Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming

Simon Foster, Frank Zeyda, Yakoub Nemouchi, Pedro Ribeiro, and Burkhart Wolff

June 11, 2019

Abstract

Isabelle/UTP is a mechanised theory engineering toolkit based on Hoare and He’s Unifying Theories of Programming (UTP). UTP enables the creation of denotational, algebraic, and operational semantics for different programming languages using an alphabetised relational calculus. We provide a semantic embedding of the alphabetised relational calculus in Isabelle/HOL, including new type definitions, relational constructors, automated proof tactics, and accompanying algebraic laws. Isabelle/UTP can be used to both capture laws of programming for different languages, and put these fundamental theorems to work in the creation of associated verification tools, using calculi like Hoare logics. This document describes the relational core of the UTP in Isabelle/HOL.

Contents

1 Introduction 7
2 UTP Variables 8
  2.1 Initial syntax setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.2 Variable foundations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  2.3 Variable lens properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  2.4 Lens simplifications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.5 Syntax translations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3 UTP Expressions 14
  3.1 Expression type . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
  3.2 Core expression constructs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  3.3 Type class instantiations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
  3.4 Syntax translations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
  3.5 Evaluation laws for expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
  3.6 Misc laws . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
  3.7 Literalise tactics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
4 Expression Type Class Instantiations 20
  4.1 Expression construction from HOL terms . . . . . . . . . . . . . . . . . . . . . . . 22
  4.2 Lifting set collectors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
  4.3 Lifting limits . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

*Department of Computer Science, University of York. simon.foster@york.ac.uk
1 Introduction

This document contains the description of our mechanisation of Hoare and He’s *Unifying Theories of Programming* [22, 7] (UTP) in Isabelle/HOL. UTP uses the “programs-as-predicates” approach, pioneered by Hehner [20, 18, 19], to encode denotational semantics and facilitate reasoning about programs. It uses the alphabetised relational calculus, which combines predicate calculus and relation algebra, to denote programs as relations between initial variables ($x$) and their subsequent values ($x'$). Isabelle/UTP\footnote{Isabelle/UTP website: https://www.cs.york.ac.uk/circus/isabelle-utp/} [16, 28, 15] semantically embeds this relational calculus into Isabelle/HOL, which enables application of the latter’s proof facilities to program verification. For an introduction to UTP, we recommend two tutorials [6, 7], and also the UTP book [22].

The Isabelle/UTP core mechanises most of definitions and theorems from chapters 1, 2, 4, and 7 of [22], and some material contained in chapters 5 and 10. This essentially amounts to alphabetised predicate calculus, its core laws, the UTP theory infrastructure, and also parallel-by-merge [22, chapter 5], which adds concurrency primitives. The Isabelle/UTP core does not contain the theory of designs [6] and CSP [7], which are both represented in their own theory developments.

A large part of the mechanisation, however, is foundations that enable these core UTP theories. In particular, Isabelle/UTP builds on our implementation of lenses [16, 14], which gives a formal semantics to state spaces and variables. This, in turn, builds on a previous version of Isabelle/UTP [9, 10], which provided a shallow embedding of UTP by using Isabelle record types to represent alphabets. We follow this approach and, additionally, use the lens laws [11, 16] to characterise well-behaved variables. We also add meta-logical infrastructure for dealing with free variables and substitution. All this, we believe, adds an additional layer rigour to the UTP.

The alphabets-as-types approach does impose a number of theoretical limitations. For example, alphabets can only be extended when an injection into a larger state-space type can be exhibited. It is therefore not possible to arbitrarily augment an alphabet with additional variables, but new types must be created to do this. This is largely because as in previous work [9, 10], we actually encode state spaces rather than alphabets, the latter being implicit. Namely, a relation is typed by the state space type that it manipulates, and the alphabet is represented by collection of lenses into this state space. This aspect of our mechanisation is actually much closer to the relational program model in Back’s refinement calculus [3].

The pay-off is that the Isabelle/HOL type checker can be directly applied to relational constructions, which makes proof much more automated and efficient. Moreover, our use of lenses mitigates the limitations by providing meta-logical style operators, such as equality on variables, and alphabet membership [16]. Isabelle/UTP can therefore directly harness proof automation from Isabelle/HOL, which allows its use in building efficient verification tools [13, 12]. For a detailed discussion of semantic embedding approaches, please see [28].

In addition to formalising variables, we also make a number of generalisations to UTP laws. Notably, our lens-based representation of state leads us to adopt Back’s approach to both assignment and local variables [3]. Assignment becomes a point-free operator that acts on state-space update functions, which provides a rich set of algebraic theorems. Local variables are represented using stacks, unlike in the UTP book where they utilise alphabet extension.
We give a summary of the main contributions within the Isabelle/UTP core, which can all be seen in the table of contents.

1. Formalisation of variables and state-spaces using lenses [16];
2. an expression model, together with lifted operators from HOL;
3. the meta-logical operators of unrestriction, used-by, substitution, alphabet extrusion, and alphabet restriction;
4. the alphabetised predicate calculus and associated algebraic laws;
5. the alphabetised relational calculus and associated algebraic laws;
6. proof tactics for the above based on interpretation [23];
7. a formalisation of UTP theories using locales [4] and building on HOL-Algebra [5];
8. Hoare logic [21] and dynamic logic [17];
9. weakest precondition and strongest postcondition calculi [8];
10. concurrent programming with parallel-by-merge;
11. relational operational semantics.

2 UTP Variables

theory utp-var
  imports
  UTP−Toolkit.utp-toolkit
  utp-parser-utils
begin

In this first UTP theory we set up variables, which are are built on lenses [11, 16]. A large part of this theory is setting up the parser for UTP variable syntax.

2.1 Initial syntax setup

We will overload the square order relation with refinement and also the lattice operators so we will turn off these notations.

purge-notation
  Order.le (infixl \(\sqsubseteq\) 50) and
  Lattice.sup (\([\sqcup\]) 90) and
  Lattice.inf (\([\sqcap\]) 90) and
  Lattice.join (infixl \(\sqcup\) 65) and
  Lattice.meet (infixl \(\sqcap\) 70) and
  Set.member (op :) and
  Set.member ((-/ : -) \([51, 51]\) 50) and
  disj (infixr | 30) and
  conj (infixr & 35)

declare fst-vwb-lens [simp]
declare snd-vwb-lens [simp]
declare comp-vwb-lens [simp]
2.2 Variable foundations

This theory describes the foundational structure of UTP variables, upon which the rest of our model rests. We start by defining alphabets, which following [9, 10] in this shallow model are simply represented as types \( \alpha \), though by convention usually a record type where each field corresponds to a variable. UTP variables in this frame are simply modelled as lenses \( a = \Rightarrow \alpha \), where the view type \( a \) is the variable type, and the source type \( \alpha \) is the alphabet or state-space type.

We define some lifting functions for variables to create input and output variables. These simply lift the alphabet to a tuple type since relations will ultimately be defined by a tuple alphabet.

\[
\text{definition in-var} :: (a = \Rightarrow \alpha) \Rightarrow (a = \Rightarrow \alpha \times \beta)
\]

\text{definition out-var} :: (a = \Rightarrow \beta) \Rightarrow (a = \Rightarrow \alpha \times \beta)

Variables can also be used to effectively define sets of variables. Here we define the the universal alphabet (\( \Sigma \)) to be the bijective lens \( 1_L \). This characterises the whole of the source type, and thus is effectively the set of all alphabet variables.

\[
\text{abbreviation (input) univ-alpha :: (} \alpha = \Rightarrow (} \alpha \times \Sigma \text{)} \quad \text{where}
\]

\[
\text{univ-alpha} \equiv 1_L
\]

The next construct is vacuous and simply exists to help the parser distinguish predicate variables from input and output variables.

\[
\text{definition pr-var} :: (a = \Rightarrow \beta) \Rightarrow (a = \Rightarrow \beta)
\]

2.3 Variable lens properties

We can now easily show that our UTP variable construction are various classes of well-behaved lens.

\[
\text{lemma in-var-weak-lens [simp]};
\]

\[
\text{weak-lens } x \Rightarrow \text{weak-lens} \ (\text{in-var } x)
\]

\[
\text{by (simp add: comp-weak-lens in-var-def)}
\]

\[
\text{lemma in-var-semi-uvar [simp]};
\]

\[
\text{mwb-lens } x \Rightarrow \text{mwb-lens} \ (\text{in-var } x)
\]

\[
\text{by (simp add: comp-mwb-lens in-var-def)}
\]

\[
\text{lemma pr-var-weak-lens [simp]};
\]

\[
\text{weak-lens } x \Rightarrow \text{weak-lens} \ (\text{pr-var } x)
\]

\[
\text{by (simp add: pr-var-def)}
\]

\[
\text{lemma pr-var-mwb-lens [simp]};
\]

\[
\text{mwb-lens } x \Rightarrow \text{mwb-lens} \ (\text{pr-var } x)
\]

\[
\text{by (simp add: pr-var-def)}
\]
\textbf{lemma} \textit{pr-var-vwb-lens} [simp]:
\begin{align*}
vwb-lens \ x & \implies vwb-lens \ (pr-var \ x)
\end{align*}
\textit{by} (simp add: \textit{pr-var-def})

\textbf{lemma} \textit{in-var-uvar} [simp]:
\begin{align*}
vwb-lens \ x & \implies vwb-lens \ (in-var \ x)
\end{align*}
\textit{by} (simp add: \textit{in-var-def})

\textbf{lemma} \textit{out-var-weak-lens} [simp]:
\begin{align*}
weak-lens \ x & \implies weak-lens \ (out-var \ x)
\end{align*}
\textit{by} (simp add: \textit{comp-weak-lens out-var-def})

\textbf{lemma} \textit{out-var-semi-uvar} [simp]:
\begin{align*}
mwb-lens \ x & \implies mwb-lens \ (out-var \ x)
\end{align*}
\textit{by} (simp add: \textit{comp-mwb-lens out-var-def})

\textbf{lemma} \textit{out-var-uvar} [simp]:
\begin{align*}
vwb-lens \ x & \implies vwb-lens \ (out-var \ x)
\end{align*}
\textit{by} (simp add: \textit{out-var-def})

Moreover, we can show that input and output variables are independent, since they refer to different sections of the alphabet.

\textbf{lemma} \textit{in-out-indep} [simp]:
\begin{align*}
in-var \ x & \bowtie in-var \ y
\end{align*}

\textbf{lemma} \textit{out-in-indep} [simp]:
\begin{align*}
out-var \ x & \bowtie out-var \ y
\end{align*}

\textbf{lemma} \textit{in-var-indep} [simp]:
\begin{align*}
x \bowtie y & \implies in-var \ x \bowtie in-var \ y
\end{align*}
\textit{by} (simp add: \textit{in-var-def out-var-def})

\textbf{lemma} \textit{out-var-indep} [simp]:
\begin{align*}
x \bowtie y & \implies out-var \ x \bowtie out-var \ y
\end{align*}
\textit{by} (simp add: \textit{out-var-def})

\textbf{lemma} \textit{pr-var-indeps} [simp]:
\begin{align*}
x \bowtie y & \implies pr-var \ x \bowtie y
\end{align*}
\textit{by} (simp-all add: \textit{pr-var-def})

\textbf{lemma} \textit{prod-lens-indep-in-var} [simp]:
\begin{align*}
\ a \bowtie x & \implies a \times_b in-var \ x
\end{align*}
\textit{by} (metis \textit{in-var-def in-var-indep out-in-indep out-var-def plus-pres-lens-indep prod-as-plus})

\textbf{lemma} \textit{prod-lens-indep-out-var} [simp]:
\begin{align*}
\ b \bowtie x & \implies a \times_b out-var \ x
\end{align*}
\textit{by} (metis \textit{in-out-indep in-var-def out-var-def out-var-indep plus-pres-lens-indep prod-as-plus})

\textbf{lemma} \textit{in-var-pr-var} [simp]:
\begin{align*}
in-var \ (pr-var \ x) & = \text{in-var} \ x
\end{align*}
\textit{by} (simp add: \textit{pr-var-def})
lemma out-var-pr-var [simp]:
out-var (pr-var x) = out-var x
by (simp add: pr-var-def)

lemma pr-var-idem [simp]:
pr-var (pr-var x) = pr-var x
by (simp add: pr-var-def)

lemma pr-var-lens-plus [simp]:
pr-var (x +L y) = (x +L y)
by (simp add: pr-var-def)

lemma in-var-plus [simp]: in-var (x +L y) = in-var x +L in-var y
by (simp add: in-var-def)

lemma out-var-plus [simp]: out-var (x +L y) = out-var x +L out-var y
by (simp add: out-var-def)

Similar properties follow for sublens

lemma in-var-sublens [simp]:
y ⊆L x ⇒ in-var y ⊆L in-var x
by (metis (no-types, hide-lams) in-var-def lens-comp-assoc sublens-def)

lemma out-var-sublens [simp]:
y ⊆L x ⇒ out-var y ⊆L out-var x
by (metis (no-types, hide-lams) out-var-def lens-comp-assoc sublens-def)

lemma pr-var-sublens [simp]:
y ⊆L x ⇒ pr-var y ⊆L pr-var x
by (simp add: pr-var-def)

2.4 Lens simplifications

We also define some lookup abstraction simplifications.

lemma var-lookup-in [simp]: lens-get (in-var x) (A, A') = lens-get x A
by (simp add: in-var-def fst-lens-def lens-comp-def)

lemma var-lookup-out [simp]: lens-get (out-var x) (A, A') = lens-get x A'
by (simp add: out-var-def snd-lens-def lens-comp-def)

lemma var-update-in [simp]: lens-put (in-var x) (A, A') v = (lens-put x A v, A')
by (simp add: in-var-def fst-lens-def lens-comp-def)

lemma var-update-out [simp]: lens-put (out-var x) (A, A') v = (A, lens-put x A' v)
by (simp add: out-var-def snd-lens-def lens-comp-def)

lemma get-lens-plus [simp]: get_x +L y s = (get_x s, get_y s)
by (simp add: lens-defs)
2.5 Syntax translations

In order to support nice syntax for variables, we here set up some translations. The first step is to introduce a collection of non-terminals.

**nonterminal** svid and svids and svar and svars and salpha

These non-terminals correspond to the following syntactic entities. Non-terminal svid is an atomic variable identifier, and svids is a list of identifier. svar is a decorated variable, such as an input or output variable, and svars is a list of decorated variables. salpha is an alphabet or set of variables. Such sets can be constructed only through lens composition due to typing restrictions. Next we introduce some syntax constructors.

**syntax — Identifiers**

- svid :: id ⇒ svid (- [999] 999)
- svid-unit :: svid ⇒ svids (-)
- svid-dot :: svid ⇒ svid ⇒ svids (-. / -)
- svid-alpha :: svid (v)
- svid-dot :: svid ⇒ svid ⇒ svids (-. [998, 999] 998)
- mk-svid-list :: svids ⇒ logic — Helper function for summing a list of identifiers

A variable identifier can either be a HOL identifier, the complete set of variables in the alphabet v, or a composite identifier separated by colons, which corresponds to a sort of qualification. The final option is effectively a lens composition.

**syntax — Decorations**

- spvar :: svid ⇒ svar (& - [990] 990)
- sinvar :: svid ⇒ svar ($ - [990] 990)
- soutvar :: svid ⇒ svar ($' - [990] 990)

A variable can be decorated with an ampersand, to indicate it is a predicate variable, with a dollar to indicate its an unprimed relational variable, or a dollar and “acute” symbol to indicate its a primed relational variable. Isabelle’s parser is extensible so additional decorations can be and are added later.

**syntax — Variable sets**

- salphaid :: svid ⇒ salpha (- [990] 990)
- salphavar :: svar ⇒ salpha (- [990] 990)
- salphaparen :: salpha ⇒ salpha ('(')
- salphacomp :: salpha ⇒ salpha ⇒ salpha (infixr ; 75)
- salphaprod :: salpha ⇒ salpha ⇒ salpha (infixr × 85)
- salpha-all :: salpha (Σ)
- salpha-none :: salpha (∅)
- svar-nil :: svar ⇒ svars (-)
- svar-cons :: svar ⇒ svars ⇒ svars (-/ -)
- salphaset :: svars ⇒ salpha ([-])
- salphamk :: logic ⇒ salpha

The terminals of an alphabet are either HOL identifiers or UTP variable identifiers. We support two ways of constructing alphabets; by composition of smaller alphabets using a semi-colon or by a set-style construction \{a, b, c\} with a list of UTP variables.

**syntax — Quotations**

- ualpha-set :: svars ⇒ logic (\{\} α)
- uvar :: svar ⇒ logic ('(') α)

For various reasons, the syntax constructors above all yield specific grammar categories and will not parser at the HOL top level (basically this is to do with us wanting to reuse the syntax for expressions). As a result we provide some quotation constructors above.
Next we need to construct the syntax translations rules. First we need a few polymorphic constants.

**consts**

```
consts
  svar :: 'v ⇒ 'e
  ivar :: 'v ⇒ 'e
  ovar :: 'v ⇒ 'e
```

**adhoc-overloading**

```
svar pr-var and ivar in-var and ovar out-var
```

The functions above turn a representation of a variable (type 'v), including its name and type, into some lens type 'e. svar constructs a predicate variable, ivar and input variables, and ovar and output variable. The functions bridge between the model and encoding of the variable and its interpretation as a lens in order to integrate it into the general lens-based framework. Overriding these functions is then all we need to make use of any kind of variables in terms of interfacing it with the system. Although in core UTP variables are always modelled using record field, we can overload these constants to allow other kinds of variables, such as deep variables with explicit syntax and type information.

Finally, we set up the translations rules.

**translations**

— Identifiers

```
-svid x → x
-svid-alpha ⇒ Σ
-svid-dot x y → y :L x
-mk-svid-list (-svid-unit x) → x
-mk-svid-list (-svid-list x xs) → x +L -mk-svid-list xs
```

— Decorations

```
-spvar Σ ← CONST svar CONST id-lens
-sinvar Σ ← CONST ivar 1_L
-soutvar Σ ← CONST ovar 1_L
-spvar (-svid-dot x y) ← CONST svar (CONST lens-comp y x)
-sinvar (-svid-dot x y) ← CONST ivar (CONST lens-comp y x)
-soutvar (-svid-dot x y) ← CONST ovar (CONST lens-comp y x)
-svid-dot (-svid-dot x y) z ← -svid-dot (CONST lens-comp y x) z
```

```
-spvar x ← CONST svar x
-sinvar x ← CONST ivar x
-soutvar x ← CONST ovar x
```

— Alphabets

```
-salphaparen a → a
-salphaid x → x
-salphacompl x y → x +L y
-salphaprod a b ⇒ a ×L b
-salphavar x → x
-svar-nil x → x
-svar-cons x xs → x +L xs
-salphaset A → A
(-svar-cons x (-salphamk y)) ← -salphamk (x +L y)
x ← -salphamk x
-salpha-all ← 1_L
-salpha-none ← 0_L
```
The translation rules mainly convert syntax into lens constructions, using a mixture of lens operators and the bespoke variable definitions. Notably, a colon variable identifier qualification becomes a lens composition, and variable sets are constructed using len sum. The translation rules are carefully crafted to ensure both parsing and pretty printing.

Finally we create the following useful utility translation function that allows us to construct a UTP variable (lens) type given a return and alphabet type.

```plaintext
syntax
-uvar-ty :: type ⇒ type ⇒ type

parse-translation (let
  fun uvar-ty-tr ty = Syntax.const @{type-syntax lens} ty $ Syntax.const @{type-syntax dummy}
  | uvar-ty-tr ts = raise TERM (uvar-ty-tr, ts);
in [[@{syntax-const -uvar-ty}, K uvar-ty-tr]] end)
end

3 UTP Expressions

theory utp-expr
imports utp-var
begin

3.1 Expression type

punge-notation BNF-Def.convol ((|--,--))

Before building the predicate model, we will build a model of expressions that generalise alphabetised predicates. Expressions are represented semantically as mapping from the alphabet 'α to the expression’s type 't. This general model will allow us to unify all constructions under one type. The majority definitions in the file are given using the lifting package [23], which allows us to reuse much of the existing library of HOL functions.

typedef ('t, 'α) uexpr = UNIV :: ('α ⇒ 't) set ..

setup-lifting type-definition-uexpr

notation Rep-uexpr ([]|e)
notation Abs-uexpr (mk_e)

lemma uexpr-eq-iff:
  e = f ←→ (∀ b. [e]_b = [f]_b)
using Rep-uexpr-inject[of e f, THEN sym] by (auto)

The term [e]_b effectively refers to the semantic interpretation of the expression under the state-space valuation (or variables binding) b. It can be used, in concert with the lifting package, to interpret UTP constructs to their HOL equivalents. We create some theorem sets to store such transfer theorems.
named-theorems \texttt{uexpr-defs} and \texttt{ueval} and \texttt{lit-simps} and \texttt{lit-norm}

3.2 Core expression constructs

A variable expression corresponds to the lens \texttt{get} function associated with a variable. Specifically, given a lens the expression always returns that portion of the state-space referred to by the lens.

\textbf{lift-definition} \texttt{var} :: \texttt{(t \Rightarrow \alpha)} \Rightarrow \texttt{(t, \alpha) uexpr is lens-get}.

A literal is simply a constant function expression, always returning the same value for any binding.

\textbf{lift-definition} \texttt{lit} :: \texttt{t \Rightarrow \texttt{(t, \alpha) uexpr} \texttt{(\_\_)} \texttt{is} \texttt{\lambda b. v. v}}.

We define lifting for unary, binary, ternary, and quaternary expression constructs, that simply take a HOL function with correct number of arguments and apply it function to all possible results of the expressions.

\textbf{lift-definition} \texttt{uop} :: \texttt{(\_\_)} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr}
\texttt{is} \texttt{\lambda f e b. f (e b)}.

\textbf{lift-definition} \texttt{bop} ::
\texttt{(\_\_)} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr}
\texttt{is} \texttt{\lambda f u v b. f (u b) (v b)}.

\textbf{lift-definition} \texttt{trop} ::
\texttt{(\_\_)} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr}
\texttt{is} \texttt{\lambda f u v w b. f (u b) (v b) (w b)}.

\textbf{lift-definition} \texttt{qtop} ::
\texttt{(\_\_)} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr}
\texttt{is} \texttt{\lambda f u v w x b. f (u b) (v b) (w b) (x b)}.

We also define a UTP expression version of function (\texttt{\lambda}) abstraction, that takes a function producing an expression and produces an expression producing a function.

\textbf{lift-definition} \texttt{ulambda} :: \texttt{(\_\_ \Rightarrow \_\_)} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr}
\texttt{is} \texttt{\lambda A x. f x A}.

We set up syntax for the conditional. This is effectively an infix version of if-then-else where the condition is in the middle.

\textbf{definition} \texttt{uif} :: \texttt{bool \Rightarrow \_\_-> \_\_ \Rightarrow \_\_ \_\_ where}
\texttt{[uexpr-defs]: uif = If}

\textbf{abbreviation} \texttt{cond} ::
\texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr}
\texttt{is} \texttt{\lambda P Q b \Rightarrow P < b \Rightarrow Q [52, 0.53]}.

\textbf{where} \texttt{P < b \Rightarrow Q \equiv trop uif b P Q}

UTP expression is equality is simply HOL equality lifted using the \texttt{bop} binary expression constructor.

\textbf{definition} \texttt{eq-apred} :: \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr} \Rightarrow \texttt{(\_\_, \alpha) uexpr}
\texttt{is} \texttt{\lambda x y = bop HOL.eq x y}

\textbf{where} \texttt{[uexpr-defs]: eq-apred x y = bop HOL.eq x y}

A literal is the expression \texttt{\_\_v}, where \texttt{v} is any HOL term. Actually, the literal construct is very versatile and also allows us to refer to HOL variables within UTP expressions, and has a variety of other uses. It can therefore also be considered as a kind of quotation mechanism.
We also set up syntax for UTP variable expressions.

```plaintext
syntax
-uuvar :: svar ⇒ logic (-)
```

```plaintext
translations
-uuvar x == CONST var x
```

Since we already have a parser for variables, we can directly reuse it and simply apply the var expression construct to lift the resulting variable to an expression.

### 3.3 Type class instantiations

Isabelle/HOL of course provides a large hierarchy of type classes that provide constructs such as numerals and the arithmetic operators. Fortunately we can directly make use of these for UTP expressions, and thus we now perform a long list of appropriate instantiations. We first lift the core arithmetic constants and operators using a mixture of literals, unary, and binary expression constructors.

```plaintext
instantiation uexpr :: (zero, type) zero
begin
  definition zero-uexpr-def [uexpr-defs]: 0 = lit 0
instance ..
end

instantiation uexpr :: (one, type) one
begin
  definition one-uexpr-def [uexpr-defs]: 1 = lit 1
instance ..
end

instantiation uexpr :: (plus, type) plus
begin
  definition plus-uexpr-def [uexpr-defs]: u + v = bop (+) u v
instance ..
end

instance uexpr :: (semigroup-add, type) semigroup-add
  by (intro-classes) (simp add: plus-uexpr-def zero-uexpr-def, transfer, simp add: add.assoc)+
```

The following instantiation sets up numerals. This will allow us to have Isabelle number representations (i.e. 3,7,42,198 etc.) to UTP expressions directly.

```plaintext
instance uexpr :: (numeral, type) numeral
  by (intro-classes, simp add: plus-uexpr-def, transfer, simp add: add.assoc)
```

We can also define the order relation on expressions. Now, unlike the previous group and ring constructs, the order relations (≤) and (≤) return a bool type. This order is not therefore the lifted order which allows us to compare the valuation of two expressions, but rather the order on expressions themselves. Notably, this instantiation will later allow us to talk about predicate refinements and complete lattices.

