Information Flow Control via Stateful Intransitive Noninterference in Language IMP

Pasquale Noce
Senior Staff Firmware Engineer at HID Global, Italy
pasquale dot noce dot lavoro at gmail dot com
pasquale dot noce at hidglobal dot com
February 13, 2024

Abstract

The scope of information flow control via static type systems is in principle much broader than information flow security, since this concept promises to cope with information flow correctness in full generality. Such a correctness policy can be expressed by extending the notion of a single stateless level-based interference relation applying throughout a program – addressed by the static security type systems described by Volpano, Smith, and Irvine, and formalized in Nipkow and Klein’s book on formal programming language semantics (in the version of February 2023) – to that of a stateful interference function mapping program states to (generally) intransitive interference relations.

This paper studies information flow control via stateful intransitive noninterference. First, the notion of termination-sensitive information flow security with respect to a level-based interference relation is generalized to that of termination-sensitive information flow correctness with respect to such a correctness policy. Then, a static type system is specified and is proven to be capable of enforcing such policies. Finally, the information flow correctness notion and the static type system introduced here are proven to degenerate to the counterparts formalized in Nipkow and Klein’s book in case of a stateless level-based information flow correctness policy. Although the operational semantics of the didactic programming language IMP employed in the book is used for this purpose, the introduced concepts apply to larger, real-world imperative programming languages as well.

Contents

1 Underlying concepts and formal definitions 2
   1.1 Global context definitions 6
   1.2 Local context definitions 7
In a passage of his book *Clean Architecture: A Craftsman’s Guide to Software Structure and Design* (Prentice Hall, 2017), Robert C. Martin defines a computer program as “a detailed description of the policy by which inputs are transformed into outputs”, remarking that “indeed, at its core, that’s all a computer program actually is”. Accordingly, the scope of information flow control via static type systems is in principle much broader than language-based information flow security, since this concept promises to cope with information flow correctness in full generality.

This is already shown by a basic program implementing the Euclidean algorithm, in Donald Knuth’s words “the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day” (from *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, third edition, Addison-Wesley, 1997). Here below is a sample such C program, where variables \( a \) and \( b \) initially contain two positive integers and \( a \) will finally contain the output, namely the greatest common divisor of those integers.
Even in a so basic program, information is not allowed to indistinctly flow from any variable to any other one, on pain of the program being incorrect. If an incautious programmer swapped \( a \) for \( b \) in the assignment at line 4, the greatest common divisor output for any two inputs \( a \) and \( b \) would invariably match \( a \), whereas swapping the sides of the assignment at line 5 would give rise to an endless loop. Indeed, despite the marked differences in the resulting program behavior, both of these potential errors originate in information flowing between variables along paths other than the demanded ones. A sound implementation of the Euclidean algorithm does not provide for any information flow from \( a \) to \( b \), or from \( b \) to \( r \).

The static security type systems addressed in [11], [10], and [7] restrict the information flows occurring in a program based on a mapping of each of its variables to a domain along with an interference relation between such domains, including any pair of domains such that the former may interfere with the latter. Accordingly, if function \( \text{dom} \) stands for such a mapping, and infix notation \( u \Rightarrow v \) denotes the inclusion of any pair of domains \((u, v)\) in such a relation (both notations are borrowed from [9]), the above errors would be detected at compile time by a static type system enforcing an interference relation such that:

- \( \text{dom } a \Rightarrow \text{dom } r \), \( \text{dom } b \Rightarrow \text{dom } r \) (line 3),
- \( \text{dom } b \Rightarrow \text{dom } a \) (line 4),
- \( \text{dom } r \Rightarrow \text{dom } b \) (line 5),

and ruling out any other pair of distinct domains. Such an interference relation would also embrace the implicit information flow from \( b \) to the other two variables arising from the loop’s termination condition (line 6).

Remarkably, as \( \text{dom } a \Rightarrow \text{dom } r \) and \( \text{dom } r \Rightarrow \text{dom } b \) but \( \neg \text{dom } a \Rightarrow \text{dom } b \), this interference relation turns out to be intransitive. Therefore, unlike the security static type systems studied in [11] and [10], which deal with level-based, and then transitive, interference relations, a static type system aimed at enforcing information flow correctness in full generality must be capable of dealing with intransitive interference relations as well.

This should come as no surprise, since [9] shows that this is the general
case already for interference relations expressing information flow security policies.

But the bar can be raised further. Considering the above program again, the information flows needed for its operation, as listed above, need not be allowed throughout the program. Indeed, information needs to flow from a and b to r at line 3, from b to a at line 4, from r to b at line 5, and then (implicitly) from b to the other two variables at line 6. Based on this observation, error detection at compile time can be made finer-grained by rewriting the program as follows, where i is a further integer variable introduced for this purpose.

```plaintext
do
{
    i = 0;
    r = a % b;
    i = 1;
    a = b;
    i = 2;
    b = r;
    i = 3;
} while (b);
```

In this program, i serves as a state variable whose value in every execution step can be determined already at compile time. Since a distinct set of information flows is allowed for each of its values, a finer-grained information flow correctness policy for this program can be expressed by extending the concept of a single, stateless interference relation applying throughout the program to that of a stateful interference function mapping program states to interference relations (in this case, according to the value of i). As a result of this extension, for each program state, a distinct interference relation – that is, the one to which the applied interference function maps that state – can be enforced at compile time by a suitable static type system.

If mixfix notation \( s: u \rightsquigarrow v \) denotes the inclusion of any pair of domains \((u, v)\) in the interference relation associated with any state \( s \), a finer-grained information flow correctness policy for this program can then be expressed as an interference function such that:

- \( s: \text{dom } a \rightsquigarrow \text{dom } r \), \( s: \text{dom } b \rightsquigarrow \text{dom } r \) for any \( s \) where \( i = 0 \) (line 4),
- \( s: \text{dom } b \rightsquigarrow \text{dom } a \) for any \( s \) where \( i = 1 \) (line 6),
- \( s: \text{dom } r \rightsquigarrow \text{dom } b \) for any \( s \) where \( i = 2 \) (line 8),
- \( s: \text{dom } b \rightsquigarrow \text{dom } a, s: \text{dom } b \rightsquigarrow \text{dom } r, s: \text{dom } b \rightsquigarrow \text{dom } i \) for any \( s \) where \( i = 3 \) (line 10),
and ruling out any other pair of distinct domains in any state.

Notably, to enforce such an interference function, a static type system would not need to keep track of the full program state in every program execution step (which would be unfeasible, as the values of a, b, and r cannot be determined at compile time), but only of the values of some specified state variables (in this case, of i alone). Accordingly, term state variable will henceforth refer to any program variable whose value may affect that of the interference function expressing the information flow correctness policy in force, namely the interference relation to be applied.

Needless to say, there would be something artificial about the introduction of such a state variable into the above sample program, since it is indeed so basic as not to provide for a state machine on its own, so that i would be aimed exclusively at enabling the enforcement of such an information flow correctness policy. Yet, real-world imperative programs, for which error detection at compile time is truly meaningful, do typically provide for state machines such that only a subset of all the potential information flows is allowed in each state; and even for those which do not, the addition of some ad hoc state variable to enforce such a policy could likely be an acceptable trade-off.

Accordingly, the goal of this paper is to study information flow control via stateful intransitive noninterference. First, the notion of termination-sensitive information flow security with respect to a level-based interference relation, as defined in [7], section 9.2.6, is generalized to that of termination-sensitive information flow correctness with respect to a stateful interference function having (generally) intransitive interference relations as values. Then, a static type system is specified and is proven to be capable of enforcing such information flow correctness policies. Finally, the information flow correctness notion and the static type system introduced here are proven to degenerate to the counterparts addressed in [7], section 9.2.6, in case of a stateless level-based information flow correctness policy.

Although the operational semantics of the didactic imperative programming language IMP employed in [7] is used for this purpose, the introduced concepts are applicable to larger, real-world imperative programming languages as well, by just affording the additional type system complexity arising from richer language constructs. Accordingly, the informal explanations accompanying formal content in what follows will keep making use of sample C code snippets.

For further information about the formal definitions and proofs contained in this paper, see Isabelle documentation, particularly [8], [4], [2], [3], and [1].
1.1 Global context definitions

**declare** [[syntax-ambiguity-warning = false]]

datatype com-flow =
    Assign vname aexp (- ::= - [1000, 61] 70) |
    Observe vname set ⟨−⟩ [61] 70

type-synonym flow = com-flow list

type-synonym config = state set × vname set

type-synonym scope = config set × bool

abbreviation eq-states :: state ⇒ state ⇒ vname set ⇒ bool
    ((· = · ⟨⊆ · −⟩) [51, 51] 50) where
    s = t (⊆ X) ⇔ ∀ x ∈ X. s x = t x

abbreviation univ-states :: state set ⇒ vname set ⇒ state set
    ((Univ - ⟨⊆ −⟩) [51, 75] 75) where
    Univ A (⊆ X) ≡ {s. ∃ t ∈ A. s = t (⊆ X)}

abbreviation univ-vars-if :: state set ⇒ vname set ⇒ vname set
    ((Univ?? - ·) [51, 75] 75) where
    Univ?? A X ≡ if A = {} then UNIV else X

abbreviation tl2 xs ≡ tl (tl xs)

fun run-flow :: flow ⇒ state ⇒ state where
    run-flow (x ::= a # cs) s = run-flow cs (s(x ::= aval a s)) |
    run-flow (- # cs) s = run-flow cs s |
    run-flow - s = s

primrec no-upd :: flow ⇒ vname ⇒ bool where
    no-upd (c # cs) x =
        ((case c of y ::= - ⇒ y ≠ x | - ⇒ True) ∧ no-upd cs x) |
    no-upd [] - = True

primrec avars :: aexp ⇒ vname set where
    avars (N i) = {} |
    avars (V x) = {x} |
    avars (Plus a1 a2) = avars a1 ∪ avars a2

primrec bvars :: bexp ⇒ vname set where
    bvars (Bc v) = {} |
    bvars (Not b) = bvars b |
    bvars (And b1 b2) = bvars b1 ∪ bvars b2 |
    bvars (Less a1 a2) = avars a1 ∪ avars a2
fun flow-aux :: com list ⇒ flow where
flow-aux ((x ::= a) # cs) = (x ::= a) # flow-aux cs |
flow-aux ((IF b THEN - ELSE -) # cs) = (bvars b) # flow-aux cs |
flow-aux ((c;; -) # cs) = flow-aux (c # cs) |
flow-aux [] = []

definition flow :: (com × state) list ⇒ flow where
flow cfs = flow-aux (map fst cfs)

function small-steps :: com × state ⇒ (com × state) list ⇒ com × state ⇒ bool
((→∗{ } -) [51, 51] 55)
where
cf →∗{[]} cf′ = (cf = cf′) |
cf →∗{cfs @ [cf']} cf'' = (cf →∗{cfs} cf' ∧ cf' → cf'')

by (atomize-elim, auto intro: rev-cases)
termination by lexicographic-order

lemmas small-steps-induct = small-steps.induct [split-format(complete)]

1.2 Local context definitions

In what follows, stateful intransitive noninterference will be formalized within the local context defined by means of a locale [1], named noninterf. Later on, this will enable to prove the degeneracy of the following definitions to the stateless level-based counterparts addressed in [11], [10], and [7], and formalized in [5] and [6], via a suitable locale interpretation.

Locale noninterf contains three parameters, as follows.

- A stateful interference function interf mapping program states to interference predicates of two domains, intended to be true just in case the former domain is allowed to interfere with the latter.
- A function dom mapping program variables to their respective domains.
- A set state collecting all state variables.

As the type of the domains is modeled using a type variable, it may be assigned arbitrarily by any locale interpretation, which will enable to set it to nat upon proving degeneracy. Moreover, the above mixfix notation s: u ⇞ v is adopted to express the fact that any two domains u, v satisfy the interference predicate interf s associated with any state s, namely the fact that u is allowed to interfere with v in state s.
Locale noninterf also contains an assumption, named interf-state, which serves the purpose of supplying parameter state with its intended semantics, namely standing for the set of all state variables. The assumption is that function interf maps any two program states agreeing on the values of all the variables in set state to the same interference predicate. Correspondingly, any locale interpretation instantiating parameter state as the empty set must instantiate parameter interf as a function mapping any two program states, even if differing in the values of all variables, to the same interference predicate – namely, as a constant function. Hence, any such locale interpretation refers to a single, stateless interference predicate applying throughout the program. Unsurprisingly, this is the way how those parameters will be instantiated upon proving degeneracy.

The one just mentioned is the only locale assumption. Particularly, the following formalization does not rely upon the assumption that the interference predicates returned by function interf be reflexive, although this will be the case for any meaningful real-world information flow correctness policy.

locale noninterf =
  fixes
    interf :: state ⇒ 'd ⇒ 'd ⇒ bool
    (v - ⊳ -) [51, 51, 51] 50) and
    dom :: vname ⇒ 'd and
    state :: vname set
  assumes
    interf-state: s = t (⊆ state) =⇒ interf s = interf t

context noninterf
begin

Locale parameters interf and dom are provided with their intended semantics by the definitions of functions sources and correct, which are formalized here below based on the following underlying ideas.

As long as a stateless transitive interference relation between domains is considered, the condition for the correctness of the value of a variable resulting from a full or partial program execution need not take into account the execution flow producing it, but rather the initial program state only. In fact, this is what happens with the stateless level-based correctness condition addressed in [11], [10], and [7]: the resulting value of a variable of level \( l \) is correct if the same value is produced for any initial state agreeing with the given one on the value of every variable of level not higher than \( l \).

Things are so simple because, for any variables \( x, y, \) and \( z \), if \( dom z \rightsquigarrow dom y \) and \( dom y \rightsquigarrow dom x \), transitivity entails \( dom z \rightsquigarrow dom x \), and these interference relationships hold statelessly. Therefore, \( z \) may be counted among
the variables whose initial values are allowed to affect \( x \) independently of whether some intermediate value of \( y \) may affect \( x \) within the actual execution flow.

Unfortunately, switching to stateful intransitive interference relations puts an end to that happy circumstance – indeed, even statefulness or intransitivity alone would suffice for this sad ending. In this context, deciding about the correctness of the resulting value of a variable \( x \) still demands the detection of the variables whose initial values are allowed to interfere with \( x \), but the execution flow leading from the initial program state to the resulting one needs to be considered to perform such detection.

This is precisely the task of function \( \text{sources} \), so named after its finite state machine counterpart defined in [9]. It takes as inputs an execution flow \( cs \), an initial program state \( s \), and a variable \( x \), and outputs the set of the variables whose values in \( s \) are allowed to affect the value of \( x \) in the state \( s' \) into which \( cs \) turns \( s \), according to \( cs \) as well as to the information flow correctness policy expressed by parameters \( \text{interf} \) and \( \text{dom} \).

In more detail, execution flows are modeled as lists comprising items of two possible kinds, namely an assignment of the value of an arithmetic expression \( a \) to a variable \( z \) or else an observation of the values of the variables in a set \( X \), denoted through notations \( z := a \) (same as with assignment commands) and \( (X) \) and keeping track of explicit and implicit information flows, respectively. Particularly, item \( (X) \) refers to the act of observing the values of the variables in \( X \) leaving the program state unaltered. During the execution of an IMP program, this happens upon any evaluation of a boolean expression containing all and only the variables in \( X \).

Function \( \text{sources} \) is defined along with an auxiliary function \( \text{sources-aux} \) by means of mutual recursion. Based on this definition, \( \text{sources} \) \( cs \) \( s \) \( x \) contains a variable \( y \) if there exist a descending sequence of left sublists \( cs_{n+1}, cs_n \atop \{c_n\}, \ldots, cs_1 \atop \{c_1\} \) of \( cs \) and a sequence of variables \( y_{n+1}, \ldots, y_1 \), where \( n \geq 1 \), \( cs_{n+1} = cs, y_{n+1} = x \), and \( y_1 = y \), satisfying the following conditions.

- For each positive integer \( i \leq n \), \( c_i \) is an assignment \( y_{i+1} := a_i \) where:
  - \( y_i \in \text{avars} \ a_i \),
  - \( \text{run-flow} \ cs_i : \text{dom} \ y_i \leadsto \text{dom} \ y_{i+1} \), and
  - the right sublist of \( cs_{i+1} \) complementary to \( cs_i \atop \{c_i\} \) does not comprise any assignment to variable \( y_{i+1} \) (as assignment \( c_i \) would otherwise be irrelevant),

or else an observation \( (X_i) \) where:

- \( y_i \in X_i \) and
- \( \text{run-flow} \ cs_i : \text{dom} \ y_i \leadsto \text{dom} \ y_{i+1} \).
• $cs_1$ does not comprise any assignment to variable $y$.

In addition, $sources\ cs\ s\ x$ contains variable $x$ also if $cs$ does not comprise any assignment to variable $x$.

**function**

$sources :: flow \Rightarrow state \Rightarrow vname \Rightarrow vname\ set$ and $sources-aux :: flow \Rightarrow state \Rightarrow vname \Rightarrow vname\ set$ where

$sources\ (cs @ [c])\ s\ x = (case\ c\ of$
\hfill $z ::= a \Rightarrow if\ z = x$
\hfill $\text{then}\ sources-aux\ cs\ s\ x\ \cup\ \{ sources\ cs\ s\ y \mid y.$
\hfill $\text{run-flow}\ cs\ s:\ dom\ y\ \rightsquigarrow\ dom\ x\ \land\ y\ \in\ avars\ a \} \text{ else } sources\ cs\ s\ x$ |
\hfill $(X) \Rightarrow$
\hfill $sources\ cs\ s\ x\ \cup\ \{ sources\ cs\ s\ y \mid y.$
\hfill $\text{run-flow}\ cs\ s:\ dom\ y\ \rightsquigarrow\ dom\ x\ \land\ y\ \in\ X \} \}$ |
\hfill $sources\ []\ -\ x = \{ x \}$ |

$sources-aux\ (cs @ [c])\ s\ x = (case\ c\ of$
\hfill $- ::= - \Rightarrow$
\hfill $sources-aux\ cs\ s\ x$ |
\hfill $(X) \Rightarrow$
\hfill $sources-aux\ cs\ s\ x\ \cup\ \{ sources\ cs\ s\ y \mid y.$
\hfill $\text{run-flow}\ cs\ s:\ dom\ y\ \rightsquigarrow\ dom\ x\ \land\ y\ \in\ X \} \}$ |
\hfill $sources-aux\ []\ -\ - = \{ \}$

**proof** (atomize-elim)

fix $a :: flow \times state \times vname + flow \times state \times vname$

{ assume
\hfill $\forall\ cs\ c\ s\ x.\ a\ \neq\ \text{Inl}\ (cs @ [c],\ s,\ x)$ and
\hfill $\forall\ s\ x.\ a\ \neq\ \text{Inl}\ ([],\ s,\ x)$ and
\hfill $\forall\ s\ x.\ a\ \neq\ \text{Inr}\ ([],\ s,\ x)$
\hence $\exists\ cs\ c\ s\ x.\ a = \text{Inr}\ (cs @ [c],\ s,\ x)$
\hfill by (metis obj-sumE prod-cases3 rev-exhaust)
}

thus
\hfill $\exists\ cs\ c\ s\ x.\ a = \text{Inl}\ (cs @ [c],\ s,\ x)$
\hfill $\vee$
\hfill $\exists\ s\ x.\ a = \text{Inl}\ ([],\ s,\ x)$
\hfill $\vee$
\hfill $\exists\ cs\ c\ s\ x.\ a = \text{Inr}\ (cs @ [c],\ s,\ x)$
\hfill $\vee$
\hfill $\exists\ s\ x.\ a = \text{Inr}\ ([],\ s,\ x)$
\hfill by blast

qed auto

**termination** by lexicographic-order
lemmas $\text{sources-induct} = \text{sources-sources-aux.induct}$

Predicate $\text{correct}$ takes as inputs a program $c$, a set of program states $A$, and a set of variables $X$. Its truth value equals that of the following termination-sensitive information flow correctness condition: for any state $s$ agreeing with a state in $A$ on the values of the state variables in $X$, if the small-step program semantics turns configuration $(c, s)$ into configuration $(c_1, s_1)$, and $(c_1, s_1)$ into configuration $(c_2, s_2)$, then for any state $t_1$ agreeing with $s_1$ on the values of the variables in $\text{sources cs s}_1 x$, where $cs$ is the execution flow leading from $(c_1, s_1)$ to $(c_2, s_2)$, the small-step semantics turns $(c_1, t_1)$ into some configuration $(c_2', t_2)$ such that:

- $c_2' = \text{SKIP}$ (namely, $(c_2', t_2)$ is a final configuration) just in case $c_2 = \text{SKIP}$, and
- the value of variable $x$ in state $t_2$ is the same as in state $s_2$.

Here below are some comments about this definition.

- As $\text{sources cs s}_1 x$ is the set of the variables whose values in $s_1$ are allowed to affect the value of $x$ in $s_2$, this definition requires any state $t_1$ indistinguishable from $s_1$ in the values of those variables to produce a state where variable $x$ has the same value as in $s_2$ in the continuation of program execution.

- Configuration $(c_2', t_2)$ must be the same one for any variable $x$ such that $s_1$ and $t_1$ agree on the values of any variable in $\text{sources cs s}_1 x$. Otherwise, even if states $s_2$ and $t_2$ agreed on the value of $x$, they could be distinguished all the same based on a discrepancy between the respective values of some other variable. Likewise, if state $t_2$ alone had to be the same for any such $x$, while command $c_2'$ were allowed to vary, state $t_1$ could be distinguished from $s_1$ based on the continuation of program execution. This is the reason why the universal quantification over $x$ isnested within the existential quantification over both $c_2'$ and $t_2$.

- The state machine for a program typically provides for a set of initial states from which its execution is intended to start. In any such case, information flow correctness need not be assessed for arbitrary initial states, but just for those complying with the settled tuples of initial values for state variables. The values of any other variables do not matter, as they do not affect function $\text{interf}$’s ones. This is the motivation for parameter $A$, which then needs to contain just one state for each of such tuples, while parameter $X$ enables to exclude the state variables, if any, whose initial values are not settled.
If locale parameter state matches the empty set, s will be any state agreeing with some state in A on the value of possibly even no variable at all, that is, a fully arbitrary state provided that A is nonempty. This makes s range over all possible states, as required for establishing the degeneracy of the present definition to the stateless level-based counterpart addressed in [7], section 9.2.6.

Why express information flow correctness in terms of the small-step program semantics, instead of resorting to the big-step one as happens with the stateless level-based correctness condition in [7], section 9.2.6? The answer is provided by the following sample C programs, where i is a state variable.

```c
1 y = i;
2 i = (i) ? 1 : 0;
3 x = i + y;
```

```c
1 x = 0;
2 if (i == 10)
3 {
4   x = 10;
5 }
6 i = (i) ? 1 : 0;
7 x += i;
```

Let i be allowed to interfere with x just in case i matches 0 or 1, and y be never allowed to do so. If s₁ were constrained to be the initial state, for both programs i would be included among the variables on which t₁ needs to agree with s₁ in order to be indistinguishable from s₁ in the value of x resulting from the final assignment. Thus, both programs would fail to be labeled as wrong ones, although in both of them the information flow blatantly bypasses the sanitization of the initial value of i, respectively due to an illegal explicit flow and an illegal implicit flow. On the contrary, the present information flow correctness definition detects any such illegal information flow by checking every partial program execution on its own.

**abbreviation** ok-flow :: com ⇒ com ⇒ state ⇒ state ⇒ flow ⇒ bool where

ok-flow c₁ c₂ s₁ s₂ cs ≡

∀ t₁, ∃ c₂′ t₂, ∀ x.

s₁ = t₁ (∪ sources cs s₁ x) →

(c₁, t₁) →⁺ (c₂′, t₂) ∧ (c₂ = SKIP) = (c₂′ = SKIP) ∧ s₂ x = t₂ x

**definition** correct :: com ⇒ state set ⇒ vname set ⇒ bool where

correct c A X ≡

12
∀ s ∈ Univ A (⊆ state ∩ X). ∀ c₁ c₂ s₁ s₂ cfs.
(c, s) →∗ (c₁, s₁) ∧ (c₁, s₁) →∗{cfs} (c₂, s₂) →∗

ok-flow c₁ c₂ s₁ s₂ (flow cfs)

abbreviation interf-set :: state set ⇒ ′d set ⇒ ′d set ⇒ bool
((⋯ → ⋯) [51, 51, 51] 50) where
A: U ⇝ W ≡ ∀ s ∈ A. ∀ u ∈ U. ∀ w ∈ W. u ⇝ w

abbreviation ok-flow-aux ::
config set ⇒ com ⇒ com ⇒ state ⇒ state ⇒ flow ⇒ bool
where
ok-flow-aux U c₁ c₂ s₁ s₂ cs ≡
(∀ t₁. ∃ c₂′ t₂. ∀ x.
 (s₁ = t₁ (⊆ sources-aux cs s₁ x) →
  (c₁, t₁) →∗ (c₂′, t₂) ∧ (c₂ = SKIP) = (c₂′ = SKIP)) ∧
  (s₁ = t₁ (⊆ sources cs s₁ x) → s₂ x = t₂ x)) ∧
(∀ x. (∃ p ∈ U. case p of (B, Y) ⇒
  ∃ s ∈ B. ∃ y ∈ Y. ¬ s: dom y → dom x) → no-upd cs x)

The next step is defining a static type system guaranteeing that well-typed programs satisfy this information flow correctness criterion. Whenever defining a function, and the pursued type system is obviously no exception, the primary question that one has to answer is: which inputs and outputs should it provide for? The type system formalized in [6] simply makes a pass/fail decision on an input program, based on an input security level, and outputs the verdict as a boolean value. Is this still enough in the present case? The answer can be found by considering again the above C program that computes the greatest common divisor of two positive integers a, b using a state variable i, along with its associated stateful interference function. For the reader’s convenience, the program is reported here below.

```c
1 do
2 {
3 i = 0;
4 r = a % b;
5 i = 1;
6 a = b;
7 i = 2;
8 b = r;
9 i = 3;
10 } while (b);
```

As s: dom a ⇝ dom r only for a state s where i = 0, the type system cannot determine that the assignment r = a % b at line 4 is well-typed without knowing that i = 0 whenever that step is executed. Consequently, upon
checking the assignment \( i = \emptyset \) at line 3, the type system must output information indicating that \( i = 0 \) as a result of its execution. This information will then be input to the type system when it is recursively invoked to check line 4, so as to enable the well-typedness of the next assignment to be ascertained.

Therefore, in addition to the program under scrutiny, the type system needs to take a set of program states as input, and as long as the program is well-typed, the output must include a set of states covering any change to the values of the state variables possibly triggered by the input program. In other words, the type system has to simulate the execution of the input program at compile time as regards the values of its state variables. In the following formalization, this results in making the type system take an input of type \textit{state set} and output a value of the same type. Yet, since state variables alone are relevant, a real-world implementation of the type system would not need to work with full \textit{state} values, but just with tuples of state variables’ values.

Is the input/output of a set of program states sufficient to keep track of the possible values of the state variables at each execution step? Here below is a sample C program helping find an answer, which determines the minimum of two integers \( a, b \) and assigns it to variable \( a \) using a state variable \( i \).

```c
i = (a > b) ? 1 : 0;
if (i > 0)
{
    a = b;
}
```

Assuming that the initial value of \( i \) is 0, the information flow correctness policy for this program will be such that:

- \( s: \text{dom } a \rightsquigarrow \text{dom } i, s: \text{dom } b \rightsquigarrow \text{dom } i \) for any program state \( s \) where \( i = 0 \) (line 1),
- \( s: \text{dom } i \rightsquigarrow \text{dom } a \) for any \( s \) where \( i = 0 \) or \( i = 1 \) (line 2, more on this later),
- \( s: \text{dom } b \rightsquigarrow \text{dom } a \) for any \( s \) where \( i = 1 \) (line 4),

ruling out any other pair of distinct domains in any state.

So far, everything has gone smoothly. However, what happens if the program is changed as follows?

```c
i = a - b;
```
if (i > 0) {
    a = b;
}

Upon simulating the execution of the former program, the type system can
determine the set \{0, 1\} of the possible values of variable \(i\) arising from the
conditional assignment \(i = (a > b) \ ? 1 : 0\) at line 1. On the contrary, in
the case of the latter program, the possible values of \(i\) after the assignment
\(i = a - b\) at line 1 must be marked as being *indeterminate*, since they depend
on the initial values of variables \(a\) and \(b\), which are unknown at compile time.
Hence, the type system needs to provide for an additional input/output
parameter of type \(vname\) set, whose input and output values shall collect
the variables whose possible values before and after the execution of the
input program are *determinate*.

The correctness of the simulation of program execution by the type system
can be expressed as the following condition. Suppose that the type sys-

em system outputs a state set \(A'\) and a vname set \(X'\) when it is input a program
\(c\), a state set \(A\), and a vname set \(X\). Then, for any state \(s\) agreeing with
some state in \(A\) on the value of every state variable in \(X\), if \((c, s) \Rightarrow s'\), \(s'\)
must agree with some state in \(A'\) on the value of every state variable in \(X'\).
This can be summarized by saying that the type system must overapprox-
imate program semantics, since any algorithm simulating program execution
cannot but be imprecise (see [7], *incipit* of chapter 13).

In turn, if the outputs for \(c\), \(A', X'\) are \(A''\), \(X''\) and \((c, s') \Rightarrow s''\), \(s''\) must
agree with some state in \(A''\) on the value of every state variable in \(X''\).
But if \(c\) is a loop and \((c, s) \Rightarrow s'\), then \((c, s') \Rightarrow s''\) just in case \(s' = s''\),
so that the type system is guaranteed to overapproximate the semantics
of \(c\) only if states consistent with \(A', X'\) are also consistent with \(A'', X''\)
and vice versa. Thus, the type system needs to be idempotent if \(c\) is a loop,
that is, it must be such that \(A' = A''\) and \(X' = X''\) in this case.
Since idempotence is not required for control structures other than loops,
the main type system \(ctyping2\) formalized in what follows will delegate the
simulation of the execution of loop bodies to an auxiliary, idempotent type
system \(ctyping1\).

This type system keeps track of the program state updates possibly occurring
in its input program using sets of lists of functions of type \(vname \Rightarrow val\)
option option. Command \texttt{SKIP} is mapped to a singleton made of the empty
list, as no state update takes place. An assignment to a variable \(x\) is mapped
to a singleton made of a list comprising a single function, whose value is
\texttt{Some (Some i)} or \texttt{Some None} for \(x\) if it is a state variable and the right-
hand side is a constant \(N\) \(i\) or a non-constant expression, respectively, and
\texttt{None} otherwise. That is, \texttt{None} stands for unchanged/non-state variable
(remember, only state variable updates need to be tracked), whereas Some
None stands for indeterminate variable, since the value of a non-constant
expression in a loop iteration (remember, ctyping1 is meant for simulating
the execution of loop bodies) is in general unknown at compile time.

At first glance, a conditional statement could simply be mapped to the
union of the sets tracking the program state updates possibly occurring in
its branches. However, things are not so simple, as shown by the sample C
loop here below, which has a conditional statement as its body.

```
for (i = 0; i < 2; i++)
{
    if (n % 2)
    {
        a = 1;
        b = 1;
        n++;;
    }
    else
    {
        a = 2;
        c = 2;
        n++;
    }
}
```

If the initial value of the integer variable $n$ is even, the final values of variables
$a$, $b$, and $c$ will be 1, 1, 2, whereas if the initial value of $n$ is odd, the final
values of the aforesaid variables will be 2, 1, 2. Assuming that their initial
value is 0, the potential final values tracked by considering each branch
individually are 1, 1, 0 and 2, 0, 2 instead. These are exactly the values
generated by a single loop iteration; if they are fed back into the loop body
along with the increased value of $n$, the actual final values listed above are
produced.

As a result, a mere union of the sets tracking the program state updates
possibly occurring in each branch would not be enough for the type system
to be idempotent. The solution is to rather construct every possible alternate
concatenation without repetitions of the lists contained in each set,
which is referred to as merging those sets in the following formalization. In
fact, alternating the state updates performed by each branch in the previous
example produces the actual final values listed above. Since the latest
occurrence of a state update makes any previous occurrence irrelevant for
the final state, repetitions need not be taken into account, which ensures
the finiteness of the construction provided that the sets being merged are
finite. In the special case where the boolean condition can be evaluated at
compile time, considering the picked branch alone is of course enough.

Another case trickier than what one could expect at first glance is that of sequential composition. This is shown by the sample C loop here below, whose body consists of the sequential composition of some assignments with a conditional statement.

```c
for (i = 0; i < 2; i++)
{
    a = 1;
    b = 1;
    if (n % 2)
    {
        a = 2;
        c = 2;
        n++;
    }
    else
    {
        b = 3;
        d = 3;
        n++;
    }
}
```

If the initial value of the integer variable \( n \) is even, the final values of variables \( a, b, c, \) and \( d \) will be 2, 1, 2, 3, whereas if the initial value of \( n \) is odd, the final values of the aforesaid variables will be 1, 3, 2, 3. Assuming that their initial value is 0, the potential final values tracked by considering the sequences of the state updates triggered by the starting assignments with the updates, simulated as described above, possibly triggered by the conditional statement rather are:

- 2, 1, 2, 0,
- 1, 3, 0, 3,
- 2, 3, 2, 3.

The first two tuples of values match the ones generated by a single loop iteration, and produce the actual final values listed above if they are fed back into the loop body along with the increased value of \( n \).

Hence, concatenating the lists tracking the state updates possibly triggered by the first command in the sequence (the starting assignment sequence in the previous example) with the lists tracking the updates possibly triggered by the second command in the sequence (the conditional statement in
the previous example) would not suffice for the type system to be idempotent. The solution is to rather append the latter lists to those constructed by merging the sets tracking the state updates possibly performed by each command in the sequence. Again, provided that such sets are finite, this construction is finite, too. In the special case where the latter set is a singleton, the aforesaid merging is unnecessary, as it would merely insert a preceding occurrence of the single appended list into the resulting concatenated lists, and such repetitions are irrelevant as observed above.

Surprisingly enough, the case of loops is actually simpler than possible first-glance expectations. A loop defines two branches, namely its body and an implicit alternative branch doing nothing. Thus, it can simply be mapped to the union of the set tracking the state updates possibly occurring in its body with a singleton made of the empty list. As happens with conditional statements, in the special case where the boolean condition can be evaluated at compile time, considering the selected branch alone is obviously enough.

Type system \textit{ctyping1} uses the set of lists resulting from this recursion over the input command to construct a set \( F \) of functions of type \textit{vname} \( \Rightarrow \text{val option option} \), as follows: for each list \( \textit{ys} \) in the former set, \( F \) contains the function mapping any variable \( \textit{x} \) to the rightmost occurrence, if any, of pattern \textit{Some v} to which \( \textit{x} \) is mapped by any function in \( \textit{ys} \) (that is, to the latest update, if any, of \( \textit{x} \) tracked in \( \textit{ys} \)), or else to \textit{None}. Then, if \( \textit{A} \), \( \textit{X} \) are the input \textit{state set} and \textit{vname set}, and \( \textit{B} \), \( \textit{Y} \) the output ones:

- \( \textit{B} \) is the set of the program states constructed by picking a function \( \textit{f} \) and a state \( \textit{s} \) from \( \textit{F} \) and \( \textit{A} \), respectively, and mapping any variable \( \textit{x} \) to \( \textit{i} \) if \( \textit{f x = Some (Some i)} \), or else to \( \textit{s x} \) if \( \textit{f x = None} \) (namely, to its value in the initial state \( \textit{s} \) if \( \textit{f} \) marks it as being unchanged).

- \( \textit{Y} \) is \textit{UNIV} if \( \textit{A} = \{\} \) (more on this later), or else the set of the variables not mapped to \textit{Some None} (that is, not marked as being indeterminate) by any function in \( \textit{F} \), and contained in \( \textit{X} \) (namely, being initially determinate) if mapped to \textit{None} (that is, if marked as being unchanged) by some function in \( \textit{F} \).

When can \textit{ctyping1} evaluate the boolean condition of a conditional statement or a loop, so as to possibly detect and discard some “dead” branch? This question can be answered by examining the following sample C loop, where \( \textit{n} \) is a state variable, while integer \( \textit{j} \) is unknown at compile time.

\begin{verbatim}
for (i = 0; i != j; i++)
{
    if (n == 1)
    {
        n = 2;
    }
\end{verbatim}
Assuming that the initial value of \( n \) is 0, its final value will be 0, 1, or 2 according to whether \( j \) matches 0, 1, or any other positive integer, respectively, whereas the loop will not even terminate if \( j \) is negative. Consequently, the type system cannot avoid tracking the state updates possibly triggered in every branch, on pain of failing to be idempotent. As a result, evaluating the boolean conditions in the conditional statement at compile time so as to discard some branch is not possible, even though they only depend on an initially determinate state variable. The conclusion is that ctyping1 may generally evaluate boolean conditions just in case they contain constants alone, namely only if they are trivial enough to be possibly eliminated by program optimization. This is exactly what ctyping1 does by passing any boolean condition found in the input program to the type system btyping1 for boolean expressions, defined here below as well.

```
primrec btyping1 :: bexp ⇒ bool option ((¬) [51]) 55 where
  ⊢ Bc v = Some v |
  ⊢ Not b = (case ⊢ b of
    Some v ⇒ Some (¬ v) | - ⇒ None) |
  ⊢ And b1 b2 = (case (⊢ b1, ⊢ b2) of
    (Some v1, Some v2) ⇒ Some (v1 ∧ v2) | - ⇒ None) |
  ⊢ Less a1 a2 = (if avars a1 ∪ avars a2 = {} then Some (aval a1 (λx. 0) < aval a2 (λx. 0)) else None)

type-synonym state-upd = vname ⇒ val option option

inductive-set ctyping1-merge-aux :: state-upd list set ⇒
  state-upd list set ⇒ (state-upd list × bool) list set
  (infix ⊔ 55) for A and B where
xs ∈ A ⇒ [(xs, True)] ∈ A ⊔ B |
y ∈ B ⇒ [(ys, False)] ∈ A ⊔ B |
[(ws ∈ A ⊔ B; ¬ snd (last ws); xs ∈ A; (xs, True) ∉ set ws)] ⇒
```
\[\text{ws} \circledast \left[ (\text{xs}, \text{True}) \right] \in A \bigcup B\]

\[[\text{ws} \in A \bigcup B; \text{snd} \ (\text{last ws})}; \text{ys} \in B; (\text{ys}, \text{False}) \notin \text{set ws}] \implies \text{ws} \circledast \left[ (\text{ys}, \text{False}) \right] \in A \bigcup B\]

\text{declare} \ ctyping1-merge-aux.intros [intro]

\text{definition} \ ctyping1-append ::
state-upd list set \Rightarrow state-upd list set \Rightarrow state-upd list set
(infixl \circledast \ 55) \where
A \circledast B \equiv \{ \text{xs} \circledast ys \mid \text{xs} \; \text{ys} \in A \land \text{ys} \in B \}

\text{definition} \ ctyping1-merge ::
state-upd list set \Rightarrow state-upd list set \Rightarrow state-upd list set
(infixl \cup \ 55) \where
A \cup B \equiv \{ \text{concat} \ (\text{map} \ \text{fst} \; \text{ws}) \mid \text{ws} \in A \bigcup B \}

\text{definition} \ ctyping1-merge-append ::
state-upd list set \Rightarrow state-upd list set \Rightarrow state-upd list set
(infixl \⊔ \circledast \ 55) \where
A \⊔ \circledast B \equiv (\text{if} \ \text{card} B = \text{Suc} \ 0 \ \text{then} \ A \ \text{else} \ A \cup B) \circledast B

\text{primrec} \ ctyping1-aux :: \text{com} \Rightarrow \text{state-upd list set}
((\vdash -) \ [51] 60) \where
\vdash \text{SKIP} = \{\} \mid
\vdash y ::= a = \{[\lambda x. \text{if} \ x = y \land y \in \text{state}
\text{then} \text{if} \ \text{avars} \ a = \{\} \text{then} \text{Some} \ (\text{Some} (\text{aval} \ (\lambda x. \text{0}))) \text{else} \text{Some} \text{None} \text{else} \text{None}]\} \mid
\vdash c_1;; c_2 = \vdash c_1 \cup \vdash c_2 \mid
\vdash \text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 = (\text{let} \ f = \vdash b \ \text{in}
(\text{if} \ f \in \{\text{Some} \ True, \text{None}\} \ \text{then} \vdash c_1 \ \text{else} \{\}) \cup
(\text{if} \ f \in \{\text{Some} \ False, \text{None}\} \ \text{then} \vdash c_2 \ \text{else} \{\})) \mid
\vdash \text{WHILE} \ b \ \text{DO} \ c = (\text{let} \ f = \vdash b \ \text{in}
(\text{if} \ f \in \{\text{Some} \ False, \text{None}\} \ \text{then} \{\} \ \text{else} \{\}) \cup
(\text{if} \ f \in \{\text{Some} \ True, \text{None}\} \ \text{then} \vdash c \ \text{else} \{\}))

\text{definition} \ ctyping1-seq :: \text{state-upd} \Rightarrow \text{state-upd} \Rightarrow \text{state-upd}
(infixl \ :: \ 55) \where
S ;; T \equiv \lambda x. \text{case} \ T \ x \ \text{of} \ \text{None} \Rightarrow S \ x \mid \text{Some} \ v \Rightarrow \text{Some} \ v

\text{definition} \ ctyping1 :: \text{com} \Rightarrow \text{state} \Rightarrow \text{vname} \Rightarrow \text{config}
((\vdash -) \ (\subseteq, -)) \ [51] 55) \where
\vdash c \ (\subseteq A, X) \equiv \text{let} \ F = \{\lambda x. \text{foldl} \ (;;) \ (\lambda x. \text{None}) \ y x \mid \text{ys} \ y x \in \vdash c \} \ \text{in}
A further building block propaedeutic to the definition of the main type system \( \mathfrak{ctyping}_2 \) is the definition of its own companion type system \( \mathfrak{btyping}_2 \) for boolean expressions. The goal of \( \mathfrak{btyping}_2 \) is splitting, whenever feasible at compile time, an input \( \text{state set} \) into two complementary subsets, respectively comprising the program states making the input boolean expression true or false. This enables \( \mathfrak{ctyping}_2 \) to apply its information flow correctness checks to conditional branches by considering only the program states in which those branches are executed.

As opposed to \( \mathfrak{btyping}_1 \), \( \mathfrak{btyping}_2 \) may evaluate its input boolean expression even if it contains variables, provided that all of their values are known at compile time, namely that all of them are determinate state variables – hence \( \mathfrak{btyping}_2 \), like \( \mathfrak{ctyping}_2 \), needs to take a \( \text{vname set} \) collecting determinate variables as an additional input. In fact, in the case of a loop body, the dirty work of covering any nested branch by skipping the evaluation of non-constant boolean conditions is already done by \( \mathfrak{ctyping}_1 \), so that any \( \text{state set} \) and \( \text{vname set} \) input to \( \mathfrak{btyping}_2 \) already encompass every possible execution flow.

It is eventually time to define the main type system \( \mathfrak{ctyping}_2 \). Its output consists of the \( \text{state set} \) of the final program states and the \( \text{vname set} \) of the finally determinate variables produced by simulating the execution of
the input program, based on the state set of initial program states and the vname set of initially determinate variables taken as inputs, if information flow correctness checks are passed; otherwise, the output is None.

An additional input is the counterpart of the level input to the security type systems formalized in [6], in that it specifies the scope in which information flow correctness is validated. It consists of a set of state set × vname set pairs and a boolean flag. The set keeps track of the variables contained in the boolean conditions, if any, nesting the input program, in association with the program states in which they are evaluated. The flag is False if the input program is nested in a loop, in which case state variables set to non-constant expressions are marked as being indeterminate (as observed previously, the value of a non-constant expression in a loop iteration is in general unknown at compile time).

In the recursive definition of ctyping2, the equations dealing with conditional branches, namely those applying to conditional statements and loops, construct the output state set and vname set respectively as the union and the intersection of the sets computed for each branch. In fact, a possible final state is any one resulting from either branch, and a variable is finally determinate just in case it is such regardless of the branch being picked. Yet, a “dead” branch should have no impact on the determinateness of variables, as it only depends on the other branch. Accordingly, provided that information flow correctness checks are passed, the cases where the output is constructed non-recursively, namely those of SKIP, assignments, and loops, return UNIV as vname set if the input state set is empty. In the case of a loop, the state set and the vname set resulting from one or more iterations of its body are computed using the auxiliary type system ctyping1. This explains why ctyping1 returns UNIV as vname set if the input state set is empty, as mentioned previously.

As happens with the syntax-directed security type system formalized in [6], the cases performing non-recursive information flow correctness checks are those of assignments and loops. In the former case, ctyping2 verifies that the sets of variables contained in the scope, as well as any variable occurring in the expression on the right-hand side of the assignment, are allowed to interfere with the variable on the left-hand side, respectively in their associated sets of states and in the input state set. In the latter case, ctyping2 verifies that the sets of variables contained in the scope, as well as any variable occurring in the boolean condition of the loop, are allowed to interfere with every variable, respectively in their associated sets of states and in the states in which the boolean condition is evaluated. In both cases, if the applying interference relation is unknown as some state variable is indeterminate, each of those checks must be passed for any possible state (unless the respective set of states is empty).

Why do the checks performed for loops test interference with every variable?
The answer is provided by the following sample C program, which sets variables $a$ and $b$ to the terms in the zero-based positions $j$ and $j + 1$ of the Fibonacci sequence.

```c
a = 0;
b = 1;
for (i = 0; i != j; i++)
{
    c = b;
b += a;
a = c;
}
```

The loop in this program terminates for any nonnegative value of $j$. For any variable $x$, suppose that $j$ is not allowed to interfere with $x$ in such an initial state, say $s$. According to the above information flow correctness definition, any initial state $t$ differing from $s$ in the value of $j$ must make execution terminate all the same in order for the program to be correct. However, this is not the case, since execution does not terminate for any negative value of $j$. Thus, the type system needs to verify that $j$ may interfere with $x$, on pain of returning a wrong pass verdict.

The cases that change the scope upon recursively calling the type system are those of conditional statements and loops. In the latter case, the boolean flag is set to False, and the set of state set × vname set pairs is empty as the whole scope nesting the loop body, including any variable occurring in the boolean condition of the loop, must be allowed to interfere with every variable. In the former case, for both branches, the boolean flag is left unchanged, whereas the set of pairs is extended with the pair composed of the input state set (or of UNIV if some state variable is indeterminate, unless the input state set is empty) and of the set of the variables, if any, occurring in the boolean condition of the statement.

Why is the scope extended with the whole input state set for both branches, rather than just with the set of states in which each single branch is selected? Once more, the question can be answered by considering a sample C program, namely a previous one determining the minimum of two integers $a$ and $b$ using a state variable $i$. For the reader’s convenience, the program is reported here below.

```c
i = (a > b) ? 1 : 0;
if (i > 0)
{
    a = b;
}
```
Since the branch changing the value of variable \( a \) is executed just in case \( i = 1 \), suppose that in addition to \( b \), \( i \) also is not allowed to interfere with \( a \) for \( i = 0 \), and let \( s \) be any initial state where \( a \leq b \). Based on the above information flow correctness definition, any initial state \( t \) differing from \( s \) in the value of \( b \) (not bound by the interference of \( i \) with \( a \)) must produce the same final value of \( a \) in order for the program to be correct. However, this is not the case, as the final value of \( a \) will change for any state \( t \) where \( a > b \). Therefore, the type system needs to verify that \( i \) may interfere with \( a \) for \( i = 0 \), too, on pain of returning a wrong \textit{pass} verdict. This is the reason why, as mentioned previously, an information flow correctness policy for this program should be such that \( s \): \textit{dom} \( i \to \textit{dom} \ a \) even for any state \( s \) where \( i = 0 \).

An even simpler example explains why, in the case of an assignment or a loop, the information flow correctness checks described above need to be applied to the set of \textit{state set} \times \textit{vname set} pairs in the scope even if the input \textit{state set} is empty, namely even if the assignment or the loop are nested in a “dead” branch. Here below is a sample C program showing this.

```c
if (i)
{
    a = 1;
}
```

Assuming that the initial value of \( i \) is 0, the assignment nested within the conditional statement is not executed, so that the final value of \( a \) matches the initial one, say 0. Suppose that \( i \) is not allowed to interfere with \( a \) in such an initial state, say \( s \). According to the above information flow correctness definition, any initial state \( t \) differing from \( s \) in the value of \( i \) must produce the same final value of \( a \) in order for the program to be correct. However, this is not the case, as the final value of \( a \) is 1 for any nonzero value of \( i \). Therefore, the type system needs to verify that \( i \) may interfere with \( a \) in state \( s \) even though the conditional branch is not executed in that state, on pain of returning a wrong \textit{pass} verdict.

abbreviation \textit{atyping} :: \textit{bool} \to \textit{acexp} \to \textit{vname set} \to \textit{bool}
\(([.] = \{ \subseteq \}) \[51, 51\] 50)\ where
\( v \mid a \ (\subseteq X) \equiv \textit{avars} \ a = \{\} \lor \textit{avars} \ a \subseteq \textit{state} \cap X \land v \)

definition \textit{univ-states-if} :: \textit{state set} \to \textit{vname set} \to \textit{state set}
\(((\textit{Univ} - -) \[51, 75\] 75)\ where
\( \textit{Univ} \ A \ X \equiv \text{if state} \subseteq X \ then \ A \ else \ \textit{Univ} \ A \ (\subseteq \{\}) \))
fun ctyping2 :: scope ⇒ com ⇒ state set ⇒ vname set ⇒ config option

((- |= - (⊆ - -))) [51, 51] 55) where

- |= SKIP (⊆ A, X) = Some (A, Univ?? A X) |

(U, v) |= x ::= a (⊆ A, X) =
(if (∀ (B, Y) ∈ insert (Univ? A X, avars a) U. B: dom ' Y ↩ {dom x})
then Some (if x ∈ state ∧ A ≠ {})
then if v |= a (⊆ X)
then ({s(x := aval a s) | s, s ∈ A}, insert x X) else (A, X - {x})
else (A, Univ?? A X))
else None) |

(U, v) |= c1; c2 (⊆ A, X) =
(case (U, v) |= c1 (⊆ A, X) of
Some (B, Y) ⇒ (U, v) |= c2 (⊆ B, Y) | - ⇒ None) |

(U, v) |= IF b THEN c1 ELSE c2 (⊆ A, X) =
(case (insert (Univ? A X, bvars b) U, |= b (⊆ A, X)) of (U', B1, B2) ⇒
case ((U', v) |= c1 (⊆ B1, X), (U', v) |= c2 (⊆ B2, X)) of
(Some (C1, Y1), Some (C2, Y2)) ⇒ Some (C1 ∪ C2, Y1 ∩ Y2) |
- ⇒ None) |

(U, v) |= WHILE b DO c (⊆ A, X) = (case |= b (⊆ A, X) of (B1, B2) ⇒
case c (⊆ B1, X) of (C, Y) ⇒ case |= b (⊆ C, Y) of (B1', B2') ⇒
if ∀ (B, W) ∈ insert (Univ? A X ∪ Univ? C Y, bvars b) U.
B: dom ' W ↩ UNIV
then case ({{}, False} |= c (⊆ B1, X), ({}}, False) |= c (⊆ B1', Y)) of
(Some - , Some -) ⇒ Some (B2 ∪ B2', Univ?? B2 X ∩ Y) |
- ⇒ None
else None)
end
end

2 Idempotence of the auxiliary type system meant for loop bodies

theory Idempotence
imports Definitions
begin

The purpose of this section is to prove that the auxiliary type system ctyping1 used to simulate the execution of loop bodies is idempotent, namely that if its output for a given input is the pair composed of state set B and
vname set Y, then the same output is returned if B and Y are fed back into the type system (lemma ctyping1-idem).

