingle_actions tr)
        @(#checkall_actions tr)
        @(#fresh_actions tr)
        @(#update_actions tr)
        @(#send_actions tr)
        @(#attack_actions tr))) ^
    "."
  end

fun cMsg_vars t =
  let fun f (cVar (x,_)) = [x]
        | f (cConst _) = []
        | f (cFun (_,ps)) = List.concat (map f ps)
        | f cAttack = []
        | f (cSet (_,ps)) = List.concat (map f ps)
        | f (cAbs _) = []
        | f (cOccursFact t) = f t
        | f cPrivFunSec = []
        | f (cEnum _) = []
  in distinct (op =) (f t)
  end

fun cAction_fvs (cReceive ts)           = distinct (op =) (List.concat (map cMsg_vars ts))
  | cAction_fvs (cEquality (_,(t,t')))  = distinct (op =) (cMsg_vars t@cMsg_vars t')
  | cAction_fvs (cInSet (_,(t,t')))     = distinct (op =) (cMsg_vars t@cMsg_vars t')
  | cAction_fvs (cNotInAny (t,_))       = cMsg_vars t
  | cAction_fvs (cNegChecks (bvars,ns)) =
      let
        fun f (cInequality (t,t')) = cMsg_vars t@cMsg_vars t'
          | f (cNotInSet (t,t'))   = cMsg_vars t@cMsg_vars t'
      in
        filter_out (member (op =) (map fst bvars)) (distinct (op =) (List.concat (map f ns)))
      end
  | cAction_fvs (cNew xs)               = distinct (op =) (map fst xs)
  | cAction_fvs (cInsert (t,t'))        = distinct (op =) (cMsg_vars t@cMsg_vars t')
  | cAction_fvs (cDelete (t,t'))        = distinct (op =) (cMsg_vars t@cMsg_vars t')
  | cAction_fvs (cSend ts)              = distinct (op =) (List.concat (map cMsg_vars ts))
  | cAction_fvs cAssertAttack           = []
```

```
fun is_priv_fun_trac (trac:TracProtocol.protocol) f =
  let val funs = #private (Option.valOf (#function_spec trac))
  in List.exists (fn (g,n,_) => f = g andalso n <> 0) funs end

fun get_enum_consts_trac (trac:TracProtocol.protocol) =
  distinct (op =) (TracProtocol.extract_Consts (#enum_spec trac))

fun flatten_enum_spec_trac (trac:TracProtocol.protocol) =
  let
    open TracProtocol
    fun step taus (s,e) =
      case e of
        Union es =>
          let
            fun f e = case List.find (fn (a,_) => e = a) taus of
                        SOME (_,Union es') =>
                          let
                            val _ = if List.exists (fn a => e = a) es'
                                    then error ("Error: There is a cyclic dependency for " ^
                                                "enumeration " ^ e)
                                    else ()
                          in es' end
                      | SOME _ => [e]
                      | NONE => error ("Error: Enumeration " ^ e ^ " has not been declared")
          in
            (s,Union (distinct (op =) (List.concat (map f es))))
          end
      | c => (s,c)
    fun loop taus =
      let
        val taus' = map (step taus) taus
      in
        if taus = taus'
        then taus
        else loop taus'
      end
    fun postproc _    (e,InfiniteSet) = (e,[],[e])
      | postproc _    (e,Consts cs)   = (e,distinct (op =) cs,[])
      | postproc spec (e,Union es)    =
          let
            fun get e' = case List.find (fn (x,_) => x = e') spec of
                SOME p => p
              | NONE => error ("Error: Enumeration " ^ e ^ " has not been declared")
            fun ins (_,Consts cs) (fes,ies) = (distinct (op =) (fes@cs),ies)
              | ins (e',InfiniteSet) (fes,ies) = (fes,distinct (op =) (ies@[e']))
              | ins _ _ = error "Error: Couldn't flatten the enumerations"
            val (fes,ies) = fold (ins o get) es ([],[])
          in (e,fes,ies) end
    val flat_enum_spec = loop (#enum_spec trac)
  in
    map (postproc flat_enum_spec) flat_enum_spec
  end

fun flatten_finite_enum_spec_trac (trac:TracProtocol.protocol) =
  map_filter (fn (e,cs,es) => if null es then SOME (e,cs) else NONE) (flatten_enum_spec_trac trac)

fun priv_fun_type_enc trac (Trac_Term.ComposedType (f,ts)) =
      if is_priv_fun_trac trac f andalso
         (case ts of Trac_Term.PrivFunSecType::_ => false | _ => true)
      then Trac_Term.ComposedType (f,Trac_Term.PrivFunSecType::map (priv_fun_type_enc trac) ts)
      else Trac_Term.ComposedType (f,map (priv_fun_type_enc trac) ts)
  | priv_fun_type_enc _ tau = tau
```

```
fun priv_fun_enc trac t =
  let
    open Trac_Term

    fun aux constr f ts =
      if is_priv_fun_trac trac f andalso
         (case ts of cPrivFunSec::_ => false | _ => true)
      then constr (f,cPrivFunSec::map (priv_fun_enc trac) ts)
      else constr (f,map (priv_fun_enc trac) ts)
  in
    case t of
      cVar (x,tau) => cVar (x, priv_fun_type_enc trac tau)
    | cFun (f,ts) => aux cFun f ts
    | cSet (s,ts) => aux cSet s ts
    | _ => t
  end

fun transform_cMsg trac =
  let
    open Trac_Term

    fun conv_enum_consts trac (t:cMsg) =
      let
        val enums = get_enum_consts_trac trac
        fun aux (cFun (f,ts)) =
              if List.exists (fn x => x = f) enums
              then if null ts
                   then cEnum f
                   else error ("Error: Enumeration constant " ^ f ^
                               " should not have a parameter list")
              else
                cFun (f,map aux ts)
          | aux (cConst c) =
              if List.exists (fn x => x = c) enums
              then cEnum c
              else cConst c
          | aux (cSet (s,ts)) = cSet (s,map aux ts)
          | aux (cOccursFact bs) = cOccursFact (aux bs)
          | aux t = t
      in
        aux t
      end

    fun val_to_abs (t:cMsg) =
      let
        fun aux t = case t of cEnum b => b | _ => error "Error: Invalid val parameter list"

        fun val_to_abs_list [] = []
          | val_to_abs_list (cConst "0"::ts) = val_to_abs_list ts
          | val_to_abs_list (cFun (s,ps)::ts) = (s, map aux ps)::val_to_abs_list ts
          | val_to_abs_list (cSet (s,ps)::ts) = (s, map aux ps)::val_to_abs_list ts
          | val_to_abs_list ts = error ("Error: Invalid val parameter list: [" ^
                                 String.concatWith ", " (map cMsg_str ts) ^ "]")
      in
        case t of
          cFun (f,ts) =>
            if f = valN
            then cAbs (val_to_abs_list ts)
            else cFun (f,map val_to_abs ts)
        | cSet (s,ts) =>
            cSet (s,map val_to_abs ts)
        | cOccursFact bs =>
            cOccursFact (val_to_abs bs)
```

```
        | t => t
      end

    fun occurs_enc t =
      let
        fun aux [cVar x] = cVar x
          | aux [cAbs bs] = cAbs bs
          | aux ts = error ("Error: Invalid occurs parameter list: [" ^
                            String.concatWith ", " (map cMsg_str ts) ^ "]")
        fun enc (cFun (f,ts)) = (
              if f = occursN
              then cOccursFact (aux ts)
              else cFun (f,map enc ts))
          | enc (cSet (s,ts)) =
              cSet (s,map enc ts)
          | enc (cOccursFact bs) =
              cOccursFact (enc bs)
          | enc t = t
      in
        enc t
      end
  in
    occurs_enc o val_to_abs o conv_enum_consts trac o priv_fun_enc trac
  end

fun certify_fixpoint trac fp =
  let
    open Trac_Term

    fun mk_enum_substs (vars:(string * cType) list) =
      let
        val flat_enum_spec = flatten_finite_enum_spec_trac trac
        val deltas =
          let
            fun f (s,Enumeration tau) = (
                case List.find (fn x => fst x = tau) flat_enum_spec of
                  SOME x => map (fn c => (s,c)) (snd x)
                | NONE => error ("Error: Enumeration " ^ tau ^
                                 " was not found in the finite enumeration specification"))
              | f (s,_) = error ("Error: Variable " ^ s ^ " is not of finite enumeration type")
          in
            list_product (map f vars)
          end
      in
        map (fn d => map (fn (x,t) => (x,cEnum t)) d) deltas
      end

    fun ground_enum_variables (fp:cMsg list) =
      let
        fun do_grounding t = map (fn d => subst_apply_cMsg d t) (mk_enum_substs (fv_cMsg t))
      in
        List.concat (map do_grounding fp)
      end

    fun split_fp (fp:cMsg list) =
      let
        fun fa t = case t of cFun (s,_) => s <> timpliesN | _ => true
        fun fb (t,ts) = case t of cOccursFact (cAbs bs) => bs::ts | _ => ts
        fun fc (cFun (s, [cAbs bs, cAbs cs]),ts) =
            if s = timpliesN
            then (bs,cs)::ts
            else ts
          | fc (_,ts) = ts
```

```
        val eq = eq_set (op =)
        fun eq_pairs ((a,b),(c,d)) = eq (a,c) andalso eq (b,d)

        val timplies_trancl =
          let
            fun trans_step ts =
              let
                fun aux (s,t) = map (fn (_,u) => (s,u)) (filter (fn (v,_) => eq (t,v)) ts)
              in
                distinct eq_pairs (filter (not o eq) (ts@List.concat (map aux ts)))
              end
            fun loop ts =
              let
                val ts' = trans_step ts
              in
                if eq_set eq_pairs (ts,ts')
                then ts
                else loop ts'
              end
          in
            loop
          end

        val ti = List.foldl fc [] fp
      in
        (filter fa fp, distinct eq (List.foldl fb [] fp@map snd ti), timplies_trancl ti)
      end

    fun check_no_vars_and_consts (fp:cMsg list) =
      let
        fun aux (cVar _) = false
          | aux (cConst _) = false
          | aux (cFun (_,ts)) = List.all aux ts
          | aux (cSet (_,ts)) = List.all aux ts
          | aux (cOccursFact bs) = aux bs
          | aux _ = true
      in
        if List.all aux fp
        then fp
        else error ("There shouldn't be any cVars and cConsts at this point in the " ^
                    "fixpoint translation")
      end
  in
    fp |> map (fn (m,t) => certifyMsg (map snd t, [], map (fn (a,b) => ([a],TAtom b)) t, []) m)
       |> ground_enum_variables
       |> map (transform_cMsg trac)
       |> check_no_vars_and_consts
       |> split_fp
  end

fun certifyAction params              (lbl,SEND ts)           = (lbl,cSend
      (map (certifyMsg params) ts))
  | certifyAction params              (lbl,RECEIVE ts)        = (lbl,cReceive
      (map (certifyMsg params) ts))
  | certifyAction params              (lbl,LETBINDING (t,t')) = (lbl,cEquality
      (cAssignment, (certifyMsg params t, certifyMsg params t')))
  | certifyAction params              (lbl,EQUATION (t,t'))   = (lbl,cEquality
      (cCheck, (certifyMsg params t, certifyMsg params t')))
  | certifyAction params              (lbl,IN (x,(s,ps)))     =
      let
        fun f (Enumeration _) = true
          | f (InfiniteEnumeration _) = true
```

```
        | f EnumType = true
        | f ValueType = true
        | f _ = false
      val taus = distinct (op =) (map (certifyMsgType params) (action_fvs (IN (x,(s,ps)))))
      val poscheckvariant = cAssignment (* TODO: fix *)
          (* if List.all f taus then cCheck else cAssignment *)
    in
      (lbl,cInSet (poscheckvariant, (certifyMsg params x,
                                     cSet (s, map (certifyMsg params) ps))))
    end
  | certifyAction params              (lbl,NOTINANY (x,s))    = (lbl,cNotInAny
      (certifyMsg params x, s))
  | certifyAction params              (lbl,NEGCHECKS (xs,ns)) = (lbl,cNegChecks
      (map (fn (x,tau) => (x,certifyMsgType' (#1 params) (#2 params) tau)) (typedvars_flatten xs),
       map (fn n => case n of
                      INEQ (t,t') => cInequality (certifyMsg params t,
                                                  certifyMsg params t')
                    | NOTIN (t,(s,ps)) => cNotInSet (certifyMsg params t,
                                                     cSet (s, map (certifyMsg params) ps)))
         ns))
  | certifyAction params              (lbl,INSERT (x,(s,ps))) = (lbl,cInsert
      (certifyMsg params x, cSet (s, map (certifyMsg params) ps)))
  | certifyAction params              (lbl,DELETE (x,(s,ps))) = (lbl,cDelete
      (certifyMsg params x, cSet (s, map (certifyMsg params) ps)))
  | certifyAction (fenums,ienums,_,_) (lbl,NEW xs)            = (lbl,cNew
      (let
        val xs = typedvars_flatten xs
        fun check (TAtom a) =
              if a = enum_trac_typeN
              then error "Error: The special enum type is not allowed in \"new\" actions"
              else if List.exists (fn e => a = e) (fenums@ienums)
              then error "Error: Enumeration annotations are not allowed in \"new\" actions"
              else TAtom a
          | check (TComp _) =
              error "Error: Composed type annotations in \"new\" actions are not allowed"
       in map (fn (x,tau) => (x,certifyMsgType' fenums ienums (check tau))) xs end))
  | certifyAction _                   (lbl,ATTACK)            = (lbl,cAssertAttack)

fun certifyTransactionName fenums ienums ((name, vars, ineqs):transaction_name) =
  let val cert_vars = map (fn x => (x,certifyMsgType (fenums, ienums, vars, []) x))
  in (name, List.concat (map (cert_vars o fst) vars), ineqs) end

fun certifyTransaction finite_enumerations infinite_enumerations (tr:transaction) =
  let
    val tr_acs = map (fn a => case a of
                               LABELED_ACTION p => p
                             | ABBREVIATION p =>
                                 error ("Error: Got an unexpected action sequence abbreviation " ^
                                        "(they should have all been expanded and removed at " ^
                                        "this point): " ^
                                        labeled_action_str (ABBREVIATION p)))
                     (#actions tr)

    val fresh_vars = List.concat (map_filter (maybe_the_NEW o snd) tr_acs)
    val fresh_vars' = typedvars_flatten fresh_vars
    val decl_vars = #2 (#transaction tr)
    val decl_vars' = List.concat (map fst decl_vars)
    val neq_constrs = #3 (#transaction tr)
    val bvars = List.concat (map fst (map_filter (maybe_the_NEGCHECKS o snd) tr_acs))

    val _ = if     List.exists (fn x     => List.exists (fn (y,_) => x = y) fresh_vars') decl_vars'
            orelse List.exists (fn (x,_) => List.exists (fn y     => x = y) decl_vars') fresh_vars'
            then error "The fresh and the declared variables must not overlap"
```

```
          else ()

    val _ = case List.find (fn (x,y) => x = y) neq_constrs of
              SOME (x,y) => error ("Illegal inequality constraint: " ^ x ^ " != " ^ y)
            | NONE => ()

    val cactions =
      let val xs = decl_vars@bvars
      in map (certifyAction (finite_enumerations, infinite_enumerations, xs, fresh_vars)) tr_acs
      end

    val cname = certifyTransactionName finite_enumerations infinite_enumerations (#transaction tr)

    fun is_poscheck1 (_,a) = is_Equality a orelse is_InSet a
    fun is_check1 p = is_poscheck1 p orelse is_NegChecks (snd p)

    val receives = filter (is_Receive o snd) cactions
    val checksingles = filter is_check1 cactions
    val checkalls = filter (is_NotInAny o snd) cactions
    val updates = filter (fn (_,a) => is_Insert a orelse is_Delete a) cactions
    val fresh = filter (is_Fresh o snd) cactions
    val sends = filter (is_Send o snd) cactions
    val attack_signals = filter (is_Attack o snd) cactions
  in
    {transaction = cname,
     receive_actions = receives,
     checksingle_actions = checksingles,
     checkall_actions = checkalls,
     fresh_actions = fresh,
     update_actions = updates,
     send_actions = sends,
     attack_actions = attack_signals}:cTransaction
  end

  fun get_finite_enum_spec_trac (trac:protocol) =
    let
      val spec = #enum_spec trac
      val finite_enum_spec =
        let
          fun is_finite e =
            List.exists
              (fn (s,t) => s = e andalso (case t of
                  TracProtocol.Consts _ => true
                | TracProtocol.Union ts => List.all is_finite ts
                | TracProtocol.InfiniteSet => false))
              spec
        in
          filter (is_finite o fst) spec
        end
    in
      finite_enum_spec
    end

  fun get_infinite_enum_spec_trac (trac:protocol) =
    filter_out (member (op =) (get_finite_enum_spec_trac trac)) (#enum_spec trac)

  fun get_finite_enum_names_trac (trac:protocol) =
    map fst (get_finite_enum_spec_trac trac)

  fun get_infinite_enum_names_trac (trac:protocol) =
    map fst (get_infinite_enum_spec_trac trac)

  fun get_enum_names_trac (trac:protocol) =
```

```
  map fst (#enum_spec trac)

fun get_funs_trac (trac:protocol) =
    let
      fun rm_special_funs sel l = list_minus (list_rm_pair sel) l special_funs
      fun append_sec fs = fs@[(priv_fun_secN, 0, NONE)]
      val filter_funs = filter (fn (_,n,_) => n <> 0)
      val filter_consts = filter (fn (_,n,_) => n = 0)
      fun inc_ar (s,n,tau) = (s, 1+n, tau)
    in
      case (#function_spec trac) of
           NONE => ([],[],[])
         | SOME ({public=pub, private=priv}) =>
             let
               val pub_symbols = rm_special_funs #1 (pub@map inc_ar (filter_funs priv))
               val pub_funs = filter_funs pub_symbols
               val pub_consts = filter_consts pub_symbols
               val priv_consts = append_sec (rm_special_funs #1 (filter_consts priv))
             in
               (pub_funs, pub_consts, priv_consts)
             end
    end


fun get_term_abbreviations_trac (trac:protocol) =
  map_filter (fn a => case a of TracProtocol.TermAbbreviation t => SOME t | _ => NONE)
             (#abbreviation_spec trac)

fun get_action_abbreviations_trac (trac:protocol) =
  map_filter (fn a => case a of TracProtocol.ActionsAbbreviation t => SOME t | _ => NONE)
             (#abbreviation_spec trac)

fun check_for_invalid_trac_specification (trac:TracProtocol.protocol) = let
    open Trac_Term TracProtocol

    datatype action_status =
      Passed | InvalidSetParam | WrongPosition | IllformedVars | InvalidAnnotationNewAction |
      InvalidFunctionSymbols of (string * int) list |
      InvalidSetSymbols of (string * int option) list |
      ComplexNegCheck

    val has_dups = has_duplicates (op =)
    val dups_str = String.concatWith ", " o duplicates (op =)

    val expand_abbrevs =
      expand_action_abbreviations (get_action_abbreviations_trac trac)

    val enumerations = get_enum_names_trac trac
    val finite_enumerations = get_finite_enum_names_trac trac
    val infinite_enumerations = get_infinite_enum_names_trac trac
    val set_names = map #1 (#set_spec trac)
    val set_spec = map (fn (s,n,_) => (s,n)) (#set_spec trac)
    val enum_consts = get_enum_consts_trac trac
    val fun_names = case #function_spec trac of
          SOME fs => map #1 ((#public fs)@(#private fs))
        | NONE => []
    val fun_spec = case #function_spec trac of
          SOME fs => map_filter
                       (fn (s,n,tau) => if n > 0 andalso tau = NONE then SOME (s,n) else NONE)
                       ((#public fs)@(#private fs))
        | NONE => []

    val ana_funs = map (#1 o #1) (#analysis_spec trac)
```

```
val ana_args = map (#2 o #1) (#analysis_spec trac)
val ana_has_illegal_var_in_body = not o
      (fn ((_,xs),ts,ys) => subset (op =) (ys@List.concat (map Trac_Term.fv_Msg ts), xs))

val abb_funs = map (fn a => case a of
                              TermAbbreviation ((f,_),_) => f
                            | ActionsAbbreviation ((f,_),_) => f)
                   (#abbreviation_spec trac)
val abb_args = map (fn a => case a of
                              TermAbbreviation ((_,xs),_) => xs
                            | ActionsAbbreviation ((_,xs),_) => xs)
                   (#abbreviation_spec trac)
fun abb_has_illegal_var_in_body (TermAbbreviation ((_,xs),t)) =
      not (subset (op =) (Trac_Term.fv_Msg t, xs))
  | abb_has_illegal_var_in_body (ActionsAbbreviation ((_,xs),acs)) =
      not (subset (op =) (List.concat (map (action_fvs o snd) (expand_abbrevs acs)), xs))

val trs = List.concat (map snd (#transaction_spec trac))
val tr_names = map (#1 o #transaction) trs
val tr_sec_names = map_filter #1 (#transaction_spec trac)
val tr_hds =
  map (fn tr => (#1 (#transaction tr), #2 (#transaction tr))) trs
val tr_acs =
  map (fn tr => (#1 (#transaction tr), #2 (#transaction tr),
                 map snd (expand_abbrevs (#actions tr)))) trs
val tr_mem_acs_sets =
  let
    val tr_mem_acs = filter (fn a => is_IN a orelse is_NOTINANY a orelse is_NEGCHECKS a)
                            (List.concat (map #3 tr_acs))
    fun f a =
      case a of
        IN (_,(s,_)) => [s]
      | NOTINANY (_,s) => [s]
      | NEGCHECKS (_,bs) =>
          map_filter (fn b => case b of NOTIN (_,(s,_)) => SOME s | _ => NONE) bs
      | _ => []
    val s = tr_mem_acs |> map f |> List.concat |> distinct (op =)
  in s end

val illegal_atomic_types = extended_extra_types
val new_action_illegal_annotations = enumerations@enum_trac_typeN::illegal_atomic_types
val illegal_composed_type_subterms = enumerations@value_trac_typeN::illegal_atomic_types

val user_types_overlapping_enums =
  filter (member (op =) (#type_spec trac)) enumerations

fun value_free_type (TAtom e) = e <> value_trac_typeN
  | value_free_type (TComp (_,ts)) = List.all value_free_type ts

fun var_decl_has_illegal_type (_,TAtom a) = List.exists (fn b => a = b) illegal_atomic_types
  | var_decl_has_illegal_type (_,TComp ts) =
      let
        val funs =
          case (#function_spec trac) of
            NONE => []
          | SOME {private=privs, public=pubs} => pubs@privs

        fun illegal_symbol a = List.exists (fn b => a = b) illegal_composed_type_subterms

        fun wrong_arity a bs =
          null bs orelse List.exists (fn (f,n,_) => f = a andalso length bs <> n) funs

        fun check (TAtom a) = illegal_symbol a
```

```
          | check (TComp (s,ts)) =
              illegal_symbol s orelse wrong_arity s ts orelse List.exists check ts
        in
          check (TComp ts)
        end

  fun no_value_vars_in_decl (tr:transaction) =
    List.all (value_free_type o snd) (#2 (#transaction tr))

  fun no_value_vars_in_decl_and_new_acs (tr:transaction) =
    no_value_vars_in_decl tr andalso
    List.all (List.all (value_free_type o snd))
(* (fn (_,t) => case t of SOME tau => value_free_type tau | _ => false) *)
             (map_filter (maybe_the_NEW o snd) (expand_abbrevs (#actions tr)))

  fun is_value_init_transaction (tr:transaction) =
    let
      val acs = map snd (expand_abbrevs (#actions tr))
      val priv_funs = case #function_spec trac of SOME fs => map #1 (#private fs) | NONE => []
      val decl = #2 (#transaction tr)
      fun is_not_value_var x =
        List.exists (fn (ys,t) => member (op =) ys x andalso value_free_type t) decl
      fun is_not_priv f = List.all (fn g => f <> g) priv_funs
      fun valid_msg (Var x) = is_not_value_var x
        | valid_msg (Const c) = is_not_priv c
        | valid_msg (Fun (f,ts)) = is_not_priv f andalso List.all valid_msg ts
        | valid_msg (Abbrev _) = false
        | valid_msg Attack = true
      fun NEW_action_with_value_annotations_only a =
        case a of
          NEW xss => List.all (fn (_,t) => t = TAtom value_trac_typeN) xss
        | _ => false
    in
      no_value_vars_in_decl tr andalso
      List.exists NEW_action_with_value_annotations_only acs andalso
      List.all (List.all valid_msg) (map_filter maybe_the_RECEIVE acs) andalso
      List.all (fn a => is_NEW a orelse is_INSERT a orelse is_SEND a) acs andalso
      List.all (fn (_,(s,_)) => not (member (op =) tr_mem_acs_sets s))
               (map_filter maybe_the_INSERT acs)
    end
  fun value_producing_transactions_requirement tr_secs =
    List.all (List.exists is_value_init_transaction o snd) tr_secs orelse
    List.all (List.all no_value_vars_in_decl_and_new_acs o snd) tr_secs

  fun is_value_var decls x =
    List.exists
      (fn (y,t) => x = y andalso t = TAtom value_trac_typeN)
      decls

  fun is_enum_var decls x =
    List.exists
      (fn (y,t) => x = y andalso List.exists (fn e => t = TAtom e) finite_enumerations)
      decls

  fun set_action_enum_params decls ps =
    List.all (fn p => case p of
        Var x => is_enum_var decls x
      | Const c => List.exists (fn b => b = c) enum_consts
      | Fun (c,ps) => ps = [] andalso List.exists (fn b => b = c) enum_consts
      | _ => false) ps

  fun set_action_param_check f ds (INSERT (_,(_,ps))) = f ds ps
    | set_action_param_check f ds (DELETE (_,(_,ps))) = f ds ps
```