```plaintext
instantiation uexpr :: (ord, type) ord
begin
  lift-definition less-eq-uexpr :: ('a, 'b) uexpr ⇒ ('a, 'b) uexpr ⇒ bool
```
is $\lambda P \ Q. (\forall A. P \ A \leq Q \ A)$.

**definition** less-uxexpr :: $(\alpha, \beta) \ uexpr \Rightarrow (\alpha, \beta) \ uexpr \Rightarrow \text{bool}$

**where** [uexpr-defs]: less-uxexpr $P \ Q \equiv (P \leq Q \land \neg Q \leq P)$

**instance** ..

end

UTP expressions whose return type is a partial ordered type, are also partially ordered as the following instantiation demonstrates.

**instance** uexpr :: (order, type) order

**proof**

fix $x \ y \ z :: (\alpha, \beta) \ uexpr$

show $(x < y) \equiv (x \leq y \land \neg y \leq x)$ by (simp add: less-uxexpr-def)

show $x < y$ by (transfer, auto)

show $x \leq y \Rightarrow y \leq z \Rightarrow x \leq z$

by (transfer, blast intro:order.trans)

show $x \leq y \Rightarrow y \leq x \Rightarrow x = y$

by (transfer, rule ext, simp add: eq-iff)

qed

### 3.4 Syntax translations

The follows a large number of translations that lift HOL functions to UTP expressions using the various expression constructors defined above. Much of the time we try to keep the HOL syntax but add a "\u" subscript.

**abbreviation** (input) ulens-overrd $x \ f \ g \equiv \text{lens-overrd} \ f \ g \ x$

This operator allows us to get the characteristic set of a type. Essentially this is UNIV, but it retains the type syntactically for pretty printing.

**definition** set-of :: \(\alpha\) itself $\Rightarrow \ 'a\ \text{set}$ where

[uexpr-defs]: set-of $t = \text{UNIV}$

We add new non-terminals for UTP tuples and maplets.

**nonterminal** utuple-args and umaplet and umaplets

**syntax** — Core expression constructs

- ucoerce :: logic $\Rightarrow$ type $\Rightarrow$ logic (infix $\in\ u\ 50$)
- ulambda :: pttrn $\Rightarrow$ logic $\Rightarrow$ logic ($\lambda \cdot \cdot \cdot [0, 10] 10$)
- ulens-ovrd :: logic $\Rightarrow$ logic $\Rightarrow$ salpha $\Rightarrow$ logic ($\ast \cdot \cdot \cdot \text{on} \cdot \cdot \cdot [85, 0, 86] 86$)
- ulens-get :: logic $\Rightarrow$ logic $\Rightarrow$ svar $\Rightarrow$ logic ($\ast \cdot \cdot \cdot [900, 901] 901$)
- umem :: $(\alpha, \alpha) \ uexpr \Rightarrow (\alpha \ \text{set}, \ 'a\ \text{expr} \Rightarrow (\text{bool}, \ 'a\ \text{expr} \ \text{infix} \in_u \ 50)$

**translations**

$\lambda \ x \ p \equiv \text{CONST ulambda} \ (\lambda \ x. \ p)$

$x_\ u \ 'a \ x_\ u \equiv \ x :: (\alpha, \cdot) \ uexpr$

ulens-ovrd $f \ g \ a \Rightarrow \text{CONST bop} \ (\text{CONST ulens-overrd} \ a) \ f \ g$

ulens-ovrd $f \ g \ a \Leftarrow \text{CONST bop} \ (\lambda x \ y. \ \text{CONST lens-overrd} \ x1 \ y1 \ a) \ f \ g$

ulens-get $x \ y \equiv \text{CONST wop} \ (\text{CONST lens-get} \ y) \ x$

$x \in_u A \equiv \text{CONST bop} \ (\in) \ x \ A$

**syntax** — Tuples

- utuple :: $(\alpha, \alpha) \ uexpr \Rightarrow \text{utuple-args} \Rightarrow (\alpha \ * \ \beta, \ 'a) \ uexpr \ ((1 \ '(-) / -)\_u)$
- utuple-arg :: $(\alpha, \ 'a) \ uexpr \Rightarrow \text{utuple-args} \ (-)$
- utuple-args :: $(\alpha, \ 'a) \ uexpr \Rightarrow \text{utuple-args} \Rightarrow \text{utuple-args} \ (-, / -)$
- uunit :: $(\alpha, \ 'a) \ uexpr \ ((\cdot)\_u)$
- `ufst` :: (‘a × ‘b, ‘α) uexpr ⇒ (‘a, ‘α) uexpr (π₁(-))
- `usnd` :: (‘a × ‘b, ‘α) uexpr ⇒ (‘b, ‘α) uexpr (π₂(-))

**translations**

\( ()_u \equiv \langle () \rangle \)
\( (x, y)_u \equiv \CONST\ bop (\CONST\ Pair)\ x\ y \)
- `utuple x (-utuple-args y z)` == `utuple x (-utuple-arg (-utuple y z))`
\( \pi_1(x) \equiv \CONST\ wop\ \CONST\ fst\ x \)
\( \pi_2(x) \equiv \CONST\ wop\ \CONST\ snd\ x \)

**syntax — Orders**

- `uless` :: logic ⇒ logic ⇒ logic (infix \(<_u\) 50)
- `uleq` :: logic ⇒ logic ⇒ logic (infix \(\leq_u\) 50)
- `ugreat` :: logic ⇒ logic ⇒ logic (infix \(>_u\) 50)
- `ugeq` :: logic ⇒ logic ⇒ logic (infix \(\geq_u\) 50)

**translations**

\( x <_u y \equiv \CONST\ bop (\langle\rangle)\ x\ y \)
\( x \leq_u y \equiv \CONST\ bop (\langle\rangle)\ x\ y \)
\( x >_u y \Rightarrow y <_u x \)
\( x \geq_u y \Rightarrow y \leq_u x \)

### 3.5 Evaluation laws for expressions

The following laws show how to evaluate the core expressions constructs in terms of which the above definitions are defined. Thus, using these theorems together, we can convert any UTP expression into a pure HOL expression. All these theorems are marked as `ueval` theorems which can be used for evaluation.

**lemma** `lit-ueval [ueval]`: \([\langle x\rangle]_e\ b = x\)
by (transfer, simp)

**lemma** `var-ueval [ueval]`: \([\var x]_e\ b = \get_x\ b\)
by (transfer, simp)

**lemma** `uop-ueval [ueval]`: \([\uop f x]_e\ b = f (\[x]_e\ b)\)
by (transfer, simp)

**lemma** `bop-ueval [ueval]`: \([\bop f x y]_e\ b = f (\[x]_e\ b) (\[y]_e\ b)\)
by (transfer, simp)

**lemma** `trop-ueval [ueval]`: \([\trop f x y z]_e\ b = f (\[x]_e\ b) (\[y]_e\ b) (\[z]_e\ b)\)
by (transfer, simp)

**lemma** `qtop-ueval [ueval]`: \([\qtop f x y z w]_e\ b = f (\[x]_e\ b) (\[y]_e\ b) (\[z]_e\ b) (\[w]_e\ b)\)
by (transfer, simp)

### 3.6 Misc laws

We also prove a few useful algebraic and expansion laws for expressions.

**lemma** `uop-const [simp]`: \(\uop id\ u = u\)
by (transfer, simp)

**lemma** `bop-const-1 [simp]`: \(\bop (\lambda x. y)\ u v = v\)
by (transfer, simp)
lemma bop-const-2 [simp]: bop (\lambda x y. x) u v = u
by (transfer, simp)

lemma uexpr-fst [simp]: \pi_1((e, f)_u) = e
by (transfer, simp)

lemma uexpr-snd [simp]: \pi_2((e, f)_u) = f
by (transfer, simp)

3.7 Literalise tactics

The following tactic converts literal HOL expressions to UTP expressions and vice-versa via a collection of simplification rules. The two tactics are called "literalise", which converts UTP to expressions to HOL expressions – i.e. it pushes them into literals – and unliteralise that reverses this. We collect the equations in a theorem attribute called "lit.simps".

lemma lit-fun-simps [lit-simps]:
\begin{align*}
\langle i x y z u \rangle & = qtop i \langle x \rangle \langle y \rangle \langle z \rangle \langle u \rangle \\
\langle h x y z \rangle & = trop h \langle x \rangle \langle y \rangle \langle z \rangle \\
\langle g x y \rangle & = bop g \langle x \rangle \langle y \rangle \\
\langle f x \rangle & = uop f \langle x \rangle 
\end{align*}
by (transfer, simp)

The following two theorems also set up interpretation of numerals, meaning a UTP numeral can always be converted to a HOL numeral.

lemma numeral-uexpr-rep-eq [ueval]: 
\begin{align*}
\numeral x \ e b & = \numeral x \\
\text{apply (induct x)} \\
\text{apply (simp add: lit-rep-eq one-uexpr-def)} \\
\text{apply (simp add: bop-rep-eq numeral-Bit0 plus-uexpr-def)} \\
\text{apply (simp add: bop-rep-eq lit-rep-eq numeral-code(3) one-uexpr-def plus-uexpr-def)} \\
\text{done}
\end{align*}

lemma numeral-uexpr-simp:
\numeral x = \langle \numeral x \rangle
by (simp add: uexpr-eq-iff numeral-uexpr-rep-eq lit-rep-eq)

In general unliteralising converts function applications to corresponding expression liftings. Since some operators, like + and *, have specific operators we also have to use u\textit{f} = If
\begin{align*}
(\nu x \nu y) & = bop (\nu x \nu y) \\
0 & = \langle 0 :: ?'a \rangle \\
1 & = \langle 1 :: ?'a \rangle \\
\nu u \nu v & = bop (\nu u \nu v) \\
(\nu P < ?Q) & = (\nu P \leq ?Q \land \neg ?Q \leq ?P) \\
\text{set-of ?t} & = UNIV \text{ in reverse to correctly interpret these. Moreover, numerals must be handled separately by first simplifying them and then converting them into UTP expression numerals; hence the following two simplification rules.}
\end{align*}

lemma lit-numeral-1: uop \numeral x = Abs-uxexpr (\lambda b. \numeral \langle [x]_e b \rangle)
by (simp add: uop-def)
lemma lit-numeral-2: Abs-uepr (λ b. numeral v) = numeral v
by (metis lit.abs-eq lit-numeral)

method literalise = (unfold lit-simps[THEN sym])
method unliteralise = (unfold lit-simps  uepr-defs[THEN sym];
(unfold lit-numeral-1 ; (unfold uepr-defs ueval); (unfold lit-numeral-2))?)

The following tactic can be used to evaluate literal expressions. It first literalises UTP expressions, that is pushes as many operators into literals as possible. Then it tries to simplify, and final unliteralises at the end.

method uepr-simp uses simps = ((literalise)?, simp add: lit-norm simps, (unliteralise)?)

lemma (1::(int, 'a) uepr) + «2» = 4  ↔ «3» = 4
  apply (literalise)
  apply (uepr-simp) oops
end

4  Expression Type Class Instantiations

theory utp-expr-insts
  imports utp-expr
begin

It should be noted that instantiating the unary minus class, uminus, will also provide negation UTP predicates later.

instantiation uepr :: (uminus, type) uminus
begin
  definition uminus-uepr-def [uepr-defs]: − u = uop uminus u
  instance ..
end

instantiation uepr :: (minus, type) minus
begin
  definition minus-uepr-def [uepr-defs]: u − v = bop (−) u v
  instance ..
end

instantiation uepr :: (times, type) times
begin
  definition times-uepr-def [uepr-defs]: u * v = bop times u v
  instance ..
end

instance uepr :: (Rings.dvd, type) Rings.dvd ..

instantiation uepr :: (divide, type) divide
begin
  definition divide-uepr :: ('a, 'b) uepr ⇒ ('a, 'b) uepr ⇒ ('a, 'b) uepr where
    [uepr-defs]: divide-uepr u v = bop divide u v
  instance ..

begin

definition inverse-uexpr :: ('a, 'b) uexpr ⇒ ('a, 'b) uexpr
where [uexpr-defs]: inverse-uexpr u = uop inverse u
instance ..
end

instantiation uexpr :: (modulo, type) modulo
begin
  definition mod-uexpr-def [uexpr-defs]: u mod v = bop (mod) u v
instance ..
end

instantiation uexpr :: (sgn, type) sgn
begin
  definition sgn-uexpr-def [uexpr-defs]: sgn u = uop sgn u
instance ..
end

instantiation uexpr :: (abs, type) abs
begin
  definition abs-uexpr-def [uexpr-defs]: abs u = uop abs u
instance ..
end

Once we’ve set up all the core constructs for arithmetic, we can also instantiate the type classes for various algebras, including groups and rings. The proofs are done by definitional expansion, the transfer tactic, and then finally the theorems of the underlying HOL operators. This is mainly routine, so we don’t comment further.

instance uexpr :: (semigroup-mult, type) semigroup-mult
  by (intro-classes) (simp add: times-uexpr-def one-uexpr-def, transfer, simp add: mult.assoc)+

instance uexpr :: (monoid-mult, type) monoid-mult
  by (intro-classes) (simp add: times-uexpr-def one-uexpr-def, transfer, simp)+

instance uexpr :: (monoid-add, type) monoid-add
  by (intro-classes) (simp add: plus-uexpr-def zero-uexpr-def, transfer, simp)+

instance uexpr :: (ab-semigroup-add, type) ab-semigroup-add
  by (intro-classes) (simp add: plus-uexpr-def, transfer, simp add: add.commute)+

instance uexpr :: (cancel-semigroup-add, type) cancel-semigroup-add
  by (intro-classes) (simp add: plus-uexpr-def, transfer, simp add: fun-eq-iff)+

instance uexpr :: (cancel-ab-semigroup-add, type) cancel-ab-semigroup-add
  by (intro-classes, simp add: plus-uexpr-def minus-uexpr-def, transfer, simp add: fun-eq-iff add.commute cancel-ab-semigroup-add-class.diff-diff-add)+)

instance uexpr :: (group-add, type) group-add
  by (intro-classes)
    (simp add: plus-uexpr-def uminus-uexpr-def minus-uexpr-def zero-uexpr-def, transfer, simp)+

instance uexpr :: (ab-group-add, type) ab-group-add
by (intro-classes)
  (simp add: plus-expr-def minus-expr-def times-expr-def zero-expr-def, transfer, simp)+

instance uexpr :: (semiring, type) semiring
by (intro-classes) (simp add: plus-expr-def times-expr-def, transfer, simp add: fun-eq-iff commute
  semiring-class.distrib-right semiring-class.distrib-left)+

instance uexpr :: (ring-1, type) ring-1
by (intro-classes) (simp add: plus-expr-def minus-expr-def minus-expr-def times-expr-def zero-expr-def
  one-expr-def, transfer, simp add: fun-eq-iff)+

We also lift the properties from certain ordered groups.

instance uexpr :: (ordered-ab-group-add, type) ordered-ab-group-add
by (intro-classes) (simp add: plus-expr-def, transfer, simp)

instance uexpr :: (ordered-ab-group-add-abs, type) ordered-ab-group-add-abs
apply (intro-classes)
  apply (simp add: abs-expr-def zero-expr-def plus-expr-def minus-expr-def, transfer, simp
    add: abs-ge-self abs-le-iff abs-triangle-ineq)+
  apply (metis ab-group-add-class.abs-add-splits abs-ge-self.add-mono-thms-linordered-semiring
    done)

The next theorem lifts powers.

lemma power-rep-eq [uexpr]: \[ [P \cdot n]_e = (\lambda b. [P]_e \cdot b ^ n) \]
by (induct n, simp-all add: lit-rep-eq one-expr-def bop-rep-eq times-expr-def)

lemma of-nat-expr-rep-eq [uexpr]: \[ [\text{of-nat } x]_e = \text{of-nat } x \]
by (induct x, simp-all add: expr-defs ueval)

lemma lit-uminus [lit-simps]: \[ < - x > = - < x > \]
by (simp add: expr-defs, transfer, simp)

lemma lit-minus [lit-simps]: \[ < x - y > = < x > - < y > \]
by (simp add: expr-defs, transfer, simp)

lemma lit-times [lit-simps]: \[ < x \ast y > = < x > \ast < y > \]
by (simp add: expr-defs, transfer, simp)

lemma lit-divide [lit-simps]: \[ < x \div y > = < x > \div < y > \]
by (simp add: expr-defs, transfer, simp)

lemma lit-power [lit-simps]: \[ < x ^ n > = < x > ^ n \]
by (simp add: lit-rep-eq power-rep-eq expr-eq-iff)

4.1 Expression construction from HOL terms

Sometimes it is convenient to cast HOL terms to UTP expressions, and these simplifications
automate this process.

named-theorems mkueexpr

lemma mkueexpr-lens-get [mkueexpr]: \[ mk_e \ get_s = &x \]
by (transfer, simp add: pr-var-def)

lemma mkueexpr-zero [mkueexpr]: \[ mk_e \ (\lambda s. 0) = 0 \]
by (simp add: zero-expr-def, transfer, simp)

lemma mkueexpr-one [mkueexpr]: \[ mk_e \ (\lambda s. 1) = 1 \]
by (simp add: one-expr-def, transfer, simp)

lemma mkueexpr-numeral [mkueexpr]: \[ mk_e \ (\lambda s. \text{numeral } n) = \text{numeral } n \]
using lit-numeral-2 by blast
lemma mkuexpr-lit [mkuexpr]: \( mk_e \ (\lambda \ s. \ k) = \langle k \rangle \)
  by (transfer, simp)

lemma mkuexpr-pair [mkuexpr]: \( mk_e \ (\lambda s. \ (f \ s, \ g \ s)) = (mk_e \ f, \ mk_e \ g)_u \)
  by (transfer, simp)

lemma mkuexpr-plus [mkuexpr]: \( mk_e \ (\lambda s. \ f \ s + g \ s) = mk_e \ f + mk_e \ g \)
  by (simp add: plus-expr-def, transfer, simp)

lemma mkuexpr-uminus [mkuexpr]: \( mk_e \ (\lambda s. \ -f \ s) = -mk_e \ f \)
  by (simp add: uminus-expr-def, transfer, simp)

lemma mkuexpr-minus [mkuexpr]: \( mk_e \ (\lambda s. \ f \ s - g \ s) = mk_e \ f - mk_e \ g \)
  by (simp add: minus-expr-def, transfer, simp)

lemma mkuexpr-times [mkuexpr]: \( mk_e \ (\lambda s. \ f \ s \ * g \ s) = mk_e \ f \ * \ mk_e \ g \)
  by (simp add: times-expr-def, transfer, simp)

lemma mkuexpr-divide [mkuexpr]: \( mk_e \ (\lambda s. \ f \ s \ / g \ s) = \frac{mk_e \ f}{mk_e \ g} \)
  by (simp add: divide-expr-def, transfer, simp)

end

theory utp-expr-funcs
  imports utp-expr-insts
begin

syntax — Polymorphic constructs
-uceil :: logic \Rightarrow logic \ ([\cdot]_u)
-ufloor :: logic \Rightarrow logic \ ([\cdot]_u)
-umin :: logic \Rightarrow logic \ (\text{min} \langle-, -\rangle)
-umax :: logic \Rightarrow logic \ (\text{max} \langle-, -\rangle)
-ugcd :: logic \Rightarrow logic \ (\text{gcd} \langle-, -\rangle)

translations — Type-class polymorphic constructs
\( \text{min}_u(x, \ y) == \text{CONST} \ \text{bop} \ (\text{CONST} \ \text{min}) \ x \ y \)
\( \text{max}_u(x, \ y) == \text{CONST} \ \text{bop} \ (\text{CONST} \ \text{max}) \ x \ y \)
\( \text{gcd}_u(x, \ y) == \text{CONST} \ \text{bop} \ (\text{CONST} \ \text{gcd}) \ x \ y \)
\( \lfloor x \rfloor_u == \text{CONST} \ \text{wop} \ \text{CONST} \ \text{floor} \ x \)
\( \lceil x \rceil_u == \text{CONST} \ \text{wop} \ \text{CONST} \ \text{ceiling} \ x \)

syntax — Lists / Sequences
-ucons :: logic \Rightarrow logic \Rightarrow logic \ (\text{infixr} \ \#_u \ 65)
-unist :: (\text{a list}, \ alpha) \ \text{uxpr} \ (\langle \rangle)
-ulist :: args \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\langle \langle \rangle \rangle)
-uappend :: (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\text{infixr} \ \langle - \rangle_\alpha \ 80)
-ucconcat :: logic \Rightarrow logic \Rightarrow logic \ (\text{infixr} \ \langle - \rangle_\alpha \ 90)
-ucons :: (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\text{last}_u(\cdot))
-ufront :: (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\text{front}_u(\cdot))
-uhead :: (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\text{head}_u(\cdot))
-utail :: (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\text{tail}_u(\cdot))
-utake :: (\text{nat}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\text{take}_u(\cdot/ \cdot))
-udrop :: (\text{nat}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\text{drop}_u(\cdot/ \cdot))
-ufilter :: (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\text{infixl} \ \langle - \rangle_\alpha \ 75)
-uextract :: (\text{a set}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \Rightarrow (\text{a list}, \ alpha) \ \text{uxpr} \ (\text{infixl} \ \langle - \rangle_\alpha \ 75)
-ulems :: (‘a list, ‘a) uexpr ⇒ (‘a set, ‘a) uexpr (elems_u(’))
-usorted :: (‘a list, ‘a) uexpr ⇒ (bool, ‘a) uexpr (sorted_u(’))
-udistinct :: (‘a list, ‘a) uexpr ⇒ (bool, ‘a) uexpr (distinct_u(’))
-upto :: logic ⇒ logic ⇒ logic (⟨...⟩)
-upt :: logic ⇒ logic ⇒ logic (⟨...⟩)
-umap :: logic ⇒ logic ⇒ logic (map_u)
-uzip :: logic ⇒ logic ⇒ logic (zip_u)

translations

x #_u ys == CONST bop (∥) x ys
⟨⟩ == [[]]
⟨x, xs⟩ == x #_u ⟨xs⟩
⟨x⟩ == x #_u [[]]
x #_u y == CONST bop (⋯) x y
A ⊆_u B == CONST bop (∩) A B
last_u(xs) == CONST uop CONST last xs
front_u(xs) == CONST uop CONST butlast xs
head_u(xs) == CONST uop CONST hd xs
tail_u(xs) == CONST uop CONST tl xs
drop_u(n, xs) == CONST bop CONST drop n xs
take_u(n, xs) == CONST bop CONST take n xs
elems_u(xs) == CONST uop CONST set xs
sorted_u(xs) == CONST uop CONST sorted xs
distinct_u(xs) == CONST uop CONST distinct xs
xs ⊂_u A == CONST bop CONST seq-filter xs A
A ⊂_u xs == CONST bop (⊆) A xs
⟨n..k⟩ == CONST bop CONST upto n k
⟨n..<k⟩ == CONST bop CONST up n k
map_u f xs == CONST bop CONST map f xs
zip_u xs ys == CONST bop CONST zip xs ys

syntax — Sets

-ufinite :: logic ⇒ logic (finite_u(’))
-ucmpset :: (‘a set, ‘a) uexpr ([]_u)
-uset :: args ≥––> (‘a set, ‘a) uexpr ([](_))
-union :: (‘a set, ‘a) uexpr ⇒ (‘a set, ‘a) uexpr ⇒ (‘a set, ‘a) uexpr (infixl ∪_u 65)
-inter :: (‘a set, ‘a) uexpr ⇒ (‘a set, ‘a) uexpr ⇒ (‘a set, ‘a) uexpr (infixl ∩_u 70)
-insert :: logic ⇒ logic ⇒ logic (insert_u)
-union :: logic ⇒ logic ⇒ logic (infixl 10)
-uset :: (‘a set, ‘a) uexpr ⇒ (‘a set, ‘a) uexpr ⇒ (‘a set, ‘a) uexpr (infixl ⊆_u 50)
-useteq :: (‘a set, ‘a) uexpr ⇒ (‘a set, ‘a) uexpr ⇒ (bool, ‘a) uexpr (infixl ⊂_u 50)
-union :: logic ⇒ logic (⟨...⟩) [100] 999)
-ucarrier :: type ⇒ logic ([(_)]_u)
-uid :: type ⇒ logic (id[_])
-urelcomp :: logic ⇒ logic ⇒ logic (infixr ×_u 80)
-urelcomp :: logic ⇒ logic ⇒ logic (infixr ×_u 75)

translations

finite_u(x) == CONST uop (CONST finite) x
[]_u == [[]]
insert_u x xs == CONST bop CONST insert x xs
{x, xs}_u == insert_u x {xs}_u
{x}_u == insert_u x [[]]
A ∪_u B == CONST bop (∪) A B
A ∩_u B == CONST bop (∩) A B
translations — syntax

-umap-plus :: logic ⇒ logic ⇒ logic (infixl ⊕ u 85)
-umap-minus :: logic ⇒ logic ⇒ logic (infixl − u 85)

translations

\( f \circ g \Rightarrow \{ \begin{array}{ll}
& f \circ (\cdot) \ pfun \cdot uexpr + g \\
& f \circ (\cdot) \ pfun \cdot uexpr − g 
\end{array} \)

translations

\( \text{inl}_u(x) == \text{CONST } uop \text{ CONST } \text{Inl } x \)
\( \text{inr}_u(x) == \text{CONST } uop \text{ CONST } \text{Inr } x \)

4.2 Lifting set collectors

We provide syntax for various types of set collectors, including intervals and the Z-style set comprehension which is purpose built as a new lifted definition.

syntax — Sum types

-umul :: logic ⇒ logic ⇒ logic (infixl × u \inf )
-umdiv :: logic ⇒ logic ⇒ logic (infixl ÷ u \inf )

translations

\( \{ x . y \}_u \Rightarrow \{ \begin{array}{ll}
& \text{CONST } bop \text{ CONST } \text{atLeastAtMost } x \ y \\
& \text{CONST } bop \text{ CONST } \text{atLeastLessThan } x \ y \\
& \{ x \mid P \cdot F \}_u \Rightarrow \text{CONST } \text{ZedSetCompr } (\text{CONST } \text{lit } \text{CONST } \text{UNIV}) (\lambda x. (P \ F)) \\
& \{ x : A \mid P \cdot F \}_u \Rightarrow \text{CONST } \text{ZedSetCompr } A (\lambda x. (P \ F)) 
\end{array} \)

4.3 Lifting limits

We also lift the following functions on topological spaces for taking function limits, and describing continuity.
definition ulim-left :: 'a::order-topology ⇒ ('a ⇒ 'b) ⇒ 'b::t2-space where
[uxpr-defs]: ulim-left = (λ p f. Lim (at-left p) f)

definition ulim-right :: 'a::order-topology ⇒ ('a ⇒ 'b) ⇒ 'b::t2-space where
[uxpr-defs]: ulim-right = (λ p f. Lim (at-right p) f)

definition ucont-on :: ('a::topological-space ⇒ 'b::topological-space) ⇒ 'a set ⇒ bool where
[uxpr-defs]: ucont-on = (λ f A. continuous-on A f)

syntax
-ulim-left :: id ⇒ logic ⇒ logic ⇒ logic (lim u ('\x' → p) ('\x') e)
-ulim-right :: id ⇒ logic ⇒ logic ⇒ logic (lim u ('\x' → p) ('\x' +) e)
-ucont-on :: logic ⇒ logic ⇒ logic (infix cont-on u 90)

translations
lim_u(x → p)(e) == CONST bop CONST ulim-left p (λ x · e)
lim_u(x → p)(e) == CONST bop CONST ulim-right p (λ x · e)
f cont-on_u A == CONST bop CONST continuous-on A f

lemma uset-minus-empty [simp]: x - { } = x
by (simp add: uexpr-defs, transfer, simp)

lemma uinter-empty-1 [simp]: x ∩ { } = { }
by (transfer, simp)

lemma uinter-empty-2 [simp]: { } ∩ x = { }
by (transfer, simp)

lemma uunion-empty-1 [simp]: { } ∪ x = x
by (transfer, simp)

lemma uunion-insert [simp]: (bop insert x A) ∪_u B = bop insert x (A ∪_u B)
by (transfer, simp)

lemma uist-filter-empty [simp]: x↾_u { } = ⟨⟩
by (transfer, simp)

lemma tail-cons [simp]: tail_u(⟨x⟩穿上xs) = xs
by (transfer, simp)

lemma uconcat-units [simp]: ⟨⟩穿上xs = xs穿上_ua(⟨⟩) = xs
by (transfer, simp)

end

5 Unrestriction

theory utp-unrest
imports utp-expr-insts
begin

5.1 Definitions and Core Syntax

Unrestriction is an encoding of semantic freshness that allows us to reason about the presence of variables in predicates without being concerned with abstract syntax trees. An expression p
is unrestricted by lens \( x \), written \( x \# p \), if altering the value of \( x \) has no effect on the valuation of \( p \). This is a sufficient notion to prove many laws that would ordinarily rely on an \( \text{fv} \) function.

Unrestriction was first defined in the work of Marcel Oliveira \([27, 26]\) in his UTP mechanisation in \textit{ProofPowerZ}. Our definition modifies his in that our variables are semantically characterised as lenses, and supported by the lens laws, rather than named syntactic entities. We effectively fuse the ideas from both Feliachi \([9]\) and Oliveira’s \([26]\) mechanisations of the UTP, the former being also purely semantic in nature.

We first set up overloaded syntax for unrestriction, as several concepts will have this defined.

\begin{verbatim}
consts
  unrest :: 'a ⇒ 'b ⇒ bool

syntax
  -unrest :: salpha ⇒ logic ⇒ logic ⇒ logic (infix # 20)

translations
  -unrest x p == CONST unrest x p
  -unrest (-salphaset (-salphamk (x +L y))) P <= -unrest (x +L y) P
\end{verbatim}

Our syntax translations support both variables and variable sets such that we can write down predicates like \&\( x \# P \) and also \{\&\( x \), \&\( y \), \&\( z \}\} \# P. We set up a simple tactic for discharging unrestriction conjectures using a simplification set.

\begin{verbatim}
named-theorems unrest
method unrest-tac = (simp add: unrest)\
\end{verbatim}

Unrestriction for expressions is defined as a lifted construct using the underlying lens operations. It states that lens \( x \) is unrestricted by expression \( e \) provided that, for any state-space binding \( b \) and variable valuation \( v \), the value which the expression evaluates to is unaltered if we set \( x \) to \( v \) in \( b \). In other words, we cannot effect the behaviour of \( e \) by changing \( x \). Thus \( e \) does not observe the portion of state-space characterised by \( x \). We add this definition to our overloaded constant.

\begin{verbatim}
lift-definition unrest-uexpr :: ('a ⇒ 'α) ⇒ ('b, 'α) uexpr ⇒ bool
is \( \lambda \) x e. \( \forall \) b v. \( e (\text{put}_x b v) = e b \).

adhoc-overloading
  unrest unrest-uexpr

lemma unrest-expr-alt-def:
  weak-lens x ⇒ (x # P) = (∀ b b'. [P]_e (b ⊕L b' on x) = [P]_e b)
  by (transfer, metis lens-override-def weak-lens.put-get)
\end{verbatim}

5.2 Unrestriction laws

We now prove unrestriction laws for the key constructs of our expression model. Many of these depend on lens properties and so variously employ the assumptions \( \text{mwb-lens} \) and \( \text{vwb-lens} \), depending on the number of assumptions from the lenses theory is required.

Firstly, we prove a general property – if \( x \) and \( y \) are both unrestricted in \( P \), then their composition is also unrestricted in \( P \). One can interpret the composition here as a union – if the two sets of variables \( x \) and \( y \) are unrestricted, then so is their union.

\begin{verbatim}
lemma unrest-var-comp unrest[unrest]:
  \[ x # P; y # P \] ⇒ x;y # P
  by (transfer, simp add: lens-defs)
\end{verbatim}
No lens is restricted by a literal, since it returns the same value for any state binding.

If one lens is smaller than another, then any unrestriction on the larger lens implies unrestriction on the smaller.

If two lenses are equivalent, and thus they characterise the same state-space regions, then clearly unrestrictions over them are equivalent.

If we can show that an expression is unrestricted on a bijective lens, then it is unrestricted on the entire state-space.

If an expression is unrestricted by all variables, then it is unrestricted by any variable

We can split an unrestriction composed by lens plus
The following laws demonstrate the primary motivation for lens independence: a variable expression is unrestricted by another variable only when the two variables are independent. Lens independence thus effectively allows us to semantically characterise when two variables, or sets of variables, are different.

**Lemma** unrest-var [unrest]: \[ mwb-lens x; x \triangleright y \] \implies y \sharp var x
by (transfer, auto)

**Lemma** unrest-iuvar [unrest]: \[ mwb-lens x; x \triangleright y \] \implies y \sharp \$x
by (simp add: unrest-var)

**Lemma** unrest-ouvar [unrest]: \[ mwb-lens x; x \triangleright y \] \implies y' \sharp \$x'
by (simp add: unrest-var)

The following laws follow automatically from independence of input and output variables.

**Lemma** unrest-iuvar-ouvar [unrest]:
fixes x :: ('a \Rightarrow 'a)
assumes mwb-lens y
shows $x \sharp y$
by (metis prod.collapse unrest-uexpr.rep-eq.rep-eq var.rep-eq var-lookup-out var-update-in)

**Lemma** unrest-ouvar-iuvar [unrest]:
fixes x :: ('a \Rightarrow 'a)
assumes mwb-lens y
shows $x' \sharp y$
by (simp add: unrest-var)

Unrestriction distributes through the various function lifting expression constructs; this allows us to prove unrestrictions for the majority of the expression language.

**Lemma** unrest-uop [unrest]: \[ x \sharp e \implies x \sharp uop f e \]
by (transfer, simp)

**Lemma** unrest-bop [unrest]: \[ x \sharp u; x \sharp v \implies x \sharp bop f u v \]
by (transfer, simp)

**Lemma** unrest-trop [unrest]: \[ x \sharp u; x \sharp v; x \sharp w \implies x \sharp trop f u v w \]
by (transfer, simp)

**Lemma** unrest-qtop [unrest]: \[ x \sharp u; x \sharp v; x \sharp w; x \sharp y \implies x \sharp qtop f u v w y \]
by (transfer, simp)

For convenience, we also prove unrestriction rules for the bespoke operators on equality, numbers, arithmetic etc.

**Lemma** unrest-eq [unrest]: \[ x \sharp u; x \sharp v \] \implies x \sharp u =_u v
by (simp add: eq-upred-def, transfer, simp)

**Lemma** unrest-zero [unrest]: \[ x \sharp 0 \]
by (simp add: unrest-lit zero-uexpr-def)

**Lemma** unrest-one [unrest]: \[ x \sharp 1 \]
by (simp add: one-uexpr-def unrest-lit)

**Lemma** unrest-numeral [unrest]: \[ x \sharp (numeral n) \]
by (simp add: numeral-uexpr-simp unrest-lit)
lemma unrest-sgn [unrest]: \( x \uplus u \Rightarrow x \uplus \operatorname{sgn} u \)  
  by (simp add: sgn-uexpr-def unrest-uop)

lemma unrest-abs [unrest]: \( x \uplus u \Rightarrow x \uplus \operatorname{abs} u \)  
  by (simp add: abs-uexpr-def unrest-uop)

lemma unrest-plus [unrest]: \( \begin{bmatrix} x \uplus u; x \uplus v \end{bmatrix} \Rightarrow x \uplus u + v \)  
  by (simp add: plus-uexpr-def unrest)

lemma unrest-uminus [unrest]: \( x \uplus u \Rightarrow x \uplus -u \)  
  by (simp add: uminus-uexpr-def unrest)

lemma unrest-minus [unrest]: \( \begin{bmatrix} x \uplus u; x \uplus v \end{bmatrix} \Rightarrow x \uplus u - v \)  
  by (simp add: minus-uexpr-def unrest)

lemma unrest-times [unrest]: \( \begin{bmatrix} x \uplus u; x \uplus v \end{bmatrix} \Rightarrow x \uplus u \ast v \)  
  by (simp add: times-uexpr-def unrest)

lemma unrest-divide [unrest]: \( \begin{bmatrix} x \uplus u; x \uplus v \end{bmatrix} \Rightarrow x \uplus u / v \)  
  by (simp add: divide-uexpr-def unrest)

lemma unrest-case-prod [unrest]: \( \begin{bmatrix} \forall i j. x \uplus P i j \end{bmatrix} \Rightarrow x \uplus \operatorname{case-prod} P v \)  
  by (simp add: prod-split-sel-asn)

For a \( \lambda \)-term we need to show that the characteristic function expression does not restrict \( v \) for any input value \( x \).

lemma unrest-ulambda [unrest]: \( \begin{bmatrix} \forall x. v \uplus F x \end{bmatrix} \Rightarrow v \uplus (\lambda x. F x) \)  
  by (transfer, simp)

end

6 Used-by

theory utp-usedby
  imports utp-unrest
begin

The used-by predicate is the dual of unrestriction. It states that the given lens is an upper-bound on the size of state space the given expression depends on. It is similar to stating that the lens is a valid alphabet for the predicate. For convenience, and because the predicate uses a similar form, we will reuse much of unrestriction’s infrastructure.

consts
  usedBy :: \(\alpha \Rightarrow \beta \Rightarrow \operatorname{bool} \)

syntax
  -usedBy :: salpha \Rightarrow logic \Rightarrow logic \Rightarrow logic (infix \(\uplus\) 20)

translations
  -usedBy x p == CONST usedBy x p
  -usedBy (-salphaset (-salphamk (x +L y))) P <= -usedBy (x +L y) P

lift-definition usedBy-uexpr :: ('a \Rightarrow 'a) \Rightarrow ('a, 'a) uexpr \Rightarrow bool
  is \lambda x e. \forall b b'. e (b' \oplus_L b \text{ on } x) = e b .

30
神通变换 usedBy usedBy-uexpr

**lemma** usedBy-lit [unrest]: \( x \vDash v \)
  by (transfer, simp)

**lemma** usedBy-sublens:
  fixes \( P :: (\alpha', \alpha) \) uexpr
  assumes \( x \vDash P \subseteq_L y \) vwb-lens y
  shows \( y \vDash P \)
  using assms
  by (transfer, auto, metis Lens-Order.lens-override-idem lens-override-def sublens-obs-get vwb-lens-mwb)

**lemma** usedBy-sear [unrest]: \( x \vDash P \implies \& x \vDash P \)
  by (transfer, simp add: lens-defs)

**lemma** usedBy-lens-plus-1 [unrest]: \( x \vDash P \implies x; y \vDash P \)
  by (transfer, simp add: lens-defs)

**lemma** usedBy-lens-plus-2 [unrest]: \([x \vDash y; y \vDash P]\) \implies x; y \vDash P
  by (transfer, auto simp add: lens-defs lens-indep-comm)

Linking used-by to unrestriction: if \( x \) is used-by \( P \), and \( x \) is independent of \( y \), then \( P \) cannot depend on any variable in \( y \).

**lemma** usedBy-indep-uses:
  fixes \( P :: (\alpha', \alpha) \) uexpr
  assumes \( x \vDash P \vDash y \)
  shows \( y \vDash P \)
  using assms
  by (transfer, auto, metis lens-indep-get lens-override-def)

**lemma** usedBy-var [unrest]:
  assumes vwb-lens \( x y \subseteq_L x \)
  shows \( x \vDash \text{var} y \)
  using assms
  by (transfer, simp add: uexpr-defs pr-var-def)
  (metis lens-override-def sublens-obs-get vwb-lens-def wb-lens.get-put)

**lemma** usedBy-uop [unrest]: \( x \vDash e \implies x \vDash u \circ f \circ e \)
  by (transfer, simp)

**lemma** usedBy-bop [unrest]: \([x \vDash u; x \vDash v]\) \implies x \vDash \text{bop} u v
  by (transfer, simp)

**lemma** usedBy-trop [unrest]: \([x \vDash u; x \vDash v; x \vDash w]\) \implies x \vDash \text{trop} f u v w
  by (transfer, simp)

**lemma** usedBy-qtop [unrest]: \([x \vDash u; x \vDash v; x \vDash w; x \vDash y]\) \implies x \vDash \text{qtop} f u v w y
  by (transfer, simp)

For convenience, we also prove used-by rules for the bespoke operators on equality, numbers, arithmetic etc.

**lemma** usedBy-eq [unrest]: \([x \vDash u; x \vDash v]\) \implies x \vDash u =_a v
  by (simp add: eq-upred-def, transfer, simp)

**lemma** usedBy-zero [unrest]: \( x \vDash 0 \)

31
by (simp add: usedBy-lit zero-uexpr-def)

lemma usedBy-one [unrest]: \( x \uparrow 1 \)
by (simp add: one-uexpr-def usedBy-lit)

lemma usedBy-numeral [unrest]: \( x \uparrow (\text{numeral } n) \)
by (simp add: numeral-uexpr-simp usedBy-lit)

lemma usedBy-sgn [unrest]: \( x \uparrow u \Rightarrow x \uparrow \text{sgn } u \)
by (simp add: sgn-uexpr-def usedBy-uop)

lemma usedBy-abs [unrest]: \( x \uparrow u \Rightarrow x \uparrow \text{abs } u \)
by (simp add: abs-uexpr-def usedBy-uop)

lemma usedBy-plus [unrest]: \[\[ x \uparrow u; x \uparrow v \]\] \Rightarrow x \uparrow u + v
by (simp add: plus-uexpr-def unrest)

lemma usedBy-uminus [unrest]: \( x \uparrow u \Rightarrow x \uparrow - u \)
by (simp add: uminus-uexpr-def unrest)

lemma usedBy-minus [unrest]: \[\[ x \uparrow u; x \uparrow v \]\] \Rightarrow x \uparrow u - v
by (simp add: minus-uexpr-def unrest)

lemma usedBy-times [unrest]: \[\[ x \uparrow u; x \uparrow v \]\] \Rightarrow x \uparrow u * v
by (simp add: times-uexpr-def unrest)

lemma usedBy-divide [unrest]: \[\[ x \uparrow u; x \uparrow v \]\] \Rightarrow x \uparrow u / v
by (simp add: divide-uexpr-def unrest)

lemma usedBy-ulambda [unrest]:
\[\[ \lambda x \cdot F x \]\] \Rightarrow v \uparrow (\lambda x \cdot F x)
by (transfer, simp)

lemma unrest-var-sep [unrest]:
vwb-lens x \Rightarrow x \uparrow \& x:y
by (transfer, simp add: lens-defs)

end

7 Substitution

theory utp-subst
imports
  utp-expr
  utp-unrest
begin

7.1 Substitution definitions

Variable substitution, like unrestricted, will be characterised semantically using lenses and
state-spaces. Effectively a substitution \( \sigma \) is simply a function on the state-space which can be
applied to an expression \( e \) using the syntax \( \sigma \uparrow e \). We introduce a polymorphic constant that
will be used to represent application of a substitution, and also a set of theorems to represent
laws.
named-theorems usubst

A substitution is simply a transformation on the alphabet; it shows how variables should be mapped to different values. Most of the time these will be homogeneous functions but for flexibility we also allow some operations to be heterogeneous.

type-synonym ('α, 'β) psubst = 'α ⇒ 'β

type-synonym 'α usubst = 'α ⇒ 'α

Application of a substitution simply applies the function σ to the state binding b before it is handed to e as an input. This effectively ensures all variables are updated in e.

lift-definition subst :: ('α, 'β) psubst ⇒ ('a, 'α) uexpr ⇒ ('α, 'β) psubst

Substitutions can be updated by associating variables with expressions. We thus create an additional polymorphic constant to represent updating the value of a variable to an expression in a substitution, where the variable is modelled by type 'v. This again allows us to support different notions of variables, such as deep variables, later.