2.1 Global context proofs

**Lemma** remdups-filter-last:

\[ \text{last } [x \leftarrow \text{remdups } xs. P x] = \text{last } [x \leftarrow xs. P x] \]

by (induction xs, auto simp: filter-empty-conv)

**Lemma** remdups-append:

\[ \text{set } xs \subseteq \text{set } ys \implies \text{remdups } (xs @ ys) = \text{remdups } ys \]

by (induction xs, simp-all)

**Lemma** remdups-concat-1:

\[ \text{remdups } (\text{concat } (\text{remdups } [])) = \text{remdups } (\text{concat } []) \]

by simp

**Lemma** remdups-concat-2:

\[ \text{remdups } (\text{concat } (\text{remdups } xs)) = \text{remdups } (\text{concat } xs) \implies \]

\[ \text{remdups } (\text{concat } (\text{remdups } (x \# xs))) = \text{remdups } (\text{concat } (x \# xs)) \]

by (simp, subst (2 3) remdups-append2 [symmetric], clarsimp, subst remdups-append, auto)

**Lemma** remdups-concat:

\[ \text{remdups } (\text{concat } (\text{remdups } xs)) = \text{remdups } (\text{concat } xs) \]

by (induction xs, rule remdups-concat-1, rule remdups-concat-2)

2.2 Local context proofs

context noninterf

begin

**Lemma** ctyping1-seq-last:

\[ \text{foldl } (\text{;;}) S xs = (\lambda x. \text{let } xs' = [T \leftarrow xs. T x \neq \text{None}] \text{ in} \]

\[ \text{if } xs' = [] \text{ then } S x \text{ else last } xs' x) \]

by (rule ext, induction xs rule: rev-induct, auto simp: ctyping1-seq-def)

**Lemma** ctyping1-seq-remdups:

\[ \text{foldl } (\text{;;}) S (\text{remdups } xs) = \text{foldl } (\text{;;}) S xs \]

by (simp add: Let-def ctyping1-seq-last, subst remdups-filter-last, simp add: remdups-filter [symmetric])

**Lemma** ctyping1-seq-remdups-concat:

\[ \text{foldl } (\text{;;}) S (\text{concat } (\text{remdups } xs)) = \text{foldl } (\text{;;}) S (\text{concat } xs) \]

by (subst (1 2) ctyping1-seq-remdups-concat [symmetric], simp add: remdups-concat)

**Lemma** ctyping1-seq-eq:

\[ \text{assumes } A: \text{foldl } (\text{;;}) (\lambda x. \text{None}) xs = \text{foldl } (\text{;;}) (\lambda x. \text{None}) ys \]
shows foldl (;) S xs = foldl (;) S ys

proof
  have ∀ x. \([T \leftarrow xs. T x \neq None] = [] \iff [T \leftarrow ys. T x \neq None] = []\) ∧
  last [T \leftarrow xs. T x \neq None] x = last [T \leftarrow ys. T x \neq None] x
  (is ∀ x. (?xs' x = [] \iff ?ys' x = []) ∧ )
  proof
  fix x
  from A have (if ?xs' x = [] then None else last (?xs' x) x) =
  (if ?ys' x = [] then None else last (?ys' x) x)
  by (drule-tac fun-cong [where x = x], auto simp: ctyping1-seq-last)
  moreover have ?xs' x ≠ [] \implies last (?xs' x) x ≠ None
  by (drule last-in-set, simp)
  moreover have ?ys' x ≠ [] \implies last (?ys' x) x ≠ None
  by (drule last-in-set, simp)
  ultimately show (?xs' x = [] \iff ?ys' x = []) ∧
  last (?xs' x) x = last (?ys' x) x
  by (auto split: if-split-asm)
  qed
  thus ?thesis
  by (auto simp: ctyping1-seq-last)
  qed

lemma ctyping1-merge-aux-butlast:
  \([ws \in A \uplus B; butlast ws \neq []] \implies
  snd (last (butlast ws)) = (¬ snd (last ws))\)
  by (erule ctyping1-merge-aux.cases, simp-all)

lemma ctyping1-merge-aux-distinct:
  ws \in A \uplus B \implies distinct ws
  by (induction rule: ctyping1-merge-aux.induct, simp-all)

lemma ctyping1-merge-aux-nonempty:
  ws \in A \uplus B \implies ws ≠ []
  by (induction rule: ctyping1-merge-aux.induct, simp-all)

lemma ctyping1-merge-aux-item:
  \([ws \in A \uplus B; w \in set ws] \implies \text{fst } w \in (\text{if } snd w \text{ then } A \text{ else } B)\)
  by (induction rule: ctyping1-merge-aux.induct, auto)

lemma ctyping1-merge-aux-take-1 [elim]:
  \([\text{take } n \text{ ws } \in A \uplus B; \neg \text{snd } (\text{last ws})]; \text{xs } \in A; (\text{xs, True}) \notin \text{ set ws}] \implies
  \text{take } n \text{ ws } @ \text{take } (n - \text{length ws}) [(\text{xs, True})] \in A \uplus B\)
  by \(\text{cases } n \leq \text{length ws}, \text{auto}\)

lemma ctyping1-merge-aux-take-2 [elim]:
  \([\text{take } n \text{ ws } \in A \uplus B; \text{snd } (\text{last ws})]; \text{ys } \in B; (\text{ys, False}) \notin \text{ set ws}] \implies
  \text{take } n \text{ ws } @ \text{take } (n - \text{length ws}) [(\text{ys, False})] \in A \uplus B\)
  by \(\text{cases } n \leq \text{length ws}, \text{auto}\)
lemma ctyping1-merge-aux-take:
\[ ws \in A \bigcup B; \ 0 < n \] \implies \text{take } n \ ws \in A \bigcup B
by (induction rule: ctyping1-merge-aux.induct, auto)

lemma ctyping1-merge-aux-drop-1 [elim]:
assumes
A: \( xs \in A \) and
B: \( ys \in B \)
shows \( \text{drop } n \ [(xs, True)] \ @ \ [(ys, False)] \in A \bigcup B \)
proof –
from A have \( [(xs, True)] \in A \bigcup B \) ..
with B have \( [(xs, True)] \ @ \ [(ys, False)] \in A \bigcup B \)
by fastforce
with B show \( \text{thesis} \)
by (cases n, auto)
qed

lemma ctyping1-merge-aux-drop-2 [elim]:
assumes
A: \( xs \in A \) and
B: \( ys \in B \)
shows \( \text{drop } n \ [(ys, False)] \ @ \ [(xs, True)] \in A \bigcup B \)
proof –
from B have \( [(ys, False)] \in A \bigcup B \) ..
with A have \( [(ys, False)] \ @ \ [(xs, True)] \in A \bigcup B \)
by fastforce
with A show \( \text{thesis} \)
by (cases n, auto)
qed

lemma ctyping1-merge-aux-drop-3:
assumes
A: \( \forall v. (xs, True) \notin \text{set } (\text{drop } n \ ws) \implies
xs \in A \implies v \implies \text{drop } n \ ws \ @ \ [(xs, True)] \in A \bigcup B \) and
B: \( zs \in A \) and
C: \( ys \in B \) and
D: \( (xs, True) \notin \text{set } ws \) and
E: \( (ys, False) \notin \text{set } (\text{drop } n \ ws) \)
shows \( \text{drop } n \ ws \ @ \ \text{drop } (n - \text{length } ws) \ [(xs, True)] \ @ \ [(ys, False)] \in A \bigcup B \)
proof –
have \( \text{set } (\text{drop } n \ ws) \subseteq \text{set } ws \)
by (rule set-drop-subset)
hence \( \text{drop } n \ ws \ @ \ [(xs, True)] \in A \bigcup B \)
using A and B and D by blast
hence \( \text{drop } n \ ws \ @ \ [(xs, True)] \ @ \ [(ys, False)] \in A \bigcup B \)
using C and E by fastforce

28
thus \text{thesis} \\
using C by (cases \( n \leq \text{length} \ ws \), auto) \\
qed

lemma \texttt{ctyping1-merge-aux-drop-4}:
assumes
\( A: \forall \ ys \ v. \ (ys, \ False) \notin \text{set} \ (\text{drop} \ n \ ws) \implies \)
\( ys \in B \implies \neg v \implies \text{drop} \ n \ ws \ (ys, \ False) \in A \bigcup B \) and
\( B: \ ys \in B \) and
\( C: \ xs \in A \) and
\( D: \ (ys, \ False) \notin \text{set} \ ws \) and
\( E: \ (xs, \ True) \notin \text{set} \ (\text{drop} \ n \ ws) \)
shows \( \text{drop} \ n \ ws \ (ys, \ False) \in A \bigcup B \)
proof
- have \( \text{set} \ (\text{drop} \ n \ ws) \subseteq \text{set} \ ws \)
  by (rule \texttt{set-drop-subset})
  hence \( \text{drop} \ n \ ws \ (ys, \ False) \in A \bigcup B \)
  using \( A \) and \( B \) and \( D \) by blast
  hence \( \text{drop} \ n \ ws \ (ys, \ False) \in A \bigcup B \)
    using \( C \) and \( E \) by \texttt{fastforce}
thus \text{thesis}
  using C by (cases \( n \leq \text{length} \ ws \), auto)
qed

lemma \texttt{ctyping1-merge-aux-drop}:
\( [[ws \in A \bigcup B; \ w \notin \text{set} \ (\text{drop} \ n \ ws)]; \)
\( \text{fst} \ w \in (\text{if} \ \text{snd} \ w \text{ then } A \text{ else } B); \)
\( \text{snd} \ w = (\neg \text{snd} \ (\text{last} \ ws)) \implies \)
\( \text{drop} \ n \ ws \ (w) \in A \bigcup B \)
proof (induction arbitrary: \( w \) rule: \texttt{ctyping1-merge-aux-induct})
  fix \( xs \ ws \ w \)
  show \( [[ws \in A \bigcup B; \)
    \( \wedge \ w, \ w \notin \text{set} \ (\text{drop} \ n \ ws) \implies \)
    \( \text{fst} \ w \in (\text{if} \ \text{snd} \ w \text{ then } A \text{ else } B) \implies \)
    \( \text{snd} \ w = (\neg \text{snd} \ (\text{last} \ ws)) \implies \)
    \( \text{drop} \ n \ ws \ (w) \in A \bigcup B; \)
    \( \neg \text{snd} \ (\text{last} \ ws); \)
    \( xs \in A; \)
    \( (xs, \ True) \notin \text{set} \ ws; \)
    \( w \notin \text{set} \ (\text{drop} \ n \ (ws \ (xs, \ True))); \)
    \( \text{fst} \ w \in (\text{if} \ \text{snd} \ w \text{ then } A \text{ else } B); \)
    \( \text{snd} \ w = (\neg \text{snd} \ (\text{last} \ (ws \ (xs, \ True)))) \implies \)
    \( \text{drop} \ n \ ws \ (ws \ (xs, \ True)) \ (w) \in A \bigcup B \)
by (cases \( w \), auto intro: \texttt{ctyping1-merge-aux-drop-3})
next
  fix \( ys \ ws \ w \)
  show \( [[ws \in A \bigcup B; \)

\[ \forall w, w \notin \text{set} \ (\text{drop} \ n \ ws) \implies \\
\text{fst} \ w \in (\text{if} \ \text{snd} \ w \ \text{then} \ A \ \text{else} \ B) \implies \\
\text{snd} \ w = (\neg \ \text{snd} \ (\text{last} \ ws)) \implies \\
\text{drop} \ n \ ws \ @ [w] \in A \bigcup B; \\
\text{snd} \ (\text{last} ws); \\
ys \in B; \\
(y, False) \notin \text{set} \ ws; \\
w \notin \text{set} \ (\text{drop} \ n \ (ws \ @ [(ys, False)])); \\
\text{fst} \ w \in (\text{if} \ \text{snd} \ w \ \text{then} \ A \ \text{else} \ B); \\
\text{snd} \ w = (\neg \ \text{snd} \ (\text{last} \ (ws \ @ [(ys, False)]))); \\
\text{drop} \ n \ (ws \ @ [(ys, False)]) \ @ [w] \in A \bigcup B \\
\text{by} \ (\text{cases} \ w, \ \text{auto} \ \text{intro: ctyping1-merge-aux-drop-4}) \\
\text{qed}
\]

**lemma ctyping1-merge-aux-closed-1:**

**assumes**

A: \(\forall \ vs. \ \text{length} \ vs \leq \text{length} \ us \implies \)
\((\forall ls, rs. \ vs = ls \ @ rs \implies \ ls \in A \bigcup B \implies rs \in A \bigcup B \implies \\
(\exists ws \in A \bigcup B. \ \text{foldl} \ (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ ws)) = \\
\text{foldl} \ (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ (ls \ @ rs))) \wedge \\
\text{length} \ ws \leq \text{length} \ (ls \ @ rs) \wedge \text{snd} \ (\text{last} \ ws) = \text{snd} \ (\text{last} \ rs)) \\
(\exists ws \in . . .) \) and

B: \(us \in A \bigcup B\) and

**shows** \(\exists ws \in A \bigcup B. \ \text{foldl} \ (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ ws)) = \\
\text{foldl} \ (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ (us \ @ [w]))) \wedge \\
\text{length} \ ws \leq \text{Suc} \ (\text{length} \ us) \wedge \text{snd} \ (\text{last} \ ws) = \text{snd} \ v\)

**proof** (\(\text{cases} \ v \in \text{set} \ us, \ \text{cases} \ hd \ us = v\))

**assume** E: \(hd \ us = v\)

**moreover have** distinct us

**using** B by (\text{rule ctyping1-merge-aux-distinct})

**ultimately have** \(v \notin \text{set} \ (\text{drop} \ (\text{Suc} \ 0) \ us)\)

**by** (\text{cases us, simp-all})

**with** B **have** \(\text{drop} \ (\text{Suc} \ 0) \ us \ @ [v] \in A \bigcup B\)

**is ?ws in . . .**

**using** C and D **by** (\text{rule ctyping1-merge-aux-drop})

**moreover have** \(\text{foldl} \ (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ ?ws)) = \\
\text{foldl} \ (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ (us \ @ [v]))) \\
\text{using} \ E \ \text{by} \ (\text{cases us, simp, subst} \ (1 \ 2) \ ctyping1-seq-remdups-concat \\
[\text{symmetric}], \text{simp}) \\
\text{ultimately show} \ ?\text{thesis}\)

**by** fastforce

**next**

**assume** \(v \in \text{set} \ us\)

**then obtain** ls and rs **where** E: \(us = ls \ @ v \neq rs \wedge v \notin \text{set} \ rs\)

**by** (\text{blast dest: split-list-last})

**moreover assume** \(hd \ us \neq v\)

30
ultimately have \( ls \neq [] \)
  by \( \{ \text{cases } ls, \text{ simp-all} \} \)
hence take (length \( ls \)) \( us \in A \bigcup B \)
  by \( \{ \text{simp add: ctyping1-merge-aux-take } B \} \)
moreover have \( v \notin \text{set (drop (Suc (length ls)) us)} \)
  using \( E \) by simp
with \( B \) have drop (Suc (length \( ls \))) \( us @ [v] \in A \bigcup B \)
  using \( C \) and \( D \) by \( \{ \text{rule ctyping1-merge-aux-drop} \} \)
ultimately have \( \exists us @ A \bigcup B. \ ?P us ls \ (rs @ [v]) \)
  using \( A \) and \( E \) by \( \{ \text{drule-tac spec [of - ls @ rs @ [v]},
\text{ simp, drule-tac spec [of - ls], simp} \} \)
moreover have \( \text{foldl } (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ (ls @ rs @ [v]))) \)
  \( = \)
  \( \text{foldl } (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ (us @ [v]))) \)
  \( \land \)
  length \( \ (us @ [v]) \leq \text{Suc} \ (\text{length } us) \) \( \land \)
  \( \text{snd} \ (\text{last} \ (us @ [v])) \ = \ \text{snd} \ v \)
  by simp
next
from \( B \) and \( C \) and \( D \) and \( E \) show \( us @ [v] \in A \bigcup B \)
  by \( \{ \text{cases } v, \text{ cases snd (last } us), \text{ auto} \} \)
qed

**lemma** ctyping1-merge-aux-closed:

**assumes**
\( A: \forall xs @ A. \forall ys @ A. \exists zs @ A. \)
  \( \text{foldl } (;) (\lambda x. \text{None}) zs = \text{foldl } (;) (\lambda x. \text{None}) (xs @ ys) \) \( \land \)
\( B: \forall xs @ B. \forall ys @ B. \exists zs @ B. \)
  \( \text{foldl } (;) (\lambda x. \text{None}) zs = \text{foldl } (;) (\lambda x. \text{None}) (xs @ ys) \)

**shows** \( [us @ A \bigcup B; vs @ A \bigcup B] \Longrightarrow \)
\( \exists ws @ A \bigcup B. \text{foldl } (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ ws)) \)
  \( = \)
  \( \text{foldl } (;) (\lambda x. \text{None}) (\text{concat} (\text{map} \ \text{fst} \ (us @ ws))) \) \( \land \)
  length \( ws \leq \text{length} \ (us @ vs) \) \( \land \)
  \( \text{snd} \ (\text{last } ws) = \text{snd} \ (\text{last } vs) \)
\( (\text{is } [\_ ; ] \Longrightarrow \exists ws \in -. \ ?P us vs vs) \)

**proof** \( \{ \text{induction } us @ vs \text{ arbitrary: } us @ vs \text{ rule: length-induct} \} \)

**fix** \( us \) \( vs \)

**let** \( \_ = \text{foldl } (;) (\lambda x. \text{None}) \)

**assume**
\( C: \forall ts. \text{length } ts < \text{length} \ (us @ vs) \Longrightarrow \)
\( (\forall ls rs. \ ts = ls @ rs \Longrightarrow ls @ A \bigcup B \Longrightarrow rs @ A \bigcup B \Longrightarrow \)
\( (\exists \, ws \in A \uplus B. \, ?f \, (\text{concat} \, (\text{map} \, \text{fst} \, ws)) = \)
\( \neg f \, (\text{concat} \, (\text{map} \, \text{fst} \, (\text{us} \, @ \, rs))) \land \)
\( \text{length} \, ws \leq \text{length} \, (\text{ls} \, @ \, rs) \land \text{snd} \, (\text{last} \, ws) = \text{snd} \, (\text{last} \, rs)) \)
\( (\forall \, \cdot. \, - \rightarrow (\forall \, rs \, . \, - \rightarrow - \rightarrow - \rightarrow (\exists \, ws \in - . \, ?Q \, ws \, ls \, rs)) \) and
\( D: \, us \in A \uplus B \) and
\( E: \, vs \in A \uplus B \)
\{ 
  \text{fix} \, vs' \, v 
  \text{assume} \, F: \, vs = vs' \, @ \, [v] 
  \text{have} \, (\exists \, ws \in A \uplus B. \, ?f \, (\text{concat} \, (\text{map} \, \text{fst} \, ws)) = \)
  \( ?f \, (\text{concat} \, (\text{map} \, \text{fst} \, (\text{us} \, @ \, vs' \, @ \, [v]))) \) \land 
  \text{length} \, ws \leq \text{Suc} \, (\text{length} \, us \, + \, \text{length} \, vs') \land \text{snd} \, (\text{last} \, ws) = \text{snd} \, v 
  \text{proof} \, (\text{cases} \, vs', \, \text{cases} \, (\neg \, \text{snd} \, (\text{last} \, us)) = \text{snd} \, v) 
  \text{assume} \, vs' = [] \land (\neg \, \text{snd} \, (\text{last} \, ws)) = \text{snd} \, v 
  \text{thus} \, ?\text{thesis} 
  \text{using} \, \text{ctyping1-merge-aux-closed-1} \, [OF \, - \, D] \) and 
  \text{ctyping1-merge-aux-item} \, [OF \, E] \) and \text{C} and \text{F} 
  \text{by} \, (\text{auto} \, \text{simp: less}\text{-Suc-eq-le}) 
\}
\text{next} 
\text{have} \, G: \, \text{us} \neq [] 
\text{using} \, D \text{ by} \, (\text{rule ctyping1-merge-aux-nonempty}) 
\text{hence} \, \text{fst} \, (\text{last} \, us) \in (\text{if} \, \text{snd} \, (\text{last} \, us) \text{ then} \, A \text{ else} \, B) 
\text{using} \, \text{ctyping1-merge-aux-item} \text{ and} \, D \text{ by} \, \text{auto} 
\text{moreover assume} \, H: \, (\neg \, \text{snd} \, (\text{last} \, us)) \neq \text{snd} \, v 
\text{ultimately have} \, \text{fst} \, (\text{last} \, us) \in (\text{if} \, \text{snd} \, v \text{ then} \, A \text{ else} \, B) 
\text{by} \, \text{simp} 
\text{moreover have} \, \text{fst} \, v \in (\text{if} \, \text{snd} \, v \text{ then} \, A \text{ else} \, B) 
\text{using} \, \text{ctyping1-merge-aux-item} \text{ and} \, E \text{ and} \, F \text{ by} \, \text{auto} 
\text{ultimately have} \, (\exists \, zs \in \text{if} \, \text{snd} \, v 
\text{then} \, A \text{ else} \, B. \, ?zs = \, ?f \, (\text{concat} \, (\text{map} \, \text{fst} \, [\text{last} \, us, \, v]))) 
\text{is} \, \exists \, zs \in - . \, ?R \, zs) 
\text{using} \, A \text{ and} \, B \text{ by} \, \text{auto} 
\text{then obtain} \, zs \text{ where} 
\text{I:} \, zs \in (\text{if} \, \text{snd} \, v \text{ then} \, A \text{ else} \, B) \) and \text{J:} \, ?R \, zs . . 
\text{let} \, ?w = (zs, \, \text{snd} \, v) 
\text{assume} \, K: \, vs' = [] 
\{ 
  \text{fix} \, us' \, u 
  \text{assume} \, \text{Cons:} \, \text{butlast} \, us = u \neq us' 
  \text{hence} \, L: \, \text{snd} \, v = (\neg \, \text{snd} \, (\text{last} \, (\text{butlast} \, us))) 
  \text{using} \, D \text{ and} \, H \text{ by} \, (\text{drule-tac ctyping1-merge-aux-butlast, simp-all}) 
  \text{let} \, ?S = \, ?f \, (\text{concat} \, (\text{map} \, \text{fst} \, (\text{butlast} \, us))) 
  \text{have} \, \text{take} \, (\text{length} \, (\text{butlast} \, us)) \, \text{us} \in A \uplus B 
  \text{using} \, \text{Cons} \text{ by} \, (\text{auto intro: ctyping1-merge-aux-take} \, [OF \, D]) 
  \text{hence} \, M: \, \text{butlast} \, us \in A \uplus B 
  \text{by} \, (\text{subst (asm) (2) append-butlast-last-id} \, [OF \, G, \, \text{symmetric}, \, \text{simp}) 
  \text{have} \, N: \, \forall \, ts. \, \text{length} \, ts < \text{length} \, (\text{butlast} \, us \, @ \, [\text{last} \, us, \, v]) \rightarrow 
  (\forall \, \cdot. \, ts = \text{ls} \, @ \, rs \rightarrow \text{ls} \in A \uplus B \rightarrow \text{rs} \in A \uplus B \rightarrow 
  (\exists \, ws \in A \uplus B. \, ?Q \, ws \, ls \, rs)) \)
using C and F and K by (subst (asm) append-butlast-last-id [OF G, symmetric], simp)

have \( \exists ws \in A \bigcup B. \ ?f (\text{concat} (\text{map} \ \text{fst} \ ws)) = \ ?f (\text{concat} (\text{map} \ \text{fst} (\text{butlast} \ us \ @ [?w]))) \land \text{length} \ ws \leq \text{Suc} (\text{length} (\text{butlast} \ us)) \land \text{snd} (\text{last} \ ws) = \text{snd} \ ?w \)

proof (rule ctyping1-merge-aux-closed-1)

show \( \forall ts. \text{length} \ ts \leq \text{length} (\text{butlast} \ us) \rightarrow \)

\( (\forall ls \ rs. \ ts = ls @ rs \rightarrow ls \in A \bigcup B \rightarrow rs \in A \bigcup B \rightarrow \)

\( (\exists ws \in A \bigcup B. \ ?Q \ ws \ ls \ rs) \)

using N by force

next

from M show butlast us \( \in A \bigcup B \).

next

show \( \text{fst} (zs, \text{snd} \ v) \in (\text{if} \ \text{snd} (zs, \text{snd} \ v) \text{ then } A \text{ else } B) \)

using I by simp

next

show \( \text{snd} (zs, \text{snd} \ v) = (\neg \text{snd} (\text{last} (\text{butlast} \ us))) \)

using L by simp

qed

moreover have foldl (;;;;) \( ?S \ zs = \)

foldl (;;;;) \( ?S (\text{concat} (\text{map} \ \text{fst} [\text{last} \ us, v])) \)

using J by (rule ctyping1-seq-eq)

ultimately have \( \exists ws \in A \bigcup B. \ ?f (\text{concat} (\text{map} \ \text{fst} ws)) = \ ?f (\text{concat} (\text{map} \ \text{fst} ((\text{butlast} \ us @ [\text{last} \ us]) @ [v]))) \land \)

\( \text{length} \ ws \leq \text{Suc} (\text{length} \ us) \land \text{snd} (\text{last} \ ws) = \text{snd} \ v \)

by auto

} with K and I and J show \(?thesis \)

by (simp, subst append-butlast-last-id [OF G, symmetric], cases butlast us, (force split: if-split-asm)+

next

case Cons

hence take (length vs') vs' \( \in A \bigcup B \)

by (auto intro: ctyping1-merge-aux-take [OF E])

hence vs' \( \in A \bigcup B \)

using F by simp

then obtain ws where G: \( \text{ws} \in A \bigcup B \text{ and } H: \ ?Q \ \text{ws} \ \text{vs}' \)

using C and D and F by force

have I: \( \forall ts. \text{length} \ ts \leq \text{length} \ ws \rightarrow \)

\( (\forall ls \ rs. \ ts = ls @ rs \rightarrow ls \in A \bigcup B \rightarrow rs \in A \bigcup B \rightarrow \)

\( (\exists ws \in A \bigcup B. \ ?Q \ ws \ ls \ rs) \)

proof (rule allI, rule impI)

fix ts :: (state-upd list \times bool) list

assume J: \( \text{length} \ ts \leq \text{length} \ ws \)

show \( \forall ls \ rs. \ ts = ls @ rs \rightarrow ls \in A \bigcup B \rightarrow rs \in A \bigcup B \rightarrow \)

\( (\exists ws \in A \bigcup B. \ ?Q \ ws \ ls \ rs) \)

proof (rule spec [OF C, THEN mp])

show length ts < length (us @ vs)

using F and H and J by simp

33
qed

hence \( J : \text{snd \ (last \ (\text{butlast \ vs}))} = (\neg \ \text{snd \ (last \ vs))} \)
by (metis E F Cons butlast-snoc ctyping1-merge-aux-butlast
list.distinct(I))

have \( \exists \text{ ws' } \in A \biguplus B. \ \text{\textbullet \ (concat \ (map \ \text{fst} \ \text{ws'}))} = \)
\( \text{\textbullet \ (\text{concat} \ (\text{map} \ \text{fst} \ (\text{ws} \ @ \ [v])))} \) \( \wedge \)
\( \text{length \ ws'} \leq \text{Suc} \ (\text{length} \ \text{ws}) \wedge \text{snd} \ (\text{last} \ \text{ws'}) = \text{snd} \ v \)
proof (rule ctyping1-merge-aux-closed-1 [OF I G])
show \( \text{fst} \ v \in (\text{if} \ \text{snd} \ v \ \text{then} \ A \ \text{else} \ B) \)
by (rule ctyping1-merge-aux-item [OF E], simp add: F)
next
show \( \text{snd} \ v = (\neg \ \text{snd \ (last \ \text{ws})}) \)
using F and H and J by simp
qed
thus \( \text{?thesis} \)
using H by auto
qed

note \( F = \text{this} \)
show \( \exists \ \text{ws } \in A \biguplus B. \ ? P \ \text{ws } \ \text{us } \ \text{vs} \)
proof (rule rev-cases [of vs])
assume \( \text{vs } = \[] \)
thus \( \text{?thesis} \)
by (simp add: ctyping1-merge-aux-nonempty [OF E])
next
fix \( \text{vs'} \ v \)
assume \( \text{vs } = \text{vs'} \ @ \ [v] \)
thus \( \text{?thesis} \)
using F by simp
qed
qed

lemma ctyping1-merge-closed:

assumes
\( A : \forall xs \in A. \ \forall ys \in A. \ \exists zs \in A. \) \( \text{foldl} \ (;;) (\lambda x. \ \text{None}) \ \text{zs} = \text{foldl} \ (;;) (\lambda x. \ \text{None}) \ (xs \ @ \ ys) \) \( \text{and} \)
\( B : \forall xs \in B. \ \forall ys \in B. \ \exists zs \in B. \) \( \text{foldl} \ (;;) (\lambda x. \ \text{None}) \ \text{zs} = \text{foldl} \ (;;) (\lambda x. \ \text{None}) \ (xs \ @ \ ys) \) \( \text{and} \)
\( C : \ \text{xs } \in A \biguplus B \) \( \text{and} \)
\( D : \ \text{ys } \in A \biguplus B \)
shows \( \exists zs \in A \biguplus B. \ \text{foldl} \ (;;) (\lambda x. \ \text{None}) \ \text{zs} = \)
\( \text{foldl} \ (;;) (\lambda x. \ \text{None}) \ (xs \ @ \ ys) \)
proof
let \( \text{?f } = \text{foldl} \ (;;) (\lambda x. \ \text{None}) \)
obtain us where us \( \in A \biguplus B \) \( \text{and} \)
E: \( \text{zs } = \text{concat} \ (\text{map} \ \text{fst} \ \text{us}) \)
using C by (auto simp: ctyping1-merge-def)
moreover obtain \( vs \) where \( vs \in A \cup B \) and

\[ F \colon \text{ys} = \text{concat} \ (\text{map} \ \text{fst} \ vs) \]

using \( D \) by \( (\text{auto simp: ctyping1-merge-def}) \)

ultimately have \( \exists vs \in A \cup B. \ \section{foldl} \ (\text{concat} \ (\text{map} \ \text{fst} \ vs)) = \section{foldl} \ (\text{concat} \ (\text{map} \ \text{fst} \ (\text{us} \ @ \ vs))) \) \wedge

\[ \text{length} \ vs \leq \text{length} \ (\text{us} \ @ \ vs) \wedge \text{snd} \ (\text{last} \ vs) = \text{snd} \ (\text{last} \ vs) \]

using \( A \) and \( B \) by \( (\text{blast intro: ctyping1-merge-aux-closed}) \)

then obtain \( ws \) where \( ws \in A \cup B \) and

\[ \section{foldl} \ (\text{concat} \ (\text{map} \ \text{fst} \ ws)) = \section{foldl} \ (\text{concat} \ (\text{map} \ \text{fst} \ ws)) \]

using \( E \) and \( F \) by auto

thus \( \section{thesis} \)

by \( (\text{auto simp: ctyping1-merge-def}) \)

qed

lemma ctyping1-merge-append-closed:

assumes

\[ A \colon \forall xs \in A. \forall ys \in A. \exists zs \in A. \section{foldl} \ (\lambda x. \text{None}) \ zs = \section{foldl} \ (\lambda x. \text{None}) \ (xs \ @ \ ys) \]

\( B \colon \forall xs \in B. \forall ys \in B. \exists zs \in B. \section{foldl} \ (\lambda x. \text{None}) \ zs = \section{foldl} \ (\lambda x. \text{None}) \ (xs \ @ \ ys) \)

\( C \colon zs \in A \cup_B B \)

\( D \colon ys \in A \cup_B B \)

shows \( \exists zs \in A \cup_B B. \section{foldl} \ (\lambda x. \text{None}) \ zs = \section{foldl} \ (\lambda x. \text{None}) \ (xs \ @ \ ys) \)

proof –

let \( \section{foldl} \ (\lambda x. \text{None}) \)

{ }

assume \( E \colon \text{card} \ B = \text{Suc} \ 0 \)

moreover from \( C \) and this obtain \( as \) bs where

\[ xs = as \ @ bs \wedge as \in A \wedge bs \in B \]

by \( (\text{auto simp: ctyping1-append-def ctyping1-merge-append-def}) \)

moreover from \( D \) and \( E \) obtain \( as' \) bs' where

\[ ys = as' \ @ bs' \wedge as' \in A \wedge bs' \in B \]

by \( (\text{auto simp: ctyping1-append-def ctyping1-merge-append-def}) \)

ultimately have \( F \colon \exists as \subseteq A \wedge bs \subseteq B \)

by \( (\text{auto simp: card-1-singleton-iff}) \)

hence \( \exists as \subseteq A \wedge bs \subseteq B \)

by \( (\text{simp add: ctyping1-seq-remdups}) \)

also have \( \exists as \subseteq A \wedge bs \subseteq B \)

by \( (\text{simp add: ctyping1-seq-remdups}) \)

finally have \( G \colon \exists as \subseteq A \wedge bs \subseteq B \)

by \( (\text{simp add: ctyping1-seq-remdups}) \)

obtain \( as'' \) where \( H \colon \exists as'' \subseteq A \wedge I \colon \exists as'' \subseteq A \wedge as'' \subseteq bs \)

using \( A \) and \( F \) by auto

have \( \exists zs \subseteq A \wedge bs \subseteq B \)

by \( (\text{simp add: ctyping1-seq-remdups}) \)

show \( \exists zs \subseteq A \wedge bs \subseteq B \)

proof \( (\text{rule bexI [of - as'' \ @ bs]}) \)

show \( \exists zs \subseteq A \wedge bs \subseteq B \)

proof \( (\text{rule bexI [of - as'' \ @ bs]}) \)

show \( \exists zs \subseteq A \wedge bs \subseteq B \)

proof \( (\text{rule bexI [of - as'' \ @ bs]}) \)
using $G$ and $I$ by simp

next
  show as'' @ bs ∈ A @ B
  using F and $H$ by (auto simp: ctyping1-append-def)
qed
}

moreover {
  fix $n$
  assume $E$: card $B ≠ Suc 0$
  moreover from $C$ and this obtain ws bs where
  $xs = ws @ bs ∧ ws ∈ A ⊔ B ∧ bs ∈ B$
  by (auto simp: ctyping1-append-def ctyping1-merge-append-def)
  moreover from $D$ and $E$ obtain ws' bs' where
  $ys = ws' @ bs' ∧ ws' ∈ A ⊔ B ∧ bs' ∈ B$
  by (auto simp: ctyping1-append-def ctyping1-merge-append-def)
  ultimately have $F$: $xs @ ys = ws @ bs @ ws' @ bs'$ ∧
  ${ws, ws'} ⊆ A ⊔ B ∧ {bs, bs'} ⊆ B$
  by simp
  hence $[(bs, False)] ∈ A ⊔ B$
  by blast
  hence $G$: $bs ∈ A ⊔ B$
  by (force simp: ctyping1-merge-def)
  have $∃ vs ∈ A ⊔ B. ?f vs = ?f (ws @ bs)$
  by simp
  (is $∃ vs ∈ _. ?P vs ws bs$)
proof (rule ctyping1-merge-closed)
  show $∀ xs ∈ A. ∀ ys ∈ A. ∃ zs ∈ A. foldl (;) (λx. None) (xs @ ys)$
  using A by simp
next
  show $∀ xs ∈ B. ∀ ys ∈ B. ∃ zs ∈ B. foldl (;) (λx. None) (xs @ ys)$
  using B by simp
next
  show ws ∈ A ⊔ B
  using F by simp
next
  from $G$ show bs ∈ A ⊔ B.
qed
then obtain vs where $H$: $vs ∈ A ⊔ B$ and $I$: ?P vs ws bs ..
  have $∃ vs' ∈ A ⊔ B. ?P vs' vs ws'$
proof (rule ctyping1-merge-closed)
  show $∀ xs ∈ A. ∀ ys ∈ A. ∃ zs ∈ A. foldl (;) (λx. None) (xs @ ys)$
  using A by simp
next
  show $∀ xs ∈ B. ∀ ys ∈ B. ∃ zs ∈ B. foldl (;) (λx. None) (xs @ ys)$
  using B by simp
next

36
from $H$ show $vs \in A \sqcup B$.

next

show $ws' \in A \sqcup B$

using $F$ by simp

qed

then obtain $vs$ where $J$: $vs \in A \sqcup B$ and $K$: $?P vs \in ws'$ ..

have $\exists zs \in A \sqcup B$. $?f zs = ?f (xs @ ys)$

proof (rule bexI [of - vs' @ bs'])

show $foldl (;;) (\lambda x. None) (us' @ bs') =$

$foldl (;;) (\lambda x. None) (xs @ ys)$

using $F$ and $I$ and $K$ by simp

next

show $vs' @ bs' \in A \sqcup B @ B$

using $F$ and $J$ by (auto simp: ctyping1-append-def)

qed

ultimately show $?thesis$

using $A$ and $B$ and $C$ and $D$ by (auto simp: ctyping1-merge-append-def)

qed

lemma $ctyping1$-aux-closed:

$[zs \in \vdash c; ys \in \vdash c] \implies \exists zs \in \vdash c. foldl (;;) (\lambda x. None) zs =$

$foldl (;;) (\lambda x. None) (xs @ ys)$

by (induction $c$ arbitrary: $xs$ $ys$, auto

intro: ctyping1-merge-closed ctyping1-merge-append-closed
simpl: Let-def ctyping1-seq-def simp del: foldl-append)

lemma $ctyping1$-idem-1:

assumes

$A$: $s \in A$ and
$B$: $xs \in \vdash c$ and
$C$: $ys \in \vdash c$

shows $\exists f r.$

$(\exists t.$

$(\lambda x. case foldl (;;) (\lambda x. None) ys x of

None \Rightarrow case foldl (;;) (\lambda x. None) zs x of

None \Rightarrow s x | Some None \Rightarrow t' x | Some (Some $i$) \Rightarrow i |$

Some None \Rightarrow t'' x | Some (Some $i$) \Rightarrow i =

$(\lambda x. case f x of$

None \Rightarrow r x | Some None \Rightarrow t x | Some (Some $i$) \Rightarrow i) \land$

$(\exists zs. f = foldl (;;) (\lambda x. None) zs \land zs \in \vdash c) \land

r \in A$

proof

let $?f = foldl (;;) (\lambda x. None)$

let $?t = \lambda x. case $?f ys x of

None \Rightarrow case $?f zs x of Some None \Rightarrow t' x | - \Rightarrow (0 :: val) |$

Some None \Rightarrow t'' x | - \Rightarrow 0$

have $\exists zs \in \vdash c. $?f zs = $?f (xs @ ys)$
using $B$ and $C$ by (rule $ctyping1$-aux-closed)
then obtain $zs$ where $zs \in \vdash c$ and $?f \, zs = ?f \, (xs @ ys)$ ..
with $A$ show $\negthesis$
by (rule-tac $exI$ [of $- ?f \, zs$], rule-tac $exI$ [of $- s$],
rule-tac $conjI$, rule-tac $exI$ [of $- ?t$], fastforce dest: last-in-set
simp: Let-def $ctyping1$-seq-last split: option.split, blast)
qued

lemma $ctyping1$-idem-2:

assumes
$A$: $s \in A$ and
$B$: $xs \in \vdash c$

shows $\exists f \, r$.
$(\exists t.
(\lambda x. \text{case foldl} (;;) (\lambda x. \text{None}) \, xs \, x \, of
None \Rightarrow s \, x \mid \text{Some} \, None \Rightarrow t' \, x \mid \text{Some} \, (\text{Some} \, i) \Rightarrow i) =$
$(\lambda x. \text{case} \, f \, x \, of
None \Rightarrow r \, x \mid \text{Some} \, None \Rightarrow t \, x \mid \text{Some} \, (\text{Some} \, i) \Rightarrow i) \land$
$(\exists xs. \, f = \text{foldl} (;;) (\lambda x. \text{None}) \, xs \land xs \in \vdash c) \land$
$(\exists f \, s.
$(\exists t. \, r = (\lambda x. \text{case} \, f \, x \, of
None \Rightarrow s \, x \mid \text{Some} \, None \Rightarrow t \, x \mid \text{Some} \, (\text{Some} \, i) \Rightarrow i) \land$
$(\exists xs. \, f = \text{foldl} (;;) (\lambda x. \text{None}) \, xs \land xs \in \vdash c) \land$
$s \in A)$)

proof
let $?f = \text{foldl} (;;) (\lambda x. \text{None})$
let $?g = \lambda f \, s \, t \, x. \text{case} \, f \, x \, of
None \Rightarrow s \, x \mid \text{Some} \, None \Rightarrow t \, x \mid \text{Some} \, (\text{Some} \, i) \Rightarrow i$

show $\negthesis$
by (rule $exI$ [of $- ?f \, xs$], rule $exI$ [of $- ?g \, (\text{if} \, xs) \, s \, t'$],
(fastforce simp: $A \, B$ split: option.split)+)
qued

lemma $ctyping1$-idem:
$\vdash c \subseteq (A, X) = (B, Y) \Longrightarrow \vdash c \subseteq (B, Y) = (B, Y)$
by (cases $A = \{\}$, auto simp: $ctyping1$-def
intro: $ctyping1$-idem-1 $ctyping1$-idem-2)

end

end

3 Overapproximation of program semantics by the type system

theory $Overapproximation$
imports $Idempotence$
begin
The purpose of this section is to prove that type system $ctyping2$ overapproximates program semantics, namely that if (a) $(c, s) \Rightarrow t$, (b) the type system outputs a state set $B$ and a vname set $Y$ when it is input program $c$, state set $A$, and vname set $X$, and (c) state $s$ agrees with a state in $A$ on the value of every state variable in $X$, then $t$ must agree with some state in $B$ on the value of every state variable in $Y$ (lemma $ctyping2$-approx).

This proof makes use of the lemma $ctyping1$idem proven in the previous section.

### 3.1 Global context proofs

**Lemma avars-aval:**

$s = t (\subseteq avars a) \Rightarrow \text{aval a } s = \text{aval a } t$

by (induction a, simp-all)

### 3.2 Local context proofs

**Context noninterf:**

**Lemma interf-set-mono:**

$[A' \subseteq A; X \subseteq X'; \forall (B', Y') \in U'; \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y;\forall (B, Y) \in \text{insert (Univ? A X, Z)} U. B: \text{dom} ^' Y \rightsquigarrow W] \Rightarrow \forall (B, Y) \in \text{insert (Univ? A' X', Z)} U'. B: \text{dom} ^' Y \rightsquigarrow W$

by (subgoal-tac Univ? A' X' \subseteq Univ? A X, fastforce, auto simp: univ-states-if-def)

**Lemma btyping1-btyping2-aux-1 [elim]:**

assumes

$A: \text{avars a}_1 = \{\}$ and
$B: \text{avars a}_2 = \{\}$ and
$C: \text{aval a}_1 (\lambda x. 0) < \text{aval a}_2 (\lambda x. 0)$

shows $\text{aval a}_1 s < \text{aval a}_2 s$

**Proof** –

have $\text{aval a}_1 s = \text{aval a}_1 (\lambda x. 0) \land \text{aval a}_2 s = \text{aval a}_2 (\lambda x. 0)$

using $A$ and $B$ by (blast intro: avars-aval)

thus ?thesis

using $C$ by simp

qed

**Lemma btyping1-btyping2-aux-2 [elim]:**

assumes

$A: \text{avars a}_1 = \{\}$ and
$B: \text{avars a}_2 = \{\}$ and
$C: \neg \text{aval a}_1 (\lambda x. 0) < \text{aval a}_2 (\lambda x. 0)$ and
\[ D: \text{aval} \ a_1 \ s < \text{aval} \ a_2 \ s \]
shows \( \text{False} \)
proof
have \( \text{aval} \ a_1 \ s = \text{aval} \ a_1 \ (\lambda x. \ 0) \wedge \text{aval} \ a_2 \ s = \text{aval} \ a_2 \ (\lambda x. \ 0) \)
using \( A \) and \( B \) by (blast intro: avars-aval)
thus \( \forall \text{thesis} \)
using \( C \) and \( D \) by simp
qed

lemma \( \text{btyping1-btyping2-aux} \):
\( \vdash b = \text{Some} \ v \implies \vdash b \ (\subseteq \ A, X) = \text{Some} \ (\text{if} \ v \ \text{then} \ A \ \text{else} \ \{\}) \)
by (induction \( b \) arbitrary; \( v \), auto split: if-split-asm option.split-asm)

lemma \( \text{btyping1-btyping2} \):
\( \vdash b = \text{Some} \ v \implies \vdash b \ (\subseteq \ A, X) = (\text{if} \ v \ \text{then} \ (A, \{\}) \ \text{else} \ (\{\}, \ A)) \)
by (simp add: btyping2-def btyping1-btyping2-aux)

lemma \( \text{btyping2-aux-subset} \):
\( \vdash b \ (\subseteq \ A, X) = \text{Some} \ A' \implies A' = \{s. \ s \in A \wedge \text{bval} \ b \ s\} \)
by (induction \( b \) arbitrary; \( A' \), auto split: if-split-asm option.split-asm)

lemma \( \text{btyping2-aux-diff} \):
\( \exists B'. \vdash b \ (\subseteq \ A, X) = \text{Some} \ B' \land A' \subseteq A; B' \subseteq B \implies A' - B' \subseteq A - B \)
by (blast dest: btyping2-aux-subset)

lemma \( \text{btyping2-aux-mono} \):
\( \exists B'. \vdash b \ (\subseteq \ A, X) = \text{Some} \ B' \land B' \subseteq B \)
by (induction \( b \) arbitrary; \( B' \), auto dest: btyping2-aux-diff split: if-split-asm option.split-asm)

lemma \( \text{btyping2-mono} \):
\( \exists B'. \vdash b \ (\subseteq \ A, X) = (B_1', B_2'); A' \subseteq A; X \subseteq X' \implies B_1' \subseteq B_1 \land B_2' \subseteq B_2 \)
by (simp add: btyping2-def split: option.split-asm, frule-tac \( \exists - \forall \) btyping2-aux-mono, auto dest: btyping2-aux-subset)

lemma \( \text{btyping2-fst-empty} \):
\( \vdash b \ (\subseteq \ \{\}, X) = (\{\}, \{\}) \)
by (auto simp: btyping2-def dest: btyping2-aux-subset split: option.split-asm)

lemma \( \text{btyping2-aux-eq} \):
\( \vdash b \ (\subseteq \ A, X) = \text{Some} \ A'; s = t \ (\subseteq \text{state} \cap X) \implies \text{bval} \ b \ s = \text{bval} \ b \ t \)
proof (induction \( b \) arbitrary: \( A' \)
fix $A'$ v
show
\[\begin{align*}
&\models Bc v (\subseteq A, X) = Some A'; s = t (\subseteq state \cap X) \\
&bval (Bc v) s = bval (Bc v) t
\end{align*}\]
by simp

next
fix $A'$ b
show
\[\begin{align*}
&\forall A'. \models b (\subseteq A, X) = Some A' \Rightarrow s = t (\subseteq state \cap X) \\
&bval b s = bval b t;
&\models b (\subseteq A, X) = Some A'; s = t (\subseteq state \cap X) \\
&bval b s = bval b t
\end{align*}\]
by (simp split: option.split-asn)

next
fix $A'\ b_1\ b_2$
show
\[\begin{align*}
&\forall A'. \models b_1 (\subseteq A, X) = Some A' \Rightarrow s = t (\subseteq state \cap X) \\
&bval b_1 s = bval b_1 t;
&\forall A'. \models b_2 (\subseteq A, X) = Some A' \Rightarrow s = t (\subseteq state \cap X) \\
&bval b_2 s = bval b_2 t;
&\models And b_1 b_2 (\subseteq A, X) = Some A'; s = t (\subseteq state \cap X) \\
&bval (And b_1 b_2) s = bval (And b_1 b_2) t
\end{align*}\]
by (simp split: option.split-asn)

next
fix $A'\ a_1\ a_2$
show
\[\begin{align*}
&\models Less a_1 a_2 (\subseteq A, X) = Some A'; s = t (\subseteq state \cap X) \\
&bval (Less a_1 a_2) s = bval (Less a_1 a_2) t
\end{align*}\]
by (subgoal-tac aval a_1 s = aval a_1 t, subgoal-tac aval a_2 s = aval a_2 t, auto intro: avars-aval split: if-split-asn)

qed

lemma ctyping1-merge-in:
\[xs \in A \cup B \implies xs \in A \cup B\]
by (force simp: ctyping1-merge-def)

lemma ctyping1-merge-append-in:
\[[xs \in A; ys \in B] \implies xs \oplus ys \in A \cup B\]
by (force simp: ctyping1-merge-append-def ctyping1-append-def ctyping1-merge-in)

lemma ctyping1-aux-nonempty:
\[\vdash c \neq \emptyset\]
by (induction c, simp-all add: Let-def ctyping1-append-def ctyping1-merge-def ctyping1-merge-append-def, fastforce+)

lemma ctyping1-mono:
\[[B, Y] = \vdash c (\subseteq A, X); (B', Y') = \vdash c (\subseteq A', X'); A' \subseteq A; X \subseteq X\]
lemma ctyping2-fst-empty:
 Some (B, Y) = (U, v) ⊢ c (⊆ {}, X) → (B, Y) = ({}: UNION);

proof (induction (U, v) c {} :: state set X arbitrary: B Y U v
 rule: ctyping2.induct)

fix C X Y U v b c_1 c_2

show

\[ \forall U \cdot p \, B_2 \cdot C \cdot Y. \]
\[ \forall U \cdot p = (insert (Univ? {} X, bvars b) U, \vdash b (⊆ {}, X)) \rightarrow \]
\[ (\{\}, B_2) \vdash p \rightarrow Some (C, Y) = (U', v) \vdash c_1 (⊆ {}, X) \rightarrow \]
\[ (C, Y) = ({}: UNION); \]
\[ \forall U \cdot p \, B_1 \cdot C \cdot Y. \]
\[ \forall U \cdot p = (insert (Univ? {} X, bvars b) U, \vdash b (⊆ {}, X)) \rightarrow \]
\[ (B_1, \{\}) \vdash p \rightarrow Some (C, Y) = (U', v) \vdash c_2 (⊆ {}, X) \rightarrow \]
\[ (C, Y) = ({}: UNION); \]
\[ Some (C, Y) = (U, v) \vdash IF b THEN c_1 ELSE c_2 (⊆ {}, X) \rightarrow \]
\[ (C, Y) = ({}: UNION) \]
by (fastforce simp: btyping2-fst-empty split: option.split-asn)

next

fix B X Z U v b c

show

\[ \forall \Lambda B_2 \cdot C \cdot Y \cdot B_1 \cdot B_2' \cdot B \cdot Z. \]
\[ (\{\}, B_2) \vdash b (⊆ {}, X) \rightarrow \]
\[ (C, Y) \vdash c (⊆ {}, X) \rightarrow \]
\[ (B_1', B_2') \vdash b (⊆ C, Y) \rightarrow \]
\[ \forall (B, W) \in insert (Univ? {} X \cup Univ? C \cdot Y, bvars b) U. \]
\[ B : dom ' W \rightarrow UNION \rightarrow \]
\[ Some (B, Z) = ({}: False) \vdash c (⊆ {}, X) \rightarrow \]
\[ (B, Z) = ({}: UNION); \]
\[ \forall \Lambda B_1 \cdot B_2 \cdot C \cdot Y \cdot B_2' \cdot B \cdot Z. \]
\[ (B_1, B_2) \vdash b (⊆ {}, X) \rightarrow \]
\[ (C, Y) \vdash c (⊆ B_1, X) \rightarrow \]
\[ (\{\}, B_2') \vdash b (⊆ C, Y) \rightarrow \]
\[ \forall (B, W) \in insert (Univ? {} X \cup Univ? C \cdot Y, bvars b) U. \]
\[ B : dom ' W \rightarrow UNION \rightarrow \]
\[ Some (B, Z) = ({}: False) \vdash c (⊆ {}, Y) \rightarrow \]
\[ (B, Z) = ({}: UNION); \]
\[ Some (B, Z) = (U, v) \vdash WHILE b DO c (⊆ {}, X) \rightarrow \]
\[ (B, Z) = ({}: UNION) \]
by (simp split: if-split-asn option.split-asn prod.split-asn, 
  fastforce simp: btyping2-fst-empty ctyping1-def)+

qed (simp-all split: if-split-asn option.split-asn prod.split-asn)

lemma ctyping2-mono-assign [elim!]:
\[ ([U, False] \vdash x := a (⊆ A, X) = Some (C, Z); A' ⊆ A; X ⊆ X'; \]
\[ \forall (B', Y') \in U'. \exists (B, Y) \in U. B' ⊆ B \land Y' ⊆ Y] \rightarrow \]
\[ \exists C' Z'. \ (U', \ False) \vdash x ::= a (\subseteq A', X') = \text{Some} (C', Z') \land \]
\[ C' \subseteq C \land Z \subseteq Z' \]
by \( \text{frule interf-set-mono [where } W = \{ \text{dom } x \} , \text{auto split: if-split-asm} \)

**Lemma ctyping2-mono-seq:**

**Assumes**

1. \( \bigwedge A' B X' Y U' \)
   \[
   (U, False) \vdash c_1 (\subseteq A, X) = \text{Some} (B, Y) \implies A' \subseteq A \implies X \subseteq X' \implies \exists (B', Y') \in U'. \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y \implies \exists B' Y'. \ (U', False) \vdash c_1 (\subseteq A', X') = \text{Some} (B', Y') \land B' \subseteq B \land Y \subseteq Y' \]
2. \( \bigwedge B Y B' C Y' Z U' \)
   \[
   (U, False) \vdash c_1 (\subseteq A, X) = \text{Some } p \implies (B, Y) = p \implies \exists (B', Y') \in U'. \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y \implies \exists C' Z'. \ (U', False) \vdash c_2 (\subseteq A', X') = \text{Some} (C', Z') \land C' \subseteq C \land Z \subseteq Z' \]

**Proof**

- **Obtain** \( B Y \) where \( (U, False) \vdash c_1 (\subseteq A, X) = \text{Some} (B, Y) \land \)
  \( (U, False) \vdash c_2 (\subseteq B, Y) = \text{Some} (C, Z) \)
  using \( C \) by \( \text{auto split: option.split-asm} \)

- **Moreover from this** obtain \( B' Y' \) where
  \( G: (U', False) \vdash c_1 (\subseteq A', X') = \text{Some} (B', Y') \land B' \subseteq B \land Y \subseteq Y' \)
  using \( A \) and \( D \) and \( E \) and \( F \) by fastforce

- **Ultimately obtain** \( C' Z' \) where
  \( (U', False) \vdash c_2 (\subseteq B', Y') = \text{Some} (C', Z') \land C' \subseteq C \land Z \subseteq Z' \)
  using \( B \) and \( F \) by fastforce

- **Thus** \( \exists \theta \text{thesis} \)
  using \( G \) by \( \text{simp} \)