```
  | set_action_param_check f ds (IN (_,(_,ps))) = f ds ps
  | set_action_param_check f ds (NEGCHECKS (_,ns)) =
      List.all (fn n => case n of NOTIN (_,(_,ps)) => f ds ps | _ => true) ns
  | set_action_param_check _ _ _ = true

fun wfst' xs [] = xs
  | wfst' xs (a::acs) = case a of
      (RECEIVE ts) => wfst' (distinct (op =) (xs@action_fvs (RECEIVE ts))) acs
    | (LETBINDING (t,_)) => wfst' (distinct (op =) (xs@msg_vars t)) acs
    | (IN p) => wfst' (distinct (op =) (xs@action_fvs (IN p))) acs
    | (NEW ys) => wfst' (xs@typedvars_fvs ys) acs
    | _ => wfst' xs acs

fun wfstp decl xs prev_fvs insert_send_fvs a = case a of
      (RECEIVE ts) => subset (op =) (action_fvs (RECEIVE ts), decl)
    | (SEND ts) => subset (op =) (action_fvs (SEND ts), xs)
    | (EQUATION p) => subset (op =) (action_fvs (EQUATION p), decl)
    | (LETBINDING (t,t')) =>
        subset (op =) (msg_vars t, decl) andalso
        subset (op =) (msg_vars t', xs)
    | (IN p) => subset (op =) (action_fvs (IN p), decl)
    | (NOTINANY p) => subset (op =) (action_fvs (NOTINANY p), decl)
    | (NEGCHECKS p) => subset (op =) (action_fvs (NEGCHECKS p), decl)
    | (INSERT p) => subset (op =) (action_fvs (INSERT p), xs)
    | (DELETE p) => subset (op =) (action_fvs (DELETE p), decl@xs)
    | (NEW ys) =>
        let val zs = typedvars_fvs ys
        in not (has_dups zs) andalso
           subset (op =) (zs, insert_send_fvs) andalso
           List.all (fn y =>
                       not (member (op =) decl y) andalso
                       not (member (op =) prev_fvs y))
                    zs
        end
    | ATTACK => true

fun wfst decl prev_acs next_acs a =
  let val f = map fst
      fun g (_,tau) = case tau of TAtom ta => member (op =) enumerations ta | _ => true
      val h = List.concat o map action_fvs
      val prev_fvs = h prev_acs
      val insert_send_fvs = h (filter (fn b => is_INSERT b orelse is_SEND b) next_acs)
  in wfstp (f decl) (wfst' (f (filter g decl)) prev_acs) prev_fvs insert_send_fvs a end

fun action_order_check _        (RECEIVE _) = true
  | action_order_check next_acs (LETBINDING _) = List.all (not o is_RECEIVE) next_acs
  | action_order_check next_acs (EQUATION _) = List.all (not o is_RECEIVE) next_acs
  | action_order_check next_acs (NEGCHECKS _) = List.all (not o is_RECEIVE) next_acs
  | action_order_check next_acs (IN _) = List.all (not o is_RECEIVE) next_acs
  | action_order_check next_acs (NOTINANY _) = List.all (not o is_RECEIVE) next_acs
  | action_order_check next_acs (NEW _) = List.all
      (fn a => is_NEW a orelse is_INSERT a orelse is_DELETE a orelse is_SEND a)
      next_acs
  | action_order_check next_acs (INSERT _) = List.all
      (fn a => is_NEW a orelse is_INSERT a orelse is_DELETE a orelse is_SEND a)
      next_acs
  | action_order_check next_acs (DELETE _) = List.all
      (fn a => is_NEW a orelse is_INSERT a orelse is_DELETE a orelse is_SEND a)
      next_acs
  | action_order_check next_acs (SEND _) = List.all is_SEND next_acs
  | action_order_check next_acs ATTACK = next_acs = []

fun new_action_legal_type_annotations a =
```

```
  let
    fun f (TAtom a) = List.all (fn b => a <> b) new_action_illegal_annotations
      | f (TComp _) = false
  in
    case a of
      (NEW xs) => List.all (f o snd) xs
    | _ => true
  end

val invalid_funs_in_msg =
  let
    val dist = distinct (op =)
    val conc = dist o List.concat o (fn (f,ms) => map f ms)
    fun f (Var _) = []
      | f (Const _) = []
      | f (Fun (g,ms)) =
          let val n = (g,length ms)
              val ns = conc (f,ms)
          in if member (op =) fun_spec n then ns else dist (n::ns)
          end
      | f (Abbrev (_,ms)) = conc (f,ms)
      | f Attack = []
  in
    f
  end

fun invalid_funs_in_action a =
  let
    val dist = distinct (op =)
    val conc = dist o List.concat o (fn (f,ms) => map f ms)
    val f = invalid_funs_in_msg
    fun fnc [] = []
      | fnc (INEQ (t,t')::ps) = t::t'::fnc ps
      | fnc (NOTIN (t,(_,ts))::ps) = t::ts@fnc ps
  in
    case a of
      (RECEIVE ts) => conc (f,ts)
    | (SEND ts) => conc (f,ts)
    | (EQUATION (t,t')) => conc (f,[t,t'])
    | (LETBINDING (t,t')) => conc (f,[t,t'])
    | (IN (t,(_,ts))) => conc (f,t::ts)
    | (NOTINANY (t,_)) => f t
    | (NEGCHECKS (_,ps)) => conc (f,fnc ps)
    | (INSERT (t,(_,ts))) => conc (f,t::ts)
    | (DELETE (t,(_,ts))) => conc (f,t::ts)
    | (NEW _) => []
    | ATTACK => []
  end

fun invalid_sets_in_action a =
  let
    val dist = distinct (op =)
    val conc = dist o (fn (f,ns) => f ns)
    fun f [] = []
      | f ((s,SOME n)::ns) =
          if member (op =) set_spec (s,n) then f ns else (s,SOME n)::f ns
      | f ((s,NONE)::ns) =
          if member (op =) set_names s then f ns else (s,NONE)::f ns
    fun fnc [] = []
      | fnc (INEQ _::ps) = fnc ps
      | fnc (NOTIN (_,(s,ts))::ps) = (s,SOME (length ts))::fnc ps
  in
    case a of
```

```
        (RECEIVE _) => []
      | (SEND _) => []
      | (EQUATION _) => []
      | (LETBINDING _) => []
      | (IN (_,(s,ts))) => conc (f,[(s,SOME (length ts))])
      | (NOTINANY (_,s)) => conc (f,[(s,NONE)])
      | (NEGCHECKS (_,ps)) => conc (f,fnc ps)
      | (INSERT (_,(s,ts))) => conc (f,[(s,SOME (length ts))])
      | (DELETE (_,(s,ts))) => conc (f,[(s,SOME (length ts))])
      | (NEW _) => []
      | ATTACK => []
    end

  val invalid_funs_in_abbrevs =
    let
      val distconc = distinct (op =) o List.concat
      fun f (LABELED_ACTION (_,a)) = invalid_funs_in_action a
        | f (ABBREVIATION (_,ms)) = distconc (map invalid_funs_in_msg ms)
      fun g (TermAbbreviation ((s,_),m)) = (s,invalid_funs_in_msg m)
        | g (ActionsAbbreviation ((s,_),acs)) = (s,distconc (map f acs))
    in
      filter (fn (_,l) => l <> []) (map g (#abbreviation_spec trac))
    end

  val invalid_sets_in_abbrevs =
    let
      val distconc = distinct (op =) o List.concat
      fun f (LABELED_ACTION (_,a)) = invalid_sets_in_action a
        | f (ABBREVIATION _) = []
      fun g (TermAbbreviation ((s,_),_)) = (s,[])
        | g (ActionsAbbreviation ((s,_),acs)) = (s,distconc (map f acs))
    in
      filter (fn (_,l) => l <> []) (map g (#abbreviation_spec trac))
    end

  fun negcheck_is_empty_bvars_val_ineq_or_notin decls a =
    case a of
      (NOTINANY (Var x, _)) =>
        is_value_var decls x
    | (NEGCHECKS ([], [INEQ (Var x, Var y)])) =>
        is_value_var decls x andalso is_value_var decls y
    | (NEGCHECKS ([], [NOTIN (Var x, (_, ps))])) =>
        is_value_var decls x andalso set_action_enum_params decls ps
    | _ => true

  fun check_actions (tr_name,decl,acs) =
    let fun chk i =
          let val decl_flat = List.concat (map (fn (xs,t) => map (fn x => (x,t)) xs) decl)
              val a = nth acs i
              fun result st = (st,tr_name,a)
              val fs = invalid_funs_in_action a
              val gs = invalid_sets_in_action a
              val prev_acs = List.take (acs,i)
              val next_acs = List.drop (acs,i+1)
          in if fs <> []
             then result (InvalidFunctionSymbols fs)
             else if gs <> []
             then result (InvalidSetSymbols gs)
             else if not (set_action_param_check set_action_enum_params decl_flat a)
             then result InvalidSetParam
             else if not (action_order_check next_acs a)
             then result WrongPosition
             else if not (wfst decl_flat prev_acs next_acs a)
```

```
            then result IllformedVars
            else if not (new_action_legal_type_annotations a)
            then result InvalidAnnotationNewAction
            else if not (negcheck_is_empty_bvars_val_ineq_or_notin decl_flat a)
            then result ComplexNegCheck
            else result Passed
         end
      in map chk (0 upto (length acs - 1))
      end

    val checked_tr_acs = List.concat (map check_actions tr_acs)

    fun violating_action_exists' f =
      List.exists (f o #1) checked_tr_acs

    fun violating_action_exists status =
      violating_action_exists' (fn a => a = status)

    val violating_action_exists_unk_fun_sym =
      violating_action_exists' (fn a => case a of InvalidFunctionSymbols _ => true | _ => false)

    val violating_action_exists_unk_set_sym =
      violating_action_exists' (fn a => case a of InvalidSetSymbols _ => true | _ => false)

    fun violating_actions_str' f g =
      String.concatWith "\n" (
        map (fn (st,n,a) => g (st,n,action_str a))
            (filter (f o #1) checked_tr_acs))

    val violating_actions_str_unk_fun_sym =
      let
        fun f a = case a of InvalidFunctionSymbols fs => SOME fs | _ => NONE
      in
        violating_actions_str' (fn a => f a <> NONE)
          (fn (st,n,_) => "symbol(s) " ^
                          String.concatWith ", " (map (fn (s,n) => s ^ "/" ^ Int.toString n)
                                                      (Option.getOpt (f st, []))) ^
                          " in transaction \"" ^ n ^ "\"")
      end

    val violating_actions_str_unk_set_sym =
      let
        fun f a = case a of InvalidSetSymbols fs => SOME fs | _ => NONE
      in
        violating_actions_str' (fn a => f a <> NONE)
          (fn (st,n,_) => "symbol(s) " ^
                          String.concatWith ", "
                            (map (fn (s,n) => s ^ (case n of SOME n' => "/" ^ Int.toString n'
                                                            | _ => ""))
                                 (Option.getOpt (f st, []))) ^
                          " in transaction \"" ^ n ^ "\"")
      end

    fun violating_actions_str status =
      violating_actions_str'
        (fn a => a = status)
        (fn (_,n,a) => "action \"" ^ a ^ "\" in transaction \"" ^ n ^ "\"")
  in
    if has_dups tr_sec_names
    then error (
      "Multiple Transactions sections declared with the same name:\n" ^ dups_str tr_sec_names)
    else if has_dups tr_names
    then error (
```

```
    "Duplicate transaction declarations:\n" ^ dups_str tr_names)
  else if has_dups enumerations
  then error (
    "Multiple declarations of the same enumeration:\n" ^ dups_str enumerations)
  else if List.exists (fn n => n = value_trac_typeN) enumerations
  then error (
    "The special type \"" ^ value_trac_typeN ^ "\" should not be declared in the trac " ^
    "specification.")
  else if List.exists (fn n => n = enum_trac_typeN) enumerations
  then error (
    "The special type \"" ^ enum_trac_typeN ^ "\" should not be declared in the trac " ^
    "specification.")
  else if has_dups set_names
  then error (
    "Multiple declarations of the same set families:\n" ^ dups_str set_names)
  else if has_dups (fun_names@enum_consts)
  then error (
    "Multiple declarations of the same constant or function symbols:\n" ^
    dups_str (fun_names@enum_consts))
  else if has_dups ana_funs
  then error (
    "Multiple analysis rules declared for the same function symbols:\n" ^ dups_str ana_funs)
  else if has_dups abb_funs
  then error (
    "Multiple abbreviations declared with the same name:\n" ^ dups_str abb_funs)
  else if List.exists has_dups ana_args
  then error (
    "The heads of the analysis rules must be linear terms, " ^
    "i.e., of the form f(X1,...,Xn) for distinct X1,...,Xn.\n" ^
    "The analysis rules with the following heads violate this condition:\n" ^
    String.concatWith "\n" (
      map (fn i => nth ana_funs i ^ "(" ^ String.concatWith "," (nth ana_args i) ^ ")")
          (filter (has_dups o (nth ana_args)) (0 upto (length (#analysis_spec trac) - 1)))))
  else if List.exists ana_has_illegal_var_in_body (#analysis_spec trac)
  then error (
    "Variables occurring in the body of an analysis rule must also occur in its head.\n" ^
    "The analysis rules with the following heads violate this condition:\n" ^
    String.concatWith "\n" (
      map (fn i => nth ana_funs i ^ "(" ^ String.concatWith "," (nth ana_args i) ^ ")")
          (filter (ana_has_illegal_var_in_body o (nth (#analysis_spec trac)))
                  (0 upto (length (#analysis_spec trac) - 1)))))
  else if List.exists has_dups abb_args
  then error (
    "The heads of the abbreviation declarations must be linear terms, " ^
    "i.e., of the form f[X1,...,Xn] for distinct X1,...,Xn.\n" ^
    "The abbreviation declaration with the following heads violate this condition:\n" ^
    String.concatWith "\n" (
      map (fn i => nth abb_funs i ^ "![" ^ String.concatWith "," (nth abb_args i) ^ "]")
          (filter (has_dups o (nth abb_args)) (0 upto (length (#abbreviation_spec trac) - 1)))))
  else if List.exists abb_has_illegal_var_in_body (#abbreviation_spec trac)
  then error (
    "Variables occurring in the body of an abbreviation declaration must also occur in its " ^
    "head.\nThe abbreviation declarations with the following heads violate this condition:\n" ^
    String.concatWith "\n" (
      map (fn i => nth abb_funs i ^ "![" ^ String.concatWith "," (nth abb_args i) ^ "]")
          (filter (abb_has_illegal_var_in_body o (nth (#abbreviation_spec trac)))
                  (0 upto (length (#abbreviation_spec trac) - 1)))))
  else if not (null user_types_overlapping_enums)
  then error (
    "Types declared in the \"Types\" section cannot also be declared as enumerations in " ^
    "the \"Enumerations\" section.\nThe following types violate this condition:\n" ^
    String.concatWith ", " user_types_overlapping_enums)
  else if List.exists (List.exists var_decl_has_illegal_type o snd) tr_hds
```

```
then error (
  "Transactions must satisfy certain well-formedness requirements on the variables " ^
  "declared in their heads:\n" ^
  "1. The only special atomic types that may occur in the variable declarations are " ^
  "\"" ^ value_trac_typeN ^ "\" and \"" ^ enum_trac_typeN ^ "\". In particular, the " ^
  "following special types are not allowed: " ^
  String.concatWith ", " illegal_atomic_types ^ "\n" ^
  "2. For variables declared with composed types no enumeration or special type besides " ^
  "\"" ^ enum_trac_typeN ^ "\" may occur in their types. In particular, the following " ^
  "cannot occur in composed types: " ^
  String.concatWith ", " illegal_composed_type_subterms ^ "\n" ^
  "3. The number of parameters applied to a composed type must agree with the arity of " ^
  "the function symbol associated with that type.\n" ^
  "The following variable declarations violate these requirements:\n" ^
  String.concatWith "\n" (
    map_filter (fn (n,decls) =>
                  let val ds =
                        List.concat (map (fn (xs,t) => map (fn x => (x,t)) xs) decls)
                      val ds' = filter var_decl_has_illegal_type ds
                  in if null ds' then NONE
                  else SOME (String.concatWith "\n"
                              (map (fn (s,t) => s ^ ": " ^ MsgType_str t) ds') ^
                             " in transaction " ^ n)
                  end)
               tr_hds))
else if invalid_funs_in_abbrevs <> []
then error (
  "Function symbols occurring in abbreviations in the \"Abbreviations\" section must be " ^
  "declared in the \"Functions\" section and must be applied with the correct number of " ^
  "arguments.\nThe following function symbols violate this requirement:\n" ^
  String.concatWith "\n" (
    map (fn (ab,fs) =>
          "symbol(s) " ^
          String.concatWith ", " (map (fn (f,n) => f ^ "/" ^ Int.toString n) fs) ^
          " in abbreviation \"" ^ ab ^ "\"")
        invalid_funs_in_abbrevs))
else if invalid_sets_in_abbrevs <> []
then error (
  "Set symbols occurring in abbreviations in the \"Abbreviations\" section must be " ^
  "declared in the \"Sets\" section and must be applied with the correct number of " ^
  "arguments.\nThe following set symbols violate this requirement:\n" ^
  String.concatWith "\n" (
    map (fn (ab,fs) =>
          "symbol(s) " ^
          String.concatWith ", " (
            map (fn (f,n) => f ^ (case n of SOME n' => "/" ^ Int.toString n' | _ => ""))
                fs) ^
          " in abbreviation \"" ^ ab ^ "\"")
        invalid_sets_in_abbrevs))
else if violating_action_exists_unk_fun_sym
then error (
  "Function symbols occurring in transactions in the \"Transactions\" section must be " ^
  "declared in the \"Functions\" section and must be applied with the correct number of " ^
  "arguments.\nThe following function symbols violate this requirement:\n" ^
  violating_actions_str_unk_fun_sym)
else if violating_action_exists_unk_set_sym
then error (
  "Set symbols occurring in transactions in the \"Transactions\" section must be " ^
  "declared in the \"Sets\" section and must be applied with the correct number of " ^
  "arguments.\nThe following set symbols violate this requirement:\n" ^
  violating_actions_str_unk_set_sym)
else if violating_action_exists WrongPosition
then error (
```

```
    "The sequence of actions occurring in each transaction must either be of the form " ^
    "(written here in standard regular expression syntax)\n" ^
    "  (receive t)* (x in s | x notin s | let t = t' | t == t' | t != t')* " ^
    "(new x | insert x s | delete x s)* (send t)*\n" ^
    "or of the form\n" ^
    "  (receive t)* (x in s | x notin s | let t = t' | t == t' | t != t')* attack\n" ^
    "The following actions lead to violations of these requirements:\n" ^
    violating_actions_str WrongPosition)
  else if violating_action_exists IllformedVars
  then error (
    "The following well-formedness requirement on the occurrences of variables in " ^
    "transactions must be satisfied:\n" ^
    "1. Variables in \"send\", \"in\", \"notin\", \"let\", \"==\", and \"!=\" actions must " ^
    "be declared in the head of the transaction where these actions occur, or, in the case " ^
    "of negative checks, be bound by a \"forall\" quantifier.\n" ^
    "2. Variables in a \"new\" action must not occur previously in the same transaction, " ^
    "they must be distinct, and they must each occur in either an \"insert\" or a \"send\" " ^
    "action in the same transaction.\n" ^
    "3. Variables in \"insert\", \"delete\", and \"send\" actions must occur previously " ^
    "in the same transaction.\n" ^
    "The following actions lead to violations of these requirements:\n" ^
    violating_actions_str IllformedVars)
  else if violating_action_exists InvalidAnnotationNewAction
  then error (
    "Annotating variables in \"new\" actions with either enumerations, composed types, or " ^
    "special types besides \"" ^ value_trac_typeN ^ "\" is not allowed.\nIn particular, the " ^
    "following enumerations and atomic types cannot be used in \"new\" actions:\n" ^
    String.concatWith ", " new_action_illegal_annotations ^ "\n" ^
    "The following actions violate this requirement:\n" ^
    violating_actions_str InvalidAnnotationNewAction)
  else let
    val ws = [
      (violating_action_exists InvalidSetParam,
       "The parameters to a set-expression must be finite enumerations declared in the " ^
       "\"Enumerations\" section of the trac specification, and must furthermore be " ^
       "declared in the transaction where the set-expression occurs. In particular, they " ^
       "must not be variables of type \"" ^ value_trac_typeN ^ "\".\n" ^
       "The following actions violate these requirements:\n" ^
       violating_actions_str InvalidSetParam),
      (violating_action_exists ComplexNegCheck,
       "Each negative check occurring in the body of a transaction must either be of " ^
       "the form\n" ^
       "1. \"X != Y\" for variables \"X\" and \"Y\" of type \"" ^ value_trac_typeN ^
       "\", or\n" ^
       "2. \"X notin s(_)\", where \"X\" is a variable of type " ^ value_trac_typeN ^
       " and \"s\" is a set symbol, or\n" ^
       "3. \"X notin s(t1,...,tn)\" where \"X\" is a variable of type " ^ value_trac_typeN ^
       ", \"s\" is a set symbol, and the parameters \"ti\" range over enumerations.\n" ^
       "NB: \"!=\"-constraints on finite-enumeration-variables can be declared in the head " ^
       "of the transaction in question using the \"where\"-keyword. E.g., " ^
       "\"tr(A:agent,B:agent) where A != B\" denotes that \"A\" and \"B\" are different in " ^
       "the transaction \"tr\" and that they both range over the enumeration \"agent\".\n" ^
       "The following actions violate this requirement:\n" ^
       violating_actions_str ComplexNegCheck)
      ]
    val _ = if List.exists fst ws
            then let
              val _ = warning ("Warning: The specification is not suitable for automated " ^
                               "verification. To enable automation the following issues " ^
                               "need to be resolved:")
            in fold (fn (b,w) => fn _ => if b then warning w else ()) ws () end
            else ()
  in trac end
```

```
    end

fun certifyProtocol (trac:protocol) =
  let
    fun expand_abbreviations (trac:protocol) =
      let
        val expand_tabbs  = expand_term_abbreviations (get_term_abbreviations_trac trac)
        val expand_taabbs = expand_term_abbreviations_in_action (get_term_abbreviations_trac trac)
        val expand_aabbs  = map (fn (lbl,ac) => LABELED_ACTION (lbl,expand_taabbs ac)) o
                            expand_action_abbreviations (get_action_abbreviations_trac trac)
      in
        ({name = #name trac
         ,type_spec = #type_spec trac
         ,enum_spec = #enum_spec trac
         ,set_spec = #set_spec trac
         ,function_spec = #function_spec trac
         ,analysis_spec =
            map (fn (h,ks,rs) => (h,map expand_tabbs ks,rs)) (#analysis_spec trac)
         ,abbreviation_spec = []
         ,transaction_spec =
            map (fn (n,trs) =>
                  (n,map (fn tr => {transaction=(#transaction tr),
                                    actions=(expand_aabbs (#actions tr))})
                         trs))
                (#transaction_spec trac)
         ,fixed_point =
            Option.map (map (fn (t,xs) => (expand_tabbs t,xs))) (#fixed_point trac)
        })
      end

    fun transform_cAction (trac:protocol) =
      let
        val pfe = transform_cMsg trac
        val pte = priv_fun_type_enc trac
        fun pne (cInequality (t,t')) = cInequality (pfe t,pfe t')
          | pne (cNotInSet (t,t')) = cNotInSet (pfe t,pfe t')
        fun aux (cReceive ts) = cReceive (map pfe ts)
          | aux (cSend ts) = cSend (map pfe ts)
          | aux (cEquality (psv,(t,t'))) = cEquality (psv,(pfe t,pfe t'))
          | aux (cInSet (psv,(t,t'))) = cInSet (psv,(pfe t,pfe t'))
          | aux (cNotInAny (t,s)) = cNotInAny (pfe t,s)
          | aux (cNegChecks (xs,ns)) = cNegChecks (map (fn (x,tau) => (x,pte tau)) xs, map pne ns)
          | aux (cInsert (t,t')) = cInsert (pfe t,pfe t')
          | aux (cDelete (t,t')) = cDelete (pfe t,pfe t')
          | aux (cNew xs) = cNew (map (fn (x,tau) => (x,pte tau)) xs)
          | aux cAssertAttack = cAssertAttack
      in aux end

    fun transform_cTransaction (trac:protocol) (tr:cTransaction) =
      let
        val pae = map (fn (lbl,ac) => (lbl,transform_cAction trac ac))
        val pte = priv_fun_type_enc trac
      in
       {transaction=(case (#transaction tr) of (a,b,c) => (a,map (fn (x,tau) => (x,pte tau)) b,c))
       ,receive_actions=pae (#receive_actions tr)
       ,checksingle_actions=pae (#checksingle_actions tr)
       ,checkall_actions=pae (#checkall_actions tr)
       ,fresh_actions=pae (#fresh_actions tr)
       ,update_actions=pae (#update_actions tr)
       ,send_actions=pae (#send_actions tr)
       ,attack_actions=pae (#attack_actions tr)}
      end
```

```
fun certify (trac:protocol) =
  let
    val certify_ana_msg = transform_cMsg trac o certifyMsgUntyped
    val certify_type = priv_fun_type_enc trac o
                       certifyMsgType' (get_finite_enum_names_trac trac)
                                       (get_infinite_enum_names_trac trac)
    val certify_transaction = transform_cTransaction trac o
                              certifyTransaction (get_finite_enum_names_trac trac)
                                                 (get_infinite_enum_names_trac trac)

    val cert_fun_spec =
      let
        fun invalid (_,n,SOME (Trac_Term.TAtom _)) = n <> 0
          | invalid (_,_,SOME (Trac_Term.TComp _)) = true
          | invalid (_,_,NONE) = false
        val _ = case #function_spec trac of
                  SOME {private=priv, public=pub} =>
                    if List.exists invalid (priv@pub)
                    then error ("Error: Invalid type annotation in function specification. " ^
                                "Only constants may be annotated with types, and only with " ^
                                "atomic types.")
                    else ()
                | NONE => ()

        fun cert_const (a,b,c) =
          if b = 0
          then (a,Option.map (fn tau =>
                    (case certify_type tau of
                        AtomicType s => s
                      | _ => error ("Error: Invalid type annotation in function " ^
                                    "specification: " ^ MsgType_str tau))) c)
          else error ("Error: Expected arity 0 for function symbol " ^ a ^
                      " but got " ^ Int.toString b)
        fun cert_fun (a,b,c) =
          case c of
            NONE => (a,b)
          | SOME tau =>
              error ("Error: Expected no type annotation for function symbol " ^ a ^
                     " but got " ^ MsgType_str tau)
      in
        if #function_spec trac = NONE then NONE
        else case get_funs_trac trac of
          (pub_funs, pub_consts, priv_consts) =>
            SOME ({private_consts=map cert_const priv_consts,
                   public_consts=map cert_const pub_consts,
                   public_funs=map cert_fun pub_funs})
      end

    val cert_ana_spec =
      let
        val (pub_f, _, _) = get_funs_trac trac
        fun ana_arity (f,n) = (if is_priv_fun_trac trac f then n-1 else n)
        fun check_valid_arity ((f,ps),ks,rs) =
          case List.find (fn g => f = #1 g) pub_f of
            SOME (f',n,_) =>
              if length ps <> ana_arity (f',n)
              then error ("Error: Invalid number of parameters in the analysis rule for " ^ f ^
                          " (expected " ^ Int.toString (ana_arity (f',n)) ^
                          " but got " ^ Int.toString (length ps) ^ ")")
              else ((f,ps),ks,rs)
          | NONE => error ("Error: " ^ f ^
                           " is not a declared function symbol of arity greater than zero")
      in
```

```
        map (fn (h,ks,rs) => {
              head=h, keys=map certify_ana_msg ks,
              results=rs, is_priv_fun=is_priv_fun_trac trac (fst h)})
            (map check_valid_arity (#analysis_spec trac))
      end

    val (cert_transaction_spec:(string option * cTransaction list) list) =
      map (fn (n,trs) => (n,map certify_transaction trs))
          (#transaction_spec trac)

    val cert_fp =
      Option.map (certify_fixpoint trac) (#fixed_point trac)
  in
    ({name = #name trac
     ,type_spec = #type_spec trac
     ,enum_spec = flatten_enum_spec_trac trac
     ,set_spec = #set_spec trac
     ,function_spec = cert_fun_spec
     ,analysis_spec = cert_ana_spec
     ,transaction_spec = cert_transaction_spec
     ,fixed_point = cert_fp
    })
  end

fun add_intruder_value_gen_transaction (trac:protocol) =
  let
    val spec_tr_names =
      List.concat (map (map (#1 o #transaction) o snd) (#transaction_spec trac))
    val spec_set_names = map #1 (#set_spec trac)
    val spec_protnames =
      let
        val optnames = map #1 (#transaction_spec trac)
        val names = map_index (fn (n,optn) => Option.getOpt (optn,Int.toString n)) optnames
      in names end
    val spec_atmost1prot = case spec_protnames of (_::_::_) => false | _ => true

    fun name_free ns n = List.all (fn s => s <> n) ns
    fun gen_name prefix names n =
      if name_free names prefix then prefix
      else if name_free names (prefix ^ Int.toString n) then prefix ^ Int.toString n
      else gen_name prefix names (n+1)

    val set_def = (gen_name "intruderValues" spec_set_names 0,0,false):set_spec_elem

    fun tr_name suffix =
      let val s = if spec_atmost1prot then "" else "_" ^ suffix
      in (gen_name ("intruderValueGen" ^ s) spec_tr_names 0,[],[]):transaction_name end

    fun valuegentr protname = {
      transaction=tr_name protname,
      actions=[
        LABELED_ACTION (LabelS,NEW [(["X"],TAtom(value_trac_typeN))]),
        LABELED_ACTION (LabelS,INSERT (Var "X",(#1 set_def,[]))),
        LABELED_ACTION (LabelS,SEND [Var "X"])
      ]}:transaction

    val expand_abbrevs =
      expand_action_abbreviations (get_action_abbreviations_trac trac)

    val checks_and_deletes_sets =
      let
        fun f a = case a of
          DELETE (_,(s,_)) => [s]
```