consts subst-upd :: ('α, 'β) psubst ⇒ ('v) ⇒ ('α, 'β) psubst

The following function takes a substitution form state-space 'α to 'β, a lens with source 'β and view 'a", and an expression over "'a, and returning a value of type "'a, and produces an updated substitution. It does this by constructing a substitution function that takes state binding b, and updates the state first by applying the original substitution σ, and then updating the part of the state associated with lens x with expression evaluated in the context of b. This effectively means that x is now associated with expression v. We add this definition to our overloaded constant.

definition subst-upd-uvar :: ('α, 'β) psubst ⇒ 'v ⇒ ('a, 'α) uexpr ⇒ ('α, 'β) psubst where subst-upd-uvar σ x v = \( \lambda b. \text{put}_x (\sigma b) ([v]_e b) \)

adhoc-overloading

subst-upd subst-upd-uvar

Substitutions also exhibit a natural notion of unrestriction which states that σ does not restrict x if application of σ to an arbitrary state ρ will not effect the valuation of x. Put another way, it requires that put and the substitution commute.

definition unrest-usubst :: ('α) ⇒ 'α usubst ⇒ bool

where unrest-usubst x σ = (\( \forall \rho \ v. \sigma (\text{put}_x \rho \ v) = \text{put}_x (\sigma \rho) \ v) \)

adhoc-overloading

unrest unrest-usubst
A conditional substitution deterministically picks one of the two substitutions based on a Boolean expression which is evaluated on the present state-space. It is analogous to a functional if-then-else.

**definition** cond-subst :: \( \alpha \) usubst \( \Rightarrow \) (bool, \( \alpha \)) uexpr \( \Rightarrow \) \( \alpha \) usubst \( \Rightarrow \) \( \alpha \) usubst ((3- \( \alpha \) - \( v_s \)/ -) [52,0,53] 52) where 
cond-subst \( \sigma \) b \( \varrho \) = (\( \lambda \) s. if \( [b]_e \) s then \( \sigma \)(s) else \( \varrho \)(s))

Parallel substitutions allow us to divide the state space into three segments using two lenses, A and B. They correspond to the part of the state that should be updated by the respective substitution. The two lenses should be independent. If any part of the state is not covered by either lenses then this area is left unchanged (framed).

**definition** par-subst :: \( \alpha \) usubst \( \Rightarrow \) (\( 'a \) \( \Rightarrow \) \( 'a \)) \( \Rightarrow \) (\( 'b \) \( \Rightarrow \) \( 'a \)) \( \Rightarrow \) \( \alpha \) usubst \( \Rightarrow \) \( \alpha \) usubst where 
par-subst \( \sigma_1 \) A B \( \sigma_2 \) = (\( \lambda \) s. (s \( \oplus \) \( \sigma_1 \)(s) on A) \( \oplus \) \( \sigma_2 \)(s) on B)

### 7.2 Syntax translations

We support two kinds of syntax for substitutions, one where we construct a substitution using a maplet-style syntax, with variables mapping to expressions. Such a constructed substitution can be applied to an expression. Alternatively, we support the more traditional notation, \( P[v/x] \), which also support multiple simultaneous substitutions. We have to use double square brackets as the single ones are already well used.

We set up non-terminals to represent a single substitution maplet, a sequence of maplets, a list of expressions, and a list of alphabets. The parser effectively uses subst-upd to construct substitutions from multiple variables.

**nonterminal** smaplet and smaplets and uexp and uexprs and salphas

**syntax**

- smaplet :: [salphas, \[a\]] => smaplet (\( \rightarrow \rightarrow \rightarrow \) )
- smaplets => smaplets (\( \rightarrow \) )
- SMaplets :: [smaplet, smaplets] => smaplets (\( \rightarrow \rightarrow \rightarrow \) )
- SubstUpd :: [\( 'm \) usubst, smaplets] => \( 'm \) usubst (\( \rightarrow \rightarrow \) ) [900,0] 900
- Subst :: smaplets => \( 'a \) => \( 'b \) (([\( 'a\) ]))
- psubst :: [logic, vs, uexprs] => logic
- subst :: logic => uexp => uexprs => salphas => logic (\( \rightarrow \rightarrow \rightarrow \) ) [990,0] 0091
- uexpr-l :: logic => uexp (\( \rightarrow \rightarrow \rightarrow \) ) [64] 64
- uexprs :: [uexp, uexprs] => uexprs (\( \rightarrow \rightarrow \rightarrow \) )
- salphas :: [salpha, salphas] => salphas (\( \rightarrow \rightarrow \) )
- salpha => salphas (\( \rightarrow \rightarrow \) )
- par-subst :: logic => salpha => salpha => logic => logic (\( \rightarrow \rightarrow \rightarrow \) ) [100,0,0,101] 101

**translations**

- SubstUpd m (SMaplets xy ms) => -SubstUpd (-SubstUpd m xy) ms
- SubstUpd m (smaplet x y) => CONST subst-upd m x y
- Subst m => -SubstUpd (CONST id) m
- Subst (SMaplets ms1 ms2) <= -SubstUpd (-Subst ms1) ms2
- SMaplets ms1 (SMaplets ms2 ms3) <= -SMaplets (SMaplets ms1 ms2) ms3
- subst P vs vs => CONST subst (psubst (CONST id) vs vs) P
- psubst m (salphas x xs) (uexprs v us) => -psubst (psubst m x v) xs vs
- psubst m x v => CONST subst-upd m x v
- subst P x v <= CONST subst (CONST subst-upd (CONST id) x v) P
- subst P x v <= -subst P (svar x) v
- par-subst \( \sigma_1 \) A B \( \sigma_2 \) => CONST par-subst \( \sigma_1 \) A B \( \sigma_2 \)
Thus we can write things like $\sigma(x \mapsto_s v)$ to update a variable $x$ in $\sigma$ with expression $v$, $(x \mapsto_s e, y \mapsto_s f)$ to construct a substitution with two variables, and finally $P[v/x]$, the traditional syntax.

We can now express deletion of a substitution maplet.

**Definition:** subst-del :: `\'a usubst \Rightarrow (\'a \Rightarrow \'a) \Rightarrow \'a usubst (infix \_ _ 85)` where subst-del $\sigma x = \sigma(x \mapsto_s \_ \_ \& x)$

### 7.3 Substitution Application Laws

We set up a simple substitution tactic that applies substitution and unrestriction laws

**Method:** subst-tac = (simp add: usubst unrest)?

Evaluation of a substitution expression involves application of the substitution to different variables. Thus we first prove laws for these cases. The simplest substitution, $id$, when applied to any variable $x$ simply returns the variable expression, since $id$ has no effect.

**Lemma:** usubst-lookup-id [usubst]: $(id)_s x = \text{var } x$

by (transfer, simp)

**Lemma:** sub-upd-id-lam [usubst]: subst-upd ($\lambda x. x$) $x v = \text{subst-upd } id x v$

by (simp add: id-def)

A substitution update naturally yields the given expression.

**Lemma:** usubst-lookup-upd [usubst]:

assumes weak-lens $x$

shows $(\sigma(x \mapsto_s v))_s x = v$

using assms

by (simp add: subst-upd-uvar-def assms comp-def)

**Lemma:** usubst-lookup-upd-pr-var [usubst]:

assumes weak-lens $x$

shows $(\sigma(x \mapsto_s v))_s (pr-var x) = v$

using assms

by (simp add: subst-upd-uvar-def assms comp-def)

**Lemma:** usubst-upd-idem [usubst]:

assumes mwb-lens $x$

shows $\sigma(x \mapsto_s u, x \mapsto_s v) = \sigma(x \mapsto_s v)$

by (simp add: subst-upd-uvar-def assms comp-def)

**Lemma:** usubst-upd-idem-sub [usubst]:

assumes $x \subseteq_L y$ mwb-lens $y$

shows $\sigma(x \mapsto_s u, y \mapsto_s v) = \sigma(y \mapsto_s v)$

by (simp add: subst-upd-uvar-def assms comp-def fun-eq-iff sublens-put-put)

Substitution updates commute when the lenses are independent.

**Lemma:** usubst-upd-comm:

assumes $x \triangleright y$

shows $\sigma(x \mapsto_s u, y \mapsto_s v) = \sigma(y \mapsto_s v, x \mapsto_s u)$

using assms

by (rule-tac ext, auto simp add: subst-upd-uvar-def assms comp-def lens-indep-comm)
lemma usubst-upd-comm2:
assumes \( z \triangleright_{\sigma} y \)
shows \( \sigma(x \mapsto_{s} u, y \mapsto_{s} v, z \mapsto_{s} s) = \sigma(x \mapsto_{s} u, z \mapsto_{s} s, y \mapsto_{s} v) \)
using assms
by (rule-tac ext, auto simp add: subst-upd-uvar-def assms comp-def lens-indep-comm)

lemma subst-upd-pr-var: \( s(\& x \mapsto_{s} v) = s(x \mapsto_{s} v) \)
by (simp add: pr-var-def)

A substitution which swaps two independent variables is an injective function.

lemma swap-usubst-inj:
fixes \( x \) \( y \) :: ('a \Rightarrow 'a)
assumes vwb-lens \( x \) vwb-lens \( y \) \( x \triangleright_{\sigma} y \)
shows inj \( x \mapsto_{s} y, y \mapsto_{s} \& x \)
proof (rule injI)
fix \( b_{1} \) :: 'a \text{ and } \( b_{2} \) :: 'a
assume \( [x \mapsto_{s} y, y \mapsto_{s} \& x] \)
\( b_{1} = [x \mapsto_{s} y, y \mapsto_{s} \& x] b_{2} \)
hence \( \exists b. \text{ put}_{y} (\text{ put}_{x} b_{1} ((\& y) b_{1}) ((\& x) b_{1}) b_{2}) = \text{ put}_{y} (\text{ put}_{x} b_{2} ((\& y) b_{2}) ((\& x) b_{2}) \)
\( b \) by (auto simp add: subst-upd-uvar-def)
then have \( (\forall a b c. \text{ put}_{x} (\text{ put}_{y} a b) c = \text{ put}_{y} (\text{ put}_{x} a c) b) \land (\forall a b. \text{ get}_{x} (\text{ put}_{y} a b) = \text{ get}_{x} a) \land (\forall a b. \text{ get}_{y} (\text{ put}_{x} a b) = \text{ get}_{y} a) \)
\( b \) by (simp add: assms(3) lens-indep.lens-put-irr2 lens-indep-comm)
then show \( b_{1} = b_{2} \)
\( b \) by (metis a assms(1) assms(2) pr-var-def var.rep-eq vwb-lens.source-determination vwb-lens-def wb-lens-def weak-lens.put-get)
qed

lemma usubst-upd-var-id [usubst]:
vwb-lens \( x \) \( x \mapsto_{s} \text{ var } x \) = id
apply (simp add: subst-upd-uvar-def)
apply (transfer)
apply (rule ext)
apply (auto)
done

lemma usubst-upd-pr-var-id [usubst]:
vwb-lens \( x \) \( x \mapsto_{s} \text{ var } (\text{ pr-var } x) \) = id
apply (simp add: subst-upd-uvar-def pr-var-def)
apply (transfer)
apply (rule ext)
apply (auto)
done

lemma usubst-upd-comm-dash [usubst]:
fixes \( x \) :: ('a \Rightarrow 'a)
shows \( \sigma(\$ x \mapsto_{s} v, \$ x \mapsto_{s} u, \$ x' \mapsto_{s} v) \)
using out-indep usubst-upd-comm by blast

lemma subst-upd-lens-plus [usubst]:
subst-upd \( \sigma (x +_{L} y) \langle (u,v) \rangle = \sigma(y \mapsto_{s} \langle v \rangle, x \mapsto_{s} \langle u \rangle) \)
by (simp add: lens-defs uexpr-defs subst-upd-uvar-def, transfer, auto)

lemma subst-upd-in-lens-plus [usubst]:
subst-upd \( \sigma (\text{ivar } (x +_{L} y)) \langle (u,v) \rangle = \sigma(\$ y \mapsto_{s} \langle v \rangle, \$ x \mapsto_{s} \langle u \rangle) \)

36
by (simp add: lens-defs uexpr-defs subst-upd-var-def, transfer, auto simp add: prod.case-eq-if)

lemma subst-upd-out-lens-plus [usubst]:
  subst-upd $σ$ (ovar $(x + L, y)$) $(u, v)$ = $σ$($\$y' \mapsto_s v$, $\$x' \mapsto_s u$)
by (simp add: lens-defs uexpr-defs subst-upd-var-def, transfer, auto simp add: prod.case-eq-if)

lemma usubst-lookup-upd-indep [usubst]:
  assumes mwb-lens $x$ $x \sim y$
  shows $(σ(y \mapsto_s v))_s x = (σ)_s x$
  using assms
by (simp add: subst-upd-var-def, transfer, simp)

lemma subst-upd-plus [usubst]:
  $x \sim y \Rightarrow$ subst-upd $(x + L, y)$ $v$ $e$ = $s(x \mapsto_s π_1(e), y \mapsto_s π_2(e))$

If a variable is unrestricted in a substitution then it’s application has no effect.

lemma usubst-apply-unrest [usubst]:
  [ vwb-lens $x$; $x \sim σ$ ] = $(σ)_s x = \text{var } x$

There follows various laws about deleting variables from a substitution.

lemma subst-del-id [usubst]:
  vwb-lens $x$ = id $s$, $x = id$
by (simp add: subst-del-def subst-upd-var-def pr-var-def, transfer, auto)

lemma subst-del-upd-same [usubst]:
  mwb-lens $x$ = $σ(x \mapsto_s v) - s$, $x = $ $σ - s$, $x$
by (simp add: subst-del-def subst-upd-var-def)

lemma subst-del-upd-diff [usubst]:
  $x \sim y \Rightarrow σ(y \mapsto_s v) - s$ $x = (σ - s)_s y \mapsto_s v$
by (simp add: subst-del-def subst-upd-var-def lens-indep-comm)

If a variable is unrestricted in an expression, then any substitution of that variable has no effect on the expression .

lemma subst-unrest [usubst]: $x$ \notin $P$ = $σ(x \mapsto_s v) \vdash P = σ \vdash P$
by (simp add: subst-upd-var-def, transfer, auto)

lemma subst-unrest-2 [usubst]:
  fixes $P :: (α', α)$ uexpr
  assumes $x \notin P \sim x \sim y$
  shows $σ(x \mapsto_s u, y \mapsto_s v) \vdash P = σ(y \mapsto_s v) \vdash P$
  using assms
by (simp add: subst-upd-var-def, transfer, auto, metis lens-indep.lens-put-comm)

lemma subst-unrest-3 [usubst]:
  fixes $P :: (α', α)$ uexpr
  assumes $x \notin P \sim y \sim x \sim z$
  shows $σ(x \mapsto_s u, y \mapsto_s v, z \mapsto_s w) \vdash P = σ(y \mapsto_s v, z \mapsto_s w) \vdash P$
  using assms
by (simp add: subst-upd-var-def, transfer, auto, metis (no-types, hide-lams) lens-indep-comm)

lemma subst-unrest-4 [usubst]:
fixes $P :: (\alpha, 'a) \text{ uexpr}$
assumes $x \notin P \ x \ni y \ x \ni z \ x \ni u$
shows $\sigma(x \mapsto e, y \mapsto f, z \mapsto g, u \mapsto h) \vdash P = \sigma(y \mapsto f, z \mapsto g, u \mapsto h) \vdash P$
using assms
by (simp add: subst-upd-uvar-def, transfer, auto, metis (no-types, hide-lams) lens-indep-comm)

lemma subst-unrest-5 [subst]:
fixes $P :: (\alpha, 'a) \text{ uexpr}$
assumes $x \notin P \ x \ni y \ x \ni z \ x \ni u \ x \ni v$
shows $\sigma(x \mapsto e, y \mapsto f, z \mapsto g, u \mapsto h, v \mapsto i) \vdash P = \sigma(y \mapsto f, z \mapsto g, u \mapsto h, v \mapsto i) \vdash P$
using assms
by (simp add: subst-upd-uvar-def, transfer, auto, metis (no-types, hide-lams) lens-indep-comm)

lemma subst-compose-upd [subst]: $x \notin \sigma \implies \sigma \circ \rho(x \mapsto v) = (\sigma \circ \rho)(x \mapsto v)$
by (simp add: subst-upd-uvar-def, transfer, auto simp add: unrest-usubst-def)

Any substitution is a monotonic function.

lemma subst-mono: mono (subst $\sigma$)
by (simp add: less-eq-uexpr.rep-eq mono-def subst.rep-eq)

7.4 Substitution laws

We now prove the key laws that show how a substitution should be performed for every expression operator, including the core function operators, literals, variables, and the arithmetic operators. They are all added to the subst theorem attribute so that we can apply them using the substitution tactic.

lemma id-subst [subst]: $id \vdash v = v$
by (transfer, simp)

lemma subst-lit [subst]: $\sigma \vdash \langle v \rangle = \langle v \rangle$
by (transfer, simp)

lemma subst-var [subst]: $\sigma \vdash \text{var } x = \langle \sigma \rangle_s \ x$
by (transfer, simp)

lemma usubst-ulambda [subst]: $\sigma \vdash (\lambda x \cdot \text{P}(x)) = (\lambda x \cdot \sigma \vdash \text{P}(x))$
by (transfer, simp)

lemma unrest-usubst-del [unrest]: $\text{[ vwb-lens } x; x \notin \langle (\sigma)_s \ x \rangle; x \notin \sigma\_s \ x \implies x \notin (\sigma \vdash P)\$
by (simp add: subst-del-uvar-def unrest-ucexpr-del unrest-usubst-def subst.rep-eq usubst-lookup.rep-eq)

We add the symmetric definition of input and output variables to substitution laws so that the variables are correctly normalised after substitution.

lemma subst-uop [subst]: $\sigma \vdash \text{uop } f \ v = \text{uop } f \ (\sigma \vdash v)$
by (transfer, simp)

lemma subst-bop [subst]: $\sigma \vdash \text{bop } f \ u \ v = \text{bop } f \ (\sigma \vdash u) \ (\sigma \vdash v)$
by (transfer, simp)

lemma subst-trop [subst]: $\sigma \vdash \text{trop } f \ u \ v \ w = \text{trop } f \ (\sigma \vdash u) \ (\sigma \vdash v) \ (\sigma \vdash w)$
by (transfer, simp)

lemma subst-qtop [subst]: $\sigma \vdash \text{qtop } f \ u \ v \ w \ x = \text{qtop } f \ (\sigma \vdash u) \ (\sigma \vdash v) \ (\sigma \vdash w) \ (\sigma \vdash x)$
by (transfer, simp)

lemma subst-case-prod [usubst]:
  fixes P :: '('i ⇒ 'j ⇒ ('a, 'α) uexpr)
  shows σ † case-prod (λ x y. P x y) v = case-prod (λ x y. σ † P x y) v
  by (simp add: case-prod-beta')

lemma subst-plus [usubst]: σ † (x + y) = σ † x + σ † y
  by (simp add: plus-uexpr-def subst-bop)

lemma subst-times [usubst]: σ † (x * y) = σ † x * σ † y
  by (simp add: times-uexpr-def subst-bop)

lemma subst-mod [usubst]: σ † (x mod y) = σ † x mod σ † y
  by (simp add: mod-uexpr-def usubst)

lemma subst-div [usubst]: σ † (x div y) = σ † x div σ † y
  by (simp add: divide-uexpr-def usubst)

lemma subst-minus [usubst]: σ † (x − y) = σ † x − σ † y
  by (simp add: minus-uexpr-def subst-bop)

lemma subst-uminus [usubst]: σ † (− x) = − (σ † x)
  by (simp add: uminus-uexpr-def subst-uop)

lemma usubst-sgn [usubst]: σ † sgn x = sgn (σ † x)
  by (simp add: sgn-uexpr-def subst-uop)

lemma usubst-abs [usubst]: σ † abs x = abs (σ † x)
  by (simp add: abs-uexpr-def subst-uop)

lemma subst-zero [usubst]: σ † 0 = 0
  by (simp add: zero-uexpr-def subst-lit)

lemma subst-one [usubst]: σ † 1 = 1
  by (simp add: one-uexpr-def subst-lit)

lemma subst-eq-upred [usubst]: σ † (x =_u y) = (σ † x =_u σ † y)
  by (simp add: eq-upred-def usubst)

  This laws shows the effect of applying one substitution after another – we simply use function composition to compose them.

lemma subst-subst [usubst]: σ † ϱ † e = (ϱ ◦ σ) † e
  by (transfer, simp)

The next law is similar, but shows how such a substitution is to be applied to every updated variable additionally.

lemma subst-upd-comp [usubst]:
  fixes x :: ('a ⇒ 'α)
  shows ϱ(x ↦→ s v) ◦ σ = (ϱ ◦ σ)(x ↦→ σ † v)
  by (rule ext, simp add: uexpr-defs subst-upd-uvar-def, transfer, simp)

lemma subst-singleton:
  fixes x :: ('a ⇒ 'α)
  assumes x # σ
shows $\sigma(x \mapsto_s v) \uparrow P = (\sigma \uparrow P)[v/x]$

using assms
by (simp add: usubst)

lemmas subst-to-singleton = subst-singleton id-subst

7.5 Ordering substitutions

A simplification procedure to reorder substitutions maplets lexicographically by variable syntax

\[ \langle \text{fn} \mapsto \text{fn ctxt} \mapsto \text{fn ct} \mapsto \rangle \]
\[ \text{case} (\text{Thm.term-of ct}) \text{ of} \]
\[ \text{Const} (\text{utp-subst}.subst-upd-uvar, -) \mapsto \text{if} (\text{YXML.content-of (Syntax.string-of-term ctxt x)} > \text{YXML.content-of (Syntax.string-of-term ctxt y)}) \]
\[ \text{then SOME (mk-meta-\text{eq} \@ \{thm usubst-upd-comm\})} \]
\[ \text{else NONE |} \]
\[- \mapsto \text{NONE} \]
\[ \langle \text{fn} - \mapsto \text{fn ctxt} > \text{fn ct} > \rangle \]

7.6 Unrestriction laws

These are the key unrestricted theorems for substitutions and expressions involving substitutions.

\textbf{lemma unrest-usubst-single} \ [unrest]:
\[ [[\text{mwb-lens x}; x \uparrow v]] \Rightarrow x \uparrow P[v/x] \]
by (transfer, simp add: subst-upd-uvar-def unrest-uexpr-def)

\textbf{lemma unrest-usubst-id} \ [unrest]:
\[ \text{mwb-lens x} \Rightarrow x \uparrow \text{id} \]
by (simp add: unrest-usubst-def)

\textbf{lemma unrest-usubst-upd} \ [unrest]:
\[ [[x \uparrow y]; x \uparrow \sigma; x \vdash v] \Rightarrow x \uparrow \sigma(y \mapsto_s v) \]

\textbf{lemma unrest-subst} \ [unrest]:
\[ [[x \uparrow P]; x \uparrow \sigma] \Rightarrow x \uparrow (\sigma \uparrow P) \]
by (transfer, simp add: unrest-usubst-def)

7.7 Conditional Substitution Laws

\textbf{lemma usubst-cond-upd-1} \ [usubst]:
\[ \sigma(x \mapsto_s u) \triangleleft b \triangleright x \cdot g(x \mapsto_s v) = (\sigma \triangleleft b \triangleright u \cdot g)(x \mapsto_s u \triangleleft b \triangleright v) \]
by (simp add: cond-subst-def subst-upd-uvar-def uexpr-defs, transfer, auto)

\textbf{lemma usubst-cond-upd-2} \ [usubst]:
\[ [[\text{vwb-lens x}; x \vdash g]] \Rightarrow \sigma(x \mapsto_s u) \triangleleft b \triangleright x \cdot g = (\sigma \triangleleft b \triangleright u \cdot g)(x \mapsto_s u \triangleleft b \triangleright \& x) \]
by (simp add: subst-upd-uvar-def unrest-usubst-def uexpr-defs, transfer)
\[ \text{(metis (full-\text{types}, \text{hide-\text{lams}}) id-apply pr-\text{var-def subst-upd-uvar-def usubst-upd-pr-\text{var-id var.rep-eq})} \]

\textbf{lemma usubst-cond-upd-3} \ [usubst]:
\[ [[\text{vwb-lens x}; x \vdash \sigma]] \Rightarrow \sigma \triangleleft b \triangleright x \cdot g(x \mapsto_s v) = (\sigma \triangleleft b \triangleright \& x \triangleleft b \triangleright v) \]
by (simp add: cond-subst-def subst-upd-uvar-def unrest-usubst-def uexpr-defs, transfer)
lemma usubst-cond-id [usubst]:
\[ \sigma \circ b \triangleright_{s} \sigma = \sigma \]
by (auto simp add: cond-subst-def)

\section{Parallel Substitution Laws}

\subsection{Parallel Substitution Laws}

lemma par-subst-id [usubst]:
\[ [\text{vwb-lens } A; \text{vwb-lens } B] \implies \text{id } [A|B] s = \text{id} \]
by (simp add: par-subst-def id-def)

lemma par-subst-left-empty [usubst]:
\[ [\text{vwb-lens } A] \implies \sigma [\emptyset|A] s = \text{id} [\emptyset|A] s \]
by (simp add: par-subst-def pr-var-def)

lemma par-subst-right-empty [usubst]:
\[ [\text{vwb-lens } A] \implies \sigma [A|\emptyset] s = \sigma [A|\emptyset] s \]
by (simp add: par-subst-def pr-var-def)

lemma par-subst-comm:
\[ [A \triangleright_{s} B] \implies \sigma [A|B] s = \sigma [B|A] s \]
by (simp add: par-subst-def lens-override-def lens-indep-comm)

lemma par-subst-upd-left-in [usubst]:
\[ [\text{vwb-lens } A; A \triangleright_{s} B; x \subseteq_{L} A] \implies \sigma (x \mapsto_{s} v) [A|B] s = (\sigma [A|B] s)(x \mapsto_{s} v) \]

lemma par-subst-upd-left-out [usubst]:
\[ [\text{vwb-lens } A; x \triangleright_{s} A] \implies \sigma (x \mapsto_{s} v) [A|B] s = (\sigma [A|B] s)(x \mapsto_{s} v) \]
by (simp add: par-subst-def subst-upd-uvar-def lens-override-sublens-pres-sym)

lemma par-subst-upd-right-in [usubst]:
\[ [\text{vwb-lens } B; A \triangleright_{s} B; x \subseteq_{L} B] \implies \sigma [A|B] s = (\sigma [A|B] s)(x \mapsto_{s} v) \]
using lens-indep-sym par-subst-comm par-subst-upd-left-in by fastforce

lemma par-subst-upd-right-out [usubst]:
\[ [\text{vwb-lens } B; A \triangleright_{s} B] \implies \sigma [A|B] s = (\sigma [A|B] s)(x \mapsto_{s} v) \]
by (simp add: par-subst-comm par-subst-upd-left-out)

end
8 UTP Tactics

theory utp-tactics
  imports
    utp-expr utp-unrest utp-usedby
keywords update-uxpr-rep-eq-thms :: thy-decl
begin

declare image-comp [simp]

In this theory, we define several automatic proof tactics that use transfer techniques to re-interpret proof goals about UTP predicates and relations in terms of pure HOL conjectures. The fundamental tactics to achieve this are pred-simp and rel-simp; a more detailed explanation of their behaviour is given below. The tactics can be given optional arguments to fine-tune their behaviour. By default, they use a weaker but faster form of transfer using rewriting; the option robust, however, forces them to use the slower but more powerful transfer of Isabelle’s lifting package. A second option no-interp suppresses the re-interpretation of state spaces in order to eradicate record for tuple types prior to automatic proof.

In addition to pred-simp and rel-simp, we also provide the tactics pred-auto and rel-auto, as well as pred-blast and rel-blast; they, in essence, sequence the simplification tactics with the methods auto and blast, respectively.

8.1 Theorem Attributes

The following named attributes have to be introduced already here since our tactics must be able to see them. Note that we do not want to import the theories utp-pred and utp-rel here, so that both can potentially already make use of the tactics we define in this theory.

named-theorems upred-defs upred definitional theorems
named-theorems urel-defs urel definitional theorems

8.2 Generic Methods

We set up several automatic tactics that recast theorems on UTP predicates into equivalent HOL predicates, eliminating artefacts of the mechanisation as much as this is possible. Our approach is first to unfold all relevant definition of the UTP predicate model, then perform a transfer, and finally simplify by using lens and variable definitions, the split laws of alphabet records, and interpretation laws to convert record-based state spaces into products. The definition of the respective methods is facilitated by the Eisbach tool: we define generic methods that are parametrised by the tactics used for transfer, interpretation and subsequent automatic proof. Note that the tactics only apply to the head goal.

Generic Predicate Tactics

method gen-pred-tac methods transfer-tac interp-tac prove-tac =
  
"((unfold upred-defs) [1])?;
(transfer-tac),
(simp add: fun-eq-iff
  lens-defs upred-defs alpha-splits Product-Type.split-beta)?,
(interp-tac)?);
(prove-tac)

Generic Relational Tactics
method gen-rel-tac methods transfer-tac interp-tac prove-tac = (n
 unfold upred-defs urel-defs) [1];
 (transfer-tac),
 (simp add: fun-eq-iff relcomp-unfold OO-def
 lens-defs upred-defs alpha-splits Product-Type.split-beta) (?,
 (interp-tac) ?);
 (prove-tac)

8.3 Transfer Tactics
Next, we define the component tactics used for transfer.

8.3.1 Robust Transfer
Robust transfer uses the transfer method of the lifting package.

method slow-uexpr-transfer = (transfer)

8.3.2 Faster Transfer
Fast transfer side-steps the use of the (transfer) method in favour of plain rewriting with the underlying rep-eq-... laws of lifted definitions. For moderately complex terms, surprisingly, the transfer step turned out to be a bottle-neck in some proofs; we observed that faster transfer resulted in a speed-up of approximately 30% when building the UTP theory heaps. On the downside, tactics using faster transfer do not always work but merely in about 95% of the cases. The approach typically works well when proving predicate equalities and refinements conjectures.

A known limitation is that the faster tactic, unlike lifting transfer, does not turn free variables into meta-quantified ones. This can, in some cases, interfere with the interpretation step and cause subsequent application of automatic proof tactics to fail. A fix is in progress [TODO].

Attribute Setup We first configure a dynamic attribute uexpr-rep-eq-thms to automatically collect all rep-eq- laws of lifted definitions on the uexpr type.

ML-file uexpr-rep-eq.ML

setup (Global-Theory.add-thms-dynamic (@{binding uexpr-rep-eq-thms},
 uexpr-rep-eq.get-uexpr-rep-eq-thms o Context.theory-of))

We next configure a command update-uexpr-rep-eq-thms in order to update the content of the uexpr-rep-eq-thms attribute. Although the relevant theorems are collected automatically, for efficiency reasons, the user has to manually trigger the update process. The command must hence be executed whenever new lifted definitions for type uexpr are created. The updating mechanism uses find-theorems under the hood.

ML (Outer-Syntax.command @{command-keyword update-uexpr-rep-eq-thms}  
reread and update content of the uexpr-rep-eq-thms attribute  
(Scan.succeed (Toplevel.theory uexpr-rep-eq.thms););)
Lastly, we require several named-theorem attributes to record the manual transfer laws and extra simplifications, so that the user can dynamically extend them in child theories.

**named-theorems** `ueexpr-transfer-laws` `ueexpr-transfer-extra` extra simplifications for `ueexpr` transfer

**declare** `ueexpr-eq-iff` `ueexpr-transfer-laws`

**declare** `unrest-ueexpr.rep-eq` `ueexpr-transfer-extra`

**declare** `utp-expr.numeral-ueexpr.rep-eq` `ueexpr-transfer-extra`

**declare** `utp-expr.less-eq-ueexpr.rep-eq` `ueexpr-transfer-extra`

**declare** `Abs-ueexpr-inverse` `simplified`, `ueexpr-transfer-extra`

**declare** `Rep-ueexpr-inverse` `ueexpr-transfer-extra`

**Tactic Definition** We have all ingredients now to define the fast transfer tactic as a single simplification step.

**method** `fast-ueexpr-transfer` = (simp add: `ueexpr-transfer-laws` `ueexpr-rep-eq-thms` `ueexpr-transfer-extra`)

**8.4 Interpretation**

The interpretation of record state spaces as products is done using the laws provided by the utility theory `Interp`. Note that this step can be suppressed by using the `no-interp` option.

**method** `ueexpr-interp-tac` = (simp add: `lens-interp-laws`)

**8.5 User Tactics**

In this section, we finally set-up the six user tactics: `pred-simp`, `rel-simp`, `pred-auto`, `rel-auto`, `pred-blast` and `rel-blast`. For this, we first define the proof strategies that are to be applied after the transfer steps.

**method** `utp-simp-tac` = `(clarsimp)`

**method** `utp-auto-tac` = `clarsimp`; `auto`

**method** `utp-blast-tac` = `clarsimp`; `blast`

The ML file below provides ML constructor functions for tactics that process arguments suitable and invoke the generic methods `gen-pred-tac` and `gen-rel-tac` with suitable arguments.

**ML-file** `utp-tactics.ML`

Finally, we execute the relevant outer commands for method setup. Sadly, this cannot be done at the level of Eisbach since the latter does not provide a convenient mechanism to process symbolic flags as arguments. It may be worth to put in a feature request with the developers of the Eisbach tool.

**method-setup** `pred-simp` = (Scan.lift UTP-Tactics.scan-args) >> (`fn args => fn ctxt =>
let val prove-tac = Basic-Tactics.utp-simp-tac in
  (UTP-Tactics.inst-gen-pred-tac args prove-tac ctxt)
end)
method-setup rel-simp =
(Scan.lift UTP-Tactics.scan-args) >>
  (fn args => fn ctxt =>
    let val prove-tac = Basic-Tactics.utp-simp-tac in
      (UTP-Tactics.inst-gen-rel-tac args prove-tac ctxt)
    end)
)

method-setup pred-auto =
(Scan.lift UTP-Tactics.scan-args) >>
  (fn args => fn ctxt =>
    let val prove-tac = Basic-Tactics.utp-auto-tac in
      (UTP-Tactics.inst-gen-pred-tac args prove-tac ctxt)
    end)
)

method-setup rel-auto =
(Scan.lift UTP-Tactics.scan-args) >>
  (fn args => fn ctxt =>
    let val prove-tac = Basic-Tactics.utp-auto-tac in
      (UTP-Tactics.inst-gen-rel-tac args prove-tac ctxt)
    end)
)

method-setup pred-blast =
(Scan.lift UTP-Tactics.scan-args) >>
  (fn args => fn ctxt =>
    let val prove-tac = Basic-Tactics.utp-blast-tac in
      (UTP-Tactics.inst-gen-pred-tac args prove-tac ctxt)
    end)
)

method-setup rel-blast =
(Scan.lift UTP-Tactics.scan-args) >>
  (fn args => fn ctxt =>
    let val prove-tac = Basic-Tactics.utp-blast-tac in
      (UTP-Tactics.inst-gen-rel-tac args prove-tac ctxt)
    end)
)

Simpler, one-shot versions of the above tactics, but without the possibility of dynamic arguments.

method rel-simp’
  uses simp

method rel-auto’
  uses simp intro elim dest

method rel-blast’
  uses simp intro elim dest
\[(\text{rel-simp'} \ simp: \ simp, \ blast \ intro: \ intro \ elim: \ elim \ dest: \ dest)\]
9 Meta-level Substitution

theory utp-meta-subst
imports utp-subst utp-tactics
begin

Meta substitution substitutes a HOL variable in a UTP expression for another UTP expression. It is analogous to UTP substitution, but acts on functions.

lift-definition msubst : (′b ⇒ (′a, ′α) uexpr) ⇒ (′b, ′α) uexpr ⇒ (′a, ′α) uexpr
is λ F v b. F (v b) b.


syntax
-msubst :: logic ⇒ pttrn ⇒ logic ⇒ logic (([-→-]) [990,0,0] 991)
translations
-msubst P x v == CONST msubst (λ x. P) v

lemma msubst-lit [usubst]: ≪x≫[x→v] = v
  by (pred-auto)

lemma msubst-const [usubst]: P[x→v] = P
  by (pred-auto)

lemma msubst-pair [usubst]: (P x y)[(x,y)→(e,f)]u = (P x y)[x→e][y→f]
  by (rel-auto)

lemma msubst-lit-2-1 [usubst]: ≪x≫[(x,y)→(u,v)]u = u
  by (pred-auto)

lemma msubst-lit-2-2 [usubst]: ≪y≫[(x,y)→(u,v)]u = v
  by (pred-auto)

lemma msubst-lit′ [usubst]: ≪y≫[x→v] = ≪y≫
  by (pred-auto)

lemma msubst-lit′-2 [usubst]: ≪z≫[(x,y)→v] = ≪z≫
  by (pred-auto)

lemma msubst-aop [usubst]: (uop f (v x))[x→u] = uop f ((v x)[x→u])
  by (rel-auto)

lemma msubst-aop-2 [usubst]: (uop f (v x y))[x,y→u] = uop f ((v x y)[x,y→u])
  by (pred-simp, pred-simp)

lemma msubst-bop [usubst]: (bop f (v x) (w x))[x→u] = bop f ((v x)[x→u]) ((w x)[x→u])
  by (rel-auto)

lemma msubst-bop-2 [usubst]: (bop f (v x y) (w x y))[x,y→u] = bop f ((v x y)[x,y→u]) ((w x y)[x,y→u])
  by (pred-simp, pred-simp)
lemma msubst-var [usubst]:
  \( (\text{utp-expr}. \var x)[y\mapsto u] = \text{utp-expr}. \var x \) by (pred-simp)

lemma msubst-var-2 [usubst]:
  \( (\text{utp-expr}. \var x)[(y,z)\mapsto u] = \text{utp-expr}. \var x \) by (pred-simp+)

lemma msubst-unrest [unrest]:
  \[ \bigwedge v. \var x \# P(v); \var x \# k \] \( \implies \) \( \var x \# P(v)[v\mapsto k] \) by (pred-auto)

end

10 Alphabetised Predicates

theory utp-pred
imports
  utp-expr-funcs
  utp-subst
  utp-meta-subst
  utp-tactics
begin

In this theory we begin to create an Isabelle version of the alphabetised predicate calculus that is described in Chapter 1 of the UTP book [22].

10.1 Predicate type and syntax

An alphabetised predicate is a simply a boolean valued expression.

type-synonym \('a\) upred = (bool, 'a) uexpr

translations
  (type) 'a upred <= (type) (bool, 'a) uexpr

We want to remain as close as possible to the mathematical UTP syntax, but also want to be conservative with HOL. For this reason we chose not to steal syntax from HOL, but where possible use polymorphism to allow selection of the appropriate operator (UTP vs. HOL). Thus we will first remove the standard syntax for conjunction, disjunction, and negation, and replace these with adhoc overloaded definitions. We similarly use polymorphic constants for the other predicate calculus operators.

purge-notation
  conj (infixr \& 35) and
  disj (infixr \lor 30) and
  Not (\neg - [40] 40)

consts
  utrue :: 'a (true)
  ufalse :: 'a (false)
  uconj :: 'a \Rightarrow 'a \Rightarrow 'a (infixr \& 35)
  udisj :: 'a \Rightarrow 'a \Rightarrow 'a (infixr \lor 30)
  uimpl :: 'a \Rightarrow 'a \Rightarrow 'a (infixr \Rightarrow 25)
  uiff :: 'a \Rightarrow 'a \Rightarrow 'a (infixr \Leftrightarrow 25)
  unot :: 'a \Rightarrow 'a (\neg - [40] 40)
translations

**syntax**

- **idt-el** :: idt ⇒ idt-list (-)
- **idt-list** :: idt ⇒ idt-list ⇒ idt-list ((-,-) \[0,1]\)
- **ushEx** :: salpha ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushAll** :: salpha ⇒ logic ⇒ logic (∀ \[0,10\])
- **ushBAll** :: pttrn ⇒ logic ⇒ logic (∀ \[0,10\])
- **ushGAll** :: pttrn ⇒ logic ⇒ logic (∀ \[0,10\])
- **ushBEx** :: pttrn ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushGEx** :: pttrn ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushEx** :: pttrn ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushAll** :: pttrn ⇒ logic ⇒ logic (∀ \[0,10\])
- **ushBAll** :: pttrn ⇒ logic ⇒ logic (∀ \[0,10\])
- **ushGAll** :: pttrn ⇒ logic ⇒ logic (∀ \[0,10\])
- **ushBEx** :: pttrn ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushGEx** :: pttrn ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushEx** :: pttrn ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushAll** :: pttrn ⇒ logic ⇒ logic (∀ \[0,10\])
- **ushBAll** :: pttrn ⇒ logic ⇒ logic (∀ \[0,10\])
- **ushGAll** :: pttrn ⇒ logic ⇒ logic (∀ \[0,10\])
- **ushBEx** :: pttrn ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushGEx** :: pttrn ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushEx** :: pttrn ⇒ logic ⇒ logic (∃ \[0,10\])
- **ushAll** :: pttrn ⇒ logic ⇒ logic (∀ \[0,10\])

translations

- **ushEx** x P == CONST uex x P
- **ushAll** x P == CONST ushAll x P
- **ushBAll** x P == CONST ushBAll x P
- **ushGAll** x P == CONST ushGAll x P
- **ushBEx** x P == CONST ushBEx x P
- **ushGEx** x P == CONST ushGEx x P
- **ushEx** x P == CONST ushEx x P
- **ushAll** x P == CONST ushAll x P

10.2 Predicate operators

We chose to maximally reuse definitions and laws built into HOL. For this reason, when introducing the core operators we proceed by lifting operators from the polymorphic algebraic hierarchy of HOL. Thus the initial definitions take place in the context of type class instantiations. We first introduce our own class called **refine** that will add the refinement operator syntax to the HOL partial order class.

class refine = order

abbreviation refineBy :: 'a::refine ⇒ 'a ⇒ bool (infix ⊑ 50) where
\[ P \sqsubseteq Q \equiv \text{less-eq } Q \ P \]

Since, on the whole, lattices in UTP are the opposite way up to the standard definitions in HOL, we syntactically invert the lattice operators. This is the one exception where we do steal HOL syntax, but I think it makes sense for UTP. Indeed we make this inversion for all of the lattice operators.