**Lemma ctyping2-mono-if:**

**Assumes**

1. \( \bigwedge W p B_1 B_2 B_1' C_1 X' Y_1 W' . (W, p) = \)
   \[
   (\text{insert} \ (\text{Univ}? A X, \text{bears } b) U, \vdash b (\subseteq A, X)) \implies (B_1, B_2) = p \implies \]
   \( (W, False) \vdash c_1 (\subseteq B_1, X) = \text{Some} (C_1, Y_1) \implies B_1' \subseteq B_1 \implies \]
   \( X \subseteq X' \implies \forall (B', Y') \in W'. \exists (B, Y) \in W. B' \subseteq B \land Y' \subseteq Y \implies \exists C_1' Y_1'. (W', False) \vdash c_1 (\subseteq B_1', X') = \text{Some} (C_1', Y_1') \land C_1' \subseteq C_1 \land Y_1 \subseteq Y_1' \)
2. \( \bigwedge B Y B_2 B_1 B_1' C_2 X' Y_2 W' . (W, p) = \)
   \[
   (\text{insert} \ (\text{Univ}? A X, \text{bears } b) U, \vdash b (\subseteq A, X)) \implies (B_1, B_2) = p \implies \]
   \( (W, False) \vdash c_2 (\subseteq B_2, X) = \text{Some} (C_2, Y_2) \implies B_2' \subseteq B_2 \implies \]
   \( X \subseteq X' \implies \forall (B', Y') \in W'. \exists (B, Y) \in W. B' \subseteq B \land Y' \subseteq Y \implies \]

```
\[ \exists C', Y', (W, \text{False}) \models c_2 (\subseteq B', X') = \text{Some} (C', Y') \quad \land \]
\[ C' \subseteq C_2 \land Y_2 \subseteq Y_2' \quad \text{and} \]
\[ C: (U, \text{False}) \models IF b \quad \text{THEN} \quad c_1 \quad \text{ELSE} \quad c_2 (\subseteq A, X) = \text{Some} (C, Y) \quad \text{and} \]
\[ D: A' \subseteq A \quad \land \]
\[ E: X \subseteq X' \quad \land \]
\[ F: \forall (B', Y') \in U'. \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y \]
\[ \text{shows} \quad \exists C', Y'. (U', \text{False}) \models IF b \quad \text{THEN} \quad c_1 \quad \text{ELSE} \quad c_2 (\subseteq A', X') = \\
\text{Some} (C', Y') \quad \land \quad C' \subseteq C \land Y \subseteq Y' \]
proof –
let \( ?W = \text{insert} \ A X, \text{bvars} b U \)
let \( ?W' = \text{insert} \ A' X', \text{bvars} b U' \)
obtain \( B_1, B_2 \ C_1, C_2, Y_1, Y_2 \) where
\[ G: (C, Y) = (C_1 \cup C_2, Y_1 \cap Y_2) \land (B_1, B_2) = \models b (\subseteq A, X) \land \\
\text{Some} (C_1, Y_1) = (?W, \text{False}) \models c_1 (\subseteq B_1, X) \land \\
\text{Some} (C_2, Y_2) = (?W, \text{False}) \models c_2 (\subseteq B_2, X) \]
using \( C \) by (simp split: option.split-asms prod.split-asms)
moreover obtain \( B'_1, B'_2 \) where \( H: (B'_1, B'_2) = \models b (\subseteq A', X') \)
by (cases \( \models b (\subseteq A', X'), \text{simp} \))
ultimately have \( I: B'_1 \subseteq B_1 \land B'_2 \subseteq B_2 \)
by (metis btypyng2-mono D E)
moreover have \( J: \forall (B', Y') \in ?W'. \exists (B, Y) \in ?W. B' \subseteq B \land Y' \subseteq Y \)
using \( D \) and \( E \) and \( F \) by (auto simp: univ-states-if-def)
ultimately have \( \exists C'_1, Y'_1 \).
\[ (?W', \text{False}) \models c_1 (\subseteq B'_1, X') = \text{Some} (C'_1, Y'_1) \land C'_1 \subseteq C_1 \land Y_1 \subseteq Y_1' \]
using \( A \) and \( E \) and \( G \) by force
moreover have \( \exists C'_2, Y'_2 \).
\[ (?W', \text{False}) \models c_2 (\subseteq B'_2, X') = \text{Some} (C'_2, Y'_2) \land C'_2 \subseteq C_2 \land Y_2 \subseteq Y_2' \]
using \( B \) and \( E \) and \( G \) and \( I \) and \( J \) by force
ultimately show \( \text{thesis} \)
using \( G \) and \( H \) by (auto split: prod.split)
qed

lemma ctypyng2-mono-while:
assumes
\[ A: \forall B_1, B_2, C, Y, B_1', B_2', D_1, E, X', V, U'. (B_1, B_2) = \models b (\subseteq A, X) \Longrightarrow \\
(C, Y) = \exists c (\subseteq B_1, X) (B_1, B_2) \models b (\subseteq C, Y) \Longrightarrow \\
\forall (B, W) \in \text{insert} \ (\text{Univ} \ A X \cup \text{Univ} \ C Y, \text{bvars} b) U. \]
\( B: \text{dom} \ B \text{ \ ~ \ UNIV} \Longrightarrow \\
\{\} \models c (\subseteq B_1, X) = \text{Some} (E, V) \models D_1 \subseteq B_1 \Longrightarrow \\
X \subseteq X' \Longrightarrow \forall (B', Y') \in U'. \exists (B, Y) \in \{\}. B' \subseteq B \land Y' \subseteq Y \Longrightarrow \\
\exists E', V'. (U', \text{False}) \models c (\subseteq D_1, X') = \text{Some} (E', V') \land \\
E' \subseteq E \land V \subseteq V' \quad \text{and} \]
\[ B: \forall B_1, B_2, C, Y, B_1', B_2', D_1, F, Y', W, U'. (B_1, B_2) = \models b (\subseteq A, X) \Longrightarrow \\
(C, Y) = \exists c (\subseteq B_1, X) (B_1, B_2) \models b (\subseteq C, Y) \Longrightarrow \\
\forall (B, W) \in \text{insert} \ (\text{Univ} \ A X \cup \text{Univ} \ C Y, \text{bvars} b) U. \]
\( B: \text{dom} \ B \text{ \ ~ \ UNIV} \Longrightarrow \\
\{\} \models c (\subseteq B_1', Y) = \text{Some} (F, W) \models D_1' \subseteq B_1' \Longrightarrow \\
Y \subseteq Y' \Longrightarrow \forall (B', Y') \in U'. \exists (B, Y) \in \{\}. B' \subseteq B \land Y' \subseteq Y \Longrightarrow \\
\exists F', W'. (U', \text{False}) \models c (\subseteq D_1', Y') = \text{Some} (F', W') \land \\
44
\[ F' \subseteq F \land W \subseteq W' \text{ and } \]

\( C: (U, \text{False}) \models \text{WHILE } b \text{ DO } c (\subseteq A, X) = \text{Some } (B, Z) \text{ and } \)

\( D: A' \subseteq A \text{ and } \)

\( E: X \subseteq X' \text{ and } \)

\( F: \forall (B', Y') \in U'. \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y' \)

\( \text{shows } \exists B' Z'. (U', \text{False}) \models \text{WHILE } b \text{ DO } c (\subseteq A', X') = \text{Some } (B', Z') \land B' \subseteq B \land Z' \subseteq Z' \)

\text{proof –}

\( \text{obtain } B_1, B_1', B_2, B_2' C E F V W Y \text{ where } G: (B_1, B_2) = \models b (\subseteq A, X) \land (C, Y) = \models c (\subseteq B_1, X) \land (B_1', B_2') = \models b (\subseteq C, Y) \land (\forall (B, W) \in \text{insert } (\text{Univ} A X \cup \text{Univ} Y C Y, \text{bvars } b) U. B: \text{dom } ' W \leadsto \text{UNIV} ) \land \)

\( \text{Some } (E, V) = (\{\}, \text{False}) \models c (\subseteq B_1, X) \land \)

\( \text{Some } (F, W) = (\{\}, \text{False}) \models c (\subseteq B_1', Y) \land \)

\( (B, Z) = (B_2 \cup B_2', \text{Univ} B_2 X \cup Y) \)

\( \text{using } C \text{ by (force split: if-split-asm option.split-asm prod.split-asm)} \)

\( \text{moreover obtain } D_1, D_2 \text{ where } H: \models b (\subseteq A', X') = (D_1, D_2) \)

\( \text{by (cases } b (\subseteq A', X'), \text{simp)} \)

\( \text{ultimately have } I: D_1 \subseteq B_1 \land D_2 \subseteq B_2 \)

\( \text{by (smt (verit) btyping2-mono D E)} \)

\( \text{moreover obtain } C', Y' \text{ where } J: (C', Y') = \models c (\subseteq D_1, X') \)

\( \text{by (cases } c (\subseteq D_1, X'), \text{simp)} \)

\( \text{ultimately have } K: C' \subseteq C \land Y' \subseteq Y' \)

\( \text{by (meson ctyping1-mono E G)} \)

\( \text{moreover obtain } D_1', D_2' \text{ where } L: \models b (\subseteq C', Y') = (D_1', D_2') \)

\( \text{by (cases } b (\subseteq C', Y'), \text{simp)} \)

\( \text{ultimately have } M: D_1' \subseteq B_1' \land D_2' \subseteq B_2' \)

\( \text{by (smt (verit) btyping2-mono G)} \)

\( \text{then obtain } F', W' \text{ where } \)

\( (\{\}, \text{False}) \models c (\subseteq D_1', Y') = \text{Some } (F', W') \land F' \subseteq F \land W \subseteq W' \)

\( \text{using } B \text{ and } F \text{ and } G \text{ and } I \text{ by force} \)

\( \text{moreover obtain } E', V' \text{ where } \)

\( (\{\}, \text{False}) \models c (\subseteq D_1, X') = \text{Some } (E', V') \land E' \subseteq E \land V \subseteq V' \)

\( \text{using } A \text{ and } E \text{ and } F \text{ and } G \text{ and } I \text{ by force} \)

\( \text{moreover have } \text{Univ} A' X' \subseteq \text{Univ} A X \)

\( \text{using } D \text{ and } E \text{ by (auto simp: univ-states-if-def)} \)

\( \text{moreover have } \text{Univ} A' Y' \subseteq \text{Univ} A Y \)

\( \text{using } K \text{ by (auto simp: univ-states-if-def)} \)

\( \text{ultimately have } (U', \text{False}) \models \text{WHILE } b \text{ DO } c (\subseteq A', X') = \text{Some } (D_2 \cup D_2', \text{Univ} D_2 X' \cap Y') \)

\( \text{using } F \text{ and } G \text{ and } H \text{ and } J \text{ [symmetric] and } L \text{ by force} \)

\( \text{moreover have } D_2 \cup D_2' \subseteq B \)

\( \text{using } G \text{ and } I \text{ and } M \text{ by auto} \)

\( \text{moreover have } Z \subseteq \text{Univ} D_2 X' \cap Y' \)

\( \text{using } E \text{ and } G \text{ and } I \text{ and } K \text{ by auto} \)

\( \text{ultimately show } ? \text{thesis} \)

\( \text{by simp} \)

\( \text{qed} \)

45
lemma ctying2-mono:
\[\forall (B', Y') \in U'. \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y \implies \exists C' Z'. (U', False) \models c (\subseteq A', X') = \text{Some} (C', Z') \land C' \subseteq C \land Z \subseteq Z'\]

proof (induction \((U, False)\) \(c A X\) arbitrary: \(A' C X' Z U U'\))

rule: ctying2.induct

fix \(A' X X' U U' C Z c_1 c_2\)

show
\[\forall A' B X' Y' U'. (U, False) \models c_1 (\subseteq A, X) = \text{Some} (B, Y) \implies A' \subseteq A \implies X \subseteq X' \implies \forall (B', Y') \in U'. \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y \implies \exists C' Z'. (U', False) \models c_1 (\subseteq A', X') = \text{Some} (B', Y') \land B' \subseteq B \land Y \subseteq Y';\]

\[\land_{p\in B Y A' C X' Z U'}. (U, False) \models c_1 (\subseteq A, X) = \text{Some} p \implies (B, Y) = p \implies (U, False) \models c_2 (\subseteq B, Y) = \text{Some} (C, Z) \implies A' \subseteq B \implies Y \subseteq X' \implies \forall (B', Y') \in U'. \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y \implies \exists C' Z'. (U', False) \models c_2 (\subseteq A', X') = \text{Some} (C', Z') \land C' \subseteq C \land Z \subseteq Z';\]

\((U, False) \models c_1; c_2 (\subseteq A, X) = \text{Some} (C, Z);\)

\(A' \subseteq A; X \subseteq X';\)

\[\forall (B', Y') \in U'. \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y \implies \exists C' Z'. (U', False) \models c_1; c_2 (\subseteq A', X') = \text{Some} (C', Z') \land C' \subseteq C \land Z \subseteq Z';\]

by (rule ctying2-mono-seq)

next

fix \(A' X X' U U' C Z b c_1 c_2\)

show
\[\forall U'' p B_1 B_2 A' C X' Z U'. (U'', p) = (\text{insert} (\text{Univ'?} A X, \text{bvars} b) U, \models b (\subseteq A, X)) \implies (B_1, B_2) = p \implies (U'', False) \models c_1 (\subseteq B_1, X) = \text{Some} (C, Z) \implies A' \subseteq B_1 \implies X \subseteq X' \implies \forall (B', Y') \in U'. \exists (B, Y) \in U''. B' \subseteq B \land Y' \subseteq Y \implies \exists C' Z'. (U', False) \models c_1 (\subseteq A', X') = \text{Some} (C', Z') \land C' \subseteq C \land Z \subseteq Z';\]

\[\land_{U'' p B_1 B_2 A' C X' Z U'}. (U'', p) = (\text{insert} (\text{Univ'?} A X, \text{bvars} b) U, \models b (\subseteq A, X)) \implies (B_1, B_2) = p \implies (U'', False) \models c_2 (\subseteq B_2, X) = \text{Some} (C, Z) \implies A' \subseteq B_2 \implies X \subseteq X' \implies \forall (B', Y') \in U'. \exists (B, Y) \in U''. B' \subseteq B \land Y' \subseteq Y \implies \exists C' Z'. (U', False) \models c_2 (\subseteq A', X') = \text{Some} (C', Z') \land C' \subseteq C \land Z \subseteq Z';\]

\((U, False) \models \text{IF} b \text{ THEN } c_1 \text{ ELSE } c_2 (\subseteq A, X) = \text{Some} (C, Z);\)

\(A' \subseteq A; X \subseteq X';\)

\[\forall (B', Y') \in U'. \exists (B, Y) \in U. B' \subseteq B \land Y' \subseteq Y \implies \exists C' Z'. (U', False) \models \text{IF} b \text{ THEN } c_1 \text{ ELSE } c_2 (\subseteq A', X') = \text{Some} (C', Z') \land C' \subseteq C \land Z \subseteq Z';\]

by (rule ctying2-mono-if)
next

fix A A' X X' U U' B Z b c

show

\[ \forall B_1 \, B_2 \, C \, Y \, B_1' \, B_2' \, A' \, B \, X' \, Z \, U'. \]
\[ (B_1, \, B_2) \models b \, (\subseteq \, A, \, X) \implies \]
\[ (C, \, Y) \models c \, (\subseteq \, B_1, \, X) \implies \]
\[ (B_1', \, B_2') \models b \, (\subseteq \, C, \, Y) \implies \]
\[ \forall (B, \, W) \in \text{insert} \, (\text{Univ}? \, A \, X \cup \text{Univ}? \, C \, Y, \, \text{bvars} \, b) \, U. \]
\[ B : \text{dom} \, \cdot \, W \rightsquigarrow \text{UNIV} \implies \]
\[ \exists \, A', \, B' \, (\subseteq \, A, \, X) = \text{some} \, (B, \, Z) \implies \]
\[ A' \subseteq B_1 \implies X \subseteq X' \implies \]
\[ \forall (B', \, Y') \in U'. \exists (B, \, Y) \in \{\}. \, B' \subseteq B \land Y' \subseteq Y \implies \]
\[ \exists B' \, Z'. (U', \, False) \models c \, (\subseteq \, A', \, X') = \text{some} \, (B', \, Z') \land \]
\[ B' \subseteq B \land Z \subseteq Z'; \]

\[ \wedge B_1 \, B_2 \, C \, Y \, B_1' \, B_2' \, A' \, B \, X' \, Z \, U'. \]
\[ (B_1, \, B_2) \models b \, (\subseteq \, A, \, X) \implies \]
\[ (C, \, Y) \models c \, (\subseteq \, B_1, \, X) \implies \]
\[ (B_1', \, B_2') \models b \, (\subseteq \, C, \, Y) \implies \]
\[ \forall (B, \, W) \in \text{insert} \, (\text{Univ}? \, A \, X \cup \text{Univ}? \, C \, Y, \, \text{bvars} \, b) \, U. \]
\[ B : \text{dom} \, \cdot \, W \rightsquigarrow \text{UNIV} \implies \]
\[ \exists \, A' \subseteq B_1' \implies Y \subseteq X' \implies \]
\[ \forall (B', \, Y') \in U'. \exists (B, \, Y) \in \{\}. \, B' \subseteq B \land Y' \subseteq Y \implies \]
\[ \exists B' \, Z'. (U', \, False) \models c \, (\subseteq \, A', \, X') = \text{some} \, (B', \, Z') \land \]
\[ B' \subseteq B \land Z \subseteq Z'; \]
\[ (U, \, False) \models \text{WHILE} \, b \, \text{DO} \, c \, (\subseteq \, A, \, X) = \text{some} \, (B, \, Z); \]
\[ A' \subseteq A; \, X \subseteq X'; \]
\[ \forall (B', \, Y') \in U'. \exists (B, \, Y) \in U. \, B' \subseteq B \land Y' \subseteq Y \implies \]
\[ \exists B' \, Z'. (U', \, False) \models \text{WHILE} \, b \, \text{DO} \, c \, (\subseteq \, A', \, X') = \]
\[ \text{some} \, (B', \, Z') \land B' \subseteq B \land Z \subseteq Z' \]

by (rule \text{ctyping2-mono-while})

qed \text{fastforce+}

\text{lemma \text{ctyping1-ctyping2-fst-assign} \, [elim!]:}
\text{assumes}
A : (C, \, Z) = \models x := a \, (\subseteq \, A, \, X) \, \text{and}
B : \text{some} \, (C', \, Z') = (U, \, False) \models x := a \, (\subseteq \, A, \, X)
\text{shows} \ C' \subseteq C
\text{proof} -
\{ 
\text{fix} \, s
\text{assume} \, s \in \, A
\text{moreover assume} \, \text{avars} \, a = \{\}
\text{hence} \, \text{aval} \, a \, s = \text{aval} \, a \, (\lambda x. \, \emptyset)
\text{by} \, \text{blast intro; avars-aval}
\text{ultimately have} \, \exists \, s'. (\exists \, t. \, s(x := \text{aval} \, a \, s) = (\lambda x'. \, \text{case case if} \, x' = x \, \text{then} \, \text{some} \, (\text{some} \, (\text{aval} \, a \, (\lambda x. \, \emptyset))) \, \text{else None of None} \, \Rightarrow \, \text{None} \mid \, \text{some} \, v \, \Rightarrow \, \text{some} \, v \, \text{of} \}
\}
None \Rightarrow s' \, x' \mid \text{Some None} \Rightarrow t \, x' \mid \text{Some (Some i) \Rightarrow i}) \land s' \in A

\text{by fastforce}
\}\)

\text{note} \ C = \text{this}
\text{from} \ A \text{ and } B \text{ show } \text{?thesis}
\text{by } (\text{clarsimp simp ctyping1-def ctyping1-seq-def split: if-split-asm, erule-tac C, simp, fastforce})
\text{qed}

\text{lemma} \ ctyping1-ctyping2-fst-seq:
\text{assumes}
A: \forall B \, B' \, Y \, Y'. \ (B, Y) = \vdash c_1 (\subseteq A, X) \Rightarrow 
\text{Some (B', Y')} = (U, \text{False}) \vdash c_1 (\subseteq A, X) \Rightarrow B' \subseteq B \text{ and }
B: \forall B \, Y \, C \, C' \, Z \, Z'. \ (U, \text{False}) \vdash c_1 (\subseteq A, X) \Rightarrow \text{Some p} \Rightarrow 
(B, Y) = p \Rightarrow (C, Z) = \vdash c_2 (\subseteq B, Y) \Rightarrow 
\text{Some (C', Z')} = (U, \text{False}) \vdash c_2 (\subseteq B, Y) \Rightarrow C' \subseteq C \text{ and }
C: (C, Z) = \vdash c_1 \sqcup c_2 (\subseteq A, X) \text{ and }
D: \text{Some (C', Z')} = (U, \text{False}) \vdash c_1; c_2 (\subseteq A, X)
\text{shows} \ C' \subseteq C
\text{proof} –
\text{let} \ \text{？F = foldl} \ (\_); (\lambda x. \text{None})
\text{let} \ \text{?P = } \lambda x \ A \ S. \exists f \ s. \exists t. \ r = (\lambda x. \text{case f x of} 
None \Rightarrow s \ x \mid \text{Some None} \Rightarrow t \ x \mid \text{Some (Some i) \Rightarrow i}) \land 
(\exists y \ s \ f = \text{？F} \ y \ s \ s \in S) \land s \in A
\text{let} \ \text{？F = } \lambda A \ S. \{ r. \text{？P} \ r \ A \ S \}
\{ 
\text{fix} \ s_3 \ B' \ Y'
\text{assume} 
E: \forall B'' \ B \ C \ C' \ Z'. \ B' = B'' \Rightarrow B = B'' \Rightarrow C = \text{？F} B'' (\vdash c_2) \Rightarrow 
\text{Some (C', Z')} = (U, \text{False}) \vdash c_2 (\subseteq B'', Y') \Rightarrow 
C' \subseteq \text{？F} B'' (\vdash c_2) \text{ and }
F: \forall B B''. \ B = \text{？F} A (\vdash c_1) \Rightarrow B'' = B' \Rightarrow B' \subseteq \text{？F} A (\vdash c_1) \text{ and }
G: \text{Some (C', Z')} = (U, \text{False}) \vdash c_2 (\subseteq B', Y') \text{ and }
H: s_3 \in C'
\text{have} \ \text{？P} s_3 \ A (\vdash c_1 \sqcup A \vdash c_2)
\text{proof} –
\text{obtain} \ s_2 \text{ and } t_2 \text{ and } y_2 \text{ where}
I: \ s_3 = (\lambda x. \text{case } \text{？F} y_2 \ x \ of 
None \Rightarrow s_2 \ x \mid \text{Some None} \Rightarrow t_2 \ x \mid \text{Some (Some i) \Rightarrow i}) \land 
s_2 \in B' \land y_2 \in \vdash c_2
\text{using E and G and H by fastforce}
\text{from this obtain} \ s_1 \text{ and } t_1 \text{ and } y_1 \text{ where}
J: \ s_2 = (\lambda x. \text{case } \text{？F} y_1 \ x \ of 
None \Rightarrow s_1 \ x \mid \text{Some None} \Rightarrow t_1 \ x \mid \text{Some (Some i) \Rightarrow i}) \land 
s_1 \in A \land y_1 \in \vdash c_1
\text{using F by fastforce}
\text{let} \ \text{？F = } (\lambda x. \text{case } \text{？F} y_1 \ x \ of 
None \Rightarrow case } \text{？F} y_1 \ x \text{ of Some None \Rightarrow t_1 \ x \mid - \Rightarrow 0} \mid
\text{Some None} \Rightarrow t_2 \ x \mid - \Rightarrow 0
\text{48}
lemma ctyping1-ctyping2-fst-if:
assumes
A: \( \forall U' \ p B_1 B_2 C_1 C_1' Y_1 Y_1' \).
\( (U', p) = (\text{insert} (\text{Univ} ? A X, \text{bvars} b) U, \models b (\subseteq A, X)) \implies (B_1, B_2) = p \implies (C_1, Y_1) = \vdash c_1 (\subseteq B_1, X) \implies \) 
Some \((C_1', Y_1') = (U', \text{False}) \models c_1 (\subseteq B_1, X) \implies C_1' \subseteq C_1 \) and
B: \( \forall U' \ p B_1 B_2 C_2 C_2' Y_2 Y_2' \).
\( (U', p) = (\text{insert} (\text{Univ} ? A X, \text{bvars} b) U, \models b (\subseteq A, X)) \implies (B_1, B_2) = p \implies (C_2, Y_2) = \vdash c_2 (\subseteq B_2, X) \implies \) 
Some \((C_2', Y_2') = (U', \text{False}) \models c_2 (\subseteq B_2, X) \implies C_2' \subseteq C_2 \) and
C: \( (C, Y) = \vdash \text{IF} b \text{ THEN} c_1 \text{ ELSE} c_2 (\subseteq A, X) \) and
D: Some \((C', Y') = (U, \text{False}) \models \text{IF} b \text{ THEN} c_1 \text{ ELSE} c_2 (\subseteq A, X) \)
shows \( C' \subseteq C \)
proof –
let \( \forall f = \text{foldl} \ (\_); (\lambda x. \text{None}) \)
let \( ?F = \lambda A \ S. \exists f s. (\exists t. r = (\lambda x. \text{case} f x \text{ of}) \) 
\( \text{None} \Rightarrow s x | \text{Some} \text{ None} \Rightarrow t x | \text{Some} \text{ (Some i) \Rightarrow i}) \) \wedge \( (\exists y s f = \forall g s \wedge g s \in S) \wedge s \in A \)
let \( ?F = \lambda A \ S. \exists r. \ ?F \ r \ A \ S \)
let \( ?S_1 = \lambda f. \text{if} f = \text{Some True} \lor f = \text{None} \text{ then} \vdash c_1 \text{ else} \) {}
let \( ?S_2 = \lambda f. \text{if} f = \text{Some False} \lor f = \text{None} \text{ then} \vdash c_2 \text{ else} \) {}
\{ 
fix \( s' B_1 B_2 C_1 \)
assume
E: \( \forall U' B_1' C_1' C_1'' U' = \text{insert} (\text{Univ} ? A X, \text{bvars} b) U \implies (B_1', B_2) = (\vdash c_1) \implies C_1'' = C_1 \) and
\( C_1 \subseteq ?F B_1 (\vdash c_1) \)
F: \( \vdash b (\subseteq A, X) = (B_1, B_2) \) and
G: \( s' \in C_1 \)
have \( ?F \ s' A \) (let \( f = \vdash b \text{ in} \ ?S_1 \ f \cup \ ?S_2 \ f \))
proof –
obtain \( s \) and \( t \) and \( y s \) where
H: \( s' = (\lambda x. \text{case} \ ?f \ y s \ x \text{ of}) \) 
\( \text{None} \Rightarrow s x | \text{Some} \text{ None} \Rightarrow t x | \text{Some} \text{ (Some i) \Rightarrow i} \) \wedge
\[ s \in B_1 \land ys \in \vdash c_1 \]

using \( E \) and \( G \) by fastforce

moreover from \( F \) and this have \( s \in A \)

by \((\text{blast dest: btyping2-un-eq})\)

moreover from \( F \) and \( H \) have \( \vdash b \neq \text{Some False} \)

by \((\text{auto dest: btyping1-btyping2 [where } A = A \text{ and } X = X])\)

hence \( ys \in (\text{let } f = \vdash b \text{ in } ?S_1 f \cup ?S_2 f) \)

using \( H \) by \((\text{auto simp: Let-def})\)

hence \( ys \in (\text{let } f = \vdash b \text{ in } ?S_1 f \sqcup ?S_2 f) \)

by \((\text{auto simp: Let-def intro: ctyping1-merge-in})\)

ultimately show \( ?\text{thesis} \)

by \( \text{blast} \)

\( \text{qed} \)

\( \text{note } E = \text{this} \)

\( \{ \)

\( \text{fix } s' B_1 B_2 C_2 \)

\( \text{assume } \)

\[ F: \forall U' B_2' C_2'' . \; U' = \text{insert} (\text{Univ} \; A \; X, \; bvars \; b) \; U \implies B_2' = B_1 \implies C_2' = ?F B_2' (\vdash c_2) \implies C_2'' = C_2 \implies C_2 \subseteq ?F B_2 (\vdash c_2) \text{ and} \]

\( G: \vdash b (\subseteq A, X) = (B_1, B_2) \text{ and} \)

\( H: s' \in C_2 \)

have \( ?P s' A (\text{let } f = \vdash b \text{ in } ?S_1 f \sqcup ?S_2 f) \)

\( \text{proof} - \)

obtain \( s \) and \( t \) and \( ys \) where

\[ I: s' = (\lambda x . \text{case } f \; ys \; x \text{ of} \]

None \implies s x | Some None \implies t x | Some (Some i) \implies i \land

\( s \in B_2 \land ys \in \vdash c_2 \)

using \( F \) and \( H \) by fastforce

moreover from \( G \) and this have \( s \in A \)

by \((\text{blast dest: btyping2-un-eq})\)

moreover from \( G \) and \( I \) have \( \vdash b \neq \text{Some True} \)

by \((\text{auto dest: btyping1-btyping2 [where } A = A \text{ and } X = X])\)

hence \( ys \in (\text{let } f = \vdash b \text{ in } ?S_1 f \cup ?S_2 f) \)

using \( I \) by \((\text{auto simp: Let-def})\)

hence \( ys \in (\text{let } f = \vdash b \text{ in } ?S_1 f \sqcup ?S_2 f) \)

by \((\text{auto simp: Let-def intro: ctyping1-merge-in})\)

ultimately show \( ?\text{thesis} \)

by \( \text{blast} \)

\( \text{qed} \)

\( \text{note } F = \text{this} \)

from \( A \) and \( B \) and \( C \) and \( D \) show \( ?\text{thesis} \)

by \((\text{auto simp: ctyping1-def split: option.split-asm prod.split-asm,}
\text{ erule-tac [2] } F, \text{ erule-tac } E)\)

\( \text{qed} \)

\( \text{lemma ctyping1-ctyping2-fst-while:} \)
assumes
A: (C, Y) = ⊢ WHILE b DO c (⊆ A, X) and
B: Some (C', Y') = (U, False) ⊢ WHILE b DO c (⊆ A, X)
shows C' ⊆ C
proof
let \( \bar{?}F = \text{foldl (:) (λx. None)} \)
let \( \bar{?}P = \lambda x A S. \exists f s. (\exists t. r = (\lambda x. \text{case } f x \text{ of}) \text{ None } ⇒ s x | \text{ Some None } ⇒ t x | \text{ Some (Some } i \Rightarrow i \text{) } \land (\exists ys. f = ?f ys \land ys \in S) \land s \in A) \)
let \( \bar{?}F = \lambda A S. \{ r. ?P r A S \} \)
let \( \bar{?}S_1 = \lambda f. if f = \text{Some False} ∨ f = \text{None then} \{ [] \} \text{ else} \{ \} \)
let \( \bar{?}S_2 = \lambda f. if f = \text{Some True} ∨ f = \text{None then} ⊢ c \text{ else} \{ \} \)
fix s' B_1 B_2 B_1' B_2'
assume
C: \( \vdash b (⊆ A, X) = (B_1, B_2) \) and
D: \( \vdash b (⊆ ω F B_1 (⇒ c), \text{Univ}?? B_1 \{ x. \forall f \in \{ ?f ys \mid ys \in \vdash c \}. f x \neq \text{Some None} \land (f x = \text{None } \implies x \in X) \}) = (B_1', B_2') \)
(is \( \vdash - (⊆ C, ∀ Y) = -) \)
assume s' ∈ C' and Some (C', Y') = (if (∀ s \in \text{Univ}?? A X) \lor Univ?? C Y, ∀ x \in \text{bears} b. All (interf s (dom x))) \land (∀ p \in U. ∀ B W. p = (B, W) \rightarrow (∀ s \in B, ∀ x \in W. All (interf s (dom x)))
then Some (B_2 \cup B_2', \text{Univ}?? B_2 X \cap ?Y)
else None)
hence s' ∈ B_2 \cup B_2'
by (simp split: if-split-asn)

hence ?P s' A (let f = ⊢ b in ?S_1 f \cup ?S_2 f)

proof
assume E: s' ∈ B_2
hence s' ∈ A
using C by (blast dest: btyping2-un-eq)
moreover from C and E have ⊢ b ≠ Some True
by (auto dest: btyping1-btyping2 [where A = A and X = X])
hence [] ∈ (let f = ⊢ b in ?S_1 f \cup ?S_2 f)
by (auto simp: Let-def)
ultimately show ?thesis
by force

next
assume s' ∈ B_2'
then obtain s and t and ys where
E: s' = (λx. case f ys x of
None ⇒ s x | Some None ⇒ t x | Some (Some i ⇒ i) \land s ∈ B_1 \land ys ∈ \vdash c
using D by (blast dest: btyping2-un-eq)
moreover from C and this have s ∈ A
by (blast dest: btyping2-un-eq)
moreover from C and E have ⊢ b ≠ Some False
by (auto dest: btyping1-btyping2 [where A = A and X = X])
hence ys ∈ (let f = ⊢ b in ?S_1 f \cup ?S_2 f)
using E by (auto simp: Let-def)
ultimately show \( ?\text{thesis} \)
by blast

qed

note \( C = \text{this} \)
from \( A \) and \( B \) show \( ?\text{thesis} \)
by (auto intro: C simp: ctyping1-def
split: option.split-asm prod.split-asm)

qed

lemma ctyping1-ctyping2-fst:
\[
[(C, Z) \vdash c (\subseteq A, X); \text{Some (C', Z')} = (U, False) \models c (\subseteq A, X)] \implies C' \subseteq C
\]

proof (induction (U, False) c A X arbitrary: C C' Z Z' U
rule: ctyping2.induct)
fix A X C C' Z Z' U c1 c2

show
\[
\forall C C' Z Z'.
\]
\[
(C, Z) \vdash c_1 (\subseteq A, X) \implies
\text{Some (C', Z')} = (U, False) \models c_1 (\subseteq A, X) \implies C' \subseteq C;
\]
\[
\forall p B Y C C' Z Z'. (U, False) \models c_1 (\subseteq A, X) = \text{Some p} \implies
\]
\[
(B, Y) = p \implies (C, Z) \vdash c_2 (\subseteq B, Y) \implies
\text{Some (C', Z')} = (U, False) \models c_2 (\subseteq B, Y) \implies C' \subseteq C;
\]
\[
(C, Z) \vdash c_1 c_2 (\subseteq A, X);
\text{Some (C', Z')} = (U, False) \models c_1 c_2 (\subseteq A, X) \implies C' \subseteq C
\]
by (rule ctyping1-ctyping2-fst-seq)

next
fix A X C C' Z Z' U b c1 c2

show
\[
\forall U' p B_1 B_2 C C' Z Z'.
\]
\[
(U', p) = (\text{insert (Univ? A X, bvars b) U}, \models b (\subseteq A, X)) \implies
\]
\[
(B_1, B_2) = p \implies (C, Z) \vdash c_1 (\subseteq B_1, X) \implies
\text{Some (C', Z')} = (U', False) \models c_1 (\subseteq B_1, X) \implies C' \subseteq C;
\]
\[
\forall U' p B_1 B_2 C C' Z Z'.
\]
\[
(U', p) = (\text{insert (Univ? A X, bvars b) U}, \models b (\subseteq A, X)) \implies
\]
\[
(B_1, B_2) = p \implies (C, Z) \vdash c_2 (\subseteq B_2, X) \implies
\text{Some (C', Z')} = (U', False) \models c_2 (\subseteq B_2, X) \implies C' \subseteq C;
\]
\[
(C, Z) \vdash IF b \text{ THEN } c_1 \text{ ELSE } c_2 (\subseteq A, X);
\text{Some (C', Z')} = (U, False) \models IF b \text{ THEN } c_1 \text{ ELSE } c_2 (\subseteq A, X) \implies C' \subseteq C
\]
by (rule ctyping1-ctyping2-fst-if)

next
fix A X B B' Z Z' U b c
proof

\[ \begin{align*}
\&\forall B_1, B_2 \ C Y B_1' B_2' B B' Z Z'. \\
& (B_1, B_2) = [ b (\subseteq A, X) ] \Rightarrow \\
& (C, Y) = [ c (\subseteq B_1, X) ] \Rightarrow \\
& (B_1', B_2') = [ b (\subseteq C, Y) ] \Rightarrow \\
& \forall (B, W) \in \text{insert} \ (\text{Univ} \ W \ X \cup \text{Univ} \ Y, bvars \ b) \ U. \\
& \quad B: \text{dom} \ W \rightarrow \text{UNIV} \Rightarrow \\
& \quad (B, Z) = [ c (\subseteq B_1, Y) ] \Rightarrow \\
& \quad \text{Some} \ (B', Z') = (\{\}, \text{False}) \vdash c (\subseteq B_1, X) \Rightarrow \\
& \quad B' \subseteq B; \\
\&\forall B_1, B_2 \ C Y B_1' B_2' B B' Z Z'. \\
& (B_1, B_2) = [ b (\subseteq A, X) ] \Rightarrow \\
& (C, Y) = [ c (\subseteq B_1, X) ] \Rightarrow \\
& (B_1', B_2') = [ b (\subseteq C, Y) ] \Rightarrow \\
& \forall (B, W) \in \text{insert} \ (\text{Univ} \ W \ X \cup \text{Univ} \ Y, bvars \ b) \ U. \\
& \quad B: \text{dom} \ W \rightarrow \text{UNIV} \Rightarrow \\
& \quad (B, Z) = [ c (\subseteq B_1, Y) ] \Rightarrow \\
& \quad \text{Some} \ (B', Z') = (\{\}, \text{False}) \vdash c (\subseteq B_1, Y) \Rightarrow \\
& \quad B' \subseteq B; \\
& \text{by} \ (\text{rule ctyping1-ctyping2-fst-while}) \\
\end{align*} \]

qed \ (\text{simp add: ctyping1-def, auto})

lemma ctyping1-ctyping2-snd-assgn [elim!]:

\[ [(C, Z) = [ t \ x := a (\subseteq A, X) ] \Rightarrow \text{Some} \ (C', Z') = (\text{U}, \text{False}) \vdash t \ x := a (\subseteq A, X) ] \Rightarrow Z \subseteq Z' \]

by \ (\text{auto simp: ctyping1-def ctyping1-seq-def split: if-split-asm})

lemma ctyping1-ctyping2-snd-seq:

assumes

\[ \begin{align*}
& A: \forall B' \ Y' \ (B, Y) = [ c_1 (\subseteq A, X) ] \Rightarrow \\
& \quad \text{Some} \ (B', Y') = (\text{U}, \text{False}) \vdash c_1 (\subseteq A, X) \Rightarrow Y \subseteq Y' \quad \text{and} \\
& B: \forall B \ Y C C' Z Z'. (\text{U}, \text{False}) \vdash c_1 (\subseteq A, X) = \text{Some} \ p \Rightarrow \\
& \quad (B, Y) = p \Rightarrow (C, Z) = [ c_2 (\subseteq B, Y) ] \Rightarrow \\
& \quad \text{Some} \ (C', Z') = (\text{U}, \text{False}) \vdash c_2 (\subseteq B, Y) \Rightarrow Z \subseteq Z' \quad \text{and} \\
& C: (C, Z) = [ c_1; c_2 (\subseteq A, X) ] \quad \text{and} \\
& D: \text{Some} \ (C', Z') = (\text{U}, \text{False}) \vdash c_1; c_2 (\subseteq A, X) \\
\end{align*} \]

shows \( Z \subseteq Z' \)

proof –

let \( ?f = \text{foldl} \ (\; \cdot \;) \ (\lambda x. \text{None}) \)

let \( ?F = \lambda A \ S. \ \{ r. \ \exists f s. (\exists t. \ r = (\lambda x. \ \text{case} \ f \ x \ \text{of} \\
\quad \text{None} \Rightarrow t \ s | \ \text{Some} \ \text{None} \Rightarrow t \ x | \ \text{Some} \ (\text{Some} \ i) \Rightarrow i) \} \land \\
\quad (\exists ys. f = ?f \ ys \land \ ys \in S) \land s \in A \} \)

let \( ?G = \lambda X \ S. \ \{ x. \ \forall f \in \{ ?f \ ys \mid \ ys \in S \}. \\
\quad f x \neq \text{Some} \ \text{None} \land (f x = \text{None} \Rightarrow x \in X) \} \)

\{

53
\}

\]
fix \( x B Y \)

assume \( \bigwedge B' B'' C' Z', B = B' \implies B'' = B' \implies C = ?F B' (\vdash c_2) \implies \)

\( \text{Some} \ (C', Z') = (U, \text{False}) \models c_2 (\subseteq B', Y) \implies \)

\( \text{Univ}? B' (\vdash c_2) \subseteq Z' \) and

\( \text{Some} \ (C', Z') = (U, \text{False}) \models c_2 (\subseteq B, Y) \)

hence \( E: \text{Univ}? A (\vdash c_1) \subseteq Y \)

by simp

assume \( \bigwedge C B'. C = ?F A (\vdash c_1) \implies B' = B \implies \)

\( \text{Univ}? A (\vdash c_1) \subseteq Y \)

hence \( F: \text{Univ}? A (\vdash c_1) \subseteq Y \)

by simp

assume \( G: \forall f. (\exists zs. f = ?f \; zs \land zs \in \vdash c_1 \cup \vdash c_2) \implies \)

\( f x \neq \text{Some None} \land (f x = \text{None} \implies x \in X) \)

\{ 

\begin{align*}
\text{fix} & \quad ys \\
\text{have} & \quad \vdash c_1 \neq \{\} \\
& \quad \text{by (rule ctyping1-aux-nonempty)} \\
\text{then obtain} & \quad xs \text{ where} \quad xs \in \vdash c_1 \\
& \quad \text{by blast} \\
\text{moreover assume} & \quad ys \in \vdash c_2 \\
\text{ultimately have} & \quad xs @ ys \in \vdash c_1 \cup \vdash c_2 \\
& \quad \text{by (rule ctyping1-merge-append-in)} \\
\text{moreover assume} & \quad ?f \; ys \; x = \text{Some None} \\
\text{hence} & \quad ?f \; (xs @ ys) \; x = \text{Some None} \\
& \quad \text{by (simp add: Let-def ctyping1-seq-last split: if-split-asm)} \\
\text{ultimately have} & \quad \text{False} \\
& \quad \text{using} \ G \ \text{by blast} 
\end{align*} 

\}

hence \( H: \forall ys \in \vdash c_2. \ ?f \; ys \; x \neq \text{Some None} \)

by blast

\{ 

\begin{align*}
\text{fix} & \quad xs \; ys \\
\text{assume} & \quad xs \in \vdash c_1 \text{ and} \quad ys \in \vdash c_2 \\
\text{hence} & \quad xs @ ys \in \vdash c_1 \cup \vdash c_2 \\
& \quad \text{by (rule ctyping1-merge-append-in)} \\
\text{moreover assume} & \quad ?f \; xs \; x = \text{Some None} \text{ and} \quad ?f \; ys \; x = \text{None} \\
\text{hence} & \quad ?f \; (xs @ ys) \; x = \text{Some None} \\
& \quad \text{by (auto dest: last-in-set simp: Let-def ctyping1-seq-last split: if-split-asm)} \\
\text{ultimately have} & \quad (\exists ys \in \vdash c_2. \ ?f \; ys \; x = \text{None}) \implies \\
(\forall xs \in \vdash c_1. \ ?f \; xs \; x \neq \text{Some None}) \\
& \quad \text{using} \ G \ \text{by blast} 
\end{align*} 

\}

hence \( I: (\exists ys \in \vdash c_2. \ ?f \; ys \; x = \text{None}) \implies \\
(\forall xs \in \vdash c_1. \ ?f \; xs \; x \neq \text{Some None}) \\
& \quad \text{by blast} 
\}

\{ 

\begin{align*}
\text{fix} & \quad xs \; ys \\
\text{assume} & \quad xs \in \vdash c_1 \text{ and} \quad J: \quad ys \in \vdash c_2 
\end{align*} 

\}
hence $xs \otimes ys \in \vdash c_1 \cup c_2$
  
  by (rule ctyping1-merge-append-in)
moreover assume $\exists f \; xs \; x = None$ and $K$: $\exists f \; ys \; x = None$
  
  hence $\exists f \; (xs \otimes ys) \; x = None$
  
  by (simp add: Let-def ctyping1-seq-last split: if-split-asm)
ultimately have $x \in X$
  
  using $G$ by blast
moreover have $\forall xs \in \vdash c_1. \; \exists f \; xs \; x \neq Some None$
  
  using $I$ and $J$ and $K$ by blast
ultimately have $x \in Z'$
  
  using $E$ and $F$ and $H$ by fastforce

moreover { fix $ys$
  
  assume $ys \in \vdash c_2$ and $\exists f \; ys \; x = None$
  
  hence $\forall xs \in \vdash c_1. \; \exists f \; xs \; x \neq Some None$
  
  using $I$ by blast
moreover assume $\forall xs \in \vdash c_1. \; \exists v. \; \exists f \; xs \; x = Some v$
  
  ultimately have $x \in Z'$
  
  using $E$ and $F$ and $H$ by fastforce

moreover {
  
  assume $\forall ys \in \vdash c_2$. $\exists v. \; \exists f \; ys \; x = Some v$
  
  hence $x \in Z'$
  
  using $E$ and $H$ by fastforce
}
ultimately have $x \in Z'$
  
  by (cases $\exists ys \in \vdash c_2$. $\exists f \; ys \; x = None$, cases $\exists xs \in \vdash c_1$. $\exists f \; xs \; x = None$, auto)
moreover assume $x \notin Z'$
  
  ultimately have $False$
  
  by contradiction
}

note $E = this$
from $A$ and $B$ and $C$ and $D$ show $?thesis$
  
  by (auto dest: ctyping2-fst-empty ctyping2-fst-empty [OF sym]
  
  simp: ctyping1-def split: option.split-asm, erule-tac $E$)

qed

lemma ctyping1-ctyping2-snd-if:

assumes
  
  $A$: $\bigwedge (U', p) = (insert (Univ? A X, bears b) U, \models b (\subseteq A, X)) \Rightarrow$

  $(B_1, B_2) = p \Rightarrow (C_1, Y_1) = \vdash c_1 (\subseteq B_1, X) \Rightarrow$

  $\exists (C_1', Y_1') = (U', False) \models c_1 (\subseteq B_1, X) \Rightarrow Y_1 \subseteq Y_1'$ and

  $B$: $\bigwedge (U', p) = (insert (Univ? A X, bears b) U, \models b (\subseteq A, X)) \Rightarrow$

  $(B_1, B_2) = p \Rightarrow (C_2, Y_2) = \vdash c_2 (\subseteq B_2, X) \Rightarrow$

  $\exists (C_2', Y_2') = (U', False) \models c_2 (\subseteq B_2, X) \Rightarrow Y_2 \subseteq Y_2'$ and
proof –

let \( \mathcal{F} = \text{foldl} \ (\_ \_ \_ \_ \_ \text{None}) \ (\lambda x. \text{None}) \)

let \( \mathcal{G} = \lambda A \ S. \{ x. \forall f \in \{ ?fys \mid ys \in S \}. x \neq \text{Some} \text{None} \land (f x = \text{None} \implies x \in X) \}

let \( \mathcal{S}_1 = \lambda f. \text{if} f = \text{Some} \text{True} \lor f = \text{None} \text{then} \vdash c_1 \text{ else} \{ \}

let \( \mathcal{S}_2 = \lambda f. \text{if} f = \text{Some} \text{False} \lor f = \text{None} \text{then} \vdash c_2 \text{ else} \{ \}

let \( \mathcal{P} = \lambda x. \forall f. (\exists ys. f = ?fys \land ys \in (\text{let } f = \vdash b \text{ in } ?S_1 f \sqcup ?S_2 f) \implies f x \neq \text{Some} \text{None} \land (f x = \text{None} \implies x \in X) \)

let \( \mathcal{U} = \text{insert} \ (\text{Univ} \ A X, \text{bears} b) \ U \)

fix \( B_1, B_2, Y_1', Y_2', \text{and} \ C_1' :: \text{state set and} \ C_2' :: \text{state set} \)

assume \( \bigwedge U' B_1', C_1'', U'' = \vdash U \implies B_1' = B_1 \implies C_1 = \vdash F B_1' (\vdash c_1) \implies C_1'' = C_1' \implies \text{Univ} \ ? B_1 (\vdash G X (\vdash c_1)) \subseteq Y_1' \)

hence \( E: \text{Univ} \ ? B_1 (\vdash G X (\vdash c_1)) \subseteq Y_1' \)

by \( \text{simpl} \)

moreover assume \( \bigwedge U' B_1', C_2' C_2'', U'' = \vdash U \implies B_1' = B_1 \implies C_2 = \vdash F B_2 (\vdash c_2) \implies C_2'' = C_2' \implies \text{Univ} \ ? B_2 (?G X (\vdash c_2)) \subseteq Y_2' \)

hence \( F: \text{Univ} \ ? B_2 (\vdash G X (\vdash c_2)) \subseteq Y_2' \)

by \( \text{simpl} \)

moreover assume \( G: \vdash b (\subseteq A, X) = (B_1, B_2) \)

moreover \{ \)

fix \( x \)

assume \( ?P x \)

have \( x \in Y_1' \)

proof \( (\text{cases} \vdash b = \text{Some} \text{False}) \)

\( \text{case True} \)

with \( E \text{ and} \ G \text{ show} \ ? \text{thesis} \)

by \( (\text{drule-tac btyping1-btyping2 [where } A = A \text{ and } X = X], \text{auto}) \)

next \( \)

\( \text{case False} \)

\{ \)

fix \( xs \)

assume \( xs \in \vdash c_1 \)

with \( \text{False} \) have \( xs \in (\text{let } f = \vdash b \text{ in } ?S_1 f \sqcup ?S_2 f) \)

by \( (\text{auto intro: ctyping1-merge-in simpl: Let-def}) \)

hence \( ?f xs x \neq \text{Some} \text{None} \land (?f xs x = \text{None} \implies x \in X) \)

using \( (?P x) \text{ by auto} \)

\} \)

hence \( x \in \text{Univ} \ ? B_1 (\vdash G X (\vdash c_1)) \)

by \( \text{auto} \)

thus \( ? \text{thesis} \)

using \( E \ . . \)

qed
moreover { 
  fix x
  assume ?P x
  have x ∈ Y' 
  proof (cases ⊢ b = Some True) 
    case True
    with F and G show ?thesis
      by (drule-tac btyping1-btyping2 [where A = A and X = X], auto)
  next
  case False
  { 
    fix ys
    assume ys ∈ ⊢ c_2
    with False have ys ∈ (let f = ⊢ b in ?S_1 f ∪ ?S_2 f)
      by (auto intro: ctyping1-merge-in simp: Let-def)
    hence ?f ys ≠ Some None ∧ (?f ys x = None → x ∈ X) 
      using ‹?P x› by auto
  }
  hence x ∈ Unif?? B_2 (?G X (⊢ c_2))
    by auto
  thus ?thesis
    using F ..
  qed 
}
ultimately have (A = {} → UNIV ⊆ Y'_1 ∧ UNIV ⊆ Y'_2) ∧ 
  (A ≠ {} → {x. ?P x} ⊆ Y'_1 ∧ {x. ?P x} ⊆ Y'_2)
  by (auto simp: btyping2-fst-empty)
}

note E = this
from A and B and C and D show ?thesis
  by (clarsimp simp: ctyping1-def split: option.split-asm prod.split-asm, 
erule-tac E)

lemma ctyping1-ctyping2-snd-while:
  assumes
    A: (C, Y) = ⊢ WHILE b DO c (⊆ A, X) and 
    B: Some (C', Y') = (U, False) ⪯ WHILE b DO c (⊆ A, X)
  shows Y ⊆ Y'
proof -
  let ?f = foldl (;;) (λx. None)
  let ?F = λA S. {r. ∀f s. (∃ t. r = (λx. case f x of 
    None ⇒ s x | Some None ⇒ t x | Some (Some i) ⇒ i)) ∧ 
    (∃ ys. f = ?f ys ∧ ys ∈ S) ∧ s ∈ A}
  let ?S_1 = λf. if f = Some False ∨ f = None then {} else {}
  let ?S_2 = λf. if f = Some True ∨ f = None then c else {}
  let ?P = λx. ∀ f. (∃ ys. f = ?f ys ∧ ys ∈ (let f = ⊢ b in ?S_1 f ∪ ?S_2 f)) → 
    f x ≠ Some None ∧ (f x = None → x ∈ X)
let \( Y = \lambda A. \text{Univ}\{\{x. \forall f \in \{\forall y s \mid y s \in \vdash c\}. f x \neq \text{Some None} \land (f x = \text{None} \rightarrow x \in X)\}\} \)

\[
\begin{align*}
\text{fix } B_1 B_2 B_1' B_2' \\
\text{assume } C: \vdash b (\subseteq A, X) = (B_1, B_2) \\
\text{assume } \text{Some } (C' \setminus Y') = (\text{if } \forall s \in \text{Univ } A X \cup \text{Univ } (\forall Y_1 (\vdash c)) (\forall Y B_1) \forall x \in \text{bears } b. \text{All } (\text{interf } s (\text{dom } x)) \land (\forall p \in U. \forall B \ W. p = (B, W) \rightarrow (\forall s \in B. \forall x \in W. \text{All } (\text{interf } s (\text{dom } x)))) & \\
\text{then Some } (B_2 \cup B_2', \text{Univ}\{\text{Univ } B_2 X \cap Y B_1\}) & \\
\text{else None}) \\
\text{hence } D: Y' = \text{Univ}\{\text{Univ } B_2 X \cap Y B_1\} & \\
\text{by } (\text{simp split: if-split-asm}) \end{align*}
\]

\[
\begin{align*}
\text{fix } x \\
\text{assume } A = \{\} & \\
\text{hence } x \in Y' & \\
\text{using } C \text{ and } D \text{ by } (\text{simp add: btyping2-fst-empty}) \end{align*}
\]

moreover \{
\begin{align*}
\text{fix } x \\
\text{assume } ?P x \\
\{ \\