```
          | IN (_,(s,_)) => [s]
          | NOTINANY (_,s) => [s]
          | NEGCHECKS (_,bs) =>
              map_filter (fn b => case b of NOTIN (_,(s,_)) => SOME s | _ => NONE) bs
          | _ => []

          val acs = List.concat (map #actions (List.concat (map (#2) (#transaction_spec trac))))
          val sets = List.concat (map (f o #2) (expand_abbrevs acs))
        in sets end

      fun has_valuegentr (trs:transaction list) =
        let fun is_valuegentr_variant1 acs = case acs of
                [LABELED_ACTION (LabelS,NEW [([x],TAtom(tau))]),
                 LABELED_ACTION (lbl,SEND ts)]
                  => tau = value_trac_typeN andalso
                     member (op =) ts (Var x) andalso
                     (lbl = LabelS orelse spec_atmost1prot)
              | _ => false

            fun is_valuegentr_variant2 acs = case acs of
                [LABELED_ACTION (LabelS,NEW [([x],TAtom(tau))]),
                 LABELED_ACTION (lbl,INSERT (y,(s,[]))),
                 LABELED_ACTION (lbl',SEND ts)]
                  => tau = value_trac_typeN andalso
                     member (op =) ts (Var x) andalso
                     y = Var x andalso
                     not (member (op =) checks_and_deletes_sets s) andalso
                     ((lbl = LabelS andalso lbl' = LabelS) orelse spec_atmost1prot)
              | _ => false

            fun is_valuegentr {transaction=(_,args,ineqs),actions=acs} =
              List.null args andalso List.null ineqs andalso
              (is_valuegentr_variant1 acs orelse is_valuegentr_variant2 acs)
        in List.exists (is_valuegentr) trs end
    in
      ({name = #name trac
      ,type_spec = #type_spec trac
      ,enum_spec = #enum_spec trac
      ,set_spec = set_def::(#set_spec trac)
      ,function_spec = #function_spec trac
      ,analysis_spec = #analysis_spec trac
      ,abbreviation_spec = #abbreviation_spec trac
      ,transaction_spec =
          map_index (fn (i,(n,trs)) =>
                          if has_valuegentr trs then (n,trs)
                          else (n,valuegentr (nth spec_protnames i)::trs))
                    (#transaction_spec trac)
      ,fixed_point = #fixed_point trac
      }):protocol
    end
  in
    (trac |> check_for_invalid_trac_specification
          |> add_intruder_value_gen_transaction
          |> expand_abbreviations
          |> certify
    ):cProtocol
  end


end
›
```

end

4.4 Parser for Trac FP definitions

theory
  trac_fp_parser
  imports
    "trac_term"
begin

ML_file *"trac_parser/trac_fp.grm.sig"*
ML_file *"trac_parser/trac_fp.lex.sml"*
ML_file *"trac_parser/trac_fp.grm.sml"*

ML‹
```
structure TracFpParser : sig
     val parse_file: string -> (Trac_Term.Msg * (string * string) list) list
       val parse_str: string -> (Trac_Term.Msg * (string * string) list) list
end =
struct

  open Trac_Term

  structure TracLrVals =
    TracLrValsFun(structure Token = LrParser.Token)

  structure TracLex =
    TracLexFun(structure Tokens = TracLrVals.Tokens)

  structure TracParser =
    Join(structure LrParser = LrParser
  structure ParserData = TracLrVals.ParserData
  structure Lex = TracLex)

  fun invoke lexstream =
      let fun print_error (s,i:(int * int * int),_) =
       TextIO.output(TextIO.stdOut,
       "Error, line .... " ^ (Int.toString (#1 i)) ^"."^(Int.toString (#2 i ))^ ", " ^ s ^ "\n")
       in TracParser.parse(0,lexstream,print_error,())
      end

 fun parse_fp lexer =  let
    val dummyEOF = TracLrVals.Tokens.EOF((0,0,0),(0,0,0))
    fun loop lexer =
      let
        val _ = (TracLex.UserDeclarations.pos := (0,0,0);())
        val (res,lexer) = invoke lexer
        val (nextToken,lexer) = TracParser.Stream.get lexer
      in if TracParser.sameToken(nextToken,dummyEOF) then ((),res) else loop lexer end
  in #2(loop lexer)
  end

 fun parse_file tracFile = let
      val infile = TextIO.openIn tracFile
      val lexer = TracParser.makeLexer  (fn _ => case ((TextIO.inputLine) infile) of
                                             SOME s => s
                                           | NONE   => "")
     in
       parse_fp lexer
     end

 fun parse_str trac_fp_str = let
```

```
       val parsed = Unsynchronized.ref false
       fun input_string _  = if !parsed then "" else (parsed := true ;trac_fp_str)
      val lexer = TracParser.makeLexer input_string
     in
       parse_fp lexer
     end
end
›
```

end

4.5 Parser for the Trac Format

theory
  trac_protocol_parser
  imports
    "trac_term"
begin

ML_file *"trac_parser/trac_protocol.grm.sig"*
ML_file *"trac_parser/trac_protocol.lex.sml"*
ML_file *"trac_parser/trac_protocol.grm.sml"*

ML‹
```
structure TracProtocolParser : sig
     val parse_file: string -> TracProtocol.protocol
       val parse_str:  string -> TracProtocol.protocol
end =
struct

  structure TracLrVals =
    TracTransactionLrValsFun(structure Token = LrParser.Token)

  structure TracLex =
    TracTransactionLexFun(structure Tokens = TracLrVals.Tokens)

  structure TracParser =
    Join(structure LrParser = LrParser
  structure ParserData = TracLrVals.ParserData
  structure Lex = TracLex)

  fun invoke lexstream =
      let fun print_error (s,i:(int * int * int),_) =
       error("Error, line .... " ^ (Int.toString (#1 i)) ^"."^(Int.toString (#2 i ))^ ", " ^ s ^ "\n")
       in TracParser.parse(0,lexstream,print_error,())
      end

 fun parse_fp lexer =  let
   val dummyEOF = TracLrVals.Tokens.EOF((0,0,0),(0,0,0))
   fun loop lexer =
       let
    val _ = (TracLex.UserDeclarations.pos := (0,0,0);())
    val (res,lexer) = invoke lexer
    val (nextToken,lexer) = TracParser.Stream.get lexer
        in if TracParser.sameToken(nextToken,dummyEOF) then ((),res)
    else loop lexer
       end
       in  (#2(loop lexer))
      end

 fun parse_file tracFile =
```

```
    let
        val infile = TextIO.openIn tracFile
        val lexer = TracParser.makeLexer  (fn _ => case ((TextIO.inputLine) infile) of
                                                      SOME s => s
                                                     | NONE   => "")
    in
      parse_fp lexer
     handle LrParser.ParseError => TracProtocol.empty
    end

 fun parse_str str =
    let
         val parsed = Unsynchronized.ref false
         fun input_string _  = if !parsed then "" else (parsed := true ;str)
        val lexer = TracParser.makeLexer input_string
    in
      parse_fp lexer
     handle LrParser.ParseError => TracProtocol.empty
    end

end
›
```

end

4.6 Support for the Trac Format

theory
  "trac"
imports
  trac_fp_parser
  trac_protocol_parser
keywords
      "trac" :: thy_decl
  and *"trac_import" :: thy_decl*
  and *"print_transaction_strand" :: thy_decl*
  and *"print_transaction_strand_list" :: thy_decl*
  and *"print_attack_trace" :: thy_decl*
  and *"print_fixpoint" :: thy_decl*
  and *"save_fixpoint" :: thy_decl*
  and *"load_fixpoint" :: thy_decl*
  and *"protocol_model_setup" :: thy_decl*
  and *"protocol_security_proof" :: thy_decl*
  and *"protocol_security_proof_parallel" :: thy_decl*
  and *"protocol_security_proof_safe_heuristic" :: thy_decl*
  and *"protocol_composition_proof" :: thy_decl*
  and *"manual_protocol_model_setup" :: thy_decl*
  and *"manual_protocol_security_proof" :: thy_decl*
  and *"manual_protocol_composition_proof" :: thy_decl*
  and *"compute_fixpoint" :: thy_decl*
  and *"compute_SMP" :: thy_decl*
  and *"compute_shared_secrets" :: thy_decl*
  and *"setup_protocol_checks" :: thy_decl*
begin

ML‹
```
val pspsp_timing = let
  val (pspsp_timing_config, pspsp_timing_setup) =
      Attrib.config_bool (Binding.name "pspsp_timing") (K false)
in
  Context.>>(Context.map_theory pspsp_timing_setup);
```

```
  pspsp_timing_config
end

structure trac_time = struct
  fun ap_thy thy msg f x = if Config.get_global thy pspsp_timing
                             then Timing.timeap_msg ("PSPSP Timing: "^msg) f x
                             else f x
  fun ap_lthy lthy = ap_thy (Proof_Context.theory_of lthy)
end
›
```

ML ‹

```
(* Some of this is based on code from the following files distributed with Isabelle 2018:
    * HOL/Tools/value_command.ML
    * HOL/Code_Evaluation.thy
    * Pure.thy
*)

fun assert_nonempty_name n =
  if n = "" then error "Error: No name given" else n

fun is_defined lthy name =
  let
    val full_name = Local_Theory.full_name lthy (Binding.name name)
    val thy = Proof_Context.theory_of lthy
  in
    Sign.const_type thy full_name <> NONE
  end

fun protocol_model_interpretation_defs name =
  let
    fun f s =
      (Binding.empty_atts:Attrib.binding, ((Binding.name s, NoSyn), name ^ "." ^ s))
  in
    (map f [
      "public", "arity", "Ana", "Γ", "Γ_v", "timpls_transformable_to", "intruder_synth_mod_timpls",
      "analyzed_closed_mod_timpls", "timpls_transformable_to'", "intruder_synth_mod_timpls'",
      "analyzed_closed_mod_timpls'", "admissible_transaction_checks",
      "admissible_transaction_updates", "admissible_transaction_terms", "admissible_transaction",
      "admissible_transaction'", "admissible_transaction_no_occurs_msgs",
      "admissible_transaction_send_occurs_form", "admissible_transaction_occurs_checks",
      "has_initial_value_producing_transaction", "add_occurs_msgs",
      "abs_substs_set", "abs_substs_fun", "in_trancl", "transaction_poschecks_comp",
      "transaction_negchecks_comp", "transaction_check_comp", "transaction_check'",
      "transaction_check", "transaction_check_pre", "transaction_check_post",
      "compute_fixpoint_fun'", "compute_fixpoint_fun", "compute_fixpoint_with_trace",
      "compute_fixpoint_from_trace", "compute_reduced_attack_trace", "attack_notin_fixpoint",
      "protocol_covered_by_fixpoint", "reduce_fixpoint'", "reduce_fixpoint", "analyzed_fixpoint",
      "wellformed_protocol''", "wellformed_protocol'", "wellformed_protocol",
      "wellformed_protocol_SMP_set", "wellformed_fixpoint", "wellformed_fixpoint'",
      "wellformed_term_implication_graph", "wellformed_composable_protocols",
      "wellformed_composable_protocols'", "composable_protocols",
      "welltyped_leakage_free_invkey_conditions'", "welltyped_leakage_free_invkey_conditions",
      "fun_point_inter", "fun_point_Inter", "fun_point_union", "fun_point_Union", "ticl_abs",
      "ticl_abss", "match_abss'", "match_abss", "synth_abs_substs_constrs_aux",
      "synth_abs_substs_constrs", "transaction_check_coverage_rcv", "protocol_covered_by_fixpoint_coverage_rcv"
    ]):string Interpretation.defines
 end

fun assert_defined lthy def =
  if is_defined lthy def then ()
  else error ("Error: The constant " ^ def ^ " is not defined.")
```

```
fun assert_not_defined lthy def =
  if not (is_defined lthy def) then ()
  else error ("Error: The constant " ^ def ^ " has already been defined.")

fun assert_all_defined lthy name defs =
  let
    fun errmsg s =
      "Error: The following constants were expected to be defined, but are not:\n" ^
      String.concatWith ", " s ^
      "\n\nProbable causes:\n" ^
      "1. The trac command failed to parse the protocol specification.\n" ^
      "2. The provided protocol-specification name (" ^ name ^ ") " ^
      "does not match the name given in the trac specification.\n" ^
      "3. Manually provided parameters (e.g., " ^ name ^ "_fixpoint, " ^ name ^ "_SMP) " ^
      "may have been misspelled.\n" ^
      "4. Any of the following commands were used before a call to the (manual_)" ^
      "protocol_model_setup command:\n" ^
      "   compute_fixpoint, compute_SMP, protocol_security_proof, manual_protocol_security_proof"
    val undefs = filter (not o is_defined lthy) defs
  in
    if null undefs then defs else error (errmsg undefs)
  end

fun protocol_model_interpretation_params name lthy =
  let
    fun f s = name ^ "_" ^ s
    val defs = [f "arity", f "sets_arity", f "public", f "Ana", f "Γ"]
    val _ = assert_all_defined lthy name defs
  in
    map SOME (defs@["0::nat", "1::nat"])
  end

fun declare_thm_attr attribute name print lthy =
  let
    val arg = [(Facts.named name, [[Token.make_string (attribute, Position.none)]])]
    val (_, lthy') = Specification.theorems_cmd "" [(Binding.empty_atts, arg)] [] print lthy
  in
    lthy'
  end

fun declare_def_attr attribute name = declare_thm_attr attribute (name ^ "_def")

val declare_code_eqn = declare_def_attr "code"

val declare_protocol_check = declare_def_attr "protocol_checks"

fun declare_protocol_checks print =
  declare_protocol_check "attack_notin_fixpoint" print #>
  declare_protocol_check "protocol_covered_by_fixpoint" print #>
  declare_protocol_check "protocol_covered_by_fixpoint_coverage_rcv" print #>
  declare_protocol_check "analyzed_fixpoint" print #>
  declare_protocol_check "has_initial_value_producing_transaction" print #>
  declare_protocol_check "wellformed_protocol''" print #>
  declare_protocol_check "wellformed_protocol'" print #>
  declare_protocol_check "wellformed_protocol" print #>
  declare_protocol_check "wellformed_fixpoint'" print #>
  declare_protocol_check "wellformed_fixpoint" print #>
  declare_protocol_check "compute_fixpoint_fun" print

fun eval_term lthy t =
  Code_Evaluation.dynamic_value_strict lthy t
```

```
fun eval_define_declare (name, t) print lthy =
  let
    val t' = eval_term lthy t
    val arg = ((Binding.name name, NoSyn), ((Binding.name (name ^ "_def"),@{attributes [code]}), t'))
    val lthy' =  snd ( Local_Theory.begin_nested lthy )
    val (_, lthy'') = Local_Theory.define arg lthy'
  in
    (t', Local_Theory.end_nested lthy'')
  end

fun eval_define_declare_nbe (name, t) print lthy =
  let
    val t' = Nbe.dynamic_value lthy t
    val arg = ((Binding.name name, NoSyn), ((Binding.name (name ^ "_def"),@{attributes [code]}), t'))
    val lthy' =  snd ( Local_Theory.begin_nested lthy )
    val (_, lthy'') = Local_Theory.define arg lthy'
  in
    (t', Local_Theory.end_nested lthy'')
  end
›
```

ML‹

```
structure ml_isar_wrapper = struct
  fun define_constant_definition' (constname, trm) print lthy =
    let
      val lthy' =  snd ( Local_Theory.begin_nested lthy )
      val arg = ((Binding.name constname, NoSyn), ((Binding.name (constname^"_def"),@{attributes
[code]}), trm))
      val ((_, (_ , thm)), lthy'') = Local_Theory.define arg lthy'
    in
      (thm, Local_Theory.end_nested lthy'')
    end

  fun define_simple_abbrev (constname, trm) lthy =
    let
      val arg = ((Binding.name constname, NoSyn), trm)
      val ((_, _), lthy') = Local_Theory.abbrev Syntax.mode_default arg lthy
    in
      lthy'
    end

  fun define_simple_type_synonym (name, typedecl) lthy =
    let
      val (_, lthy') = Typedecl.abbrev_global (Binding.name name, [], NoSyn) typedecl lthy
    in
      lthy'
    end

  fun define_simple_datatype (dt_tyargs, dt_name) constructors =
    let
      val options = Plugin_Name.default_filter
      fun lift_c (tyargs, name) =  (((Binding.empty, Binding.name name), map (fn t => (Binding.empty,
t)) tyargs), NoSyn)
      val c_spec = map lift_c constructors
      val datatyp = ((map (fn ty => (NONE, ty)) dt_tyargs, Binding.name dt_name), NoSyn)
      val dtspec =
        ((options,false),
         [(((datatyp, c_spec), (Binding.empty, Binding.empty, Binding.empty)), [])])
    in
      BNF_FP_Def_Sugar.co_datatypes BNF_Util.Least_FP BNF_LFP.construct_lfp dtspec
    end

   fun define_simple_primrec pname precs lthy =
```

```
     let
       val rec_eqs = map (fn (lhs,rhs) => (((Binding.empty,[]), HOLogic.mk_Trueprop (HOLogic.mk_eq
(lhs,rhs))),[],[])) precs
     in
       snd (BNF_LFP_Rec_Sugar.primrec false [] [(Binding.name pname, NONE, NoSyn)] rec_eqs lthy)
     end

   fun define_simple_fun pname precs lthy =
     let
       val rec_eqs = map (fn (lhs,rhs) => (((Binding.empty,[]), HOLogic.mk_Trueprop (HOLogic.mk_eq
(lhs,rhs))),[],[])) precs
     in
       Function_Fun.add_fun [(Binding.name pname, NONE, NoSyn)] rec_eqs  Function_Common.default_config
lthy
     end

   fun prove_simple name stmt tactic lthy =
     let
       val thm = Goal.prove_future lthy [] [] stmt (fn {context, ...} => tactic context)
                  |> Goal.norm_result lthy
                  |> Goal.check_finished lthy
     in
       lthy |>
       snd o  Local_Theory.note ((Binding.name name, []), [thm])
     end

    fun prove_state_simple method proof_state =
           (Proof.refine_singleton method proof_state)
               |> Proof.global_done_proof

end
›
```

ML‹

```
structure trac_definitorial_package = struct
  type hide_tvar_tab = (TracProtocol.protocol) Symtab.table
  fun trac_eq (a, a') = (#name a) = (#name a')
  fun merge_trac_tab (tab,tab') = Symtab.merge trac_eq (tab,tab')
  structure Data = Generic_Data
  (
    type T = hide_tvar_tab
    val empty  = Symtab.empty:hide_tvar_tab
    val extend = I
    fun merge(t1,t2)  = merge_trac_tab (t1, t2)
  );

  fun update  p thy = Context.theory_of
                      ((Data.map (fn tab => Symtab.update (#name p, p) tab) (Context.Theory thy)))
  fun lookup name thy = (Symtab.lookup ((Data.get o Context.Theory) thy) name,thy)

  fun lookup_trac (pname:string) lthy =
    Option.valOf (fst (lookup pname (Proof_Context.theory_of lthy)))

  (* constant names *)
  open Trac_Utils
  val enum_constsN="enum_consts"
  val setsN="sets"
  val funN="fun"
  val atomN="atom"
  val arityN="arity"
  val set_arityN=setsN^"_"^arityN
  val publicN = "public"
```

```
val gammaN = "Γ"
val anaN = "Ana"

fun mk_listT T =  Type ("List.list", [T])
val mk_setT = HOLogic.mk_setT
val boolT = HOLogic.boolT
val natT = HOLogic.natT
val mk_tupleT =  HOLogic.mk_tupleT
val mk_prodT = HOLogic.mk_prodT

val mk_set = HOLogic.mk_set
val mk_list = HOLogic.mk_list
val mk_nat = HOLogic.mk_nat
val mk_eq = HOLogic.mk_eq
val mk_Trueprop = HOLogic.mk_Trueprop
val mk_tuple = HOLogic.mk_tuple
val mk_prod = HOLogic.mk_prod

fun mkN (a,b) = a^"_"^b

val info = Output.information

fun full_name name lthy =
  Local_Theory.full_name lthy (Binding.name name)

fun full_name' n (trac:TracProtocolCert.cProtocol) lthy = full_name (mkN (#name trac, n)) lthy

fun mk_prot_type name targs (trac:TracProtocolCert.cProtocol) lthy =
  Term.Type (full_name' name trac lthy, targs)

val enum_constsT = mk_prot_type enum_constsN []

fun mk_enum_const a trac lthy =
  Term.Const (full_name' enum_constsN trac lthy ^ "." ^ a, enum_constsT trac lthy)

val setexprT = mk_prot_type setsN []

val funT = mk_prot_type funN []

val atomT = mk_prot_type atomN []

fun messageT (trac:TracProtocolCert.cProtocol) lthy =
  Term.Type ("Transactions.prot_term", [funT trac lthy, atomT trac lthy, setexprT trac lthy, natT])

fun message_funT (trac:TracProtocolCert.cProtocol) lthy =
  Term.Type ("Transactions.prot_fun", [funT trac lthy, atomT trac lthy, setexprT trac lthy, natT])

fun message_varT (trac:TracProtocolCert.cProtocol) lthy =
  Term.Type ("Transactions.prot_var", [funT trac lthy, atomT trac lthy, setexprT trac lthy, natT])

fun message_term_typeT (trac:TracProtocolCert.cProtocol) lthy =
  Term.Type ("Transactions.prot_term_type",
             [funT trac lthy, atomT trac lthy, setexprT trac lthy, natT])

fun message_term_type_listT (trac:TracProtocolCert.cProtocol) lthy =
  mk_listT (message_term_typeT trac lthy)

fun message_atomT (trac:TracProtocolCert.cProtocol) lthy =
  Term.Type ("Transactions.prot_atom", [atomT trac lthy])

fun messageT' varT (trac:TracProtocolCert.cProtocol) lthy =
  Term.Type ("Term.term", [message_funT trac lthy, varT])
```

```
fun message_listT (trac:TracProtocolCert.cProtocol) lthy =
  mk_listT (messageT trac lthy)

fun message_listT' varT (trac:TracProtocolCert.cProtocol) lthy =
  mk_listT (messageT' varT trac lthy)

fun absT (trac:TracProtocolCert.cProtocol) lthy =
  mk_setT (setexprT trac lthy)

fun abssT (trac:TracProtocolCert.cProtocol) lthy =
  mk_setT (absT trac lthy)

val poscheckvariantT =
  Term.Type ("Strands_and_Constraints.poscheckvariant", [])

val strand_labelT =
  Term.Type ("Labeled_Strands.strand_label", [natT])

fun strand_stepT (trac:TracProtocolCert.cProtocol) lthy =
  Term.Type ("Stateful_Strands.stateful_strand_step",
             [message_funT trac lthy, message_varT trac lthy])

fun labeled_strand_stepT (trac:TracProtocolCert.cProtocol) lthy =
  mk_prodT (strand_labelT, strand_stepT trac lthy)

fun prot_strandT (trac:TracProtocolCert.cProtocol) lthy =
  mk_listT (labeled_strand_stepT trac lthy)

fun prot_transactionT (trac:TracProtocolCert.cProtocol) lthy =
  Term.Type ("Transactions.prot_transaction",
             [funT trac lthy, atomT trac lthy, setexprT trac lthy, natT])

val mk_star_label =
  Term.Const ("Labeled_Strands.strand_label.LabelS", strand_labelT)

fun mk_prot_label (lbl:int) =
  Term.Const ("Labeled_Strands.strand_label.LabelN", natT --> strand_labelT) $
    mk_nat lbl

fun mk_labeled_step (label:term) (step:term) =
  mk_prod (label, step)

fun mk_Send_step (trac:TracProtocolCert.cProtocol) lthy (label:term) (msgs:term list) =
  mk_labeled_step label
    (Term.Const ("Stateful_Strands.stateful_strand_step.Send",
                 mk_listT (messageT trac lthy) --> strand_stepT trac lthy) $
                  mk_list (messageT trac lthy) msgs)

fun mk_Receive_step (trac:TracProtocolCert.cProtocol) lthy (label:term) (msgs:term list) =
  mk_labeled_step label
    (Term.Const ("Stateful_Strands.stateful_strand_step.Receive",
                 mk_listT (messageT trac lthy) --> strand_stepT trac lthy) $
                  mk_list (messageT trac lthy) msgs)

fun mk_InSet_step (trac:TracProtocolCert.cProtocol) lthy psv (label:term) (elem:term) (set:term) =
  let
    val psT = [poscheckvariantT, messageT trac lthy, messageT trac lthy]
    val psvN =
      case psv of TracProtocolCert.cCheck => "Check" | TracProtocolCert.cAssignment => "Assign"
  in
    mk_labeled_step label
      (Term.Const ("Stateful_Strands.stateful_strand_step.InSet",
                   psT ---> strand_stepT trac lthy) $
```

```
      Term.Const ("Strands_and_Constraints.poscheckvariant." ^ psvN, poscheckvariantT) $
      elem $ set)
  end

fun mk_NegChecks_step (trac:TracProtocolCert.cProtocol) lthy (label:term)
                      (bvars:term list) (ineqs:(term*term) list) (notins:(term*term) list) =
  let
    val msgT = messageT trac lthy
    val varT = message_varT trac lthy
    val trm_prodT = mk_prodT (messageT trac lthy, messageT trac lthy)
    val psT = [mk_listT varT, mk_listT trm_prodT, mk_listT trm_prodT]
  in
    mk_labeled_step label
      (Term.Const ("Stateful_Strands.stateful_strand_step.NegChecks",
                   psT ---> strand_stepT trac lthy) $
       (case bvars of
          [] => mk_list varT []
        | [x] => mk_list varT [Term.Const (@{const_name "the_Var"}, msgT --> varT) $ x]
        | xs =>
            Term.Const (@{const_name "map"}, [msgT --> varT, mk_listT msgT] ---> mk_listT varT) $
            Term.Const (@{const_name "the_Var"}, msgT --> varT) $
            mk_list msgT xs) $
       mk_list trm_prodT (map mk_prod ineqs) $
       mk_list trm_prodT (map mk_prod notins))
  end

fun mk_NotInSet_step (trac:TracProtocolCert.cProtocol) lthy (label:term) (elem:term) (set:term) =
  mk_NegChecks_step trac lthy label [] [] [(elem,set)]

fun mk_Equality_step (trac:TracProtocolCert.cProtocol) lthy psv (label:term) (t1:term) (t2:term) =
  let
    val psT = [poscheckvariantT, messageT trac lthy, messageT trac lthy]
    val psvN =
      case psv of TracProtocolCert.cCheck => "Check" | TracProtocolCert.cAssignment => "Assign"
  in
    mk_labeled_step label
      (Term.Const ("Stateful_Strands.stateful_strand_step.Equality",
                   psT ---> strand_stepT trac lthy) $
       Term.Const ("Strands_and_Constraints.poscheckvariant." ^ psvN, poscheckvariantT) $ t1 $ t2)
  end

fun mk_Insert_step (trac:TracProtocolCert.cProtocol) lthy (label:term) (elem:term) (set:term) =
  mk_labeled_step label
    (Term.Const ("Stateful_Strands.stateful_strand_step.Insert",
                 [messageT trac lthy, messageT trac lthy] ---> strand_stepT trac lthy) $
     elem $ set)

fun mk_Delete_step (trac:TracProtocolCert.cProtocol) lthy (label:term) (elem:term) (set:term) =
  mk_labeled_step label
    (Term.Const ("Stateful_Strands.stateful_strand_step.Delete",
                 [messageT trac lthy, messageT trac lthy] ---> strand_stepT trac lthy) $
     elem $ set)

fun mk_Transaction (trac:TracProtocolCert.cProtocol) lthy S0 S1 S2 S3 S4 S5 S6 =
  let
    val varT = message_varT trac lthy
    val msgT = messageT trac lthy
    val var_listT = mk_listT varT
    val msg_listT = mk_listT msgT
    val fun_setT = mk_setT (funT trac lthy)
    val trT = prot_transactionT trac lthy
    val declT = mk_prodT (varT, fun_setT)
    val decl_listT = mk_listT declT
```