```
purge-notation Lattices.inf (infixl \ / 70)
notation Lattices.inf (infixl \ / 70)
purge-notation Lattices.sup (infixl \ / 65)
notation Lattices.sup (infixl \ / 65)
```

```
purge-notation Inf (\ \ / \ [900] 900)
notation Inf (\ \ / \ [900] 900)
purge-notation Sup (\ \ / \ [900] 900)
notation Sup (\ \ / \ [900] 900)
```

```
purge-notation Orderings.bot (\)
notation Orderings.bot (\)
purge-notation Orderings.top (\)
notation Orderings.top (\)
```

```
purge-syntax
-\ INF1 :: ptttrns \ b \ / \ b \ ((\ \ / \ / \ [0, 10] 10))
-\ INF :: ptttrns \ a \ set \ b \ / \ b \ ((\ \ / \ / \ [0, 0, 10] 10))
-\ SUP1 :: ptttrns \ b \ / \ b \ ((\ \ / \ / \ [0, 10] 10))
-\ SUP :: ptttrns \ a \ set \ b \ / \ b \ ((\ \ / \ / \ [0, 0, 10] 10))
```

```
syntax
-\ INF1 :: ptttrns \ b \ / \ b \ ((\ \ / \ / \ [0, 10] 10))
-\ INF :: ptttrns \ a \ set \ b \ / \ b \ ((\ \ / \ / \ [0, 0, 10] 10))
-\ SUP1 :: ptttrns \ b \ / \ b \ ((\ \ / \ / \ [0, 10] 10))
-\ SUP :: ptttrns \ a \ set \ b \ / \ b \ ((\ \ / \ / \ [0, 0, 10] 10))
```

We trivially instantiate our refinement class

```
instance uexpr :: (order, type) refine ..
```

— Configure transfer law for refinement for the fast relational tactics.

```
theorem upred-ref-iff [uexpr-transfer-laws]:
(P \sqsubseteq Q) = (\forall b. [Q]c b \longrightarrow [P]c b)
apply (transfer)
apply (clarsimp)
done
```

Next we introduce the lattice operators, which is again done by lifting.