\text{assume } \vdash b \neq \text{Some True} & \\
\text{hence } [] \in (\text{let } f = \vdash b \text{ in } \forall S_1 f \cup \forall S_2 f) & \\
\text{by } (\text{auto simp: Let-def}) & \\
\text{hence } x \in X & \\
\text{using } (?P x) \text{ by auto} & \\
\}
\text{hence } E: \vdash b \neq \text{Some True} \rightarrow x \in \text{Univ}\{\text{Univ } B_2 X\} & \\
\text{by } \text{auto} \end{align*}
\]

\[
\begin{align*}
\text{fix } y s \\
\text{assume } \vdash b \neq \text{Some False and } y s \in \vdash c & \\
\text{hence } y s \in (\text{let } f = \vdash b \text{ in } \forall S_1 f \cup \forall S_2 f) & \\
\text{by } (\text{auto simp: Let-def}) & \\
\text{hence } \forall y s x \neq \text{Some None } \land (\forall y s x = \text{None} \rightarrow x \in X) & \\
\text{using } (?P x) \text{ by auto} \end{align*}
\]

\[
\begin{align*}
\text{hence } F: \vdash b \neq \text{Some False } \rightarrow x \in ?Y B_1 & \\
\text{by } \text{auto} & \\
\text{have } x \in Y' & \\
\text{proof } (\text{cases } \vdash b) & \\
\text{case None} & \\
\text{thus } ?\text{thesis} & \\
\text{using } D \text{ and } E \text{ and } F \text{ by simp} & \\
\text{next} & \\
\text{case } (\text{Some } v) & \\
\text{show } ?\text{thesis} & \\
\text{proof } (\text{cases } v) & \\
\end{align*}
\]
case True
with C and D and F and Some show ?thesis
  by (drule-tac btyping1-btyping2 [where A = A and X = X], simp)
next
case False
with C and D and E and Some show ?thesis
  by (drule-tac btyping1-btyping2 [where A = A and X = X], simp)
qed
qed
}
ultimately have \( (A = \{\} \rightarrow UNIV \subseteq Y') \land (A \neq \{\} \rightarrow \{x. \ ?P x\} \subseteq Y') \)
by auto
}

note C = this
from A and B show ?thesis
  by (auto intro: C simp: ctyping1-def
   split: option.split-asm prod.split-asm)

qed

lemma ctyping1-ctyping2-snd:
\([((C, Z) = \vdash c (\subseteq A, X); \text{Some } (C', Z') = (U, \text{False}) \models c (\subseteq A, X)] \Rightarrow Z \subseteq Z')\)

proof (induction (U, False) c A X arbitrary: C C' Z Z' U
rule: ctyping2.induct)

fix A X C' Z Z' U c_1 c_2

show \( \forall A B Y Y'. \)
\( (B, Y) = \vdash c_1 (\subseteq A, X) \Rightarrow \)
\text{Some } (B', Y') = (U, \text{False}) \models c_1 (\subseteq A, X) \Rightarrow Y \subseteq Y'; \)
\( \land_p B Y C' Z Z'. (U, \text{False}) \models c_1 (\subseteq A, X) = \text{Some } p \Rightarrow \)
\( (B, Y) = p \Rightarrow (C, Z) = \vdash c_2 (\subseteq B, Y) \Rightarrow \)
\text{Some } (C', Z') = (U, \text{False}) \models c_2 (\subseteq B, Y) \Rightarrow Z \subseteq Z'; \)
\( (C, Z) = \vdash c_1; \ c_2 (\subseteq A, X); \)
\text{Some } (C', Z') = (U, \text{False}) \models c_1; \ c_2 (\subseteq A, X) \Rightarrow \)
\( Z \subseteq Z'; \)
by (rule ctyping1-ctyping2-snd-seq)
next

fix A X C' Z Z' U b c_1 c_2

show \( \forall U' p B_1 B_2 C' Z Z'. \)
\( (U', p) = (\text{insert } (\text{Univ? } A X, \text{bvars } b) \ U, \models b (\subseteq A, X)) \Rightarrow \)
\( (B_1, B_2) = p \Rightarrow (C, Z) = \vdash c_1 (\subseteq B_1, X) \Rightarrow \)
\text{Some } (C', Z') = (U', \text{False}) \models c_1 (\subseteq B_1, X) \Rightarrow Z \subseteq Z'; \)
\( \land_{U'} p B_1 B_2 C' Z Z'. \)
\( (U', p) = (\text{insert } (\text{Univ? } A X, \text{bvars } b) \ U, \models b (\subseteq A, X)) \Rightarrow \)
\( (B_1, B_2) = p \Rightarrow (C, Z) = \vdash c_2 (\subseteq B_2, X) \Rightarrow \)
Some \((C', Z') = (U', False) \models c_2 (\subseteq B_2, X) \Rightarrow Z \subseteq Z'\):

\((C, Z) = \vdash \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 (\subseteq A, X);\)

Some \((C', Z') = (U', False) \models \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 (\subseteq A, X)] \Rightarrow Z \subseteq Z'\)

by (rule ctyping1-ctyping2-snd-if)

next

fix \(A X B B' Z Z' U b c\)

show

\([\forall B_1 B_2 C Y B_1' B_2' B B' Z Z'.\]

\((B_1, B_2) = \models b (\subseteq A, X) \Rightarrow (C, Y) = \vdash c (\subseteq B_1, X) \Rightarrow (B_1', B_2') = \models b (\subseteq C, Y) \Rightarrow \forall (B, W) \in \text{insert } (\text{Univ? } A X \cup \text{Univ? } C Y, \text{bvars } b) U.\]

\(B: \text{dom } W \rightarrow \text{UNIV } \Rightarrow (B, Z) = \vdash c (\subseteq B_1, X) \Rightarrow \text{Some } (B', Z') = (\{\}, False) \models c (\subseteq B_1, X) \Rightarrow Z \subseteq Z'.\)

by (rule ctyping1-ctyping2-snd-while)

\text{qed (simp add: ctyping1-def, auto)}

\text{lemma ctyping1-ctyping2:}

\([\vdash c (\subseteq A, X) = (C, Z); (U, False) \models c (\subseteq A, X) = \text{Some } (C', Z')] \Rightarrow C' \subseteq C \land Z \subseteq Z'\)

by (rule conjI, ((rule ctyping1-ctyping2-fst [OF sym sym] | rule ctyping1-ctyping2-snd [OF sym sym]), assumption+)+)

\text{lemma btyping2-aux-approx-1 [elim]:}

\text{assumes}

A: \models b_1 (\subseteq A, X) = \text{Some } B_1 \text{ and}

B: \models b_2 (\subseteq A, X) = \text{Some } B_2 \text{ and}

C: bval b_1 s \text{ and}

D: bval b_2 s \text{ and}

E: r \in A \text{ and}

F: s = r (\subseteq \text{state } \cap X)
shows $\exists r' \in B_1 \cap B_2. \, r = r'$ (⊆ state ∩ X)

proof –
from A and C and E and F have $r \in B_1$
by (frule-tac btyping2-aux-subset, drule-tac btyping2-aux-eq, auto)
moreover from B and D and E and F have $r \in B_2$
by (frule-tac btyping2-aux-subset, drule-tac btyping2-aux-eq, auto)
ultimately show ?thesis
by blast
qed

lemma btyping2-aux-approx-2 [elim]:
assumes
A: avars a₁ ⊆ state and
B: avars a₂ ⊆ state and
C: avars a₁ ⊆ X and
D: avars a₂ ⊆ X and
E: aval a₁ s < aval a₂ s and
F: r ∈ A and
G: s = r (⊆ state ∩ X)
shows $\exists r'. \, r' \in A \land aval a₁ r' < aval a₂ r' \land r = r'$ (⊆ state ∩ X)
proof –
have aval a₁ s = aval a₁ r ∧ aval a₂ s = aval a₂ r
using A and B and C and D and G by (blast intro: avars-aval)
thus ?thesis
using E and F by auto
qed

lemma btyping2-aux-approx-3 [elim]:
assumes
A: avars a₁ ⊆ state and
B: avars a₂ ⊆ state and
C: avars a₁ ⊆ X and
D: avars a₂ ⊆ X and
E: ¬ aval a₁ s < aval a₂ s and
F: r ∈ A and
G: s = r (⊆ state ∩ X)
shows $\exists r' \in A - \{s \in A. \, aval a₁ s < aval a₂ s\}. \, r = r'$ (⊆ state ∩ X)
proof –
have aval a₁ s = aval a₁ r ∧ aval a₂ s = aval a₂ r
using A and B and C and D and G by (blast intro: avars-aval)
thus ?thesis
using E and F by auto
qed

lemma btyping2-aux-approx:
[|= b (⊆ A, X) = Some A'; s ∈ Univ A (⊆ state ∩ X)] \implies s ∈ Univ (if bval b s then A' else A) (⊆ state ∩ X)
by (induction b arbitrary: A', auto dest: btyping2-aux-subset
split: if-split-asm option.split-asm)
lemma ctyping2-approx:
\[ \vdash \text{btyping2-approx} \]
\[ (B_1, B_2); \ s \in \text{Univ} (\subseteq \text{state} \cap X) \]
\[ s \in \text{Univ} (\text{if bval} \ b \ s \ \text{then} \ B_1 \ \text{else} \ B_2) (\subseteq \text{state} \cap X) \]
by (drule sym, simp add: btyping2-def split: option.split-asn, 
drue btyping2-aux-approx, auto)

lemma ctyping2-approx-assgn [elim!]:
\[ \forall t'. \ \text{aval} \ a \ s = t': x \rightarrow (\forall s'. \ t' = s(x := \text{aval} \ a \ s) \rightarrow s \notin A) \]
\[ (\exists y \in \text{state} \cap X. \ y \neq x \land t \neq t'y); \]
\[ v \vdash a (\subseteq X); t \in A; s = t (\subseteq \text{state} \cap X) \Rightarrow \text{False} \]
by (drule spec [of - t(x := \text{aval} \ a \ t)], cases a, 
(fastforce simp del: aval.simps(3) intro: avars-aval+)

lemma ctyping2-approx-if-1:
\[ \text{bval} \ b \ s; \vdash b (\subseteq A, X) = (B_1, B_2); r \in A; s = r (\subseteq \text{state} \cap X); \]
\[ (\text{insert} \ (\text{Univ} A X, \text{bvars} b) U, v) \vdash c_1 (\subseteq B_1, X) = \text{Some} (C_1, Y_1); \]
\[ \forall A B X Y U v. \ (U, v) \vdash c_1 (\subseteq A, X) = \text{Some} (B, Y) \Rightarrow \]
\[ \exists r \in A. \ s = r (\subseteq \text{state} \cap X) \Rightarrow \exists r' \in B. \ t = r' (\subseteq \text{state} \cap Y) \Rightarrow \]
\[ \exists r' \in C_1 \cup C_2. \ t = r' (\subseteq \text{state} \cap (Y_1 \cap Y_2)) \]
by (drule btyping2-approx, blast, fastforce)

lemma ctyping2-approx-if-2:
\[ \neg \text{bval} \ b \ s; \vdash b (\subseteq A, X) = (B_1, B_2); r \in A; s = r (\subseteq \text{state} \cap X); \]
\[ (\text{insert} \ (\text{Univ} A X, \text{bvars} b) U, v) \vdash c_2 (\subseteq B_2, X) = \text{Some} (C_2, Y_2); \]
\[ \forall A B X Y U v. \ (U, v) \vdash c_2 (\subseteq A, X) = \text{Some} (B, Y) \Rightarrow \]
\[ \exists r \in A. \ s = r (\subseteq \text{state} \cap X) \Rightarrow \exists r' \in B. \ t = r' (\subseteq \text{state} \cap Y) \Rightarrow \]
\[ \exists r' \in C_1 \cup C_2. \ t = r' (\subseteq \text{state} \cap (Y_1 \cap Y_2)) \]
by (drule btyping2-approx, blast, fastforce)

lemma ctyping2-approx-while-1 [elim]:
\[ \neg \text{bval} \ b \ s; r \in A; s = r (\subseteq \text{state} \cap X); \vdash b (\subseteq A, X) = (B, \{\}) \Rightarrow \]
\[ \exists t \in C. \ s = t (\subseteq \text{state} \cap Y) \]
by (drule btyping2-approx, blast, simp)

lemma ctyping2-approx-while-2 [elim]:
\[ \forall t \in B_2 \cup B_2'. \ \exists x \in \text{state} \cap (X \cap Y). \ r x \neq t x; \neg \text{bval} \ b \ s; \]
\[ r \in A; s = r (\subseteq \text{state} \cap X); \vdash b (\subseteq A, X) = (B_1, B_2) \Rightarrow \text{False} \]
by (drule btyping2-approx, blast, auto)

lemma ctyping2-approx-while-aux:

assumes
\[ A: \vdash b (\subseteq A, X) = (B_1, B_2) \text{ and} \]
\[ B: r c (\subseteq B_1, X) = (C, Y) \text{ and} \]
\[ C: \vdash b (\subseteq C, Y) = (B_1', B_2') \text{ and} \]
\[ D: (\{\}, \text{False}) \vdash c (\subseteq B_1, X) = \text{Some} (D, Z) \text{ and} \]
\[ E: (\{\}, \text{False}) \vdash c (\subseteq B_1', Y) = \text{Some} (D', Z') \text{ and} \]
\[ F: r_1 \in A \text{ and} \]

62
\[ G : s_1 = r_1 \ (\subseteq \text{state} \cap X) \text{ and} \]
\[ H : \text{hval} \ b \ s_1 \text{ and} \]
\[ I : \forall C \ B \ Y \ W \ U. \ (\text{case} \ b \ (\subseteq C, Y) \ (B_1', B_2') \Rightarrow \]
\[ \text{case} \ b \ (\subseteq B_1', Y) \ (C', Y') \Rightarrow \]
\[ \text{case} \ b \ (\subseteq C', Y') \ (B_1'', B_2'') \Rightarrow \]
\[ \text{if} \ (\forall s \in \text{Univ} \ C Y \cup \text{Univ} \ C' Y' : \forall x \in \text{bears} \ b. \ \text{All} \ (\text{interf} \ s \ (\text{dom} \ x))) \land \]
\[ (\forall p \in U. \ \text{case} \ p \ (B, W) \Rightarrow \forall s \in B. \ \forall x \in W. \ \text{All} \ (\text{interf} \ s \ (\text{dom} \ x))) \]
\[ \text{then case} \ (\emptyset, \text{False}) \ (= c \ (\subseteq B_1', Y)) \text{ of} \]
\[ \text{None} \Rightarrow \text{None} \ | \ \text{Some} \ - \Rightarrow \text{case} \ (\emptyset, \text{False}) \ (= c \ (\subseteq B_1'', Y')) \text{ of} \]
\[ \text{None} \Rightarrow \text{None} \ | \ \text{Some} \ - \Rightarrow \text{Some} \ (B_2' \cup B_2'', \ \text{Univ}?? B_2' Y' \cap Y') \]
\[ \text{else None} = \text{Some} \ (B, W) \Rightarrow \]
\[ \exists r \in C. \ s_2 = r \ (\subseteq \text{state} \cap Y) \Rightarrow \exists r \in B. \ s_3 = r \ (\subseteq \text{state} \cap W) \]
\[ (\text{is} \ \forall C B Y W U. \ ?P C B Y W U \Rightarrow \ - \ - \ - \ | \text{and} \]
\[ J : \forall A B X Y U v. \ (U, v) \Rightarrow c \ (\subseteq A, X) \Rightarrow \text{Some} \ (B, Y) \Rightarrow \]
\[ \exists r \in A. \ s_1 = r \ (\subseteq \text{state} \cap X) \Rightarrow \exists r \in B. \ s_2 = r \ (\subseteq \text{state} \cap Y) \text{ and} \]
\[ K : \forall s \in \text{Univ} \ A X \cup \text{Univ} \ C Y. \ \forall x \in \text{bears} \ b. \ \text{All} \ (\text{interf} \ s \ (\text{dom} \ x)) \text{ and} \]
\[ L : \forall p \in U. \ \forall B W. \ p \Rightarrow (B, W) \Rightarrow \]
\[ (\forall s \in B. \ \forall x \in W. \ \text{All} \ (\text{interf} \ s \ (\text{dom} \ x))) \]
\[ \text{shows} \ \exists r \in B_2 \cup B_2'. \ s_3 = r \ (\subseteq \text{state} \cap \text{Univ}?? B_2 X \cap Y) \]
\[ \text{proof} --\]
\[ \text{obtain} \ C' Y' \text{ where} \ M : \ (C', Y') = \text{val} \ c \ (\subseteq B_1', Y) \]
\[ \text{by (cases} \ b \ (\subseteq B_1', Y), \text{simp)} \]
\[ \text{obtain} \ B_1'' B_2'' \text{ where} \ N : \ (B_1'', B_2'') = \text{val} \ b \ (\subseteq C', Y') \]
\[ \text{by (cases} \ b \ (\subseteq C', Y'), \text{simp)} \]
\[ \text{let} \ ?B = B_2' \cup B_2'' \]
\[ \text{let} \ ?W = \text{Univ}?? B_2' Y' \cap Y' \]
\[ \text{have} \ (C, Y) = \text{val} \ c \ (\subseteq C, Y) \]
\[ \text{using ctyping1-idem and B by auto} \]
\[ \text{moreover have} \ B_1' \subseteq C \]
\[ \text{using C by (blast dest: btyping2-un-eq)} \]
\[ \text{ultimately have} \ O : \ C' \subseteq C \land Y \subseteq Y' \]
\[ \text{by (rule ctyping1-mono [OF - M], simp)} \]
\[ \text{hence} \ \text{Univ}?? C' Y' \subseteq \text{Univ}?? C Y \]
\[ \text{by (auto simp: univ-states-if-def)} \]
\[ \text{moreover from I have} \ ?P C ?B Y ?W U \Rightarrow \]
\[ \exists r \in C. \ s_2 = r \ (\subseteq \text{state} \cap Y) \Rightarrow \exists r \in ?B. \ s_3 = r \ (\subseteq \text{state} \cap ?W) . \]
\[ \text{ultimately have} \ (\text{case} \ (\emptyset, \text{False}) \ (= c \ (\subseteq B_1'', Y')) \text{ of} \]
\[ \text{None} \Rightarrow \text{None} \ | \ \text{Some} \ - \Rightarrow \text{case} \ (\emptyset, \text{False}) \ (= c \ (\subseteq B_1'', Y')) \text{ of} \]
\[ \text{None} \Rightarrow \text{None} \ | \ \text{Some} \ - \Rightarrow \text{Some} \ (\text{Univ}?? B_2' Y' \cap Y') \]
\[ \text{using C and E and K and L and M and N} \]
\[ \text{by (fastforce split: if-split-asm prod.split-asm)} \]
\[ \text{moreover have} \ P : \ B_1'' \subseteq B_1' \land B_2'' \subseteq B_2' \]
\[ \text{by (metis btyping2-mono C N O)} \]
\[ \text{hence} \ \exists D'' Z''. \ (\emptyset, \text{False}) \ (= c \ (\subseteq B_1'', Y')) = \]
\[ \text{Some} \ (D''', Z''') \land \text{D'' } \subseteq \text{D' } \land \text{Z'' } \subseteq \text{Z''' } \]
\[ \text{using E and O by (auto intro: ctyping2-mono)} \]
\[ \text{ultimately have} \]
\[ \exists r \in C. \ s_2 = r \ (\subseteq \text{state} \cap Y) \Rightarrow \exists r \in \ ?B. \ s_3 = r \ (\subseteq \text{state} \cap ?W) \]
\[ \text{by fastforce} \]
moreover from $A$ and $D$ and $F$ and $G$ and $H$ and $J$ obtain $r_2$ where $r_2 \in D$ and $s_2 = r_2 \subseteq \text{state } \cap Z$

by (drule tac btyping2-approx, blast, force)

moreover have $D \subseteq C \land Y \subseteq Z$

using $B$ and $D$ by (rule ctyping1-ctyping2)

ultimately obtain $r_3$ where $Q$: $r_3 \in ?B$ and $R$: $s_3 = r_3 \subseteq \text{state } \cap ?W$ by blast

show $\text{?thesis}$

proof (rule bexI [of - $r_3$])

show $s_3 = r_3 \subseteq \text{state } \cap \text{Univ}? B_2 X \cap Y$

using $O$ and $R$ by auto

next

show $r_3 \in B_2 \cup B_2'$

using $P$ and $Q$ by blast

qed

qed

lemmas ctyping2-approx-while-3 =

ctyping2-approx-while-aux [where $B_2 = \{\}$, simplified]

lemma ctyping2-approx-while-4:

$[|= b \subseteq A, X) = (B_1, B_2);$

$\vdash c \subseteq B_1, X) = (C, Y);$

$\vdash b \subseteq C, Y) = (B_1', B_2');$

($\{\}$, False) $|= c \subseteq B_1, X) = \text{Some} (D, Z);$

($\{\}$, False) $|= c \subseteq B_2', Y) = \text{Some} (D', Z');$

$r_1 \in A; s_1 = r_1 \subseteq \text{state } \cap X); \text{val} b s_1;$

$\forall C B Y W U. (\text{case } \vdash b \subseteq C, Y) \text{ of } (B_1', B_2') \Rightarrow$

case $\vdash c \subseteq B_1', Y) \text{ of } (C', Y') \Rightarrow$

case $\vdash b \subseteq C', Y) \text{ of } (B_1'', B_2'') \Rightarrow$

if ($\forall s \in \text{Univ}? C Y \cup \text{Univ}? C' Y'. \forall x \in \text{bvars b}. \text{All (interf s (dom x))} \land$

$(\forall p \in U. \text{case p of } (B, W) \Rightarrow \forall s \in B. \forall x \in W. \text{All (interf s (dom x)))}$

then case ($\{\}$, False) $|= c \subseteq B_1', Y) \text{ of }$

None $\Rightarrow$ None | Some $\Rightarrow$ case ($\{\}$, False) $|= c \subseteq B_1', Y) \text{ of }$

None $\Rightarrow$ None | Some $\Rightarrow$ Some $(B_2' \cup B_2'', \text{Univ}? B_2' Y \cap Y')$

else None) = Some $(B, W) \Rightarrow$

$\forall r \in C. s_2 = r \subseteq \text{state } \cap Y) \Rightarrow \exists r \in B. s_3 = r \subseteq \text{state } \cap W);$

$\forall A B X Y U v. (U, v) \vdash c \subseteq A, X) = \text{Some} (B, Y) \Rightarrow$

$\exists r \in A. s_1 = r \subseteq \text{state } \cap X) \Rightarrow \exists r \in B. s_2 = r \subseteq \text{state } \cap Y);$

$\forall s \in \text{Univ}? A X \cup \text{Univ}? C Y. \forall x \in \text{bvars b}. \text{All (interf s (dom x))};$

$\forall p \in U. \forall B W. p = (B, W) \Rightarrow (\forall s \in B. \forall x \in W. \text{All (interf s (dom x)))};$

$\forall r \in B_2 \cup B_2'. \exists x \in \text{state } \cap (X \cap Y). s_3 x \neq r x \Rightarrow$

False

by (drule ctyping2-approx-while-aux, assumption+, auto)

lemma ctyping2-approx:

$[[c, s)] \Rightarrow t; (U, v) \vdash c \subseteq A, X) = \text{Some} (B, Y);$

$s \in \text{Univ} A \subseteq \text{state } \cap X) \Rightarrow t \in \text{Univ} B \subseteq \text{state } \cap Y)$

proof (induction arbitrary: $A B X Y U v$ rule: big-step-induct)
fix A B X Y U v b c_1 c_2 s t

show
\[ bval b \; s.; \; (c_1, s) \Rightarrow t; \]
\[ \forall A \; C X Y U \; v. \; (U, v) \models c_1 (\subseteq A, X) = \text{Some} \; (C, Y) \Rightarrow \]
\[ s \in \text{Univ} \; A \; (\subseteq \text{state} \cap X) \Rightarrow \]
\[ t \in \text{Univ} \; C \; (\subseteq \text{state} \cap Y); \]
\[ (U, v) \models \text{IF} \; b \; \text{THEN} \; c_1 \; \text{ELSE} \; c_2 \; (\subseteq A, X) = \text{Some} \; (B, Y); \]
\[ s \in \text{Univ} \; A \; (\subseteq \text{state} \cap X) \Rightarrow \]
\[ t \in \text{Univ} \; B \; (\subseteq \text{state} \cap Y) \]
by (auto split: option.split-asm prod.split-asm, 
rule ctyping2-approx-if-1)

next

fix A B X Y U v b c_1 c_2 s t

show
\[ \neg \; bval b \; s.; \; (c_2, s) \Rightarrow t; \]
\[ \forall A \; C X Y U \; v. \; (U, v) \models c_2 (\subseteq A, X) = \text{Some} \; (C, Y) \Rightarrow \]
\[ s \in \text{Univ} \; A \; (\subseteq \text{state} \cap X) \Rightarrow \]
\[ t \in \text{Univ} \; C \; (\subseteq \text{state} \cap Y); \]
\[ (U, v) \models \text{IF} \; b \; \text{THEN} \; c_1 \; \text{ELSE} \; c_2 \; (\subseteq A, X) = \text{Some} \; (B, Y); \]
\[ s \in \text{Univ} \; A \; (\subseteq \text{state} \cap X) \Rightarrow \]
\[ t \in \text{Univ} \; B \; (\subseteq \text{state} \cap Y) \]
by (auto split: option.split-asm prod.split-asm, 
rule ctyping2-approx-if-2)

next

fix A B X Y U v b c_1 s_1 s_2 s_3

show
\[ bval b \; s.; \; (c_1, s_1) \Rightarrow s_2; \]
\[ \forall A \; B X Y U \; v. \; (U, v) \models c \; (\subseteq A, X) = \text{Some} \; (B, Y) \Rightarrow \]
\[ s_1 \in \text{Univ} \; A \; (\subseteq \text{state} \cap X) \Rightarrow \]
\[ s_2 \in \text{Univ} \; B \; (\subseteq \text{state} \cap Y); \]
\[ (\text{WHILE} \; b \; \text{DO} \; c, s_2) \Rightarrow s_3; \]
\[ \forall A \; B X Y U \; v. \; (U, v) \models \text{WHILE} \; b \; \text{DO} \; c \; (\subseteq A, X) = \text{Some} \; (B, Y) \Rightarrow \]
\[ s_2 \in \text{Univ} \; A \; (\subseteq \text{state} \cap X) \Rightarrow \]
\[ s_3 \in \text{Univ} \; B \; (\subseteq \text{state} \cap Y); \]
\[ (U, v) \models \text{WHILE} \; b \; \text{DO} \; c \; (\subseteq A, X) = \text{Some} \; (B, Y); \]
\[ s_1 \in \text{Univ} \; A \; (\subseteq \text{state} \cap X) \Rightarrow \]
\[ s_3 \in \text{Univ} \; B \; (\subseteq \text{state} \cap Y) \]
by (auto split: if-split-asm option.split-asm prod.split-asm, 
erule_tac [2] ctyping2-approx-while-4, 
erule ctyping2-approx-while-3)

qed (auto split: if-split-asm option.split-asm prod.split-asm)

end
4 Sufficiency of well-typedness for information flow correctness

theory Correctness
  imports Overapproximation
begin

The purpose of this section is to prove that type system $ctyping2$ is correct in that it guarantees that well-typed programs satisfy the information flow correctness criterion expressed by predicate $correct$, namely that if the type system outputs a value other than $None$ (that is, a pass verdict) when it is input program $c$, state set $A$, and vname set $X$, then $correct c A X$ (theorem $ctyping2$-correct).

This proof makes use of the lemmas $ctyping1$-idem and $ctyping2$-approx proven in the previous sections.

4.1 Global context proofs

lemma flow-append-1:
  assumes $A$: $\forall cfs :: (com \times state) \list$. $c \# \map f st (cfs :: (com \times state) \list) = \map f st cfs' \Rightarrow$
  flow-aux ($\map f st cfs' @ \map f st cfs''$) =
  flow-aux ($\map f st cfs' @ flow-aux (\map f st cfs'')$)
  shows flow-aux ($c \# \map f st cfs @ flow-aux (\map f st cfs'')$) =
  flow-aux ($c \# \map f st cfs @ flow-aux (\map f st cfs'')$)
using $A \of \([c, \lambda x. 0] \# cfs\]$ by simp

lemma flow-append:
  $flow (cfs @ cfs') = flow cfs @ flow cfs'$
by (simp add: flow-def, induction map $f st cfs$ arbitrary: $cfs$
rule: flow-aux.induct, auto, rule flow-append-1)

lemma flow-cons:
  $flow (cf \# cfs) = flow-aux (fst cf \# []) @ flow cfs$
by (subgoal-tac cf \# cfs = [cf] \# cfs, simp only: flow-append,
simp-all add: flow-def)

lemma small-steps1-append:
  $[(c, s) \rightarrow\*\{cfs\} (c', s'); (c', s') \rightarrow\*\{cfs'\} (c'', s'')] \Rightarrow$
  $(c, s) \rightarrow\*\{cfs @ cfs'\} (c'', s''')$
by (induction $c' s' cfs' c'' s'''$ rule: small-steps1-induct,
simp, simp only: append-assoc [symmetric] small-steps1.simps)

lemma small-steps1-cons-1:
  $(c, s) \rightarrow\*\{[cf]\} (c', s''') \Rightarrow$
  $cf = (c, s) \&$

66
(∃ c' s'. (c, s) → (c', s') ∧ (c', s') →*{[]} (c'', s''))
by (subst (asm) append-Nil [symmetric],
simp only: small-steps.simps, simp)

**lemma** small-steps-cons-2:
((c, s) →*{cf ≠ cfs} (c'', s''))
  cf = (c, s) ∧
  (∃ c' s'. (c, s) → (c', s') ∧ (c', s') →*{cfs} (c'', s''));
(c, s) →*{cf ≠ cfs @ [(c'', s'')]} (c'', s'')
  cf = (c, s) ∧
  (∃ c' s'. (c, s) → (c', s') ∧
  (c', s') →*{cfs @ [(c'', s'')]} (c'', s''))
by (simp only: append-Cons [symmetric],
simp only: small-steps.simps, simp)

**lemma** small-steps-cons:
(c, s) →*{cf ≠ cfs} (c'', s'')
  cf = (c, s) ∧
  (∃ c' s'. (c, s) → (c', s') ∧ (c', s') →*{cfs} (c'', s''))
by (induction c s cfs c'' s'' rule: small-steps-induct,
    rule small-steps-cons-1, rule small-steps-cons-2)

**lemma** small-steps-steps-1:
∃ cfs. (c, s) →*{cfs} (c, s)
by (rule exl [af - []], simp)

**lemma** small-steps-steps-2:
{(c, s) → (c', s'); (c', s') →*{cfs} (c'', s'')}
  ∃ cfs'. (c, s) →*{cfs'} (c'', s'')
by (rule exl [af - [(c, s) @ cfs]], rule small-steps-append
    where c' = c and s' = s), subst append-Nil [symmetric],
simp only: small-steps.simps)

**lemma** small-steps-steps:
(c, s) →*{cfs} (c', s'')
by (induction c s c' s' rule: star-induct,
    rule small-steps-steps-1, blast intro: small-steps-steps-2)

**lemma** small-steps-steps:
(c, s) →*{cfs} (c', s')
by (induction c s c' s' rule: small-steps-induct,
    auto intro: star-trans)

**lemma** small-steps-skip:
(SKIP, s) →*{cfs} (c, t)
by (induction SKIP s cfs c t rule: small-steps-induct,
    auto simp: flow-def)
lemma small-steps-assign-1:
\[(x ::= a, s) \rightarrow\{c\} \ (c', s') \Rightarrow (c', s') = (x ::= a, s) \wedge \text{flow } [] = [] \lor (c', s') = (\text{SKIP}, (s(x := \text{aval } a \ s)) \wedge \text{flow } [] = [x ::= a])\]
by (simp add: flow-def)

lemma small-steps-assign-2:
\[[x ::= a, s] \rightarrow\{cfs\} \ (c, t) \Rightarrow (c, t) = (x ::= a, s) \wedge \text{flow } cfs = [] \lor (c, t) = (\text{SKIP}, (s(x := \text{aval } a \ s)) \wedge \text{flow } cfs = [x ::= a])\]
by (induction x ::= a :: com s cfs c t rule: small-steps-induct, erule small-steps-assign-1, rule small-steps-assign-2)

lemma small-steps-seq-1:
\[(c_1; c_2, s) \rightarrow\{cfs\} \ (c', s') \Rightarrow (\exists c'' \ c' = c''; c_2 \land (c_1, s) \rightarrow\{cfs'\} \ (c'', s') \land \text{flow } [] = \text{flow } cfs') \lor (\exists s'' \ c' = c''; \text{length } cfs'' < \text{length } [] \land (c_1, s) \rightarrow\{cfs''\} \ (\text{SKIP}, s'') \land (c_2, s'') \rightarrow\{cfs''\} \ (c', s') \land \text{flow } [] = \text{flow } cfs' \land \text{flow } cfs' \land \text{flow } cfs'')\]
by force

lemma small-steps-seq-2:
assumes
A: (c_1; c_2, s) \rightarrow\{cfs\} \ (c', s') \Rightarrow (\exists c'' \ c' = c''; c_2 \land (c_1, s) \rightarrow\{cfs'\} \ (c'', s') \land \text{flow } cfs = \text{flow } cfs' \lor (\exists s'' \ c' = c''; \text{length } cfs'' < \text{length } cfs \land (c_1, s) \rightarrow\{cfs''\} \ (\text{SKIP}, s'') \land (c_2, s'') \rightarrow\{cfs''\} \ (c', s') \land \text{flow } cfs = \text{flow } cfs' \land \text{flow } cfs' \land \text{flow } cfs'')\] and
B: (c_1; c_2, s) \rightarrow\{cfs @ \{[c', s']\}\} \ (c'', s'')
shows
(∃ d cfs'. c'' = d; c2 ∧
 (c1, s) →∗{cfs'} (d, s'') ∧
flow (cfs @ [(c', s')]) = flow cfs') ∨
(∃ t cfs' cfs''. length cfs'' < length (cfs @ [(c', s')]) ∧
 (c1, s) →∗{cfs'} (SKIP, t) ∧
 (c2, t) →∗{cfs''} (c'', s'') ∧
flow (cfs @ [(c', s')]) = flow cfs' @ flow cfs'')
(is ?P ∨ ?Q)

proof –
{assume C: (c', s') → (c'', s'')
 assume
(∃ d. c' = d; c2 ∧ (∃ cfs'.
 (c1, s) →∗{cfs'} (d, s') ∧
flow cfs = flow cfs') ∨
(∃ t cfs' cfs''. length cfs'' < length cfs ∧
 (c1, s) →∗{cfs'} (SKIP, t) ∧
 (c2, t) →∗{cfs''} (c', s') ∧
flow cfs = flow cfs' @ flow cfs'')
(is (∃ d. ?R d ∧ (∃ cfs'. ?S d cfs')) ∨
 (∃ t cfs' cfs''. ?T t cfs' cfs''))

hence ?thesis
proof
assume ∃ c''. ?R c'' ∧ (∃ cfs'. ?S c'' cfs')
then obtain d and cfs' where
 D: c' = d; c2 and
E: (c1, s) →∗{cfs'} (d, s') and
F: flow cfs = flow cfs'
by blast
hence (d; c2, s') → (c'', s'')
using C by simp
moreover {
assume
 G: d = SKIP and
 H: (c'', s'') = (c2, s')
 have ?Q
proof (rule exI [of - s'], rule exI [of - cfs'],
 rule exI [of - []])
 from D and E and F and G and H show
 length [] < length (cfs @ [(c', s')]) ∧
 (c1, s) →∗{cfs'} (SKIP, s') ∧
 (c2, s') →∗{[]} (c'', s'') ∧
flow (cfs @ [(c', s')]) = flow cfs' @ flow []
 by (simp add: flow-append, simp add: flow-def)
qed
}
moreover {
 fix d' t'

69
assume  
\[ G: (d, s') \rightarrow (d', t') \text{ and} \]
\[ H: (c'', s'') = (d'\_; c_2, t') \]

have \(?P\)

proof  
(rule exI [of - d'], rule exI [of - cfs' @ [(d, s')]])

from \(D\) and \(E\) and \(F\) and \(G\) and \(H\) show
\(c'' = d'\_; c_2 \land\)
\((c_1, s) \rightarrow\{cfs'\} \:(d', s'') \land\)
flow \(\{cfs @ [(c', s')]\} = \) flow \(\{cfs' @ [(d, s')]\}\)

by (simp add: flow-append, simp add: flow-def)

qed

}

ultimately show \(?thesis\)

by blast

next

assume \(\exists t\) cfs' cfs'' \(\&\) T t cfs' cfs''

then obtain t and cfs' and cfs'' where
\(D: \text{ length cfs'' < length cfs } \text{ and}\)
\(E: (c_1, s) \rightarrow\{cfs\} \:(SKIP, t) \text{ and}\)
\(F: (c_2, t) \rightarrow\{cfs''\} \:(c', s') \text{ and}\)
\(G: \text{ flow cfs } = \) flow cfs' \& flow cfs''

by blast

show \(?thesis\)

proof  
(rule disjI2, rule exI [of - t], rule exI [of - cfs'],
rule exI [of - cfs'' @ [(c', s')]])

from \(C\) and \(D\) and \(E\) and \(F\) and \(G\) show
length \(\{cfs'' @ [(c', s')]\}\) \(<\)
length \(\{cfs @ [(c', s')]\}\) \land
\((c_1, s) \rightarrow\{cfs\} \:(SKIP, t) \land\)
\((c_2, t) \rightarrow\{cfs'' @ [(c', s')]\} \:(c'', s'') \land\)
flow \(\{cfs @ [(c', s')]\} = \)
flow cfs' \& flow cfs'' @ [(c', s')]

by (simp add: flow-append)

qed

qed

}

with \(A\) and \(B\) show \(?thesis\)

by simp

qed

lemma small-steps-seq:
\( (c_1; c_2, s) \rightarrow\{cfs\} \:(c, t) \implies\)
\(\exists c' cfs', c = c_2 \land\)
\((c_1, s) \rightarrow\{cfs'\} \:(c', t) \land\)
flow cfs = flow cfs' \lor
\(\exists s' cfs' cfs'', \text{ length cfs'' < length cfs } \land\)
\((c_1, s) \rightarrow\{cfs\} \:(SKIP, s') \land (c_2, s') \rightarrow\{cfs'\} \:(c, t) \land\)
flow cfs = flow cfs' @ flow cfs''

by (induction \(c_1; c_2\) s cfs c t arbitrary: \(c_1\) \(c_2\)
rule: small-steps-induct, erule small-steps-seq-1,
lemma small-steps-if-1:
(IF b THEN c₁ ELSE c₂, s) →∗{[]} (c', s') →∗
(c', s') = (IF b THEN c₁ ELSE c₂, s) ∧
flow [] = ∅ ∨
¬ bval b s ∧ (c₁, s) →∗{tl []} (c', s') ∧
flow [] = (bvars b) # flow (tl []) ∨
¬ bval b s ∧ (c₂, s) →∗{tl []} (c', s') ∧
flow [] = (bvars b) # flow (tl [])
by (simp add: flow-def)

lemma small-steps-if-2:
assumes
A: (IF b THEN c₁ ELSE c₂, s) →∗{cfs} (c', s') →∗
(c', s') = (IF b THEN c₁ ELSE c₂, s) ∧
flow cfs = [] ∨
¬ bval b s ∧ (c₁, s) →∗{tl cfs} (c', s') ∧
flow cfs = (bvars b) # flow (tl cfs) ∨
¬ bval b s ∧ (c₂, s) →∗{tl cfs} (c', s') ∧
flow cfs = (bvars b) # flow (tl cfs) and
B: (IF b THEN c₁ ELSE c₂, s) →∗{cfs @ [(c', s')]} (c'', s'')
shows
(c'', s'') = (IF b THEN c₁ ELSE c₂, s) ∧
flow (cfs @ [(c', s')]) = [] ∨
¬ bval b s ∧ (c₁, s) →∗{tl (cfs @ [(c', s')])} (c'', s'') ∧
flow (cfs @ [(c', s')]) = ⟨bvars b⟩ # flow (tl (cfs @ [(c', s')])) ∨
¬ bval b s ∧ (c₂, s) →∗{tl (cfs @ [(c', s')])} (c'', s'') ∧
flow (cfs @ [(c', s')]) = ⟨bvars b⟩ # flow (tl (cfs @ [(c', s')]))
(is b ∨ ?P)

proof –

assume
C: (IF b THEN c₁ ELSE c₂, s) →∗{cfs} (c', s') and
D: (c', s') → (c'', s'')

assume
c' = IF b THEN c₁ ELSE c₂ ∧ s' = s ∧
flow cfs = [] ∨
¬ bval b s ∧ (c₁, s) →∗{tl cfs} (c', s') ∧
flow cfs = (bvars b) # flow (tl cfs) ∨
¬ bval b s ∧ (c₂, s) →∗{tl cfs} (c', s') ∧
flow cfs = (bvars b) # flow (tl cfs)
(is ?Q ∨ ?R ∨ ?S)
hence ?P

proof (rule disjE,erule-tac [2] disjE)
assume ?Q
moreover from this have (IF b THEN c₁ ELSE c₂, s) → (c'', s'')
using D by simp

71
ultimately show \( \text{thesis} \)

using \( C \) by (erule-tac IFE, auto dest: small-steps1-cons
simp: tl-append flow-cons split: list.split)

next
assume \( ?R \)
with \( C \) and \( D \) show \( \text{thesis} \)

by (auto simp: tl-append flow-cons split: list.split)

next
assume \( ?S \)
with \( C \) and \( D \) show \( \text{thesis} \)

by (auto simp: tl-append flow-cons split: list.split)

qed

} with \( \text{A} \) and \( \text{B} \) show \( \text{thesis} \)

by simp

qed

lemma small-steps1-if:

\[
(IF b \ THEN \ c_1 \ ELSE \ c_2, s) \rightarrow^* \{ \text{cfs} \} \ (c, t) \Rightarrow \\
(c, t) = (IF b \ THEN \ c_1 \ ELSE \ c_2, s) \land \\
\text{flow cfs} = [] \lor \\
\text{bval b s} \land (c_1, s) \rightarrow^* \{ \text{tl cfs} \} \ (c, t) \land \\
\text{flow cfs} = (\text{bvars b}) \# \text{flow} (\text{tl cfs}) \lor \\
\neg \text{bval b s} \land (c_2, s) \rightarrow^* \{ \text{tl cfs} \} \ (c, t) \land \\
\text{flow cfs} = (\text{bvars b}) \# \text{flow} (\text{tl cfs})
\]

by (induction \( IF b \ THEN \ c_1 \ ELSE \ c_2 \ s \ cfs \ c \ t \), arbitrary; \( b \ c_1 \ c_2 \) rule: small-steps1-induct, erule small-steps1-if-1, rule small-steps1-if-2)

lemma small-steps1-while-1:

\[
(\text{WHILE } b \ \text{DO } c, s) \rightarrow^* \{ [] \} \ (c', s') \Rightarrow \\
(c', s') = (\text{WHILE } b \ \text{DO } c, s) \land \text{flow } [] = [] \lor \\
(IF b \ THEN \ c_1 \ ;; \ \text{WHILE } b \ \text{DO } c \ \text{ELSE } \text{SKIP}, s) \rightarrow^* \{ \text{tl } [] \} \ (c', s') \land \\
\text{flow } [] = \text{flow} (\text{tl } [])
\]

by (simp add: flow-def)

lemma small-steps1-while-2:

assumes
A: (\text{WHILE } b \ \text{DO } c, s) \rightarrow^* \{ \text{cfs} \} \ (c', s') \Rightarrow \\
(c', s') = (\text{WHILE } b \ \text{DO } c, s) \land \\
\text{flow cfs} = [] \lor \\
(IF b \ THEN \ c_1 \ ;; \ \text{WHILE } b \ \text{DO } c \ \text{ELSE } \text{SKIP}, s) \rightarrow^* \{ \text{tl cfs} \} \ (c', s') \land \\
\text{flow cfs} = \text{flow} (\text{tl cfs}) \ \text{and} \\
B: (\text{WHILE } b \ \text{DO } c, s) \rightarrow^* \{ \text{cfs @ } [(c', s')] \} \ (c'', s'')

shows
\[
(c'', s'') = (\text{WHILE } b \ \text{DO } c, s) \land \\
\text{flow} (\text{cfs @ } [(c', s')]) = [] \lor \\
(IF b \ THEN \ c_1 \ ;; \ \text{WHILE } b \ \text{DO } c \ \text{ELSE } \text{SKIP}, s)
\]
\[\rightarrow^*\{tl\ (cfs \circ [(c', s')])\} (c'', s'') \land
\text{flow\ (cfs \circ [(c', s')])} = \text{flow\ (tl\ (cfs \circ [(c', s')]))}\]

**proof** –

\{  
\quad \text{assume}
\quad C: (\text{WHILE\ }b \text{ DO\ } c, s) \rightarrow^*\{cfs\} (c', s') \text{ and }
\quad D: (c', s') \rightarrow (c'', s'')
\quad \text{assume}
\quad c' = \text{WHILE\ }b \text{ DO\ } c \land s' = s \land
\quad \text{flow\ cfs} = [] \lor
\quad (\text{IF\ }b\ \text{THEN \ }c;\ \text{WHILE\ }b\ \text{DO\ } c\ \text{ELSE\ SKIP}, s) \rightarrow^*\{tl\ cfs\} (c', s') \land
\quad \text{flow\ cfs} = \text{flow\ (tl\ cfs)}
\quad \text{(is \ }?Q \lor \ ?R)
\quad \text{hence \ }?P
\quad \text{proof}
\quad \text{assume \ }?Q
\quad \text{moreover from this have } (\text{WHILE\ }b \text{ DO\ } c, s) \rightarrow (c'', s'')
\quad \text{using \ } D \text{ by \ } \text{simp}
\quad \text{ultimately show \ } ?\text{thesis}
\quad \text{using \ } C \text{ by \ } (\text{erule-tac\ WhileE}, \text{ auto\ dest}: \text{small-stepsl-cons}
\quad \text{ simp: tl-append flow-cons split: list.split})
\quad \text{next}
\quad \text{assume \ } ?R
\quad \text{with \ } C \text{ and \ } D \text{ show \ } ?\text{thesis}
\quad \quad \text{by (auto \ simp: tl-append flow-cons split: list.split)}
\quad \text{qed}
\}

\text{with \ } A \text{ and \ } B \text{ show \ } ?\text{thesis}
\quad \text{by \ } \text{simp}
\text{qed}

**lemma** small-stepsl-while:
\begin{align*}
(\text{WHILE\ }b \text{ DO\ } c, s) & \rightarrow^*\{cfs\} (c', s') \implies \\
(c', s') & = (\text{WHILE\ }b \text{ DO\ } c, s) \land
\text{flow\ cfs} = [] \lor
(\text{IF\ }b\ \text{THEN \ }c;\ \text{WHILE\ }b\ \text{DO\ } c\ \text{ELSE\ SKIP}, s) & \rightarrow^*\{tl\ cfs\} (c', s') \land
\text{flow\ cfs} = \text{flow\ (tl\ cfs)}
\end{align*}

\text{by (induction \ WHILE\ } b \text{ DO\ } c \text{ s \ cfs \ c' s' arbitrary: } b \ c
\text{ rule: small-stepsl-induct, erule small-stepsl-while-1,}
\text{ rule small-stepsl-while-2)}

**lemma** bears-bval:
\begin{align*}
s & = t (\subseteq \text{ bears\ } b) \implies \text{bval\ } b\ s = \text{bval\ } b\ t
\end{align*}

\text{by (induction \ } b, \text{ simp-all, rule arg-cong2, auto intro: avars-aval)}

**lemma** run-flow-append:
\begin{align*}
\text{run-flow\ (cs \circ cs') } s & = \text{run-flow\ cs'} (\text{run-flow\ cs\ s})
\end{align*}
by (induction cs s rule: run-flow.induct, simp-all (no-asmp))

lemma no-upd-append:
no-upd (cs @ cs') x = (no-upd cs x & no-upd cs' x)
by (induction cs, simp-all)

lemma no-upd-run-flow:
no-upd cs x \rightarrow run-flow cs s x = s x
by (induction cs s rule: run-flow.induct, auto)

lemma small-stepsl-run-flow-1:
(c, s) \rightarrow*{} (c', s') \implies s' = run-flow (flow []) s
by (simp add: flow-def)

lemma small-stepsl-run-flow-2:
(c, s) \rightarrow (c', s') \implies s' = run-flow (flow-aux [c]) s
by (induction [c] arbitrary: c c' rule: flow-aux.induct, auto)

lemma small-stepsl-run-flow-3:
[(c, s) \rightarrow*{cfs} (c', s')] \implies s' = run-flow (flow cfs) s;
(c, s) \rightarrow*{cfs @ [(c', s')]} (c'', s'') \implies
s'' = run-flow (flow (cfs @ [(c', s')])) s
by (simp add: flow-append run-flow-append, auto intro: small-stepsl-run-flow-2 simp: flow-def)

lemma small-stepsl-run-flow:
(c, s) \rightarrow*{cfs} (c', s') \implies s' = run-flow (flow cfs) s
by (induction c s cfs c' s' rule: small-stepsl-induct, erule small-stepsl-run-flow-1, rule small-stepsl-run-flow-3)

4.2 Local context proofs

context noninterf
begin

lemma no-upd-sources:
no-upd cs x \rightarrow x \in sources cs s x
by (induction cs s rule: rev-induct, auto simp: no-upd-append split: com-flow.split)

lemma sources-aux-sources:
sources-aux cs s x \subseteq sources cs s x
by (induction cs s rule: rev-induct, auto split: com-flow.split)

lemma sources-aux-append:
sources-aux cs s x \subseteq sources-aux (cs @ cs') s x
by (induction cs' s rule: rev-induct, simp, subst append-assoc [symmetric], auto simp del: append-assoc split: com-flow.split)
lemma sources-aux-observe-hd-1:
\[ \forall y \in X. \text{dom } y \hookrightarrow \text{dom } x \implies X \subseteq \text{sources-aux } ([X]) s x \]
by (subst append-Nil [symmetric], subst sources-aux.simps, auto)

lemma sources-aux-observe-hd-2:
\[ (\forall y \in X. \text{dom } y \hookrightarrow \text{dom } x \implies X \subseteq \text{sources-aux } ((X) \# xs) s x) \implies \]
\[ (\forall y \in X. \text{dom } y \hookrightarrow \text{dom } x \implies X \subseteq \text{sources-aux } ((X) \# xs @ [x]) s x) \]
by (subst append-Cons [symmetric], subst sources-aux.simps, auto split: com-flow.split)

lemma sources-aux-observe-hd:
\[ \forall y \in X. \text{dom } y \hookrightarrow \text{dom } x \implies X \subseteq \text{sources-aux } ((X) \# cs) s x \]

lemma sources-observe-tl-1:
assumes
A: \( \forall z a. \ c = (x ::= a :: \text{com-flow}) \implies z = x \implies \)
\[ \text{sources-aux } cs s x \subseteq \text{sources-aux } ((X) \# cs) s x \text{ and} \]
B: \( \forall z a y. \ c = (x ::= a :: \text{com-flow}) \implies z = x \implies \)
\[ \text{sources } cs s y \subseteq \text{sources } ((X) \# cs) s y \text{ and} \]
C: \( \forall z a. \ c = (z ::= a :: \text{com-flow}) \implies z \neq x \implies \)
\[ \text{sources } cs s x \subseteq \text{sources } ((X) \# cs) s x \text{ and} \]
D: \( \forall Y y. \ c = (Y) \implies \)
\[ \text{sources } cs s y \subseteq \text{sources } ((X) \# cs) s y \text{ and} \]
E: \( z \in (\text{case } c \text{ of} \)
\[ z ::= a \Rightarrow \text{if } z = x \]
\[ \text{then } \text{sources-aux } cs s x \cup \bigcup \{ \text{sources } cs s y \mid y. \]
\[ \text{run-flow } cs s: \text{dom } y \hookrightarrow \text{dom } x \text{ and } y \in \text{avars } a \} \]
\[ \text{else } \text{sources } cs s x \mid \]
\[ (X) \Rightarrow \]
\[ \text{sources } cs s x \cup \bigcup \{ \text{sources } cs s y \mid y. \]
\[ \text{run-flow } cs s: \text{dom } y \hookrightarrow \text{dom } x \text{ and } y \in X \} \)
shows \( z \in \text{sources } ((X) \# cs @ [c]) s x \)
proof –
{ }
fix a
assume
F: \( \forall A. \ (\forall y. \ \text{run-flow } cs s: \text{dom } y \hookrightarrow \text{dom } x \implies \)
\[ A = \text{sources } ((X) \# cs) s y \rightarrow y \notin \text{avars } a) \lor z \notin A \text{ and} \]
G: \( c = x ::= a \)
have \( z \in \text{sources-aux } cs s x \cup \bigcup \{ \text{sources } cs s y \mid y. \]
\[ \text{run-flow } cs s: \text{dom } y \hookrightarrow \text{dom } x \text{ and } y \in \text{avars } a \} \)
using E and G by simp
hence \( z \in \text{sources-aux } ((X) \# cs) s x \)
using A and G proof (erule-tac UnE, blast)
assume \( z \in \bigcup \{ \text{sources } cs s y \mid y. \)
run-flow cs s: dom y ↦ dom x ∧ y ∈ avars a\}
then obtain y where
  H: z ∈ sources cs s y and
  I: run-flow cs s: dom y ↦ dom x and
  J: y ∈ avars a
  by blast
have z ∈ sources (\langle X \rangle ≠ cs) s y
using B and G and H by blast
hence y ∉ avars a
using F and I by blast
thus ?thesis
using J by contradiction
qed
}
moreover {
  fix y a
  assume c = y ::= a and y ≠ x
moreover from this have z ∈ sources cs s x
  using E by simp
ultimately have z ∈ sources (\langle X \rangle ≠ cs) s x
  using C by blast
}
moreover {
  fix Y
  assume
    F: ∀ A. (∀ y. run-flow cs s: dom y ↦ dom x −→ A = sources (\langle X \rangle ≠ cs) s y ↦ y ∉ Y) ∨ z ∉ A and
    G: c = \langle Y \rangle
  have z ∈ sources cs s x \cup ∪ \{sources cs s y | y.
  run-flow cs s: dom y ↦ dom x ∧ y ∈ Y\}
using E and G by simp
hence z ∈ sources (\langle X \rangle ≠ cs) s x
using D and G proof (erule-tac UnE, blast)
  assume z ∈ ∪ \{sources cs s y | y.
  run-flow cs s: dom y ↦ dom x ∧ y ∈ Y\}
then obtain y where
  H: z ∈ sources cs s y and
  I: run-flow cs s: dom y ↦ dom x and
  J: y ∈ Y
  by blast
have z ∈ sources (\langle X \rangle ≠ cs) s y
using D and G and H by blast
hence y ∉ Y
using F and I by blast
thus ?thesis
using J by contradiction
qed
}
ultimately show ?thesis