```
    val decl_list_funT = HOLogic.unitT --> decl_listT
    val stepT = labeled_strand_stepT trac lthy
    val strandT = prot_strandT trac lthy
    val strandsT = mk_listT strandT
    val paramsT = [decl_list_funT, var_listT, strandT, strandT, strandT, strandT]
  in
    Term.Const ("Transactions.prot_transaction.Transaction", paramsT ---> trT) $
    (Term.Const ("Product_Type.unit.case_unit", decl_listT --> decl_list_funT) $
     mk_list declT S0) $
    (if null S4 then mk_list varT []
     else Term.Const (@{const_name "map"}, [msgT --> varT, msg_listT] ---> var_listT) $
          Term.Const (@{const_name "the_Var"}, msgT --> varT) $
          mk_list msgT S4) $
    mk_list stepT S1 $
    (if null S3 then mk_list stepT S2
     else Term.Const (@{const_name "append"}, [strandT,strandT] ---> strandT) $
          mk_list stepT S2 $
          (Term.Const (@{const_name "concat"}, strandsT --> strandT) $ mk_list strandT S3)) $
    mk_list stepT S5 $
    mk_list stepT S6
  end

fun get_funs (trac:TracProtocolCert.cProtocol) =
    case #function_spec trac of
      NONE => ([],[],[])
    | SOME ({public_funs=pub_funs, public_consts=pub_consts, private_consts=priv_consts}) =>
        (pub_funs, pub_consts, priv_consts)

(* TODO: consider differentiating between "/" sets and "//" sets *)
fun get_set_spec (trac:TracProtocolCert.cProtocol) =
  distinct (op =) (map (fn (s,n,_) => (s,n)) (#set_spec trac))

fun get_general_set_family_set_spec (trac:TracProtocolCert.cProtocol) =
  distinct (op =) (map_filter (fn (s,n,b) => if b then SOME (s,n) else NONE) (#set_spec trac))

fun is_general_set_family (trac:TracProtocolCert.cProtocol) s =
  List.exists (fn (s',_) => s = s') (get_general_set_family_set_spec trac)

fun set_arity (trac:TracProtocolCert.cProtocol) s =
  case List.find (fn x => fst x = s) (get_set_spec trac) of
    SOME (_,n) => SOME n
  | NONE => NONE

fun get_enum_consts (trac:TracProtocolCert.cProtocol) =
  distinct (op =) (List.concat (map #2 (#enum_spec trac)))

fun get_finite_enum_spec (trac:TracProtocolCert.cProtocol) =
  filter (null o #3) (#enum_spec trac)

fun get_infinite_enum_spec (trac:TracProtocolCert.cProtocol) =
  filter_out (null o #3) (#enum_spec trac)

fun get_nonunion_infinite_enum_spec (trac:TracProtocolCert.cProtocol) =
  filter (fn (e,cs,ies) => null cs andalso ies = [e])
        (get_infinite_enum_spec trac)

fun get_typed_constants_in_function_spec (trac:TracProtocolCert.cProtocol) =
  case #function_spec trac of
    SOME {private_consts=priv, public_consts=pub, ...} =>
      map_filter (fn (c,t) => Option.map (fn a => (c,a)) t) (priv@pub)
  | NONE => []

fun get_user_atom_spec_pre (trac:TracProtocolCert.cProtocol) =
```

```
  map (fn s => (s,([boolT,natT],s^"_constant"))) (#type_spec trac)

fun get_user_atom_spec (trac:TracProtocolCert.cProtocol) =
  map (fn (c,a) => (a,([],c))) (get_typed_constants_in_function_spec trac)@
  get_user_atom_spec_pre trac

fun is_attack_transaction (tr:TracProtocolCert.cTransaction) =
  not (null (#attack_actions tr))

fun get_transaction_name (tr:TracProtocolCert.cTransaction) =
  #1 (#transaction tr)

fun get_transaction_head_variables (tr:TracProtocolCert.cTransaction) =
  #2 (#transaction tr)

fun get_bound_variables (tr:TracProtocolCert.cTransaction) =
  let
    val a = map_filter (TracProtocolCert.maybe_the_NegChecks o snd) (#checksingle_actions tr)
  in
    distinct (op =) (List.concat (map fst a))
  end

fun get_fresh_variables (tr:TracProtocolCert.cTransaction) =
  List.concat (map_filter (TracProtocolCert.maybe_the_Fresh o snd) (#fresh_actions tr))

fun get_fresh_value_variables (tr:TracProtocolCert.cTransaction) =
  map_filter (fn (x,tau) => case tau of Trac_Term.ValueType => SOME x | _ => NONE)
           (get_fresh_variables tr)

fun get_nonfresh_value_variables (tr:TracProtocolCert.cTransaction) =
  map fst (filter (fn x => snd x = Trac_Term.ValueType) (get_transaction_head_variables tr))

fun get_value_variables (tr:TracProtocolCert.cTransaction) =
  get_nonfresh_value_variables tr@get_fresh_value_variables tr

fun get_finite_enum_variables (tr:TracProtocolCert.cTransaction) =
  distinct (op =) (filter (fn (_,tau) => case tau of
                                          Trac_Term.Enumeration _ => true
                                        | _ => false)
                         (get_transaction_head_variables tr))

fun get_infinite_enum_variables (tr:TracProtocolCert.cTransaction) =
  distinct (op =) (filter (fn (_,tau) => case tau of
                                          Trac_Term.InfiniteEnumeration _ => true
                                        | _ => false)
                         (get_transaction_head_variables tr))


fun get_enumtype_variables (tr:TracProtocolCert.cTransaction) =
  distinct (op =) (filter (fn (_,tau) => tau = Trac_Term.EnumType)
                         (get_transaction_head_variables tr))

fun get_nonenum_variables (tr:TracProtocolCert.cTransaction) =
  map_filter (fn (x,tau) => case tau of
                Trac_Term.Enumeration _ => NONE
              | Trac_Term.InfiniteEnumeration _ => NONE
              | _ => SOME (x,tau))
           (get_transaction_head_variables tr@get_fresh_variables tr)

fun get_variable_restrictions (tr:TracProtocolCert.cTransaction) =
  let
    val enum_vars = get_finite_enum_variables tr
    val value_vars = get_value_variables tr
```

```
    fun enum_member x = List.exists (fn y => x = fst y)
    fun value_member x = List.exists (fn y => x = y)
    fun aux [] = ([],[])
      | aux ((a,b)::rs) =
          if enum_member a enum_vars andalso enum_member b enum_vars
          then let val (es,vs) = aux rs in ((a,b)::es,vs) end
          else if value_member a value_vars andalso value_member b value_vars
          then let val (es,vs) = aux rs in (es,(a,b)::vs) end
          else error ("Error: Ill-formed or ill-typed variable restriction: " ^ a ^ " != " ^ b)
  in
    aux (#3 (#transaction tr))
  end

fun setexpr_to_hol (db:string * Trac_Term.cMsg list) (trac:TracProtocolCert.cProtocol) lthy =
  let
    open Trac_Term
    fun mkN' n = mkN (#name trac, n)
    val s_prefix = full_name (mkN' setsN) lthy ^ "."
    val e_prefix = full_name (mkN' enum_constsN) lthy ^ "."
    val (s,es) = db
    val tau = enum_constsT trac lthy
    val setexprT = setexprT trac lthy
    val a = Term.Const (s_prefix ^ s, map (fn _ => tau) es ---> setexprT)
    fun param_to_hol (cVar (x,Enumeration _)) = Term.Free (x, tau)
      | param_to_hol (cVar (x,EnumType)) = Term.Free (x, tau)
      | param_to_hol (cEnum e) = Term.Const (e_prefix ^ e, tau)
      | param_to_hol t = error ("Error: Invalid set parameter: " ^ cMsg_str t)
  in
    fold (fn e => fn b => b $ param_to_hol e) es a
  end

fun abs_to_hol (bs:(string * string list) list) (trac:TracProtocolCert.cProtocol) lthy =
  mk_set (setexprT trac lthy)
         (map (fn (s,cs) => setexpr_to_hol (s, map Trac_Term.cEnum cs) trac lthy) bs)

fun cType_to_hol (t:Trac_Term.cType) trac lthy =
  let
    open Trac_Term
    val atomT = atomT trac lthy
    val prot_atomT = message_atomT trac lthy
    val tT = message_term_typeT trac lthy
    val fT = message_funT trac lthy
    val tsT = message_term_type_listT trac lthy
    val TAtomT = prot_atomT --> tT
    val TCompT = [fT, tsT] ---> tT
    val funT = funT trac lthy
    val setexprT = setexprT trac lthy
    val SetT = setexprT --> fT
    val FuT = funT --> fT
    val TAtomC = Term.Const (@{const_name "Var"}, TAtomT)
    val TCompC = Term.Const (@{const_name "Fun"}, TCompT)
    val AtomC = Term.Const ("Transactions.prot_atom.Atom", atomT --> prot_atomT)
    fun full_name'' n = full_name' n trac lthy
    fun mk_prot_fun_trm f tau = Term.Const ("Transactions.prot_fun." ^ f, tau)
    fun mk_Fu_trm f =
          mk_prot_fun_trm "Fu" FuT $ Term.Const (full_name'' funN ^ "." ^ f, funT)
    fun mk_Set_trm (s,ts) = (* TODO: use? *)
          mk_prot_fun_trm "Set" SetT $ setexpr_to_hol (s,ts) trac lthy
    fun c_to_h s = cType_to_hol s trac lthy
    fun c_list_to_h ts = mk_list tT (map c_to_h ts)

    fun mk_atom_trm n = Term.Const (full_name'' atomN ^ "." ^ n, atomT)
    val EnumType_trm = TAtomC $ (AtomC $ mk_atom_trm enum_typeN)
```

```
    val ValueType_trm = TAtomC $ Term.Const ("Transactions.prot_atom.Value", prot_atomT)
  in
    case t of
      Enumeration _ => EnumType_trm
    | InfiniteEnumeration _ => EnumType_trm
    | EnumType => EnumType_trm
    | ValueType => ValueType_trm
    | PrivFunSecType => TAtomC $ (AtomC $ mk_atom_trm secret_typeN)
    | AtomicType s => TAtomC $ (AtomC $ mk_atom_trm s)
    | ComposedType (f,ts) => TCompC $ mk_Fu_trm f $ c_list_to_h ts
    | Untyped => error "Error: Expected a type but got untyped"
  end

fun cMsg_to_hol (t:Trac_Term.cMsg) lbl varT var_map free_enum_var free_message_var trac lthy =
  let
    open Trac_Term
    val tT = messageT' varT trac lthy
    val fT = message_funT trac lthy
    val enum_constsT = enum_constsT trac lthy
    val tsT = message_listT' varT trac lthy
    val VarT = varT --> tT
    val FunT = [fT, tsT] ---> tT
    val absT = absT trac lthy
    val setexprT = setexprT trac lthy
    val AbsT = absT --> fT
    val funT = funT trac lthy
    val FuT = funT --> fT
    val SetT = setexprT --> fT
    val enumT = enum_constsT --> funT
    val VarC = Term.Const (@{const_name "Var"}, VarT)
    val FunC = Term.Const (@{const_name "Fun"}, FunT)
    val NilC = Term.Const (@{const_name "Nil"}, tsT)
    val prot_label = mk_prot_label lbl
    fun full_name'' n = full_name' n trac lthy
    fun mk_enum_const' a = mk_enum_const a trac lthy
    fun mk_prot_fun_trm f tau = Term.Const ("Transactions.prot_fun." ^ f, tau)
    fun mk_enum_trm etrm =
      mk_prot_fun_trm "Fu" FuT $ (Term.Const (full_name'' funN ^ "." ^ enumN, enumT) $ etrm)
    fun mk_Fu_trm f =
      mk_prot_fun_trm "Fu" FuT $ Term.Const (full_name'' funN ^ "." ^ f, funT)
    fun c_to_h s = cMsg_to_hol s lbl varT var_map free_enum_var free_message_var trac lthy
    fun c_list_to_h ts = mk_list tT (map c_to_h ts)
  in
    case t of
      cVar x =>
        if free_enum_var x
        then FunC $ mk_enum_trm (Term.Free (fst x, enum_constsT)) $ NilC
        else if free_message_var x
        then (* Term.Free (fst x, tT) *) (* TODO: somehow Isabelle doesn't realize that tT is the
                                                  same as messageT when varT is the right type
                                                  - maybe it's the type synonym in messageT which
                                                  is to blame *)
             Term.Free (fst x, messageT trac lthy)
        else VarC $ var_map x
    | cConst f =>
        FunC $
        mk_Fu_trm f $
        NilC
    | cFun (f,ts) =>
        FunC $
        mk_Fu_trm f $
        c_list_to_h ts
    | cSet (s,ts) =>
```

```
        if is_general_set_family trac s
        then FunC $
             (mk_prot_fun_trm "Set" SetT $ setexpr_to_hol (s,[]) trac lthy) $
             mk_list tT (map c_to_h (cPrivFunSec::ts))
        else FunC $
             (mk_prot_fun_trm "Set" SetT $ setexpr_to_hol (s,ts) trac lthy) $
             NilC
    | cAttack =>
        FunC $
        (mk_prot_fun_trm "Attack" (strand_labelT --> fT) $ prot_label) $
        NilC
    | cAbs bs =>
        FunC $
        (mk_prot_fun_trm "Abs" AbsT $ abs_to_hol bs trac lthy) $
        NilC
    | cOccursFact bs =>
        FunC $
        mk_prot_fun_trm "OccursFact" fT $
        mk_list tT [
          FunC $ mk_prot_fun_trm "OccursSec" fT $ NilC,
          c_to_h bs]
    | cPrivFunSec =>
        FunC $
        mk_Fu_trm priv_fun_secN $
        NilC
    | cEnum a =>
        FunC $
        mk_enum_trm (mk_enum_const' a) $
        NilC
end

fun ground_cMsg_to_hol t lbl trac lthy =
  cMsg_to_hol t lbl (message_varT trac lthy) (fn _ => error "Error: Term not ground")
              (fn _ => false) (fn _ => false) trac lthy

fun ana_cMsg_to_hol inc_vars t (ana_var_map:string list) =
  let
    open Trac_Term
    fun var_map (x,Untyped) = (
          case list_find (fn y => x = y) ana_var_map of
            SOME (_,n) => if inc_vars then mk_nat (1+n) else mk_nat n
          | NONE => error ("Error: Analysis variable " ^ x ^ " not found"))
      | var_map _ = error "Error: Analysis variables must be untyped"
    val lbl = 0 (* There's no constants in analysis messages requiring labels anyway *)
  in
    cMsg_to_hol t lbl natT var_map (fn _ => false) (fn _ => false)
  end

fun transaction_cMsg_to_hol t lbl transaction_var_map free_vars trac lthy =
  let
    open Trac_Term
    val varT = message_varT trac lthy
    fun var_map (x,tau) =
          case list_find (fn y => (x,tau) = y) transaction_var_map of
            SOME (_,n) => HOLogic.mk_prod (cType_to_hol tau trac lthy, mk_nat n)
          | NONE => error ("Error: Transaction variable " ^ cMsg_str (cVar (x,tau)) ^ " not found")
    fun free_enum_var (_,Enumeration _) = true
      | free_enum_var _ = false
  in
    cMsg_to_hol t lbl varT var_map free_enum_var (fn _ => free_vars) trac lthy
  end

fun fp_triple_to_hol (fp,occ,ti) trac lthy =
```

```
  let
    val prot_label = 0
    val tau_abs = absT trac lthy
    val tau_fp_elem = messageT trac lthy
    val tau_occ_elem = tau_abs
    val tau_ti_elem = mk_prodT (tau_abs, tau_abs)
    fun a_to_h bs = abs_to_hol bs trac lthy
    fun c_to_h t = ground_cMsg_to_hol t prot_label trac lthy
    val fp' = mk_list tau_fp_elem (map c_to_h fp)
    val occ' = mk_list tau_occ_elem (map a_to_h occ)
    val ti' = mk_list tau_ti_elem (map (mk_prod o map_prod a_to_h) ti)
  in
    mk_tuple [fp', occ', ti']
  end

fun absfreeprod tau xs trm =
  let
    val tau_out = Term.fastype_of trm
    fun absfree' x = absfree (x,tau)
    fun aux _ [] = trm
      | aux _ [x] = absfree' x trm
      | aux len (x::y::xs) =
          Term.Const (@{const_name "case_prod"},
                 [[tau,mk_tupleT (replicate (len-1) tau)] ---> tau_out,
                  mk_tupleT (replicate len tau)] ---> tau_out) $
          absfree' x (aux (len-1) (y::xs))
  in
    aux (length xs) xs
  end

fun abstract_over_finite_enum_vars enum_vars enum_ineqs trm trac lthy =
  let
    val enum_constsT = enum_constsT trac lthy
    val absfreeprod' = absfreeprod enum_constsT

    fun enumlistelemT n = mk_tupleT (replicate n enum_constsT)
    fun enumlistT n = mk_listT (enumlistelemT n)
    fun mk_enum_const' a = mk_enum_const a trac lthy

    fun mk_enumlist ns = mk_list enum_constsT (map mk_enum_const' ns)

    fun mk_enum_neq (a,b) = (HOLogic.mk_not o HOLogic.mk_eq)
      (Term.Free (a, enum_constsT), Term.Free (b, enum_constsT))

    fun mk_enum_neqs_list [] = Term.Const (@{const_name "True"}, HOLogic.boolT)
      | mk_enum_neqs_list [x] = mk_enum_neq x
      | mk_enum_neqs_list (x::y::xs) = HOLogic.mk_conj (mk_enum_neq x, mk_enum_neqs_list (y::xs))

    val enum_types =
      let
        open Trac_Term

        val flat_enum_spec = map (fn (a,b,_) => (a,b)) (get_finite_enum_spec trac)
        val err_pre = "Error: Expected a finite enumeration, but got "
        fun aux (Enumeration t) = (
              case List.find (fn (s,_) => t = s) flat_enum_spec of
                SOME (_,cs) => (t,cs)
              | NONE => error ("Error: " ^ t ^ " has not been declared as an enumeration"))
          | aux Untyped = (enum_constsN,get_enum_consts trac)
          | aux (InfiniteEnumeration t) = error (err_pre ^ "an infinite enumeration: " ^ t)
          | aux tau = error (err_pre ^ "type " ^ cType_str tau)
      in
        map (aux o snd) enum_vars
```

```
      end

    fun enumlist_product f nil_case =
      let
        fun aux _ [] = nil_case ()
          | aux _ [ns] = f ns
          | aux len (ns::ms::elists) =
              Term.Const ("List.product", [enumlistT 1, enumlistT (len-1)] ---> enumlistT len) $
              f ns $ aux (len-1) (ms::elists)
      in
        aux (length enum_types) enum_types
      end

    val enable_let_bindings = false

    val absfp = absfreeprod' (map fst enum_vars) trm
    val eptrm = if length enum_vars > 1 andalso enable_let_bindings
                then enumlist_product
                       (fn (x,_) => Term.Free (x, mk_listT enum_constsT))
                       (fn () => error "Error: Nil in enumlist_product")
                else enumlist_product (mk_enumlist o snd) (fn () => mk_enumlist [])
    val typof = Term.fastype_of
    val evseT = enumlistelemT (length enum_vars)
    val evslT = enumlistT (length enum_vars)
    val eneqs = absfreeprod' (map fst enum_vars) (mk_enum_neqs_list enum_ineqs)
  in
    if null enum_vars
    then mk_list (typof trm) [trm]
    else let
      val a = Term.Const(@{const_name "map"},
                [typof absfp, typof eptrm] ---> mk_listT (typof trm)) $
              absfp
      val b = if null enum_ineqs
              then eptrm
              else Term.Const (@{const_name "filter"},
                     [evseT --> HOLogic.boolT, evslT] ---> evslT) $
                   eneqs $ eptrm
      val c = absfreeprod (mk_listT enum_constsT) (distinct (op =) (map fst enum_types)) (a$b)
      val d = mk_tuple (map mk_enumlist (distinct (op =) (map snd enum_types)))
      val e = Term.Const (@{const_name "Let"}, [typof d, typof c] ---> typof (c$d))$d$c
    in if length enum_vars > 1 andalso enable_let_bindings then e else a $ b
    end
  end

fun mk_type_of_name lthy pname name ty_args
    = Type(Local_Theory.full_name lthy (Binding.name (mkN(pname, name))), ty_args)

fun mk_mt_list t = Term.Const (@{const_name "Nil"}, mk_listT t)

fun name_of_typ (Type (s, _)) = s
  | name_of_typ (TFree _)     = error "name_of_type: unexpected TFree"
  | name_of_typ (TVar _ )     = error "name_of_type: unexpected TVAR"

fun prove_UNIV name typ elems thmsN lthy =
  let
    val rhs = mk_set typ elems
    val lhs = Const("Set.UNIV",mk_setT typ)
    val stmt = mk_Trueprop (mk_eq (lhs,rhs))
    val fq_tname = name_of_typ typ

    fun inst_and_prove_enum thy =
      let
        val _ = writeln("Inst enum: "^name)
```

```
        val lthy = Class.instantiation ([fq_tname], [], @{sort enum}) thy
        val enum_eq = Const("Pure.eq",mk_listT typ --> mk_listT typ --> propT)
                          $Const(@{const_name "enum_class.enum"},mk_listT typ)
                          $(mk_list typ elems)

        val ((_, (_, enum_def')), lthy) = Specification.definition NONE [] []
                                             ((Binding.name ("enum_"^name),[]), enum_eq) lthy
        val ctxt_thy = Proof_Context.init_global (Proof_Context.theory_of lthy)
        val enum_def = singleton (Proof_Context.export lthy ctxt_thy) enum_def'

        val enum_all_eq = Const("Pure.eq", boolT --> boolT --> propT)
                          $(Const(@{const_name "enum_class.enum_all"},(typ --> boolT) --> boolT)
                                            $Free("P",typ --> boolT))
                          $(Const(@{const_name "list_all"},(typ --> boolT) --> (mk_listT typ) -->
boolT)
                                 $Free("P",typ --> boolT)$(mk_list typ elems))
        val ((_, (_, enum_all_def')), lthy) = Specification.definition NONE [] []
                                             ((Binding.name ("enum_all_"^name),[]), enum_all_eq)
lthy
        val ctxt_thy = Proof_Context.init_global (Proof_Context.theory_of lthy)
        val enum_all_def = singleton (Proof_Context.export lthy ctxt_thy) enum_all_def'

        val enum_ex_eq = Const("Pure.eq", boolT --> boolT --> propT)
                          $(Const(@{const_name "enum_class.enum_ex"},(typ --> boolT) --> boolT)
                                            $Free("P",typ --> boolT))
                          $(Const(@{const_name "list_ex"},(typ --> boolT) --> (mk_listT typ) -->
boolT)
                                 $Free("P",typ --> boolT)$(mk_list typ elems))
        val ((_, (_, enum_ex_def')), lthy) = Specification.definition NONE [] []
                                             ((Binding.name ("enum_ex_"^name),[]), enum_ex_eq) lthy
        val ctxt_thy = Proof_Context.init_global (Proof_Context.theory_of lthy)
        val enum_ex_def = singleton (Proof_Context.export lthy ctxt_thy) enum_ex_def'
      in
        Class.prove_instantiation_exit (fn ctxt =>
          (Class.intro_classes_tac ctxt [])  THEN
             PARALLEL_ALLGOALS (simp_tac (ctxt addsimps  [Proof_Context.get_thm ctxt (name^"_UNIV"),

                                                          enum_def, enum_all_def, enum_ex_def]) )
          )lthy
      end
    fun inst_and_prove_finite thy =
      let
        val lthy = Class.instantiation ([fq_tname], [], @{sort finite}) thy
      in
        Class.prove_instantiation_exit (fn ctxt =>
          (Class.intro_classes_tac ctxt []) THEN
           (simp_tac (ctxt addsimps[Proof_Context.get_thm ctxt (name^"_UNIV")])) 1) lthy
      end
  in
    lthy
    |> ml_isar_wrapper.prove_simple (name^"_UNIV") stmt
       (fn c =>     (safe_tac c)
               THEN (PARALLEL_ALLGOALS(simp_tac c))
               THEN (PARALLEL_ALLGOALS(Metis_Tactic.metis_tac ["full_types"]
                                  "combs"  c
                                  (map (Proof_Context.get_thm c) thmsN)))
       )
    |> Local_Theory.raw_theory inst_and_prove_finite
    |> Local_Theory.raw_theory inst_and_prove_enum
  end

 fun def_enum_consts (trac:TracProtocolCert.cProtocol) lthy =
  let
```

```
      val pname = #name trac
      val defname = mkN(pname, enum_constsN)
      val _ = info("  Defining "^defname)
      val enames = get_enum_consts trac
      val econsts = map (fn x => ([],x)) enames
    in
      ([defname], ml_isar_wrapper.define_simple_datatype ([], defname) econsts lthy)
    end

  fun def_sets (trac:TracProtocolCert.cProtocol) lthy =
    let
      val pname = #name trac
      val defname = mkN(pname, setsN)
      val _ = info ("  Defining "^defname)

      val sspec = get_set_spec trac
      val gsspec = get_general_set_family_set_spec trac
      val tfqn = full_name' enum_constsN trac lthy
      val ttyp = Type(tfqn, [])
      val eqs =
        map (fn (x,n) => if member (op =) gsspec (x,n) then ([],x) else (replicate n ttyp,x)) sspec
    in
      lthy
      |> ml_isar_wrapper.define_simple_datatype ([], defname) eqs
    end

  fun def_funs (trac:TracProtocolCert.cProtocol) lthy =
    let
      val pname = #name trac
      val (pub_f, pub_c, priv_c) = get_funs trac
      val pub = (map (fn (f,n) => (f,n,NONE)) pub_f)@(map (fn (f,a) => (f,0,a)) pub_c)
      val priv = map (fn (f,a) => (f,0,a)) priv_c
      val declared_types = #type_spec trac

      fun def_atom lthy =
        let
          val def_atomname = mkN(pname, atomN)
          val extra_types =
            if null pub_c
            then default_extra_types
            else extended_extra_types
          val types = declared_types@extra_types
          fun define_atom_dt lthy =
            let
              val _ = info("  Defining "^def_atomname)
            in
              lthy
              |> ml_isar_wrapper.define_simple_datatype ([], def_atomname) (map (fn x => ([],x))
types)
            end
          fun prove_UNIV_atom lthy =
            let
              val _ = info ("    Proving "^def_atomname^"_UNIV")
              val thmsN = [def_atomname^".exhaust"]
              val fqn = full_name (mkN(pname, atomN)) lthy
              val typ = Type(fqn, [])
            in
              lthy
              |> prove_UNIV (def_atomname) typ (map (fn c => Const(fqn^"."^c,typ)) types) thmsN
            end
        in
           lthy
           |> define_atom_dt
```

```
        |> prove_UNIV_atom
    end

  fun def_fun_dt lthy =
    let
      val def_funname = mkN(pname, funN)
      val _ = info("  Defining "^def_funname)
      val decl_funs = map (fn x => ([],x)) (map #1 (pub@priv))
      val enum_fun = ([Type (full_name (mkN(pname, enum_constsN)) lthy, [])],enumN)

      val all_funs = decl_funs@enum_fun::map snd (get_user_atom_spec_pre trac)@
                     map (fn (e,_,_) => ([natT],infenumN e))
                         (get_nonunion_infinite_enum_spec trac)
    in
      ml_isar_wrapper.define_simple_datatype ([], def_funname) all_funs lthy
    end

  fun def_fun_arity lthy =
    let
      val fqn_name = full_name (mkN(pname, funN)) lthy
      val ctyp = Type (fqn_name, [])
      val ctyp' = Type (full_name (mkN(pname, enum_constsN)) lthy, [])
      val name = mkN(pname, arityN)

      fun mk_rec_eq typs (fname,arity,_) =
        let
          val a = Const(fqn_name^"."^fname, typs ---> ctyp)
          val b = fold (fn t => fn p => p$(Term.dummy_pattern t)) typs a
        in
          (Free(name,ctyp --> natT)$b, mk_nat(arity))
        end

      val _ = info("  Defining "^name)
    in
      ml_isar_wrapper.define_simple_fun name
          (map (mk_rec_eq []) (pub@priv)@
           mk_rec_eq [ctyp'] (enumN,0,NONE)::
           map (fn (_,(ts,f)) => mk_rec_eq ts (f,0,NONE)) (get_user_atom_spec_pre trac)@
           map (fn (e,_,_) => mk_rec_eq [natT] (infenumN e,0,NONE))
               (get_nonunion_infinite_enum_spec trac))
          lthy
    end

  fun def_set_arity lthy =
    let
      val fqn_name = full_name' setsN trac lthy
      val ctyp = Type (fqn_name, [])
      val ctyp' = Type (full_name' enum_constsN trac lthy, [])
      val name = mkN(pname, set_arityN)

      val sspec = get_set_spec trac
      val gsspec = get_general_set_family_set_spec trac

      val sspec' =
        map (fn (x,n) => if member (op =) gsspec (x,n)
                         then (x,n+1,[])
                         else (x,0,replicate n ctyp'))
            sspec

      fun mk_rec_eq (fname,arity,typs) =
        let
          val a = Const(fqn_name^"."^fname, typs ---> ctyp)
          val b = fold (fn t => fn p => p$(Term.dummy_pattern t)) typs a
```