```
instantiation uexpr :: (lattice, type) lattice
begin
lift-definition sup-uexpr :: ('a, 'b) uexpr \ (\ b) uexpr \ (\ 'a, 'b) uexpr
is \AP Q A. Lattices.sup (P A) (Q A).
lift-definition inf-uexpr :: ('a, 'b) uexpr \ (\ b) uexpr \ (\ 'a, 'b) uexpr
is \AP Q A. Lattices.inf (P A) (Q A).
instance
by (intro-classes) (transfer, auto+)
end
```
instantiation uexpr :: (bounded-lattice, type) bounded-lattice
begin
  lift-definition bot-uexpr :: (a, 'b) uexpr is λ A. Orderings.bot.
  lift-definition top-uexpr :: (a, 'b) uexpr is λ A. Orderings.top.
instance
  by (intro-classes) (transfer, auto)+
end

lemma top-uexpr-rep-eq [simp]:
  [Orderings.bot] a b = False
  by (transfer, auto)

lemma bot-uexpr-rep-eq [simp]:
  [Orderings.top] a b = True
  by (transfer, auto)

instance uexpr :: (distrib-lattice, type) distrib-lattice
  by (intro-classes) (transfer, rule ext, auto simp add: sup-inf-distrib1)

Finally we show that predicates form a Boolean algebra (under the lattice operators), a complete
lattice, a completely distribute lattice, and a complete boolean algebra. This equip us with a
very complete theory for basic logical propositions.

instance uexpr :: (boolean-algebra, type) boolean-algebra
  apply (intro-classes, unfold uexpr-defs; transfer, rule ext)
  apply (simp-all add: sup-inf-distrib1 diff-eq)
done

instance uexpr :: (complete-lattice, type) complete-lattice
begin
  lift-definition Inf-uexpr :: (a, 'b) uexpr set ⇒ (a, 'b) uexpr
    is λ PS A. INF P:PS. P(A).
  lift-definition Sup-uexpr :: (a, 'b) uexpr set ⇒ (a, 'b) uexpr
    is λ PS A. SUP P:PS. P(A).
instance
  by (intro-classes)
    (transfer, auto intro: INF-lower SUP-upper simp add: INF-greatest SUP-least)+
end

instance uexpr :: (complete-distrib-lattice, type) complete-distrib-lattice
  by (intro-classes; transfer; auto simp add: INF-SUP-set)

instance uexpr :: (complete-boolean-algebra, type) complete-boolean-algebra ..

From the complete lattice, we can also define and give syntax for the fixed-point operators. Like
the lattice operators, these are reversed in UTP.

syntax
  -mu :: pttrn ⇒ logic ⇒ logic (µ · · · [0, 10] 10)
  -nu :: pttrn ⇒ logic ⇒ logic (ν · · · [0, 10] 10)

notation gfp (µ)
notation lfp (ν)

translations
  ν X · P == CONST lfp (λ X. P)
  µ X · P == CONST gfp (λ X. P)
With the lattice operators defined, we can proceed to give definitions for the standard predicate operators in terms of them.

**definition** true-upred = (Orderings.top :: 'α upred)
**definition** false-upred = (Orderings.bot :: 'α upred)
**definition** conj-upred = (Lattices.inf :: 'α upred => 'α upred => 'α upred)
**definition** disj-upred = (Lattices.sup :: 'α upred => 'α upred => 'α upred)
**definition** not-upred = (uminus :: 'α upred => 'α upred)
**definition** diff-upred = (minus :: 'α upred => 'α upred => 'α upred)

**abbreviation** Conj-upred :: 'α upred set => 'α upred (∧- [900] 900) where
∧ A ≡ ⋃ A

**abbreviation** Disj-upred :: 'α upred set => 'α upred (∨- [900] 900) where
∨ A ≡ ⋂ A

**notation**
- **conj-upred** (infixr ∧p 35) and
- **disj-upred** (infixr ∨p 30)

Perhaps slightly confusingly, the UTP infimum is the HOL supremum and vice-versa. This is because, again, in UTP the lattice is inverted due to the definition of refinement and a desire to have miracle at the top, and abort at the bottom.

**lift-definition** UINF :: ('a => 'α upred) => ('a => ('b::complete-lattice, 'α) uexpr) => ('b, 'α) uexpr
is λ P F b. Sup {[[F x]]_b | x. [P x]_b}.

**lift-definition** USUP :: ('a => 'α upred) => ('a => ('b::complete-lattice, 'α) uexpr) => ('b, 'α) uexpr
is λ P F b. Inf {[[F x]]_b | x. [P x]_b}.

**syntax**
- USup :: pttrn ⇒ logic ⇒ logic (∧ ∙ ∙ [0, 10] 10)
- USup :: pttrn ⇒ logic ⇒ logic (∧ ∙ ∙ [0, 10] 10)
- USup-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic (∧ ∙ ∙ [0, 10] 10)
- USup-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic (∧ ∙ ∙ [0, 10] 10)
- USUP :: pttrn ⇒ logic ⇒ logic ⇒ logic (∧ ∙ ∙ [0, 10] 10)
- USUP :: pttrn ⇒ logic ⇒ logic ⇒ logic (∧ ∙ ∙ [0, 10] 10)
- USup :: pttrn ⇒ logic ⇒ logic ⇒ logic (∈ ∙ ∙ [0, 10] 10)
- USup-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic (∩ ∙ ∙ [0, 10] 10)
- USUP-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic (∩ ∙ ∙ [0, 10] 10)
- Uinf :: pttrn ⇒ logic ⇒ logic (∨ ∙ ∙ [0, 10] 10)
- Uinf :: pttrn ⇒ logic ⇒ logic (∨ ∙ ∙ [0, 10] 10)
- Uinf-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic (∨ ∙ ∙ [0, 10] 10)
- Uinf-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic (∨ ∙ ∙ [0, 10] 10)
- UINF :: pttrn ⇒ logic ⇒ logic ⇒ logic (∨ ∙ ∙ [0, 10] 10)
- UINF :: pttrn ⇒ logic ⇒ logic ⇒ logic (∨ ∙ ∙ [0, 10] 10)

**translations**
- x | P · F => CONST UINF (λ x. P) (λ x. F)
- x · F =⇒ ⊢ x | true · F
- x · F =⇒ ⊢ x | true · F
- x ∈ A · F => ⊢ x | «x» ∈ u «A» · F
- x ∈ A · F <= ⊢ x | «y» ∈ u «A» · F
- x | P · F <= CONST UINF (λ y. P) (λ x. F)
- x | P · F(x) <= CONST UINF (λ x. P) F
- x | P · F => CONST USUP (λ x. P) (λ x. F)
- x · F =⇒ ⊢ x | true · F
- x ∈ A · F => ⊢ x | «x» ∈ u «A» · F
- x ∈ A · F <= ⊢ x | «y» ∈ u «A» · F
- x | P · F <= CONST USUP (λ y. P) (λ x. F)

52
\[ \bigcup x \mid P \cdot F(x) \leq \text{CONST USUP} \ (\lambda \ x. \ P) \ F \]

We also define the other predicate operators

**lift-definition** impl :: 'a upred \Rightarrow 'a upred \Rightarrow 'a upred is
\[
\lambda P \ Q \ A. \ P \ A \rightarrow Q \ A.
\]

**lift-definition** iff-upred :: 'a upred \Rightarrow 'a upred \Rightarrow 'a upred is
\[
\lambda P \ Q \ A. \ P \ A \leftrightarrow Q \ A.
\]

**lift-definition** ex :: ('a \Rightarrow 'a) \Rightarrow 'a upred \Rightarrow 'a upred is
\[
\lambda x P \ b. \ (\exists \ v. \ P(\text{put}_x b \ v)).
\]

**lift-definition** shEx :: ['β \Rightarrow 'α upred] \Rightarrow 'α upred is
\[
\lambda P \ A. \ \exists \ x. \ (P \ x) \ A.
\]

**lift-definition** all :: ('a \Rightarrow 'a) \Rightarrow 'a upred \Rightarrow 'a upred is
\[
\lambda x P \ b. \ (\forall \ v. \ P(\text{put}_x b \ v)).
\]

**lift-definition** shAll :: ['β \Rightarrow 'α upred] \Rightarrow 'α upred is
\[
\lambda P \ A. \ \forall \ x. \ (P \ x) \ A.
\]

We define the following operator which is dual of existential quantification. It hides the valuation of variables other than \( x \) through existential quantification.

**lift-definition** var-res :: 'a upred \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a upred is
\[
\lambda P \ x \ b. \ \exists \ b'. \ P(b' \oplus \ L b \ on \ x).
\]

**translations** - uvar-res P a \Rightarrow \text{CONST var-res P a}

We have to add a \( u \) subscript to the closure operator as I don’t want to override the syntax for HOL lists (we’ll be using them later).

**lift-definition** closure :: 'a upred \Rightarrow 'a upred \ (\_[]_u) is
\[
\lambda P \ A. \ \forall A'. \ P \ A'.
\]

**lift-definition** taut :: 'a upred \Rightarrow \text{bool} ('\_') is
\[
\lambda P. \ \forall A. \ P \ A.
\]

Configuration for UTP tactics

**update-uexpr-rep-eq-thms** — Reread rep-eq theorems.

**declare** utp-pred.taut.rep-eq [upred-defs]

**adhoc-overloading**
\[
\text{uttrue \ true-upred and}
\text{utfalse \ false-upred and}
\text{unot \ not-upred and}
\text{uconj \ conj-upred and}
\text{udisj \ disj-upred and}
\text{uimpl \ impl and}
\text{uiff \ iff-upred and}
\text{ux \ ex and}
\text{uall \ all and}
\text{ushEx \ shEx and}
\text{ushAll \ shAll}
\]

53
syntax
- uneq  :: logic ⇒ logic ⇒ logic (infixl \(\neq\) 50)
- umem  :: ('a, 'a) uexpr ⇒ ('a set, 'a) uexpr ⇒ (bool, 'a) uexpr (infix \(\ni\) 50)

translations
\[ x \neq_u y \Rightarrow \text{CONST unot} (x =_u y) \]
\[ x \ni_u A \Rightarrow \text{CONST unot} (\text{CONST bop} (\in) x A) \]

declare true-upred-def [upred-defs]
declare false-upred-def [upred-defs]
declare conj-upred-def [upred-defs]
declare disj-upred-def [upred-defs]
declare not-upred-def [upred-defs]
declare diff-upred-def [upred-defs]
declare subst-apd-uvar-def [upred-defs]
declare cond-subst-def [upred-defs]
declare par-subst-def [upred-defs]
declare subst-del-def [upred-defs]
declare unrest-asubst-def [upred-defs]
declare uexpr-defs [upred-defs]

lemma true-alt-def: true = «Truc»
by (pred-auto)

lemma false-alt-def: false = «False»
by (pred-auto)

declare true-alt-def [THEN sym, simp]
declare false-alt-def [THEN sym, simp]

10.3 Unrestriction Laws

lemma unrest-allE:
\[ [\Sigma i. x \#_u P(i); (\bigwedge i. x \# P(i)) \Rightarrow x \# P \land Q ] \Rightarrow Q \]
by (pred-auto)

lemma unrest-true [unrest]: x \# true
by (pred-auto)

lemma unrest-false [unrest]: x \# false
by (pred-auto)

lemma unrest-conj [unrest]: [ x \# (P :: 'a upred); x \# Q ] \Rightarrow x \# P \land Q
by (pred-auto)

lemma unrest-disj [unrest]: [ x \# (P :: 'a upred); x \# Q ] \Rightarrow x \# P \lor Q
by (pred-auto)

lemma unrest-UINF [unrest]:
\[ \left( \bigwedge i. x \# P(i) \right) \Rightarrow x \# (\prod i. P(i) \cdot Q(i)) \]
by (pred-auto)

lemma unrest-USUP [unrest]:
\[ \left( \bigwedge i. x \# P(i) \right) \Rightarrow x \# (\bigvee i. P(i) \cdot Q(i)) \]
by (pred-auto)
lemma unrest-UINF-mem [unrest]:
\[
([\forall i. \ i \in A \implies x \not\in i \cdot P(i))] \implies x \not\in (\prod i \in A) \cdot P(i)
\]
by (pred-simp, metis)

lemma unrest-USUP-mem [unrest]:
\[
([\forall i. \ i \in A \implies x \not\in i \cdot P(i))] \implies x \not\in (\bigsqcup i \in A) \cdot P(i)
\]
by (pred-simp, metis)

lemma unrest-impl [unrest]:
\[
x \not\in P; \ x \not\in Q \implies x \not\in P \implies Q
\]
by (pred-auto)

lemma unrest-iff [unrest]:
\[
x \not\in P; \ x \not\in Q \implies x \not\in P \iff Q
\]
by (pred-auto)

lemma unrest-not [unrest]:
\[
x \not\in (P :: \alpha \upred) \implies x \not\in (\neg P)
\]
by (pred-auto)

The sublens proviso can be thought of as membership below.

lemma unrest-ex-in [unrest]:
\[
mwb-lens y; \ x \subseteq L y \implies x \not\in (\exists y \cdot P)
\]
by (pred-auto)

declare sublens-refl [simp]
declare lens-plus-ub [simp]
declare lens-plus-right-sublens [simp]
declare comp-wb-lens [simp]
declare comp-mwb-lens [simp]
declare plus-mwb-lens [simp]

lemma unrest-ex-diff [unrest]:
\[
assumes x \not\in y y; \ y \not\in P \implies y \not\in (\exists x \cdot P)
\]
using assms lens-indep-comm
by (rel-simp', fastforce)

lemma unrest-all-in [unrest]:
\[
mwb-lens y; \ x \subseteq L y \implies x \not\in (\forall y \cdot P)
\]
by (pred-auto)

lemma unrest-all-diff [unrest]:
\[
assumes x \not\in y y; \ y \not\in P \implies y \not\in (\forall x \cdot P)
\]
using assms
by (pred-simp, simp-all add: lens-indep-comm)

lemma unrest-var-res-diff [unrest]:
\[
assumes x \not\in y \implies y \not\in (P \downarrow x)
\]
using assms by (pred-auto)

lemma unrest-var-res-in [unrest]:
\[
assumes mwb-lens x y; \ x \not\in y \not\in x \not\in P
\]
using assms

55
apply (pred-auto)
apply fastforce
apply (metis (no-types, lifting) mwblens-weak weak-lens.put-get)
done

lemma unrest-shEx [unrest]:
  assumes \( \forall y. x \not\in P(y) \)
  shows \( x \not\in (\exists y \cdot P(y)) \)
  using assms by (pred-auto)

lemma unrest-shAll [unrest]:
  assumes \( \forall y. x \not\in P(y) \)
  shows \( x \not\in (\forall y \cdot P(y)) \)
  using assms by (pred-auto)

lemma unrest-closure [unrest]:
  \( x \not\in [P]w \)
  by (pred-auto)

10.4 Used-by laws

lemma usedBy-not [unrest]:
  \([ x \not\in P ] \implies x \not\in (\neg P) \)
  by (pred-simp)

lemma usedBy-conj [unrest]:
  \([ x \not\in P; x \not\in Q ] \implies x \not\in (P \land Q) \)
  by (pred-simp)

lemma usedBy-disj [unrest]:
  \([ x \not\in P; x \not\in Q ] \implies x \not\in (P \lor Q) \)
  by (pred-simp)

lemma usedBy-impl [unrest]:
  \([ x \not\in P; x \not\in Q ] \implies x \not\in (P \Rightarrow Q) \)
  by (pred-simp)

lemma usedBy-iff [unrest]:
  \([ x \not\in P; x \not\in Q ] \implies x \not\in (P \Leftrightarrow Q) \)
  by (pred-simp)

10.5 Substitution Laws

Substitution is monotone

lemma subst-mono: \( P \sqsubseteq Q \implies (\sigma \uparrow P) \sqsubseteq (\sigma \uparrow Q) \)
  by (pred-auto)

lemma subst-true [usubst]: \( \sigma \uparrow \text{true} = \text{true} \)
  by (pred-auto)

lemma subst-false [usubst]: \( \sigma \uparrow \text{false} = \text{false} \)
  by (pred-auto)

lemma subst-not [usubst]: \( \sigma \uparrow (\neg P) = (\neg \sigma \uparrow P) \)
  by (pred-auto)
lemma subst-impl [usubst]: $\sigma \uparrow (P \Rightarrow Q) = (\sigma \uparrow P \Rightarrow \sigma \uparrow Q)$
by (pred-auto)

lemma subst-iff [usubst]: $\sigma \uparrow (P \Leftrightarrow Q) = (\sigma \uparrow P \Leftrightarrow \sigma \uparrow Q)$
by (pred-auto)

lemma subst-disj [usubst]: $\sigma \uparrow (P \lor Q) = (\sigma \uparrow P \lor \sigma \uparrow Q)$
by (pred-auto)

lemma subst-conj [usubst]: $\sigma \uparrow (P \land Q) = (\sigma \uparrow P \land \sigma \uparrow Q)$
by (pred-auto)

lemma subst-sup [usubst]: $\sigma \uparrow (P \sqcap Q) = (\sigma \uparrow P \sqcap \sigma \uparrow Q)$
by (pred-auto)

lemma subst-inf [usubst]: $\sigma \uparrow (P \sqcup Q) = (\sigma \uparrow P \sqcup \sigma \uparrow Q)$
by (pred-auto)

lemma subst-UINF [usubst]: $\sigma \uparrow (\downarrow i | P(i) \cdot Q(i)) = (\downarrow i | (\sigma \uparrow P(i)) \cdot (\sigma \uparrow Q(i)))$
by (pred-auto)

lemma subst-USUP [usubst]: $\sigma \uparrow (\bigvee i | P(i) \cdot Q(i)) = (\bigvee i | (\sigma \uparrow P(i)) \cdot (\sigma \uparrow Q(i)))$
by (pred-auto)

lemma subst-closure [usubst]: $\sigma \uparrow [P]_u = [P]_u$
by (pred-auto)

lemma subst-shEx [usubst]: $\sigma \uparrow (\exists x \cdot P(x)) = (\exists x \cdot \sigma \uparrow P(x))$
by (pred-auto)

lemma subst-shAll [usubst]: $\sigma \uparrow (\forall x \cdot P(x)) = (\forall x \cdot \sigma \uparrow P(x))$
by (pred-auto)

TODO: Generalise the quantifier substitution laws to n-ary substitutions

lemma subst-ex-same [usubst]:
    mwb-lens $x \Rightarrow \sigma (x \mapsto \cdot s v) \uparrow (\exists x \cdot P) = (\exists \sigma \uparrow P(x))$
by (pred-auto)

lemma subst-ex-same’ [usubst]:
    mwb-lens $x \Rightarrow \sigma (x \mapsto \cdot \& x \cdot P) = (\exists x \cdot \& x \cdot P)$
by (pred-auto)

lemma subst-ex-indep [usubst]:
    assumes $x \not\bowtie y \not\bowtie v$
    shows $(\exists y \cdot P)[v/x] = (\exists y \cdot P[v/x])$
using assms
apply (pred-auto)
using lens-indep-comm apply fastforce+
done

lemma subst-ex-unrest [usubst]:
    $x \not\bowtie \sigma \Rightarrow \sigma \uparrow (\exists x \cdot P) = (\exists x \cdot \sigma \uparrow P)$
by (pred-auto)
lemma subst-all-same [subst]:
\[ \text{msub-lens } x \Rightarrow \sigma(x \mapsto v) \upharpoonright (\forall \ x \cdot P) = \sigma \upharpoonright (\forall \ x \cdot P) \]
by (simp add: id-subst subst-unrest unrest-all-in)

lemma subst-all-indep [subst]:
assumes \( x \ni y \not\ni v \)
shows \( (\forall \ y \cdot P)[v/x] = (\forall \ y \cdot P[v/x]) \)
using assms
by (pred-simp, simp-all add: lens-indep-comm)

lemma msubst-true [subst]: \( \text{true}[x \mapsto v] = \text{true} \)
by (pred-auto)

lemma msubst-false [subst]: \( \text{false}[x \mapsto v] = \text{false} \)
by (pred-auto)

lemma msubst-not [subst]: \( (\neg P(x))[x \mapsto v] = (\neg ((P x)[x \mapsto v])) \)
by (pred-auto)

lemma msubst-not-2 [subst]: \( (\neg P x y)(x, y) \Rightarrow v = (\neg ((P x y)(x, y) \Rightarrow v)) \)
by (pred-auto)+

lemma msubst-disj [subst]: \( (P(x) \lor Q(x))[x \mapsto v] = ((P(x))[x \mapsto v] \lor (Q(x))[x \mapsto v]) \)
by (pred-auto)

lemma msubst-disj-2 [subst]: \( (P x y \lor Q x y)(x, y) \Rightarrow v = ((P x y)(x, y) \Rightarrow v) \lor (Q x y)(x, y) \Rightarrow v \)
by (pred-auto)+

lemma msubst-conj [subst]: \( (P(x) \land Q(x))[x \mapsto v] = ((P(x))[x \mapsto v] \land (Q(x))[x \mapsto v]) \)
by (pred-auto)

lemma msubst-conj-2 [subst]: \( (P x y \land Q x y)(x, y) \Rightarrow v = ((P x y)(x, y) \Rightarrow v) \land (Q x y)(x, y) \Rightarrow v \)
by (pred-auto)+

lemma msubst-implies [subst]:
\( (P x \Rightarrow Q x)[x \mapsto v] = ((P x)[x \mapsto v] \Rightarrow (Q x)[x \mapsto v]) \)
by (pred-auto)

lemma msubst-implies-2 [subst]:
\( (P x y \Rightarrow Q x y)(x, y) \Rightarrow v = ((P x y)(x, y) \Rightarrow v) \Rightarrow (Q x y)(x, y) \Rightarrow v \)
by (pred-auto)+

lemma msubst-shAll [subst]:
\( (\forall x \cdot P x y)[y \mapsto v] = (\forall x \cdot (P x y)[y \mapsto v]) \)
by (pred-auto)

lemma msubst-shAll-2 [subst]:
\( (\forall x \cdot P x y z)(y, z) \Rightarrow v = (\forall x \cdot (P x y z)(y, z) \Rightarrow v) \)
by (pred-auto)+

10.6 Sandbox for conjectures

definition utp-sandbox :: 'a upred ⇒ bool \((\text{TRY}')(\cdot)) \text{ where}
\( \text{TRY}(P) = (P = \text{undefined}) \)

translations
\( P \Leftarrow \text{CONST utp-sandbox } P \)
11 Alphabet Manipulation

Alphabets are simply types that characterise the state-space of an expression. Thus the Isabelle type system ensures that predicates cannot refer to variables not in the alphabet as this would be a type error. Often one would like to add or remove additional variables, for example if we wish to have a predicate which ranges only a smaller state-space, and then lift it into a predicate over a larger one. This is useful, for example, when dealing with relations which refer only to undashed variables (conditions) since we can use the type system to ensure well-formedness.

In this theory we will set up operators for extending and contracting and alphabet. We first set up a theorem attribute for alphabet laws and a tactic.

```isar
named-theorems alpha
method alpha-tac = (simp add: alpha unrest)?
```

11.2 Alphabet Extrusion

Alter an alphabet by application of a lens that demonstrates how the smaller alphabet ($\beta$) injects into the larger alphabet ($\alpha$). This changes the type of the expression so it is parametrised over the large alphabet. We do this by using the lens `get` function to extract the smaller state binding, and then apply this to the expression.

We call this ”extrusion” rather than ”extension” because if the extension lens is bijective then it does not extend the alphabet. Nevertheless, it does have an effect because the type will be different which can be useful when converting predicates with equivalent alphabets.

```isar
lift-definition aext :: ('a \alpha , 'b \beta) uexpr \Rightarrow ('a \alpha , 'b \beta) lens \Rightarrow ('a 'b) uexpr (infixr \oplus)
is \lambda P x b. P (get x b).
```

Next we prove some of the key laws. Extending an alphabet twice is equivalent to extending by the composition of the two lenses.

```isar
lemma aext-twice: (P \oplus a) \oplus b = P \oplus (a :L b)
  by (pred-auto)
```

The bijective $\Sigma$ lens identifies the source and view types. Thus an alphabet extension using this has no effect.

```isar
lemma aext-id [simp]: P \oplus L = P
  by (pred-auto)
```

Literals do not depend on any variables, and thus applying an alphabet extension only alters the predicate’s type, and not its valuation.