by \(\text{simp only; append-Cons\ symmetric\ sources.simps, auto split: com-flow.split}\)

qed

lemma \textit{sources-observe-tl-2}:

assumes
A: \(\forall x a. c = (z ::= a :: \text{com-flow}) \implies \text{sources-aux cs s x} \subseteq \text{sources-aux } ((X) \# \text{cs}) s x\) and
B: \(\forall Y. c = \langle Y \rangle \implies \text{sources-aux cs s x} \subseteq \text{sources-aux } ((X) \# \text{cs}) s x\) and
C: \(\forall Y y. c = \langle Y \rangle \implies \text{sources cs s y} \subseteq \text{sources } ((X) \# \text{cs}) s y\) and
\[D: z \in (\text{case c of}\]
\[z ::= a \Rightarrow \text{sources-aux cs s x} \mid \langle X \rangle \Rightarrow \text{sources-aux cs s x} \cup \bigcup \{\text{sources cs s y} \mid y. \text{run-flow cs s: dom y }\sim\text{ dom x }\wedge y \in X\}\]

shows \(z \in \text{sources-aux } ((X) \# \text{cs @ [c]}) s x\)

proof –

{ fix \(y a\)

\[\text{assume } c = y ::= a\]

moreover from this have \(z \in \text{sources-aux cs s x}\)

using \(D\) by simp

ultimately have \(z \in \text{sources-aux } ((X) \# \text{cs}) s x\)

using \(A\) by blast
}

moreover {

fix \(Y\)

assume
\[E: \forall A. (\forall y. \text{run-flow cs s: dom y }\sim\text{ dom x }\implies A = \text{sources } ((X) \# \text{cs}) s y \implies y \notin Y) \lor z \notin A\text{ and}\]
\[F: c = \langle Y \rangle\]

have \(z \in \text{sources-aux cs s x} \cup \bigcup \{\text{sources cs s y} \mid y. \text{run-flow cs s: dom y }\sim\text{ dom x }\wedge y \in Y\}\)

using \(D\) and \(F\) by simp

hence \(z \in \text{sources-aux } ((X) \# \text{cs}) s x\)

using \(B\) and \(F\) proof (erude-tac UnE, blast)

assume \(z \in \bigcup \{\text{sources cs s y} \mid y. \text{run-flow cs s: dom y }\sim\text{ dom x }\wedge y \in Y\}\)

then obtain \(y\) where
\[H: z \in \text{sources cs s y}\]
\[I: \text{run-flow cs s: dom y }\sim\text{ dom x}\]
\[J: y \in Y\]

by blast

have \(z \in \text{sources } ((X) \# \text{cs}) s y\)

using \(C\) and \(F\) and \(H\) by blast

hence \(y \notin Y\)

77
using $E$ and $I$ by blast
thus $\neg \text{thesis}$
using $J$ by contradiction
qed
}
ultimately show $\neg \text{thesis}$
by (simp only: append-Cons [symmetric] sources-aux.simps,
auto split: com-flow.split)
qed

**lemma** sources-observe-tl:
\[
sources\ cs\ s\ x \subseteq sources\ (\langle X \rangle \# cs)\ s\ x
\]
and sources-aux-observe-tl:
\[
sources-aux\ cs\ s\ x \subseteq sources-aux\ (\langle X \rangle \# cs)\ s\ x
\]
**proof** (induction $cs\ s\ x$ and $cs\ s\ x$ rule: sources-induct)
fix $cs\ c\ s\ x$
show
\[
\forall z. a. \ c = z ::= a \Rightarrow z = x \Rightarrow
\quad sources-aux\ cs\ s\ x \subseteq sources-aux\ (\langle X \rangle \# cs)\ s\ x;
\]
\[
\forall z. a. c = z ::= a \Rightarrow z = x \Rightarrow
\quad sources\ cs\ s\ y \subseteq sources\ (\langle X \rangle \# cs)\ s\ y;
\]
\[
\forall Y. c = \langle Y \rangle \Rightarrow
\quad sources\ cs\ s\ x \subseteq sources\ (\langle X \rangle \# cs)\ s\ x;
\]
\[
\forall Y. a. c = \langle Y \rangle \Rightarrow
\quad sources\ cs\ s\ y \subseteq sources\ (\langle X \rangle \# cs)\ s\ y \Rightarrow
\quad sources\ (cs @ [c])\ s\ x \subseteq sources\ (\langle X \rangle \# cs @ [c])\ s\ x
\]
by (auto, rule sources-observe-tl-1)
next
fix $s\ x$
show $\ []\ s\ x \subseteq sources\ [\langle X \rangle]\ s\ x$
by (subst (3) append-Nil [symmetric],
simp only: sources.simps, simp)
next
fix $cs\ c\ s\ x$
show
\[
\forall z. a. c = z ::= a \Rightarrow
\quad sources-aux\ cs\ s\ x \subseteq sources-aux\ (\langle X \rangle \# cs)\ s\ x;
\]
\[
\forall Y. c = \langle Y \rangle \Rightarrow
\quad sources-aux\ cs\ s\ x \subseteq sources-aux\ (\langle X \rangle \# cs)\ s\ x;
\]
\[
\forall Y. a. c = \langle Y \rangle \Rightarrow
\quad sources\ cs\ s\ y \subseteq sources\ (\langle X \rangle \# cs)\ s\ y \Rightarrow
\quad sources-aux\ (cs @ [c])\ s\ x \subseteq sources-aux\ (\langle X \rangle \# cs @ [c])\ s\ x
\]
by (auto, rule sources-observe-tl-2)
qed simp

**lemma** sources-member-1:
assumes
A: \( \forall z. a. \ c = (x := a :: \text{com-flow}) \rightarrow z = x \rightarrow \)
y \in \text{sources-aux cs'} (\text{run-flow cs} \ s) \ x \rightarrow
\text{sources cs} s \ y \subseteq \text{sources-aux} (\text{cs @ cs'}) s x \text{ and}
B: \( \forall z a. w. c = (x := a :: \text{com-flow}) \rightarrow z = x \rightarrow \)
y \in \text{sources cs'} (\text{run-flow cs} \ s) \ w \rightarrow
\text{sources cs} s \ y \subseteq \text{sources} (\text{cs @ cs'}) s w \text{ and}
C: \( \forall z a. c = (z := a :: \text{com-flow}) \rightarrow z \neq x \rightarrow \)
y \in \text{sources cs'} (\text{run-flow cs} \ s) \ x \rightarrow
\text{sources cs} s \ y \subseteq \text{sources} (\text{cs @ cs'}) s x \text{ and}
D: \( \forall y w. c = \langle Y \rangle \rightarrow \)
y \in \text{sources cs'} (\text{run-flow cs} \ s) \ w \rightarrow
\text{sources cs} s \ y \subseteq \text{sources} (\text{cs @ cs'}) s w \text{ and}
E: y \in (\text{case c of})
z := a \Rightarrow \text{if} \ z = x
\text{then sources-aux cs'} (\text{run-flow cs} \ s) \ x \cup
\bigcup \{ \text{sources cs'} (\text{run-flow cs} \ s) \ y \mid y.
\text{run-flow cs'} (\text{run-flow cs} \ s); \ \text{dom y} \rightarrow \text{dom x} \land y \in \text{avars a}\}
\text{else sources cs'} (\text{run-flow cs} \ s) \ x \mid
\langle X \rangle \Rightarrow
\text{sources cs'} (\text{run-flow cs} \ s) \ x \cup
\bigcup \{ \text{sources cs'} (\text{run-flow cs} \ s) \ y \mid y.
\text{run-flow cs'} (\text{run-flow cs} \ s); \ \text{dom y} \rightarrow \text{dom x} \land y \in X\}\} \text{ and}
F: z \in \text{sources cs} s \ y
shows z \in \text{sources} (\text{cs @ cs'} @ [c]) s x
proof –

\{ 
  \text{fix a}
  \text{assume}
  G: \( \forall A. (\forall y. \text{run-flow cs'} (\text{run-flow cs} \ s); \ \text{dom y} \rightarrow \text{dom x} \rightarrow \)
  A = \text{sources} (\text{cs @ cs'}) s y \rightarrow y \notin \text{avars a}) \lor z \notin A \text{ and}
  H: c = x := a
  \text{have} y \in \text{sources-aux cs'} (\text{run-flow cs} \ s) \ x \cup
  \bigcup \{ \text{sources cs'} (\text{run-flow cs} \ s) \ y \mid y.
  \text{run-flow cs'} (\text{run-flow cs} \ s); \ \text{dom y} \rightarrow \text{dom x} \land y \in \text{avars a}\}
  \text{using E and H by simp}
  \text{hence} z \in \text{sources-aux} (\text{cs @ cs'}) s x
  \text{using A and F and H proof (erule-tac UnE, blast)}
  \text{assume} y \in \bigcup \{ \text{sources cs'} (\text{run-flow cs} \ s) \ y \mid y.
  \text{run-flow cs'} (\text{run-flow cs} \ s); \ \text{dom y} \rightarrow \text{dom x} \land y \in \text{avars a}\}
  \text{then obtain w where}
  I: y \in \text{sources cs'} (\text{run-flow cs} \ s) \ w \text{ and}
  J: \text{run-flow cs'} (\text{run-flow cs} \ s); \ \text{dom w} \rightarrow \text{dom x} \text{ and}
  K: w \in \text{avars a}
  \text{by blast}
  \text{have} z \in \text{sources} (\text{cs @ cs'}) s w
  \text{using B and F and H and I by blast}
  \text{hence} w \notin \text{avars a}
  \text{using G and J by blast}

79
thus \( \text{thesis} \)
the \( K \) by contradiction
qed

moreover {
fix \( w \) a
assume \( c = w ::= a \) and \( w \neq x \)
moreover from this have \( y \in \text{sources} cs' (\text{run-flow} cs s) x \)
using \( E \) by simp
ultimately have \( z \in \text{sources} (cs @ cs') s x \)
using \( C \) and \( F \) by blast
}

moreover {
fix \( Y \)
assume \( G: \forall A. (\forall y. \text{run-flow} cs' (\text{run-flow} cs s): \text{dom} y \rightsquigarrow \text{dom} x \longrightarrow A = \text{sources} (cs @ cs') s y \rightarrow y \notin Y) \lor z \notin A \) and
\( H: c = \langle Y \rangle \)

have \( y \in \text{sources} cs' (\text{run-flow} cs s) x \cup \)
\( \{ \text{sources} cs' (\text{run-flow} cs s) y \mid y. \text{run-flow} cs' (\text{run-flow} cs s): \text{dom} y \rightsquigarrow \text{dom} x \land y \in Y \} \)
using \( E \) and \( H \) by simp

hence \( z \in \text{sources} (cs @ cs') s x \)
using \( D \) and \( F \) and \( H \) proof (erule-tac UnE, blast)

assume \( y \in \bigcup \{ \text{sources} cs' (\text{run-flow} cs s) y \mid y. \text{run-flow} cs' (\text{run-flow} cs s): \text{dom} y \rightsquigarrow \text{dom} x \land y \in Y \} \)

then obtain \( w \) where
\( I: y \in \text{sources} cs' (\text{run-flow} cs s) w \) and
\( J: \text{run-flow} cs' (\text{run-flow} cs s): \text{dom} w \rightsquigarrow \text{dom} x \) and
\( K: w \in Y \)
by blast

have \( z \in \text{sources} (cs @ cs') s w \)
using \( D \) and \( F \) and \( H \) and \( I \) by blast

hence \( w \notin Y \)
using \( G \) and \( J \) by blast
thus \( \text{thesis} \)
using \( K \) by contradiction
qed

ultimately show \( \text{thesis} \)
by (simp only: append-assoc [symmetric] sources.simps, 
auto simp: run-flow-append split: com-flow.split)

qed

lemma sources-member-2:
assumes
\( A: \forall z a. c = (z ::= a :: \text{com-flow}) \longrightarrow y \in \text{sources-aux cs' (run-flow cs s) x} \longrightarrow \)
\( \text{s} \text{ources cs s y} \subseteq \text{sources-aux} (cs @ cs') s x \) and
\( B: \forall Y. \ c = \langle Y \rangle \implies \langle Y \rangle \)
\( y \in \text{sources-aux cs' (run-flow cs s)} \ x \implies \)
\( \text{sources cs s y } \subseteq \text{sources-aux (cs @ cs')} s x \) \text{ and}
\( C: \forall Y w. \ c = \langle Y \rangle \implies \langle Y \rangle \)
\( y \in \text{sources cs' (run-flow cs s)} w \implies \)
\( \text{sources cs s y } \subseteq \text{sources (cs @ cs')} s w \) \text{ and}
\( D: y \in (\text{case c of}) \)
\( z := a \Rightarrow \)
\( \text{sources-aux cs' (run-flow cs s)} x | \)
\( \langle X \rangle \Rightarrow \)
\( \text{sources-aux cs' (run-flow cs s)} X \cup \)
\( \bigcup \{ \text{sources cs' (run-flow cs s)} y | y. \)
\( \text{run-flow cs' (run-flow cs s)}: \text{dom y } \mapsto \text{dom x } \land \ y \in X \}) \) \text{ and}
\( E: z \in \text{sources cs s y} \)
\( \text{shows z } \in \text{sources-aux (cs @ cs' @ [c]) s x} \)
\( \text{proof} – \)
\{ \)
\( \text{fix w a} \)
\( \text{assume c = w ::= a} \)
\( \text{moreover from this have y } \in \text{sources-aux cs' (run-flow cs s)} x \)
\( \text{using D by simp} \)
\( \text{ultimately have z } \in \text{sources-aux (cs @ cs')} s x \)
\( \text{using A and E by blast} \)
\}\n\text{moreover { \)
\( \text{fix Y} \)
\( \text{assume} \)
\( G: \forall A. (\forall y. \text{run-flow cs' (run-flow cs s)}: \text{dom y } \mapsto \text{dom x } \implies \)
\( A = \text{sources (cs @ cs')} s y \implies y \notin Y) \lor z \notin A \) \text{ and}
\( H: c = \langle Y \rangle \)
\( \text{have y } \in \text{sources-aux cs' (run-flow cs s)} x \cup \)
\( \bigcup \{ \text{sources cs' (run-flow cs s)} y | y. \)
\( \text{run-flow cs' (run-flow cs s)}: \text{dom y } \mapsto \text{dom x } \land \ y \in Y \}) \) \text{ and}
\( \text{using D and H by simp} \)
\( \text{hence z } \in \text{sources-aux (cs @ cs')} s x \)
\text{using B and E and H proof (erule-tac UnE; blast) \)
\( \text{assume y } \in \bigcup \{ \text{sources cs' (run-flow cs s)} y | y. \)
\( \text{run-flow cs' (run-flow cs s)}: \text{dom y } \mapsto \text{dom x } \land \ y \in Y \}) \)
\text{then obtain w where \)
\( I: y \in \text{sources cs' (run-flow cs s)} w \) \text{ and}
\( J: \text{run-flow cs' (run-flow cs s)}: \text{dom w } \mapsto \text{dom x } \) \text{ and}
\( K: w \in Y \)
\text{by blast} \)
\( \text{have z } \in \text{sources (cs @ cs')} s w \)
\text{using C and E and H and I by blast} \)
\( \text{hence w } \notin Y \)
\text{using G and J by blast} \)
\text{thus ?thesis} \)
\text{using K by contradiction}
ultimately show ?thesis
by (simp only: append-assoc [symmetric] sources-aux.simps,
auto simp: run-flow-append split: com-flow.split)

qed

lemma sources-member:
y ∈ sources cs' (run-flow cs s) x ⇒
sources cs s y ⊆ sources (cs @ cs') s x

and sources-aux-member:
y ∈ sources-aux cs' (run-flow cs s) x ⇒
sources cs s y ⊆ sources-aux (cs @ cs') s x

proof (induction cs' s x and cs' s x rule: sources-induct)
fix cs' c s x

show

[⋀z a. c = z ::= a ⇒ z = x ⇒
y ∈ sources-aux cs' (run-flow cs s) x ⇒
sources cs s y ⊆ sources-aux (cs @ cs') s x;]
⋀z a b w. c = z ::= a ⇒ z = x ⇒
y ∈ sources cs' (run-flow cs s) w ⇒
sources cs s y ⊆ sources (cs @ cs') s w;
⋀z a. c = z ::= a ⇒ z ≠ x ⇒
y ∈ sources cs' (run-flow cs s) x ⇒
sources cs s y ⊆ sources (cs @ cs') s x;
⋀Y. c = ⟨Y⟩ ⇒
y ∈ sources cs' (run-flow cs s) x ⇒
sources cs s y ⊆ sources (cs @ cs') s x;
⋀Y a w. c = ⟨Y⟩ ⇒
y ∈ sources cs' (run-flow cs s) w ⇒
sources cs s y ⊆ sources (cs @ cs') s w;
y ∈ sources (cs' @ [c]) (run-flow cs s) x ⇒
sources cs s y ⊆ sources (cs @ cs' @ [c]) s x
by (auto, rule sources-member-1)

next

fix cs' c s x

show

[⋀z a. c = z ::= a ⇒
y ∈ sources-aux cs' (run-flow cs s) x ⇒
sources cs s y ⊆ sources-aux (cs @ cs') s x;]
⋀Y. c = ⟨Y⟩ ⇒
y ∈ sources-aux cs' (run-flow cs s) x ⇒
sources cs s y ⊆ sources-aux (cs @ cs') s x;
⋀Y a w. c = ⟨Y⟩ ⇒
y ∈ sources cs' (run-flow cs s) w ⇒
sources cs s y ⊆ sources (cs @ cs') s w;
y ∈ sources-aux (cs' @ [c]) (run-flow cs s) x ⇒
sources cs s y ⊆ sources-aux (cs @ cs' @ [c]) s x
by (auto, rule sources-member-2)
lemma ctying2-confine:
[(c, s) ⇒ s'; (U, v) |= c (⊆ A, X) = Some (B, Y);
  ∃(C, Z) ∈ U. ¬ C: dom ' Z ⇒ {dom x}] ⇒ s' x = s x
by (induction arbitrary: A B X Y U v rule: big-step-induct,
  auto split: if-split-asn option.split-asn prod.split-asn, fastforce+)

lemma ctying2-term-if:
[∀x' y' z'' s. x' = x ⇒ y' = y ⇒ z = z'' ⇒ ∃s'. (c1, s) ⇒ s';
  ∃x' y' z'' s. x' = x ⇒ y' = y ⇒ z = z'' ⇒ ∃s'. (c2, s) ⇒ s'] ⇒
  ∃s'. (IF b THEN c1 ELSE c2, s) ⇒ s'
by (cases b s, fastforce+)

lemma ctying2-term:
[(U, v) |= c (⊆ A, X) = Some (B, Y);
  ∃(C, Z) ∈ U. ¬ C: dom ' Z ⇒ UNIV] ⇒ ∃s'. (c, s) ⇒ s'
by (induction (U, v) c A X arbitrary: B Y U s rule: ctying2-induct,
  auto split: if-split-asn option.split-asn prod.split-asn, fastforce,
  erule ctying2-term-if)

lemma ctying2-correct-aux-assign [elim]:
[(SKIP, s) →*{cfs1} (c1, s1); (c1, s1) →*{cfs2} (c2, s2)] ⇒
  (∀t1. ∃c2', t2. ∀x.
    (s1 = t1 (⊆ sources-aux (flow cfs2) s1 x) −→
      (c1, t1) →* (c2', t2) ∧ (c2 = SKIP) = (c2' = SKIP)) ∧
    (s1 = t1 (⊆ sources (flow cfs2) s1 x) −→
      (s2 x = t2 x)) ∧
    (∀x. (∀p ∈ U. case p of (B, W) ⇒
      ∃s ∈ B. ∃y ∈ W. ¬ s: dom y −→ dom x) −→ no-upd (flow cfs2) x)
  )
by (fastforce dest: small-steps-skip)

lemma ctying2-correct-aux-assign [elim]:
assumes
  A: (if (∀s ∈ Univ? A X. ∀y ∈ avars a. s: dom y −→ dom x) ∧
    (∀p ∈ U. ∀B Y. p = (B, Y) −→
      (∀s ∈ B. ∀y ∈ Y. s: dom y −→ dom x))
    then Some (if x ∈ state ∧ A ≠ {})
    then if v |= a (⊆ X)
    then (s(x := aval a s) | s. s ∈ A), insert x X)
    else (A, X = {x})
    else (A, Univ?? A X))
  else None) = Some (B, Y)
(is (if ?P then - else -) = -) and
B: (x := a, s) →*{cfs1} (c1, s1) and
C: (c1, s1) →*{cfs2} (c2, s2) and
D: r ∈ A and
E: s = r (⊆ state ∩ X)

qed simp-all
shows

\( \forall t_1. \exists c_2' t_2. \forall x.
\)
\( (s_1 = t_1 \subseteq \text{sources-aux} (\text{flow cfs}_2) s_1 x) \rightarrow \\
(c_1, t_1) \rightarrow^* (c_2', t_2) \land (c_2 = \text{SKIP}) = (c_2' = \text{SKIP}) \land \\
(s_1 = t_1 \subseteq \text{sources} (\text{flow cfs}_2) s_1 x) \rightarrow s_2 x = t_2 x) \land \\
(\forall x. (\exists p \in U. \text{case } p \text{ of } (B, Y) \Rightarrow \\
\exists s \in B. \exists y \in Y. \neg s: \text{dom } y \rightarrow \text{dom } x) \rightarrow \text{no-upd} (\text{flow cfs}_2) x)
\)

proof

have \( ?P \)

using \( A \) by (simp split: \text{if-split-asm})

have \( F \): avars \( a \subseteq \{ y, s : \text{dom } y \rightarrow \text{dom } x \} \)

proof (cases state \( \subseteq X \))

case True

with \( E \) have \( \text{interf } s = \text{interf } r \)

by (blast intro: interf-state)

with \( D \) and \( (?P) \) show \( ?thesis \)

by (erule-tac conjE, drule-tac bspec, auto simp: \text{univ-states-if-def})

next

case False

with \( D \) and \( (?P) \) show \( ?thesis \)

by (erule-tac conjE, drule-tac bspec, auto simp: \text{univ-states-if-def})

qed

have \( (c_1, s_1) = (x := a, s) \lor (c_1, s_1) = (\text{SKIP}, s(x := \text{aval } a s)) \)

using \( B \) by (blast dest: small-steps-assign)

thus \( ?thesis \)

proof

assume \( (c_1, s_1) = (x := a, s) \)

moreover from this have \( (x := a, s) \rightarrow^* \{cfs}_2 (c_2, s_2) \)

using \( C \) by simp

hence \( (c_2, s_2) = (x := a, s) \land \text{flow cfs}_2 = [] \lor \\
(c_2, s_2) = (\text{SKIP}, s(x := \text{aval } a s)) \land \text{flow cfs}_2 = [x := a] \)

by (rule small-steps-assign)

moreover \{ 

fix \( t \)

have \( \exists c' t'. \forall y.
\)
\( (y = x \rightarrow \\
(s = t \subseteq \text{sources-aux} [x := a] s x) \rightarrow \\
(x := a, t) \rightarrow^* (c', t') \land c' = \text{SKIP} \land \\
(s = t \subseteq \text{sources} [x := a] s x) \rightarrow \text{aval } a s = t' x) \land \\
(y \neq x \rightarrow \\
(s = t \subseteq \text{sources-aux} [x := a] s y) \rightarrow \\
(x := a, t) \rightarrow^* (c', t') \land c' = \text{SKIP} \land \\
(s = t \subseteq \text{sources} [x := a] s y) \rightarrow s y = t' y)
\)

proof (rule extl [of - \text{SKIP}], rule extl [of - \text{t}(x := \text{aval } a t)])

\{ 

assume \( s = t \subseteq \text{sources} [x := a] s x \)

hence \( s = t \subseteq \{ y. s : \text{dom } y \rightarrow \text{dom } x \land y \in \text{avers } a \} \)

by (subst (asm) append-Nil [symmetric],
simp only: sources.simps, auto)

\}

84
hence \( \text{aval } a \ s = \text{aval } a \ t \)
using \( F \) by (blast intro: avars-aval)
}\}
moreover { 
fix \( y \)
assume \( s = t (\subseteq \text{sources} \ [x ::= a] \ s \ y) \) and \( y \not= x \)
hence \( s \ y = t \ y \)
by (subst (asm) append-Nil [symmetric],
simp only: sources.simps, auto)
}\}
ultimately show \( \forall y \).
\( (y = x) \)
\( (s = t (\subseteq \text{sources-aux} \ [x ::= a] \ s \ x) \)
\( (x ::= a, t) \rightarrow (\text{SKIP}, t(x ::= \text{aval } a \ t)) \wedge \text{SKIP} = \text{SKIP} \wedge \)
\( (s = t (\subseteq \text{sources} \ [x ::= a] \ s \ y) \)
\( \text{aval } a \ s = (t(x ::= \text{aval } a \ t)) \ y) \wedge \)
\( (y \not= x) \)
\( (s = t (\subseteq \text{sources-aux} \ [x ::= a] \ s \ y) \)
\( (x ::= a, t) \rightarrow (\text{SKIP}, t(x ::= \text{aval } a \ t)) \wedge \text{SKIP} = \text{SKIP} \wedge \)
\( (s = t (\subseteq \text{sources} \ [x ::= a] \ s \ y) \)
\( s \ y = (t(x ::= \text{aval } a \ t)) \ y) \)
by simp
qed
}\}
ultimately show \( \exists \text{thesis} \)
using \( \exists \text{P} \) by fastforce
next
assume \( (c_1, s_1) = (\text{SKIP}, s(x ::= \text{aval } a \ s)) \)
moreover from this have \( (\text{SKIP}, s(x ::= \text{aval } a \ s)) \rightarrow \{ \text{cfs}_2 \} (c_2, s_2) \)
using \( C \) by simp
hence \( (c_2, s_2) = (\text{SKIP}, s(x ::= \text{aval } a \ s)) \wedge \text{flow cfs}_2 = [] \)
by (rule small-steps-skip)
ultimately show \( \exists \text{thesis} \)
by auto
qed
qed

lemma ctyping2-correct-aux-seq:
assumes
\( A: \bigwedge B' \ s \ c' \ c'' \ s_1 \ s_2 \ \text{cfs}_1 \ \text{cfs}_2. \ B = B' \implies \)
\( \exists r \in A. \ s = r (\subseteq \text{state} \cap X) \implies \)
\( (c_1, s) \rightarrow (\text{cfs}_1) (c', s_1) \implies (c', s_1) \rightarrow (\text{cfs}_2) (c'', s_2) \implies \)
\( \forall t_1. \exists c_2', t_2. \forall x. \)
\( (s_1 = t_1 (\subseteq \text{sources-aux} (\text{flow cfs}_2) s_1 x) \implies \)
\( (c', t_1) \rightarrow (c_2', t_2) \wedge (c'' = \text{SKIP}) = (c_2' = \text{SKIP}) \wedge \)
\( (s_1 = t_1 (\subseteq \text{sources} (\text{flow cfs}_2) s_1 x) \rightarrow s_2 x = t_2 x) \wedge \)
\( \forall x. (\exists p \in U. \text{ case } p \ of (B, W) \Rightarrow \)
\( \exists s \in B. \exists y \in W. \neg s \dom y \rightarrow s \dom x) \implies \)
\( \text{no-upd} (\text{flow cfs}_2) x \) and
B: \( B' \cap C \cap Z = c' \cap c'' \cap s_1 \cap s_2 \cap \text{cfs}_1 \cap \text{cfs}_2 \). \( B = B' \implies B'' = B' \implies (U, v) \models c_2 (\subseteq B', Y) = \text{Some} (C, Z) \implies \exists r \in B'. s = r (\subseteq \text{state} \cap Y) \implies (c_2, s) \implies (c'_{s_1}, s_1) \implies (c', s_1) \implies (c''_{s_2}, s_2) \implies 
\begin{align*}
(\forall t_1, \exists c'_2 \cap t_2, \forall x. \\
(s_1 = t_1 (\subseteq \text{sources-aux (flow } \text{cfs}_2) s_1 x) \implies \\
(c', t_1) \implies (c'_2, t_2) \cap (c'' = \text{SKIP}) = (c'_2 = \text{SKIP})) \land \\
(s_1 = t_1 (\subseteq \text{sources (flow } \text{cfs}_2) s_1 x) \implies s_2 x = t_2 x) \land \\
(\forall x. (\exists p \in U. \text{ case } p \text{ of } (B, W) \implies \\
\exists s \in B. \exists y \in W. \neg s: \text{dom } y \rightsquigarrow \text{dom } x) \implies \\
\text{no-upd (flow } \text{cfs}_2 x) \land 
\end{align*}

C: \( (U, v) \models c_2 (\subseteq A, X) = \text{Some} (B, Y) \) and

D: \( (U, v) \models c_2 (\subseteq B, Y) = \text{Some} (C, Z) \) and

E: \( (c_1 ; c_2, s) \implies (c'_{s_1}, s_1) \) and

F: \( (c', s_1) \implies (c''_{s_2}, s_2) \) and

G: \( r \in A \) and

H: \( s = r (\subseteq \text{state} \cap X) \)

shows
\( (\forall t_1, \exists c'_2 \cap t_2, \forall x. \\
(s_1 = t_1 (\subseteq \text{sources-aux (flow } \text{cfs}_2) s_1 x) \implies \\
(c', t_1) \implies (c'_2, t_2) \cap (c'' = \text{SKIP}) = (c'_2 = \text{SKIP})) \land \\
(s_1 = t_1 (\subseteq \text{sources (flow } \text{cfs}_2) s_1 x) \implies s_2 x = t_2 x) \land \\
(\forall x. (\exists p \in U. \text{ case } p \text{ of } (B, W) \implies \\
\exists s \in B. \exists y \in W. \neg s: \text{dom } y \rightsquigarrow \text{dom } x) \implies \\
\text{no-upd (flow } \text{cfs}_2 x) \)

\textbf{proof –}

\textbf{have}
\( (\exists d' \in \text{cfs}. c' = d' ; c_2 \land \\
(c_1, s) \implies (c'_{s_1}) \lor \\
(\exists s' \in \text{cfs}\text{cfs}'. \\
(c_1, s) \implies (c'_{s_1}) \land \\
(c_2, s') \implies (c''_{s_2}) \land \\
\text{using } E \text{ by } (\text{blast dest: small-steps-seq}) \)

\textbf{thus } \text{?thesis}

\textbf{proof } (\text{rule disjE}, (\text{erule-tac exE}+) ; (\text{erule-tac } [2] \text{ exE})+, \text{erule-tac } [1], \text{conjE})

\textbf{fix } d' \text{ cfs}

\textbf{assume}
I: \( c' = d' ; c_2 \) and

J: \( (c_1, s) \implies (c'_{s_1}) \)

\textbf{hence } (d' ; c_2, s_1) \implies (c''_{s_2}) \land 

\textbf{using } F \text{ by } \text{simp}

\hence
\( (\exists d'' \text{ cfs'}. c'' = d'' ; c_2 \land \\
(d', s_1) \implies (c''_{s_2}) \land \\
\text{flow cfs}_2 = \text{flow cfs'} \lor \\
(\exists s' \in \text{cfs'cfs''}. \\
(d', s_1) \implies (c''_{s_2}) \land \\
(c_2, s') \implies (c''_{s_2}) \land \\
\text{flow cfs}_2 = \text{flow cfs'} \& \text{flow cfs''}) \)

86
by (blast dest: small-steps-seq)
thus ?thesis
proof (rule disjE, (erule-tac exE)+, (erule-tac [2] exE)+,
(erule-tac [|] conjE)+)
fix d'' cfs'
assume (d', s1) ->{cfs'} (d'', s2)
hence K:
(\forall t_1. \exists c_2 t_2. \forall x.
  (s_1 = t_1 (\subseteq sources-aux (flow cfs') s_1 x) \longrightarrow
   (d', t_1) ->* (c_2', t_2) \wedge (d'' = SKIP) = (c_2' = SKIP)) \wedge
  (s_1 = t_1 (\subseteq sources (flow cfs') s_1 x) \longrightarrow s_2 x = t_2 x)) \wedge
(\forall x. (\exists p \in U. case p of (B, W) \Rightarrow
  \exists s \in B. \exists y \in W. \neg s: dom y \sim dom x) \longrightarrow no-upd (flow cfs') x)
using A [of B s cfs d' s_1 cfs'' s_2] and J and G and H by blast
moreover assume c'' = d'';; c_2 and flow cfs_2 = flow cfs'
moreover {
fix t_1
obtain c_2' and t_2 where L: \forall x.
  (s_1 = t_1 (\subseteq sources-aux (flow cfs') s_1 x) \longrightarrow
   (d', t_1) ->* (c_2', t_2) \wedge (d'' = SKIP) = (c_2' = SKIP)) \wedge
  (s_1 = t_1 (\subseteq sources (flow cfs') s_1 x) \longrightarrow s_2 x = t_2 x)
using K by blast
have \exists c_2' t_2. \forall x.
  (s_1 = t_1 (\subseteq sources-aux (flow cfs') s_1 x) \longrightarrow
   (d'', c_2', t_1) ->* (c_2', t_2) \wedge c_2' \neq SKIP) \wedge
  (s_1 = t_1 (\subseteq sources (flow cfs') s_1 x) \longrightarrow s_2 x = t_2 x)
proof (rule exI [of - c_2'';; c_2], rule exI [of - t_2])
show \forall x.
  (s_1 = t_1 (\subseteq sources-aux (flow cfs') s_1 x) \longrightarrow
   (d'', c_2, t_1) ->* (c_2'', c_2, t_2) \wedge c_2'' \neq SKIP) \wedge
  (s_1 = t_1 (\subseteq sources (flow cfs') s_1 x) \longrightarrow s_2 x = t_2 x)
using L by (auto intro: star-seq2)
qed }
ultimately show ?thesis
using I by auto
next
fix s' cfs' cfs''
assume
K: (d', s_1) ->{cfs'} (SKIP, s') and
L: (c_2, s') ->{cfs''} (c'', s_2)
moreover have M: s' = run-flow (flow cfs') s_1
using K by (rule small-steps-run-flow)
ultimately have N:
(\forall t_1. \exists c_2 t_2. \forall x.
  (s_1 = t_1 (\subseteq sources-aux (flow cfs') s_1 x) \longrightarrow
   (d', t_1) ->* (c_2', t_2) \wedge (SKIP = SKIP) = (c_2' = SKIP)) \wedge
  (s_1 = t_1 (\subseteq sources (flow cfs') s_1 x) \longrightarrow
   run-flow (flow cfs') s_1 x = t_2 x)) \wedge

87
(\forall x. (\exists p \in U. \text{ case } p \text{ of } (B, W) \Rightarrow \\
\exists s \in B. \exists y \in W. \neg s: \text{ dom } y \leadsto \text{ dom } x) \rightarrow \text{ no-upd } (\text{ flow cfs'}) x) \\
\text{ using } A [\text{ of } B \text{ s cfs' } s_1] \text{ and } J \text{ and } G \text{ and } H \text{ by blast} \\
\text{ have } O: s_2 = \text{ run-flow } (\text{ flow cfs''}) s' \\
\text{ using } L \text{ by (rule small-steps-run-flow)} \\
\text{ moreover have } (c_1, s) \rightarrow \{\text{ cfs } @ \text{ cfs'}\} (\text{ SKIP, } s') \\
\text{ using } J \text{ and } K \text{ by (simp add: small-steps-append)} \\
\text{ hence } (c_1, s) \Rightarrow s' \\
\text{ by (auto dest: small-steps-steps simp: big-iff-small)} \\
\text{ hence } s' \in \text{ Univ } B (\subseteq \text{ state } \cap Y) \\
\text{ using } C \text{ and } G \text{ and } H \text{ by (erule-tac ctyping2-approx, auto)} \\
\text{ ultimately have } P: \\
(\forall t_1. \exists c_2' t_2. \forall x. \\
(\text{ run-flow } (\text{ flow cfs'}) s_1 = t_1 \\
\subseteq \text{ sources-aux } (\text{ flow cfs''}) (\text{ run-flow } (\text{ flow cfs'}) s_1) x) \rightarrow \\
(c_2, t_1) \rightarrow \ast (c_2', t_2) \land (c'' = \text{ SKIP}) = (c_2' = \text{ SKIP}) \land \\
(\text{ run-flow } (\text{ flow cfs'}) s_1 = t_1 \\
\subseteq \text{ sources-aux } (\text{ flow cfs''}) (\text{ run-flow } (\text{ flow cfs'}) s_1) x) \rightarrow \\
\text{ run-flow } (\text{ flow cfs''}) (\text{ run-flow } (\text{ flow cfs'}) s_1) x = t_2 x) \land \\
(\forall x. (\exists p \in U. \text{ case } p \text{ of } (B, W) \Rightarrow \\
\exists s \in B. \exists y \in W. \neg s: \text{ dom } y \leadsto \text{ dom } x) \rightarrow \text{ no-upd } (\text{ flow cfs''}) x) \\
\text{ using } B [\text{ of } B \text{ C Z s' } \ll c_2' s' \text{ cfs'' c'' s}_2] \\
\text{ and } D \text{ and } L \text{ and } M \text{ by simp} \\
\text{ moreover assume } \text{ flow cfs}_2 = \text{ flow cfs' } @ \text{ flow cfs''} \\
\text{ moreover } \{ \\
\text{ fix } t_1 \\
\text{ obtain } c_2' \text{ and } t_2 \text{ where } Q: \forall x. \\
(s_1 = t_1 (\subseteq \text{ sources-aux } (\text{ flow cfs'}) s_1) x) \rightarrow \\
(d', t_1) \rightarrow \ast (\text{ SKIP}, t_2) \land (\text{ SKIP } = \text{ SKIP}) = (c_2' = \text{ SKIP}) \land \\
(s_1 = t_1 (\subseteq \text{ sources-aux } (\text{ flow cfs'}) s_1) x) \rightarrow \\
\text{ run-flow } (\text{ flow cfs'}) s_1 x = t_2 x) \\
\text{ using } N \text{ by blast} \\
\text{ obtain } c_3' \text{ and } t_3 \text{ where } R: \forall x. \\
(\text{ run-flow } (\text{ flow cfs'}) s_1 = t_2 \\
\subseteq \text{ sources-aux } (\text{ flow cfs'}) (\text{ run-flow } (\text{ flow cfs'}) s_1) x) \rightarrow \\
(c_2, t_2) \rightarrow \ast (c_3', t_3) \land (c'' = \text{ SKIP}) = (c_3' = \text{ SKIP}) \land \\
(\text{ run-flow } (\text{ flow cfs'}) s_1 = t_2 \\
\subseteq \text{ sources-aux } (\text{ flow cfs''}) (\text{ run-flow } (\text{ flow cfs'}) s_1) x) \rightarrow \\
\text{ run-flow } (\text{ flow cfs''}) (\text{ run-flow } (\text{ flow cfs'}) s_1) x = t_3 x) \\
\text{ using } P \text{ by blast} \} \\
\{ \\
\text{ fix } x \\
\text{ assume } S: s_1 = t_1 \\
(\subseteq \text{ sources-aux } (\text{ flow cfs' } @ \text{ flow cfs''}) s_1 x) \\
\text{ moreover have } \text{ sources-aux } (\text{ flow cfs'}) s_1 x \subseteq \\
\text{ sources-aux } (\text{ flow cfs' } @ \text{ flow cfs''}) s_1 x \\
\text{ by (rule sources-aux-append)} \\
\text{ ultimately have } (d', t_1) \rightarrow \ast (\text{ SKIP}, t_2) \\
\text{ using } Q \text{ by blast} \}
hence \((d';; c_2, t_1) \rightarrow^* (\text{SKIP};; c_2, t_2)\)
by (rule star-seq2)

hence \((d';; c_2, t_1) \rightarrow^* (c_2, t_2)\)
by (blast intro: star-trans)

moreover have run-flow (flow cfs') \(s_1 = t_2\)
\(\subseteq\text{sources-aux} (\text{flow cfs''})\) (run-flow (flow cfs') \(s_1\) \(x\))

proof
fix \(y\)
assume \(y \in \text{sources-aux} (\text{flow cfs''})\)
(run-flow (flow cfs') \(s_1\) \(x\))

hence sources (flow cfs') \(s_1\) \(y\) \(\subseteq\)
\(\text{sources-aux} (\text{flow cfs' @ flow cfs''})\) \(s_1\) \(x\)
by (rule sources-aux-member)

thus run-flow (flow cfs') \(s_1\) \(y\) = \(t_2\) \(y\)

using \(Q\) and \(S\) by blast

qed

hence \((c_2, t_2) \rightarrow^* (c_3', t_3) \wedge (c'' = \text{SKIP}) = (c_3' = \text{SKIP})\)
using \(R\) by simp

ultimately have \((d''; c_2, t_1) \rightarrow^* (c_3', t_3) \wedge (c'' = \text{SKIP}) = (c_3' = \text{SKIP})\)
by (blast intro: star-trans)

}

moreover {
fix \(x\)
assume \(S: s_1 = t_1\)
\(\subseteq\text{sources} (\text{flow cfs' @ flow cfs''})\) \(s_1\) \(x\)

have run-flow (flow cfs') \(s_1 = t_2\)
\(\subseteq\text{sources} (\text{flow cfs''})\) (run-flow (flow cfs') \(s_1\) \(x\))

proof
fix \(y\)
assume \(y \in \text{sources} (\text{flow cfs''})\)
(run-flow (flow cfs') \(s_1\) \(x\))

hence sources (flow cfs') \(s_1\) \(y\) \(\subseteq\)
\(\text{sources} (\text{flow cfs' @ flow cfs''})\) \(s_1\) \(x\)
by (rule sources-member)

thus run-flow (flow cfs') \(s_1\) \(y\) = \(t_2\) \(y\)

using \(Q\) and \(S\) by blast

qed

hence run-flow (flow cfs'') (run-flow (flow cfs') \(s_1\) \(x\) = \(t_3\) \(x\))
using \(R\) by simp

}

ultimately have \(\exists c_2' t_2.\ \forall x.\)
\((s_1 = t_1 \subseteq\text{sources-aux} (\text{flow cfs' @ flow cfs''})\) \(s_1\) \(x\) \(\rightarrow\)
\((d''; c_2', t_1) \rightarrow^* (c_2', t_2) \wedge (c'' = \text{SKIP}) = (c_3' = \text{SKIP})\) \(\wedge\)
\((s_1 = t_1 \subseteq\text{sources} (\text{flow cfs' @ flow cfs''})\) \(s_1\) \(x\) \(\rightarrow\)
run-flow (flow cfs'') (run-flow (flow cfs') \(s_1\) \(x\) = \(t_2\) \(x\))
by auto

}

ultimately show \(\text{thesis}\)
using $I$ and $N$ and $M$ and $O$ by (auto simp: no-upd-append)

qed

next

fix $s'$ cfs cfs'

assume $(c_1, s) \mapsto\{cfs\} (SKIP, s')$

hence $(c_1, s) \Rightarrow s'$

by (auto dest: small-steps-steps simp: big-iff-small)

hence $s' \in \text{Univ} B (\subseteq \text{state} \cap Y)$

using $C$ and $G$ and $H$ by (erule-tac ctyping2-approx, auto)

moreover assume $(c_2, s') \mapsto\{cfs'\} (c', s_1)$

ultimately show thesis

using $B \ [\text{of} B C Z s' cfs' c' s_1 cfs_2 c'' s_2]$ and $D$ and $F$ by simp

qed

qed

lemma ctyping2-correct-aux-if:

assumes

$A$: \ \ \ \ $\forall U' B C s c' c'' s_1 s_2 cfs_1 cfs_2$. $\exists r \in B_1, s = r (\subseteq \text{state} \cap X) \Rightarrow$

$(c_1, s) \mapsto\{cfs_1\} (c', s_1) \Rightarrow (c', s_1) \mapsto\{cfs_2\} (c'', s_2) \Rightarrow$

$(\forall t_1. \exists c'_2, t_2. \forall x. (s_1 = t_1 (\subseteq \text{sources-aux} (flow cfs_2) s_1 x) \Rightarrow$

$(c', t_1) \Rightarrow (c'_2, t_2) \land (c'' = SKIP) = (c'_2 = SKIP)) \land$

$(s_1 = t_1 (\subseteq \text{sources} (flow cfs_2) s_1 x) \Rightarrow s_2 x = t_2 x)) \land$

$(\forall x. (\exists s \in \text{Univ} A X. \exists y \in \text{bvars} b. \not\exists s. \text{dom} y \Rightarrow \text{dom} x) \Rightarrow$

no-upd (flow cfs_2) x) \land$

$(\exists p \in U. \text{case} p \of (B, W) \Rightarrow$

$s \in B. \exists y \in W. \not\exists s. \text{dom} y \Rightarrow \text{dom} x) \Rightarrow$

no-upd (flow cfs_2) x))$ and

$B$: $\forall U' B C s c' c'' s_1 s_2 cfs_1 cfs_2$. $\exists r \in B_2, s = r (\subseteq \text{state} \cap X) \Rightarrow$

$(c_2, s) \mapsto\{cfs_1\} (c', s_1) \Rightarrow (c', s_1) \mapsto\{cfs_2\} (c'', s_2) \Rightarrow$

$(\forall t_1. \exists c'_2, t_2. \forall x. (s_1 = t_1 (\subseteq \text{sources-aux} (flow cfs_2) s_1 x) \Rightarrow$

$(c', t_1) \Rightarrow (c'_2, t_2) \land (c'' = SKIP) = (c'_2 = SKIP)) \land$

$(s_1 = t_1 (\subseteq \text{sources} (flow cfs_2) s_1 x) \Rightarrow s_2 x = t_2 x)) \land$

$(\forall x. (\exists s \in \text{Univ} A X. \exists y \in \text{bvars} b. \not\exists s. \text{dom} y \Rightarrow \text{dom} x) \Rightarrow$

no-upd (flow cfs_2) x) \land$

$(\exists p \in U. \text{case} p \of (B, W) \Rightarrow$

$s \in B. \exists y \in W. \not\exists s. \text{dom} y \Rightarrow \text{dom} x) \Rightarrow$

no-upd (flow cfs_2) x))$ and

$C$: $\mid b (\subseteq A, X) = (B_1, B_2)$ and

$D$: $\text{insert} (\text{Univ}\ A X, \text{bvars} b) U, v \mid c_1 (\subseteq B_1, X) =$

Some $(C_1, Y_1)$ and

$E$: $\text{insert} (\text{Univ}\ A X, \text{bvars} b) U, v \mid c_2 (\subseteq B_2, X) =$

90
Some \((C_2, Y_2)\) and

\[
F : (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \rightarrow \{cfs_1\} \ (c', s_1) \text{ and}
\]

\[
G: (c', s_1) \rightarrow \{cfs_2\} \ (c'', s_2) \text{ and}
\]

\[
H: r \in A \text{ and}
\]

\[
I: s = r \ (\subseteq \text{state} \cap X)
\]

shows

\[
(\forall t_1. \exists c_2 \ ' t_2. \forall x. (s_1 = t_1 (\subseteq \text{sources-aux} \ (\text{flow cfs}_2) s_1 x) \rightarrow (c', t_1) \rightarrow (c_2', t_2) \land (c'' = \text{SKIP}) = (c_2' = \text{SKIP})) \land
\]

\[
(s_1 = t_1 (\subseteq \text{sources} \ (\text{flow cfs}_2) s_1 x) \rightarrow s_2 x = t_2 x) \land
\]

\[
(\forall x. (\exists p \in U. \text{case } p \text{ of } (B, W) \Rightarrow \exists s \in B. \exists y \in W. \neg s \ (\text{dom } y \rightarrow \text{dom } x) \rightarrow \neg \text{no-upd} \ (\text{flow cfs}_2) x)
\]

proof –

let \(?U' = \text{insert } (\text{Univ}?) A X, \text{bvars } b \) \(U \)

have \(J: \forall cs \ t \ x. \ s = t (\subseteq \text{sources-aux} \ ((\text{bvars } b) \ # cs) s x) \rightarrow \text{bval } b \ s \neq \text{bval } b \ t \rightarrow \neg \text{Univ}? A X: \text{dom } \text{bvars } b \rightarrow \{\text{dom } x\}
\)

proof (clarify del: notI)

fix cs t x

assume \(s = t (\subseteq \text{sources-aux} \ ((\text{bvars } b) \ # cs) s x)\)

moreover assume \(\text{bval } b \ s \neq \text{bval } b \ t\)

hence \(\neg s = t (\subseteq \text{bvars } b)\)

by (erule-tac contrapos-nn, auto dest: bvars-bval)

ultimately have \(\neg (\forall y \in \text{bvars } b. \ s \ (\text{dom } y \rightarrow \text{dom } x)) \rightarrow (\text{blast dest: sources-aux-observe-hd})\)

moreover {

fix r y

assume \(r \in A \text{ and } y \in \text{bvars } b \text{ and } \neg s \ (\text{dom } y \rightarrow \text{dom } x)\)

moreover assume \(\text{state} \subseteq X \text{ and } s = r (\subseteq \text{state} \cap X)\)

hence \(\text{interf } s = \text{interf } r\)

by (blast intro: interf-state)

ultimately have \(\exists s \in A. \exists y \in \text{bvars } b. \neg s \ (\text{dom } y \rightarrow \text{dom } x)\)

by auto

}

ultimately show \(\neg \text{Univ}? A X: \text{dom } \text{bvars } b \rightarrow \{\text{dom } x\}\)

using \(H \text{ and } I \) by (auto simp: Univ-states-if-def)

qed

have

\[
(c', s_1) = (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \lor
\]

\[
\text{bval } b \ s \land (c_1, s) \rightarrow \{\text{tl cfs}_1\} \ (c', s_1) \lor
\]

\[
\neg \text{bval } b \ s \land (c_2, s) \rightarrow \{\text{tl cfs}_1\} \ (c', s_1)
\]

using \(F\) by (blast dest: small-stepsl-if)

thus \(?\text{thesis}\)


assume \(K : (c', s_1) = (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s)\)

hence \((\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \rightarrow \{cfs_2\} (c'', s_2)\)

using \(G\) by simp

hence

\[
(c'', s_2) = (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \land
\]

\[
\text{flow cfs}_2 = [] \lor
\]

91
bval b s ∧ (c₁, s) →→{tl cfs₂} (c′′, s₂) ∧
flow cfs₂ = ⟨bvars b⟩ # flow (tl cfs₂) ∨
¬ bval b s ∧ (c₂, s) →→{tl cfs₂} (c′′, s₂) ∧
flow cfs₂ = ⟨bvars b⟩ # flow (tl cfs₂)
by (rule small-steps-l-if)

thus ?thesis

assume (c′′, s₂) = (IF b THEN c₁ ELSE c₂, s) ∧ flow cfs₂ = []
thus ?thesis
using K by auto

next
assume L: bval b s
with C and H and I have s ∈ Univ B₁ (⊆ state ∩ X)
by (erule-tac btyping2-approx [where s = s], auto)
moreover assume M: (c₁, s) →→{tl cfs₂} (c′′, s₂)
moreover from this have N: s₂ = run-flow (flow (tl cfs₂)) s
by (rule small-steps-run-flow)

ultimately have O:
∀ t₁, t₂. s ∈ sources-aux (flow (tl cfs₂)) s x
(c₁, t₁) →→ (c₂′, t₂) ∧ (c′′ = SKIP) = (c₂′ = SKIP) ∧
run-flow (flow (tl cfs₂)) s x = t₂ x
∀ x. s ∈ sources-aux (flow (tl cfs₂)) s x
(c₁, t₁) →→ (c₂′, t₂) ∧ (c′′ = SKIP) = (c₂′ = SKIP) ∧
run-flow (flow (tl cfs₂)) s x = t₂ x
∀ x. s ∈ sources-aux (flow (tl cfs₂)) s x
(c₁, t₁) →→ (c₂′, t₂) ∧ (c′′ = SKIP) = (c₂′ = SKIP) ∧
run-flow (flow (tl cfs₂)) s x = t₂ x

---

Proof (cases bval b t₁)
case True
hence P: (IF b THEN c₁ ELSE c₂, t₁) → (c₁, t₁)

obtain c₂′ and t₂ where Q: ∀ x.
(s = t₁ (⊆ sources-aux (flow (tl cfs₂)) s x) →
(c₁, t₁) →→ (c₂′, t₂) ∧ (c′′ = SKIP) = (c₂′ = SKIP) ∧
run-flow (flow (tl cfs₂)) s x = t₂ x)
using O by blast
fix x
assume s = t₁
(⊆ sources-aux ((bvars b) # flow (tl cfs₂)) s x)
moreover have sources-aux (flow (tl cfs₂)) s x ⊆
   sources-aux ((bvars b) # flow (tl cfs₂)) s x
by (rule sources-aux-observe-tl)
ultimately have (IF b THEN c₁ ELSE c₂, t₁) →∗ (c₂', t₂) ∧
   (c'' = SKIP) = (c₂' = SKIP)
using P and Q by (blast intro: star-trans)
}
moreover {
  fix x
  assume s = t₁
  (⊆ sources ((bvars b) # flow (tl cfs₂)) s x)
  moreover have sources (flow (tl cfs₂)) s x ⊆
     sources ((bvars b) # flow (tl cfs₂)) s x
by (rule sources-observe-tl)
ultimately have run-flow (flow (tl cfs₂)) s x = t₂ x
  using Q by blast
}
ultimately show ?thesis
  by auto
next
assume P: ¬ bval b t₁
show ?thesis
proof (cases ∃ x. s = t₁
(⊆ sources-aux ((bvars b) # flow (tl cfs₂)) s x))
from P have (IF b THEN c₁ ELSE c₂, t₁) → (c₂, t₁) ..
moreover assume ∃ x. s = t₁
(⊆ sources-aux ((bvars b) # flow (tl cfs₂)) s x)
hence ∃ x. ¬ Univ? A X: dom ' bvars b ↦ {dom x}
  using J and L and P by blast
then obtain t₂ where Q: (c₂, t₁) ⇒ t₂
  using E by (blast dest: ctyping2-term)
hence (c₂, t₁) →* (SKIP, t₂)
  by (simp add: big-iff-small)
ultimately have
R: (IF b THEN c₁ ELSE c₂, t₁) →* (SKIP, t₂)
  by (blast intro: star-trans)
show ?thesis
proof (cases c'' = SKIP)
  case True
  show ?thesis
proof (rule exI [of - SKIP], rule exI [of - t₂])
  { have (IF b THEN c₁ ELSE c₂, t₁) →* (SKIP, t₂) ∧
        (c'' = SKIP) = (SKIP = SKIP)
      using R and True by simp
  }
  {
moreover
{
fix $x$
assume $S$: $s = t_1$
($\subseteq \text{sources} ((\text{bvars} b) \# \text{flow} (tl cfs_2)) \ s \ x$)
moreover have
\text{sources-aux} ((\text{bvars} b) \# \text{flow} (tl cfs_2)) \ s \ x \subseteq
\text{sources} ((\text{bvars} b) \# \text{flow} (tl cfs_2)) \ s \ x
by (\text{rule sources-aux-sources})
ultimately have $s = t_1$
($\subseteq \text{sources-aux} ((\text{bvars} b) \# \text{flow} (tl cfs_2)) \ s \ x$)
by blast
hence $T$: $\neg \text{Univ?} \ A \ X: \text{dom} \ bvars b \hookrightarrow \{ \text{dom} \ x \}$
using $J$ and $L$ and $P$ by blast
hence $U$: $\text{no-upd} ((\text{bvars} b) \# \text{flow} (tl cfs_2)) \ x$
using $O$ by simp