```
      in
        (Free(name,ctyp --> natT)$b, mk_nat(arity))
      end

    val _ = info("  Defining "^name)
  in
    ml_isar_wrapper.define_simple_fun name
        (map mk_rec_eq sspec')
        lthy
  end

fun def_public lthy =
  let
    val fqn_name = full_name (mkN(pname, funN)) lthy
    val ctyp = Type (fqn_name, [])
    val ctyp' = Type (full_name (mkN(pname, enum_constsN)) lthy, [])
    val name = mkN(pname, publicN)

    fun mk_rec_eq bool_trm typs fname =
      let
        val a = Const(fqn_name^"."^fname, typs ---> ctyp)
        val b = fold (fn t => fn p => p$(Term.dummy_pattern t)) typs a
      in
        (Free(name,ctyp --> boolT)$b, bool_trm)
      end

    fun mk_rec_eq' fname =
      let
        val a = Const(fqn_name^"."^fname, [boolT,natT] ---> ctyp)
        val b = a$Term.Free ("b", boolT)$Term.dummy_pattern natT
      in
        (Free(name,ctyp --> boolT)$b, Term.Free ("b", boolT))
      end

    val _ = info("  Defining "^name)
  in
    ml_isar_wrapper.define_simple_fun name
        ((map (mk_rec_eq (@{term "False"}) []) (map #1 priv))
        @(map (mk_rec_eq (@{term "True"}) []) (map #1 pub))
        @mk_rec_eq (@{term "True"}) [ctyp'] enumN
        ::map (mk_rec_eq' o snd o snd) (get_user_atom_spec_pre trac)
        @map (fn (e,_,_) => mk_rec_eq (@{term "True"}) [natT] (infenumN e))
             (get_nonunion_infinite_enum_spec trac)
        ) lthy
  end

fun def_gamma lthy =
  let
    fun optionT t = Type (@{type_name "option"}, [t])
    fun mk_Some t = Const (@{const_name "Some"}, t --> optionT t)
    fun mk_None t = Const (@{const_name "None"},  optionT t)

    val fqn_name = full_name (mkN(pname, funN)) lthy
    val ctyp = Type (fqn_name, [])
    val atomFQN = full_name (mkN(pname, atomN)) lthy
    val atomT = Type (atomFQN, [])
    val ctyp' = Type (full_name (mkN(pname, enum_constsN)) lthy, [])
    val name = mkN(pname, gammaN)

    fun mk_atomT_trm tau = mk_Some atomT$Const(atomFQN^"."^tau, atomT)

    fun mk_rec_eq' (typname,(typs,fname)) =
      let
```

```
          val typtrm = case typname of NONE => mk_None atomT | SOME tau => mk_atomT_trm tau
          val a = Const(fqn_name^"."^fname, typs ---> ctyp)
          val b = fold (fn t => fn p => p$(Term.dummy_pattern t)) typs a
        in
          (Free(name,ctyp --> optionT atomT)$b, typtrm)
        end

      fun mk_rec_eq typname fname = mk_rec_eq' (typname,([],fname))

      val user_atom_spec = get_user_atom_spec trac
      val priv_rest = filter_out (member (op =) (map (snd o snd) user_atom_spec) o #1) priv
      val pub_c_rest = filter_out (member (op =) (map (snd o snd) user_atom_spec) o #1) pub_c

      val _ = info("  Defining "^name)
    in
      ml_isar_wrapper.define_simple_fun name
          (map (fn (s,p) => mk_rec_eq' (SOME s,p)) user_atom_spec
          @map (mk_rec_eq (SOME secret_typeN) o #1) priv_rest
          @map (mk_rec_eq (SOME other_pubconsts_typeN) o #1) pub_c_rest
          @mk_rec_eq' (SOME enum_typeN,([ctyp'],enumN))
          ::map (mk_rec_eq NONE o #1) pub_f
          @map (fn (e,_,_) => mk_rec_eq' (SOME enum_typeN,([natT],infenumN e)))
               (get_nonunion_infinite_enum_spec trac)
          ) lthy
    end

  fun def_ana lthy = let
    val pname = #name trac
    val (pub_f, pub_c, priv_c) = get_funs trac
    val pub = (map (fn (f,n) => (f,n,NONE)) pub_f)@(map (fn (f,a) => (f,0,a)) pub_c)
    val priv = map (fn (f,a) => (f,0,a)) priv_c

    val keyT = messageT' natT trac lthy

    val fqn_name = full_name (mkN(pname, funN)) lthy
    val ctyp = Type (fqn_name, [])
    val ctyp' = Type (full_name (mkN(pname, enum_constsN)) lthy, [])
    val name = mkN(pname, anaN)

    val ana_outputT = mk_prodT (mk_listT keyT, mk_listT natT)

    val default_output = mk_prod (mk_list keyT [], mk_list natT [])

    fun mk_ana_output ks rs = mk_prod (mk_list keyT ks, mk_list natT rs)

    fun mk_rec_eq ana_output_trm typs fname =
      let
        val a = Const(fqn_name^"."^fname, typs ---> ctyp)
        val b = fold (fn t => fn p => p$(Term.dummy_pattern t)) typs a
      in
        (Free(name,ctyp --> ana_outputT)$b, ana_output_trm)
      end

    val _ = info("  Defining "^name)

    val ana_spec =
      let
        fun var_to_nat is_priv_fun f xs x =
          let
            val n = snd (Option.valOf ((list_find (fn y => y = x) xs)))
          in
            if is_priv_fun then mk_nat (1+n) else mk_nat n
          end
```

```
          fun c_to_h is_priv_fun f xs t = ana_cMsg_to_hol is_priv_fun t xs trac lthy
          fun keys is_priv_fun f ps ks = map (c_to_h is_priv_fun f ps) ks
          fun results is_priv_fun f ps rs = map (var_to_nat is_priv_fun f ps) rs
          fun aux ({head=(f,ps), keys=ks, results=rs, is_priv_fun=b}:TracProtocolCert.cAnaRule) =
            (f, mk_ana_output (keys b f ps ks) (results b f ps rs))
        in
          map aux (#analysis_spec trac)
        end

      val other_funs =
        filter (fn f => not (List.exists (fn g => f = g) (map fst ana_spec))) (map #1 (pub@priv))
    in
      ml_isar_wrapper.define_simple_fun name
          (map (fn (f,out) => mk_rec_eq out [] f) ana_spec
          @map (mk_rec_eq default_output []) other_funs
          @mk_rec_eq default_output [ctyp'] enumN
          ::map (fn (_,(typs,f)) => mk_rec_eq default_output typs f) (get_user_atom_spec_pre trac)
          @map (fn (e,_,_) => mk_rec_eq default_output [natT] (infenumN e))
               (get_nonunion_infinite_enum_spec trac)
        )
        lthy
    end

  in
    lthy |> def_atom
         |> def_fun_dt
         |> def_fun_arity
         |> def_set_arity
         |> def_public
         |> def_gamma
         |> def_ana
  end

fun define_term_model (trac:TracProtocolCert.cProtocol) lthy =
  let
    val _ = info("Defining term model")
  in
    lthy |> snd o def_enum_consts trac
         |> def_sets trac
         |> def_funs trac
  end

fun define_fixpoint fp_triple trac print lthy =
  let
    val fp_name = mkN (#name trac, "fixpoint")
    val _ = info("Defining fixpoint")
    val _ = info("  Defining "^fp_name)
    val fp_triple_trm = fp_triple_to_hol fp_triple trac lthy
  in
    (trac, #2 (ml_isar_wrapper.define_constant_definition' (fp_name, fp_triple_trm) print lthy))
  end

fun define_protocol print ((trac:TracProtocolCert.cProtocol), lthy) = let
    val _ =
      if length (#transaction_spec trac) > 1
      then info("Defining protocols")
      else info("Defining protocol")
    val pname = #name trac

    val mk_Send = mk_Send_step trac lthy
    val mk_Receive = mk_Receive_step trac lthy
    val mk_InSet = mk_InSet_step trac lthy
    val mk_NotInSet = mk_NotInSet_step trac lthy
```

```
val mk_NegChecks = mk_NegChecks_step trac lthy
val mk_Equality = mk_Equality_step trac lthy
val mk_Insert = mk_Insert_step trac lthy
val mk_Delete = mk_Delete_step trac lthy

val star_label = mk_star_label
val prot_label = mk_prot_label

fun mk_tname i tr =
  let
    val x = #1 tr
    val y = case i of NONE => x | SOME n => mkN(n, x)
    val z = mkN("transaction", y)
  in mkN(pname, z)
  end

fun def_transaction name_prefix prot_num (transaction:TracProtocolCert.cTransaction) lthy = let
  val defname = mk_tname name_prefix (#transaction transaction)
  val _ = info("  Defining "^defname)

  val receives       = #receive_actions     transaction
  val checkssingle   = #checksingle_actions transaction
  val checksall      = #checkall_actions    transaction
  val updates        = #update_actions      transaction
  val sends          = #send_actions        transaction
  val fresh          = #fresh_actions       transaction
  val attack_signals = #attack_actions      transaction

  val fresh_vars          = get_fresh_variables            transaction
  val nonfresh_value_vars = get_nonfresh_value_variables   transaction
  val finenum_vars        = get_finite_enum_variables      transaction
  val enumtype_vars       = get_enumtype_variables         transaction
  val nonenum_vars        = get_nonenum_variables          transaction
  val infenum_vars        = get_infinite_enum_variables    transaction
  val all_decl_vars       = get_transaction_head_variables transaction
  val bvars               = get_bound_variables            transaction

  val nonfinenum_vars =
    filter (member (op =) (nonenum_vars@infenum_vars)) (all_decl_vars@fresh_vars)

  val infenum_enumtype_vars =
    filter (member (op =) (enumtype_vars@infenum_vars)) (all_decl_vars@fresh_vars)

  val (enum_ineqs_in_head, value_ineqs_in_head) = get_variable_restrictions transaction

  val value_single_ineqs_in_body =
    let
      open Trac_Term TracProtocolCert
      fun aux (_,cNegChecks ([], [cInequality (cVar (x, ValueType), cVar (y, ValueType))])) =
            SOME (x,y)
        | aux _ = NONE
    in
      map_filter aux checkssingle
    end

  val declared_value_ineqs = distinct (op =) (value_ineqs_in_head@value_single_ineqs_in_body)

  val enable_let_bindings = true

  fun c_to_h' b trm = transaction_cMsg_to_hol
    trm prot_num
    (nonfinenum_vars@bvars)
    b trac lthy
```

```
val c_to_h = c_to_h' enable_let_bindings

val abstract_over_finenum_vars = fn x => fn y => fn z =>
  abstract_over_finite_enum_vars x y z trac lthy

fun mk_transaction_term (rcvs, chcksingle, chckall, upds, snds, frsh_xs, frsh_acs, atcks) =
  let
    open Trac_Term TracProtocolCert
    fun action_filter f (lbl,a) = case f a of SOME x => SOME (lbl,x) | NONE => NONE

    fun lbl_to_h LabelS = star_label
      | lbl_to_h LabelN = prot_label prot_num

    fun lbl_trms_to_h f (lbl,ts) = f (lbl_to_h lbl) (map c_to_h ts)

    val S0 =
      let
        val msgT = messageT trac lthy
        val varT = message_varT trac lthy
        val funN = full_name' funN trac lthy
        val funT = funT trac lthy
        val enum_constsT = enum_constsT trac lthy
        val infenumspec = get_infinite_enum_spec trac
        val botinfenums = map #1 (get_nonunion_infinite_enum_spec trac)
        val enum_constructor = Term.Const (funN ^ "." ^ enumN, enum_constsT --> funT)
        fun mk_enum_const' a = mk_enum_const a trac lthy
        fun mk_union typ [] = Term.Const ("Set.empty", mk_setT typ)
          | mk_union typ (t::ts) =
              fold (fn s => fn u =>
                Term.Const ("Set.union", [mk_setT typ, mk_setT typ] ---> mk_setT typ) $
                u $ s) ts t
        val ran_trm_finenums =
          Term.Const ("Set.range", (enum_constsT --> funT) --> mk_setT funT) $
          enum_constructor
        fun ran_trm_botinfenum e =
          Term.Const ("Set.range", (natT --> funT) --> mk_setT funT) $
          Term.Const (funN ^ "." ^ infenumN e, natT --> funT)
        fun ran_trm_infenums e =
          case List.find (fn (a,_,_) => a = e) infenumspec of
            SOME (_,cs,es) => mk_union funT (map ran_trm_botinfenum es@
              (if null cs then []
               else [mk_set funT (map (fn c => enum_constructor $ mk_enum_const' c) cs)]))
          | NONE => error ("Couldn't find enumeration " ^ e)
        fun consts (_,EnumType) =
              mk_union funT (ran_trm_finenums::map ran_trm_botinfenum botinfenums)
          | consts (_,InfiniteEnumeration e) = ran_trm_infenums e
          | consts x = error ("Error: Expected an enumeration variable or a variable of " ^
                              "type " ^ enum_typeN ^ ", but got " ^ cMsg_str (cVar x))
        fun var_trm x =
          Term.Const (@{const_name "the_Var"}, msgT --> varT) $ c_to_h (cVar x)
      in
        map (fn x => mk_prod (var_trm x, consts x)) infenum_enumtype_vars
      end

    val S1 = map (lbl_trms_to_h mk_Receive)
                 (map_filter (action_filter maybe_the_Receive) rcvs)

    val additional_value_ineqs =
      let
        val tr_acs = rcvs@chcksingle@chckall@frsh_acs@upds@snds@atcks

        val (vars_in_acs, vars_in_star_acs) =
```

```
        let val f = distinct (op =) o List.concat o map (cAction_fvs o snd)
        in (f tr_acs, f (filter (fn (l,_) => l = LabelS) tr_acs)) end
      val (nonfresh_vals_in_acs, nonfresh_vals_in_star_acs) =
        let fun f xs = filter (fn x => member (op =) xs x) nonfresh_value_vars
        in (f vars_in_acs, f vars_in_star_acs) end

      fun mk_Value_cVar x = cVar (x,ValueType)
      fun mk_cInequality star_acs_vars x y =
        let val mem = member (op =) star_acs_vars
            val lbl = if mem x andalso mem y then LabelS else LabelN
        in (lbl, cNegChecks ([],[cInequality (mk_Value_cVar x, mk_Value_cVar y)])) end
      fun mk_cInequalities star_acs_vars =
        list_triangle_product (mk_cInequality star_acs_vars)
      val ineqs = mk_cInequalities nonfresh_vals_in_star_acs nonfresh_vals_in_acs
    in filter (not o member (op =) chcksingle) ineqs end

  val S2 =
    let
      fun aux (lbl,cEquality (pcv,(x,y))) =
            SOME (mk_Equality pcv (lbl_to_h lbl) (c_to_h x) (c_to_h y))
        | aux (lbl,cInSet (pcv,(e,s))) =
            SOME (mk_InSet pcv (lbl_to_h lbl) (c_to_h e) (c_to_h s))
        | aux (lbl,cNegChecks (xs,ns)) =
            let
              fun f (a,b) = (c_to_h a, c_to_h b)
              val ineqs = map f (map_filter maybe_the_Inequality ns)
              val notins = map f (map_filter maybe_the_NotInSet ns)
              val bvars = map (c_to_h o cVar) xs
            in
              SOME (mk_NegChecks (lbl_to_h lbl) bvars ineqs notins)
            end
        | aux _ = NONE
    in
      map_filter aux (additional_value_ineqs@chcksingle)
    end

  val S3 =
    let
      fun arity s = case set_arity trac s of
          SOME n => n
        | NONE => error ("Error: Not a set family: " ^ s)

      fun mk_evs s =
        map (fn n => ("X" ^ Int.toString n, Untyped)) (0 upto ((arity s) -1))

      fun mk_trm (lbl,e,s) =
        let
          val ps = map (fn x => cVar (x,EnumType)) (map fst (mk_evs s))
        in
          mk_NotInSet (lbl_to_h lbl) (c_to_h e) (c_to_h (cSet (s,ps)))
        end

      fun mk_trms (lbl,(e,s)) =
        abstract_over_finenum_vars (mk_evs s) [] (mk_trm (lbl,e,s))
    in
      map mk_trms (map_filter (action_filter maybe_the_NotInAny) chckall)
    end

  val S4 = map (c_to_h o cVar) frsh_xs

  val S5 =
    let
      fun aux (lbl,cInsert (e,s)) = SOME (mk_Insert (lbl_to_h lbl) (c_to_h e) (c_to_h s))
```

```
          | aux (lbl,cDelete (e,s)) = SOME (mk_Delete (lbl_to_h lbl) (c_to_h e) (c_to_h s))
          | aux _ = NONE
      in
        map_filter aux upds
      end

    val S6 =
      let val snds' = map_filter (action_filter maybe_the_Send) snds
      in map (lbl_trms_to_h mk_Send) (snds'@map (fn (lbl,_) => (lbl,[cAttack])) atcks) end
  in
    mk_Transaction trac lthy S0 S1 S2 S3 S4 S5 S6
      |> abstract_over_finenum_vars finenum_vars enum_ineqs_in_head
      |> (fn trm =>
            if not (null nonfinenum_vars) andalso enable_let_bindings
            then let
              val typof = Term.fastype_of
              val xs = nonfinenum_vars@bvars
              val a = absfreeprod (messageT trac lthy) (map fst xs) trm
              val b = mk_tuple (map (c_to_h' false o cVar) xs)
              val c = Term.Const (@{const_name "Let"}, [typof b, typof a] ---> typof (a$b))
            in c$b$a end
            else trm)
  end

fun def_trm trm print lthy =
  #2 (ml_isar_wrapper.define_constant_definition' (defname, trm) print lthy)
  |> declare_def_attr "protocol_defs" defname false


val value_ineqs_from_unsat_set_constrs =
  let
    open Trac_Term TracProtocolCert
    val poschecks = map_filter (maybe_the_InSet o snd) checkssingle
    val negchecks_single = List.concat (map (map_filter maybe_the_NotInSet o snd)
                                            (map_filter (maybe_the_NegChecks o snd)
                                                        checkssingle))
    val negchecks_all = map_filter (maybe_the_NotInAny o snd) checksall

    fun aux' (cVar (x,ValueType),s) (cVar (y,ValueType),t) =
          if s = t then SOME (x,y) else NONE
      | aux' _ _ = NONE

    fun aux (x,cSet (s,ps)) = SOME (
          map_filter (aux' (x,cSet (s,ps))) negchecks_single@
          map_filter (aux' (x,s)) negchecks_all
        )
      | aux _ = NONE
  in
    List.concat (map_filter aux poschecks)
  end

val all_value_ineqs =
  distinct (op =) (declared_value_ineqs@value_ineqs_from_unsat_set_constrs)

val valvarsprod =
      filter (fn p => not (List.exists (fn q => p = q orelse swap p = q) all_value_ineqs))
             (list_triangle_product (fn x => fn y => (x,y)) nonfresh_value_vars)

val transaction_trm0 = mk_transaction_term
      (receives, checkssingle, checksall, updates, sends, fresh_vars, fresh, attack_signals)
in
  if null valvarsprod
  then def_trm transaction_trm0 print lthy
```

```
    else let
      open Trac_Term TracProtocolCert
      val partitions = list_partitions nonfresh_value_vars all_value_ineqs
      val ps = filter (not o null) (map (filter (fn x => length x > 1)) partitions)

      fun mk_subst ps =
        let
          fun aux [] = NONE
            | aux (x::xs) = SOME (map (fn y => (y,cVar (x,ValueType))) xs)
        in
          List.concat (map_filter aux ps)
        end

      fun apply d =
        let
          val ap = subst_apply_cActions d
        in
          (ap receives, ap checkssingle, ap checksall, ap updates, ap sends, fresh_vars,
           ap fresh, attack_signals)
        end

      val transaction_trms = transaction_trm0::map (mk_transaction_term o apply o mk_subst) ps
      val transaction_typ = Term.fastype_of transaction_trm0

      fun mk_concat_trm tau trms =
        Term.Const (@{const_name "concat"}, mk_listT tau --> tau) $ mk_list tau trms
    in
      def_trm (mk_concat_trm transaction_typ transaction_trms) print lthy
    end
  end

  val def_transactions =
    let
      val prots = map (fn (n,pr) => map (fn tr => (n,tr)) pr) (#transaction_spec trac)
      val lbls = list_upto (length prots)
      val lbl_prots = List.concat (map (fn i => map (fn tr => (i,tr)) (nth prots i)) lbls)
      val f = fold (fn (i,(n,tr)) => def_transaction n i tr)
    in
      f lbl_prots
    end

  fun def_protocols lthy = let
      fun mk_prot_def (name,trm) lthy =
        let val _ = info("  Defining "^name)
        in #2 (ml_isar_wrapper.define_constant_definition' (name,trm) print lthy)
               |> declare_def_attr "protocol_def" name false
         end

      val prots = #transaction_spec trac
      val num_prots = length prots

      val pdefname = mkN(pname, "protocol")

      fun mk_tnames i =
        let
          val trs = case nth prots i of (j,prot) => map (fn tr => (j,tr)) prot
        in map (fn (j,s) => full_name (mk_tname j (#transaction s)) lthy) trs
        end

      val tnames = List.concat (map mk_tnames (list_upto num_prots))

      val pnames =
        let
```

```
          val f = fn i => (Int.toString i,nth prots i)
          val g = fn (i,(n,_)) => case n of NONE => i | SOME m => m
          val h = fn s => mkN (pdefname,s)
        in map (h o g o f) (list_upto num_prots)
        end

      val trtyp = prot_transactionT trac lthy
      val trstyp = mk_listT trtyp

      fun mk_prot_trm names =
        Term.Const (@{const_name "concat"}, mk_listT trstyp --> trstyp) $
        mk_list trstyp (map (fn x => Term.Const (x, trstyp)) names)

      val lthy =
        if num_prots > 1
        then fold (fn (i,pname) => mk_prot_def (pname, mk_prot_trm (mk_tnames i)))
                  (map (fn i => (i, nth pnames i)) (list_upto num_prots))
                  lthy
        else lthy

      val pnames' = map (fn n => full_name n lthy) pnames

      fun mk_prot_trm_with_star i =
        let
          fun f j =
            if j = i
            then Term.Const (nth pnames' j, trstyp)
            else (Term.Const (@{const_name "map"}, [trtyp --> trtyp, trstyp] ---> trstyp) $
                  Term.Const ("Transactions.transaction_star_proj", trtyp --> trtyp) $
                  Term.Const (nth pnames' j, trstyp))
        in
          Term.Const (@{const_name "concat"}, mk_listT trstyp --> trstyp) $
          mk_list trstyp (map f (list_upto num_prots))
        end

      fun mk_star_prot_trm () =
        let
          fun f j =
            (Term.Const (@{const_name "map"}, [trtyp --> trtyp, trstyp] ---> trstyp) $
             Term.Const ("Transactions.transaction_star_proj", trtyp --> trtyp) $
             Term.Const (nth pnames' j, trstyp))
        in
          Term.Const (@{const_name "concat"}, mk_listT trstyp --> trstyp) $
          mk_list trstyp (map f (list_upto num_prots))
        end

      val lthy =
        if num_prots > 1
        then fold (fn (i,pname) => mk_prot_def (pname, mk_prot_trm_with_star i))
                  (map (fn i => (i, nth pnames i ^ "_with_star_projs")) (list_upto num_prots))
                  lthy
        else lthy

      val lthy =
        if num_prots > 1
        then mk_prot_def (pdefname ^ "_star_projs", mk_star_prot_trm ()) lthy
        else lthy
  in
    mk_prot_def (pdefname, mk_prot_trm (if num_prots > 1 then pnames' else tnames)) lthy
  end
in
  (trac, lthy |> def_transactions |> def_protocols)
end
```

```
end
›
```

ML‹

```
structure trac = struct
  open Trac_Term

  val info = Output.information

  fun mk_abs_filename thy filename =
      let
        val filename = Path.explode filename
        val master_dir = Resources.master_directory thy
      in
        Path.implode (if (Path.is_absolute filename)
                      then filename
                      else Path.append master_dir filename)
      end

  fun def_fp print (trac:TracProtocolCert.cProtocol, lthy) =
    case #fixed_point trac of
      SOME fp => trac_definitorial_package.define_fixpoint fp trac print lthy
    | NONE => (trac, lthy)
    (* let
      val fp = TracFpParser.parse_str fp_str
      val (trac,lthy) = trac_definitorial_package.define_fixpoint fp trac print lthy
      val lthy = Local_Theory.raw_theory (update trac) lthy
    in
      (trac, lthy)
    end *)

  fun def_trac_term_model trac lthy =
    let
      val lthy:local_theory = trac_definitorial_package.define_term_model trac lthy
    in
      (trac, lthy)
    end

  val def_trac_protocol = trac_definitorial_package.define_protocol

  fun def_trac trac_str opt_fp_str print lthy =
    let
      val trac = TracProtocolParser.parse_str trac_str
      val trac = case opt_fp_str of
                   SOME fp_str =>
                     TracProtocol.update_fixed_point trac (SOME (TracFpParser.parse_str fp_str))
                 | NONE => trac
      val lthy = Local_Theory.raw_theory (trac_definitorial_package.update trac) lthy
      val ctrac = TracProtocolCert.certifyProtocol trac
    in
      (def_fp print o def_trac_protocol print o def_trac_term_model ctrac) lthy
    end

  fun def_trac_file trac_filename opt_fp_filename print lthy = let
        fun read_file filename =
          File.read (Path.explode (mk_abs_filename (Proof_Context.theory_of lthy) filename))
        val trac_str = read_file trac_filename
        val opt_fp_str = Option.map (fn fp_filename => read_file fp_filename) opt_fp_filename
        val (trac,lthy) = def_trac trac_str opt_fp_str print lthy
      in
        (trac, lthy)
      end
```

```
end
›
```

ML‹
```
  val fileNameP = Parse.name -- Parse.name

  val _ = Outer_Syntax.local_theory' @{command_keyword "trac"}
          "Define protocol and (optionally) fixpoint using trac format."
          ((Parse.cartouche -- Scan.optional Parse.cartouche "" >> (
            fn (trac,fp) => fn print => fn lthy =>
          let
            val opt_fp = if fp = "" then NONE else SOME fp
            val trac = trac.def_trac trac opt_fp print #> snd
          in
            trac_time.ap_lthy lthy ("trac") trac lthy
          end)));

  val _ = Outer_Syntax.local_theory' @{command_keyword "trac_import"}
          "Import protocol and (optionally) fixpoint from trac files."
          ((Parse.name -- Scan.optional Parse.name "" >> (
            fn (trac_filename, fp_filename) => fn print => fn lthy =>
          let
            val opt_fp_filename = if fp_filename = "" then NONE else SOME fp_filename
            val trac = trac.def_trac_file trac_filename opt_fp_filename print #> snd
          in
            trac_time.ap_lthy lthy ("trac_import") trac lthy
          end)));
›
```

ML‹
```
val name_prefix_parser = Parse.!!! (Parse.name --| Parse.$$$ ":" -- Parse.name)

(* Original definition (opt_evaluator) copied from value_command.ml *)
val opt_proof_method_choice =
  Scan.optional (keyword‹[› |-- Parse.name --| keyword‹]›) "safe";