```isar
lemma aext-lit [simp]: \langle v\rangle \oplus a = \langle v\rangle
```
Alphabet extension distributes through the function liftings.

**lemma aext-zero [simp]:** \(0 \oplus_p a = 0\)
  
  by (pred-auto)

**lemma aext-one [simp]:** \(1 \oplus_p a = 1\)
  
  by (pred-auto)

**lemma aext-numeral [simp]:** \(\text{numeral } n \oplus_p a = \text{numeral } n\)
  
  by (pred-auto)

**lemma aext-true [simp]:** \(\text{true} \oplus_p a = \text{true}\)
  
  by (pred-auto)

**lemma aext-false [simp]:** \(\text{false} \oplus_p a = \text{false}\)
  
  by (pred-auto)

**lemma aext-not [alpha]:** \((\neg P) \oplus_p x = (\neg (P \oplus_p x))\)
  
  by (pred-auto)

**lemma aext-and [alpha]:** \((P \land Q) \oplus_p x = (P \oplus_p x \land Q \oplus_p x)\)
  
  by (pred-auto)

**lemma aext-or [alpha]:** \((P \lor Q) \oplus_p x = (P \oplus_p x \lor Q \oplus_p x)\)
  
  by (pred-auto)

**lemma aext-imp [alpha]:** \((P \Rightarrow Q) \oplus_p x = (P \oplus_p x \Rightarrow Q \oplus_p x)\)
  
  by (pred-auto)

**lemma aext-iff [alpha]:** \((P \Leftrightarrow Q) \oplus_p x = (P \oplus_p x \Leftrightarrow Q \oplus_p x)\)
  
  by (pred-auto)

**lemma aext-shAll [alpha]:** \((\forall x \cdot P(x)) \oplus_p a = (\forall x \cdot P(x) \oplus_p a)\)
  
  by (pred-auto)

**lemma aext-UINF-ind [alpha]:** \((\bigcap x \cdot P(x)) \oplus_p a = (\bigcap x \cdot (P(x) \oplus_p a))\)
  
  by (pred-auto)

**lemma aext-UINF-mem [alpha]:** \((\bigcap x \in A \cdot P(x)) \oplus_p a = (\bigcap x \in A \cdot (P(x) \oplus_p a))\)
  
  by (pred-auto)

Alphabet extension distributes through the function liftings.

**lemma aext-uop [alpha]:** \(uop \ f \ u \oplus_p a = uop \ f \ (u \oplus_p a)\)
  
  by (pred-auto)

**lemma aext-bop [alpha]:** \(bop \ f \ u \ v \oplus_p a = bop \ f \ (u \oplus_p a) \ (v \oplus_p a)\)
  
  by (pred-auto)

**lemma aext-trop [alpha]:** \(trop \ f \ u \ v \ w \oplus_p a = trop \ f \ (u \oplus_p a) \ (v \oplus_p a) \ (w \oplus_p a)\)
  
  by (pred-auto)

**lemma aext-qtop [alpha]:** \(qtop \ f \ u \ v \ w \ x \oplus_p a = qtop \ f \ (u \oplus_p a) \ (v \oplus_p a) \ (w \oplus_p a) \ (x \oplus_p a)\)
  
  by (pred-auto)

**lemma aext-plus [alpha]:**
(x + y) ⊕_p a = (x ⊕_p a) + (y ⊕_p a)
by (pred-auto)

lemma aext-minus [alpha]:
(x - y) ⊕_p a = (x ⊕_p a) - (y ⊕_p a)
by (pred-auto)

lemma aext-uminus [simp]:
(- x) ⊕ a = - (x ⊕_p a)
by (pred-auto)

lemma aext-times [alpha]:
(x * y) ⊕ a = (x ⊕_p a) * (y ⊕_p a)
by (pred-auto)

lemma aext-divide [alpha]:
(x / y) ⊕ a = (x ⊕_p a) / (y ⊕_p a)
by (pred-auto)

Extending a variable expression over x is equivalent to composing x with the alphabet, thus effectively yielding a variable whose source is the large alphabet.

lemma aext-var [alpha]:
var x ⊕ a = var (x ; L)
by (pred-auto)

lemma aext-ulambda [alpha]:
((λ x · P(x)) ⊕ a) = (λ x · P(x) ⊕ a)
by (pred-auto)

Alphabet extension is monotonic and continuous.

lemma aext-mono: P ⊑ Q =⇒ P ⊕ a ⊑ Q ⊕ a
by (pred-auto)

lemma aext-cont [alpha]:
vwblens a =⇒ (∩ A) ⊕ a = (∩ P∈A. P ⊕ a)
by (pred-simp)

If a variable is unrestricted in a predicate, then the extended variable is unrestricted in the predicate with an alphabet extension.

lemma unrest-aext [unrest]:
[ mwb-lens a; x ≠ p ] =⇒ unrest (x ; L) (p ⊕ a)
by (transfer, simp add: lens-comp-def)

If a given variable (or alphabet) b is independent of the extension lens a, that is, it is outside the original state-space of p, then it follows that once p is extended by a then b cannot be restricted.

lemma unrest-aext-indep [unrest]:
a ≠ b =⇒ b ≠ (p ⊕ a)
by pred-auto

11.3 Expression Alphabet Restriction

Restrict an alphabet by application of a lens that demonstrates how the smaller alphabet (β) injects into the larger alphabet (α). Unlike extension, this operation can lose information if the expressions refers to variables in the larger alphabet.

lift-definition arestr :: ('a, 'α) uexpr ⇒ ('β, 'α) lens ⇒ ('a, 'β) uexpr (infixr ↾ 90)
is \( \lambda P \times b. \ P \ (\text{create}_x \ b) \).

**update-uxpr-rep-eq-thms**

**Lemma arestr-id** [simp]: \( P \mid_e 1_L = P \)
by (pred-auto)

**Lemma arestr-acxt** [simp]: \( \text{mwb-lens} \ a \Rightarrow (P \oplus_p a) \mid_e a = P \)
by (pred-auto)

If an expression’s alphabet can be divided into two disjoint sections and the expression does not depend on the second half then restricting the expression to the first half is loss-less.

**Lemma aext-arestr [alpha]:**
assumes \( \text{mwb-lens} \ a \ \text{bij-lens} \ (a +_L b) \ a \gg b \not\in P \)
shows \( (P \mid_e a) \oplus_p a = P \)
proof –
from assms(2) have \( 1_L \subseteq_L a +_L b \)
by (simp add: bij-lens-equiv-id lens-equiv-def)
with assms(1,3,4) show ?thesis
apply (auto simp add: id-lens-def lens-plus-def sublens-def lens-comp-def prod.case-eq-if)
apply (pred-simp)
apply (metis lens-indep-comm mwb-lens-weak weak-lens put-get)
done
qed

Alternative formulation of the above law using used-by instead of unrestriction.

**Lemma aext-arestr’ [alpha]:**
assumes \( a \not\in P \)
shows \( (P \mid_e a) \oplus_p a = P \)
by (rel-simp, metis assms lens-override-def usedBy-uxpr-rep-eq)

**Lemma arestr-lit** [simp]: \( \langle v \rangle \mid_e a = \langle v \rangle \)
by (pred-auto)

**Lemma arestr-zero** [simp]: \( 0 \mid_e a = 0 \)
by (pred-auto)

**Lemma arestr-one** [simp]: \( 1 \mid_e a = 1 \)
by (pred-auto)

**Lemma arestr-numeral** [simp]: \( \text{numeral} \ n \mid_e a = \text{numeral} \ n \)
by (pred-auto)

**Lemma arestr-var** [alpha]:
var \( x \mid_e a = \text{var} (x /_L a) \)
by (pred-auto)

**Lemma arestr-true** [simp]: \( \text{true} \mid_e a = \text{true} \)
by (pred-auto)

**Lemma arestr-false** [simp]: \( \text{false} \mid_e a = \text{false} \)
by (pred-auto)

**Lemma arestr-not** [alpha]: \( (\neg P) \mid_e a = (\neg (P \mid_e a)) \)
by (pred-auto)
lemma arestr-and [alpha]: \((P \land Q)\big|_e x = (P\big|_e x \land Q\big|_e x)\)
  by (pred-auto)

lemma arestr-or [alpha]: \((P \lor Q)\big|_e x = (P\big|_e x \lor Q\big|_e x)\)
  by (pred-auto)

lemma arestr-imp [alpha]: \((P \Rightarrow Q)\big|_e x = (P\big|_e x \Rightarrow Q\big|_e x)\)
  by (pred-auto)

11.4 Predicate Alphabet Restriction

In order to restrict the variables of a predicate, we also need to existentially quantify away the other variables. We can’t do this at the level of expressions, as quantifiers are not applicable here. Consequently, we need a specialised version of alphabet restriction for predicates. It both restricts the variables using quantification and then removes them from the alphabet type using expression restriction.

definition upred-ares :: 
  'α upred ⇒ ('β ⇒ 'α) ⇒ 'β upred

where [upred-defs]: upred-ares P a = (P \big|_e a) \big|_e a

syntax
  -upred-ares :: logic ⇒ salpha ⇒ logic (infixl \|p 90)

translations
  -upred-ares P a == CONST upred-ares P a

lemma upred-aext-ares [alpha]:
  vwb-lens a =⇒ P ⊕_p a \big|_p a = P
  by (pred-auto)

lemma upred-ares-aext [alpha]:
  a ♯ P =⇒ (P \big|_p a) ⊕_p a = P
  by (pred-auto)

lemma upred-arestr-lit [simp]: ≪v≫ \big|_p a = ≪v≫
  by (pred-auto)

lemma upred-arestr-true [simp]: true \big|_p a = true
  by (pred-auto)

lemma upred-arestr-false [simp]: false \big|_p a = false
  by (pred-auto)

lemma upred-arestr-or [alpha]: (P \lor Q)\big|_p x = (P\big|_p x \lor Q\big|_p x)
  by (pred-auto)

11.5 Alphabet Lens Laws

lemma alpha-in-var [alpha]: x :L fst_L = in-var x
  by (simp add: in-var-def)

lemma alpha-out-var [alpha]: x :L snd_L = out-var x
  by (simp add: out-var-def)

lemma in-var-prod-lens [alpha]:
wb-lens \( Y \Rightarrow \) in-var \( x :_L (X \times_L Y) = \) in-var \( (x :_L X) \)
by (simp add: in-var-def prod-as-plus lens-comp-assoc fst-lens-plus)

**lemma** out-var-prod-lens [alpha]:
wb-lens \( X \Rightarrow \) out-var \( x :_L (X \times_L Y) = \) out-var \( (x :_L Y) \)
apply (simp add: out-var-def prod-as-plus lens-comp-assoc)
apply (subst snd-lens-plus)
using comp-wb-lens fst-vwb-lens vwb-lens-wb apply blast
apply (simp add: alpha-in-var alpha-out-var)
apply (simp)
done

11.6 Substitution Alphabet Extension

This allows us to extend the alphabet of a substitution, in a similar way to expressions.

definition subst-ext :: 'α usubst ⇒ ('α ⇒ β) ⇒ 'β usubst (infix ⊕_s 65) where
[upred-defs]: \( σ \oplus_s x = (λ s. \text{put}_x s (σ (\text{get}_x s))) \)

**lemma** id-subst-ext [usubst]:
wb-lens \( x \Rightarrow \) id \( \oplus_s x = \) id
by pred-auto

**lemma** upd-subst-ext [alpha]:
vwb-lens \( x \Rightarrow \) \( σ (\lambda y. v) \oplus_p x = (σ \oplus_s x) (\lambda x. v \oplus_p x) \)
by pred-auto

**lemma** apply-subst-ext [alpha]:
vwb-lens \( x \Rightarrow \) \( (σ \uplus_e x) \oplus_p x = (σ \oplus_s x) \uplus_p (e \oplus_p x) \)
by (pred-auto)

**lemma** aext-upred-eq [alpha]:
\( (\lambda e =_u f) \oplus_p a = (\lambda e \oplus_p a) =_u (f \oplus_p a) \)
by (pred-auto)

**lemma** subst-aext-comp [usubst]:
vwb-lens \( a \Rightarrow \) \( (σ \oplus_s a) \circ (σ \circ g) = (σ \circ g) \oplus_s a \)
by pred-auto

11.7 Substitution Alphabet Restriction

This allows us to reduce the alphabet of a substitution, in a similar way to expressions.

definition subst-res :: 'α usubst ⇒ ('β ⇒ 'α) ⇒ 'β usubst (infix ↾_s 65) where
[upred-defs]: \( σ ↾_s x = (λ s. \text{create}_x s (σ (\text{get}_x s))) \)

**lemma** id-subst-res [usubst]:
mwb-lens \( x \Rightarrow \) id \( \downarrow_s x = \) id
by pred-auto

**lemma** upd-subst-res [alpha]:
mwb-lens \( a \Rightarrow \) \( σ (\lambda y. v) ↾_s x = (σ ↾_s x) (\lambda y. v) ↾_s x \)
by (pred-auto)

**lemma** subst-ext-res [usubst]:
mwb-lens \( a \Rightarrow \) \( (σ \oplus_s x) ↾_s x = σ \)

64
by (pred-auto)

lemma unrest-subst-alpha-ext [unrest]:
  \( x \bowtie y \Rightarrow x \bowtie (P \oplus_s y) \)
by (pred-simp robust, metis lens-indep-def)
end

12 Lifting Expressions

definitions

12.1 Lifting definitions

We define operators for converting an expression to and from a relational state space with the help of alphabet extrusion and restriction. In general throughout Isabelle/UTP we adopt the notation \( \lceil P \rceil \) with some subscript to denote lifting an expression into a larger alphabet, and \( \lfloor P \rfloor \) for dropping into a smaller alphabet.

The following two functions lift and drop an expression, respectively, whose alphabet is \('\alpha\)', into a product alphabet \('\alpha \times ' \beta\). This allows us to deal with expressions which refer only to undashed variables, and use the type-system to ensure this.

abbreviation lift-pre :: ('a, 'a) uexpr ⇒ ('a, 'α × 'β) uexpr (⌈-⌉)
where \( \lceil P \rceil \bowtie \equiv P \oplus_p \text{fst}_L \)

abbreviation drop-pre :: ('a, 'α × 'β) uexpr ⇒ ('a, 'α) uexpr (⌊-⌋)
where \( \lfloor P \rfloor \bowtie \equiv P \mid_{\text{fst}_L} \)

The following two functions lift and drop an expression, respectively, whose alphabet is \('\beta\)', into a product alphabet \('\alpha \times ' \beta\). This allows us to deal with expressions which refer only to dashed variables.

abbreviation lift-post :: ('a, 'β) uexpr ⇒ ('a, 'α × 'β) uexpr (⌈-⌉)
where \( \lceil P \rceil \bowtie \equiv P \oplus_p \text{snd}_L \)

abbreviation drop-post :: ('a, 'α × 'β) uexpr ⇒ ('a, 'β) uexpr (⌊-⌋)
where \( \lfloor P \rfloor \bowtie \equiv P \mid_{\text{snd}_L} \)

12.2 Lifting Laws

With the help of our alphabet laws, we can prove some intuitive laws about alphabet lifting. For example, lifting variables yields an unprimed or primed relational variable expression, respectively.

lemma lift-pre-var [simp]:
  \([\text{var } x]_{\bowtie} \equiv x\)
by (alpha-tac)

lemma lift-post-var [simp]:
  \([\text{var } x]_{\bowtie} \equiv x\)'
by (alpha-tac)
12.3 Substitution Laws

lemma pre-var-subst [usubst]:
\[ \sigma(x \mapsto s) \upharpoonright \lbrack P \rbrack_s = \sigma \upharpoonright \lbrack P \rbrack_{x/s} \]
by (pred-simp)

12.4 Unrestriction laws

Crucially, the lifting operators allow us to demonstrate unrestriction properties. For example, we can show that no primed variable is restricted in an expression over only the first element of the state-space product type.

lemma unrest-dash-var-pre [unrest]:
fixes x :: (′a ⇒ ′a)
shows $x \not\in \lbrack P \rbrack_s$
by (pred-auto)

end

13 Predicate Calculus Laws

theory utp-pred-laws
  imports utp-pred
begin

13.1 Propositional Logic

Showing that predicates form a Boolean Algebra (under the predicate operators as opposed to the lattice operators) gives us many useful laws.

interpretation boolean-algebra diff-upred not-upred conj-upred (≤) (<)
disj-upred false-upred true-upred
by (unfold-locales; pred-auto)

lemma taut-true [simp]: ′true′
by (pred-auto)

lemma taut-false [simp]: ′false′ = False
by (pred-auto)

lemma taut-conj: ′A ∧ B′ = (′A′ ∧ ′B′)
by (rel-auto)

lemma taut-conj-elim [elim!]:
\[ \lbrack A \land B \rbrack ; \lbrack A \rbrack ; \lbrack B \rbrack \imp P \imp P \]
by (rel-auto)

lemma taut-refine-impl: \[ Q \subseteq P ; \lbrack P \rbrack \imp Q \]
by (rel-auto)

lemma taut-shEx-elim:
\[ \lbrack (\exists x \cdot P x) \rbrack ; \bot \cdot x . \Sigma \not\in P x \land x . \lbrack P x \rbrack \imp Q \imp Q \]
by (rel-blast)

Linking refinement and HOL implication

lemma refine-prop-intro:
assumes $\Sigma \not\Downarrow P \not\Downarrow Q \quad 'Q' \implies 'P$

shows $P \subseteq Q$

using assms

by (pred-auto)

lemma taut-not: $\Sigma \not\Downarrow P \implies (\neg 'P') = \neg 'P$

by (rel-auto)

lemma taut-shAll-intro:

$\forall x. 'P x' \implies \forall x. 'P x'$

by (rel-auto)

lemma taut-shAll-intro-2:

$\forall x y. 'P x y' \implies \forall (x, y). 'P x y'$

by (rel-auto)

lemma taut-impl-intro:

$[[ \Sigma \not\Downarrow P; 'P' \implies 'Q' ]] \implies 'P \implies Q'$

by (rel-auto)

lemma upred-eval-taut:

$'P[\ll b\rr/\&v]' = [P]_b$

by (pred-auto)

lemma refBy-order: $P \sqsubseteq Q = 'Q \Rightarrow P'$

by (pred-auto)

lemma conj-idem [simp]: $((P::'a upred) \land P) = P$

by (pred-auto)

lemma disj-idem [simp]: $((P::'a upred) \lor P) = P$

by (pred-auto)

lemma conj-comm: $((P::'a upred) \land Q) = (Q \land P)$

by (pred-auto)

lemma disj-comm: $((P::'a upred) \lor Q) = (Q \lor P)$

by (pred-auto)

lemma conj-subst: $P = R \implies ((P::'a upred) \land Q) = (R \land Q)$

by (pred-auto)

lemma disj-subst: $P = R \implies ((P::'a upred) \lor Q) = (R \lor Q)$

by (pred-auto)

lemma conj-associ: $((P::'a upred) \land Q) \land S) = (P \land (Q \land S))$

by (pred-auto)

lemma disj-associ: $((P::'a upred) \lor Q) \lor S) = (P \lor (Q \lor S))$

by (pred-auto)

lemma conj-disj-abs: $((P::'a upred) \land (P \lor Q)) = P$

by (pred-auto)

lemma disj-conj-abs: $((P::'a upred) \lor (P \land Q)) = P$

67
by (pred-auto)

**lemma** `conj-disj-distr`: \(((P::'α upred) \land (Q \lor R)) = ((P \land Q) \lor (P \land R))
by (pred-auto)

**lemma** `disj-conj-distr`: \(((P::'α upred) \lor (Q \land R)) = ((P \lor Q) \land (P \lor R))
by (pred-auto)

**lemma** `true-disj-zero` [simp]:
\((P \lor true) = true (true \lor P) = true
by (pred-auto)

**lemma** `true-conj-zero` [simp]:
\((P \land false) = false (false \land P) = false
by (pred-auto)

**lemma** `false-sup` [simp]:
\(false \sqcap P = P P \sqcap false = P
by (pred-auto)

**lemma** `true-inf` [simp]:
\(true \sqcup P = P P \sqcup true = P
by (pred-auto)

**lemma** `imp-vacuous` [simp]:
\((false \Rightarrow u) = true
by (pred-auto)

**lemma** `imp-true` [simp]:
\((p \Rightarrow true) = true
by (pred-auto)

**lemma** `true-imp` [simp]:
\((true \Rightarrow p) = p
by (pred-auto)

**lemma** `impl-mp1` [simp]: \((P \land (P \Rightarrow Q)) = (P \land Q)
by (pred-auto)

**lemma** `impl-mp2` [simp]: \(((P \Rightarrow Q) \land P) = (Q \land P)
by (pred-auto)

**lemma** `impl-adjoin`: \(((P \Rightarrow Q) \land R) = ((P \land R \Rightarrow Q \land R) \land R)
by (pred-auto)

**lemma** `impl-refine-intro`:
\[ [Q \subseteq P_1; P_2 \subseteq (P_1 \land Q_2) ] \implies (P_1 \Rightarrow P_2) \subseteq (Q_1 \Rightarrow Q_2)
by (pred-auto)

**lemma** `spec-refine`:
\(Q \subseteq (P \land R) \implies (P \Rightarrow Q) \subseteq R
by (rel-auto)

**lemma** `impl-disjI`:
\[['P \Rightarrow R'\; ; 'Q \Rightarrow R' ] \implies '(P \lor Q) \Rightarrow R'
by (rel-auto)

**lemma** `conditional-iff`:
\((P \Rightarrow Q) = (P \Rightarrow R) \iff 'P \Rightarrow (Q \iff R)'
by (pred-auto)
lemma p-and-not-p [simp]: \((P \land \neg P) = \text{false}\)
 by (pred-auto)

lemma p-or-not-p [simp]: \((P \lor \neg P) = \text{true}\)
 by (pred-auto)

lemma p-imp-p [simp]: \((P \Rightarrow P) = \text{true}\)
 by (pred-auto)

lemma p-iff-p [simp]: \((P \iff P) = \text{true}\)
 by (pred-auto)

lemma p-imp-false [simp]: 
\((P \Rightarrow \text{false}) = (\neg P)\)
 by (pred-auto)

lemma not-conj-deMorgans [simp]: 
\((\neg ((P::\alpha \upred) \land Q)) = ((\neg P) \lor (\neg Q))\)
 by (pred-auto)

lemma not-disj-deMorgans [simp]: 
\((\neg ((P::\alpha \upred) \lor Q)) = ((\neg P) \land (\neg Q))\)
 by (pred-auto)

lemma conj-disj-not-abs [simp]: 
\(((P::\alpha \upred) \land ((\neg P) \lor Q)) = (P \land Q)\)
 by (pred-auto)

lemma subsumption1:
\('P \Rightarrow Q' \implies (P \lor Q) = Q\)
 by (pred-auto)

lemma subsumption2:
\('Q \Rightarrow P' \implies (P \lor Q) = P\)
 by (pred-auto)

lemma neg-conj-cancel1: 
\((\neg P \land (P \lor Q)) = (\neg P \land Q::\alpha \upred)\)
 by (pred-auto)

lemma neg-conj-cancel2: 
\((\neg Q \land (P \lor Q)) = (\neg Q \land P::\alpha \upred)\)
 by (pred-auto)

lemma double-negation [simp]: 
\((\neg \neg (P::\alpha \upred)) = P\)
 by (pred-auto)

lemma true-not-false [simp]: \(\text{true} \neq \text{false} \neq \text{true}\)
 by (pred-auto)

lemma closure-conj-distr: 
\([P]\land [Q] = [P \land Q]\)
 by (pred-auto)

lemma closure-imp-distr: 
\('[P \Rightarrow Q] = [P] \Rightarrow [Q]'\)
 by (pred-auto)

lemma true-iff [simp]: \((P \iff \text{true}) = P\)
 by (pred-auto)

lemma taut-iff-eq: 
\('P \iff Q' \iff (P = Q)\)
by (pred-auto)

**Lemma impl-alt-def**: \((P \Rightarrow Q) = (\neg P \lor Q)\)
by (pred-auto)

### 13.2 Lattice laws

**Lemma uinf-or**:  
fixes \(P \, Q \, ::= \, 'a \, upred\)  
shows \((P \sqcap Q) = (P \lor Q)\)
by (pred-auto)

**Lemma usup-and**:  
fixes \(P \, Q \, ::= \, 'a \, upred\)  
shows \((P \sqcup Q) = (P \land Q)\)
by (pred-auto)

**Lemma UINF-alt-def**:  
\((\prod i \, | \, A(i) \cdot P(i)) = (\prod i \cdot A(i) \land P(i))\)
by (rel-auto)

**Lemma USUP-true** [simp]: \((\bigsqcup P \mid F(P) \cdot true) = true\)
by (pred-auto)

**Lemma UINF-mem-UNIV** [simp]: \((\prod x \in UNIV \cdot P(x)) = (\prod x \cdot P(x))\)
by (pred-auto)

**Lemma USUP-mem-UNIV** [simp]: \((\bigsqcup x \in UNIV \cdot P(x)) = (\bigsqcup x \cdot P(x))\)
by (pred-auto)

**Lemma USUP-false** [simp]: \((\bigsqcup i \cdot false) = false\)
by (pred-simp)

**Lemma USUP-mem-false** [simp]: \(I \neq \{\} \Rightarrow (\bigsqcup i \in I \cdot false) = false\)
by (rel-simp)

**Lemma USUP-where-false** [simp]: \((\bigsqcup i \mid false \cdot P(i)) = true\)
by (rel-auto)

**Lemma UINF-true** [simp]: \((\prod i \cdot true) = true\)
by (pred-simp)

**Lemma UINF-ind-const** [simp]:  
\((\prod i \cdot P) = P\)
by (rel-auto)

**Lemma UINF-mem-true** [simp]: \(A \neq \{\} \Rightarrow (\prod i \in A \cdot true) = true\)
by (pred-auto)

**Lemma UINF-false** [simp]: \((\prod i \mid P(i) \cdot false) = false\)
by (pred-auto)

**Lemma UINF-where-false** [simp]: \((\bigsqcup i \mid false \cdot P(i)) = false\)
by (rel-auto)

**Lemma UINF-cong-eq**:

---

70
lemma USUP-image-eq [simp]: USUP (λi. ≪i≫ ∈u≪f ↑ A≫) g = (∪ i∈A · g(f(i)))
by (pred-simp, rule-tac cong[of Inf Inf], auto)

lemma UINF-image-eq [simp]: UINF (λi. ≪i≫ ∈u≪f ↑ A≫) g = (∩ i∈A · g(f(i)))
by (pred-simp, rule-tac cong[of Sup Sup], auto)

lemma subst-continuous [usubst]: σ † (d A) = (d {σ † P | P. P ∈ A})
by (simp add: UINF-as-Sup[THEN sym] usubst setcompr-eq-image)

lemma not-UINF: (¬ (∩ i∈A · P(i))) = (∪ i∈A · ¬ P(i))
by (pred-auto)

lemma not-USUP: (¬ (∪ i∈A · P(i))) = (∩ i∈A · ¬ P(i))
by (pred-auto)

lemma not-UINF-ind: (¬ (∩ i · P(i))) = (∪ i · ¬ P(i))
by (pred-auto)

lemma not-USUP-ind: (¬ (∪ i · P(i))) = (∩ i · ¬ P(i))
by (pred-auto)

lemma UINF-empty [simp]: (∩ i ∈ {} · P(i)) = false
by (pred-auto)

lemma UINF-insert [simp]: (∩ i∈insert x xs · P(i)) = (P(x) ∩ (∩ i∈xs · P(i)))
apply (pred-simp)
apply (subst Sup-insert[THEN sym])
apply (rule-tac cong[of Sup Sup])
apply (auto)
done

lemma UINF-atLeast-first:
P(n) ∩ (∩ i ∈ {Suc n.} · P(i)) = (∩ i ∈ {n.} · P(i))
proof –
have insert n {Suc n.} = {n.}
  by (auto)
thus ?thesis
  by (metis UINF-insert)
qed

lemma UINF-atLeast-Suc:
(∩ i ∈ {Suc m.} · P(i)) = (∩ i ∈ {m.} · P(Suc i))
by (rel-simp, metis (full-types) Suc-le-D not-less-eq-eq)

lemma USUP-empty [simp]: (∪ i ∈ {} · P(i)) = true
by (pred-auto)

lemma USUP-insert [simp]: (∪ i∈insert x xs · P(i)) = (P(x) ∪ (∪ i∈xs · P(i)))
apply (pred-simp)
apply (subst Inf-insert[THEN sym])
apply (rule-tac cong[of Inf Inf])
apply (auto)
done

lemma USUP-atLeast-first:
(P(n) ∧ (∧ i ∈ {Suc n..} · P(i))) = (∨ i ∈ {n..} · P(i))

proof –
have insert n {Suc n..} = {n..}
  by (auto)
thus ?thesis
  by (metis USUP-insert conj-upred-def)

qed

lemma USUP-atLeast-Suc:
(∅ i ∈ {Suc m..} · P(i)) = (∨ i ∈ {m..} · P(Suc i))
by (rel-simp, metis (full-types) Suc-leD not-less-eq-eq)

lemma conj-UINF-dist:
(P ∧ (∏ Q∈S · F(Q))) = (∏ Q∈S · P ∧ F(Q))
by (simp add: upred-defs bop.rep-eq lit.rep-eq, pred-auto)

lemma conj-UINF-ind-dist:
(P ∧ (Q · F(Q))) = (Q · P ∧ F(Q))
by pred-auto

lemma disj-UINF-dist:
S ≠ {} ⇒ (P ∨ (∏ Q∈S · F(Q))) = (∏ Q∈S · P ∨ F(Q))
by (simp add: upred-defs bop.rep-eq lit.rep-eq, pred-auto)

lemma UINF-conj-UINF [simp]:
(∏ i∈I · P(i)) ∨ (∏ i∈I · Q(i)) = (∏ i∈I · P(i) ∨ Q(i))
by (rel-auto)

lemma conj-USUP-dist:
S ≠ {} ⇒ (P ∧ (∨ Q∈S · F(Q))) = (∨ Q∈S · P ∧ F(Q))

lemma USUP-conj-USUP [simp]: (∨ P ∈ A · F(P)) ∧ (∨ P ∈ A · G(P)) = (∨ P ∈ A · F(P) ∧ G(P))
by (simp add: upred-defs bop.rep-eq lit.rep-eq, pred-auto)

lemma UINF-all-cong [cong]:
  assumes ∧ P. F(P) = G(P)
  shows (∏ P · F(P)) = (∏ P · G(P))
by (simp add: UINF-as-Sup-collect assms)

lemma UINF-cong:
  assumes ∧ P. P ∈ A ⇒ F(P) = G(P)
  shows (∏ P∈A · F(P)) = (∏ P∈A · G(P))
by (simp add: UINF-as-Sup-collect assms)

lemma USUP-all-cong:
  assumes ∧ P. F(P) = G(P)
  shows (∨ P · F(P)) = (∨ P · G(P))
by (simp add: assms)

lemma USUP-cong:
  assumes ∧ P. P ∈ A ⇒ F(P) = G(P)
  shows (∨ P∈A · F(P)) = (∨ P∈A · G(P))
by (simp add: USUP-as-Inf-collect assms)
lemma **UINF-subset-mono**: $A \subseteq B \implies (\bigsqcup P \in A \cdot F(P)) \subseteq (\bigsqcup P \in A \cdot F(P))$
  by (simp add: SUP-subset-mono UINF-as-Sup-collect)

lemma **USUP-subset-mono**: $A \subseteq B \implies (\bigsqcup P \in A \cdot F(P)) \subseteq (\bigsqcup P \in B \cdot F(P))$
  by (simp add: INF-superset-mono USUP-as-Inf-collect)

lemma **UINF-impl**: $(\bigsqcup P \in A \cdot F(P) \Rightarrow G(P)) = ((\bigsqcup P \in A \cdot F(P)) \Rightarrow (\bigsqcup P \in A \cdot G(P)))$
  by (pred-auto)

lemma **USUP-is-forall**: $(\bigsqcup x \cdot P(x)) = (\forall x \cdot P(x))$
  by (pred-simp)

lemma **USUP-ind-is-forall**: $(\bigsqcup x \in A \cdot P(x)) = (\forall x \in A \cdot P(x))$
  by (pred-auto)

lemma **UINF-is-exists**: $(\bigsqcup x \cdot P(x)) = (\exists x \cdot P(x))$
  by (pred-simp)

lemma **UINF-all-nats** [simp]:
  fixes $P :: \cdot \alpha \upred$
  shows $(\bigsqcup n \cdot \bigsqcup i \in \{0..n\} \cdot P(i)) = (\bigsqcup n \cdot P(n))$
  by (pred-auto)

lemma **USUP-all-nats** [simp]:
  fixes $P :: \cdot \alpha \upred$
  shows $(\bigsqcup n \cdot \bigsqcup i \in \{0..n\} \cdot P(i)) = (\bigsqcup n \cdot P(n))$
  by (pred-auto)

lemma **UINF-upto-expand-first**:
  $m < n \implies (\bigsqcup i \in \{0..<\Suc n\} \cdot P(i)) = ((P(m) :: \cdot \alpha \upred) \lor (\bigsqcup i \in \{\Suc m..<n\} \cdot P(i)))$
  apply (rel-auto) using Suc-leI le-less-or-eq by auto

lemma **UINF-upto-expand-last**:
  $(\bigsqcup i \in \{0..<\Suc(n)\} \cdot P(i)) = (\bigsqcup i \in \{0..<n\} \cdot P(i) \lor P(n))$
  apply (rel-auto)
  using less-SucE by blast

lemma **UINF-Suc-shift**: $(\bigsqcup i \in \{0..<\Suc n\} \cdot P(i)) = (\bigsqcup i \in \{0..<n\} \cdot P(Suc i))$
  apply (rel-simp)
  apply (rule cong[of Sup], auto)
  using less-Suc-eq-0-disj by auto

lemma **USUP-upto-expand-first**:
  $(\bigsqcup i \in \{0..<\Suc(n)\} \cdot P(i)) = (P(0) \land (\bigsqcup i \in \{1..<\Suc(n)\} \cdot P(i)))$
  apply (rel-auto)
  using not-less by auto

lemma **USUP-Suc-shift**: $(\bigsqcup i \in \{0..<\Suc n\} \cdot P(i)) = (\bigsqcup i \in \{0..<n\} \cdot P(Suc i))$
  apply (rel-simp)
  apply (rule cong[of Inf], auto)
  using less-Suc-eq-0-disj by auto

lemma **UINF-list-conv**:
  $(\bigsqcup i \in \{0..<\length(xs)\} \cdot f (xs ! i)) = foldr (\lor) (map f xs) false$

74
apply (induct xs)
apply (rel-auto)
apply (simp add: UINF-upto-expand-first UINF-Suc-shift)
done

lemma USUP-list-conv:
(\biguplus\ i \in \{0..<\text{length}(xs)\} \cdot f (xs ! i)) = foldr (\land) (map f xs) true
apply (induct xs)
apply (rel-auto)
apply (simp add: USUP-upto-expand-first USUP-Suc-shift)
done

lemma UINF-refines:
[ \bigwedge\ i. \ i \in I \Rightarrow P \sqsubseteq Q i \] \Rightarrow P \sqsubseteq (\bigcap\ i \cdot Q i)
by (simp add: UINF-as-Sup-collect, metis SUP-least)

lemma UINF-refines':
assumes \bigwedge\ i. P \sqsubseteq Q (i)
shows P \sqsubseteq (\bigcap\ i \cdot Q (i))
using assms
apply (rel-auto)
using Sup-le-iff by fastforce

lemma UINF-pred-ueq [simp]:
(\sharp x \xrightarrow{\|} u \sharp v \cdot P (x)) = (P v)
by (pred-auto)

lemma UINF-pred-lit-eq [simp]:
(\sharp x \xrightarrow{=} u \sharp v \cdot P (x)) = (P v)
by (pred-auto)

13.3 Equality laws

lemma eq-upred-refl [simp]: (x =\u x) = true
by (pred-auto)

lemma eq-upred-sym: (x =\u y) = (y =\u x)
by (pred-auto)

lemma eq-cong-left:
assumes wvb-lens x $x \xrightarrow{\|} Q $x' \xrightarrow{\|} Q $x \xrightarrow{\|} R $x' \xrightarrow{\|} R
shows ((\$x' =\u \$x \land Q) = (\$x' =\u \$x \land R)) \leftrightarrow (Q = R)
using assms
by (pred-simp, (meson wvb-lens-def wvb-lens-mwb weak-lens-def)+)

lemma conj-eq-in-var-subst:
fixes x :: ('a \Rightarrow 'a)
assumes wvb-lens x
shows (P \land \$x =\u v) = (P[x/$x] \land \$x =\u v)
using assms
by (pred-simp, (metis wvb-lens-wb wb-lens.get-put)+)

lemma conj-eq-out-var-subst:
fixes x :: ('a \Rightarrow 'a)
assumes wvb-lens x
shows (P \land \$x' =\u v) = (P[v/$x'] \land \$x' =\u v)
using assms

75
lemma conj-pos-var-subst:
assumes vwb-lens x
shows ($x \land Q) = (x \land Q[true/x])
using assms
by (pred-auto, metis (full-types) vwb-lens-wb wb-lens-get-put, metis (full-types) vwb-lens-wb wb-lens-get-put)

lemma conj-neg-var-subst:
assumes vwb-lens x
shows ($\neg x \land Q) = (\neg x \land Q[false/x])
using assms
by (pred-auto, metis (full-types) vwb-lens-wb wb-lens-get-put, metis (full-types) vwb-lens-wb wb-lens-get-put)

lemma upred-eq-true [simp]: (p = u true) = p
by (pred-auto)

lemma upred-eq-false [simp]: (p = u false) = (\neg p)
by (pred-auto)

lemma upred-true-eq [simp]: (true = u p) = p
by (pred-auto)

lemma upred-false-eq [simp]: (false = u p) = (\neg p)
by (pred-auto)

lemma conj-var-subst:
assumes vwb-lens x
shows (P \land var x = u v) = (P[v/x] \land var x = u v)
using assms
by (pred-simp, (metis (full-types) vwb-lens-def wb-lens-get-put)+)

13.4 HOL Variable Quantifiers

lemma shEx-unbound [simp]: (\exists x \cdot P) = P
by (pred-auto)

lemma shEx-bool [simp]: shEx P = (P True \lor P False)
by (pred-simp, metis (full-types))

lemma shEx-commute: (\exists x \cdot \exists y \cdot P x y) = (\exists y \cdot \exists x \cdot P x y)
by (pred-auto)

lemma shEx-cong: [ \land x. P x = Q x ] \implies shEx P = shEx Q
by (pred-auto)

lemma shEx-insert: (\exists x \in insert u y A \cdot P(x)) = (P(x)[x\rightarrow y] \lor (\exists x \in A \cdot P(x)))
by (pred-auto)

lemma shEx-one-point: (\exists x \cdot \ll x \rr = u v \land P(x)) = P(x)[x\rightarrow v]
by (rel-auto)

lemma shAll-unbound [simp]: (\forall x \cdot P) = P
by (pred-auto)

lemma shAll-bool [simp]: shAll P = (P True \land P False)
Lemma shAll-cong: \[ \forall x. P x = Q x \implies \text{shAll } P = \text{shAll } Q \]

by (pred-auto)

Quantifier lifting

named-theorems uquant-lift

Lemma shEx-lift-conj-1 [uquant-lift]:
\[ (\exists x \cdot P(x)) \land Q = (\exists x \cdot P(x) \land Q) \]

by (pred-auto)

Lemma shEx-lift-conj-2 [uquant-lift]:
\[ P \land (\exists x \cdot Q(x)) = (\exists x \cdot P \land Q(x)) \]

by (pred-auto)

13.5 Case Splitting

Lemma eq-split-subst:
assumes vwb-lens x
shows \( P = Q \iff (\forall v. P[v/x] = Q[v/x]) \)
using assms
by (pred-auto, metis vwb-lens-wb wb-lens.source-stability)

Lemma eq-split-substI:
assumes vwb-lens x \( \land \forall v. P[v/x] = Q[v/x] \)
shows \( P = Q \)
using assms(1) assms(2) eq-split-subst by blast

Lemma taut-split-subst:
assumes vwb-lens x
shows \( 'P' \iff (\forall v. 'P[v/x]') \)
using assms
by (pred-auto, metis vwb-lens-wb wb-lens.source-stability)

Lemma eq-split:
assumes \( 'P \Rightarrow Q', 'Q \Rightarrow P' \)
shows \( P = Q \)
using assms
by (pred-auto)

Lemma bool-eq-splitI:
assumes vwb-lens x \( P[true/x] = Q[true/x] \land P[false/x] = Q[false/x] \)
shows \( P = Q \)
by (metis (full-types) assms eq-split-subst false-alt-def true-alt-def)

Lemma subst-bool-split:
assumes vwb-lens x
shows \( 'P' = '(P[false/x] \land P[true/x])' \)

proof
from assms have \( 'P' = (\forall v. 'P[v/x]' \)
by (subt taut-split-subst[of x], auto)
also have \( ... = (P[true/x] \land P[false/x]) \)
by (metis (mono-tags, lifting))
also have \( ... = (P[false/x] \land P[true/x]) \)
by (pred-auto)

77
finally show \( ?\text{thesis} \).

qed

**lemma** subst-eq-replace:

fixes \( x :: (\alpha \Rightarrow \alpha) \)

shows \((p[u/x] \land u =_u v) = (p[v/x] \land u =_u v)\)

by (pred-auto)

### 13.6 UTP Quantifiers

**lemma** one-point:

assumes \( \text{mwb-lens } x \ x \ # \ v \)

shows \((\exists x \cdot P \land \text{var } x =_u v) = P[v/x]\)

using assms

by (pred-auto)

**lemma** exists-twice: \( \text{mwb-lens } x \Rightarrow (\exists x \cdot \exists x \cdot P) = (\exists x \cdot P)\)

by (pred-auto)

**lemma** all-twice: \( \text{mwb-lens } x \Rightarrow (\forall x \cdot \forall x \cdot P) = (\forall x \cdot P)\)

by (pred-auto)

**lemma** exists-sub: \[ \text{mwb-lens } y; x \subseteq_L y \] \Rightarrow (\exists x \cdot \exists y \cdot P) = (\exists y \cdot P)

by (pred-auto)

**lemma** all-sub: \[ \text{mwb-lens } y; x \subseteq_L y \] \Rightarrow (\forall x \cdot \forall y \cdot P) = (\forall y \cdot P)

by (pred-auto)

**lemma** ex-commute:

assumes \( x \triangleright y \)

shows \((\exists x \cdot \exists y \cdot P) = (\exists y \cdot \exists x \cdot P)\)

using assms

apply (pred-auto)

using lens-indep-comm apply fastforce+

done

**lemma** all-commute:

assumes \( x \triangleright y \)

shows \((\forall x \cdot \forall y \cdot P) = (\forall y \cdot \forall x \cdot P)\)

using assms

apply (pred-auto)

using lens-indep-comm apply fastforce+

done

**lemma** ex-eqv:

assumes \( x \approx_L y \)

shows \((\exists x \cdot P) = (\exists y \cdot P)\)

using assms

by (pred-simp, metis (no-types, lifting) lens.select-convs(2))

**lemma** all-eqv:

assumes \( x \approx_L y \)

shows \((\forall x \cdot P) = (\forall y \cdot P)\)

using assms

by (pred-simp, metis (no-types, lifting) lens.select-convs(2))
lemma ex-zero:
  \((\exists \emptyset \cdot P) = P\)
  by (pred-auto)

lemma all-zero:
  \((\forall \emptyset \cdot P) = P\)
  by (pred-auto)

lemma ex-plus:
  \((\exists y:x \cdot P) = (\exists x \cdot \exists y \cdot P)\)
  by (pred-auto)

lemma all-plus:
  \((\forall y:x \cdot P) = (\forall x \cdot \forall y \cdot P)\)
  by (pred-auto)

lemma closure-all:
  \([P]_u = (\forall \Sigma \cdot P)\)
  by (pred-auto)

lemma unrest-as-exists:
  vwb-lens x \iff (x \not\# P) \leftrightarrow ((\exists x \cdot P) = P)
  by (pred-simp, metis vwb-lens.put-eq)

lemma ex-mono: \(P \sqsubseteq Q \Rightarrow (\exists x \cdot P) \sqsubseteq (\exists x \cdot Q)\)
  by (pred-auto)

lemma ex-weakens: wb-lens x \Rightarrow (\exists x \cdot P) \sqsubseteq P
  by (pred-simp, metis wb-lens.get-put)

lemma all-mono: \(P \sqsubseteq Q \Rightarrow (\forall x \cdot P) \sqsubseteq (\forall x \cdot Q)\)
  by (pred-auto)

lemma all-strengthens: wb-lens x \Rightarrow P \sqsubseteq (\forall x \cdot P)
  by (pred-simp, metis wb-lens.get-put)

lemma ex-unrest: x \not\# P \Rightarrow (\exists x \cdot P) = P
  by (pred-auto)

lemma all-unrest: x \not\# P \Rightarrow (\forall x \cdot P) = P
  by (pred-auto)

lemma not-ex-not: \(\neg (\exists x \cdot \neg P) = (\forall x \cdot P)\)
  by (pred-auto)

lemma not-all-not: \(\neg (\forall x \cdot \neg P) = (\exists x \cdot P)\)
  by (pred-auto)

lemma ex-conj-contr-left: x \not\# P \Rightarrow (\exists x \cdot P \land Q) = (P \land (\exists x \cdot Q))
  by (pred-auto)

lemma ex-conj-contr-right: x \not\# Q \Rightarrow (\exists x \cdot P \land Q) = ((\exists x \cdot P) \land Q)
  by (pred-auto)
13.7 Variable Restriction

**lemma** var-res-all:

\[ P \models \Sigma = P \]

**by** (rel-auto)

**lemma** var-res-twice:

\[ mwb\text{-}lens \ z \rightarrow P \models v \ x \ x = P \models v \ x \]