hence $\text{run-flow} (\text{flow} (tl cfs_2)) \ s \ x = t_2 \ x$
by (simp add: no-upd-run-flow)
also from $S$ and $U$ have $\ldots = t_1 \ x$
by (blast dest: no-upd-sources)
also from $E$ and $Q$ and $T$ have $\ldots = t_2 \ x$
by (drule-tac typing2-confine, auto)
finally have $\text{run-flow} (\text{flow} (tl cfs_2)) \ s \ x = t_2 \ x$.
}
ultimately show $\forall x$.
\begin{align*}
(s = t_1) \\
(\subseteq \text{sources-aux} ((\text{bvars} b) \# \text{flow} (tl cfs_2)) \ s \ x) \longrightarrow
(IF b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2, \ t_1) \longrightarrow (\text{SKIP}, \ t_2) \land \\
(c'' = \text{SKIP}) = (\text{SKIP} = \text{SKIP}) \land \\
(s = t_1) \\
(\subseteq \text{sources} ((\text{bvars} b) \# \text{flow} (tl cfs_2)) \ s \ x) \longrightarrow
\text{run-flow} (\text{flow} (tl cfs_2)) \ s \ x = t_2 \ x) \\
by \text{blast}
\end{align*}
qed
next
\begin{proof}
\text{case False}
\show \text{thesis}
\end{proof}
moreover
fix $x$
assume $S$: $s = t_1$
($\subseteq \text{sources} ((\text{bvars} b) \# \text{flow} (tl cfs_2)) \ s \ x$)
moreover have
ultimately have \( s = t_1 \)

\[ (\subseteq \text{sources-aux} ((\text{bvars } b) \# \text{flow } (\text{tl cfs}_2)) \mathbin{s} x) \] 
by blast

hence \( \neg \text{Univ? } A X \colon \text{dom } \text{bvars } b \hookrightarrow \{ \text{dom } x \} \)
using \( J \) and \( L \) and \( P \) by blast

hence \( \text{run-flow } (\text{flow } (\text{tl cfs}_2)) \mathbin{s} x = s x \)
by (simp add: no-upd-run-flow)
also have \ldots = t_1 x
using \( S \) and \( T \) by (blast dest: no-upd-sources)

finally have \( \text{run-flow } (\text{flow } (\text{tl cfs}_2)) \mathbin{s} x = t_1 x \).

ultimately show \( \forall x . (s = t_1 \subseteq \text{sources-aux} ((\text{bvars } b) \# \text{flow } (\text{tl cfs}_2)) \mathbin{s} x) \rightarrow (IF b \text{ THEN } c_1 \text{ ELSE } c_2, t_1) \rightarrow s \)

\( (IF b \text{ THEN } c_1 \text{ ELSE } c_2 \text{, } t_1) \land \)

\( (c'' = \text{SKIP}) = (IF b \text{ THEN } c_1 \text{ ELSE } c_2 = \text{SKIP}) \) \land

\( (s = t_1 \subseteq \text{sources } ((\text{bvars } b) \# \text{flow } (\text{tl cfs}_2)) \mathbin{s} x) \rightarrow\)

\( \text{run-flow } (\text{flow } (\text{tl cfs}_2)) \mathbin{s} x = t_1 x \)
by blast

\( \text{qed} \)
\( \text{qed} \)
\( \text{blast} \)
\( \text{qed} \)

ultimately show \( \forall x . (s = t_1 \subseteq \text{sources-aux} ((\text{bvars } b) \# \text{flow } (\text{tl cfs}_2)) \mathbin{s} x) \rightarrow (c_2, s) \rightarrow^* \{ \text{tl cfs}_2 \} \mathbin{(c''', s_2)} \)

moreover assume \( M : (c_2, s) \rightarrow^* \{ \text{tl cfs}_2 \} \mathbin{(c''', s_2)} \)

moreover from this have \( N : s_2 = \text{run-flow } (\text{flow } (\text{tl cfs}_2)) \mathbin{s} \)
by (rule small-steps1-run-flow)

ultimately have \( O : \)

\( (\forall t_1, \exists c_2', t_2. \forall x. (s = t_1 \subseteq \text{sources-aux} (\text{flow } (\text{tl cfs}_2)) \mathbin{s} x) \rightarrow (c_2, t_1) \rightarrow^* (c_2', t_2) \land (c'' = \text{SKIP}) = (c_2' = \text{SKIP}) \) \land

\( (s = t_1 \subseteq \text{sources } (\text{flow } (\text{tl cfs}_2)) \mathbin{s} x) \rightarrow \)

\( \text{run-flow } (\text{flow } (\text{tl cfs}_2)) \mathbin{s} x = t_2 x ) \) \land

\( (\forall x. ((\exists s \in \text{Univ? } A X . \exists y \in \text{bvars } b. \neg s : \text{dom } y \hookrightarrow \text{dom } x) \rightarrow \) 
no-upd } (\text{flow } (\text{tl cfs}_2)) ) x) \) \land

95
\[(\exists p \in U. \text{case p of (B, W)} \Rightarrow \\
\exists s \in B. \exists y \in W. \neg s : \text{dom y} \leadsto \text{dom x}) \rightarrow \\
\text{no-upd (flow (tl cfs₂)) x})\]

using \text{B [of ?U' B₁ C₂ s [] c₂ s tl cfs₂ c'' s₂] by simp}

moreover assume \text{flow cfs₂ = (bvars b) \# flow (tl cfs₂)}

moreover \{ 
  fix \text{t₁}
  have \text{∃c₂', t₂, \forall x.}
  \text{(s = t₁ (\subseteq \text{sources-aux ((bvars b) \# flow (tl cfs₂)) s x}) \rightarrow \\
  (IF b \text{ THEN } c₁ \text{ ELSE } c₂, t₁) \rightarrow* (c₂', t₂) \land \\
  (c'' = \text{SKIP}) = (c₂' = \text{SKIP}) \land \\
  (s = t₁ (\subseteq \text{sources ((bvars b) \# flow (tl cfs₂)) s x}) \rightarrow \\
  \text{run-flow (flow (tl cfs₂)) s x = t₂ x})

  proof (cases \neg \text{bval b t₁}) 
  
  case True
  hence \text{P: (IF b \text{ THEN } c₁ \text{ ELSE } c₂, t₁) \rightarrow (c₂, t₁) ..}
  obtain \text{c₂' and t₂ where Q: \forall x.}
  \text{(s = t₁ (\subseteq \text{sources-aux (flow (tl cfs₂)) s x}) \rightarrow \\
  (c₂, t₁) \rightarrow* (c₂', t₂) \land (c'' = \text{SKIP}) = (c₂' = \text{SKIP}) \land \\
  (s = t₁ (\subseteq \text{sources (flow (tl cfs₂)) s x}) \rightarrow \\
  \text{run-flow (flow (tl cfs₂)) s x = t₂ x})

  using \text{O by blast}
  \}

  fix \text{x}
  assume \text{s = t₁}
  moreover have \text{\subseteq \text{sources-aux ((bvars b) \# flow (tl cfs₂)) s x})}
  \text{by (rule sources-aux-observe-tl)}
  ultimately have \text{(IF b \text{ THEN } c₁ \text{ ELSE } c₂, t₁) \rightarrow* (c₂', t₂) \land \\
  (c'' = \text{SKIP}) = (c₂' = \text{SKIP})}

  using \text{P and Q by (blast intro: star-trans)}
  \}

moreover \{
  fix \text{x}
  assume \text{s = t₁}
  moreover have \text{\subseteq \text{sources ((bvars b) \# flow (tl cfs₂)) s x})}
  \text{by (rule sources-observe-tl)}
  ultimately have \text{run-flow (flow (tl cfs₂)) s x = t₂ x}

  using \text{Q by blast}
  \}

ultimately show \text{?thesis}
  by \text{auto}

next
  case False
  hence \text{P: \text{bval b t₁}}
  by \text{simp}

96
show ?thesis

proof (cases ∃ x. s = t₁)

(⊆ sources-aux ((lev b) # flow (tl cfs₂)) s x)

from P have (IF b THEN c₁ ELSE c₂, t₁) → (c₁, t₁) ..

moreover assume ∃ x. s = t₁

(⊆ sources-aux ((lev b) # flow (tl cfs₂)) s x)

hence ∃ x. ¬ Univ? A X: dom ' lev b → {dom x}

using J and L and P by blast

then obtain t₂ where Q: (c₁, t₁) ⇒ t₂

using D by (blast dest: ctyping2-term)

hence (c₁, t₁) →* (SKIP, t₂)

by (simp add: big iff small)

ultimately have

R: (IF b THEN c₁ ELSE c₂, t₁) →* (SKIP, t₂)

by (blast intro: star-trans)

show ?thesis

proof (cases c'' = SKIP)

case True

show ?thesis

proof (rule exI [of - SKIP], rule exI [of - t₂])

{ have (IF b THEN c₁ ELSE c₂, t₁) →* (SKIP, t₂) ∧

(c'' = SKIP) = (SKIP = SKIP)

using R and True by simp

}

moreover {

fix x

assume S: s = t₁

(⊆ sources ((lev b) # flow (tl cfs₂)) s x)

moreover have

sources-aux ((lev b) # flow (tl cfs₂)) s x ⊆

sources ((lev b) # flow (tl cfs₂)) s x

by (rule sources-aux-sources)

ultimately have s = t₁

(⊆ sources-aux ((lev b) # flow (tl cfs₂)) s x)

by blast

hence T: ¬ Univ? A X: dom ' lev b → {dom x}

using J and L and P by blast

hence U: no-upd ((lev b) # flow (tl cfs₂)) x

using O by simp

hence run-flow (flow (tl cfs₂)) s x = s x

by (simp add: no-upd-run-flow)

also from S and U have ... = t₁ x

by (blast dest: no-upd-sources)

also from D and Q and T have ... = t₂ x

by (drule_tac ctyping2-confine, auto)

finally have run-flow (flow (tl cfs₂)) s x = t₂ x .

}

ultimately show ∀ x.
(s = t₁)
(\subseteq \text{sources-\text{aux}} ((\text{bvars } b) \not\# \text{flow } (\text{tl } \text{cfs}_2)) s x) \rightarrow
(IF b \text{ THEN } c_1 \text{ ELSE } c_2, t_1) \rightarrow^* (\text{SKIP}, t_2) \land
(c'' = \text{SKIP}) = (IF b \text{ THEN } c_1 \text{ ELSE } c_2 = \text{SKIP}) \land

(s = t₁)
(\subseteq \text{sources } ((\text{bvars } b) \not\# \text{flow } (\text{tl } \text{cfs}_2)) s x) \rightarrow
\text{run-flow } (\text{flow } (\text{tl } \text{cfs}_2)) s x = t₂ x)
by blast

qed

next
case False
show ?thesis
proof (rule exI [of - IF b THEN c₁ ELSE c₂],
rule exI [of - t₁])
{
  have (IF b THEN c₁ ELSE c₂, t₁) \rightarrow^*
    (IF b THEN c₁ ELSE c₂, t₁) \land
    (c'' = \text{SKIP}) = (IF b THEN c₁ ELSE c₂ = \text{SKIP})
    using False by simp
}
moreover {
  fix x
  assume S: s = t₁
    (\subseteq \text{sources-\text{aux}} ((\text{bvars } b) \not\# \text{flow } (\text{tl } \text{cfs}_2)) s x)
moreover have
sources-\text{aux } ((\text{bvars } b) \not\# \text{flow } (\text{tl } \text{cfs}_2)) s x \subseteq
sources ((\text{bvars } b) \not\# \text{flow } (\text{tl } \text{cfs}_2)) s x
by (rule sources-\text{aux-\text{sources}})
ultimately have s = t₁
(\subseteq \text{sources-\text{aux}} ((\text{bvars } b) \not\# \text{flow } (\text{tl } \text{cfs}_2)) s x)
by blast
hence ~ Univ? A X: dom ' bvars b \rightarrow \{\text{dom } x\}
using J and L and P by blast
hence T: no-upd ((\text{bvars } b) \not\# \text{flow } (\text{tl } \text{cfs}_2)) x
using O by simp
hence run-flow (\text{flow } (\text{tl } \text{cfs}_2)) s x = s x
by (simp add: no-upd-run-flow)
also have \ldots = t₁ x
using S and T by (blast dest: no-upd-\text{sources})
finally have run-flow (\text{flow } (\text{tl } \text{cfs}_2)) s x = t₁ x .
}
ultimately show \forall x.
(s = t₁)
(\subseteq \text{sources-\text{aux}} ((\text{bvars } b) \not\# \text{flow } (\text{tl } \text{cfs}_2)) s x) \rightarrow
(IF b \text{ THEN } c₁ \text{ ELSE } c₂, t₁) \rightarrow^*
(IF b \text{ THEN } c₁ \text{ ELSE } c₂, t₁) \land
(c'' = \text{SKIP}) = (IF b \text{ THEN } c₁ \text{ ELSE } c₂ = \text{SKIP}) \land

(s = t₁)
(\subseteq \text{sources } ((\text{bvars } b) \not\# \text{flow } (\text{tl } \text{cfs}_2)) s x) \rightarrow
run-flow (flow (tl cfs2)) s x = t1 x)
    by blast
    qed
qed
qed
qed
}
ultimately show ?thesis
using K and N by auto
qed
next
assume bval b s and (c1, s) -->{} tl cfs1 (c', s1)
moreover from this and C and H and I have s ∈ Univ B1 (⊆ state ∩ X)
by (drule-tac btyping2-approx [where s = s], auto)
ultimately show ?thesis
using A [of ?U' B1 C1 s tl cfs1 c' s1 cfs2 c'' s2] and G by simp
next
assume ¬ bval b s and (c2, s) -->{} tl cfs1 (c', s1)
moreover from this and C and H and I have s ∈ Univ B2 (⊆ state ∩ X)
by (drule-tac btyping2-approx [where s = s], auto)
ultimately show ?thesis
using B [of ?U' B1 C2 s tl cfs1 c' s1 cfs2 c'' s2] and G by simp
qed
qed

lemma ctyping2-correct-aux-while:
assumes
A: ∃ B C' B' D' s c1 c2 s1 s2 cfs1 cfs2.
    B = B1 ⊢ C' = C ⊢ B' = B1' ⊢
    (∀ s ∈ Univ′ A X ∪ Univ′ C Y. ∀ x ∈ bvars b. All (interf s (dom x))) ∧
    (∀ p ∈ U. case p of (B, W) ⇒ ∀ s ∈ B. ∀ x ∈ W. All (interf s (dom x))) ⇒
    D = D' ⊢ ∃ r ∈ B1. s = r (⊆ state ∩ X) ⊢
    (c, s) -->*{} cfs1 (c1, s1) ⊢ (c1, s1) -->*{} cfs2 (c2, s2) ⊢
    ∀ t1, cfs2 t2, ∀ x.
    (s1 = t1 (≤ sources-aux (flow cfs2) s1 x) -->
    (c1, t1) -->*{} (c2', t2) ∧ (c2 = SKIP) = (c2' = SKIP)) ∧
    (s1 = t1 (≤ sources (flow cfs2) s1 x) --> s2 x = t2 x) and
B: ∃ B C' B' D'' s c1 c2 s1 s2 cfs1 cfs2.
    B = B1 ⊢ C' = C ⊢ B' = B1' ⊢
    (∀ s ∈ Univ′ A X ∪ Univ′ C Y. ∀ x ∈ bvars b. All (interf s (dom x))) ∧
    (∀ p ∈ U. case p of (B, W) ⇒ ∀ s ∈ B. ∀ x ∈ W. All (interf s (dom x))) ⇒
    D' = D'' ⊢ ∃ r ∈ B1'. s = r (⊆ state ∩ Y) ⊢
    (c, s) -->*{} cfs1 (c1, s1) ⊢ (c1, s1) -->*{} cfs2 (c2, s2) ⊢
    ∀ t1, cfs2 t2, ∀ x.
    (s1 = t1 (≤ sources-aux (flow cfs2) s1 x) -->
    (c1, t1) -->*{} (c2', t2) ∧ (c2 = SKIP) = (c2' = SKIP)) ∧
    (s1 = t1 (≤ sources (flow cfs2) s1 x) --> s2 x = t2 x) and
C: if (∀ s ∈ Univ′ A X ∪ Univ′ C Y. ∀ x ∈ bvars b. All (interf s (dom x))) ∧
    (∀ p ∈ U. ∀ B W. p = (B, W) ⇒ (∀ s ∈ B. ∀ x ∈ W. All (interf s (dom x))))
then Some (B₂ ∪ B₂¹, Univ? B₂ X ∩ Y) else None) = Some (B, W) and

D: |= b (⊆ A, X) = (B₁, B₂) and
E: ⊨ c (⊆ B₁, X) = (C, Y) and
F: |= b (⊆ C, Y) = (B₁¹, B₂²) and
G: ({}), False) |= c (⊆ B₁, X) = Some (D, Z) and
H: ({}), False) |= c (⊆ B₁¹, Y) = Some (D¹, Z¹)

shows

[(WHILE b DO c, s) →∗{cfs₁} (c₁, s₁);
  (c₁, s₁) →∗{cfs₂} (c₂, s₂);
  s ∈ Univ A (⊆ state ∩ X) ∪ Univ C (⊆ state ∩ Y)] ⇒
(∀ t₁, ∃ c₂¹ t₂. ∀ x. (s₁ = t₁ (⊆ sources-aux (flow cfs₂) s₁ x) →
  (c₁, t₁) →∗ (c₂¹, t₂) ∧ (c₂ = SKIP) = (c₂¹ = SKIP)) ∧
  (s₁ = t₁ (⊆ sources (flow cfs₂) s₁ x) → s₂ x = t₂ x)) ∧
  (∀ x. (∃ p ∈ U, case p of (B, W) ⇒
  ∃ s ∈ B. ∃ y ∈ W. s; dom y → dom x) → no-upd (flow cfs₂) x)

proof (induction cfs₁ @ cfs₂ arbitrary; cfs₁ cfs₂ s c₁ s₁ rule: length-induct)
fix cfs₁ cfs₂ s c₁ s₁
assume I: (WHILE b DO c, s) →∗{cfs₁} (c₁, s₁) and
J: (c₁, s₁) →∗{cfs₂} (c₂, s₂)
assume ∀ cfs. length cfs < length (cfs₁ @ cfs₂) →
(∀ cfs₁ cfs₂, cfs = cfs₁ @ cfs₂ →
  (∃ s, c₁ s₁. (WHILE b DO c, s) →∗{cfs₁} (c₁, s₁) →
    (c₁, s₁) →∗{cfs₂} (c₂, s₂) →
    s ∈ Univ A (⊆ state ∩ X) ∪ Univ C (⊆ state ∩ Y) →
    (∀ t₁, ∃ c₂¹ t₂. ∀ x. (s₁ = t₁ (⊆ sources-aux (flow cfs₂) s₁ x) →
      (c₁, t₁) →∗ (c₂¹, t₂) ∧ (c₂ = SKIP) = (c₂¹ = SKIP)) ∧
      (s₁ = t₁ (⊆ sources (flow cfs₂) s₁ x) → s₂ x = t₂ x)) ∧
      (∀ x. (∃ (B, W) ∈ U. ∃ s ∈ B. ∃ y ∈ W. s; dom y → dom x) →
        no-upd (flow cfs₂) x)))

note K = this [rule-format]
assume L: s ∈ Univ A (⊆ state ∩ X) ∪ Univ C (⊆ state ∩ Y)
moreover {
  fix s¹
  assume s ∈ Univ A (⊆ state ∩ X) and bval b s
  hence N: s ∈ Univ B₁ (⊆ state ∩ X)
    using D by (drule-tac btyping2-approx, auto)
  assume (c, s) ⇒ s¹
  hence s¹ ∈ Univ D (⊆ state ∩ Z)
    using G and N by (rule ctyping2-approx)
  moreover have D ⊆ C ∧ Y ⊆ Z
    using E and G by (rule ctyping1-ctyping2)
  ultimately have s¹ ∈ Univ C (⊆ state ∩ Y)
    by blast
}
moreover {
  fix s¹
}
assume \( s \in \text{Univ} \ C \ (\subseteq \text{state} \cap \ Y) \) and \( \text{bval} \ b \ s \)
hence \( N: s \in \text{Univ} \ B_1' \ (\subseteq \text{state} \cap \ Y) \)
using \( F \) by (drule-tac \( \text{btyping2-approx} \), auto)
assume \((c, s) \Rightarrow s'\)
hence \( s' \in \text{Univ} \ D' \ (\subseteq \text{state} \cap \ Z') \)
using \( H \) and \( N \) by (rule \( \text{ctyping2-approx} \))
moreover obtain \( C' \) and \( Y' \) where \( O: c \vdash c \ (\subseteq B_1', Y) = (C', Y') \)
by (cases \( \vdash c \ (\subseteq B_1', Y) \), simp)
hence \( D' \subseteq C' \land Y' \subseteq Z' \)
using \( H \) by (rule \( \text{ctyping1-ctyping2} \))
ultimately have \( P: s' \in \text{Univ} \ C' \ (\subseteq \text{state} \cap \ Y') \)
by blast
have \( c \ (\subseteq C, Y) = (C, Y) \)
using \( E \) by (rule \( \text{ctyping1-idem} \))
moreover have \( B_1' \subseteq C \)
using \( F \) by (blast dest: \( \text{btyping2-un-eq} \))
ultimately have \( C' \subseteq C \land Y' \subseteq Y' \)
by (metis \( \text{order-refl} \) \( \text{ctyping1-mono} \) \( O \))
hence \( s' \in \text{Univ} \ C \ (\subseteq \text{state} \cap \ Y) \)
using \( P \) by blast

ultimately have \( M: \)
\( \forall s'. (c, s) \Rightarrow s' \Rightarrow \text{bval} \ b \ s \Rightarrow s' \in \text{Univ} \ C \ (\subseteq \text{state} \cap \ Y) \)
by blast
have \( N: \)
\( \forall s \in \text{Univ}? \ A \ X \cup \text{Univ}? \ C \ Y. \forall x \in \text{bvars} \ b. \ \text{All} \ (\text{interf} \ s \ (\text{dom} \ x)) \land \left( \forall \ p \in \ U. \ \forall \ B \ W. \ \ p = (B, W) \Rightarrow (\forall s \in B. \ \forall x \in W. \ \text{All} \ (\text{interf} \ s \ (\text{dom} \ x))) \right) \)
using \( C \) by (simp split: if-split-asm)
hence \( \forall x \ s. \ (\exists (B, Y) \in U. \ \exists s \in B. \ \exists y \in Y. \ \text{dom} \ y \Rightarrow \text{dom} \ x) \Rightarrow \text{no-upd} \ cs \ x \)
by auto
moreover {\}
fix \( r \ t_1 \)
assume \( O: r \in A \land P: s = r \ (\subseteq \text{state} \cap \ X) \)
have \( Q: \forall x. \ \forall y \in \text{bvars} \ b. \ s \Rightarrow \text{dom} \ y \Rightarrow \text{dom} \ x \)
proof (cases \( \text{state} \subseteq X \))
\( \text{case True} \)
\( \text{with } P \) have \( \text{interf} \ s = \text{interf} \ r \)
by (blast intro: interf-state)
\( \text{with } N \) and \( O \) show \( ? \text{thesis} \)
by (erule-tac conjE, drule-tac bspec,
auto simp: univ-states-if-def)
\( \text{next} \)
\( \text{case False} \)
\( \text{with } N \) and \( O \) show \( ? \text{thesis} \)
by (erule-tac conjE, drule-tac bspec,
auto simp: univ-states-if-def)
\( \text{qed} \)
have \( (c_1, s_1) = (\text{WHILE} \ b \ DO \ c, s) \lor \)
\(\)
\[(IF \ b \ THEN \ c; \ WHILE \ b \ DO \ c \ ELSE \ SKIP, \ s) \rightarrow \{tl \ cfs_1\} \ (c_1, \ s_1)\]

using \(I\) by \(\text{blast dest: small-steps-while}\)

hence \(\exists \ c'_2 \ t_2. \ \forall x.\)

\((s_1 = t_1) \subseteq \text{sources-aux} (flow \ cfs_2) \ s_1 \ x) \rightarrow\)

\((c_1, \ t_1) \rightarrow^* (c'_2, \ t_2) \land (c_2 = \text{SKIP}) \land\)

\((s_1 = t_1) \subseteq \text{sources} (flow \ cfs_2) \ s_1 \ x) \rightarrow\)

\(s_2 \ x = t_2 \ x)\)

proof

assume \(R: \ (c_1, \ s_1) = (\text{WHILE} \ b \ DO \ c, \ s)\)

hence \(\text{(WHILE} \ b \ DO \ c, \ s) \rightarrow^* \{cfs_2\} \ (c_2, \ s_2)\)

using \(J\) by \(\text{simpl}\)

\(\therefore\)

\((c_2, \ s_2) = (\text{WHILE} \ b \ DO \ c, \ s) \land\)

\(\text{flow} \ cfs_2 = [] \lor\)

\((IF \ b \ THEN \ c; \ WHILE \ b \ DO \ c \ ELSE \ SKIP, \ s) \rightarrow^* \{tl \ cfs_2\} \ (c_2, \ s_2) \land\)

\(\text{flow} \ cfs_2 = \text{flow} (tl \ cfs_2)\)

\(\text{is} \ ?P \lor \ ?Q \land \ ?R\)

by \(\text{(rule small-steps-while)}\)

thus \(?\text{thesis}\)

proof \(\text{(rule disjE, erule-tac [2] conjE)}\)

assume \(?P\)

with \(R\) show \(?\text{thesis}\)

by \(\text{auto}\)

next

assume \(?Q\) and \(?R\)

have\(\)

\((c_2, \ s_2) = (IF \ b \ THEN \ c; \ WHILE \ b \ DO \ c \ ELSE \ SKIP, \ s) \land\)

\(\text{flow} (tl \ cfs_2) = [] \lor\)

\(\text{bval} \ b \ s \land (c; \ WHILE \ b \ DO \ c, \ s) \rightarrow^* \{tl2 \ cfs_2\} \ (c_2, \ s_2) \land\)

\(\text{flow} (tl \ cfs_2) = \{\text{bvars} \ b\} \not\# \text{flow} (tl2 \ cfs_2) \lor\)

\(\neg \text{bval} \ b \ s \land (\text{SKIP}, \ s) \rightarrow^* \{tl2 \ cfs_2\} \ (c_2, \ s_2) \land\)

\(\text{flow} (tl \ cfs_2) = \{\text{bvars} \ b\} \not\# \text{flow} (tl2 \ cfs_2)\)

using \(?Q\) by \(\text{(rule small-steps-if)}\)

thus \(?\text{thesis}\)


assume \((c_2, \ s_2) = (IF \ b \ THEN \ c; \ WHILE \ b \ DO \ c \ ELSE \ SKIP, \ s) \land\)

\(\text{flow} (tl \ cfs_2) = []\)

with \(R\) and \(?R\) show \(?\text{thesis}\)

by \(\text{auto}\)

next

assume \(S: \ \text{bval} \ b \ s\)

with \(D\) and \(?Q\) and \(?R\) have \(T: \ s \in \text{Univ} \ B_1 \ (\subset \ \text{state} \land X)\)

by \(\text{(drule-tac btyping2-approx [where \ s = s], auto)}\)

assume \(U: \ (c; \ WHILE \ b \ DO \ c, \ s) \rightarrow^* \{tl2 \ cfs_2\} \ (c_2, \ s_2)\)

hence\(\)

\((\exists \ c' \ cfs, \ c_2 = c'; \ WHILE \ b \ DO \ c \land\)

\((c, \ s) \rightarrow^* \{cfs\} \ (c', \ s') \land\)

\(\text{flow} (tl2 \ cfs_2) = \text{flow} cfs) \lor\)

\((\exists \ s' \ cfs' \ cfs', \ \text{length} \ cfs' < \text{length} \ (tl2 \ cfs_2) \land\)

\((c, \ s) \rightarrow^* \{cfs\} \ (\text{SKIP}, \ s') \land\)

102
moreover assume \( \text{flow} \ (t_2 \ cfs_2) = \text{flow} \ cfs \ @ \text{flow} \ cfs' \) by (rule small-stepsl-seq)

moreover have \( s_2 = \text{run-flow} \ (t_2 \ cfs_2) \ s \) using \( U \) by (rule small-stepsl-run-flow)

moreover \{ 

fix \( c' \) cfs

assume \( (c, s) \rightarrow* \{cfs\} \ (c', \text{run-flow} \ (\text{flow} \ cfs) \ s) \)

then obtain \( c_2' \) and \( t_2 \) where \( V: \forall x. \)

\((s = t_1 (\subseteq \text{sources-aux} \ (\text{flow} \ cfs) \ s) \ x) \rightarrow (c, t_1) \rightarrow* (c_2', t_2) \land (c' = \text{SKIP}) \land (s = t_1 (\subseteq \text{sources} \ (\text{flow} \ cfs) \ s) \ x) \rightarrow \text{run-flow} \ (\text{flow} \ cfs) \ s \ x = t_2 \ x) \)

using \( A \) \{ of \( B_1 \ C \ B_1' \ D s \ [\ c \ s \ cfs \ c' \) run-flow (flow cfs) s] and \( N \) and \( T \) by force \}

\{ 

fix \( x \)

assume \( W: s = t_1 (\subseteq \text{sources-aux} \ ((\text{bvars} \ b) \ # \ (\text{flow} \ cfs) \ s) \ x) \)

moreover have \( \text{sources-aux} \ (\text{flow} \ cfs) \ s \ x \subseteq \text{sources-aux} \ ((\text{bvars} \ b) \ # (\text{flow} \ cfs) \ s) \ x \) by (rule sources-aux-observe-ll)

ultimately have \( (c, t_1) \rightarrow* (c_2', t_2) \)

using \( V \) by blast

hence \( (c; \ \text{WHILE} \ b \ \text{DO} \ c, t_1) \rightarrow* (c_2';, \ \text{WHILE} \ b \ \text{DO} \ c, t_2) \)

by (rule star-seq2)

moreover have \( s = t_1 (\subseteq \text{bvars} \ b) \)

using \( Q \) and \( W \) by (blast dest: sources-aux-observe-hd)

hence \( \text{bval} b \ t_1 \)

using \( S \) by (blast dest: bvars-bval)

hence \( (\text{WHILE} \ b \ \text{DO} \ c, t_1) \rightarrow* (c; \ \text{WHILE} \ b \ \text{DO} \ c, t_1) \)

by (blast intro: star-trans)

ultimately have \( (\text{WHILE} \ b \ \text{DO} \ c, t_1) \rightarrow* (c_2';, \ \text{WHILE} \ b \ \text{DO} \ c, t_2) \land c_2' \neq \text{SKIP} \)

by (blast intro: star-trans)

\}

moreover \{ 

fix \( x \)

assume \( s = t_1 (\subseteq \text{sources} \ ((\text{bvars} \ b) \ # (\text{flow} \ cfs) \ s) \ x) \)

moreover have \( \text{sources} \ (\text{flow} \ cfs) \ s \ x \subseteq \text{sources} \ ((\text{bvars} \ b) \ # (\text{flow} \ cfs) \ s) \ x \) by (rule sources-observe-ll)

ultimately have \( \text{run-flow} \ (\text{flow} \ cfs) \ s \ x = t_2 \ x \)

using \( V \) by blast

\}

ultimately have \( \exists c_2' \ t_2. \ \forall x. \)

\((s = t_1 (\subseteq \text{sources-aux} \ ((\text{bvars} \ b) \ # (\text{flow} \ cfs) \ s) \ x) \rightarrow (\text{WHILE} \ b \ \text{DO} \ c, t_1) \rightarrow* (c_2', t_2) \land c_2' \neq \text{SKIP}) \land (s = t_1 (\subseteq \text{sources} \ ((\text{bvars} \ b) \ # (\text{flow} \ cfs) \ s) \ x) \rightarrow \)

103
run-flow (flow cfs) s x = t2 x
by blast
}
moreover {
fix s' cfs cfs'
assume
V: length cfs' < length cfs2 - Suc (Suc 0) and
W: (c, s) ----> {cfs} (SKIP, s') and
X: (WHILE b DO c, s') ----> {cfs'}
(c2, run-flow (flow cfs')) (run-flow (flow cfs) s)
then obtain c2' and t2 where ∀ x.
(s = t1 (⊆ sources-aux (flow cfs) s x) ---->
(c, t1) ----> (c2', t2) ∧ (SKIP = SKIP) = (c2' = SKIP)) ∧
(s = t1 (⊆ sources (flow cfs) s x) ----> s' x = t2 x)
using A [of B1 C B1' D s [] c s cfs SKIP s']
and N and T by force
moreover have Y: s' = run-flow (flow cfs) s
using W by (rule small-steps-run-flow)
ultimately have Z: ∀ x.
(s = t1 (⊆ sources-aux (flow cfs) s x) ---->
(c, t1) ----> (SKIP, t3)) ∧
(s = t1 (⊆ sources (flow cfs) s x) ---->
run-flow (flow cfs) s x = t2 x)
by blast
assume s2 = run-flow (flow cfs') (run-flow (flow cfs) s)
moreover have (c, s) ⇒ s'
using W by (auto dest: small-steps-steps simp: big-iff-small)
hence s' ∈ Univ C (⊆ state ∩ Y)
using M and S by blast
ultimately obtain c3' and t3 where AA: ∀ x.
(run-flow (flow cfs) s = t2
(⊆ sources-aux (flow cfs') (run-flow (flow cfs) s) x) ---->
(WHILE b DO c, t2) ----> (c3', t3) ∧
(c2 = SKIP) = (c3' = SKIP)) ∧
(run-flow (flow cfs) s = t2
(⊆ sources (flow cfs) (run-flow (flow cfs) s) x) ---->
run-flow (flow cfs') (run-flow (flow cfs) s) x = t3 x)
using K [of cfs' [] cfs' s' WHILE b DO c s']
and V and X and Y by force
{
fix x
assume AB: s = t1
(⊆ sources-aux ((bvars b) # flow cfs @ flow cfs') s x)
moreover have sources-aux (flow cfs) s x ⊆
sources-aux (flow cfs @ flow cfs') s x
by (rule sources-aux-append)
moreover have AC: sources-aux (flow cfs @ flow cfs') s x ⊆
sources-aux ((bvars b) # flow cfs @ flow cfs') s x
by (rule sources-aux-observe-II)
ultimately have \((c, t_1) \rightarrow^* (\text{SKIP}, t_2)\)
using Z by blast

hence \((c;\ WHILE\ b\ DO\ c, t_1) \rightarrow^* (\text{SKIP};\ WHILE\ b\ DO\ c, t_2)\)
by (rule star-seq2)
moreover have \(s = t_1\ (\subseteq \text{bvars}\ b)\)
using Q and AB by (blast dest: sources-aux-observe-hd)
hence \(bval\ b\ t_1\)
using S by (blast dest: bvars-bval)
hence \((WHILE\ b\ DO\ c, t_1) \rightarrow^* (WHILE\ b\ DO\ c, t_2)\)
by (blast intro: star-trans)
moreover have \(\text{run-flow}\ (\text{flow}\ cfs)\ s = t_2\)
\((\subseteq \text{sources-aux}\ (\text{flow}\ cfs')\ (\text{run-flow}\ (\text{flow}\ cfs)\ s)\ x)\)
proof
fix y
assume \(y \in \text{sources-aux}\ (\text{flow}\ cfs')\)
\((\text{run-flow}\ (\text{flow}\ cfs)\ s)\ x\)
hence \(\text{sources}\ (\text{flow}\ cfs)\ s\ y \subseteq\)
\(\text{sources-aux}\ (\text{flow}\ cfs @ \text{flow}\ cfs')\ s\ x\)
by (rule sources-aux-member)
hence \(\text{sources}\ (\text{flow}\ cfs)\ s\ y \subseteq\)
\(\text{sources-aux}\ ((\text{bvars}\ b) \# \text{flow}\ cfs @ \text{flow}\ cfs')\ s\ x\)
using AC by simp
thus \(\text{run-flow}\ (\text{flow}\ cfs)\ s\ y = t_2\ y\)
using Z and AB by blast
qed

ultimately have \((WHILE\ b\ DO\ c, t_1) \rightarrow^* (c_3', t_3)\ \land\)
\((c_2 = \text{SKIP}) = (c_3' = \text{SKIP})\)
using AA by simp
moreover have \((WHILE\ b\ DO\ c, t_1) \rightarrow^* (c_3', t_3)\ \land\)
\((c_2 = \text{SKIP}) = (c_3' = \text{SKIP})\)
by (blast intro: star-trans)

moreover {
    fix x
    assume AB: \(s = t_1\)
    \((\subseteq \text{sources}\ ((\text{bvars}\ b) \# \text{flow}\ cfs @ \text{flow}\ cfs')\ s)\ x)\)
    have \(\text{run-flow}\ (\text{flow}\ cfs)\ s = t_2\)
    \((\subseteq \text{sources}\ (\text{flow}\ cfs')\ (\text{run-flow}\ (\text{flow}\ cfs)\ s)\ x)\)
proof
fix y
assume \(y \in \text{sources}\ (\text{flow}\ cfs')\)
\((\text{run-flow}\ (\text{flow}\ cfs)\ s)\ x\)
hence \(\text{sources}\ (\text{flow}\ cfs)\ s\ y \subseteq\)
\(\text{sources}\ (\text{flow}\ cfs @ \text{flow}\ cfs')\ s\ x\)
by (rule sources-member)
moreover have \(\text{sources}\ (\text{flow}\ cfs @ \text{flow}\ cfs')\ s\ x \subseteq\)
\(\text{sources}\ ((\text{bvars}\ b) \# \text{flow}\ cfs @ \text{flow}\ cfs')\ s\ x\)

ultimately have \( \exists c_3', t_3, \forall x. (s = t_1) \) 
\( (s \subseteq \text{sources-aux} \ (\{bvars b\} \ # \ \text{flow cfs} \ @ \ \text{flow cfs}') \ s x) \rightarrow \)
\( (\text{WHILE } b \text{ DO } c, t_1) \rightarrow^* (c_3', t_3) \land \)
\( (c_2 = \text{SKIP}) = (c_3' = \text{SKIP}) \land \)
\( (s = t_1) \) 
ultimately have \( \exists c_3', t_3, \forall x. (s = t_1) \) 
\( (s \subseteq \text{sources-aux} \ (\{bvars b\} \ # \ \text{flow cfs} \ @ \ \text{flow cfs}') \ s x) \rightarrow \)
\( \text{run-flow} (\text{flow cfs}') (\text{run-flow} (\text{flow cfs}) s) x = t_3 x \)
ultimately show \( \exists \theta \text{thesis} \) 
using \( R \) and \( \langle \theta R \rangle \) by (auto simp: run-flow-append)
next
assume 
\( S: \neg bval b s \) and 
\( T: \text{flow} (tl cfs_2) = (\{bvars b\} \ # \ \text{flow} (tl2 cfs_2)) \) 
moreover assume \( \text{flow} \ (\{bvars b\} \ # \ \text{flow} (tl2 cfs_2)) (c_2, s_2) \) 
hence \( U: (c_2, s_2) = (\text{SKIP}, s) \land \text{flow} (tl2 cfs_2) = [] \) 
by (rule small-steps-l-skip)
show \( \exists \theta \text{thesis} \)
proof (rule exI [of - \text{SKIP}], rule exI [of - \ t_1])
\{ 
fix \( x \)
\begin{align*}
\text{have} & \ (\text{WHILE } b \text{ DO } c, t_1) \rightarrow \\
& \ (\text{IF } b \text{ THEN } c; \text{WHILE } b \text{ DO } c \text{ ELSE SKIP}, t_1) .. \\
\text{moreover assume} & \ s = t_1 \ (s \subseteq \text{sources-aux} \ (\{bvars b\} \ s x) \) \\
\text{hence} & \ s = t_1 \ (s \subseteq \text{bvars b}) \\
\text{using} & \ Q \text{ by (blast dest: sources-aux-observe-hd)} \\
\text{hence} & \ \neg bval b t_1 \\
\text{using} & \ S \text{ by (blast dest: bvars-bval)} \\
\text{hence} & \ (\text{IF } b \text{ THEN } c; \text{WHILE } b \text{ DO } c \text{ ELSE SKIP}, t_1) \rightarrow \\
& \ (\text{SKIP}, t_1) .. \\
\text{ultimately have} & \ (\text{WHILE } b \text{ DO } c, t_1) \rightarrow^* (\text{SKIP}, t_1) \\
\text{by} & \ (\text{blast intro: star-trans}) \\
\} \\
moreover \{ 
fix \( x \)
assume \( s = t_1 \ (s \subseteq \text{sources} \ (\{bvars b\} \ s x) \) \\
\text{hence} & \ s \ x = t_1 \ x 
\} 

by (subt (asm) append-Nil [symmetric],
simp only: sources.simps, auto)
}
ultimately show ∃x.
(s₁ = t₁ (⊆ sources-aux (flow cfs₂) s₁ x) →
 (c₁, t₁) →* (SKIP, t₁) ∧ (c₂ = SKIP) = (SKIP = SKIP)) ∧
(s₁ = t₁ (⊆ sources (flow cfs₂) s₁ x) → s₂ x = t₁ x)
using R and T and U and (?R) by auto
qed
qed
qed
next
assume (IF b THEN c;; WHILE b DO c ELSE SKIP, s) →*{tl cfs₁} (c₁, s₁)
hence
(c₁, s₁) = (IF b THEN c;; WHILE b DO c ELSE SKIP, s) ∧
flow (tl cfs₁) = [] ∨
val b s ∧ (c;; WHILE b DO c, s) →*{tl cfs₁} (c₁, s₁) ∧
flow (tl cfs₁) = ⟨bvars b⟩ # flow (tl cfs₁) ∨
¬ val b s ∧ (SKIP, s) →*{tl cfs₁} (c₁, s₁) ∧
flow (tl cfs₁) = ⟨bvars b⟩ # flow (tl cfs₁)
by (rule small-steps-l-if)
thus ?thesis
proof (rule disjE, erule-tac [2] disjE, erule-tac conjE,
(erule-tac [2 − 3] conjE)+)
assume R: (c₁, s₁) = (IF b THEN c;; WHILE b DO c ELSE SKIP, s)
hence (IF b THEN c;; WHILE b DO c ELSE SKIP, s) →*{cfs₂} (c₂, s₂)
using J by simp
hence
(c₂, s₂) = (IF b THEN c;; WHILE b DO c ELSE SKIP, s) ∧
flow cfs₂ = [] ∨
val b s ∧ (c;; WHILE b DO c, s) →*{tl cfs₂} (c₂, s₂) ∧
flow cfs₂ = ⟨bvars b⟩ # flow (tl cfs₂) ∨
¬ val b s ∧ (SKIP, s) →*{tl cfs₂} (c₂, s₂) ∧
flow cfs₂ = ⟨bvars b⟩ # flow (tl cfs₂)
by (rule small-steps-l-if)
thus ?thesis
assume (c₂, s₂) = (IF b THEN c;; WHILE b DO c ELSE SKIP, s) ∧
flow cfs₂ = []
with R show ?thesis
by auto
next
assume S: val b s
with D and O and P have T: s ∈ Univ B₁ (⊆ state ∩ X)
by (erule-tac btyping2-approx [where s = s], auto)
assume U: (c;; WHILE b DO c, s) →*{tl cfs₂} (c₂, s₂)
hence
∃ c' cfs. c₂ = c';; WHILE b DO c ∧
(c, s) →*{cfs} (c', s₂) ∧
flow (tl cfs₂) = flow cfs)

(∃ s' cfs cfs'. length cfs' < length (tl cfs₂) ∧
  (c, s) →∗ {cfs} (SKIP, s') ∧
  (WHILE b DO c, s') →∗ {cfs'} (c₂, s₂) ∧

flow (tl cfs₂) = flow cfs @ flow cfs')

by (rule small-steps!-seq)

moreover assume flow cfs₂ = ⟨bvars b⟩ # flow (tl cfs₂)

moreover have s₂ = run-flow (flow (tl cfs₂)) s
using U by (rule small-steps!-run-flow)

moreover { fix c' cfs
assume (c, s) →∗ {cfs} (c', run-flow (flow cfs) s)
then obtain c₂' and t₂ where V: ∀ x.
  (s = t₁ (⊆ sources-aux (flow cfs) s x) →
   (c, t₁) →∗ (c₂', t₂) ∧ (c' = SKIP) = (c₂' = SKIP)) ∧
  (s = t₁ (⊆ sources-aux (flow cfs) s x) →
run-flow (flow cfs) s x = t₂ x)
using A [of B₁ C B₁' D s [] c s cfs c']
run-flow (flow cfs) s and N and T by force
}

{ fix x
assume W: s = t₁ (⊆ sources-aux ((bvars b) # flow cfs) s x)
moreover have sources-aux (flow cfs) s x ⊆
  sources-aux ((bvars b) # (flow cfs)) s x
by (rule sources-aux-observe-tl)
ultimately have (c, t₁) →∗ (c₂', t₂)
using V by blast
hence (c;; WHILE b DO c, t₁) →∗ (c₂';;; WHILE b DO c, t₂)
by (rule star-seq2)
moreover have s = t₁ (⊆ bvars b)
using Q and W by (blast dest: sources-aux-observe-hd)
hence bval b t₁
using S by (blast dest: bvars-bval)
hence (IF b THEN c;; WHILE b DO c ELSE SKIP, t₁) →
  (c;; WHILE b DO c, t₁) ..
ultimately have
  (IF b THEN c;; WHILE b DO c ELSE SKIP, t₁) →*
  (c₂';; WHILE b DO c, t₂) ∧ c₂';; WHILE b DO c ≠ SKIP
by (blast intro: star-trans)
}

moreover { fix x
assume s = t₁ (⊆ sources ((bvars b) # flow cfs) s x)
moreover have sources (flow cfs) s x ⊆
  sources ((bvars b) # (flow cfs)) s x
by (rule sources-observe-tl)
ultimately have run-flow (flow cfs) s x = t₂ x
using V by blast
}
ultimately have $\exists c'_2 t_2. \forall x.$

$$(s = t_1 (\subseteq \text{sources-aux} ((\text{bvars} b) \# \text{flow cfs} s) x) \longrightarrow
\text{(IF } b \text{ THEN } c; \text{ WHILE } b \text{ DO } c \text{ ELSE SKIP, } t_1) \longrightarrow* (c'_2, t_2) \land
s = t_1 (\subseteq \text{sources} ((\text{bvars} b) \# \text{flow cfs} s) x) \longrightarrow
\text{run-flow (flow cfs) s x = t}_2 x)$$

by blast
}

moreover {

fix $s' \text{ cfs cfs'}$

assume

$$V: (c, s) \longrightarrow* \{\text{cfs} \}, (\text{SKIP}, s') \text{ and }$$

$$X: (\text{WHILE } b \text{ DO } c, s') \longrightarrow* \{\text{cfs}'\}$$

then obtain $c'_2$ and $t_2$ where $\forall x.$

$$(s = t_1 (\subseteq \text{sources-aux} (\text{flow cfs} s) x) \longrightarrow
(c, t_1) \longrightarrow* (c'_2, t_2) \land (\text{SKIP} = \text{SKIP}) \land
s = t_1 (\subseteq \text{sources} (\text{flow cfs} s) x) \longrightarrow s' x = t_2 x)$$

using $A \text{ [of } B_1 C B_1' D s \text{ ] c s cfs SKIP s']}$

and $N$ and $T$ by force

moreover have $Y: s' = \text{run-flow (flow cfs) s}$

using $W$ by (rule small-steps-run-flow)

ultimately have $Z: \forall x.$

$$(s = t_1 (\subseteq \text{sources-aux} (\text{flow cfs} s) x) \longrightarrow
(c, t_1) \longrightarrow* (\text{SKIP}, t_2) \land
s = t_1 (\subseteq \text{sources} (\text{flow cfs} s) x) \longrightarrow
\text{run-flow (flow cfs) s x = t}_2 x)$$

by blast

assume $s_2 = \text{run-flow (flow cfs') (run-flow (flow cfs) s}$

moreover have $(c, s) \Rightarrow s'$

using $W$ by (auto dest: small-steps-steps simp: big-iff-small)

hence $s' \in \text{Univ C} (\subseteq \text{state} \cap Y)$$

using $M$ and $S$ by blast

ultimately obtain $c'_3$ and $t_3$ where $AA: \forall x.$

$$(\text{run-flow (flow cfs) s} = t_2$$

$$\subseteq \text{sources-aux (flow cfs') (run-flow (flow cfs) s) x) \longrightarrow
\text{(WHILE } b \text{ DO } c, t_2) \longrightarrow* (c'_3, t_3) \land
(c_2 = \text{SKIP}) = (c'_3 = \text{SKIP}) \land
\text{run-flow (flow cfs) s} = t_2 $$

$$\subseteq \text{sources (flow cfs') (run-flow (flow cfs) s) x) \longrightarrow
\text{run-flow (flow cfs') (run-flow (flow cfs) s) x = t}_3 x$$

using $K \text{ [of } cfs' [cfs' s' WHILE } b \text{ DO } c s']$

and $V$ and $X$ and $Y$ by force

{ }

fix $x$

assume $AB: s = t_1$

$$\subseteq \text{sources-aux (\text{bvars} b) \# \text{flow cfs @ flow cfs'} s x)$$

moreover have $\text{sources-aux (flow cfs) s x } \subseteq
sources-aux (flow cfs @ flow cfs') s x
by (rule sources-aux-append)
moreover have AC: sources-aux (flow cfs @ flow cfs') s x ⊆
sources-aux ((bvars b) # flow cfs @ flow cfs') s x
by (rule sources-aux-observe-tl)
ultimately have (c, t₁) →* (SKIP, t₂)
using Z by blast
hence (c;; WHILE b DO c, t₁) →* (SKIP;; WHILE b DO c, t₂)
by (rule star-seq2)
moreover have s = t₁ (⊆ bvars b)
using Q and AB by (blast dest: sources-aux-observe-hd)
hence bval b t₁
by (simp dest: bvars-bval)
hence (IF b THEN c;; WHILE b DO c ELSE SKIP, t₁) →*
(WHILE b DO c, t₂)
by (blast intro: star-trans)
moreover have run-flow (flow cfs) s = t₂
(⊆ sources-aux (flow cfs') (run-flow (flow cfs) s) x)
proof
fix y
assume y ∈ sources-aux (flow cfs')
(run-flow (flow cfs) s) x
hence sources (flow cfs) s y ⊆
sources-aux (flow cfs @ flow cfs') s x
by (rule sources-aux-member)
hence sources (flow cfs) s y ⊆
sources-aux ((bvars b) # flow cfs @ flow cfs') s x
using AC' by simp
thus run-flow (flow cfs) s y = t₂ y
using Z and AB by blast
qed
hence (WHILE b DO c, t₂) →* (c₃', t₃) ∧
(c₂ = SKIP) = (c₃' = SKIP)
using AA by simp
ultimately have
(IF b THEN c;; WHILE b DO c ELSE SKIP, t₁) →*
(c₃', t₃) ∧ (c₂ = SKIP) = (c₃' = SKIP)
by (blast intro: star-trans)
}
moreover {
fix x
assume AB: s = t₁
(⊆ sources ((bvars b) # flow cfs @ flow cfs') s x)
have run-flow (flow cfs) s = t₂
(⊆ sources (flow cfs') (run-flow (flow cfs) s) x)
proof
fix y
assume $y \in \text{sources (flow cfs')}\)
\[(\text{run-flow (flow cfs)} s) x\]

**hence** $\text{sources (flow cfs)} s y \subseteq \text{sources (flow cfs @ flow cfs')} s x$
by (rule sources-member)

**moreover have** $\text{sources (flow cfs @ flow cfs')} s x \subseteq \text{sources ((\text{bvars b}) \# flow cfs @ flow cfs')} s x$
by (rule sources-observe-tl)

**ultimately have** $\text{sources (flow cfs)} s y \subseteq \text{sources ((\text{bvars b}) \# flow cfs @ flow cfs')} s x$
by simp

**thus** $\text{run-flow (flow cfs)} s y = t_2 y$

using $Z$ and $AB$ by blast

**qed**

**hence** $\text{run-flow (flow cfs')} (\text{run-flow (flow cfs)} s) x = t_3 x$

using $AA$ by simp

}\)

ultimately have $\exists c_3' t_3. \forall x.$

\((s = t_1 \quad \subseteq \text{sources-aux ((\text{bvars b}) \# flow cfs @ flow cfs')} s x) \rightarrow (\text{IF b THEN c ;; WHILE b DO c ELSE SKIP, t_1}) \rightarrow (c_3', t_3) \land (c_2 = \text{SKIP}) = (c_3' = \text{SKIP})) \land \quad (s = t_1 \quad \subseteq \text{sources ((\text{bvars b}) \# flow cfs @ flow cfs')} s x) \rightarrow \text{run-flow (flow cfs')} (\text{run-flow (flow cfs)} s) x = t_3 x)\)

by auto

}\)

ultimately show $\exists \text{thesis}$

using $R$ by (auto simp: run-flow-append)

next

assume $S: \neg \text{bval b s}$ and $T: \text{flow cfs}_2 = \langle \text{bvars b} \rangle \# \text{flow (tl cfs}_2)\)

assume $(\text{SKIP, s}) \rightarrow* \langle \text{tl cfs}_2 \rangle (c_2, s_2)$

**hence** $U: (c_2, s_2) = (\text{SKIP, s}) \land \text{flow (tl cfs}_2) = []$

by (rule small-stepsl-skip)

**show** $\text{thesis}$

proof (rule exI [of - \text{SKIP}], rule exI [of - t_1])

\(
\begin{align*}
\{ \\
\text{fix } x \\
\text{assume } s = t_1 (\subseteq \text{sources-aux ((\text{bvars b}) \# flow cfs)}) s x \\
\text{hence } s = t_1 (\subseteq \text{bvars b}) \\
\text{using } Q \text{ by (blast dest: sources-aux-observe-hd)} \\
\text{hence } \neg \text{bval b t}_1 \\
\text{using } S \text{ by (blast dest: bvars-bval)} \\
\text{hence } (\text{IF b THEN c ;; WHILE b DO c ELSE SKIP, t}_1) \rightarrow (\text{SKIP, t}_1) \ldots \\
\}
\end{align*}
\)

moreover {
\[\text{fix } x\]
\[\text{assume } s = t_1 (\subseteq \text{sources } [(bvars\ b)] s \ x)\]
\[\text{hence } s \ x = t_1 \ x\]
\[\text{by } (\text{subst\ (asm)} \ \text{append-Nil} \ [\text{symmetric}], \simp\ \text{only}: \text{sources.simps, auto})\]

ultimately show \(\forall x.\)
\[
(s_1 = t_1 (\subseteq \text{sources-aux} \ (\text{flow } cfs_2) s_1 \ x) \rightarrow (c_1, t_1) \rightarrow^{*} (\text{SKIP}, t_1) \wedge (c_2 = \text{SKIP}) = (\text{SKIP} = \text{SKIP})) \wedge (s_1 = t_1 (\subseteq \text{sources} \ (\text{flow } cfs_2) s_1 \ x) \rightarrow s_2 \ x = t_1 \ x)\]
\[\text{using } R \text{ and } T \text{ and } U \text{ by auto}\]
\[\text{qed}\]

\text{next}\]
\[
\text{assume } R: \text{bval\ b s} \\text{with } D \text{ and } O \text{ and } P \text{ have } S: s \in \text{Univ\ B}_1 (\subseteq \text{state} \cap X)\]
\[\text{by } (\text{drule-tac\ btyping2-approx} \ [\text{where } s = s], \text{auto})\]
\[\text{assume } (c\ ;; \text{WHILE } b \text{ DO } c, s) \rightarrow^{*} \{\text{tl2 cfs}_1\} \ (c_1, s_1)\]
\[\text{hence}\]
\[
(\exists c' \ cfs'. c_1 = c'_1; \ \text{WHILE } b \text{ DO } c \wedge (c, s) \rightarrow^{*} \{cfs'\} (c'_1, s_1) \wedge \text{flow\ (tl2 cfs}_1) = \text{flow\ cfs'} \vee (\exists s' \ cfs' \ cfs'' . \text{length\ cfs''} < \text{length\ (tl2 cfs}_1) \wedge (c, s) \rightarrow^{*} \{cfs'\} (\text{SKIP}, s') \wedge (\text{WHILE } b \text{ DO } c, s') \rightarrow^{*} \{cfs''\} (c_1, s_1) \wedge \text{flow\ (tl2 cfs}_1) = \text{flow\ cfs'} \otimes \text{flow\ cfs''})\]
\[\text{by } (\text{rule\ small-steps-l-seq})\]

moreover {\text{fix } c' \ cfs} \\text{assume}\]
\[T: (c, s) \rightarrow^{*} \{cfs\} (c'_1, s_1) \text{ and}\]
\[U: c_1 = c'_1; \ \text{WHILE } b \text{ DO } c\]
\[\text{hence } V: (c'_1\ ;; \text{WHILE } b \text{ DO } c, s_1) \rightarrow^{*} \{cfs_2\} (c_2, s_2)\]
\[\text{by } J \text{ by simp}\]
\[\text{hence } W: s_2 = \text{run-flow} \ (\text{flow cfs}_2) s_1\]
\[\text{by } (\text{rule\ small-steps-l-run-flow})\]
\[\text{have}\]
\[\exists c'' \ cfs', c_2 = c''_1; \ \text{WHILE } b \text{ DO } c \wedge (c', s_1) \rightarrow^{*} \{cfs'\} (c''_1, s_2) \wedge \text{flow\ cfs}_2 = \text{flow\ cfs'} \vee (\exists s' \ cfs' \ cfs''. \text{length\ cfs''} < \text{length\ cfs}_2 \wedge (c', s_1) \rightarrow^{*} \{cfs'\} (\text{SKIP}, s') \wedge (\text{WHILE } b \text{ DO } c, s') \rightarrow^{*} \{cfs''\} (c_2, s_2) \wedge \text{flow\ cfs}_2 = \text{flow\ cfs'} \otimes \text{flow\ cfs''})\]
\[\text{using } V \text{ by } (\text{rule\ small-steps-l-seq})\]

moreover {\text{fix } c'' \ cfs'} \\text{assume } (c', s_1) \rightarrow^{*} \{cfs'\} (c''_1, s_2) \\text{then obtain } c_2' \text{ and } t_2 \text{ where } X: \forall x.\]

112
(s_1 = t_1 (\subseteq \text{sources-aux} (\text{flow cfs}') s_1 \ x) \implies
(c', t_1) \rightarrow^* (c_2', t_2) \land (c'' = \text{SKIP}) = (c_2' = \text{SKIP}) \land