(* Original definition (locale_expression) copied from parse_spec.ML *)
val security_proof_locale_opt_defs_list = Scan.optional
  (keyword‹for› |-- Scan.repeat1 Parse.name >>
      (fn xs => if length xs > 3 then error "Too many optional arguments" else xs))
  [];

val composed_protocol_locale_defs_list =
  (keyword‹for› |-- Parse.!!! (
                    Parse.name --   (* The composed protocol *)
                    Parse.name --   (* Its SMP set *)
                    Parse.name)) -- (* The (symbolic) list of shared secrets *)
  (keyword‹and› |-- Scan.repeat1 Parse.name >> (* The component protocols *)
      (fn xs => if length xs < 2 then error "Too few arguments" else xs)) --
  (keyword‹and› |-- Scan.repeat1 Parse.name >> (* The component protocols with star projections *)
      (fn xs => if length xs < 2 then error "Too few arguments" else xs)) --
  (keyword‹and› |-- Scan.repeat1 Parse.name >> (* Their GSMPs *)
      (fn xs => if length xs < 2 then error "Too few arguments" else xs));

val security_proof_locale_parser =
  name_prefix_parser -- security_proof_locale_opt_defs_list

val security_proof_locale_parser_with_method_choice =
  opt_proof_method_choice -- name_prefix_parser -- security_proof_locale_opt_defs_list

val composed_protocol_locale_parser =
  name_prefix_parser -- composed_protocol_locale_defs_list
```

```
val composed_protocol_locale_parser_with_method_choice =
  opt_proof_method_choice -- name_prefix_parser -- composed_protocol_locale_defs_list

fun protocol_model_setup_proof_state name prefix lthy =
  let
    fun f x y z = ([((x,Position.none),((y,true),(Expression.Positional z,[])))],[])
    val _ = assert_nonempty_name name
    val pexpr = f "stateful_protocol_model" name (protocol_model_interpretation_params prefix lthy)
    val pdefs = protocol_model_interpretation_defs name
    val proof_state = Interpretation.global_interpretation_cmd pexpr pdefs lthy
  in
    proof_state
  end

fun protocol_security_proof_defs manual_proof name prefix opt_defs opt_meth_level lthy =
  let
    fun f x y z = ([((x,Position.none),((y,true),(Expression.Positional z,[])))],[])
    val _ = assert_nonempty_name name
    val num_defs = length opt_defs
    val pparams = protocol_model_interpretation_params prefix lthy
    val default_defs = [prefix ^ "_" ^ "protocol", prefix ^ "_" ^ "fixpoint"]
    val meth_variant = if String.isSuffix "_coverage_rcv" opt_meth_level then "_coverage_rcv" else ""
    fun g locale_name extra_params = f locale_name name (pparams@map SOME extra_params)
    fun h locale_variant = g ("secure_stateful_protocol" ^ meth_variant ^ locale_variant)
    val (prot_fp_smp_names, pexpr) = if manual_proof
      then (case num_defs of
        0 => (default_defs, h "'" default_defs)
      | 1 => (opt_defs, h "''" opt_defs)
      | 2 => (opt_defs, h "'" opt_defs)
      | _ => (opt_defs, h "" opt_defs))
      else (case num_defs of
        0 => (default_defs, h "''''" default_defs)
      | 1 => (opt_defs, h "''" opt_defs)
      | 2 => (opt_defs, h "''''" opt_defs)
      | _ => (opt_defs, h "'''" opt_defs))
    val _ = assert_all_defined lthy prefix prot_fp_smp_names
  in
    (prot_fp_smp_names, pexpr)
  end

fun protocol_security_proof_proof_state manual_proof name prefix opt_defs opt_meth_level print lthy =
  let
    val (prot_fp_smp_names, pexpr) =
      protocol_security_proof_defs manual_proof name prefix opt_defs opt_meth_level lthy
    val proof_state = lthy |> declare_protocol_checks print
                           |> Interpretation.global_interpretation_cmd pexpr []
  in
    (prot_fp_smp_names, proof_state)
  end

fun protocol_composition_proof_defs name prefix remaining_params lthy =
  let
    fun f x y z = ([((x,Position.none),((y,true),(Expression.Positional z,[])))],[])
    fun g xs = "[" ^ String.concatWith ", " xs ^ "]"
    fun h xs = g (map_index (fn (i,x) => "(" ^ Int.toString i ^ ", " ^ x ^ ")") xs)
    val _ = assert_nonempty_name name
    val (((((pc,smp),sec),ps),psstarprojs),gsmps) = remaining_params
    val _ = assert_all_defined lthy prefix ([pc,smp,sec]@ps@psstarprojs@gsmps)
    val _ = if length ps = length psstarprojs andalso length ps = length gsmps then ()
            else error "Missing arguments"
    val pparams = protocol_model_interpretation_params prefix lthy
    val params = [pc, g ps, g psstarprojs, smp, sec, h gsmps]
```

```
    val pexpr = f "composable_stateful_protocols" name (pparams@map SOME params)
  in
    pexpr
  end

fun protocol_composition_proof_proof_state name prefix params print lthy =
  let
    val pexpr = protocol_composition_proof_defs name prefix params lthy
    val state = lthy |> (declare_protocol_check "wellformed_composable_protocols" print #>
                         declare_protocol_check "wellformed_composable_protocols'" print #>
                         declare_protocol_check "composable_protocols" print)
                     |> Interpretation.global_interpretation_cmd pexpr []
  in
    state
  end

val select_proof_method_error_prefix =
  "Error: Invalid option: "

fun select_proof_method_error msg opt_meth_level =
  error (
    select_proof_method_error_prefix ^ opt_meth_level ^ "\n\nValid options:\n" ^
    "1. safe: Instructs Isabelle to " ^ msg ^ " using \"code_simp\" " ^
    "(this is the default setting).\n" ^
    "2. nbe: Instructs Isabelle to use \"normalization\" instead of \"code_simp\".\n" ^
    "3. eval: Instructs Isabelle to use \"eval\" instead of \"code_simp\".")

 fun select_proof_method _ "safe" = "check_protocol"
  | select_proof_method _ "safe_coverage_rcv" = "check_protocol"
  | select_proof_method _ "nbe" = "check_protocol_nbe"
  | select_proof_method _ "nbe_coverage_rcv" = "check_protocol_nbe"
  | select_proof_method _ "eval" = "check_protocol_eval"
  | select_proof_method _ "eval_coverage_rcv" = "check_protocol_eval"
  | select_proof_method msg opt_meth_level = error (
          select_proof_method_error_prefix ^ opt_meth_level ^ "\n\nValid options:\n" ^
          "1. safe: Instructs Isabelle to " ^ msg ^ " using \"code_simp\" " ^
          "(this is the default setting).\n" ^
          "2. nbe: Instructs Isabelle to use \"normalization\" instead of \"code_simp\".\n" ^
          "3. eval: Instructs Isabelle to use \"eval\" instead of \"code_simp\".\n" ^
          "4. safe_coverage_rcv: Instructs Isabelle to " ^ msg ^ " using \"code_simp\" " ^
          "(with alternative coverage check).\n" ^
          "5. nbe_coverage_rcv: Instructs Isabelle to use \"normalization\" instead of \"code_simp\" "
^
          "(with alternative coverage check).\n" ^
          "6. eval_coverage_rcv: Instructs Isabelle to use \"eval\" instead of \"code_simp\"" ^
          "(with alternative coverage check).")

  | select_proof_method msg opt_meth_level =
      select_proof_method_error msg opt_meth_level

fun select_proof_method_compositionality _ "safe" = "check_protocol_compositionality"
  | select_proof_method_compositionality _ "nbe" = "check_protocol_compositionality_nbe"
  | select_proof_method_compositionality _ "eval" = "check_protocol_compositionality_eval"
  | select_proof_method_compositionality msg opt_meth_level =
      select_proof_method_error msg opt_meth_level


val _ =
  Outer_Syntax.local_theory command_keyword‹protocol_model_setup›
    "prove interpretation of protocol model locale into global theory"
    (name_prefix_parser >> (fn (name,prefix) => fn lthy =>
    let fun protocol_model_setup ((name,prefix),lthy) =
      let
```

```
        val proof_state = protocol_model_setup_proof_state name prefix lthy
        val meth =
          let
            val m = "protocol_model_interpretation"
            val _ = Output.information (
                      "Proving protocol model locale instance with proof method " ^ m)
          in
            Method.Source (Token.make_src (m, Position.none) [])
          end
      in
        ml_isar_wrapper.prove_state_simple meth proof_state
      end
    in
      trac_time.ap_lthy lthy ("protocol_model_setup ("^name^")") protocol_model_setup
((name,prefix),lthy)
    end));

val _ =
  Outer_Syntax.local_theory_to_proof command_keyword‹manual_protocol_model_setup›
    "prove interpretation of protocol model locale into global theory"
    (name_prefix_parser >> (fn (name,prefix) => fn lthy =>
    let
      val proof_state = protocol_model_setup_proof_state name prefix lthy
      val subgoal_proof = "  subgoal by protocol_model_subgoal\n"
      val _ = Output.information ("Example proof:\n" ^
                Active.sendback_markup_command ("  apply unfold_locales\n"^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                "  done\n"))
    in
      proof_state
  end));
›
```

ML‹

```
structure protocol_security_proof = struct
        fun protocol_security_proof (params, print, lthy) =
        let
          val ((opt_meth_level,(name,prefix)),opt_defs) = params
          val (defs, proof_state) =
            protocol_security_proof_proof_state false name prefix opt_defs opt_meth_level print lthy
          val num_defs = length defs
          val meth =
              let
                val m = select_proof_method "prove the protocol secure" opt_meth_level
                val info = Output.information
                val _ = info ("Proving security of protocol " ^ nth defs 0 ^
                              " with proof method " ^ m ^(" ("^opt_meth_level^")"))
                val _ = if num_defs > 1 then info ("Using fixed point " ^ nth defs 1) else ()
                val _ = if num_defs > 2 then info ("Using SMP set " ^ nth defs 2) else ()
              in
                Method.Source (Token.make_src (m, Position.none) [])
              end
        in
           ml_isar_wrapper.prove_state_simple meth proof_state
        end


      fun protocol_security_proof_with_error_messages (params, (name,prefix, opt_defs,
opt_meth_level), print, lthy) =
```

```
protocol_security_proof(params, print, lthy)
handle
  ERROR msg =>
    if String.isPrefix "Duplicate fact declaration" msg
    then error (
      "Failed to finalize proof because of duplicate fact declarations.\n" ^
      "This might happen if \"" ^ name ^ "\" was used previously.\n" ^
      "\n\nOriginal error message:\n" ^ msg)
    else if String.isPrefix select_proof_method_error_prefix msg
    then error msg
    else (* if String.isPrefix "Wellsortedness error" msg orelse
            String.isPrefix "Failed to finish proof" msg orelse
            String.isPrefix "error in proof state" msg
    then *)
    let
      val (def_names,_) =
        protocol_security_proof_defs false name prefix opt_defs opt_meth_level lthy
      val (prot_name,fp_name,smp_name) = case length def_names of
          0 => (prefix^"_protocol", prefix^"_fixpoint", prefix^"_SMP")
        | 1 => (nth def_names 0,    prefix^"_fixpoint", prefix^"_SMP")
        | 2 => (nth def_names 0,    nth def_names 1,    prefix^"_SMP")
        | _ => (nth def_names 0,    nth def_names 1,    nth def_names 2)
    in error (
      "Failed to prove the protocol secure.\n" ^
      "Click on the following to inspect which parts of the proof failed:\n" ^
      Active.sendback_markup_command (
        (if length def_names < 2
         then "— First compute a fixed-point\n" ^
              "compute_fixpoint "^prot_name^" "^fp_name^"\n\n"
         else "")^
        "— Is the fixed point free of attack signals?\n" ^
        "value \"attack_notin_fixpoint "^fp_name^"\"\n\n" ^
        "— Is the protocol covered by the fixed point?\n" ^
        "value \"protocol_covered_by_fixpoint "^fp_name^" "^prot_name^"\"\n\n" ^
        "— Is the fixed point analyzed?\n" ^
        "value \"analyzed_fixpoint "^fp_name^"\"\n\n" ^
        "— Is the protocol well-formed?\n" ^
        (if length def_names < 3
         then "value \"wellformed_protocol "^prot_name^"\"\n\n"
         else "value \"wellformed_protocol' "^prot_name^" "^smp_name^"\"\n\n")^
        "— Is the fixed point well-formed?\n" ^
        "value \"wellformed_fixpoint "^fp_name^"\"") ^
      "\n\nOriginal error message:\n" ^ msg)
    end
    (* else error msg *)


fun parallel' (params, print, lthy) =
let
  val ((opt_meth_level,(name,prefix)),opt_defs) = params
  val (defs, proof_state) =
    protocol_security_proof_proof_state false name prefix opt_defs opt_meth_level print lthy
  val num_defs = length defs
  val meth =
      let
        val m = select_proof_method "prove the protocol secure" opt_meth_level
        val info = Output.information
        val _ = info ("Proving security of protocol " ^ nth defs 0 ^
                      " with proof method " ^ m ^(" ("^opt_meth_level^")"))
        val _ = if num_defs > 1 then info ("Using fixed point " ^ nth defs 1) else ()
        val _ = if num_defs > 2 then info ("Using SMP set " ^ nth defs 2) else ()
      in
        Method.Source (Token.make_src (m, Position.none) [])
```

```
          end
      in
         ml_isar_wrapper.prove_state_simple meth proof_state
      end

              fun run_in_parallel (params', print, lthy) = let
              val ((opt_meth_level,(name,prefix)),opt_defs) = params'
              val _ = if opt_meth_level = "safe" orelse opt_meth_level = "nbe"
                                                 orelse opt_meth_level = "eval"
                      then ()
                      else error ("Only methods \"eval\", \"nbe\", and \"safe\", supported for
parallel execution.")

              val opt_meth_level_classic = opt_meth_level
              val opt_meth_level_receive = opt_meth_level^"_coverage_rcv"
              fun f variant =
                   SOME (variant, protocol_security_proof
                                 (((variant,(name,prefix)),opt_defs), print, lthy))
            in
              case Par_List.get_some f [opt_meth_level_classic, opt_meth_level_receive] of
                   SOME(v,t) => let val _ = warning ("First Successful Termination: "^v) in (v,t) end

                 |NONE => ("",lthy)
            end


    fun parallel (params, (name,prefix, opt_defs, opt_meth_level), print, lthy) =
      run_in_parallel (params, print, lthy)
      |> snd
      handle
        ERROR msg =>
          if String.isPrefix "Duplicate fact declaration" msg
          then error (
            "Failed to finalize proof because of duplicate fact declarations.\n" ^
            "This might happen if \"" ^ name ^ "\" was used previously.\n" ^
            "\n\nOriginal error message:\n" ^ msg)
          else if String.isPrefix select_proof_method_error_prefix msg
          then error msg
          else (* if String.isPrefix "Wellsortedness error" msg orelse
                  String.isPrefix "Failed to finish proof" msg orelse
                  String.isPrefix "error in proof state" msg
          then *)
          let
            val (def_names,_) =
              protocol_security_proof_defs false name prefix opt_defs opt_meth_level lthy
            val (prot_name,fp_name,smp_name) = case length def_names of
                0 => (prefix^"_protocol", prefix^"_fixpoint", prefix^"_SMP")
              | 1 => (nth def_names 0,    prefix^"_fixpoint", prefix^"_SMP")
              | 2 => (nth def_names 0,    nth def_names 1,    prefix^"_SMP")
              | _ => (nth def_names 0,    nth def_names 1,    nth def_names 2)
          in error (
            "Failed to prove the protocol secure.\n" ^
            "Click on the following to inspect which parts of the proof failed:\n" ^
            Active.sendback_markup_command (
              (if length def_names < 2
               then "— First compute a fixed-point\n" ^
                    "compute_fixpoint "^prot_name^" "^fp_name^"\n\n"
               else "")^
              "— Is the fixed point free of attack signals?\n" ^
              "value \"attack_notin_fixpoint "^fp_name^"\"\n\n" ^
              "— Is the protocol covered by the fixed point?\n" ^
              "value \"protocol_covered_by_fixpoint "^fp_name^" "^prot_name^"\"\n\n" ^
              "— Is the fixed point analyzed?\n" ^
```

```
            "value \"analyzed_fixpoint "^fp_name^"\"\n\n" ^
            "— Is the protocol well-formed?\n" ^
            (if length def_names < 3
             then "value \"wellformed_protocol "^prot_name^"\"\n\n"
             else "value \"wellformed_protocol' "^prot_name^" "^smp_name^"\"\n\n")^
            "— Is the fixed point well-formed?\n" ^
            "value \"wellformed_fixpoint "^fp_name^"\"") ^
          "\n\nOriginal error message:\n" ^ msg)
        end
        (* else error msg *)


    fun heuristic (params, (name, prefix, opt_defs), print, lthy) =
    let
      val method = case run_in_parallel ((("nbe",(name,prefix)),opt_defs), print, lthy) of
                     ("nbe",_) => "safe"
                     | ("nbe_coverage_rcv",_) => "safe_coverage_rcv"

    in
      protocol_security_proof(((method,(name,prefix)),opt_defs), print, lthy)
    end

end


val _ =
  Outer_Syntax.local_theory' command_keyword‹protocol_security_proof›
    "prove interpretation of secure protocol locale into global theory"
    (security_proof_locale_parser_with_method_choice >>
    (fn params => fn print => fn lthy =>
    let
        val ((opt_meth_level,(name,prefix)),opt_defs) = params
    in
      trac_time.ap_lthy lthy ("protocol_security_proof ("^name^")")
                     protocol_security_proof.protocol_security_proof_with_error_messages (params,
(name, prefix, opt_defs, opt_meth_level), print, lthy)
    end));

val _ =
  Outer_Syntax.local_theory' command_keyword‹protocol_security_proof_parallel›
    "prove interpretation of secure protocol locale into global theory"
    (security_proof_locale_parser_with_method_choice >>
    (fn params => fn print => fn lthy =>
    let
        val ((opt_meth_level,(name,prefix)),opt_defs) = params
    in
      trac_time.ap_lthy lthy ("protocol_security_proof ("^name^")")
                     protocol_security_proof.parallel (params, (name, prefix, opt_defs,
opt_meth_level), print, lthy)
    end));

val _ =
  Outer_Syntax.local_theory' command_keyword‹protocol_security_proof_safe_heuristic›
    "prove interpretation of secure protocol locale into global theory"
    (security_proof_locale_parser_with_method_choice >>
    (fn params => fn print => fn lthy =>
    let
        val ((opt_meth_level,(name,prefix)),opt_defs) = params
    in
      trac_time.ap_lthy lthy ("protocol_security_proof ("^name^")")
                     protocol_security_proof.heuristic (params, (name, prefix, opt_defs), print,
lthy)
    end));
```

```
val _ =
  Outer_Syntax.local_theory_to_proof' command_keyword‹manual_protocol_security_proof›
    "prove interpretation of secure protocol locale into global theory"
    (security_proof_locale_parser >> (fn params => fn print => fn lthy =>
    let
      val ((name,prefix),opt_defs) = params
      val (defs, proof_state) =
        protocol_security_proof_proof_state true name prefix opt_defs "safe" print lthy
      val subgoal_proof =
        let
          val m = "code_simp" (* case opt_meth_level of
              "safe" => "code_simp"
            | "nbe" => "normalization"
            | "eval" => "eval"
            | _ => error ("Invalid option: " ^ opt_meth_level) *)
        in
          "  subgoal by " ^ m ^ "\n"
        end
      val _ = Output.information ("Example proof:\n" ^
                Active.sendback_markup_command ("  apply check_protocol_intro\n"^
                                                subgoal_proof^
                                                (if length defs = 1 then ""
                                                 else subgoal_proof^
                                                      subgoal_proof^
                                                      subgoal_proof^
                                                      subgoal_proof)^
                                                 "  done\n"))
    in
      proof_state
    end
    ));

val _ =
  Outer_Syntax.local_theory' command_keyword‹protocol_composition_proof›
    "prove interpretation of composed protocol locale into global theory"
    (composed_protocol_locale_parser_with_method_choice >> (fn params => fn print => fn lthy =>
    let val ((_,(name,_)),_) = params
        fun protocol_composition_proof (params,lthy) =
      let
        val ((opt_meth_level,(name,prefix)),remaining_params) = params
        val proof_state =
              protocol_composition_proof_proof_state name prefix remaining_params print lthy
        val meth =
          let
            val m = select_proof_method_compositionality "use" opt_meth_level
            val _ = Output.information (
                        "Proving composability of protocol " ^ name ^ " with proof method " ^ m)
          in
            Method.Source (Token.make_src (m, Position.none) [])
          end
      in
        ml_isar_wrapper.prove_state_simple meth proof_state
      end
    in
      trac_time.ap_lthy lthy
        ("protocol_composition_proof ("^name^")")
        protocol_composition_proof (params,lthy)
    end));

val _ =
  Outer_Syntax.local_theory_to_proof' command_keyword‹manual_protocol_composition_proof›
```

```
    "prove interpretation of composed protocol locale into global theory"
    (composed_protocol_locale_parser >> (fn params => fn print => fn lthy =>
    let
      val ((name,prefix),remaining_params) = params
      val proof_state =
            protocol_composition_proof_proof_state name prefix remaining_params print lthy
      val subgoal_proof = "  subgoal by code_simp\n"
      val _ = Output.information ("Example proof:\n" ^
                Active.sendback_markup_command ("  apply check_protocol_intro\n"^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                "  done\n"))
    in
      proof_state
    end
    ));
›
```

ML‹

```
fun listterm_to_list' _    (Const ("List.list.Nil",tau)) = ([],tau)
  | listterm_to_list' lthy ((Const ("List.list.Cons",_) $ t1) $ t2) =
    let val (s,tau) = listterm_to_list' lthy t2
    in (t1::s,tau) end
  | listterm_to_list' lthy t =
      error ("Unexpected term (expected a list constructor): " ^ Syntax.string_of_term lthy t)

fun listterm_to_list lthy = fst o listterm_to_list' lthy

fun pairterm_to_pair' _    ((Const ("Product_Type.Pair",tau) $ x) $ y) = ((x,y),tau)
  | pairterm_to_pair' lthy t =
      error ("Error: Expected a pair term but got " ^ Syntax.string_of_term lthy t)

fun pairterm_to_pair lthy = fst o pairterm_to_pair' lthy

fun constrexpr_to_string lthy protocol t = let
  val trac = trac_definitorial_package.lookup_trac protocol lthy
  val trac_name = #name trac

  val sets_type_name = Local_Theory.full_name lthy (Binding.name (trac_name ^ "_sets"))
  val enum_type_name = Local_Theory.full_name lthy (Binding.name (trac_name ^ "_enum_consts"))
  val fun_type_name = Local_Theory.full_name lthy (Binding.name (trac_name ^ "_fun"))
  val atom_type_name = Local_Theory.full_name lthy (Binding.name (trac_name ^ "_atom"))

  fun print_const_expr x =
        if String.isPrefix sets_type_name x
        then String.extract (x,size sets_type_name+1,NONE)
        else if String.isPrefix enum_type_name x
        then String.extract (x,size enum_type_name+1,NONE)
        else if String.isPrefix fun_type_name x
        then String.extract (x,size fun_type_name+1,NONE)
        else if String.isPrefix atom_type_name x
        then String.extract (x,size atom_type_name+1,NONE)
        else error ("Unexpected constant expression: " ^ x)
in print_const_expr t end

fun setexpr_to_string lthy protocol t = let
  fun err msg t = error (msg ^ ": " ^ Syntax.string_of_term lthy t)

  fun print_set_expr' (Const (x,_)) = [constrexpr_to_string lthy protocol x]
```

```
    | print_set_expr' (t1 $ t2) = print_set_expr' t1@print_set_expr' t2
    | print_set_expr' t = err "Unexpected set expression subterm" t

  fun print_set_expr t =
    case print_set_expr' t of
      [x] => x
    | x::xs => x ^ "(" ^ String.concatWith "," xs ^ ")"
    | _ => err "Unexpected set expression" t
in print_set_expr t end

fun abstractionexpr_to_list lthy protocol t = let
  fun print_abs (Const ("Orderings.bot_class.bot",_)) = []
    | print_abs (t $ Const ("Orderings.bot_class.bot",_)) = print_abs t
    | print_abs (Const ("Set.insert",_) $ t) = [setexpr_to_string lthy protocol t]
    | print_abs (t1 $ t2) = print_abs t1@print_abs t2
    | print_abs t = error ("Unexpected abstract value expression: " ^ Syntax.string_of_term lthy t)
in print_abs t end

fun protterm_to_string_no_eval var_printer protocol protterm lthy = let
  fun print_raw (Const (x,_)) = "Const (" ^ x ^ ",_)"
    | print_raw (t $ s) = "(" ^ print_raw t ^  " $ " ^ print_raw s ^ ")"
    | print_raw _ = "_"

  fun err msg t = error (msg ^ ": " ^ Syntax.string_of_term lthy t ^ "\n" ^ print_raw t)
  val trac = trac_definitorial_package.lookup_trac protocol lthy
  val trac_name = #name trac
  val trac_fun_spec = Option.getOpt (#function_spec trac, {private = [], public = []})
  val is_priv_fun = member (fn (s,t) => s = #1 t) (#private trac_fun_spec)

  val fun_type_name = Local_Theory.full_name lthy (Binding.name (trac_name ^ "_fun"))

  val print_list = listterm_to_list lthy
  val print_const_expr = constrexpr_to_string lthy protocol
  val print_set_expr = setexpr_to_string lthy protocol

  fun print_abs t =
      let val a = abstractionexpr_to_list lthy protocol t
      in "val(" ^ (if a = [] then "0" else String.concatWith "," a) ^ ")" end

  fun print_trm (Const ("Term.term.Var",_) $ t
        ) = (case t of
              ((Const ("Product_Type.Pair",_) $ _) $ _) => var_printer (pairterm_to_pair lthy t)
            | _ => print_trm t)
    | print_trm ((Const ("Term.term.Fun",_) $ (Const ("Transactions.prot_fun.Abs",_) $ t)) $
                 Const ("List.list.Nil",_)
        ) = print_abs t
    | print_trm (
          (Const ("Term.term.Fun",_)$Const ("Transactions.prot_fun.OccursFact",_))
          $((Const ("List.list.Cons",_)
            $((Const ("Term.term.Fun",_)$Const ("Transactions.prot_fun.OccursSec",_))
             $Const ("List.list.Nil",_)))
           $((Const ("List.list.Cons",_) $ t) $ Const ("List.list.Nil",_)))
        ) = "occurs(" ^ print_trm t ^ ")"
    | print_trm (
          (Const ("Term.term.Fun",_) $ (Const ("Transactions.prot_fun.Attack",_) $ _)) $ _
        ) = "attack"
    | print_trm (
          (Const ("Term.term.Fun",_) $ (Const ("Transactions.prot_fun.Set",_) $ f)) $ ts
        ) = let val gs = map print_trm (print_list ts)
            in (case gs of
                  [] => print_set_expr f
                | xs => print_trm f ^ "(" ^ String.concatWith "," xs ^ ")")
            end
```