**by** (pred-auto)

13.8 Conditional laws

**lemma** cond-def:

\[ (P \triangleleft b \triangleright Q) = ((b \land P) \lor ((\neg b) \land Q)) \]

**by** (pred-auto)

**lemma** cond-idem [simp]:\( (P \triangleleft b \triangleright P) = P \)

**by** (pred-auto)

**lemma** cond-true-false [simp]: \( true \triangleleft b \triangleright false = b \)

**by** (pred-auto)

**lemma** cond-symm: \( (P \triangleleft b \triangleright Q) = (Q \triangleleft \neg b \triangleright P) \)

**by** (pred-auto)

**lemma** cond-assoc: \( ((P \triangleleft b \triangleright Q) \triangleleft c \triangleright R) = (P \triangleleft b \land c \triangleright (Q \triangleleft c \triangleright R)) \)

**by** (pred-auto)

**lemma** cond-distr: \( (P \triangleleft b \triangleright (Q \triangleleft c \triangleright R)) = ((P \triangleleft b \triangleright Q) \triangleleft c \triangleright (P \triangleleft b \triangleright R)) \)

**by** (pred-auto)

**lemma** cond-unit-T [simp]:\( (P \triangleleft true \triangleright Q) = P \)

**by** (pred-auto)

**lemma** cond-unit-F [simp]:\( (P \triangleleft false \triangleright Q) = Q \)

**by** (pred-auto)

**lemma** cond-conj-not: \( ((P \triangleleft b \triangleright Q) \land (\neg b)) = (Q \land (\neg b)) \)

**by** (rel-auto)

**lemma** cond-and-T-integrate:

\[ ((P \land b) \lor (Q \triangleleft b \triangleright R)) = ((P \lor Q) \triangleleft b \triangleright R) \]

**by** (pred-auto)

**lemma** cond-L6: \( (P \triangleleft b \triangleright (Q \triangleleft b \triangleright R)) = (P \triangleleft b \triangleright R) \)

**by** (pred-auto)

**lemma** cond-L7: \( (P \triangleleft b \triangleright (P \triangleleft c \triangleright Q)) = (P \triangleleft b \lor c \triangleright Q) \)

**by** (pred-auto)

**lemma** cond-and-distr: \( ((P \land Q) \triangleleft b \triangleright (R \land S)) = ((P \triangleleft b \triangleright R) \land (Q \triangleleft b \triangleright S)) \)

**by** (pred-auto)

**lemma** cond-or-distr: \( ((P \lor Q) \triangleleft b \triangleright (R \lor S)) = ((P \triangleleft b \triangleright R) \lor (Q \triangleleft b \triangleright S)) \)

**by** (pred-auto)

**lemma** cond-imp-distr:

\[ ((P \Rightarrow Q) \triangleleft b \triangleright (R \Rightarrow S)) = ((P \triangleleft b \triangleright R) \Rightarrow (Q \triangleleft b \triangleright S)) \]

**by** (pred-auto)

**lemma** cond-eq-distr:

\[ ((P \Leftrightarrow Q) \triangleleft b \triangleright (R \Leftrightarrow S)) = ((P \triangleleft b \triangleright R) \Leftrightarrow (Q \triangleleft b \triangleright S)) \]

**by** (pred-auto)

**lemma** cond-conj-distr: \( (P \land (Q \triangleleft b \triangleright S)) = ((P \land Q) \triangleleft b \triangleright (P \land S)) \)

**by** (pred-auto)

**lemma** cond-disj-distr: \( (P \lor (Q \triangleleft b \triangleright S)) = ((P \lor Q) \triangleleft b \triangleright (P \lor S)) \)

**by** (pred-auto)

**lemma** cond-neg: \( \neg (P \triangleleft b \triangleright Q) = ((\neg P) \triangleleft b \triangleright (\neg Q)) \)

**by** (pred-auto)
**Lemma** \( \text{cond-conj} \): \( P \triangleleft b \land c \triangleright Q \equiv (P \triangleleft c \triangleright Q) \triangleleft b \triangleright Q \)

by \( \text{pred-auto} \)

**Lemma** \( \text{spec-cond-dist} \): \( (P \Rightarrow (Q \triangleleft b \triangleright R)) = ((P \Rightarrow Q) \triangleleft b \triangleright (P \Rightarrow R)) \)

by \( \text{pred-auto} \)

**Lemma** \( \text{cond-USUP-dist} \): \( \bigcup P \in S \cdot F(P) \triangleleft b \triangleright \bigcup P \in S \cdot G(P) \equiv \bigcup P \in S \cdot F(P) \triangleleft b \triangleright G(P) \)

by \( \text{pred-auto} \)

**Lemma** \( \text{cond-USUP-dist} \): \( \bigcap P \in S \cdot F(P) \triangleleft b \triangleright \bigcap P \in S \cdot G(P) \equiv \bigcap P \in S \cdot F(P) \triangleleft b \triangleright G(P) \)

by \( \text{pred-auto} \)

**Lemma** \( \text{cond-var-subst-left} \):

assumes \( \text{vwb-lens x} \)

shows \( (P[\text{true}/x] \triangleleft \text{var } x \triangleright Q) = (P \triangleleft \text{var } x \triangleright Q) \)

using \( \text{assms} \) by \( \text{(pred-auto, metis (full-types) vwb-lens-wb wb-lens.get-put)} \)

**Lemma** \( \text{cond-var-subst-right} \):

assumes \( \text{vwb-lens x} \)

shows \( (P \triangleleft \text{var } x \triangleright Q[\text{false}/x]) = (P \triangleleft \text{var } x \triangleright Q) \)

using \( \text{assms} \) by \( \text{(pred-auto, metis (full-types) vwb-lens.put-eq)} \)

**Lemma** \( \text{cond-var-split} \):

\( \text{vwb-lens } x \implies (P[\text{true}/x] \triangleleft \text{var } x \triangleright P[\text{false}/x]) = P \)

by \( \text{(rel-simp, (metis (full-types) vwb-lens.put-eq)+)} \)

**Lemma** \( \text{cond-assign-subst} \):

\( \text{vwb-lens } x \implies (P \triangleleft \text{utp-expr.var } x =_u v \triangleright Q) = (P[v/x] \triangleleft \text{utp-expr.var } x =_u v \triangleright Q) \)

apply \( \text{(rel-simp) using vwb-lens.put-eq by force} \)

**Lemma** \( \text{conj-conds} \):

\( (P1 \triangleleft b \triangleright Q1 \land P2 \triangleleft b \triangleright Q2) \equiv (P1 \land P2) \triangleleft b \triangleright (Q1 \land Q2) \)

by \( \text{pred-auto} \)

**Lemma** \( \text{disj-conds} \):

\( (P1 \triangleleft b \triangleright Q1 \lor P2 \triangleleft b \triangleright Q2) \equiv (P1 \lor P2) \triangleleft b \triangleright (Q1 \lor Q2) \)

by \( \text{pred-auto} \)

**Lemma** \( \text{cond-mono} \):

\[ [P_1 \subseteq P_2; Q_1 \subseteq Q_2] \implies (P_1 \triangleleft b \triangleright Q_1) \subseteq (P_2 \triangleleft b \triangleright Q_2) \]

by \( \text{(rel-auto)} \)

**Lemma** \( \text{cond-mono-strict} \):

\[ [\text{mono } P; \text{mono } Q] \implies \text{mono } (\lambda X. P X \triangleleft b \triangleright Q X) \]

by \( \text{(simp add: mono-def, rel-blast)} \)

### 13.9 Additional Expression Laws

**Lemma** \( \text{le-pred-refl simp} \):

fixes \( x :: (a::\text{preorder}, \alpha) \) \( uexpr \)

shows \( (x \leq_u x) = \text{true} \)

by \( \text{(pred-auto)} \)

**Lemma** \( \text{uzero-le-laws simp} \):

\( (\emptyset :: (a::\{\text{linordered-semidom}, \alpha) \text{ uexpr}) \leq_u \text{numeral } x = \text{true} \)
(1 :: ('a::(linordered-semidom), 'α) uexpr) ≤_u numeral x = true
(0 :: ('a::(linordered-semidom), 'α) uexpr) ≤_u 1 = true
by (pred-simp)+

lemma unumeral-le-1 [simp]:
assumes (numeral i :: 'a::{numeral, ord}) ≤ numeral j
shows (numeral i :: ('a, 'α) uexpr) ≤_u numeral j = true
using assms by (pred-auto)

lemma unumeral-le-2 [simp]:
assumes (numeral i :: 'a::{numeral, linorder}) > numeral j
shows (numeral i :: ('a, 'α) uexpr) ≤_u numeral j = false
using assms by (pred-auto)

lemma uset-laws [simp]:
x ∈_u {} = false
x ∈_u {m..n} = (m ≤_u x ∧ x ≤_u n)
by (pred-auto)+

lemma ulit-eq [simp]: x = y =⇒ (≪x≫ =_u ≪y≫) = true
by (rel-auto)

lemma ulit-neq [simp]: x ≠ y =⇒ (≪x≫ =_u ≪y≫) = false
by (rel-auto)

lemma uset-mems [simp]:
x ∈_u {y} = (x =_u y)
x ∈_u A ∪_u B = (x ∈_u A ∨ x ∈_u B)
x ∈_u A ∩_u B = (x ∈_u A ∧ x ∈_u B)
by (rel-auto)+

13.10 Refinement By Observation

Function to obtain the set of observations of a predicate

definition obs-upred :: 'α upred ⇒ 'α set ([].o)
where [upred-defs]: [P].o = {b. [P].b}

lemma obs-upred-refine-iff:
P ⊑ Q =⇒ [Q].o ⊆ [P].o
by (pred-auto)

A refinement can be demonstrated by considering only the observations of the predicates which are relevant, i.e. not unrestricted, for them. In other words, if the alphabet can be split into two disjoint segments, x and y, and neither predicate refers to y then only x need be considered when checking for observations.

lemma refine-by-obs:
assumes x ⊢ y bij-lens (x +_L y) y ⊌ P y ⊌ Q {v. 'P'[≪v≫/x]'} ⊆ {v. 'Q'[≪v≫/x]'}
shows Q ⊆ P
using assms(3-5)
apply (simp add: obs-upred-refine-iff subset-eq)
apply (pred-simp)
apply (rename-tac b)
apply (drule-tac x =_get_x b in spec)
apply (auto simp add: assms)
apply (metis assms(1) assms(2) bij-lens.axioms(2) bij-lens-axioms-def lens-override-def lens-override-plus)+
done

13.11 Cylindric Algebra

lemma C1: (∃ x · false) = false
  by (pred-auto)

lemma C2: wb-lens x ⇒ (P ⇒ (∃ x · P))
  by (pred-simp, metis wb-lens.get-put)

lemma C3: mwb-lens x =⇒ (∃ x · P ⇒ (∃ x · P))
  by (pred-auto)

lemma C4a: x ≈L y =⇒ (∃ x · (∃ x · P)) = (∃ y · (∃ x · P))
  using ex-commute by blast

lemma C4b: x ⊿◁ y =⇒ (∃ x · (∃ y · P)) = (∃ y · (∃ x · P))
  using ex-commute by blast

lemma C5:
  fixes x :: (′a ⇒ ′α)
  shows (λx. u) = true
  by (pred-auto)

lemma C6:
  assumes wb-lens x y z
  shows (λy. u) = (λx. &x =ₚ &x ∧ &x =ₚ &z)
  using assms
  by (pred-simp, (metis lens-indep-def)+)

lemma C7:
  assumes weak-lens x y
  shows ((∃ x · &x =ₚ &y ∧ P)) ∧ (∃ x · &x =ₚ &y ∧ ¬ P)) = false
  using assms
  by (pred-simp, simp add: lens-indep-sym)

end

14 Healthiness Conditions

theory utp-healthy
  imports utp-pred-laws
begin

14.1 Main Definitions

We collect closure laws for healthiness conditions in the following theorem attribute.

named-theorems closure

type-synonym ′α health = ′α upred ⇒ ′α upred

A predicate P is healthy, under healthiness function H, if P is a fixed-point of H.

definition Healthy :: ′α upred ⇒ ′α health ⇒ bool (infix is 30)
where $P$ is $H \equiv (H P = P)$

lemma **Healthy-def**: $P$ is $H \iff (H P = P)$
  unfolding **Healthy-def** by auto

lemma **Healthy-if**: $P$ is $H \implies (H P = P)$
  unfolding **Healthy-def** by auto

lemma **Healthy-intro**: $H(P) = P \implies P$ is $H$
  by (simp add: **Healthy-def**)

declare **Healthy-def’** [upred-defs]

abbreviation **Healthy-carrier** :: $'\alpha$ health $\Rightarrow '\alpha$ upred set $([]_H)$
  where $[[H]]_H \equiv \{P. P \text{ is } H\}$

lemma **Healthy-carrier-image**: $A \subseteq [[H]]_H \implies H^\prime A = A$
  by (auto simp add: image-def, (metis Healthy-if mem-Collect-eq subsetCE)+)

lemma **Healthy-carrier-Collect**: $A \subseteq [[H]]_H \implies A = \{H(P) \mid P. P \in A\}$
  by (simp add: Healthy-carrier-image Setcompr-eq-image)

lemma **Healthy-func**: $[[P \in [[H]]_1 \rightarrow [[H]]_2; P \text{ is } H_1]] \Rightarrow H_2(F(P)) = F(P)$
  using **Healthy-if** by blast

lemma **Healthy-comp**: $[[P \in [[H_1]]_H \rightarrow [[H_2]]_H; P \text{ is } H_1]] \Rightarrow P \text{ is } H_1 \circ H_2$
  by (simp add: **Healthy-def**)

lemma **Healthy-apply-closed**: 
  assumes $F \in [[H]]_H \rightarrow [[H]]_H P \text{ is } H$
  shows $F(P)$ is $H$
  using assms(1) assms(2) by auto

lemma **Healthy-set-image-member**: $[[P \in F \cdot A; \forall x. F x \text{ is } H]] \Rightarrow P \text{ is } H$
  by blast

lemma **Healthy-case-prod** [closure]:
  $[[\forall x y. P x y \text{ is } H]] \Rightarrow \text{case-prod } P \nu \text{ is } H$
  by (simp add: prod.case-eq-if)

lemma **Healthy-SUPREMUM**:
  $A \subseteq [[H]]_H \Rightarrow \operatorname{SUPREMUM} A H = \bigsqcup A$
  by (drule Healthy-carrier-image, presburger)

lemma **Healthy-INFIMUM**:
  $A \subseteq [[H]]_H \Rightarrow \operatorname{INFIMUM} A H = \bigsqcap A$
  by (drule Healthy-carrier-image, presburger)

lemma **Healthy-nu** [closure]:
  assumes mono $F \ F \in [\text{id}]_H \rightarrow [[H]]_H$
  shows $\nu \ F \text{ is } H$

84
by (metis (mono-tags) Healthy-def Healthy-func assms eq-id-iff lfp-unfold)

lemma Healthy-mu [closure]:
assumes mono F F ∈ [id]_H → [H]_H
shows µ F is H
by (metis (mono-tags) Healthy-def Healthy-func assms eq-id-iff gfp-unfold)

lemma Healthy-subset-member: [ A ⊆ [H]_H; P ∈ A ] ⇒ H(P) = P
by (meson Ball-Collect Healthy-if)

lemma is-Healthy-subset-member: [ A ⊆ [H]_H; P ∈ A ] ⇒ P is H
by blast

14.2 Properties of Healthiness Conditions

definition Idempotent :: 'α health ⇒ bool where
Idempotent(H) ←→ (∀ P. H(H(P)) = H(P))

abbreviation Monotonic :: 'α health ⇒ bool where
Monotonic(H) ≡ mono H

definition IMH :: 'α health ⇒ bool where
IMH(H) ←→ Idempotent(H) ∧ Monotonic(H)

definition Antitone :: 'α health ⇒ bool where
Antitone(H) ←→ (∀ P Q. Q ⊑ P −→ (H(P) ⊑ H(Q)))

definition Conjunctive :: 'α health ⇒ bool where
Conjunctive(H) ←→ (∃ Q. ∀ P. H(P) = (P ∧ Q))

definition FunctionalConjunctive :: 'α health ⇒ bool where
FunctionalConjunctive(H) ←→ (∃ F. ∀ P. H(P) = (P ∧ F(P)) ∧ Monotonic(F))

definition WeakConjunctive :: 'α health ⇒ bool where
WeakConjunctive(H) ←→ (∀ P. ∃ Q. H(P) = (P ∧ Q))

definition Disjunctuous :: 'α health ⇒ bool where
[upred-defs]: Disjunctuous H = (∀ P Q. H(P ∩ Q) = (H(P) ∩ H(Q)))

definition Continuous :: 'α health ⇒ bool where
[upred-defs]: Continuous H = (∀ A. A ≠ {} −→ H (∪ A) = ∪ (H ' A))

lemma Healthy-Idempotent [closure]:
Idempotent H ⇒ H(P) is H
by (simp add: Healthy-def Idempotent-def)

lemma Healthy-range: Idempotent H ⇒ range H = [H]_H
by (auto simp add: image-def Healthy-if Healthy-Idempotent, metis Healthy-if)

lemma Idempotent-id [simp]: Idempotent id
by (simp add: Idempotent-def)

lemma Idempotent-comp [intro]:
[ Idempotent f; Idempotent g; f o g = g o f ] −→ Idempotent (f o g)
by (auto simp add: Idempotent-def comp_def, metis)
lemma `Idempotent-image`: $\text{Idempotent } f \Rightarrow f \circ f \cdot A = f \cdot A$
by (metis (mono-tags, lifting) `Idempotent-def image-cong image-image)

lemma `Monotonic-id` [simp]: $\text{Monotonic } id$
by (simp add: monoI)

lemma `Monotonic-id'` [closure]:
mono $(\lambda \cdot X. X)$
by (simp add: monoI)

lemma `Monotonic-const` [closure]:
Monotonic $(\lambda x. c)$
by (simp add: mono-def)

lemma `Monotonic-comp` [intro]:
$[\text{Monotonic } f; \text{Monotonic } g] \Rightarrow \text{Monotonic } (f \circ g)$
by (simp add: mono-def)

lemma `Monotonic-inf` [closure]:
assumes `Monotonic P Monotonic Q`
shows `Monotonic (\lambda X. P(X) \cap Q(X))`
using assms by (simp add: mono-def, rel-auto)

lemma `Monotonic-cond` [closure]:
assumes `Monotonic P Monotonic Q`
shows `Monotonic (\lambda X. P(X) \triangleq b \triangleleft Q(X))`
by (simp add: assms cond-monotonic)

lemma `Conjunctive-Idempotent`:
Conjunctive$(H) \Rightarrow \text{Idempotent}(H)$
by (auto simp add: Conjunctive-def `Idempotent-def`)

lemma `Conjunctive-Monotonic`:
Conjunctive$(H) \Rightarrow \text{Monotonic}(H)$
unfolding Conjunctive-def mono-def
using dual-order.trans by fastforce

lemma `Conjunctive-conj`:
assumes `Conjunctive(HC)`
shows `HC(P \land Q) = (HC(P) \land Q)`
using assms unfolding Conjunctive-def
by (metis `utp-pred-laws.inf.assoc utp-pred-laws.inf.commute`)

lemma `Conjunctive-distr-conj`:
assumes `Conjunctive(HC)`
shows `HC(P \land Q) = (HC(P) \land HC(Q))`
using assms unfolding Conjunctive-def
by (metis `Conjunctive-conj` assms `utp-pred-laws.inf.assoc utp-pred-laws.inf-right-idem`)

lemma `Conjunctive-distr-disj`:
assumes `Conjunctive(HC)`
shows `HC(P \lor Q) = (HC(P) \lor HC(Q))`
using assms unfolding Conjunctive-def
using `utp-pred-laws.inf-sup-distrib2` by fastforce

86
lemma Conjunctive-distr-cond:
  assumes Conjunctive(HC)
  shows HC(P ≪ b ▷ Q) = (HC(P) ≪ b ▷ HC(Q))
  using assms unfolding Conjunctive-def
  by (metis cond-conj-distr utp-pred-laws.inf-commute)

lemma FunctionalConjunctive-Monotonic:
  FunctionalConjunctive(H) ⇒ Monotonic(H)
  unfolding FunctionalConjunctive-def by (metis mono-def utp-pred-laws.inf-mono)

lemma WeakConjunctive-Refinement:
  assumes WeakConjunctive(HC)
  shows P ⊑ HC(P)
  using assms unfolding WeakConjunctive-def by (metis utp-pred-laws.inf.cobounded1)

lemma WeakCojunctive-Healthy-Refinement:
  assumes WeakConjunctive(HC) and P is HC
  shows HC(P) ⊑ P
  using assms unfolding WeakConjunctive-def Healthy-def by simp

lemma WeakConjunctive-implies-WeakConjunctive:
  Conjunctive(H) ⇒ WeakConjunctive(H)
  unfolding WeakConjunctive-def Conjunctive-def by pred-auto

declare Conjunctive-def [upred-defs]
declare mono-def [upred-defs]

lemma Disjunctuous-Monotonic: Disjunctuous H ⇒ Monotonic H
  by (metis Disjunctuous-def mono-def semilattice-sup-class.le-iff-sup)

lemma ContinuousD [dest]: [ Continuous H; A ≠ {} ] ⇒ H (⋂ A) = (⋂ P∈A. H(P))
  by (simp add: Continuous-def)

lemma Continuous-Disjunctuous: Continuous H ⇒ Disjunctuous H
  apply (auto simp add: Continuous-def Disjunctuous-def)
  apply (rename-tac P Q)
  apply (drule-tac x=\{P\,Q\} in spec)
  apply (simp)
  done

lemma Continuous-Monotonic [closure]: Continuous H ⇒ Monotonic H
  by (simp add: Continuous-Disjunctuous Disjunctuous-Monotonic)

lemma Continuous-comp [intro]:
  [ Continuous f; Continuous g ] ⇒ Continuous (f o g)
  by (simp add: Continuous-def)

lemma Continuous-const [closure]: Continuous (λ X. P)
  by pred-auto

lemma Continuous-cond [closure]:
  assumes Continuous F Continuous G
  shows Continuous (λ X. F(X) ≪ b ▷ G(X))
  using assms by (pred-auto)

Closure laws derived from continuity
lemma Sup-Continuous-closed [closure]:
\[\text{Continuous } H \ \land \ i, i \in A \implies P(i) \text{ is } H; \ A \neq \{\} \implies (\prod \ i \in A. P(i)) \text{ is } H\]
by (drule ContinuousD[of H P A], simp add: UINF-mem-UNIV[THEN sym] UINF-as-Sup[THEN sym])

(metis (no-types, lifting) Healthy-def SUP-cong image-image)

lemma UINF-mem-Continuous-closed [closure]:
\[\text{Continuous } H \ \land \ i, i \in A \implies P(i) \text{ is } H; \ A \neq \{\} \implies (\prod \ i \in A \cdot P(i)) \text{ is } H\]
by (simp add: Sup-Continuous-closed assms)

lemma UINF-mem-Continuous-closed-pair [closure]:
assumes Continuous H \( \land \ i, j \in A \implies P i j \text{ is } H\)
shows (\prod \ i, j \in A \cdot P i j) is H
proof -
  show (\prod \ i, j \in A \cdot P i j) = (\prod \ x \in A \cdot P (fst x) (snd x))
    by (rel-auto)
also have ... is H
  by (metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse)
finally show ?thesis .
qed

lemma UINF-mem-Continuous-closed-triple [closure]:
assumes Continuous H \( \land \ i, j, k \in A \implies P i j k \text{ is } H\)
shows (\prod \ i, j, k \in A \cdot P i j k) is H
proof -
  show (\prod \ i, j, k \in A \cdot P i j k) = (\prod \ x \in A \cdot P (fst (snd x)) (snd (snd x)))
    by (rel-auto)
also have ... is H
  by (metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse)
finally show ?thesis .
qed

lemma UINF-mem-Continuous-closed-quad [closure]:
assumes Continuous H \( \land \ i, j, k, l \in A \implies P i j k l \text{ is } H\)
shows (\prod \ i, j, k, l \in A \cdot P i j k l) is H
proof -
  show (\prod \ i, j, k, l \in A \cdot P i j k l) = (\prod \ x \in A \cdot P (fst x) (fst (snd x)) (fst (snd (snd x))) (snd (snd (snd x))))
    by (rel-auto)
also have ... is H
  by (metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse)
finally show ?thesis .
qed

lemma UINF-mem-Continuous-closed-quint [closure]:
assumes Continuous H \( \land \ i, j, k, l, m \in A \implies P i j k l m \text{ is } H\)
shows (\prod \ i, j, k, l, m \in A \cdot P i j k l m) is H
proof -
  show (\prod \ i, j, k, l, m \in A \cdot P i j k l m) = (\prod \ x \in A \cdot P (fst x) (fst (snd x)) (fst (snd (snd x))) (fst (snd (snd (snd x)))) (snd (snd (snd (snd x)))))
    by (rel-auto)
also have ... is H
  by (metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse)
finally show ?thesis .
qed
lemma UINF-ind-closed [closure]:
assumes Continuous H \ \ & i. \ P i = \ true \ & i. \ Q i \ is \ H
shows UINF P Q is H
proof
\[\text{from assms(2) have UINF P Q = } (\prod i \cdot Q i)\]
by (rel-auto)
also have ... is H
using UINF-mem-Continuous-closed[of H UNIV P]
by (simp add: Sup-Continuous-closed UINF-as-Sup-collect' assms)
finally show ?thesis.
qed

All continuous functions are also Scott-continuous

lemma sup-continuous-Continuous [closure]: Continuous F \implies sup-continuous F
by (simp add: Continuous-def sup-continuous-def)

lemma USUP-healthy: A \subseteq [H]_H \implies (\bigsqcup P \in A \cdot F(P)) = (\bigsqcup P \in A \cdot F(H(P)))
by (rule USUP-cong, simp add: Healthy-subset-member)

lemma UINF-healthy: A \subseteq [H]_H \implies (\bigsqcap P \in A \cdot F(P)) = (\bigsqcap P \in A \cdot F(H(P)))
by (rule UINF-cong, simp add: Healthy-subset-member)

end

15 Alphabetised Relations

theory utp-rel
imports
  utp-pred-laws
  utp-healthy
  utp-lift
  utp-tactics
begin

An alphabetised relation is simply a predicate whose state-space is a product type. In this
theory we construct the core operators of the relational calculus, and prove a library of associated
theorems, based on Chapters 2 and 5 of the UTP book [22].

15.1 Relational Alphabets

We set up convenient syntax to refer to the input and output parts of the alphabet, as is common
in UTP. Since we are in a product space, these are simply the lenses \textit{fst}_L and \textit{snd}_L.

definition in \alpha :: ('\alpha \Rightarrow '\alpha \times '\beta) where
[lens-defs]: in \alpha = \textit{fst}_L

definition out \alpha :: ('\beta \Rightarrow '\alpha \times '\beta) where
[lens-defs]: out \alpha = \textit{snd}_L

lemma ina-uvar [simp]: vwb-lens in \alpha
by (unfold-locales, auto simp add: ina-def)
lemma outa-uvar [simp]: vwb-lens outa
  by (unfold-locales, auto simp add: outa-def)

lemma var-in-alpha [simp]: x ;\_\_ ina α = ivar x
  by (simp add: fst-lens-def ina-def in-var-def)

lemma var-out-alpha [simp]: x ;\_\_ outα = oвар x
  by (simp add: outa-def out-var-def snd-lens-def)

lemma drop-pre-inv [simp]: [ outα z p ] \implies [p]_\< \epsilon = p
  by (pred-simp)

lemma usubst-lookup-ivar-unrest [usubst]:
  ina z σ \implies (σ)s (ivar x) = $x
  by (rel-simp, metis fstI)

lemma usubst-lookup-ovar-unrest [usubst]:
  outα z σ \implies (σ)s (ovar x) = $x'
  by (rel-simp, metis sndI)

lemma out-alpha-in-indep [simp]:
  outα ⊿◁ in-var x in-var x ⊿◁ outα
  by (simp-all add: in-var-def ina-def lens-indep-def fst-lens-def snd-lens-def lens-comp-def)

lemma in-alpha-out-indep [simp]:
  inα ⊿◁ out-var x out-var x ⊿◁ inα
  by (simp-all add: in-var-def ina-def lens-indep-def fst-lens-def snd-lens-def lens-comp-def)

The following two functions lift a predicate substitution to a relational one.

abbreviation usubst-rel-lift :: 'α usubst ⇒ ('α × 'β) usubst ([σ]s)
where
[σ]s ≡ σ ⊕ s ina

abbreviation usubst-rel-drop :: ('α × 'α) usubst ⇒ 'α usubst ([σ]s)
where
[σ]s ≡ σ |s inα

The alphabet of a relation then consists wholly of the input and output portions.

lemma alpha-in-out:
  Σ ≅ inα +_L outα
  by (simp add: fst-snd-id-lens ina-def outa-def)

15.2 Relational Types and Operators

We create type synonyms for conditions (which are simply predicates) – i.e. relations without dashed variables –, alphabetised relations where the input and output alphabet can be different, and finally homogeneous relations.

type-synonym 'α cond = 'α upred

translations
(type) ('α, 'β) urel <= (type) ('α × 'β) upred

We set up some overloaded constants for sequential composition and the identity in case we want to overload their definitions later.
consts
  useq :: 'a ⇒ 'b ⇒ 'c (infixr ;; 61)
  uassigns :: 'a usubst ⇒ 'b (\(_a\))
  uskip :: 'a (II)

We define a specialised version of the conditional where the condition can refer only to undashed variables, as is usually the case in programs, but not universally in UTP models. We implement this by lifting the condition predicate into the relational state-space with construction \([b]_<\).

definition lift-rcond ([[-]]) where
[upred-defs]: \([b]_< = [b]_<\)

abbreviation
  rcond :: ('α, 'β) urel ⇒ 'α cond ⇒ ('α, 'β) urel ⇒ ('α, 'β) urel
where \((P \triangleleft b \triangleright_r Q) \equiv (P \triangleleft [b]_< \triangleright Q)\)

Sequential composition is heterogeneous, and simply requires that the output alphabet of the first matches then input alphabet of the second. We define it by lifting HOL’s built-in relational composition operator \(((O))\). Since this returns a set, the definition states that the state binding \(b\) is an element of this set.

lift-definition seqr :: ('α, 'β) urel ⇒ ('β, 'γ) urel ⇒ ('α × 'γ) upred
is \(\lambda P\ Q\ b\ ∈\ ((\{p . P\ p\}) O\ \{q . Q\ q\})\).

adhoc-overloading
  useq seqr

We also set up a homogeneous sequential composition operator, and versions of true and false that are explicitly typed by a homogeneous alphabet.

abbreviation
  segh :: 'α hrel ⇒ 'α hrel ⇒ 'α hrel (infixr ;;h 61) where
segh P Q \equiv (P ;; Q)

abbreviation
  truer :: 'α hrel (trueh) where
truer \equiv true

abbreviation
  falser :: 'α hrel (falseh) where
falser \equiv false

We define the relational converse operator as an alphabet extrusion on the bijective lens swap\(_L\) that swaps the elements of the product state-space.

abbreviation
  conv-r :: ('a, 'α × 'β) uexpr ⇒ ('a, 'β × 'α) uexpr (\(_-\) [999] 999)
where \(conv-r e \equiv e \oplus_p swap\(_L\)\)

Assignment is defined using substitutions, where latter defines what each variable should map to. This approach, which is originally due to Back [3], permits more general assignment expressions. The definition of the operator identifies the after state binding, \(b'\), with the substitution function applied to the before state binding \(b\).

lift-definition
assigns-r :: 'α usubst ⇒ 'α hrel
is \(\lambda \sigma\ b , b' . b' = \sigma(b)\).

adhoc-overloading
  uassigns assigns-r

Relational identity, or skip, is then simply an assignment with the identity substitution: it simply identifies all variables.

91
definition skip-r :: 'a hrel where
[urel-defs]: skip-r = assigns-r id

adhoc-overloading

skip-r

Non-deterministic assignment, also known as “choose”, assigns an arbitrarily chosen value to
the given variable
definition nd-assign :: ('a ⇒ 'α) ⇒ 'α hrel where
[urel-defs]: nd-assign x = (∏ v · assigns-r [x ↦ s v])

We set up iterated sequential composition which iterates an indexed predicate over the elements
of a list.
definition seqr-iter :: 'a list ⇒ ('a ⇒ 'b hrel) ⇒ 'b hrel where
[urel-defs]: seqr-iter xs P = foldr (λ i Q. P (i) ;; Q) xs II

A singleton assignment simply applies a singleton substitution function, and similarly for a
double assignment.

abbreviation assign-r :: ('t =⇒ 'α) ⇒ ('t × 'α) hrel
where assign-r x v ≡ ⟨x ↦ s v⟩

abbreviation assign-2-r :: ('t1 =⇒ 'α) ⇒ ('t2 =⇒ 'α) ⇒ ('t1 × 'α) hrel
where assign-2-r x y u v ≡ assigns-r [x ↦ s u, y ↦ s v]

We also define the alphabetised skip operator that identifies all input and output variables in
the given alphabet lens. All other variables are unrestricted. We also set up syntax for it.
definition skip-ra :: ('β, 'α) lens ⇒ 'α hrel where
[urel-defs]: skip-ra v = ($ v' = a $ v)

Similarly, we define the alphabetised assignment operator.
definition assigns-ra :: 'α usubst ⇒ ('β, 'α) lens ⇒ 'α hrel where
(σ) a = ([σ]s ↦ skip-ra a)

Assumptions (c ⊤) and assertions (c ⊥) are encoded as conditionals. An assumption behaves like
skip if the condition is true, and otherwise behaves like false (miracle). An assertion is the
same, but yields true, which is an abort. They are the same as tests, as in Kleene Algebra
with Tests [24, 1] (KAT), which embeds a Boolean algebra into a Kleene algebra to represent
conditions.
definition rassume :: 'α upred ⇒ 'α hrel where
[urel-defs]: rassume c = II a c V_r false
definition rassert :: 'α upred ⇒ 'α hrel where
[urel-defs]: rassert c = II a c V_r true

We define two variants of while loops based on strongest and weakest fixed points. The former
is false for an infinite loop, and the latter is true.
definition while-top :: 'α cond ⇒ 'α hrel ⇒ 'α hrel where
[urel-defs]: while-top b P = (ν X · (P ;; X) < b V_r II)
definition while-bot :: 'α cond ⇒ 'α hrel ⇒ 'α hrel where
[urel-defs]: while-bot b P = (μ X · (P ;; X) < b V_r II)
While loops with invariant decoration (cf. [1]) – partial correctness.

**definition while-inv :: 'α cond ⇒ 'α hrel ⇒ 'α hrel where**

\[\text{urel-defs}:: \text{while-inv } b \ p \ S = \text{while-top } b \ S\]

While loops with invariant decoration – total correctness.

**definition while-inv-bot :: 'α cond ⇒ 'α hrel ⇒ 'α hrel where**

\[\text{urel-defs}:: \text{while-inv-bot } b \ p \ S = \text{while-bot } b \ S\]

While loops with invariant and variant decorations – total correctness.

**definition while-vrt :: 'α cond ⇒ 'α hrel ⇒ (nat, 'α) uexpr ⇒ 'α hrel ⇒ 'α hrel where**

\[\text{urel-defs}:: \text{while-vrt } b \ p \ v \ S = \text{while-bot } b \ S\]

**syntax**

- **-uassume** :: uexp ⇒ logic \([\TL]\)
- **-uassert** :: uexp ⇒ logic \([\TL]\)
- **-uwhile** :: uexp ⇒ logic ⇒ logic \((\text{while } \bot - \text{ do } - \text{ od})\)
- **-uwhile-top** :: uexp ⇒ logic ⇒ logic \((\text{while } - \text{ do } - \text{ od})\)
- **-uwhile-bot** :: uexp ⇒ logic ⇒ logic \((\text{while } \bot - \text{ do } - \text{ od})\)
- **-uwhile-inv** :: uexp ⇒ uexp ⇒ logic ⇒ logic \((\text{while } - \text{ invr - do } - \text{ od})\)
- **-uwhile-inv-bot** :: uexp ⇒ uexp ⇒ logic ⇒ logic \((\text{while } \bot - \text{ invr - do } - \text{ od})\)
- **-uwhile-vrt** :: uexp ⇒ uexp ⇒ uexp ⇒ logic ⇒ logic \((\text{while } - \text{ invr - vrt - do } - \text{ od})\)

**translations**

- **-uassume b == CONST rassume b**
- **-uassert b == CONST rassert b**
- **-uwhile b P == CONST while-top b P**
- **-uwhile-top b P == CONST while-top b P**
- **-uwhile-bot b P == CONST while-bot b P**
- **-uwhile-inv b p S == CONST while-inv b p S**
- **-uwhile-inv-bot b p S == CONST while-inv-bot b p S**
- **-uwhile-vrt b p v S == CONST while-vrt b p v S**

We implement a poor man’s version of alphabet restriction that hides a variable within a relation.

**definition rel-var-res :: 'α hrel ⇒ ('a ⇒ 'α) ⇒ 'α hrel (infix \(\uparrow\)) where**

\[\text{urel-defs}: P \uparrow x = (\exists \text{ } x · \exists \text{ } x' · P)\]

Alphabet extension and restriction add additional variables by the given lens in both their primed and unprimed versions.

**definition rel-aext :: 'α hrel ⇒ ('α ⇒ 'β) ⇒ 'β hrel where**

\[\text{upred-defs}: \text{rel-aext } P a = P \oplus_p (a \times L a)\]

**definition rel-ares :: 'α hrel ⇒ ('β ⇒ 'α) ⇒ 'β hrel where**

\[\text{upred-defs}: \text{rel-ares } P a = (P \upharpoonright_p (a \times a))\]

We next describe frames and antiframes with the help of lenses. A frame states that \(P\) defines how variables in \(a\) changed, and all those outside of \(a\) remain the same. An antiframe describes the converse: all variables outside \(a\) are specified by \(P\), and all those in remain the same. For more information please see [25].

**definition frame :: ('a ⇒ 'α) ⇒ 'α hrel ⇒ 'α hrel where**

\[\text{urel-defs}: \text{frame } a P = (P \land \text{ $v' =a $v} \oplus \text{ $v' on } &a)\]

93
definition antiframe :: ('a ⇒ α) ⇒ α hrel ⇒ α hrel where
 urel-defs: antiframe a P = (P ∧ $v' =< α v' ⊕ $v on &a)

Frame extension combines alphabet extension with the frame operator to both add additional variables and then frame those.

definition rel-frext :: ('β ⇒ α) ⇒ β hrel ⇒ α hrel where
 urel-defs: rel-frext a P = frame a (rel-aext P a)

The nameset operator can be used to hide a portion of the after-state that lies outside the lens a. It can be useful to partition a relation’s variables in order to conjoin it with another relation.

definition nameset :: ('a ⇒ α) ⇒ α hrel ⇒ α hrel where
 urel-defs: nameset a P = (P \ P \ {sv, sv'} )

15.3 Syntax Translations

syntax
— Alternative traditional conditional syntax

-utp-if : uexp ⇒ logic ⇒ logic ⇒ logic ((if α (-)/ then (-)/ else (-)) [0, 0, 71] 71)
— Iterated sequential composition

-seq-iter : pitrn ⇒ 'a list ⇒ σ hrel ⇒ σ hrel ((3; - : - /) [0, 0, 10] 10)
— Single and multiple assignment

-assignment :: svids ⇒ uexprs ⇒ 'a hrel (('-' := '(-'))

-assignment :: svids ⇒ uexprs ⇒ 'a hrel (infixr := 62)
— Non-deterministic assignment

-nd-assign :: svids ⇒ logic (- := * [62] 62)
— Substitution constructor

-mk-usubst :: svids ⇒ uexprs ⇒ 'a usubst
— Alphabetised skip

-skip-ra :: salpha ⇒ logic (II.)
— Frame

-frame :: salpha ⇒ logic ⇒ logic (-[·] [99,0] 100)
— Antiframe

-antiframe :: salpha ⇒ logic ⇒ logic (-[·] [79,0] 80)
— Relational Alphabet Extension

-rel-aext :: logic ⇒ salpha ⇒ logic (infixl := 90)
— Relational Alphabet Restriction

-rel-ares :: logic ⇒ salpha ⇒ logic (infixl := 90)
— Frame Extension

-rel-frext :: salpha ⇒ logic ⇒ logic (-[·]+ [99,0] 100)
— Nameset

-nameset :: salpha ⇒ logic ⇒ logic (ns - · - [0, 999] 999)

translations

-utp-if b P Q ==⇒ P ∧ b ⇒ Q

; x : l : P == (CONST seqr-iter) l (λx. P)

-mk-usubst σ (-svid-unit x) v =⇒ σ(&x ↦+, v)

-mk-usubst σ (-svid-list x xs) (-uexprs v vs) ==⇒ (-mk-usubst (σ(&x ↦+, v)) xs vs)

-assignment xs vs ==⇒ CONST uassigns (-mk-usubst (CONST id) xs vs)

-assignment x v <=⇒ CONST uassigns (CONST subst-upd (CONST id) x v)

-assignment x v <=⇒ -assignment (-spvar x) v

-nd-assign x ==⇒ CONST nd-assign (-mk-svid-list x)

-nd-assign x <=⇒ CONST nd-assign x

x,y := u,v <=⇒ CONST uassigns (CONST subst-upd (CONST subst-upd (CONST id) (CONST svar x) u) (CONST svar y) v)

94
The following code sets up pretty-printing for homogeneous relational expressions. We cannot do this via the “translations” command as we only want the rule to apply when the input and output alphabet types are the same. The code has to deconstruct a (′a, ′α) uexpr type, determine that it is relational (product alphabet), and then checks if the types alpha and beta are the same. If they are, the type is printed as a hexpr. Otherwise, we have no match. We then set up a regular translation for the hrel type that uses this.