(s_1 = t_1 (\subseteq \text{sources} (\text{flow cfs}') s_1 \ x) \implies
\text{run-flow (flow cfs_2) s_1 x} = t_2 x)
using A \ [\text{of } B_1 \ C B_1' D s \ cfs' c_1 cfs' c'']
\text{run-flow (flow cfs_2) s_1} \text{ and } N \text{ and } S \text{ and } T \text{ and } W \text{ by force}
assume
Y: \ (c_2 = c''; \ \text{WHILE } b \ DO \ c \ and
Z: \ (\text{flow cfs}_2 = \text{flow cfs}')
have \ \lnot \text{thesis}
proof \ (\text{rule exI [of - c_2']}; \ \text{WHILE } b \ DO \ c], \ \text{rule exI [of - t_2]})
from U \ \text{and } W \ \text{and } X \ \text{and } Y \ \text{and } Z \ \text{show } \forall x.
(s_1 = t_1 (\subseteq \text{sources-aux} (\text{flow cfs}_2) s_1 x) \implies
(c_1, t_1) \rightarrow^* (c_2'; \ \text{WHILE } b \ DO \ c, t_2) \land
(c_2 = \text{SKIP}) = (c_2'; \ \text{WHILE } b \ DO \ c = \text{SKIP}) \land
(s_1 = t_1 (\subseteq \text{sources} (\text{flow cfs}_2) s_1 x) \implies s_2 x = t_2 x)
by \ (\text{auto intro: star-seq2})
qed
}
moreover \{
fix s' cfs' cfs''
assume
X: \ (\text{flow cfs}' < \text{length cfs}_2 \ and
Y: \ (c', s_1) \rightarrow^* \{cfs' \} (\text{SKIP}, s') \ and
Z: \ (\text{WHILE } b \ DO \ c, s') \rightarrow^* \{cfs''\} (c_2, s_2)
then obtain c_2' and t_2 where \forall x.
(s_1 = t_1 (\subseteq \text{sources-aux} (\text{flow cfs}') s_1 x) \implies
(c', t_1) \rightarrow^* (c_2', t_2) \land (\text{SKIP} = \text{SKIP}) = (c_2' = \text{SKIP}) \land
(s_1 = t_1 (\subseteq \text{sources} (\text{flow cfs}') s_1 x) \implies s' x = t_2 x)
using A \ [\text{of } B_1 \ C B_1' D s \ cfs' c_1 cfs' \text{SKIP} s']
and N \text{ and } S \text{ and } T \text{ by force}
moreover have AA: s' = \text{run-flow (flow cfs') s_1}
using Y \ by \ (\text{rule small-stepsl-run-flow})
ultimately have AB: \forall x.
(s_1 = t_1 (\subseteq \text{sources-aux} (\text{flow cfs}') s_1 x) \implies
(c', t_1) \rightarrow^* (\text{SKIP}, t_2) \land
(s_1 = t_1 (\subseteq \text{sources} (\text{flow cfs}') s_1 x) \implies
\text{run-flow (flow cfs') s_1 x} = t_2 x)
by \ \text{blast}
have AC: s_2 = \text{run-flow (flow cfs''')} s'
using Z \ by \ (\text{rule small-stepsl-run-flow})
moreover have (c, s) \rightarrow^* \{cfs \ @ cfs' \} (\text{SKIP}, s')
using T \ and \ Y \ by \ \text{(simp add: small-stepsl-append)}
\text{hence } (c, s) \Rightarrow s'
by \ (\text{auto dest: small-stepsl-steps simp: big-iff-small})
\text{hence } s' \in \text{Univ } C (\subseteq \text{state} \cap Y)
using M \text{ and } R \ \text{by blast}
ultimately obtain c_2' and t_3 where AD: \forall x.
(\text{run-flow (flow cfs')} s_1 = t_2)
\[\subseteq sources-aux (\text{flow } cfs'') (\text{run-flow } (\text{flow } cfs') s_1) x \longrightarrow \]
\[
(WHILE b DO c, t_2) \rightarrow* (c_2', t_3) \land \\
(c_2 = \text{SKIP}) = (c_2' = \text{SKIP}) \land \\
(\text{run-flow } (\text{flow } cfs') s_1 = t_2)
\]
\[
(\subseteq sources (\text{flow } cfs'') (\text{run-flow } (\text{flow } cfs') s_1) x) \longrightarrow \\
\text{run-flow } (\text{flow } cfs'') (\text{run-flow } (\text{flow } cfs') s_1) x = t_3 x
\]
\[
\text{using } K \left[ \text{of } cfs'' \right] cfs'' s' WHILE b DO c s'
\]
and \(X\) and \(Z\) and \(AA\) by force
Moreover assume \(\text{flow } cfs_2 = \text{flow } cfs' \circ \text{flow } cfs''\)
Moreover { 
fix \(x\)
assume \(AE: s_1 = t_1\)
(\(\subseteq sources-aux (\text{flow } cfs' \circ \text{flow } cfs'') s_1 x\))
moreover have \(\text{sources-aux } (\text{flow } cfs') s_1 x \subseteq \text{sources-aux } (\text{flow } cfs' \circ \text{flow } cfs'') s_1 x\)
by (rule sources-aux-append)
ultimately have \((c', t_1) \rightarrow* (\text{SKIP}, t_2)\)
using \(AB\) by blast
hence \((c'; WHILE b DO c, t_1) \rightarrow* (\text{SKIP}; WHILE b DO c, t_2)\)
by (rule star-seq2)
hence \((c'; WHILE b DO c, t_1) \rightarrow* (WHILE b DO c, t_2)\)
by (blast intro: star-trans)
moreover have \(\text{run-flow } (\text{flow } cfs') s_1 = t_2\)
(\(\subseteq sources-aux (\text{flow } cfs'') (\text{run-flow } (\text{flow } cfs') s_1) x\))
proof
fix \(y\)
assume \(y \in sources-aux (\text{flow } cfs'')\)
(\(\text{run-flow } (\text{flow } cfs') s_1 x\))
hence \(\text{sources } (\text{flow } cfs') s_1 y \subseteq \text{sources-aux } (\text{flow } cfs' \circ \text{flow } cfs'') s_1 x\)
by (rule sources-aux-member)
thus \(\text{run-flow } (\text{flow } cfs') s_1 y = t_2 y\)
using \(AB\) and \(AE\) by blast
qed
hence \((WHILE b DO c, t_2) \rightarrow* (c_2', t_3) \land \\
(c_2 = \text{SKIP}) = (c_2' = \text{SKIP}) \land \\
(\text{run-flow } (\text{flow } cfs') s_1 = t_2)
\]
using \(AD\) by simp
ultimately have \((c'; WHILE b DO c, t_1) \rightarrow* (c_2', t_3) \land \\
(c_2 = \text{SKIP}) = (c_2' = \text{SKIP}) \land \\
by (blast intro: star-trans)
}
moreover { 
fix \(x\)
assume \(AE: s_1 = t_1\)
(\(\subseteq sources (\text{flow } cfs' \circ \text{flow } cfs'') s_1 x\))
have \(\text{run-flow } (\text{flow } cfs') s_1 = t_2\)
(\(\subseteq sources (\text{flow } cfs'') (\text{run-flow } (\text{flow } cfs') s_1) x\))
proof
fix \(y\)
fix ...
assume $y \in \text{sources}(\text{flow cfs}')$

$(\text{run-flow (flow cfs')} s_1) x$

hence $\text{sources}(\text{flow cfs')} s_1 y \subseteq \text{sources}(\text{flow cfs' @ flow cfs''}) s_1 x$

by (rule sources-member)

thus $(\text{run-flow (flow cfs')} s_1 y = t_2 y)$

using $AB$ and $AE$ by blast

qed

hence $(\text{run-flow (flow cfs')} s_1 x = t_3 x)$

using $AD$ by simp

}\}

ultimately have $?thesis$

by (metis U AA AC)

}\}

ultimately have $?thesis$

by blast

}\}

moreover { fix $s' \text{ cfs cfs'}$

assume

(length cfs' < length ($\text{tl2 cfs}_1$)) and

$(c, s) \rightarrow_{\text{cfs}} (\text{SKIP}, s')$ and

$(\text{WHILE b DO c, s'} \rightarrow_{\text{cfs'}} (c_1, s_1)$

moreover from this have $(c, s) \Rightarrow s'$

by (auto dest: small-steps-steps simp: big-iff-small)

hence $s' \in \text{Univ C} (\subseteq \text{state} \cap Y)$

using $M$ and $R$ by blast

ultimately have $?thesis$

using $K [\text{of cfs' @ cfs_2 cfs' cfs_2 s'} c_1 s_1]$ and $J$ by force

}\}

ultimately show $?thesis$

by blast

next

assume $(\text{SKIP}, s) \rightarrow_{\text{cfs'}} (\text{tl2 cfs}_1) (c_1, s_1)$

hence $(c_1, s_1) = (\text{SKIP}, s)$

by (blast dest: small-steps-skip)

moreover from this have $(c_2, s_2) = (\text{SKIP}, s) \land \text{flow cfs}_2 = []$

using $J$ by (blast dest: small-steps-skip)

ultimately show $?thesis$

by auto

qed

}\}

moreover {

fix $r t_1$

assume $O: r \in C$ and $P: s = r (\subseteq \text{state} \cap Y)$

have $Q: \forall x. \forall y \in \text{bvars} b. s: \text{dom y} \rightarrow \text{dom x}$

proof (cases state $\subseteq Y$)
case True
  with P have interf s = interf r
    by (blast intro: interf-state)
with N and O show ?thesis
  by (erule-tac conjE, drule-tac bspec,
    auto simp: univ-states-if-def)

next
case False
  with N and O show ?thesis
    by (erule-tac conjE, drule-tac bspec,
      auto simp: univ-states-if-def)

qed

have \((c_1, s_1) = (\text{WHILE } b \text{ DO } c, s) \lor \) 
  \((\text{IF } b \text{ THEN } c\text{;; WHILE } b \text{ DO } c \text{ ELSE SKIP}, s) \rightarrow^* \{tl\ cfs_1\} \ (c_1, s_1)\)
using I by (blast dest: small-stepsl-while)

hence \(\exists c_2', t_2. \forall x.\) 
  \((s_1 = t_1 (\subseteq \text{sources-aux} (flow\ cfs_2) \ s_1 \ x) \rightarrow\) 
  \((c_1, t_1) \rightarrow^* (c_2', t_2) \land (c_2 = \text{SKIP}) = (c_2' = \text{SKIP}) \land \) 
  \((s_1 = t_1 (\subseteq \text{sources} (flow\ cfs_2) \ s_1 \ x) \rightarrow\ s_2 x = t_2 x)\)
proof
assume R: \((c_1, s_1) = (\text{WHILE } b \text{ DO } c, s)\)

hence \((\text{WHILE } b \text{ DO } c, s) \rightarrow^* \{cfs_2\} \ (c_2, s_2)\)
using J by simp

hence \((c_2, s_2) = (\text{WHILE } b \text{ DO } c, s) \land \) 
  \(\text{flow\ cfs}_2 = [] \lor \) 
  \((\text{IF } b \text{ THEN } c\text{;; WHILE } b \text{ DO } c \text{ ELSE SKIP}, s) \rightarrow^* \{tl\ cfs_2\} \ (c_2, s_2) \land \) 
  \(\text{flow\ cfs}_2 = \text{flow} (\{tl\ cfs_2\})\)
(is \(P \lor Q \land R\))
by (rule small-stepsl-while)

thus \(?thesis\)

proof (rule disjE, erule-tac [2] conjE)
assume \(?P\)
with R show \(?thesis\)
  by auto

next
assume \(?Q\) and \(?R\)

have \((c_2, s_2) = (\text{IF } b \text{ THEN } c\text{;; WHILE } b \text{ DO } c \text{ ELSE SKIP}, s) \land \) 
  \(\text{flow} (\{tl\ cfs_2\}) = [] \lor \) 
  \(\text{bval\ b\ }s \land (c\text{;; WHILE } b \text{ DO } c, s) \rightarrow^* \{tl2\ cfs_2\} \ (c_2, s_2) \land \) 
  \(\text{flow} (\{tl\ cfs_2\} = (\text{bvars} \ b) \# \text{flow} (\{tl2\ cfs_2\}) \lor \) 
  \(\neg \text{bval\ b\ }s \land (\text{SKIP}, s) \rightarrow^* \{tl2\ cfs_2\} \ (c_2, s_2) \land \) 
  \(\text{flow} (\{tl\ cfs_2\} = (\text{bvars} \ b) \# \text{flow} (\{tl2\ cfs_2\})\)
using \(?Q\) by (rule small-stepsl-if)

thus \(?thesis\)

assume \((c_2, s_2) = (\text{IF } b \text{ THEN } c\text{;; WHILE } b \text{ DO } c \text{ ELSE SKIP}, s) \land \) 
  \(\text{flow} (\{tl\ cfs_2\}) = []\)
with $R$ and $s$ show ?thesis
by auto

next

assume $S$: $bval b s$

with $F$ and $O$ and $P$ have $T$: $s \in \text{Univ } B_1 \ (\subseteq \text{state } \cap Y)$

by (drule_tac btyping2-approx [where $s = s$], auto)

assume $U$: $(c;\ WHILE \ b \ DO \ c, \ s) \rightarrow \{ tl2 \ cfs_2 \} \ (v_2, \ s_2)$

hence

$(\exists \ c' \ cfs, \ c_2 = c'_c;\ WHILE \ b \ DO \ c \land$
\begin{align*}
(c, s) & \rightarrow \{ cfs \} \ (c', s_2) \land \\
\text{flow} \ (tl2 \ cfs_2) & = \text{flow} \ cfs \lor
\end{align*}

$(\exists s' \ cfs \ cfs', \ \text{length} \ cfs' < \text{length} \ (tl2 \ cfs_2) \land$
\begin{align*}
(c, s) & \rightarrow \{ cfs \} \ (\text{SKIP}, \ s') \land \\
\text{flow} \ (tl2 \ cfs_2) & = \text{flow} \ cfs \land \text{flow} \ cfs'
\end{align*}

by (rule small-stepsl-seq)

moreover assume \text{flow} \ (tl \ cfs_2) = \{(bvars \ b) \ # \ \text{flow} \ (tl2 \ cfs_2)\)

moreover have $s_2 = \text{run-flow} \ (\text{flow} \ (tl2 \ cfs_2)) \ s$

using $U$ by (rule small-stepsl-run-flow)

moreover \{ fix $c'$ \ cfs 

assume $(c, s) \rightarrow \{ cfs \} \ (c', \ \text{run-flow} \ (\text{flow} \ cfs) \ s)$

then obtain $c_2'$ and $t_2$ where $V$: $\forall x.$

$(s = t_1 \ (\subseteq \text{sources-aux} \ (\text{flow} \ cfs) \ s \ x) \longrightarrow$
\begin{align*}
(c, t_1) & \rightarrow \{ c_2', t_2 \} \land (c' = \text{SKIP}) \land \\
(s = t_1 & \ (\subseteq \text{sources} \ (\text{flow} \ cfs) \ s \ x) \longrightarrow \\
\text{run-flow} \ (\text{flow} \ cfs) \ s \ x & = t_2 \ x
\end{align*}

using $B \ [\text{of } B_1 \ C \ B_1^\bot \ D' \ s \ [\ c s cfs c'\]

\text{run-flow} \ (\text{flow} \ cfs) \ s \ \text{and} \ N \ \text{and} \ T \ \text{by force} \ \{ \$

fix $x$

assume $W$: $s = t_1 \ (\subseteq \text{sources-aux} \ ((bvars \ b) \ # \ (\text{flow} \ cfs) \ s \ x)$

moreover have $\text{sources-aux} \ (\text{flow} \ cfs) \ s \ x \subseteq$
\begin{align*}
\text{sources-aux} \ ((bvars \ b) \ # \ (\text{flow} \ cfs) \ s \ x
\end{align*}

by (rule sources-aux-observe-tl)

ultimately have $(c, t_1) \rightarrow \{ c'_2, t_2 \})$

using $V$ by blast

hence $(c;\ WHILE \ b \ DO \ c, \ t_1) \rightarrow \{ c'_2, t_2;\ WHILE \ b \ DO \ c, \ t_2)$

by (rule star-seq2)

moreover have $s = t_1 \ (\subseteq \text{bvars} \ b)$

using $Q$ and $W$ by (blast dest: sources-aux-observe-hd)

hence $bval b t_1$

using $S$ by (blast dest: bvars-bval)

hence $(WHILE \ b \ DO \ c, \ t_1) \rightarrow \{ c;\ WHILE \ b \ DO \ c, \ t_1)$

by (blast intro: star-trans)

ultimately have $(WHILE \ b \ DO \ c, \ t_1) \rightarrow$
\begin{align*}
(c'_2;\ WHILE \ b \ DO \ c, \ t_2) \land c'_2;\ WHILE \ b \ DO \ c \neq \text{SKIP}
\end{align*}

by (blast intro: star-trans) \} \}
moreover { 
   \text{fix } x \\
   \text{assume } s = t_1 (\subseteq \text{sources } ((\text{bvars } b) \# \text{flow } cfs) s x) \\
   \text{moreover have } \text{sources } (\text{flow } cfs) s x \subseteq \\
   \text{sources } ((\text{bvars } b) \# (\text{flow } cfs)) s x \\
   \text{by (rule sources-observe-ll) \\
   ultimately have } \text{run-flow } (\text{flow } cfs) s x = t_2 x \\
   \text{using } V \text{ by blast} 
} \\
ultimately have \exists c_2', t_2. \forall x. \\
(s = t_1 (\subseteq \text{sources-aux } ((\text{bvars } b) \# \text{flow } cfs) s x) \rightarrow \\
(\text{WHILE } b \text{ DO } c, t_1) \rightarrow^* (c_2', t_2) \land c_2' \neq \text{SKIP}) \land \\
(s = t_1 (\subseteq \text{sources } ((\text{bvars } b) \# \text{flow } cfs) s x) \rightarrow \\
\text{run-flow } (\text{flow } cfs) s x = t_2 x \\
\text{by blast} 
} \\
moreover { 
   \text{fix } s' \text{ cfs } cfs' \\
   \text{assume} \\
   V: \text{length } cfs' < \text{length } cfs_2 - \text{Suc } (\text{Suc } 0) \text{ and} \\
   W: (c, s) \rightarrow^* \{cfs\} (\text{SKIP, } s') \text{ and} \\
   X: (\text{WHILE } b \text{ DO } c, s') \rightarrow^* \{cfs\} \\
   (c_2, \text{run-flow } (\text{flow } cfs') (\text{run-flow } (\text{flow } cfs) s)) \rightarrow \\
\text{then obtain } c_2' \text{ and } t_2 \text{ where } \forall x. \\
(s = t_1 (\subseteq \text{sources-aux } (\text{flow } cfs) s x) \rightarrow \\
(c, t_1) \rightarrow^* (c_2', t_2) \land (\text{SKIP = SKIP}) = (c_2' = \text{SKIP}) \land \\
(s = t_1 (\subseteq \text{sources } (\text{flow } cfs) s x) \rightarrow s' x = t_2 x \\
\text{using } B [\text{of } B_1 C B_1^* D' s \[ c s cfs \text{SKIP } s'] \\
\text{and } N \text{ and } T \text{ by force} \\
\text{moreover have } Y: s' = \text{run-flow } (\text{flow } cfs) s \\
\text{using } W \text{ by (rule small-steps-l-run-flow) \\
ultimately have } Z: \forall x. \\
(s = t_1 (\subseteq \text{sources-aux } (\text{flow } cfs) s x) \rightarrow \\
(c, t_1) \rightarrow^* (\text{SKIP, } t_1)) \land \\
(s = t_1 (\subseteq \text{sources } (\text{flow } cfs) s x) \rightarrow \\
\text{run-flow } (\text{flow } cfs) s x = t_2 x \\
\text{by blast} \\
\text{assume } s_0 = \text{run-flow } (\text{flow } cfs') (\text{run-flow } (\text{flow } cfs) s) \\
\text{moreover have } (c, s) \Rightarrow s' \\
\text{using } W \text{ by (auto dest: small-steps-steps simp: big-iff-small) } \\
\text{hence } s' \in \text{Univ } C (\subseteq \text{state } \cap Y) \\
\text{using } M \text{ and } S \text{ by blast} \\
\text{ultimately obtain } c_3' \text{ and } t_3 \text{ where } AA: \forall x. \\
(\text{run-flow } (\text{flow } cfs) s = t_2 \\
(\subseteq \text{sources-aux } (\text{flow } cfs') (\text{run-flow } (\text{flow } cfs) s) x) \rightarrow \\
(\text{WHILE } b \text{ DO } c, t_2) \rightarrow^* (c_3', t_3) \land \\
(c_2 = \text{SKIP}) = (c_3' = \text{SKIP}) \land \\
(\text{run-flow } (\text{flow } cfs) s) = t_2 \\
(\subseteq \text{sources } (\text{flow } cfs') (\text{run-flow } (\text{flow } cfs) s) x) \rightarrow \\
(\text{run-flow } (\text{flow } cfs') (\text{run-flow } (\text{flow } cfs) s) x) \rightarrow \\
118
run-flow (flow cfs') (run-flow (flow cfs) s) x = t_3 x
using K [of cfs' [] cfs' s' WHILE b DO c s']
and V and X and Y by force

\{ 
fix x
assume AB: s = t_1
(\subseteq sources-aux ((bvars b) \# flow cfs @ flow cfs') s x)
moreover have sources-aux (flow cfs) s x \subseteq
sources-aux (flow cfs @ flow cfs') s x
by (rule sources-aux-append)
moreover have AC: sources-aux (flow cfs @ flow cfs') s x \subseteq
sources-aux ((bvars b) \# flow cfs @ flow cfs') s x
by (rule sources-aux-observe-tl)
ultimately have (c, t_1) \rightarrow* (SKIP, t_2)
using Z by blast
hence (c; WHILE b DO c, t_1) \rightarrow* (SKIP; WHILE b DO c, t_2)
by (rule star-seq2)
moreover have s = t_1 (\subseteq bvars b)
using Q and AB by (blast dest: sources-aux-observe-hd)
hence beal b t_1
using S by (blast dest: bvars-beal)
hence (WHILE b DO c, t_1) \rightarrow* (c; WHILE b DO c, t_1)
by (blast intro: star-trans)
ultimately have (WHILE b DO c, t_1) \rightarrow* (WHILE b DO c, t_2)
by (blast intro: star-trans)
moreover have run-flow (flow cfs) s = t_2
(\subseteq sources-aux (flow cfs') (run-flow (flow cfs) s) x)
proof
fix y
assume y \in sources-aux (flow cfs')
(run-flow (flow cfs) s) y
hence sources (flow cfs) s y \subseteq
sources-aux (flow cfs @ flow cfs') s x
by (rule sources-aux-member)
hence sources (flow cfs) s y \subseteq
sources-aux ((bvars b) \# flow cfs @ flow cfs') s x
using AC by simp
thus run-flow (flow cfs) s y = t_2 y
using Z and AB by blast
qed
hence (WHILE b DO c, t_2) \rightarrow* (c_3', t_3) \land
(c_2 = SKIP) = (c_3' = SKIP)
using AA by simp
ultimately have (WHILE b DO c, t_1) \rightarrow* (c_3', t_3) \land
(c_2 = SKIP) = (c_3' = SKIP)
by (blast intro: star-trans)
\}
moreover {
fix x
}
assume $AB: s = t_1$

$(\subseteq sources ((bvars b) \# flow cfs \@ flow cfs') s x)$

have $run-flow (flow cfs) s = t_2$

$(\subseteq sources (flow cfs') (run-flow (flow cfs) s) x)$

proof

fix $y$

assume $y \in sources (flow cfs')$

$(run-flow (flow cfs) s) x$

hence $sources (flow cfs) s y \subseteq$

sources $(flow cfs \@ flow cfs') s x$

by (rule sources-member)

moreover have $sources (flow cfs \@ flow cfs') s x \subseteq$

sources $((bvars b) \# flow cfs \@ flow cfs') s x$

by (rule sources-observe-tl)

ultimately have $sources (flow cfs) s y \subseteq$

sources $((bvars b) \# flow cfs \@ flow cfs') s x$

by simp

thus $run-flow (flow cfs) s y = t_2 y$

using $Z$ and $AB$ by blast

qed

hence $run-flow (flow cfs') (run-flow (flow cfs) s) x = t_3 x$

using $AA$ by simp

}{

ultimately have $\exists c_3, t_3, \forall x.

(s = t_1$

$(\subseteq sources-aux ((bvars b) \# flow cfs \@ flow cfs') s x) \rightarrow$

$(WHILE b DO c, t_1) \rightarrow^* (c_3', t_3) \land$

$(c_2 = SKIP) = (c_3' = SKIP)) \land$

$(s = t_1$

$(\subseteq sources ((bvars b) \# flow cfs \@ flow cfs') s x) \rightarrow$

run-flow (flow cfs') (run-flow (flow cfs) s) x = t_3 x)$

by auto

}

ultimately show $?thesis$

using $R$ and $?R$ by (auto simp: run-flow-append)

next

assume $S: \neg bval b s$ and

$T: flow (tl cfs) = (bvars b) \# flow (tl2 cfs)$

assume $(SKIP, s) \rightarrow^* (tl2 cfs) (c_2, s_2)$

hence $U: (c_2, s_2) = (SKIP, s) \land flow (tl2 cfs) = []$

by (rule small-stepsl-skip)

show $?thesis$

proof (rule exI [of - SKIP], rule exI [of - t_1])

{ fix $x$

have $(WHILE b DO c, t_1) 

(IF b THEN c; WHILE b DO c ELSE SKIP, t_1) ..$

moreover assume $s = t_1 (\subseteq sources-aux [\{bvars b\} s x)
hence \( s = t_1 (\subseteq bvars b) \)
using \( Q \) by (blast dest: sources-aux-observe-hd)
hence \( \neg bval b t_1 \)
using \( S \) by (blast dest: bvars-bval)
hence \( (IF b THEN c;; WHILE b DO c ELSE SKIP, t_1) \rightarrow (SKIP, t_1) .. \)
ultimately have \( (WHILE b DO c, t_1) \rightarrow* (SKIP, t_1) \)
by (blast intro: star-trans)

moreover \{
fix \( x \)
assume \( s = t_1 (\subseteq sources [(bvars b)] s x) \)
hence \( s x = t_1 x \)
by (subst (asm) append-Nil [symmetric],simp only: sources.simps, auto)
\}
ultimately show \( \forall x. \)
\( (s_1 = t_1 (\subseteq sources-aux (flow cfs_2) s_1 x) \rightarrow \)
\( (c_1, t_1) \rightarrow* (SKIP, t_1) \land (c_2 = SKIP) = (SKIP = SKIP) \land \)
\( (s_1 = t_1 (\subseteq sources (flow cfs_2) s_1 x) \rightarrow s_2 x = t_1 x) \)
using \( R \) and \( T \) and \( U \) and \( t?R; \) by auto
qed
qed
qed

next
assume \( (IF b THEN c;; WHILE b DO c ELSE SKIP, s) \rightarrow*{tl cfs_1} (c_1, s_1) \)
hence \( (c_1, s_1) = (IF b THEN c;; WHILE b DO c ELSE SKIP, s) \land \)
flow \( (tl cfs_1) = [] \lor \)
bval b s \land (c;; WHILE b DO c, s) \rightarrow*{tl2 cfs_1} (c_1, s_1) \land
flow \( (tl cfs_1) = \langle bvars b \rangle \# flow (tl2 cfs_1) \lor \)
\neg bval b s \land (SKIP, s) \rightarrow*{tl2 cfs_1} (c_1, s_1) \land
flow \( (tl cfs_1) = \langle bvars b \rangle \# flow (tl2 cfs_1) \)
by (rule small-steps1-if)
thus \(?thesis \)
proof (rule disjE, erule-tac \([2]\) disjE, erule-tac conjE, (erule-tac \([2-3]\) conjE)+)
assume \( R: (c_1, s_1) = (IF b THEN c;; WHILE b DO c ELSE SKIP, s) \)
hence \( (IF b THEN c;; WHILE b DO c ELSE SKIP, s) \rightarrow*{cfs_2} (c_2, s_2) \)
using \( J \) by simp
hence \( (c_2, s_2) = (IF b THEN c;; WHILE b DO c ELSE SKIP, s) \land \)
flow \( cfs_2 = [] \lor \)
bval b s \land (c;; WHILE b DO c, s) \rightarrow*{tl cfs_2} (c_2, s_2) \land
flow \( cfs_2 = \langle bvars b \rangle \# flow (tl cfs_2) \lor \)
\neg bval b s \land (SKIP, s) \rightarrow*{tl cfs_2} (c_2, s_2) \land
flow \( cfs_2 = \langle bvars b \rangle \# flow (tl cfs_2) \)
by (rule small-steps1-if)
thus \(?thesis \)
assume \((c_2, s_2) = (IF b THEN c_2; WHILE b DO c ELSE SKIP, s) \land
flow cfs_2 = [\]
with \(R\) show \(?\)thesis
by auto
next
assume \(S\): bval b s
with \(P\) and \(O\) and \(P\) have \(T\): \(s \in Univ B_1^1 (\subseteq state \cap Y)\)
by (erule-tac btyping2-approx [\textbf{where} s = s, auto)
assume \(U\): (c_2; WHILE b DO c, s) \rightarrow \{tl cfs_2\} (c_2, s_2)
hence
\(\exists c' cfs. c_2 = c'\); WHILE b DO c \land
\((c, s) \rightarrow \{cfs\} (c', s_2) \land
flow (tl cfs_2) = flow cfs \lor
(\exists s' cfs' length cfs' < length (tl cfs_2) \land
\( (c, s) \rightarrow \{cfs\} (\text{SKIP}, s') \land
\( (WHILE b DO c, s') \rightarrow \{cfs'\} (c_2, s_2) \land
flow (tl cfs_2) = flow cfs @ flow cfs'\)
by (rule small-steps-l-seq)
moreover assume flow cfs_2 = (\text{bvars} b) \# flow (tl cfs_2)
moreover have \(s_2 = \text{run-flow} (flow (tl cfs_2)) s\)
using \(U\) by (rule small-steps-l-run-flow)
moreover \{ 
fix \(c' cfs\)
assume \((c, s) \rightarrow \{cfs\} (c', \text{run-flow} (flow cfs) s)\)
then obtain \(c_2'\) and \(t_2\) where \(V\): \(\forall x. \)
\((s = t_1 (\subseteq \text{sources-aux} (flow cfs) s) x) \rightarrow
(c, t_1) \rightarrow (c_2', t_2) \land (c' = \text{SKIP} = (c_2' = \text{SKIP})) \land
(s = t_1 (\subseteq \text{sources} (flow cfs) s) x) \rightarrow
\text{run-flow} (flow cfs) s x = t_2 x)\)
using \(B\) [of \(B_1^1 C B_1^1 D\)'] \[ c s cfs c' \]
\text{run-flow} (flow cfs) s] and \(N\) and \(T\) by force
\}
fix \(x\)
assume \(W\): \(s = t_1 (\subseteq \text{sources-aux} ((\text{bvars} b) \# flow cfs) s) x)\)
moreover have \(\text{sources-aux} (flow cfs) s x \subseteq
\text{sources-aux} ((\text{bvars} b) \# (flow cfs)) s x\)
by (rule sources-aux-observe-tl)
ultimately have \((c, t_1) \rightarrow (c_2', t_2)\)
using \(V\) by blast
hence \((c; ; WHILE b DO c, t_1) \rightarrow (c_2'; ; WHILE b DO c, t_2)\)
by (rule star-seq2)
moreover have \(s = t_1 (\subseteq \text{bvars} b)\)
using \(Q\) and \(W\) by (blast dest: sources-aux-observe-hd)
hence bval b \(t_1\)
using \(S\) by (blast dest: bvars-bval)
hence \((\text{IF} b \text{ THEN} c; ; \text{WHILE b DO c ELSE SKIP}), t_1) \rightarrow
(c; ; \text{WHILE b DO c, t_1} ..
ultimately have

122
\[(IF \; b \; THEN \; c_1; \; WHILE \; b \; DO \; c \; ELSE \; SKIP, \; t_1) \rightarrow^* \]
\[(c_2'; \; WHILE \; b \; DO \; c, \; t_2) \land c_2' \land \; WHILE \; b \; DO \; c \neq \; SKIP \]
by \((\text{blast intro: star-trans})\)

moreover \{ 
fix \(x\) 
assume \(s = t_1 (\subseteq \text{sources} ((\text{bvars} \; b) \neq \text{flow cfs}) \; s \; x)\)
moreover have \(\text{sources} (\text{flow cfs}) \; s \; x \subseteq \)
\(\text{sources} ((\text{bvars} \; b) \neq (\text{flow cfs})) \; s \; x\)
by \((\text{rule sources-observe-ll})\)
ultimately have \(\text{run-flow} (\text{flow cfs}) \; s \; x = t_2 \; x\)
using \(V\) by \(\text{blast}\)
\}
ultimately have \(\exists c_2', \; t_2, \; \forall x.\)
\((s = t_1 (\subseteq \text{sources-aux} ((\text{bvars} \; b) \neq \text{flow cfs}) \; s \; x) \longrightarrow \)
\((IF \; b \; THEN \; c_1; \; WHILE \; b \; DO \; c \; ELSE \; SKIP, \; t_1) \rightarrow^* (c_2', \; t_2) \land c_2' \neq \text{SKIP} \land \)
\((s = t_1 (\subseteq \text{sources} ((\text{bvars} \; b) \neq \text{flow cfs}) \; s \; x) \longrightarrow \)
\(\text{run-flow} (\text{flow cfs}) \; s \; x = t_2 \; x\)
by \(\text{blast}\)
\}
moreover \{ 
fix \(s' \; \text{cfs} \; \text{cfs}'\)
assume 
\(V: \text{length cfs}' < \text{length cfs}_2 - \text{Suc} \; 0\) \; and 
\(W: (c, \; s) \rightarrow^* \{\text{cfs}\} \; (\text{SKIP}, \; s')\) \; and 
\(X: (\text{WHILE} \; b \; \text{DO} \; c, \; s') \rightarrow^* \{\text{cfs}'\}\)
\((c_2, \; \text{run-flow} (\text{flow cfs}') \; (\text{run-flow} (\text{flow cfs}) \; s))\)
then obtain \(c_2'\) and \(t_2\) where \(\forall x.\)
\((s = t_1 (\subseteq \text{sources-aux} (\text{flow cfs}) \; s \; x) \longrightarrow \)
\((c, \; t_1) \rightarrow^* (c_2', \; t_2) \land (\text{SKIP} = \text{SKIP}) = (c_2' = \text{SKIP}) \land \)
\((s = t_1 (\subseteq \text{sources} (\text{flow cfs}) \; s \; x) \longrightarrow s' \; x = t_2 \; x)\)
using \(B\) \([\text{of} \; B_1 \; C \; B_1' \; D' \; s \; ]\) \; \text{c s cfs SKIP s'}\)
and \(N\) and \(T\) by \(\text{force}\)
moreover have \(Y: s' = \text{run-flow} (\text{flow cfs}) \; s\)
using \(W\) by \((\text{rule small-stepsl-run-flow})\)
ultimately have \(Z: \forall x.\)
\((s = t_1 (\subseteq \text{sources-aux} (\text{flow cfs}) \; s \; x) \longrightarrow \)
\((c, \; t_1) \rightarrow^* (\text{SKIP}, \; t_2)) \land \)
\((s = t_1 (\subseteq \text{sources} (\text{flow cfs}) \; s \; x) \longrightarrow \)
\(\text{run-flow} (\text{flow cfs}) \; s \; x = t_2 \; x\)
by \(\text{blast}\)
assume \(s_2 = \text{run-flow} (\text{flow cfs}') \; (\text{run-flow} (\text{flow cfs}) \; s)\)
moreover have \((c, \; s) \Rightarrow s'\)
using \(W\) by \((\text{auto dest: small-stepsl-steps simp: big-iff-small})\)
hence \(s' \in \text{Univ} \; C \; (\subseteq \text{state} \cap \; Y)\)
using \(M\) and \(S\) by \(\text{blast}\)
ultimately obtain \(c_3'\) and \(t_3\) where \(AA: \forall x.\)
\((\text{run-flow} (\text{flow cfs}) \; s \; t_2)\)
\[
(\subseteq \text{sources-aux} (\text{flow cfs}')) (\text{run-flow} (\text{flow cfs}) s) x) \rightarrow \\
(\text{WHILE } b \text{ DO } c, t_2) \rightarrow^\ast (c_3', t_3) \land \\
(c_2 = \text{SKIP}) = (c_3' = \text{SKIP}) \land \\
(\text{run-flow} (\text{flow cfs}) s = t_2) \\
(\subseteq \text{sources} (\text{flow cfs}') (\text{run-flow} (\text{flow cfs}) s) x) \rightarrow \\
\text{run-flow} (\text{flow cfs}') (\text{run-flow} (\text{flow cfs}) s) x = t_3 x) \\
\] using \( K [\text{of cfs}' \sqsubset cfs' \text{ WHILE } b \text{ DO } c s'] \)
and \( V \text{ and } X \text{ and } Y \) by force

\[
\{
\text{fix } x \\
\text{assume } AB: s = t_1 \\
(\subseteq \text{sources-aux} ((\text{bvars } b) \# \text{flow cfs} @ \text{flow cfs}') s) x) \\
\text{moreover have sources-aux (flow cfs) s x } \subseteq \\
\text{sources-aux (flow cfs @ flow cfs') s x} \\
\text{by (rule sources-aux-append)} \\
\text{moreover have AC: sources-aux (flow cfs @ flow cfs') s x } \subseteq \\
\text{sources-aux ((bvars b) # flow cfs @ flow cfs’) s x} \\
\text{by (rule sources-aux-observe-tl)} \\
\text{ultimately have } (c, t_1) \rightarrow^\ast (\text{SKIP}, t_2) \\
\text{using } Z \text{ by blast} \\
\text{hence } (c; \text{ WHILE } b \text{ DO } c, t_1) \rightarrow^\ast (\text{SKIP};, \text{ WHILE } b \text{ DO } c, t_2) \\
\text{by (rule star-seq2)} \\
\text{moreover have } s = t_1 (\subseteq \text{bvars } b) \\
\text{using } Q \text{ and } AB \text{ by } (\text{blast dest: sources-aux-observe-hd}) \\
\text{hence } \text{bval } b t_1 \\
\text{using } S \text{ by (blast dest: bvars-bval)} \\
\text{hence } (\text{IF } b \text{ THEN } c; \text{ WHILE } b \text{ DO } c \text{ ELSE SKIP}, t_1) \rightarrow \\
(c; \text{ WHILE } b \text{ DO } c, t_1) .. \\
\text{ultimately have } (\text{IF } b \text{ THEN } c; \text{ WHILE } b \text{ DO } c \text{ ELSE SKIP}, t_1) \rightarrow^\ast \\
(\text{WHILE } b \text{ DO } c, t_2) \\
\text{by (blast intro: star-trans)} \\
\text{moreover have } \text{run-flow} (\text{flow cfs}) s = t_2 \\
(\subseteq \text{sources-aux} (\text{flow cfs}') (\text{run-flow} (\text{flow cfs}) s) x) \\
\text{proof} \\
\text{fix } y \\
\text{assume } y \in \text{sources-aux} (\text{flow cfs}') \\
(\text{run-flow} (\text{flow cfs}) s) x \\
\text{hence } \text{sources} (\text{flow cfs}) s \ y \subseteq \\
\text{sources-aux (flow cfs @ flow cfs') s x} \\
\text{by (rule sources-aux-member)} \\
\text{hence } \text{sources} (\text{flow cfs}) s \ y \subseteq \\
\text{sources-aux ((bvars b) # flow cfs @ flow cfs’) s x} \\
\text{using AC by simp} \\
\text{thus } \text{run-flow} (\text{flow cfs}) s \ y = t_2 y \\
\text{using } Z \text{ and } AB \text{ by blast} \\
\text{qed} \\
\text{hence } (\text{WHILE } b \text{ DO } c, t_2) \rightarrow^\ast (c_3', t_3) \land \\
(c_2 = \text{SKIP}) = (c_3' = \text{SKIP}) \\
\text{using AA by simp}
ultimately have
(IF $b$ THEN $c$; WHILE $b$ DO $c$ ELSE SKIP, $t_1$) $\rightarrow^*$
\[(c_3', t_3) \land (c_2 = \text{SKIP}) = (c_3' = \text{SKIP})\]
by (blast intro: star-trans)

moreover {
fix $x$
assume $AB$: $s = t_1$
(\subseteq sources ((bvars $b$) \# flow cfs @ flow cfs') $s$ $x$)
have $\text{run-flow} (flow cfs) s = t_2$
(\subseteq sources (flow cfs') (run-flow (flow cfs) $s$) $x$)
proof
fix $y$
assume $y \in$ sources (flow cfs')
(run-flow (flow cfs) $s$) $x$
hence sources (flow cfs) $s$ $y \subseteq$
sources (flow cfs @ flow cfs') $s$ $x$
by (rule sources-member)
moreover have sources (flow cfs) $s$ $y \subseteq$
sources ((bvars $b$) \# flow cfs @ flow cfs') $s$ $x$
by (rule sources-observe-tl)
ultimately have sources (flow cfs) $s$ $y \subseteq$
sources ((bvars $b$) \# flow cfs @ flow cfs') $s$ $x$
by simp
thus $\text{run-flow} (flow cfs) s$ $y = t_3$ $y$
using Z and $AB$ by blast
qed
hence $\text{run-flow} (flow cfs') (\text{run-flow} (flow cfs) s)$ $x = t_3$ $x$
using AA by simp
}
ultimately have $\exists c_3', t_3$. \forall $x$.
\[(s = t_1) \rightarrow \text{run-flow} (flow cfs') (\text{run-flow} (flow cfs) s)$ $x = t_3$ $x$\]
by auto
}
ultimately show $?\text{thesis}$
using $R$ by (auto simp: run-flow-append)
next
assume
$S$: \neg bval $b$ $s$ \text{ and}
$T$: flow cfs$_2$ = (bvars $b$) \# flow (tl cfs$_2$)
assume (SKIP, $s$) $\rightarrow^*$(tl cfs$_2$) ($c_2$, $s_2$)
hence $U$: ($c_2$, $s_2$) = (SKIP, $s$) \land flow (tl cfs$_2$) = []
by (rule small-steps-skip)
show thesis

proof (rule exI [of \ - SKIP], rule exI [of \ - t1])

{ fix x
  assume s = t1 (\ sources-aux [(bears b)] s x)
  hence s = t1 (\ bears b)
  using Q by (blast dest: sources-aux-observe-hd)
  hence \ bval b t1
  using S by (blast dest: bvars-bval)
  hence (IF b THEN c;; WHILE b DO c ELSE SKIP, t1) \to
  (SKIP, t1) ..
}

moreover {
  fix x
  assume s = t1 (\ sources [(bears b)] s x)
  hence s = t1 x
  by (subst (asm) append-Nil [symmetric], simp only: sources.simps, auto)
}

ultimately show \ x.
  (s1 = t1 (\ sources-aux (flow cfs2) s1 x) \to
  (c1, t1) \to* (SKIP, t1) \land (c2 = SKIP) = (SKIP = SKIP)) \land
  (s1 = t1 (\ sources (flow cfs2) s1 x) \to s2 x = t1 x)
  using R and T and U by auto

qed

next

assume R: bval b s

with F and O and P have S: s \in Univ B1' (\ state \cap Y)

by (drule-tac btyping2-approx [where s = s], auto)

assume (c;; WHILE b DO c, s) \to*{tl2 cfs1} (c1, s1)

hence
  (\ exists c' cfs'. c1 = c'; WHILE b DO c \land
  (c, s) \to*{cfs'} (c', s1) \land
  flow (tl2 cfs1) = flow cfs') \lor
  (\ exists s' cfs''. length cfs'' < length (tl2 cfs1) \land
  (c, s) \to*{cfs'} (SKIP, s') \land
  (WHILE b DO c, s') \to*{cfs''} (c1, s1) \land
  flow (tl2 cfs1) = flow cfs' @ flow cfs'')

by (rule small-steps-seq)

moreover {
  fix c' cfs
  assume
    T: (c, s) \to*{cfs} (c', s1) and
    U: c1 = c'; WHILE b DO c
  hence V: (c';; WHILE b DO c, s1) \to*{cfs2} (c2, s2)

  using J by simp

  hence W: s2 = run-flow (flow cfs2) s1

  by (rule small-steps-run-flow)
have
(\exists c'' \text{ cfs}'', c_2 = c''; \text{ WHILE } b \text{ DO } c \wedge
(c', s_1) \rightarrow\{\text{cfs}'\} (c'', s_2) \wedge
\text{flow cfs}_2 = \text{flow cfs}') \vee
(\exists s' \text{ cfs}' \text{ cfs}''; \text{ length cfs}'' < \text{ length cfs}_2 \wedge
(c', s_1) \rightarrow\{\text{cfs}'\} (\text{SKIP}, s') \wedge
(\text{WHILE } b \text{ DO } c, s') \rightarrow\{\text{cfs}''\} (c_2, s_2) \wedge
\text{flow cfs}_2 = \text{flow cfs}' \& \text{flow cfs}''\}
\text{using } \psi \text{ by (rule small-stepsl-seq)}
\text{moreover } \{
\text{fix } c'' \text{ cfs}''
\text{assume } (c', s_1) \rightarrow\{\text{cfs}'\} (c'', s_2)
\text{then obtain } c_2' \text{ and } t_2 \text{ where } X: \forall x.
(s_1 = t_1 (\subseteq \text{sources-aux (flow cfs)} s_1 x) \rightarrow
(c', t_1) \rightarrow (c_2', t_2) \wedge (c'' = \text{SKIP}) = (c_2' = \text{SKIP}) \wedge
(s_1 = t_1 (\subseteq \text{sources (flow cfs)} s_1 x) \rightarrow
\text{run-flow (flow cfs}_2) s_1 x = t_2 x
\text{using } B [\text{of } B_1 \text{ C } B_1' \text{ D' s cfs } c_1 \text{ cfs}' c''
\text{run-flow (flow cfs}_2) s_1] \text{ and } N \text{ and } S \text{ and } T \text{ and } W \text{ by force}
\text{assume } Y: c_2 = c''; \text{ WHILE } b \text{ DO } c \text{ and}
Z: \text{flow cfs}_2 = \text{flow cfs}'
\text{have } \theta\text{thesis}
\text{proof } (\text{rule exI [of - c_2''; \text{ WHILE } b \text{ DO } c], rule exI [of - t_2]})
\text{from } U \text{ and } W \text{ and } X \text{ and } Y \text{ and } Z \text{ show } \forall x.
(s_1 = t_1 (\subseteq \text{sources-aux (flow cfs)} s_1 x) \rightarrow
(c_1, t_1) \rightarrow (c_2, t_2) \wedge (c'' = \text{SKIP}) = (c_2' = \text{SKIP}) \wedge
(s_1 = t_1 (\subseteq \text{sources (flow cfs)} s_1 x) \rightarrow
s_2 x = t_2 x
\text{by (rule intro: star-seq2)}
\text{qed}
\text{moreover } \{
\text{fix } s' \text{ cfs}' \text{ cfs}''
\text{assume } X: \text{length cfs}'' < \text{ length cfs}_2 \text{ and}
Y: (c', s_1) \rightarrow\{\text{cfs}'\} (\text{SKIP}, s') \text{ and}
Z: (\text{WHILE } b \text{ DO } c, s') \rightarrow\{\text{cfs}''\} (c_2, s_2)
\text{then obtain } c_3' \text{ and } t_2 \text{ where } \forall x.
(s_1 = t_1 (\subseteq \text{sources-aux (flow cfs)} s_1 x) \rightarrow
(c', t_1) \rightarrow (c_2', t_2) \wedge (\text{SKIP} = \text{SKIP}) = (c_2' = \text{SKIP}) \wedge
(s_1 = t_1 (\subseteq \text{sources (flow cfs)} s_1 x) \rightarrow
s' x = t_2 x
\text{using } B [\text{of } B_1 \text{ C } B_1' \text{ D' s cfs } c_1 \text{ cfs}' \text{SKIP s}]
\text{and } N \text{ and } S \text{ and } T \text{ by force}
\text{moreover have } AA: s' = \text{run-flow (flow cfs)} s_1
\text{using } Y \text{ by (rule small-stepsl-run-flow)}
\text{ultimately have } AB: \forall x.
(s_1 = t_1 (\subseteq \text{sources-aux (flow cfs)} s_1 x) \rightarrow
(c', t_1) \rightarrow (\text{SKIP}, t_2) \wedge
127
moreover assume \( \text{flow } \text{cfs} \) \( s_1 \) \( x = t_2 \) \( x \)

by blast

have \( AC \): \( s_2 = \text{run-flow } (\text{flow } \text{cfs}'') s' \)

using \( Z \) by (rule small-steps-run-flow)

moreover have \( (c, s) \xrightarrow{} \{ \text{cfs } \@ \text{cfs}' \} (\text{SKIP}, s') \)

using \( T \) and \( Y \) by (simp add: small-steps-append)

hence \( (c, s) \Rightarrow s' \)

by (auto dest: small-steps-steps simp: big-iff-small)

hence \( s' \in \text{Univ } C (\subseteq \text{state } \cap Y) \)

using \( M \) and \( R \) by blast

ultimately obtain \( c_2' \) and \( t_3 \) where \( AD \): \( \forall x. \)

(run-flow (flow cfs') \( s_1 = t_2 \)

(\( \subseteq \) sources-aux (flow cfs'') (run-flow (flow cfs') \( s_1 \)) \( x \)) \( \rightarrow \)

(\( \text{WHILE } b \text{ DO } c, t_2 \) \( \rightarrow \text{* } (c_2', t_3) \) \( \land \)

\( (c_2 = \text{SKIP}) = (c_2' = \text{SKIP}) \) \( \land \)

(run-flow (flow cfs') \( s_1 = t_2 \)

(\( \subseteq \) sources (flow cfs'') (run-flow (flow cfs') \( s_1 \)) \( x \)) \( \rightarrow \)

run-flow (flow cfs'') (run-flow (flow cfs') \( s_1 \)) \( x = t_3 \) \( x \)

using \( K \) [of cfs''] [of cfs'' s' WHILE b DO c s']

and \( X \) and \( Z \) and \( AA \) by force

moreover assume \( \text{flow } cfs_2 = \text{flow } cfs' \@ \text{flow } cfs'' \)

moreover \{ fix \( x \)

assume \( AE \): \( s_1 = t_1 \)

(\( \subseteq \) sources-aux (flow cfs' @ flow cfs'') \( s_1 \)) \( x \)

moreover have sources-aux (flow cfs') \( s_1 \) \( x \subseteq \)

souces-aux (flow cfs@flow cfs'') \( s_1 \)

by (rule sources-aux-append)

ultimately have \( (c', t_1) \xrightarrow{} (\text{SKIP}, t_2) \)

using \( AB \) by blast

hence \( (c'' ; \text{WHILE } b \text{ DO } c, t_1) \xrightarrow{} (\text{SKIP} ; \text{WHILE } b \text{ DO } c, t_2) \)

by (rule star-seq2)

hence \( (c'' ; \text{WHILE } b \text{ DO } c, t_1) \xrightarrow{} (\text{WHILE } b \text{ DO } c, t_2) \)

by (blast intro: star-trans)

moreover have \( \text{run-flow } (\text{flow cfs'}) \( s_1 = t_2 \)

(\( \subseteq \) sources-aux (flow cfs'') (run-flow (flow cfs') \( s_1 \)) \( x \)

proof

fix \( y \)

assume \( y \in \text{sources-aux } (\text{flow cfs''}) \)

(run-flow (flow cfs') \( s_1 \)) \( x \)

hence sources (flow cfs') \( s_1 \) \( y \subseteq \)

sources-aux (flow cfs@flow cfs'') \( s_1 \)

by (rule sources-aux-member)

thus \( \text{run-flow } (\text{flow cfs'}) \( s_1 \) \( y = t_2 \) \( y \)

using \( AB \) and \( AE \) by blast

qed

hence \( (\text{WHILE } b \text{ DO } c, t_2) \xrightarrow{} (c_2', t_3) \)

\( (c_2 = \text{SKIP}) = (c_2' = \text{SKIP}) \)

\( \text{128} \)
ultimately have \((c_1; \text{WHILE } b \text{ DO } c, t_1) \rightarrow^* (c_2', t_3) \land (c_2 = \text{SKIP}) = (c_2' = \text{SKIP})\)
by (blast intro: star-trans)

moreover {
  fix \(x\)
  assume \(AE: s_1 = t_1\)
  \((\subseteq \text{sources} (\text{flow cfs}' \@ \text{flow cfs}'') s_1 x)\)
  have \(\text{run-flow} (\text{flow cfs}') s_1 = t_2\)
  \((\subseteq \text{sources} (\text{flow cfs}'') (\text{run-flow} (\text{flow cfs}') s_1) x)\)
proof
  fix \(y\)
  assume \(y \in \text{sources} (\text{flow cfs}'')\)
  \((\text{run-flow} (\text{flow cfs}') s_1) x\)
  hence \(\text{sources} (\text{flow cfs}') s_1 y \subseteq \text{sources} (\text{flow cfs}' \@ \text{flow cfs}'') s_1 x\)
  by (rule sources-member)
  thus \(\text{run-flow} (\text{flow cfs}') s_1 y = t_2 y\)
  using \(AB\) and \(AE\) by blast
qed

hence \(\text{run-flow} (\text{flow cfs}'')\)
\((\text{run-flow} (\text{flow cfs}') s_1) x = t_3 x\)
using \(AD\) by \(simp\)

ultimately have ?thesis
by (metis U AA AC)
}

ultimately have ?thesis
by blast

moreover {
  fix \(s'\) \(\text{cfs} cfs'\)
  assume
  \(\text{length} \ cfs' < \text{length} (\text{tl2} \ cfs_1)\) and
  \((c, s) \rightarrow^* \{cfs\} \{\text{SKIP}, \ s'\}\) and
  \((\text{WHILE } b \text{ DO } c, s') \rightarrow^* \{cfs'\} \{c_1, s_1\}\)
moreover from this have \((c, s) \Rightarrow s'\)
  by (auto dest: small-steps-steps simp: big-iff-small)
  hence \(s' \in \text{Univ C} \subseteq \text{state} \cap \text{Y}\)
  using \(M\) and \(R\) by blast
ultimately have ?thesis
  using \(K[\text{of} \ cfs' \@ \ cfs_2 \ cfs' \ cfs_2 \ s' \ c_1 \ s_1]\) and \(J\) by force
}

ultimately show ?thesis
by blast

next
  assume \((\text{SKIP}, s) \rightarrow^* (\text{tl2} \ cfs_1) \{c_1, s_1\}\)
  hence \((c_1, s_1) = (\text{SKIP}, s)\)
by (blast dest: small-steps-skip)
moreover from this have \((c_2, s_2) = \text{(SKIP, s) \land flow cfs_2 = \_}\)
using \(J\) by (blast dest: small-steps-skip)
ultimately show \(\varepsilon\)thesis
by auto
qed

\(\{\}
ultimately show
\((\forall t_1, \exists c_2' t_2, \forall x. \)
\((s_1 = t_1 \ (\subseteq \text{sources-aux (flow cfs}_2\text{) s}_1\text{ x)} \rightarrow \)
\((c_1, t_1) \rightarrow (c_2', t_2) \land (c_2 = \text{SKIP}) = (c_2' = \text{SKIP}) \land \)
\((s_1 = t_1 \ (\subseteq \text{sources (flow cfs}_2\text{) s}_1\text{ x)} \rightarrow \ s_2\text{ x} = t_2\text{ x}) \land \)