```
    | print_trm (
          (Const ("Term.term.Fun",_) $ (Const ("Transactions.prot_fun.Fu",_) $ f)) $ ts
        ) = let val g = print_trm f; val gs = map print_trm (print_list ts)
            in (case (if is_priv_fun g then (case gs of [] => [] | _::gs' => gs') else gs) of
                  [] => g
                | xs => g ^ "(" ^ String.concatWith "," xs ^ ")")
            end
    | print_trm (
          (Const ("Transactions.prot_atom.Atom",_) $ Const (t,_))
        ) = print_const_expr t
    | print_trm ((Const ("Product_Type.Pair",_) $ t1) $ t2) =
        "(" ^ print_abs t1 ^ "," ^ print_abs t2 ^ ")"
    | print_trm (Const (x,tx) $ Const (y,ty)) =
        if x = fun_type_name ^ ".enum"
        then print_const_expr y
        else err "Unexpected protocol/fixpoint term" (Const (x,tx) $ Const (y,ty))
    | print_trm (Const (x,_)) =
        if String.isPrefix "Transactions.prot_atom" x
        then String.extract (x,size "Transactions.prot_atom"+1,NONE)
        else print_const_expr x
    | print_trm t = err "Unexpected protocol/fixpoint term" t;
in
  print_trm protterm
end

fun prottermtype_to_string_no_eval var_printer protocol protterm lthy =
  protterm_to_string_no_eval var_printer protocol protterm lthy

fun fixpoint_to_string protocol fixpoint lthy = let
  fun err msg t = error (msg ^ ": " ^ Syntax.string_of_term lthy t)
  val fpterm = "let (FP,_,TI) = (" ^ fixpoint ^ ") in (FP,TI)"

  fun print_fp' s = protterm_to_string_no_eval
          (fn _ => error "Unexpected term variable in fixpoint")
          protocol s lthy

  fun print_fp ((Const ("Product_Type.Pair",_) $ t1) $ t2) =
        String.concatWith "\n" (map print_fp' (listterm_to_list lthy t1)@
                                map (fn x => "timplies" ^ print_fp' x) (listterm_to_list lthy t2))
    | print_fp t = err "Unexpected fixpoint term pair" t;
in
  (print_fp o eval_term lthy o Syntax.read_term lthy) fpterm
end

fun transaction_label_to_string t lthy = let
    fun err msg t = error (msg ^ ": " ^ Syntax.string_of_term lthy t)

    fun print_label (Const ("Labeled_Strands.strand_label.LabelN",_) $ _) = " "
          (* Syntax.string_of_term lthy t *)
      | print_label (Const ("Labeled_Strands.strand_label.LabelS",_)) = "*"
      | print_label t = err "Unexpected action label term" t
  in print_label t end

fun transaction_action_to_string var_printer protocol t lthy = let
    fun err msg t = error (msg ^ ": " ^ Syntax.string_of_term lthy t)
    fun print_trm s = protterm_to_string_no_eval var_printer protocol s lthy

    fun print_action (Const ("Stateful_Strands.stateful_strand_step.Send",_) $ ts
          ) = "send " ^ String.concatWith ", " (map print_trm (listterm_to_list lthy ts))
      | print_action (Const ("Stateful_Strands.stateful_strand_step.Receive",_) $ ts
          ) = "receive " ^ String.concatWith ", " (map print_trm (listterm_to_list lthy ts))
      | print_action (((Const ("Stateful_Strands.stateful_strand_step.Equality",_) $ _) $ t1) $ t2
          ) = print_trm t1 ^ " == " ^ print_trm t2
```

```
      | print_action (((Const ("Stateful_Strands.stateful_strand_step.InSet",_) $ _) $ t1) $ t2
          ) = print_trm t1 ^ " in " ^ print_trm t2
      | print_action ((Const ("Stateful_Strands.stateful_strand_step.Insert",_) $ t1) $ t2
          ) = "insert " ^ print_trm t1 ^ " " ^ print_trm t2
      | print_action ((Const ("Stateful_Strands.stateful_strand_step.Delete",_) $ t1) $ t2
          ) = "delete " ^ print_trm t1 ^ " " ^ print_trm t2
      | print_action (((Const ("Stateful_Strands.stateful_strand_step.NegChecks",_) $ xs) $ ts1) $ ts2
          ) = let fun f (a,b) = print_trm a ^ " != " ^ print_trm b
                  fun g (a,b) = print_trm a ^ " notin " ^ print_trm b
                  val ys = map print_trm (listterm_to_list lthy xs)
                  val ss1 = map (f o pairterm_to_pair lthy) (listterm_to_list lthy ts1)
                  val ss2 = map (g o pairterm_to_pair lthy) (listterm_to_list lthy ts2)
              in (if ys = [] then "" else "forall " ^ String.concatWith ", " ys ^ " ") ^
                 String.concatWith " or " (ss1@ss2)
              end
      | print_action t = err "Unexpected transaction action term" t
  in print_action t end

fun transaction_labeled_action_to_string var_printer protocol t lthy = let
    fun err msg t = error (msg ^ ": " ^ Syntax.string_of_term lthy t)

    fun print_laction ((Const ("Product_Type.Pair",_) $ t1) $ t2) =
          transaction_label_to_string t1 lthy ^ " " ^
          transaction_action_to_string var_printer protocol t2 lthy
      | print_laction t = err "Unexpected labeled transaction action term" t
  in print_laction t end

fun transaction_to_string var_printer protocol transactionterm lthy = let
  val evaltrm = eval_term lthy o Syntax.read_term lthy

  val trfresh   = "Transactions.transaction_fresh (" ^ transactionterm ^ ")"
  val trreceive = "Transactions.transaction_receive (" ^ transactionterm ^ ")"
  val trchecks  = "Transactions.transaction_checks (" ^ transactionterm ^ ")"
  val trupdates = "Transactions.transaction_updates (" ^ transactionterm ^ ")"
  val trsend    = "Transactions.transaction_send (" ^ transactionterm ^ ")"

  fun print_fresh xs =
        if xs = [] then []
        else ["  new " ^ String.concatWith ", " (map (var_printer o pairterm_to_pair lthy) xs)]

  fun print_tr' s = transaction_labeled_action_to_string var_printer protocol s lthy

  val print_tr =
    String.concatWith "\n" (
      (map print_tr' o listterm_to_list lthy o evaltrm) trreceive@
      (map print_tr' o listterm_to_list lthy o evaltrm) trchecks@
      (print_fresh   o listterm_to_list lthy o evaltrm) trfresh@
      (map print_tr' o listterm_to_list lthy o evaltrm) trupdates@
      (map print_tr' o listterm_to_list lthy o evaltrm) trsend)
in
  print_tr
end

fun transaction_list_to_string var_printer protocol transactionlistterm lthy = let
  fun err msg t = error (msg ^ ": " ^ Syntax.string_of_term lthy t)
  val evaltrm = eval_term lthy o Syntax.read_term lthy

  fun print_fresh i xs =
        if xs = [] then []
        else ["  new " ^ String.concatWith ", " (map (var_printer i o pairterm_to_pair lthy) xs)]

  fun print_tr' i s = transaction_labeled_action_to_string (var_printer i) protocol s lthy
```

```
  fun print_tr i ((Const ("Product_Type.Pair",_) $ trfresh) $
                 ((Const ("Product_Type.Pair",_) $ trreceive) $
                  ((Const ("Product_Type.Pair",_) $ trchecks) $
                   ((Const ("Product_Type.Pair",_) $ trupdates) $ trsend)))
        ) = String.concatWith "\n" (
          (map (print_tr' i) o listterm_to_list lthy) trreceive@
          (map (print_tr' i) o listterm_to_list lthy) trchecks@
          (print_fresh i     o listterm_to_list lthy) trfresh@
          (map (print_tr' i) o listterm_to_list lthy) trupdates@
          (map (print_tr' i) o listterm_to_list lthy) trsend)
    | print_tr _ t = err "Unexpected term" t

  fun print_trs trs = String.concatWith "\n\n" (map_index (fn (i,x) => print_tr i x) trs)

  fun trfresh s   = "Transactions.transaction_fresh (" ^ s ^ ")"
  fun trreceive s = "Transactions.transaction_receive (" ^ s ^ ")"
  fun trchecks s  = "Transactions.transaction_checks (" ^ s ^ ")"
  fun trupdates s = "Transactions.transaction_updates (" ^ s ^ ")"
  fun trsend s    = "Transactions.transaction_send (" ^ s ^ ")"

  val f = "let f = λX. (" ^ trfresh "X" ^ ", " ^ trreceive "X" ^ ", " ^ trchecks "X" ^ ", "
                               ^ trupdates "X" ^ ", " ^ trsend "X" ^ ")" ^
          "in map f (" ^ transactionlistterm ^ ")"

  val transactiontermlist = listterm_to_list lthy (evaltrm f)
in
  print_trs transactiontermlist
end

val _ = Outer_Syntax.local_theory' @{command_keyword "print_transaction_strand"}
        "print protocol transaction as transaction strand"
        (Parse.name -- Parse.name >>
        (fn (protocol, transaction) => fn print => fn lthy =>
          let fun print_tr ((protocol,transaction), _, lthy) =
            let
              val _ = assert_defined lthy transaction
              fun f a = protterm_to_string_no_eval (fn _ => error "Unexpected variable")
                                                   protocol a lthy
              fun g (a,b) =
                  if f a = "Value" orelse f a = "value"
                  then "X" ^ Syntax.string_of_term lthy b
                  else "Y[" ^ f a ^ ", " ^ Syntax.string_of_term lthy b ^ "]"
              val _ = Output.information (transaction_to_string g protocol transaction lthy)
            in
              lthy
            end
          in
            trac_time.ap_lthy lthy
              ("print_transaction_strand ("^protocol^")")
              print_tr
              ((protocol,transaction), print, lthy)
          end ));


val _ = Outer_Syntax.local_theory' @{command_keyword "print_transaction_strand_list"}
        "print protocol transaction list as transaction strand list"
        (Parse.name -- Parse.name >>
        (fn (protocol, transaction_list) => fn print => fn lthy =>
          let fun print_tr ((protocol,transaction_list), _, lthy) =
            let
              val _ = assert_defined lthy transaction_list
              fun f a = protterm_to_string_no_eval (fn _ => error "Unexpected variable")
                                                   protocol a lthy
```

```
          fun g i (a,b) =
              if f a = "Value" orelse f a = "value"
              then "X" ^ Syntax.string_of_term lthy b ^ "_" ^ Int.toString i
              else "Y[" ^ f a ^ ", " ^ Syntax.string_of_term lthy b ^ "_" ^ Int.toString i ^ "]"
          val _ = Output.information (transaction_list_to_string g protocol transaction_list lthy)
        in
          lthy
        end
      in
        trac_time.ap_lthy lthy
          ("print_transaction_strand_list ("^protocol^")")
          print_tr
          ((protocol,transaction_list), print, lthy)
      end ));

val _ = Outer_Syntax.local_theory' @{command_keyword "print_attack_trace"}
        "print attack trace"
        (Parse.name -- Parse.name -- Parse.name >>
        (fn ((protocol, protocol_def), attack_trace) => fn print => fn lthy =>
          let fun print_tr ((protocol,protocol_def,attack_trace), _, lthy) =
            let
              val evaltrm = eval_term lthy o Syntax.read_term lthy
              val _ = assert_defined lthy protocol_def
              val _ = assert_defined lthy attack_trace
              fun f i b = "X" ^ b ^ "_" ^ Int.toString i
              fun g i (_,b) = f i (Syntax.string_of_term lthy b)
              val t1 = "map (λ(i,_). add_occurs_msgs (" ^ protocol_def ^ " ! i)) " ^ attack_trace
              val t2 = "map (map (λ((_,i),xs). (i,xs))) (map snd " ^ attack_trace ^ ")"
              val s = t2 |> evaltrm
                         |> listterm_to_list lthy
                         |> map (listterm_to_list lthy)
                         |> map (map (pairterm_to_pair lthy))
                         |> map (map (fn (a,b) => (Syntax.string_of_term lthy a,
                                                   abstractionexpr_to_list lthy protocol b)))
                         |> map_index (fn (i,ts) =>
                              map (fn (a,xs) => f i a ^ ": {" ^ String.concatWith ", " xs ^ "}") ts)
                         |> List.concat
              val u = transaction_list_to_string g protocol t1 lthy
              val _ = Output.information (
                        (if s <> []
                         then "Abstractions:\n" ^ String.concatWith "\n" s ^ "\n\n"
                         else "") ^
                        ("Attack trace:\n" ^ u))
            in
              lthy
            end
          in
            trac_time.ap_lthy lthy
              ("print_attack_trace ("^protocol^","^protocol_def^","^attack_trace^")")
              print_tr
              ((protocol,protocol_def,attack_trace), print, lthy)
          end ));

val _ = Outer_Syntax.local_theory' @{command_keyword "print_fixpoint"}
        "print protocol fixpoint"
        (Parse.name -- Parse.name >>
        (fn (protocol, fixpoint) => fn print => fn lthy =>
          let fun print_fixpoint ((protocol,fixpoint), _, lthy) =
            let
              val _ = assert_defined lthy fixpoint
              val _ = Output.information (fixpoint_to_string protocol fixpoint lthy)
            in
              lthy
```

```
          end
        in
          trac_time.ap_lthy lthy ("print_fixpoint ("^protocol^")") print_fixpoint
((protocol,fixpoint), print, lthy)
        end ));

  val _ = Outer_Syntax.local_theory' @{command_keyword "save_fixpoint"}
        "Write fixpoint to file."
        ((Parse.name -- Parse.name -- Parse.name >> (
          fn ((protocol_name, fixpoint_filename), fixpoint_name) => fn _ => fn lthy =>
        let
          fun save_fixpoint ((protocol_name, fixpoint_name), fixpoint_filename, lthy) =
            let
              val _ = assert_defined lthy fixpoint_name
              fun write f s =
                    if File.exists f
                    then error ("Error: Cannot write to file: File already exists")
                    else File.write f s
              val filename =
                    Path.explode (
                      trac.mk_abs_filename (Proof_Context.theory_of lthy) fixpoint_filename)
              val _ = Output.information ("Evaluating fixed-point term " ^ fixpoint_name)
              val fp_str = fixpoint_to_string protocol_name fixpoint_name lthy
              val _ = Output.information (
                        "Writing fixed point to file " ^ Path.print filename)
              val _ = write filename fp_str
            in
              lthy
            end
        in
          trac_time.ap_lthy lthy ("save_fixpoint") save_fixpoint ((protocol_name, fixpoint_name),
fixpoint_filename, lthy)
        end)));


  (* TODO: move? *)
  (* TODO: check that chunks are not defined before defining them *)
  fun eval_define_declare_fixpoint chunk_size chunk_suffix (name, t) print lthy = let
    val mk_tuple = trac_definitorial_package.mk_tuple
    val mk_list = trac_definitorial_package.mk_list
    val mk_listT = trac_definitorial_package.mk_listT
    val full_name = trac_definitorial_package.full_name

    fun chunk_name n = name ^ chunk_suffix ^ Int.toString n

    fun arg name t =
      ((Binding.name name, NoSyn), ((Binding.name (name ^ "_def"),@{attributes [code]}), t))

    fun def_trm (name,trm) lthy = snd (Local_Theory.define (arg name trm) lthy)

    fun append_term tau = Term.Const (@{const_name "append"}, [tau,tau] ---> tau)
    fun nil_term tau = Term.Const (@{const_name "Nil"}, tau)

    val ((fp,fp_elemT),(occ,occ_elemT),(ti,ti_elemT)) =
      let fun chop (l,Type (_,[tau])) = (chop_groups chunk_size l, tau)
            | chop (_,tau) = error ("Expected type with one type-parameter but got " ^
                                    Syntax.string_of_typ lthy tau)
          val ltl = chop o listterm_to_list' lthy
          val (fp,occ_ti) = pairterm_to_pair lthy (eval_term lthy t)
          val (occ,ti) = pairterm_to_pair lthy occ_ti
      in (ltl fp, ltl occ, ltl ti) end

    fun mk_chunk_trm ch tau lthy = Term.Const (full_name ch lthy, mk_listT tau)
```

```
  datatype ChunksVariant =
    NoChunks | SingleChunk of term | MultipleChunks of (bstring * term) list

  fun chunk [] _ _ = NoChunks
    | chunk [ch] tau _ = SingleChunk (mk_list tau ch)
    | chunk trms tau i =
        MultipleChunks (map_index (fn (j,ch) => (chunk_name (i+j), mk_list tau ch)) trms)

  fun append_chunks NoChunks tau _ = mk_list tau []
    | append_chunks (SingleChunk ch) _ _ = ch
    | append_chunks (MultipleChunks chs) tau lthy = let
        fun f [] = nil_term (mk_listT tau)
          | f [ch] = mk_chunk_trm ch tau lthy
          | f (ch::chs) = append_term (mk_listT tau) $ mk_chunk_trm ch tau lthy $ f chs
      in f (map fst chs) end

  fun chunks_len (MultipleChunks chs) = length chs
    | chunks_len _ = 0

  fun def_chunks (MultipleChunks chs) lthy = fold def_trm chs lthy
    | def_chunks _ lthy = lthy (* we don't use chunks when there would be at most one *)

  val fp_chunks_trms  = chunk fp  fp_elemT  0
  val occ_chunks_trms = chunk occ occ_elemT (chunks_len fp_chunks_trms)
  val ti_chunks_trms  = chunk ti  ti_elemT  (chunks_len fp_chunks_trms+chunks_len occ_chunks_trms)

  val lthy = snd (Local_Theory.begin_nested lthy)
  val lthy = def_chunks fp_chunks_trms lthy
  val lthy = def_chunks occ_chunks_trms lthy
  val lthy = def_chunks ti_chunks_trms lthy
  val lthy = Local_Theory.end_nested lthy

  val lthy = snd (Local_Theory.begin_nested lthy)

  val fpt_trm = mk_tuple [
    append_chunks fp_chunks_trms fp_elemT lthy,
    append_chunks occ_chunks_trms occ_elemT lthy,
    append_chunks ti_chunks_trms ti_elemT lthy]

  val lthy = def_trm (name, fpt_trm) lthy
in
  (fpt_trm, Local_Theory.end_nested lthy)
end

(* TODO: chunks *)
val _ = Outer_Syntax.local_theory' @{command_keyword "load_fixpoint"}
        "Import fixpoint from file."
        ((Parse.name -- Parse.name -- Parse.name >> (
          fn ((protocol_name, fixpoint_filename), fixpoint_name) => fn print => fn lthy =>
        let
          fun load_fixpoint ((protocol_name, fixpoint_filename), fixpoint_name, lthy) =
            let
              val _ = assert_not_defined lthy fixpoint_name
              val filename =
                    Path.explode (
                      trac.mk_abs_filename (Proof_Context.theory_of lthy) fixpoint_filename)
              val _ = Output.information (
                        "Reading fixed point from file " ^ Path.print filename)
              fun read f =
                    if File.exists f
                    then File.read f
                    else error ("Error: Cannot read file: File does not exist")
```

```
              val fp_str = TracFpParser.parse_str (read filename)
              val trac = trac_definitorial_package.lookup_trac protocol_name lthy
              val cert_fp = TracProtocolCert.certify_fixpoint trac fp_str
              val cert_trac = TracProtocolCert.certifyProtocol trac
              val fp_trm = trac_definitorial_package.fp_triple_to_hol cert_fp cert_trac lthy
            in
              #2 (ml_isar_wrapper.define_constant_definition' (fixpoint_name, fp_trm) print lthy)
            end
        in
          trac_time.ap_lthy lthy ("load_fixpoint") load_fixpoint ((protocol_name,
fixpoint_filename), fixpoint_name, lthy)
        end)));

val _ = Outer_Syntax.local_theory' @{command_keyword "compute_fixpoint"}
        "evaluate and define protocol fixpoint"
        ((Scan.option (keyword<[> |-- Parse.name --| keyword<]>)) --
         Parse.name -- Parse.name -- Scan.option Parse.name >>
        (fn (((opt, protocol), fixpoint), opt_trace) => fn print => fn lthy =>
          let fun compute_fixpoint (((protocol,fixpoint),opt_trace), print, lthy) =
            let
              val _ = assert_defined lthy protocol
              val _ = assert_not_defined lthy fixpoint
              val _ = Option.app (assert_not_defined lthy) opt_trace
              val _ = Output.information ("Computing a fixed point for protocol " ^ protocol)
              val _ = case opt of (* TODO: don't compute the fixpoint twice *)
                        NONE => ()
                      | SOME "check_for_attacks" =>
                        let val no_attack = eval_term lthy (Syntax.read_term lthy (
                              "(attack_notin_fixpoint ∘ compute_fixpoint_fun) " ^ protocol))
                            val _ = case no_attack of
                                Term.Const ("HOL.True", _) =>
                                Output.information "The fixed point is free of attack signals."
                              | Term.Const ("HOL.False", _) =>
                                Output.information "The fixed point contains an attack signal."
                              | _ => error ("Error: Unexpected term: " ^ @{make_string} no_attack)
                        in () end
                      | SOME opt => error ("Error: Invalid option " ^ opt)
              val fp = (fixpoint, Syntax.read_term lthy ("compute_fixpoint_fun " ^ protocol))
              val opt_tr = Option.map (* TODO: don't compute the fixpoint twice *)
                            (fn trace =>
                              (trace, Syntax.read_term lthy
                                        ("(compute_reduced_attack_trace " ^ protocol ^
                                         " ∘ snd ∘ compute_fixpoint_with_trace) " ^ protocol)))
                            opt_trace
            in
              ((snd o eval_define_declare_fixpoint 15 "_chunk" fp print) lthy |>
               (fn lthy => case opt_tr of
                              SOME tr => (snd o eval_define_declare tr print) lthy
                            | NONE => lthy))
              handle
                ERROR msg =>
                  let
                    val _ = warning ("Failed to compute the set with eval. Retrying with NBE.\n" ^
                                     "Original error message:\n" ^ msg)
                  in
                    ((snd o eval_define_declare_nbe fp print) lthy |>
                     (fn lthy => case opt_tr of
                                    SOME tr => (snd o eval_define_declare_nbe tr print) lthy
                                  | NONE => lthy))
                  end

            end
          in
```

```
          trac_time.ap_lthy lthy ("compute_fixpoint ("^protocol^")") compute_fixpoint
(((protocol,fixpoint),opt_trace), print, lthy)
        end ));

val _ = Outer_Syntax.local_theory' @{command_keyword "compute_SMP"}
      "evaluate and define a finite representation of the sub-message patterns of a protocol"
      ((Scan.optional (keyword‹[› |-- Parse.name --| keyword‹]›) "no_optimizations") --
        Parse.name -- Parse.name >> (fn ((opt,prot), smp) => fn print => fn lthy =>
        let fun compute_smp (((opt, prot), smp), print, lthy) =
        let
          val prot' = prot
          val rmd = "List.remdups"
          val f = "Stateful_Strands.trms_list_sst"
          val g =
            "(λT. " ^ f ^ " T@map (pair' prot_fun.Pair) (Stateful_Strands.setops_list_sst T))"
          fun s trms =
            "(" ^ rmd ^ " (List.concat (List.map (" ^ trms ^
            " ∘ Labeled_Strands.unlabel ∘ transaction_strand) " ^ prot' ^ ")))"
          val opt1 = "remove_superfluous_terms Γ"
          val opt2 = "generalize_terms Γ is_Var"
          val gsmp_opt =
            "generalize_terms Γ (λt. is_Var t ∧ t ≠ TAtom AttackType ∧ " ^
            "t ≠ TAtom SetType ∧ t ≠ TAtom OccursSecType ∧ ¬is_Atom (the_Var t))"
          val smp_fun = "SMP0 Ana Γ"
          fun smp_fun' opts =
            "(λT. let T' = (" ^ rmd ^ " ∘ " ^ opts ^ " ∘ " ^ smp_fun ^
            ") T in List.map (λt. t · Typed_Model.var_rename (Typed_Model.max_var_set " ^
            "(Messages.fv_set (set (T@T'))))) T')"
          val cmd =
            if opt = "no_optimizations" then smp_fun ^ " " ^ s f
            else if opt = "optimized"
            then smp_fun' (opt1 ^ " ∘ " ^ opt2) ^ " " ^ s f
            else if opt = "GSMP"
            then smp_fun' (opt1 ^ " ∘ " ^ gsmp_opt) ^ " " ^ s g
            else if opt = "composition"
            then smp_fun ^ " " ^ s g
            else if opt = "composition_optimized"
            then smp_fun' (opt1 ^ " ∘ " ^ opt2) ^ " " ^ s g
            else error ("Error: Invalid option: " ^ opt ^ "\n\nValid options:\n" ^
                  "1. no_optimizations: Computes the finite SMP representation set " ^
                  "without any optimizations (this is the default setting).\n" ^
                  "2. optimized: Applies optimizations to reduce the size of the computed " ^
                  "set, but this might not be sound.\n" ^
                  "3. GSMP: Computes a set suitable for use in checking GSMP disjointness.\n" ^
                  "4. composition: Computes a set suitable for checking type-flaw resistance " ^
                  "of composed protocols.\n" ^
                  "5. composition_optimized: An optimized variant of the previous setting.")
          val _ = assert_defined lthy prot
          val _ = assert_not_defined lthy smp
          val _ = Output.information (
                    "Computing a finite SMP representation set for protocol " ^ prot)
        in
          (snd o eval_define_declare (smp, Syntax.read_term lthy cmd) print) lthy
          handle
            ERROR msg =>
              let
                val _ = warning ("Failed to compute the set with eval. Retrying with NBE.\n" ^
                                 "Original error message:\n" ^ msg)
              in
                (snd o eval_define_declare_nbe (smp, Syntax.read_term lthy cmd) print) lthy
              end
        end
      in
```

```
         trac_time.ap_lthy lthy ("compute_SMP ("^prot^")") compute_smp (((opt, prot), smp), print,
lthy)
        end));

val _ = Outer_Syntax.local_theory' @{command_keyword "compute_shared_secrets"}
        "evaluate and define a finite representation of shared secrets as the intersection of GSMP
sets"
        (Scan.repeat1 Parse.name >> (fn params => fn print => fn lthy =>
          let fun compute_shared_secrets (params, print, lthy) =
          let
            val _ = if length params < 3 then error "Not enough arguments" else ()
            val (gsmps, sec) = split_last params
            val xs = "xs"
            val cmd =
              "let " ^ xs ^ " = [" ^ String.concatWith ", " gsmps ^ "] in " ^
              "(" ^
                (* "remove_superfluous_terms Γ ∘ generalize_terms Γ ((=) (TAtom SetType)) ∘ " ^ *)
                "remove_superfluous_terms Γ ∘ " ^
                "(" ^
                  "concat ∘ map " ^
                  "(λp. filter " ^
                    "(λt. list_ex (λs. Γ t = Γ s ∧ mgu t s ≠ None) (" ^ xs ^ " ! snd p))" ^
                    "(" ^ xs ^ " ! fst p)" ^
                  ")" ^
                ")" ^
              ") (" ^
                "filter (λp. fst p ≠ snd p) ((λp. List.product p p) [0..<length " ^ xs ^ "])" ^
              ")"

            val _ = map (assert_defined lthy) gsmps
            val _ = assert_not_defined lthy sec
            val _ = Output.information (
                      "Computing a finite representation of the shared secrets for the " ^
                      "protocols with GSMP sets " ^ String.concatWith ", " gsmps)
          in
            (snd o eval_define_declare (sec, Syntax.read_term lthy cmd) print) lthy
          end
        in
         trac_time.ap_lthy lthy
          ("compute_shared_secrets (["^String.concatWith ", " params^"])")
          compute_shared_secrets (params, print, lthy)
        end));

val _ = Outer_Syntax.local_theory' @{command_keyword "setup_protocol_checks"}
        "setup protocol checks"
        (Parse.name -- Scan.repeat Parse.name >>
        (fn params => fn print => fn lthy =>
        let fun setup_protocol_checks ((protocol_model, protocol_names), print, lthy) =
          let
            fun f s = protocol_model ^ "." ^ s
            val a1 = "protocol_check_intro_lemmata"
            val a2 = "coverage_check_unfold_lemmata"
            val a3 = "coverage_check_unfold_protocol_lemma"
            val a4 = "protocol_checks'"
          in
            (declare_protocol_checks print #>
             declare_thm_attr a1 (f "protocol_covered_by_fixpoint_intros") print #>
             declare_def_attr a2 (f "protocol_covered_by_fixpoint") print #>
             fold (fn s => declare_def_attr a3 s print) protocol_names #>
             declare_def_attr a4 (f "wellformed_fixpoint") print #>
             declare_def_attr a4 (f "wellformed_protocol") print #>
             declare_def_attr a4 (f "wellformed_protocol'") print #>
             declare_def_attr a4 (f "wellformed_protocol''") print #>
```

```
          declare_def_attr a4 (f "composable_protocols") print) lthy
        end
      in
        trac_time.ap_lthy lthy
          ("setup_protocol_checks (("^fst params^",["^String.concatWith "," (snd params)^"]))")
          setup_protocol_checks (params, print, lthy)
      end ));
›
```

end

5 Examples

5.1 The Keyserver Protocol

```
theory Keyserver
  imports "../PSPSP"
begin

declare [[pspsp_timing]]

trac‹
Protocol: keyserver

Enumerations:
honest = {a,b,c}
server = {s}
agents = honest ++ server

Sets:
ring/1 valid/2 revoked/2

Functions:
Public sign/2 crypt/2 pair/2
Private inv/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
pair(X,Y) -> X,Y

Transactions:
# Out-of-band registration
outOfBand(A:honest,S:server)
  new NPK
  insert NPK ring(A)
  insert NPK valid(A,S)
  send NPK.

# User update key
keyUpdateUser(A:honest,PK:value)
  PK in ring(A)
  new NPK
  delete PK ring(A)
  insert NPK ring(A)
  send sign(inv(PK),pair(A,NPK)).

# Server update key
keyUpdateServer(A:honest,S:server,PK:value,NPK:value)
  receive sign(inv(PK),pair(A,NPK))
  PK in valid(A,S)
  NPK notin valid(_)
  NPK notin revoked(_)
  delete PK valid(A,S)
  insert PK revoked(A,S)
  insert NPK valid(A,S)
  send inv(PK).
```

```
# Attack definition
authAttack(A:honest,S:server,PK:value)
  receive inv(PK)
  PK in valid(A,S)
  attack.
›‹
val(intruderValues)
val(ring(A)) where A:honest
sign(inv(val(0)),pair(A,val(ring(A)))) where A:honest
inv(val(revoked(A,S))) where A:honest S:server
pair(A,val(ring(A))) where A:honest

occurs(val(intruderValues))
occurs(val(ring(A))) where A:honest

timplies(val(ring(A)),val(ring(A),valid(A,S))) where A:honest S:server
timplies(val(ring(A)),val(0)) where A:honest
timplies(val(ring(A),valid(A,S)),val(valid(A,S))) where A:honest S:server
timplies(val(0),val(valid(A,S))) where A:honest S:server
timplies(val(valid(A,S)),val(revoked(A,S))) where A:honest S:server
›
```

5.1.1 Proof of security

protocol_model_setup *spm: keyserver*

compute_SMP *[optimized] keyserver_protocol keyserver_SMP*
manual_protocol_security_proof *ssp: keyserver*
  for *keyserver_protocol keyserver_fixpoint keyserver_SMP*
  apply *check_protocol_intro*
  subgoal by *(timeit code_simp)*
  subgoal by *(timeit code_simp)*
  subgoal by *(timeit code_simp)*
  subgoal by *(timeit code_simp)*
  subgoal by *(timeit code_simp)*
  done

end

5.2 A Variant of the Keyserver Protocol

theory *Keyserver2*
  imports *"../PSPSP"*
begin

declare *[[pspsp_timing]]*

trac‹
```
Protocol: keyserver2

Enumerations:
honest = {a,b,c}
dishonest = {i}
agent = honest ++ dishonest

Sets:
ring'/1 seen/1 pubkeys/0 valid/1

Functions:
Public h/1 sign/2 crypt/2 scrypt/2 pair/2 update/3
Private inv/1 pw/1
```

```
Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
scrypt(X,Y) ? X -> Y
pair(X,Y) -> X,Y
update(X,Y,Z) -> X,Y,Z

Transactions:
passwordGenD(A:dishonest)
  send pw(A).

pubkeysGen()
  new PK
  insert PK pubkeys
  send PK.

updateKeyPw(A:honest,PK:value)
  PK in pubkeys
  new NPK
  insert NPK ring'(A)
  send NPK
  send crypt(PK,update(A,NPK,pw(A))).

updateKeyServerPw(A:agent,PK:value,NPK:value)
  receive crypt(PK,update(A,NPK,pw(A)))
  PK in pubkeys
  NPK notin pubkeys
  NPK notin seen(_)
  insert NPK valid(A)
  insert NPK seen(A).

authAttack2(A:honest,PK:value)
  receive inv(PK)
  PK in valid(A)
  attack.
›
```

5.2.1 Proof of security

protocol_model_setup *spm: keyserver2*
compute_fixpoint *keyserver2_protocol keyserver2_fixpoint*
protocol_security_proof *ssp: keyserver2*

5.2.2 The generated theorems and definitions

thm *ssp.protocol_secure*

thm *keyserver2_enum_consts.nchotomy*
thm *keyserver2_sets.nchotomy*
thm *keyserver2_fun.nchotomy*
thm *keyserver2_atom.nchotomy*
thm *keyserver2_arity.simps*
thm *keyserver2_public.simps*
thm *keyserver2_Γ.simps*
thm *keyserver2_Ana.simps*

thm *keyserver2_transaction_passwordGenD_def*
thm *keyserver2_transaction_pubkeysGen_def*
thm *keyserver2_transaction_updateKeyPw_def*
thm *keyserver2_transaction_updateKeyServerPw_def*
thm *keyserver2_transaction_authAttack2_def*
thm *keyserver2_protocol_def*

thm *keyserver2_fixpoint_def*

end

5.3 The Composition of the Two Keyserver Protocols

theory *Keyserver_Composition*
  imports *"../PSPSP"*
begin

declare *[[pspsp_timing]]*

trac‹
```
Protocol: kscomp

Enumerations:
honest = {a,b,c}
dishonest = {i}
agent = honest ++ dishonest

Sets:
ring/1 valid/1 revoked/1 deleted/1
ring'/1 seen/1 pubkeys/0

Functions:
Public h/1 sign/2 crypt/2 scrypt/2 pair/2 update/3
Private inv/1 pw/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
scrypt(X,Y) ? X -> Y
pair(X,Y) -> X,Y
update(X,Y,Z) -> X,Y,Z


### The signature-based keyserver protocol
Transactions of p1:
intruderGen()
  new PK
* send PK, inv(PK).

outOfBand(A:honest)
  new PK
  insert PK ring(A)
* insert PK valid(A)
* send PK.

oufOfBandD(A:dishonest)
  new PK
* insert PK valid(A)
* send PK, inv(PK).

updateKey(A:honest,PK:value)
  PK in ring(A)
  new NPK
  delete PK ring(A)
  insert PK deleted(A)
  insert NPK ring(A)
  send sign(inv(PK),pair(A,NPK)).

updateKeyServer(A:agent,PK:value,NPK:value)
```

```
  receive sign(inv(PK),pair(A,NPK))
* PK in valid(A)
* NPK notin valid(_)
  NPK notin revoked(_)
* delete PK valid(A)
  insert PK revoked(A)
* insert NPK valid(A)
* send inv(PK).

authAttack(A:honest,PK:value)
  receive inv(PK)
* PK in valid(A)
  attack.


### The password-based keyserver protocol
Transactions of p2:
intruderGen'()
  new PK
* send PK, inv(PK).

passwordGenD(A:dishonest)
  send pw(A).

pubkeysGen()
  new PK
  insert PK pubkeys
* send PK.

updateKeyPw(A:honest,PK:value)
  PK in pubkeys
  new NPK
# NOTE: The ring' sets are not used elsewhere, but we have to avoid that the fresh keys generated
#       by this rule are abstracted to the empty abstraction, and so we insert them into a ring'
#       set. Otherwise the two protocols would have too many abstractions in common (in particular,
#       the empty abstraction) which leads to false attacks in the composed protocol (probably
#       because the term implication graphs of the two protocols then become 'linked' through the
#       empty abstraction)
  insert NPK ring'(A)
* send NPK
  send crypt(PK,update(A,NPK,pw(A))).

updateKeyServerPw(A:agent,PK:value,NPK:value)
  receive crypt(PK,update(A,NPK,pw(A)))
  PK in pubkeys
  NPK notin pubkeys
  NPK notin seen(_)
* insert NPK valid(A)
  insert NPK seen(A).

authAttack2(A:honest,PK:value)
  receive inv(PK)
* PK in valid(A)
  attack.
›
```

5.3.1 Proof: The composition of the two keyserver protocols is secure

protocol_model_setup *spm: kscomp*
setup_protocol_checks *spm kscomp_protocol kscomp_protocol_p1 kscomp_protocol_p2*
compute_fixpoint *kscomp_protocol kscomp_fixpoint*
manual_protocol_security_proof *ssp: kscomp*
  for *kscomp_protocol kscomp_fixpoint*

```
  apply check_protocol_intro
  subgoal by (timeit code_simp)
  subgoal
    apply coverage_check_intro
    subgoal by (timeit code_simp)
    subgoal by (timeit code_simp)
    subgoal by (timeit code_simp)
    subgoal by (timeit normalization)
    subgoal by (timeit eval)
    subgoal by (timeit eval)
    subgoal by (timeit code_simp)
    subgoal by (timeit code_simp)
    subgoal by (timeit code_simp)
    subgoal by (timeit normalization)
    subgoal by (timeit eval)
    subgoal by (timeit eval)
    done
  subgoal by (timeit eval)
  subgoal by (timeit eval)
  subgoal
    apply (unfold spm.wellformed_fixpoint_def Let_def case_prod_unfold; intro conjI)
    subgoal by (timeit code_simp)
    subgoal by (timeit eval)
    done
  done
```

5.3.2 The generated theorems and definitions

```
thm ssp.protocol_secure

thm kscomp_enum_consts.nchotomy
thm kscomp_sets.nchotomy
thm kscomp_fun.nchotomy
thm kscomp_atom.nchotomy
thm kscomp_arity.simps
thm kscomp_public.simps
thm kscomp_Γ.simps
thm kscomp_Ana.simps

thm kscomp_transaction_p1_outOfBand_def
thm kscomp_transaction_p1_oufOfBandD_def
thm kscomp_transaction_p1_updateKey_def
thm kscomp_transaction_p1_updateKeyServer_def
thm kscomp_transaction_p1_authAttack_def
thm kscomp_transaction_p2_passwordGenD_def
thm kscomp_transaction_p2_pubkeysGen_def
thm kscomp_transaction_p2_updateKeyPw_def
thm kscomp_transaction_p2_updateKeyServerPw_def
thm kscomp_transaction_p2_authAttack2_def
thm kscomp_protocol_def

thm kscomp_fixpoint_def

end
```

5.4 The PKCS Model, Scenario 3

```
theory PKCS_Model03
  imports "../../PSPSP"

begin
```

```
declare [[code_timing,pspsp_timing]]

trac‹
Protocol: ATTACK_UNSET

Enumerations:
token   = {token1}

Sets:
extract/1 wrap/1 decrypt/1 sensitive/1

Functions:
Public  senc/2 h/1
Private inv/1

Analysis:
senc(M,K2) ? K2 -> M  #This analysis rule corresponds to the decrypt2 rule in the AIF-omega
specification.
                      #M was type untyped

Transactions:
iik1()
  new K1
  insert K1 sensitive(token1)
  insert K1 extract(token1)
  send h(K1).

iik2()
  new K2
  insert K2 wrap(token1)
  send h(K2).

# ======================wrap================
wrap(K1:value,K2:value)
  receive h(K1)
  receive h(K2)
  K1 in extract(token1)
  K2 in wrap(token1)
  send senc(K1,K2).

# ======================set wrap================
setwrap(K2:value)
  receive h(K2)
  K2 notin decrypt(token1)
  insert K2 wrap(token1).

# ======================set decrypt================
setdecrypt(K2:value)
  receive h(K2)
  K2 notin wrap(token1)
  insert K2 decrypt(token1).

# ======================decrypt================
decrypt1(K2:value,M:value)  #M was untyped in the AIF-omega specification.
  receive h(K2)
  receive senc(M,K2)
  K2 in decrypt(token1)
  send M.

# ======================attacks================
attack1(K1:value)
  receive K1
  K1 in sensitive(token1)
```

```
  attack.
›
```

5.4.1 Protocol model setup

protocol_model_setup *spm: ATTACK_UNSET*

5.4.2 Fixpoint computation

compute_fixpoint *ATTACK_UNSET_protocol ATTACK_UNSET_fixpoint attack_trace*

The fixpoint contains an attack signal

lemma *"attack⟨ln 0⟩ ∈ set (fst ATTACK_UNSET_fixpoint)"*
by *code_simp*

The attack trace can be inspected as follows

print_attack_trace *ATTACK_UNSET ATTACK_UNSET_protocol attack_trace*

5.4.3 The generated theorems and definitions

thm *ATTACK_UNSET_enum_consts.nchotomy*
thm *ATTACK_UNSET_sets.nchotomy*
thm *ATTACK_UNSET_fun.nchotomy*
thm *ATTACK_UNSET_atom.nchotomy*
thm *ATTACK_UNSET_arity.simps*
thm *ATTACK_UNSET_public.simps*
thm *ATTACK_UNSET_Γ.simps*
thm *ATTACK_UNSET_Ana.simps*

thm *ATTACK_UNSET_transaction_iik1_def*
thm *ATTACK_UNSET_transaction_iik2_def*
thm *ATTACK_UNSET_transaction_wrap_def*
thm *ATTACK_UNSET_transaction_setwrap_def*
thm *ATTACK_UNSET_transaction_setdecrypt_def*
thm *ATTACK_UNSET_transaction_decrypt1_def*
thm *ATTACK_UNSET_transaction_attack1_def*

thm *ATTACK_UNSET_protocol_def*

thm *ATTACK_UNSET_fixpoint_def*

end

5.5 The PKCS Protocol, Scenario 7

theory *PKCS_Model07*
  imports *"../../PSPSP"*

begin

declare *[[code_timing,pspsp_timing]]*

trac‹

```
Protocol: RE_IMPORT_ATT

Enumerations:
token   = {token1}

Sets:
extract/1 wrap/1 unwrap/1 decrypt/1 sensitive/1

Functions:
```

```
Public  senc/2 h/2 bind/2
Private inv/1

Analysis:
senc(M1,K2) ? K2 -> M1  #This analysis rule corresponds to the decrypt2 rule in the AIF-omega
specification.
                        #M1 was type untyped

Transactions:
iik1()
  new K1
  new N1
  insert N1 sensitive(token1)
  insert N1 extract(token1)
  insert K1 sensitive(token1)
  send h(N1,K1).

iik2()
  new K2
  new N2
  insert N2 wrap(token1)
  insert N2 extract(token1)
  send h(N2,K2).

# =====set wrap=====
setwrap(N2:value,K2:value)
  receive h(N2,K2)
  N2 notin sensitive(token1)
  N2 notin decrypt(token1)
  insert N2 wrap(token1).

# =====set unwrap===
setunwrap(N2:value,K2:value)
  receive h(N2,K2)
  N2 notin sensitive(token1)
  insert N2 unwrap(token1).

# =====unwrap, generate new handler======
#-----------the senstive attr copy-------------
unwrapsensitive(M2:value, K2:value, N1:value, N2:value) #M2 was untyped in the AIF-omega
specification.
  receive senc(M2,K2)
  receive bind(N1,M2)
  receive h(N2,K2)
  N1 in sensitive(token1)
  N2 in unwrap(token1)
  new Nnew
  insert Nnew sensitive(token1)
  send h(Nnew,M2).

#-----------the wrap attr copy-------------
wrapattr(M2:value, K2:value, N1:value, N2:value) #M2 was untyped in the AIF-omega specification.
  receive senc(M2,K2)
  receive bind(N1,M2)
  receive h(N2,K2)
  N1 in wrap(token1)
  N2 in unwrap(token1)
  new Nnew
  insert Nnew wrap(token1)
  send h(Nnew,M2).

#-----------the decrypt attr copy-------------
decrypt1attr(M2:value,K2:value,N1:value,N2:value) #M2 was untyped in the AIF-omega specification.
```

```
  receive senc(M2,K2)
  receive bind(N1,M2)
  receive h(N2,K2)
  N1 in decrypt(token1)
  N2 in unwrap(token1)
  new Nnew
  insert Nnew decrypt(token1)
  send h(Nnew,M2).

decrypt2attr(M2:value,K2:value,N1:value,N2:value) #M2 was untyped in the AIF-omega specification.
  receive senc(M2,K2)
  receive bind(N1,M2)
  receive h(N2,K2)
  N1 notin sensitive(token1)
  N1 notin wrap(token1)
  N1 notin decrypt(token1)
  N2 in unwrap(token1)
  new Nnew
  send h(Nnew,M2).

# ======================wrap================
wrap(N1:value,K1:value,N2:value,K2:value)
  receive h(N1,K1)
  receive h(N2,K2)
  N1 in extract(token1)
  N2 in wrap(token1)
  send senc(K1,K2)
  send bind(N1,K1).

# =====set decrypt===
setdecrypt(Nnew:value, K2:value)
  receive h(Nnew,K2)
  Nnew notin wrap(token1)
  insert Nnew decrypt(token1).

# ======================decrypt================
decrypt1(Nnew:value, K2:value,M1:value) #M1 was untyped in the AIF-omega specification.
  receive h(Nnew,K2)
  receive senc(M1,K2)
  Nnew in decrypt(token1)
  delete Nnew decrypt(token1)
  send M1.

# ======================attacks================
attack1(K1:value)
  receive K1
  K1 in sensitive(token1)
  attack.
›
```

5.5.1 Protocol model setup

protocol_model_setup *spm: RE_IMPORT_ATT*

5.5.2 Fixpoint computation

compute_fixpoint *RE_IMPORT_ATT_protocol RE_IMPORT_ATT_fixpoint attack_trace*

The fixpoint contains an attack signal

lemma *"attack⟨ln 0⟩ ∈ set (fst RE_IMPORT_ATT_fixpoint)"*
by *code_simp*

The attack trace can be inspected as follows

print_attack_trace *RE_IMPORT_ATT RE_IMPORT_ATT_protocol attack_trace*

5.5.3 The generated theorems and definitions

thm *RE_IMPORT_ATT_enum_consts.nchotomy*
thm *RE_IMPORT_ATT_sets.nchotomy*
thm *RE_IMPORT_ATT_fun.nchotomy*
thm *RE_IMPORT_ATT_atom.nchotomy*
thm *RE_IMPORT_ATT_arity.simps*
thm *RE_IMPORT_ATT_public.simps*
thm *RE_IMPORT_ATT_Γ.simps*
thm *RE_IMPORT_ATT_Ana.simps*

thm *RE_IMPORT_ATT_transaction_iik1_def*
thm *RE_IMPORT_ATT_transaction_iik2_def*
thm *RE_IMPORT_ATT_transaction_setwrap_def*
thm *RE_IMPORT_ATT_transaction_setunwrap_def*
thm *RE_IMPORT_ATT_transaction_unwrapsensitive_def*
thm *RE_IMPORT_ATT_transaction_wrapattr_def*
thm *RE_IMPORT_ATT_transaction_decrypt1attr_def*
thm *RE_IMPORT_ATT_transaction_decrypt2attr_def*
thm *RE_IMPORT_ATT_transaction_wrap_def*
thm *RE_IMPORT_ATT_transaction_setdecrypt_def*
thm *RE_IMPORT_ATT_transaction_decrypt1_def*
thm *RE_IMPORT_ATT_transaction_attack1_def*

thm *RE_IMPORT_ATT_protocol_def*

thm *RE_IMPORT_ATT_fixpoint_def*

end

5.6 The PKCS Protocol, Scenario 9

theory *PKCS_Model09*
  imports *"../../PSPSP"*

begin

declare *[[code_timing,pspsp_timing]]*

trac‹
```
Protocol: LOSS_KEY_ATT

Enumerations:
token   = {token1}

Sets:
extract/1 wrap/1 unwrap/1 decrypt/1 sensitive/1

Functions:
Public  senc/2 h/2 bind/3
Private inv/1

Analysis:
senc(M1,K2) ? K2 -> M1  #This analysis rule corresponds to the decrypt2 rule in the AIF-omega
specification.
                        #M1 was type untyped

Transactions:
intruderValueGen()
  new K
```

```
  send K.

iik1()
  new K1
  new N1
  insert N1 sensitive(token1)
  insert N1 extract(token1)
  insert K1 sensitive(token1)
  send h(N1,K1).

iik2()
  new K2
  new N2
  insert N2 wrap(token1)
  insert N2 extract(token1)
  send h(N2,K2).

iik3()
  new K3
  new N3
  insert N3 extract(token1)
  insert N3 decrypt(token1)
  insert K3 decrypt(token1)
  send h(N3,K3)
  send K3.

# =====set wrap=====
setwrap(N2:value,K2:value) where N2 != K2
  receive h(N2,K2)
  N2 notin sensitive(token1)
  N2 notin decrypt(token1)
  insert N2 wrap(token1).

# =====set unwrap===
setunwrap(N2:value,K2:value) where N2 != K2
  receive h(N2,K2)
  N2 notin sensitive(token1)
  insert N2 unwrap(token1).

# =====unwrap, generate new handler======
#-----------add the wrap attr copy-------------
unwrapWrap(M2:value,K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 !=
N2, N1 != N2 #M2 was untyped in the AIF-omega specification.
  receive senc(M2,K2)
  receive bind(N1,M2,K2)
  receive h(N2,K2)
  N1 in wrap(token1)
  N2 in unwrap(token1)
  new Nnew
  insert Nnew wrap(token1)
  send h(Nnew,M2).

#-----------add the senstive attr copy-------------
unwrapSens(M2:value,K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 !=
N2, N1 != N2 #M2 was untyped in the AIF-omega specification.
  receive senc(M2,K2)
  receive bind(N1,M2,K2)
  receive h(N2,K2)
  N1 in sensitive(token1)
  N2 in unwrap(token1)
  new Nnew
  insert Nnew sensitive(token1)
  send h(Nnew,M2).
```

```
#-----------add the decrypt attr copy-------------
decrypt1Attr(M2:value, K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 !=
N2, N1 != N2 #M2 was untyped in the AIF-omega specification.
  receive senc(M2,K2)
  receive bind(N1,M2,K2)
  receive h(N2,K2)
  N1 in decrypt(token1)
  N2 in unwrap(token1)
  new Nnew
  insert Nnew decrypt(token1)
  send h(Nnew,M2).

decrypt2Attr(M2:value, K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 !=
N2, N1 != N2 #M2 was untyped in the AIF-omega specification.
  receive senc(M2,K2)
  receive bind(N1,M2,K2)
  receive h(N2,K2)
  N1 notin wrap(token1)
  N1 notin sensitive(token1)
  N1 notin decrypt(token1)
  N2 in unwrap(token1)
  new Nnew
  send h(Nnew,M2).

# ======================wrap================
wrap(N1:value,K1:value, N2:value, K2:value) where N1 != N2, N1 != K2, N1 != K1, N2 != K2, N2 != K1, K2
!= K1
  receive h(N1,K1)
  receive h(N2,K2)
  N1 in extract(token1)
  N2 in wrap(token1)
  send senc(K1,K2)
  send bind(N1,K1,K2).

# ======================bind generation================
bind1(K3:value,N2:value,K2:value, K1:value) where K3 != N2, K3 != K2, K3 != K1, N2 != K2, N2 != K1, K2
!= K1
  receive K3
  receive h(N2,K2)
  send bind(N2,K3,K3).

bind2(K3:value,N2:value,K2:value, K1:value) where K3 != N2, K3 != K2, K3 != K1, N2 != K2, N2 != K1, K2
!= K1
  receive K3
  receive K1
  receive h(N2,K2)
  send bind(N2,K1,K3)
  send bind(N2,K3,K1).

# =====set decrypt===
setdecrypt(Nnew:value,K2:value) where Nnew != K2
  receive h(Nnew,K2)
  Nnew notin wrap(token1)
  insert Nnew decrypt(token1).

# ======================decrypt================
decrypt1(Nnew:value,K2:value,M1:value) where Nnew != K2, Nnew != M1, K2 != M1 #M1 was untyped in the
AIF-omega specification.
  receive h(Nnew,K2)
  receive senc(M1,K2)
  Nnew in decrypt(token1)
  send M1.
```

```
# ====================attacks===============
attack1(K1:value)
  receive K1
  K1 in sensitive(token1)
  attack.
›
```

5.6.1 Protocol model setup

protocol_model_setup *spm: LOSS_KEY_ATT*

5.6.2 Fixpoint computation

compute_fixpoint *LOSS_KEY_ATT_protocol LOSS_KEY_ATT_fixpoint attack_trace*

The fixpoint contains an attack signal

lemma *"attack⟨ln 0⟩ ∈ set (fst LOSS_KEY_ATT_fixpoint)"*
by *code_simp*

The attack trace can be inspected as follows

print_attack_trace *LOSS_KEY_ATT LOSS_KEY_ATT_protocol attack_trace*

5.6.3 The generated theorems and definitions

thm *LOSS_KEY_ATT_enum_consts.nchotomy*
thm *LOSS_KEY_ATT_sets.nchotomy*
thm *LOSS_KEY_ATT_fun.nchotomy*
thm *LOSS_KEY_ATT_atom.nchotomy*
thm *LOSS_KEY_ATT_arity.simps*
thm *LOSS_KEY_ATT_public.simps*
thm *LOSS_KEY_ATT_Γ.simps*
thm *LOSS_KEY_ATT_Ana.simps*

thm *LOSS_KEY_ATT_transaction_iik1_def*
thm *LOSS_KEY_ATT_transaction_iik2_def*
thm *LOSS_KEY_ATT_transaction_iik3_def*
thm *LOSS_KEY_ATT_transaction_setwrap_def*
thm *LOSS_KEY_ATT_transaction_setunwrap_def*
thm *LOSS_KEY_ATT_transaction_unwrapWrap_def*
thm *LOSS_KEY_ATT_transaction_unwrapSens_def*
thm *LOSS_KEY_ATT_transaction_decrypt1Attr_def*
thm *LOSS_KEY_ATT_transaction_decrypt2Attr_def*
thm *LOSS_KEY_ATT_transaction_wrap_def*
thm *LOSS_KEY_ATT_transaction_bind1_def*
thm *LOSS_KEY_ATT_transaction_bind2_def*
thm *LOSS_KEY_ATT_transaction_setdecrypt_def*
thm *LOSS_KEY_ATT_transaction_decrypt1_def*
thm *LOSS_KEY_ATT_transaction_attack1_def*

thm *LOSS_KEY_ATT_protocol_def*
thm *LOSS_KEY_ATT_fixpoint_def*

end

Bibliography

[1] A. D. Brucker and S. Mödersheim. Integrating Automated and Interactive Protocol Verification. In P. Degano and J. D. Guttman, editors, *Formal Aspects in Security and Trust, 6th International Workshop, FAST 2009, Eindhoven, The Netherlands, November 5-6, 2009, Revised Selected Papers*, volume 5983 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2009. doi: 10.1007/978-3-642-12459-4_18.

[2] F. Haftmann and L. Bulwahn. Code generation from Isabelle/HOL theories, 2021. URL `http://isabelle.in.tum.de/doc/codegen.pdf`.

[3] A. V. Hess. *Typing and Compositionality for Stateful Security Protocols*. PhD thesis, 2019. URL `https://orbit.dtu.dk/en/publications/typing-and-compositionality-for-stateful-security-protocols`.

[4] A. V. Hess and S. Mödersheim. Formalizing and Proving a Typing Result for Security Protocols in Isabelle/HOL. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 451–463. IEEE Computer Society, 2017. doi: 10.1109/CSF.2017.27.

[5] A. V. Hess and S. Mödersheim. A Typing Result for Stateful Protocols. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 374–388. IEEE Computer Society, 2018. doi: 10.1109/CSF.2018.00034.

[6] A. V. Hess, S. Mödersheim, and A. D. Brucker. Stateful Protocol Composition. In J. López, J. Zhou, and M. Soriano, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, volume 11098 of *Lecture Notes in Computer Science*, pages 427–446. Springer, 2018. doi: 10.1007/978-3-319-99073-6_21.

[7] A. V. Hess, S. Mödersheim, and A. D. Brucker. Stateful Protocol Composition and Typing. *Archive of Formal Proofs*, Apr. 2020. ISSN 2150-914x. `http://isa-afp.org/entries/Stateful_Protocol_Composition_and_Typing.html`, Formal proof development.

[8] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9.

[9] M. Wenzel. The Isabelle/Isar reference manual, 2021. URL `http://isabelle.in.tum.de/doc/isar-ref.pdf`.