\((\forall x. (\exists (B, Y) \in U. \exists s \in B. \exists y \in Y. \Rightarrow s\text{: dom } y \sim \text{ dom x}) \rightarrow \)

no-upd (flow cfs\text{2\_}x)
using \(L\) by auto
qed

lemma ctyping2-correct-aux:
\([U, v] \models c \ (\subseteq A, X) = \text{Some (B, Y)}; s \in \text{Univ A (\subseteq state \cap X)}; \)
\((c, s) \rightarrow^[\{cfs\text{1 \_}x\} (c_1, s_1); (c_1, s_1) \rightarrow^[\{cfs\text{2 \_}x\} (c_2, s_2)] \)
ok-flow-aux \(U \ c_1\ c_2\ s_1\ s_2\ (flow cfs\text{2\_}x)\)
proof (induction \((U, v) c A X\) arbitrary; \(B Y U v s c_1\ c_2\ s_1\ s_2\ cfs\text{1 \_}cfs\text{2\_}x\)
rule: ctyping2.induct)
fix \(A X C Z U v c_1\ c_2\ c' c'' s s_1\ s_2\ cfs\text{1 \_}cfs\text{2\_}x\)
show
\([\forall B Y s c' c'' s_1 s_2\ cfs\text{1 \_}cfs\text{2\_}x\)
\((U, v) \models c_1 \ (\subseteq A, X) = \text{Some (B, Y)} \Rightarrow \)
\(s \in \text{Univ A (\subseteq state \cap X)} \Rightarrow \)

(c_1, s) \rightarrow^[\{cfs\text{1 \_}x\} (c_1, s_1) \Rightarrow \)
\((c', s_1) \rightarrow^[\{cfs\text{2 \_}x\} (c''_s, s_2) \Rightarrow \)
\((\forall t_1, \exists c_2' t_2, \forall x. \)
\((s_1 = t_1 \ (\subseteq \text{sources-aux (flow cfs}_2\text{) s}_1\text{ x)} \rightarrow \)
\((c', t_1) \rightarrow^[\{cfs\text{1 \_}x\} (c', t_2) \land (c'' = \text{SKIP}) = (c_2' = \text{SKIP}) \land \)
\((s_1 = t_1 \ (\subseteq \text{sources (flow cfs}_2\text{) s}_1\text{ x)} \rightarrow \ s_2\text{ x} = t_2\text{ x}) \land \)

\((\forall x. (\exists (B, W) \in U. \exists s \in B. \exists y \in W. \Rightarrow s\text{: dom } y \sim \text{ dom x}) \rightarrow \)
no-upd (flow cfs\text{2\_}x))

\(\land p B Y C Z s c' c'' s_1 s_2\ cfs\text{1 \_}cfs\text{2\_}x\)
\((U, v) \models c_1 \ (\subseteq A, X) = \text{Some } p \Rightarrow \)
\((B, Y) = p \Rightarrow \)

\((U, v) \models c_2 \ (\subseteq B, Y) = \text{Some (C, Z)} \Rightarrow \)
\(s \in \text{Univ B (\subseteq state \cap Y)} \Rightarrow \)

(c_2, s) \rightarrow^[\{cfs\text{1 \_}x\} (c', s_1) \Rightarrow \)
\((c', s_1) \rightarrow^[\{cfs\text{2 \_}x\} (c''_s, s_2) \Rightarrow \)
\((\forall t_1, \exists c_2'' t_2, \forall x. \)
\((s_1 = t_1 \ (\subseteq \text{sources-aux (flow cfs}_2\text{) s}_1\text{ x)} \rightarrow \)
\((c', t_1) \rightarrow^[\{cfs\text{1 \_}x\} (c', t_2) \land (c'' = \text{SKIP}) = (c_2'' = \text{SKIP}) \land \)

\((s_1 = t_1 \ (\subseteq \text{sources (flow cfs}_2\text{) s}_1\text{ x)} \rightarrow \ s_2\text{ x} = t_2\text{ x}) \land \)

\((\forall x. (\exists (B, W) \in U. \exists s \in B. \exists y \in W. \Rightarrow s\text{: dom } y \sim \text{ dom x}) \rightarrow \)

130
(U, v) |= c₁ :: c₂ (⊆ A, X) = Some (C, Z);

(by (auto del: conjI split: option.split_asm,
rule ctyping2-correct-asm)

next

fix A X C Y U v b c₁ c₂ c¹ c² s s¹ s² cf₁ cf₂

show

∀ U' p B₁ B₂ C Y s c¹ c² s¹ s₂ cf₁ cf₂.

(U', p) = (insert (Univ? A X, bvars b) U, |= b (⊆ A, X)) =>

(B₁, B₂) = p =>

(U', v) |= c₁ (⊆ B₁, X) = Some (C, Y) =>

s ∈ Univ B₁ (⊆ state ∩ X) =>

(c₁, s) => cf₁ (c¹, s₁) =>

(c², s₁) => cf₂ (c''², s₂) =>

∀ t₁, ∃ c²₁ t₂. ∀ x.

(s₁ = t₁ (⊆ sources-aux (flow cf₂) s₁ x) =>

(c¹, t₁) => (c²₁, t₂) ∧ (c''² = SKIP) = (c²₁ = SKIP)) ∧

(s₁ = t₁ (⊆ sources (flow cf₂) s₁ x) => s₂ x = t₂ x)) ∧

(∀ x. (∃ (B, W) ∈ U'. ∃ s ∈ B. ∃ y ∈ W. ¬ s: dom y ≠ dom x) =>

no-upd (flow cf₂) x)

∀ U' p B₁ B₂ C Y s c¹ c² s¹ s₂ cf₁ cf₂.

(U', p) = (insert (Univ? A X, bvars b) U, |= b (⊆ A, X)) =>

(B₁, B₂) = p =>

(U', v) |= c₂ (⊆ B₂, X) = Some (C, Y) =>

s ∈ Univ B₂ (⊆ state ∩ X) =>

(c₂, s) => cf₁ (c¹, s₁) =>

(c², s₁) => cf₂ (c''², s₂) =>

∀ t₁, ∃ c²'' t₂. ∀ x.

(s₁ = t₁ (⊆ sources-aux (flow cf₂) s₁ x) =>

(c¹, t₁) => (c²'', t₂) ∧ (c''² = SKIP) = (c²'' = SKIP)) ∧

(s₁ = t₁ (⊆ sources (flow cf₂) s₁ x) => s₂ x = t₂ x)) ∧

(∀ x. (∃ (B, W) ∈ U'. ∃ s ∈ B. ∃ y ∈ W. ¬ s: dom y ≠ dom x) =>

no-upd (flow cf₂) x)

(U', v) |= IF b THEN c₁ ELSE c₂ (⊆ A, X) = Some (C, Y);

s ∈ Univ A (⊆ state ∩ X);

(IF b THEN c₁ ELSE c₂, s) => cf₁ (c¹, s₁);

(c², s₁) => cf₂ (c''², s₂) =>

∀ t₁, ∃ c²₁ t₂. ∀ x.

(s₁ = t₁ (⊆ sources-aux (flow cf₂) s₁ x) =>

(c¹, t₁) => (c²₁, t₂) ∧ (c''² = SKIP) = (c²₁ = SKIP)) ∧

no-upd (flow cf₂) x)

131
\[(s_1 = t_1 (\subseteq \text{sources} (\text{flow } cfs_2) s_1 x) \rightarrow s_2 x = t_2 x)) \land
(\forall x. (\exists (B, W) \in U. \exists s \in B. \exists y \in W. \neg s \cdot \text{dom } y \rightsquigarrow \text{dom } x) \rightarrow
\text{no-upd} (\text{flow } cfs_2) x)\]

by (auto del: concl split: option.split_asm prod.split_asm, rule typing2-correct-aux-if)

next

fix \(A X B Y U v b c c_1 c_2 s s_1 s_2 cfs_1 cfs_2\)

show

\(\bigwedge B_1 B_2 C Y B_1' B_2' D Z s c_1 c_2 s_1 s_2 cfs_1 cfs_2\).

\((B_1, B_2) = |\ b (\subseteq A, X) \Rightarrow
(C, Y) = \vdash c (\subseteq B_1, X) \Rightarrow
(B_1', B_2') = |\ b (\subseteq C, Y) \Rightarrow
\forall (B, W) \in \text{insert (Univ? A X} \cup \text{Univ? C Y), bvars b} U.
B: \text{dom } \cdot W \rightarrow \text{UNIV} \Rightarrow
\{\}, \text{False} \vdash c (\subseteq B_1, Y) = \text{Some (D, Z) \Rightarrow
s \in \text{Univ B_1} (\subseteq \text{state } \& \ X) \Rightarrow
(c, s) \rightarrow*\{cfs_1\} (c_1, s_1) \Rightarrow
(c_1, s_1) \rightarrow*\{cfs_2\} (c_2, s_2) \Rightarrow
\forall t_1, \exists c_2' t_2. \forall B_1.
\)

\((s_1 = t_1 (\subseteq \text{sources-aux} (\text{flow } cfs_2) s_1 B_1) \rightarrow
(c_1, t_1) \rightarrow* (c_2', t_2) \land (c_2 = \text{SKIP}) = (c_2' = \text{SKIP}) \land
(s_1 = t_1 (\subseteq \text{sources} (\text{flow } cfs_2) s_1 B_1) \rightarrow s_2 B_1 = t_2 B_1)) \land
\forall x. (\exists (B, W) \in \{\}. \exists s \in B. \exists y \in W. \neg s \cdot \text{dom } y \rightsquigarrow \text{dom } x) \rightarrow
\text{no-upd} (\text{flow } cfs_2) x)\);
no-upd (flow cfs₂) x
by (auto del: conjI split: option.split-asm prod.split-asm,
rule ctyping2-correct-aux-while, assumption+, blast)
qed (auto del: conjI split: prod.split-asm)

theorem ctyping2-correct:
assumes \( A: (U, v) \models c (\subseteq A, X) = Some (B, Y) \)
shows correct c A X
proof –

\{ 
fix c₁ c₂ s₁ s₂ cfs t₁
assume ok-flow-aux U c₁ c₂ s₁ s₂ (flow cfs)
then obtain c₂' and t₂ where A: \( \forall x. \)
\( (s₁ = t₁ (\subseteq sources-aux (flow cfs) s₁ x) \rightarrow \)
\( (c₁, t₁) \rightarrow^* (c₂', t₂) \land (c₂ = SKIP) = (c₂' = SKIP) \) \land
\( (s₁ = t₁ (\subseteq sources (flow cfs) s₁ x) \rightarrow s₂ x = t₂ x) \)
by blast
have \( \exists c₂' t₂. \forall x. s₁ = t₁ (\subseteq sources (flow cfs) s₁ x) \rightarrow \)
\( (c₁, t₁) \rightarrow^* (c₂', t₂) \land (c₂ = SKIP) = (c₂' = SKIP) \) \land \( s₂ x = t₂ x \)
proof (rule exI [of - c₂'], rule exI [of - t₂])
have \( \forall x. s₁ = t₁ (\subseteq sources (flow cfs) s₁ x) \rightarrow \)
\( s₁ = t₁ (\subseteq sources-aux (flow cfs) s₁ x) \)
proof (rule allI, rule impI)
fix x
assume \( s₁ = t₁ (\subseteq sources (flow cfs) s₁ x) \)
moreover have sources-aux (flow cfs) s₁ x \( \subseteq \)
sources (flow cfs) s₁ x
by (rule sources-aux-sources)
ultimately show \( s₁ = t₁ (\subseteq sources-aux (flow cfs) s₁ x) \)
by blast
qed
with A show \( \forall x. s₁ = t₁ (\subseteq sources (flow cfs) s₁ x) \rightarrow \)
\( (c₁, t₁) \rightarrow^* (c₂', t₂) \land (c₂ = SKIP) = (c₂' = SKIP) \) \land \( s₂ x = t₂ x \)
by auto
qed
\}
with A show ?thesis
by (clarsimp dest!: small-steps-steps1 simp: correct-def,
drule_tac ctyping2-correct-aux, auto)
qed
end
end
5 Degeneracy to stateless level-based information flow control

theory Degeneracy
  imports Correctness HOL–IMP.Sec-TypingT
begin

The goal of this concluding section is to prove the degeneracy of the information flow correctness notion and the static type system defined in this paper to the classical counterparts addressed in [7], section 9.2.6, and formalized in [5] and [6], in case of a stateless level-based information flow correctness policy.

First of all, locale noninterf is interpreted within the context of the class sec defined in [5], as follows.

- Parameter dom is instantiated as function sec, which also sets the type variable standing for the type of the domains to nat.
- Parameter interf is instantiated as the predicate such that for any program state, the output is True just in case the former input level may interfere with, namely is not larger than, the latter one.
- Parameter state is instantiated as the empty set, consistently with the fact that the policy is represented by a single, stateless interference predicate.

Next, the information flow security notion implied by theorem noninterference in [6] is formalized as a predicate secure taking a program as input. This notion is then proven to be implied, in the degenerate interpretation described above, by the information flow correctness notion formalized as predicate correct (theorem correct-secure). Particularly:

- This theorem demands the additional assumption that the state set A input to correct is nonempty, since correct is vacuously true for A = {}.
- In order for this theorem to hold, predicate secure needs to slightly differ from the information flow security notion implied by theorem noninterference, in that it requires state t’ to exist if there also exists some variable with a level not larger than l, namely if condition s = t (≤ l) is satisfied nontrivially – actually, no leakage may arise from two initial states disagreeing on the value of every variable. In fact, predicate correct requires a nontrivial configuration (c2’, t2) to exist in case condition s1 = t1 (⊆ sources cs s1 x) is satisfied for some variable x.
Finally, the static type system \texttt{ctyping2} is proven to be equivalent to the \texttt{sec-type} one defined in [6] in the above degenerate interpretation (theorems \texttt{ctyping2-sec-type} and \texttt{sec-type-ctyping2}). The former theorem, which proves that a \textit{pass} verdict from \texttt{ctyping2} implies the issuance of a \textit{pass} verdict from \texttt{sec-type} as well, demands the additional assumptions that (a) the \textit{state set} input to \texttt{ctyping2} is nonempty, (b) the input program does not contain any loop with \textit{Bc True} as boolean condition, and (c) the input program has undergone constant folding, as addressed in [7], section 3.1.3 for arithmetic expressions and in [7], section 3.2.1 for boolean expressions. Why?

This need arises from the different ways in which the two type systems handle “dead” conditional branches. Type system \texttt{sec-type} does not try to detect “dead” branches; it simply applies its full range of information flow security checks to any conditional branch contained in the input program, even if it actually is a “dead” one. On the contrary, type system \texttt{ctyping2} detects “dead” branches whenever boolean conditions can be evaluated at compile time, and applies only a subset of its information flow correctness checks to such branches.

As parameter \textit{state} is instantiated as the empty set, boolean conditions containing variables cannot be evaluated at compile time, yet they can if they only contain constants. Thus, assumption (a) prevents \texttt{ctyping2} from handling the entire input program as a “dead” branch, while assumptions (b) and (c) ensure that \texttt{ctyping2} will not detect any “dead” conditional branch within the program. On the whole, those assumptions guarantee that \texttt{ctyping2}, like \texttt{sec-type}, applies its full range of checks to \textit{any} conditional branch contained in the input program, as required for theorem \texttt{ctyping2-sec-type} to hold.

\subsection*{5.1 Global context definitions and proofs}

\begin{verbatim}
fun cgood :: com ⇒ bool where
cgood (c1;; c2) = (cgood c1 ∧ cgood c2) |
cgood (IF - THEN c1 ELSE c2) = (cgood c1 ∧ cgood c2) |
cgood (WHILE b DO c) = (b $= Bc True ∧ cgood c) |
cgood - = True

fun seq :: com ⇒ com ⇒ com where
seq SKIP c = c |
seq c SKIP = c |
seq c1 c2 = c1;; c2

fun ifc :: bexp ⇒ com ⇒ com ⇒ com where
ifc (Bc True) c - = c |
ifc (Bc False) - c = c |
ifc b c1 c2 = (if c1 = c2 then IF b THEN c1 ELSE c2)
\end{verbatim}
fun while :: bexp ⇒ com ⇒ com where
while (Be False) = SKIP |
while b c = WHILE b DO c

primrec csimp :: com ⇒ com where
csimp SKIP = SKIP |
csimp (x ::= a) = x ::= asimp a |
csimp (c1;; c2) = seq (csimp c1) (csimp c2) |
csimp (IF b THEN c1 ELSE c2) = ifc (bsimp b) (csimp c1) (csimp c2) |
csimp (WHILE b DO c) = while (bsimp b) (csimp c)

lemma not-size:
  size (not b) ≤ Suc (size b)
by (induction b rule: not.induct, simp-all)

lemma and-size:
  size (and b1 b2) ≤ Suc (size b1 + size b2)
by (induction b1 b2 rule: and.induct, simp-all)

lemma less-size:
  size (less a1 a2) = 0
by (induction a1 a2 rule: less.induct, simp-all)

lemma bsimp-size:
  size (bsimp b) ≤ size b
by (induction b, auto intro: le-trans not-size and-size simp: less-size)

lemma seq-size:
  size (seq c1 c2) ≤ Suc (size c1 + size c2)
by (induction c1 c2 rule: seq.induct, simp-all)

lemma ifc-size:
  size (ifc b c1 c2) ≤ Suc (size c1 + size c2)
by (induction b c1 c2 rule: ifc.induct, simp-all)

lemma while-size:
  size (while b c) ≤ Suc (size c)
by (induction b c rule: while.induct, simp-all)

lemma csimp-size:
  size (csimp c) ≤ size c
by (induction c, auto intro: le-trans seq-size ifc-size while-size)

lemma avars-asimp:
  avars a = {} ⇒ ∃ i. asimp a = N i
by (induction a, auto)

lemma seq-match [dest!]:
\( \text{seq (csimp } c_1 \text{) (csimp } c_2 \text{)} = c_1 ; c_2 \implies \text{csimp } c_1 = c_1 \land \text{csimp } c_2 = c_2 \)
by (rule seq.cases [of (csimp } c_1 \text{, csimp } c_2 \text{)],
insert csimp-size [of } c_1 \text{], insert csimp-size [of } c_2 \text{], simp-all)

lemma ifc-match [dest!]:
\( \text{ifc (bsimp } b \text{) (csimp } c_1 \text{) (csimp } c_2 \text{)} = \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \implies \)
\( \text{bsimp } b = b \land (\forall v. b \neq Bc v) \land \text{csimp } c_1 = c_1 \land \text{csimp } c_2 = c_2 \)
by (insert csimp-size [of } c_1 \text{], insert csimp-size [of } c_2 \text{],
subgoal-tac csimp } c_1 \neq \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \text{, auto intro: ifc.cases}
[of (bsimp } b \text{, csimp } c_1 \text{, csimp } c_2 \text{)] split: if-split-asm)

lemma while-match [dest!]:
\( \text{while (bsimp } b \text{) (csimp } c \text{)} = \text{WHILE } b \text{ DO } c \implies \)
\( \text{bsimp } b = b \land b \neq Bc False \land \text{csimp } c = c \)
by (rule while.cases [of (bsimp } b \text{, csimp } c \text{)], auto)

5.2 Local context definitions and proofs

context sec
begin

interpretation noninterf λs. (≤) sec {}
by (unfold-locales, simp)

notation interf-set ⟨(\_ - _ - ) [51, 51, 51] 50⟩
notation univ-states-if ⟨(Univ? - _ - ) [51, 75] 75⟩
notation atyping ⟨(\_ - _ - (≤ - _ - ) [51, 51] 50)⟩
notation btyping2-aux ⟨(\_ - _ - (≤ - _ - ) [51, 51] 55)⟩
notation btyping2 ⟨(\_ - _ - (≤ - _ - ) [51, 51] 55)⟩
notation ctyping1 ⟨(\_ - _ - (≤ - _ - ) [51, 51] 55)⟩
notation ctyping2 ⟨(\_ - _ - (≤ - _ - ) [51, 51] 55)⟩

abbreviation eq-le-ext :: state ⇒ state ⇒ level ⇒ bool
\( (\_ = _ - (_ - _ - ) [51, 51, 0] 50) \) where
\( s = t (≤ l) \equiv s = t (≤ l) \land (\exists x :: vname. \text{sec } x ≤ l) \)

definition secure :: com ⇒ bool where
secure } c ≡ \forall s s₁ t l. (c, s) ⇒ s₁ ∧ s = t (≤ l) ---->
\( (\exists t'. (c, t) ⇒ t' \land s' = t' (≤ l)) \)

definition levels :: config set ⇒ level set where
levels } U ≡ insert 0 (sec \text{’} \bigcup (\text{snd } \text{’} ((B, Y) \in U. B \neq \{\}))))
lemma avars-finite:
  finite (avars a)
by (induction a, simp-all)

lemma avars-in:
n < sec a ==> sec a \in sec ' avars a
by (induction a, auto simp: max-def)

lemma avars-sec:
x \in avars a ==> sec x \leq sec a
by (induction a, auto)

lemma avars-ub:
sec a \leq l = (\forall x \in avars a. sec x \leq l)
by (induction a, auto)

lemma bvars-finite:
  finite (bvars b)
by (induction b, simp-all add: avars-finite)

lemma bvars-in:
n < sec b ==> sec b \in sec ' bvars b
by (induction b, auto dest!: avars-in simp: max-def)

lemma bvars-sec:
x \in bvars b ==> sec x \leq sec b
by (induction b, auto dest: avars-sec)

lemma bvars-ub:
sec b \leq l = (\forall x \in bvars b. sec x \leq l)
by (induction b, auto simp: avars-ub)

lemma levels-insert:
  assumes
    A: A \neq {} and
    B: finite (levels U)
  shows finite (levels (insert (A, bvars b) U)) ∧
    Max (levels (insert (A, bvars b) U)) = max (sec b) (Max (levels U))
    (is finite (levels ?U')) ∧ ?P
proof –
  have C: levels ?U' = sec ' bvars b \cup levels U
    using A by (auto simp: image-def levels-def univ-states-if-def)
  hence D: finite (levels ?U')
    using B by (simp add: bvars-finite)
  moreover have ?P
  proof (rule Max-eqI [OF D])
    fix l
assume \( l \in \text{levels} \) (\( \text{insert} (A, \text{bvars} b) U \))
thus \( l \leq \max \text{ (sec b) (Max (levels U))} \)
using \( C \) by (\( \text{auto dest: Max-ge [OF B] bvars-sec} \))

next

show \( \max \text{ (sec b) (Max (levels U))} \in \text{levels} \) (\( \text{insert} (A, \text{bvars} b) U \))
using \( C \) by (\( \text{insert Max-in [OF B], fastforce dest: bvars-in simp: max-def not-le levels-def} \))

qed

ultimately show \( ?\text{thesis} \).

qed

lemma \( \text{sources-le} \):
\( y \in \text{sources cs s x} \implies \text{sec y} \leq \text{sec x} \)
and \( \text{sources-aux-le} \):
\( y \in \text{sources-aux cs s x} \implies \text{sec y} \leq \text{sec x} \)
by (\( \text{induction cs s x} \) and \( \text{cs s x rule: sources-induct, auto split: com-flow,split-asn if-split-asn, fastforce+} \))

lemma \( \text{bsimp-btyping2-aux-not [intro]} \):
\( [\text{bsimp b} = b \implies \forall v. b \neq \text{Bc v} \implies \models b (\subseteq A, X) = \text{None}; \) \)
not (bsimp b) = Not b] \implies \models b (\subseteq A, X) = \text{None} \)
by (\( \text{rule not.cases [of bsimp b], auto} \))

lemma \( \text{bsimp-btyping2-aux-and [intro]} \):
\( \text{assumes} \)
\( A: [\text{bsimp b} = b_1; \forall v. b_1 \neq \text{Bc v} \implies \models b_1 (\subseteq A, X) = \text{None} \) \) and \( B: \text{and (bsimp b_1) (bsimp b_2) = And b_1 b_2} \)
shows \( \models b_1 (\subseteq A, X) = \text{None} \)
proof –
\{ 
assume \( \text{bsimp b_2} = \text{And} b_1 b_2 \)
hence \( \text{Bc True} = b_1 \)
by (\( \text{insert bsimp-size [of b_2], simp} \))
\}
moreover \{ 
assume \( \text{bsimp b_2} = \text{And} (\text{Bc True}) b_2 \)
hence False
by (\( \text{insert bsimp-size [of b_2], simp} \))
\}
moreover \{ 
assume \( \text{bsimp b_1} = \text{And} b_1 b_2 \)
hence False
by (\( \text{insert bsimp-size [of b_1], simp} \))
\}
ultimately have \( \text{bsimp b_1} = b_1 \land (\forall v. b_1 \neq \text{Bc v} \) \)
using \( B \) by (\( \text{auto intro: and.cases [of (bsimp b_1, bsimp b_2)]} \))
thus \( ?\text{thesis} \)
using \( A \) by simp

139
qed

lemma bsimp-btyping2-aux-less [elim]:

\[
\text{less (asimp } a_1 \text{) (asimp } a_2 \text{) = Less } a_1 \ a_2;\]

\[
\forall a_1 = \{\}; \ \forall a_2 = \{\} \implies \text{False}
\]

by (fastforce dest: avars-asimp)

lemma bsimp-btyping2-aux:

\[
[\text{bsimp } b = b; \ \forall v. \ b \neq Bc \ v] \implies \vert b \ (\subseteq A, X) = \text{None}
\]

by (induction b, auto split: option.split)

lemma bsimp-btyping2:

\[
[\text{bsimp } b = b; \ \forall v. \ b \neq Bc \ v] \implies \vert b \ (\subseteq A, X) = (A, A)
\]

by (auto dest: bsimp-btyping2-aux [of - A X] simp: btyping2-def)

lemma csimp-ctyping2-if:

\[
\forall U' \ B' \ U' = U \implies B = B_1 \implies \{\} = B' \implies B_1 \neq \{\} \implies \text{False}; \ s \in A;
\]

\[
\vert b \ (\subseteq A, X) = (B_1, B_2); \ \text{bsimp } b = b; \ \forall v. \ b \neq Bc \ \text{True}; \ b \neq Bc \ \text{False} \implies \text{False}
\]

by (drule bsimp-btyping2 [of - A X], auto)

lemma csimp-ctyping2-while:

\[
\forall (\text{if } P \ \text{then } \text{Some } (B_2' \cup B_2', Y) \ \text{else None} = \text{Some } (\{\}, Z); \ s \in A;
\]

\[
\vert b \ (\subseteq A, X) = (B_1, B_2); \ \text{bsimp } b = b; \ \forall v. \ b \neq Bc \ \text{True}; \ b \neq Bc \ \text{False} \implies \text{False}
\]

by (drule bsimp-btyping2 [of - A X], auto split: if-split-asm)

lemma csimp-ctyping2:

\[
\forall (U, v) \models c \ (\subseteq A, X) = \text{Some } (B, Y) \neq \{\}; \ \text{cgood } c; \ \text{csimp } c = c \implies \text{B } \neq \{\}
\]

proof (induction (U, v) c A X arbitrary; B Y U v rule: ctyping2.induct)

fix A X B Y U v c1 c2

show

\[
\forall B. \ (U, v) \models c_1 \ (\subseteq A, X) = \text{Some } (B, Y) \implies
\]

\[
A \neq \{\} \implies \text{cgood } c_1 \implies \text{csimp } c_1 = c_1 \implies
\]

\[
B \neq \{\};
\]

\[
\forall p \ B Y C Z. \ (U, v) \models c_1 \ (\subseteq A, X) = \text{Some } p \implies
\]

\[
(B, Y) = p \implies (U, v) \models c_2 \ (\subseteq B, Y) = \text{Some } (C, Z) \implies
\]

\[
B \neq \{\} \implies \text{cgood } c_2 \implies \text{csimp } c_2 = c_2 \implies
\]

\[
C \neq \{\};
\]

\[
(U, v) \models c_1 \cup c_2 \ (\subseteq A, X) = \text{Some } (B, Y);
\]

\[
A \neq \{\}; \ \text{cgood } (c_1 \cup c_2);
\]

\[
\text{csimp } (c_1 \cup c_2) = c_1 \cup c_2 \implies
\]

\[
B \neq \{\}
\]

by (fastforce split: option.split-asm)

next

fix A X C Y U v b c1 c2

show
\begin{align*}
\forall U' \ p \ B_1 \ B_2 \ C \ Y. \\
(U', p) &= (insert \ (Univ? \ A \ X, \ bvars \ b) \ U, \models b (\subseteq A, X)) \implies \\
(B_1, B_2) &= p \implies (U', v) \models c_1 (\subseteq B_1, X) = Some (C, Y) \implies \\
B_1 \neq \{\} \implies cgood c_1 \implies csimp c_1 = c_1 \implies \\
C \neq \{\}; \\
\forall U' \ p \ B_1 \ B_2 \ C \ Y. \\
(U', p) &= (insert \ (Univ? \ A \ X, \ bvars \ b) \ U, \models b (\subseteq A, X)) \implies \\
(B_1, B_2) &= p \implies (U', v) \models c_2 (\subseteq B_2, X) = Some (C, Y) \implies \\
B_2 \neq \{\} \implies cgood c_2 \implies csimp c_2 = c_2 \implies \\
C \neq \{\}; \\
(U, v) \models IF b THEN c_1 ELSE c_2 (\subseteq A, X) = Some (C, Y); \\
A \neq \{\}; cgood (IF b THEN c_1 ELSE c_2); \\
\begin{align*}
\begin{cases}
\begin{align*}
\forall (B, W) \in insert \ (Univ? \ A \ X \cup Univ? \ C \ Y, \ bvars \ b) \ U. \\
B : sec ' W \leadsto UNIV \implies \\
(\{\}, \text{False}) \models c (\subseteq B_1, X) = Some (B, Z) \implies \\
B_1 \neq \{\} \implies cgood c \implies csimp c = c \implies \\
B \neq \{\}; \\
\forall (B, W) \in insert \ (Univ? \ A \ X \cup Univ? \ C \ Y, \ bvars \ b) \ U. \\
B : sec ' W \leadsto UNIV \implies \\
(\{\}, \text{False}) \models c (\subseteq B_1', Y) = Some (B, Z) \implies \\
B_1' \neq \{\} \implies cgood c \implies csimp c = c \implies \\
B \neq \{\}; \\
(U, v) \models WHILE b DO c (\subseteq A, X) = Some (B, Z); \\
A \neq \{\}; cgood (WHILE b DO c); \\
\end{cases}
\end{align*}
\end{align*}
\end{align*}
\]
B: $A \neq \{\}$
shows secure $c$
proof -
\begin{itemize}
  \item fix $s s' t l$ and $x :: vname$
  \item assume $(c, s) \Rightarrow s'$
  \item then obtain $cfs$ where $C: (c, s) \Rightarrow*{cfs}$ (SKIP, $s'$)
    by (auto dest: small-steps-stepsI simp: big-iff-small)
  \item assume $D: s = t (\leq l)$
  \item have $E$: $\forall x. \sec x \leq l \Rightarrow s = t (\subseteq \text{sources (flow cfs) s})$
    \begin{itemize}
      \item proof (rule allI, rule impI)
      \item fix $x :: vname$
      \item assume $\sec x \leq l$
      \item moreover have $\text{sources (flow cfs) s} x \subseteq \{y. \sec y \leq \sec x\}$
        by (rule subsetI, simp, rule sources-le)
      \item ultimately show $s = t (\subseteq \text{sources (flow cfs) s})$
        using $D$ by auto
    \end{itemize}
  \item note $F = \text{this [rule-format]}$
  \item obtain $t'$ where $G$: $\forall x.\ sec x \leq l$
    \begin{itemize}
      \item $s = t (\subseteq \text{sources (flow cfs) s})$
        \item using $E$ and $H$ by simp
      \item hence $(c, t) \Rightarrow t'$
        \item using $G$ by (simp add: big-iff-small)
    \end{itemize}
  \item moreover \{\}
  \item fix $x :: vname$
  \item assume $\sec x \leq l$
  \item hence $s = t (\subseteq \text{sources (flow cfs) s})$
    using $E$ by simp
  \item hence $s' x = t' x$
    using $G$ by simp
  \item ultimately have $\exists t'. (c, t) \Rightarrow t' \land s' = t' (\leq l)$
    by auto
\end{itemize}
with $A$ and $B$ show \text{?thesis}
by (auto simp: correct-def secure-def split: if-split-asm)
lemma ctyping2-sec-type-assign [elim]:
assumes
A: \((\exists s. s \in \text{Univ? } A \ X) \rightarrow (\forall y \in \text{avars } a. \text{sec } y \leq \text{sec } x)\) \land
(\forall p \in U. \forall B \ Y. \ p = (B, Y) \rightarrow B = \{\} \lor (\forall y \in Y. \text{sec } y \leq \text{sec } x))
then Some (if \(x \in \{\} \land A \neq \{\})
then if \(v \mid \exists a \subseteq X\)
else (A, Univ?? A X)
else None = Some (B, Y)
is (if \((- \rightarrow ?P) \land ?Q \lor \) then _ else _) and
B: \(s \in A \land C: \text{finite (levels } U)\)
shows Max (levels U) \(\vdash x ::= a\)
proof –
  have \(?P \land ?Q\)
  using A and B by (auto simp: univ-states-if-def split: if-split-asm)
moreover from this have Max (levels U) \(\leq \text{sec } x\)
  using C by (subst Max-le-iff, auto simp: levels-def, blast)
ultimately show Max (levels U) \(\vdash x ::= a\)
  by (auto intro: Assign simp: avars-ub)
qed

lemma ctyping2-sec-type-seq:
assumes
B': \(\forall B': s = \exists s \in A \Rightarrow Max \text{ (levels U)} \vdash c_1 \land\)
B: \(\forall B' B'' C Z Y'. B = B' \Rightarrow B'' = B' \Rightarrow\)
(\(U, v) \vdash c_2 (\subseteq B', Y) = \text{Some (C, Z) \Rightarrow}\)
s' \(\in B' \Rightarrow Max \text{ (levels U)} \vdash c_2 \land\)
C: \(U, v) \vdash c_1 (\subseteq A, X) = \text{Some (B, Y) \land}\)
D: \(U, v) \vdash c_2 (\subseteq B, Y) = \text{Some (C, Z) \land}\)
E: \(s \in A \land F: \text{cgood } c_1 \land\)
G: \(\text{csimp } c_1 = c_1\)
shows Max (levels U) \(\vdash c_1; c_2\)
proof –
  have Max (levels U) \(\vdash c_1\)
  using A and E by simp
moreover from C and E and F and G have B \(\neq \{\})
  by (erule-tac csimp-ctyping2, blast)
hence Max (levels U) \(\vdash c_2\)
  using B and D by blast
ultimately show ?thesis ..
qed

lemma ctyping2-sec-type-if:
assumes
A: \( \bigwedge U' \) B C s. \( U' = \text{insert} \) (Univ? A X, bvars b) U \( \Rightarrow \)
\( B = B_1 \Rightarrow C_1 = C \Rightarrow s \in B_1 \Rightarrow \)
finite (levels (insert (Univ? A X, bvars b) U)) \( \Rightarrow \)
Max (levels (insert (Univ? A X, bvars b) U)) \( \vdash c_1 \)
(is \( \bigwedge \cdot \cdot \cdot = \cdot \Rightarrow \cdot \Rightarrow \cdot \Rightarrow \cdot \Rightarrow \cdot \Rightarrow \cdot \) assumes
B: \( \bigwedge U' \) B C s. \( U' = \text{if} \) \( \cdot \cdot \cdot U' \Rightarrow B = B_1 \Rightarrow C_2 = C \Rightarrow s \in B_2 \Rightarrow \)
finite (levels \( \cdot \cdot \cdot U' \)) \( \Rightarrow \)
Max (levels \( \cdot \cdot \cdot U' \)) \( \vdash c_2 \) and
C: \( \vdash b (\subseteq A, X) = (B_1, B_2) \) and
D: s \( \in A \) and
E: bsimp b = b and
F: \( \forall v. b \neq Be \) v and
G: finite (levels U) shows Max (levels U) \( \vdash \text{IF} b \) THEN \( c_1 \) ELSE \( c_2 \)
proof –
\( \text{from} \) D and G have H: finite (levels 
\( \cdot \cdot \cdot U' \)) \( \land \)
Max (levels \( \cdot \cdot \cdot U' \)) = max (sec b) (Max (levels U))
using levels-insert by (auto simp: univ-states-if-def)
morerover have I: \( \vdash b (\subseteq A, X) = (A, A) \)
using E and F by (rule bsimp-btyping2)
hence Max (levels \( \cdot \cdot \cdot U' \)) \( \vdash c_1 \)
using A and C and D and H by auto
moreover have Max (levels \( \cdot \cdot \cdot U' \)) \( \vdash c_2 \)
using B and C and D and H and I by auto
ultimately show \( \cdot \cdot \cdot \)thesis
by (auto intro: If)
qed

lemma ctyping2-sec-type-while:
assumes
A: \( \bigwedge B C' B' D' s. B = B_1 \Rightarrow C' = C \Rightarrow B' = B_1' \Rightarrow \)
\((\exists s. s \in Univ? A X \lor s \in Univ? C Y) \Rightarrow \)
\( (\forall x \in \text{bears b}. \text{All} (((\subseteq) (\text{sec} x)))) \land \)
\( (\forall p \in U. \text{case p of} (B, W) \Rightarrow (\exists s. s \in B) \Rightarrow \)
\( (\forall x \in W. \text{All} (((\subseteq) (\text{sec} x)))) \Rightarrow \)
\( D = D' \Rightarrow s \in B_1 \Rightarrow \text{finite} (\text{levels} \{\}) \Rightarrow \text{Max} (\text{levels} \{\}) \vdash c \)
\( \) (is \( \bigwedge \cdot \cdot \cdot \cdot \cdot \Rightarrow \cdot \Rightarrow \cdot \Rightarrow \cdot \Rightarrow \cdot \Rightarrow \cdot \) assumes
B: (if \( \cdot \cdot \cdot P \) \( \land \) (\( \forall p \in U. \forall B W. p = (B, W) \Rightarrow B = \{\} \lor \cdot \cdot \cdot Q W \))
then Some \( (B_2 \cup B_2', \text{Univ}? B_2 X \cap Y) \) else None = Some \( (B, Z) \)
\) (is (if \( \cdot \cdot \cdot R \) then - else -) = -) and
C: \( \vdash b (\subseteq A, X) = (B_1, B_2) \) and
D: s \( \in A \) and
E: bsimp b = b and
F: b \( \neq Be \) False and
G: b \( \neq Be \) True and
H: finite (levels U)
shows $\text{Max} \ (\text{levels} \ U) \vdash \text{WHILE} \ b \ \text{DO} \ c$

proof –

have $?R$
  using $B$ by (simp split: if-split-asn)
  hence $\text{sec} \ b \leq 0$
  using $D$ by (subst bvars-ub, auto simp: univ-states-if-def, fastforce)
moreover have $b \ (\subseteq A, X) = (A, A)$
  using $E$ and $F$ and $G$ by (blast intro: bsimp-btyping2)
  hence $\emptyset \vdash c$
using $A$ and $C$ and $D$ and $?R$ by (fastforce simp: levels-def)
moreover have $\text{Max} \ (\text{levels} \ U) = \emptyset$
proof (rule Max-eqI [OF $H$])
  fix $l$
  assume $l \in \text{levels} \ U$
  thus $l \leq 0$
  using $?R$ by (fastforce simp: levels-def)
next
  show $\emptyset \in \text{levels} \ U$
  by (simp add: levels-def)
qed

ultimately show $?\text{thesis}$
  by (auto intro: While)
qed


theorem ctyping2-sec-type:
$[(U, v) \vdash c \ (\subseteq A, X) = \text{Some} \ (B, Y);
  s \in A; \text{cgood} \ c; \text{csimp} \ c = c; \text{finite} \ (\text{levels} \ U)] \implies$
Max $\ (\text{levels} \ U) \vdash c$
proof (induction $(U, v) c \ A X \ Y \ U \ v \ s$ rule: ctyping2.induct)
fix $U$
  show Max $\ (\text{levels} \ U) \vdash \text{SKIP}$
  by (rule Skip)
next
  fix $A X C Z U v \ c_1 \ c_2 \ s$
  show
$\text{[[} \text{WHILE} \ Y s. \ (U, v) \vdash c_1 \ (\subseteq A, X) = \text{Some} \ (B, Y) \implies$
$s \in A \implies \text{cgood} \ c_1 \implies \text{csimp} \ c_1 = c_1 \implies \text{finite} \ (\text{levels} \ U) \implies$
Max $\ (\text{levels} \ U) \vdash c_1;$
$\text{[[} \text{WHILE} \ Y s. \ (U, v) \vdash c_1 \ (\subseteq A, X) = \text{Some} \ p \implies$
$(B, Y) = p \implies (U, v) \vdash c_2 \ (\subseteq B, Y) = \text{Some} \ (C, Z) \implies$
$s \in B \implies \text{cgood} \ c_2 \implies \text{csimp} \ c_2 = c_2 \implies \text{finite} \ (\text{levels} \ U) \implies$
Max $\ (\text{levels} \ U) \vdash c_2;$
$(U, v) \vdash c_1; c_2 \ (\subseteq A, X) = \text{Some} \ (C, Z);$
$s \in A; \text{cgood} \ (c_1; c_2);$
$\text{csimp} \ (c_1; c_2) = c_1; c_2;$
$\text{finite} \ (\text{levels} \ U)] \implies$
Max $\ (\text{levels} \ U) \vdash c_1; c_2$
by (auto split: option.split-asn, rule ctyping2-sec-type-seq)
fix $A \ X \ B \ Y \ U \ v \ b \ c_1 \ c_2 \ s$

show

\[ (U', p) = (\text{insert} \ (\text{Univ} \ A \ X, \ \text{bvars} \ b) \ U, \ v \ (\subseteq A, X)) \rightarrow \]
\[ (B_1, B_2) = p \rightarrow (U', v) \c Good \ c_1 \rightarrow \text{finite} \ (\text{levels} \ U') \rightarrow \]
\[ \text{Max} \ (\text{levels} \ U') \vdash c_1; \]
\[ \setminus A \ X \ B \ Y \ U \ v \ b \ c_1 \ c_2 \ s \text{=} \]

next

fix $A \ X \ B \ Z \ U \ v \ b \ c \ s$

show

\[ (B_1, B_2) = (\subseteq A, X) \rightarrow \]
\[ \text{finite} \ (\text{levels} \ U') \rightarrow \]
\[ \text{Max} \ (\text{levels} \ U') \vdash c; \]
\[ \setminus A \ X \ B \ Z \ U \ v \ b \ c_1 \ c_2 \ s \text{=} \]

qed (auto split: option.split-asm prod.split-asm, rule ctyping2-sec-type-if)
lemma sec-type-ctyping2-if:

assumes
\[ A: \bigwedge U' B_1 B_2. \quad U' = \text{insert} (\text{Univ}\? A X, \text{bvars} b) U \implies \]
\[ (B_1, B_2) = \vdash b (\subseteq A, X) \implies \]
\[ \text{Max} (\text{levels} (\text{insert} (\text{Univ}\? A X, \text{bvars} b) U)) \vdash c_1 \implies \]
\[ \text{finite} (\text{levels} (\text{insert} (\text{Univ}\? A X, \text{bvars} b) U)) \implies \]
\[ \exists C Y. (\text{insert} (\text{Univ}\? A X, \text{bvars} b) U, v) \vdash c_1 (\subseteq B_1, X) = \]
\[ \text{Some} (C, Y) \]
\[ \text{(is } \wedge \ldots \implies \text{?U'} \implies \ldots \implies \ldots \implies \text{-)} \]

assumes
\[ B: \bigwedge U' B_1 B_2. \quad U' = \text{?U'} \implies (B_1, B_2) = \vdash b (\subseteq A, X) \implies \]
\[ \text{Max} (\text{levels} (?U')) \vdash c_2 \implies \text{finite} (\text{levels} ?U') \implies \]
\[ \exists C Y. (?U', v) \vdash c_2 (\subseteq B_2, X) = \text{Some } (C, Y) \text{ and} \]
\[ C: \text{finite} (\text{levels } U) \text{ and} \]
\[ D: \text{max (sec } b) \text{ (Max (levels } U)) \vdash c_1 \text{ and} \]
\[ E: \text{max (sec } b) \text{ (Max (levels } U)) \vdash c_2 \]
sows \[ \exists C Y. (U, v) \vdash \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 (\subseteq A, X) = \text{Some } (C, Y) \]

proof –

obtain \[ B_1, B_2 \text{ where } F: (B_1, B_2) = \vdash b (\subseteq A, X) \]
by (cases \[ \vdash b (\subseteq A, X) \), simp)

moreover have \[ \exists C_1 C_2 Y_1 Y_2. (\text{?U'}, v) \vdash c_1 (\subseteq B_1, X) = \text{Some } (C_1, Y_1) \land\]
\[ (\text{?U'}, v) \vdash c_2 (\subseteq B_2, X) = \text{Some } (C_2, Y_2) \]

proof (cases \[ A = \{\}) \]

\[ \text{case True} \]
\[ \text{hence levels } ?U' = \text{levels } U \]
by (auto simp: levels-def univ-states-if-def)

moreover have \[ \text{Max (levels } U) \vdash c_1 \]
using \[ D \text{ by (auto intro: anti-mono)} \]

moreover have \[ \text{Max (levels } U) \vdash c_2 \]
using \[ E \text{ by (auto intro: anti-mono)} \]

ultimately show \( \text{?thesis} \)
using \[ A \text{ and } B \text{ and } C \text{ and } F \text{ by simp} \]

next

\[ \text{case False} \]

\[ \text{with } C \text{ have finite (levels } ?U') \land\]
\[ \text{Max (levels } ?U') = \text{max (sec } b) \text{ (Max (levels } U)) \]
by (simp add: levels-insert univ-states-if-def)

thus \( \text{?thesis} \)
using \[ A \text{ and } B \text{ and } D \text{ and } E \text{ and } F \text{ by simp} \]

qed

ultimately show \( \text{?thesis} \)
by (auto split: prod.split)

qed

lemma sec-type-ctyping2-while:

assumes
\[ A: \bigwedge B_1 B_2 C Y B_1' B_2'. (B_1, B_2) = \vdash b (\subseteq A, X) \implies \]

147
(C, Y) \vdash c (\subseteq B_1, X) \Longrightarrow (B_1', B_2') = \models b (\subseteq C, Y) \Longrightarrow
((\exists s. s \in \text{Univ}? A X \vee s \in \text{Univ}? C Y) \Longrightarrow
\forall x \in \text{bvars} b. \text{All} ((\leq?) \text{ (sec x)})) \wedge
\forall p \in U. \text{ case } \text{ p of } (B, W) \Rightarrow (\exists s. s \in B) \Longrightarrow
\forall x \in W. \text{ All} ((\leq?) \text{ (sec x)})) \Longrightarrow
\text{Max } \text{ (levels } \{\} \text{ )} \vdash c \Longrightarrow \text{finite } \text{ (levels } \{\} \text{ )} \Longrightarrow
\exists D Z. \{\}, \text{ False} \models c (\subseteq B_1, X) = \text{ Some } (D, Z)
\text{(is } \text{ A - C Y - . . . } \text{ } \Longrightarrow \text{ } \text{ } \text{ } \text{ } \text{ } \Longrightarrow \text{ } ?P C Y \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \text{ } \Longrightarrow \text{ )}
\text{assumes}
B: \bigwedge \text{ B}_1 \text{ B}_2 \text{ C Y B}_1' \text{ B}_2'. (\text{ B}_1, \text{ B}_2) = \models b (\subseteq A, X) \Longrightarrow
(C, Y) = \vdash c (\subseteq B_1, X) \Rightarrow (B_1', B_2') = \models b (\subseteq C, Y) \Longrightarrow
?P C Y \Longrightarrow \text{Max } \text{ (levels } \{\} \text{ )} \vdash c \Longrightarrow \text{finite } \text{ (levels } \{\} \text{ )} \Longrightarrow
\exists D Z. \{\}, \text{ False} \models c (\subseteq B_1', Y) = \text{ Some } (D, Z) \text{ and}
C: \text{ finite } \text{ (levels } U \text{ ) and}
D: \text{ Max } \text{ (levels } U \text{ ) = 0 and}
E: \text{ sec } b = 0 \text{ and}
F: 0 \vdash c
\text{shows } \exists B Y. (U, v) \models \text{ WHILE } b \text{ DO } c (\subseteq A, X) = \text{ Some } (B, Y)
\text{proof } –
\text{obtain } B_1 \text{ B}_2 \text{ where } G: (\text{ B}_1, \text{ B}_2) = \models b (\subseteq A, X)
\text{by } (\text{ cases } \models b (\subseteq A, X), \text{ simp})
\text{moreover obtain } C Y \text{ where } H: (C, Y) = \vdash c (\subseteq B_1, X)
\text{by } (\text{ cases } \vdash c (\subseteq B_1, X), \text{ simp})
\text{moreover obtain } B_1' \text{ B}_2' \text{ where } I: (B_1', B_2') = \models b (\subseteq C, Y)
\text{by } (\text{ cases } \models b (\subseteq C, Y), \text{ simp})
\text{moreover } \{ \text{fix } l x s B W \text{ assume } J : (B, W) \in U \text{ and } K : x \in W \text{ and } L : s \in B \text{ have } \text{ sec } x \leq l \text{ proof } (\text{rule le-trans, rule Max-ge } [\text{OF } C]) \text{ show } \text{ sec } x \in \text{ levels } U \text{ using } J \text{ and } K \text{ and } L \text{ by } (\text{fastforce simp: levels-def}) \text{ next } \text{show } \text{Max } \text{ (levels } U \text{ )} \leq l \text{ using } D \text{ by } \text{simp} \text{ qed } \}
\text{hence } J: ?P C Y \text{ using } E \text{ by } (\text{auto dest: bvars-sec})
\text{ultimately have } \exists D D' Z Z'. \{\}, \text{ False} \models c (\subseteq B_1, X) = \text{ Some } (D, Z) \wedge
\{\}, \text{ False} \models c (\subseteq B_1', Y) = \text{ Some } (D', Z') \text{ using } A \text{ and } B \text{ and } F \text{ by } (\text{force simp: levels-def}) \text{ thus } ?\text{thesis} \text{ using } G \text{ and } H \text{ and } I \text{ and } J \text{ by } (\text{auto split: prod.split}) \text{ qed}

\text{theorem } \text{sec-type-ctyping2:}
[\text{Max } \text{ (levels } U \text{ )} \vdash c; \text{ finite } \text{ (levels } U \text{ )}] \Longrightarrow
∃B Y. (U, v) |= c (⊆ A, X) = Some (B, Y)

proof (induction (U, v) c A X arbitrary; U v rule: ctyping2.induct)

fix A X U v x a

show Max (levels U) ⊢ x ::= a ⇒ finite (levels U) ⇒

∃B Y. (U, v) |= x ::= a (⊆ A, X) = Some (B, Y)

by (fastforce dest: avars-sec simp: levels-def)

next

fix A X U v b c₁ c₂

show

\]\ \ p B₁ B₂.

(U', p) = (insert (Univ? A X, bvars b) U, |= b (⊆ A, X)) ⇒

(B₁, B₂) = p ⇒ Max (levels U') ⊢ c₁ ⇒ finite (levels U') ⇒

∃B Y. (U', v) |= c₁ (⊆ B₁, X) = Some (B, Y);

\]\ \ p B₁ B₂.

(U', p) = (insert (Univ? A X, bvars b) U, |= b (⊆ A, X)) ⇒

(B₁, B₂) = p ⇒ Max (levels U') ⊢ c₂ ⇒ finite (levels U') ⇒

∃B Y. (U', v) |= c₂ (⊆ B₂, X) = Some (B, Y);

Max (levels U) ⇒ IF b THEN c₁ ELSE c₂; finite (levels U) \] \] ⇒

∃B Y. (U, v) |= IF b THEN c₁ ELSE c₂ (⊆ A, X) = Some (B, Y)

by (auto simp del: ctyping2.simps(4), rule sec-type-ctyping2-if)

next

fix A X U v b c

show

\]\ \ B₁ B₂ C Y B₁' B₂'.

(B₁, B₂) = |= b (⊆ A, X) ⇒

(C, Y) = ⊢ c (⊆ B₁, X) ⇒

(B₁', B₂') = |= b (⊆ C, Y) ⇒

∀(B, W) ∈ insert (Univ? A X ∪ Univ? C Y, bvars b) U.

B: sec ' W ⊲ UNIV ⇒

Max (levels {}) ⊢ c ⇒ finite (levels {}) ⇒

∃B Z. ({}, False) |= c (⊆ B₁, X) = Some (B, Z);

\]\ \ B₁ B₂ C Y B₁' B₂'.

(B₁, B₂) = |= b (⊆ A, X) ⇒

(C, Y) = ⊢ c (⊆ B₁, X) ⇒

(B₁', B₂') = |= b (⊆ C, Y) ⇒

∀(B, W) ∈ insert (Univ? A X ∪ Univ? C Y, bvars b) U.

B: sec ' W ⊲ UNIV ⇒

Max (levels {}) ⊢ c ⇒ finite (levels {}) ⇒

∃B Z. ({}, False) |= c (⊆ B₁', Y) = Some (B, Z);

Max (levels U) ⇒ WHILE b DO c; finite (levels U) \] \] ⇒

∃B Z. (U, v) |= WHILE b DO c (⊆ A, X) = Some (B, Z)

by (auto simp del: ctyping2.simps(5), rule sec-type-ctyping2-while)

qed auto

end
References