```
print-translation :
let
  fun tr' ctxt [ a, Const (@{type-syntax prod},-) $ alpha $ beta ] =
    if (alpha = beta)
      then Syntax.const @{type-syntax hexpr} $ a $ alpha
      else raise Match;
  in [(@{type-syntax uexpr},tr')]
end
```

```
translations
(type) ′a hrel <= (type) (bool, ′a) hexpr
```

### 15.4 Relation Properties

We describe some properties of relations, including functional and injective relations. We also provide operators for extracting the domain and range of a UTP relation.

```
definition afun :: (′a, ′b) urel ⇒ bool
  where [urel-defs]: afun R ≡ H ⊑ R− ;; R
```

```
definition uinj :: (′a, ′b) urel ⇒ bool
  where [urel-defs]: uinj R ≡ H ⊑ R ;; R−
```

```
definition Dom :: ′a hrel ⇒ ′a upred
  where [upred-defs]: Dom P = [∃ $v′ · P]<
```

```
definition Ran :: ′a hrel ⇒ ′a upred
  where [upred-defs]: Ran P = [∃ $v · P]>
```

— Configuration for UTP tactics.

```
```

### 15.5 Introduction laws

```
lemma urel-refine-ext:
  [ ∃ s s′. P[<s>,<s'>/$v,$v′] ⊑ Q[<s>,<s'>/$v,$v′] ] ⇒ P ⊑ Q
```
by (rel-auto)

**lemma urel-eq-ext:**
\[
\left[ \left[ \bigwedge s s'. \left[ P|_{s<s'>} / s \right] = Q|_{s<s'>} / s \right] \right] \Rightarrow P = Q
\]
by (rel-auto)

15.6 Unrestriction Laws

**lemma unrest-iuvar [unrest]:** out α ⩾ $x$
by (metis fst-snd-lens-indep lift-pre-var out α-def unrest-aext-indep)

**lemma unrest-ouvar [unrest]:** in α ⩾ $x'$
by (metis in α-def lift-post-var snd-fst-lens-indep unrest-aext-indep)

**lemma unrest-semir-undash [unrest]:**
fixes $x :: (\alpha = \Rightarrow \alpha)$
assumes $x ≧ P$
shows $x ≧ P :: Q$
using assms by (rel-auto)

**lemma unrest-semir-dash [unrest]:**
fixes $x :: (\alpha = \Rightarrow \alpha)$
assumes $x' ≧ Q$
shows $x' ≧ P :: Q$
using assms by (rel-auto)

**lemma unrest-cond [unrest]:**
\[
\left[ x ≧ P; x ≧ b; x ≧ Q \right] \Rightarrow x ≧ b ∨ Q
\]
by (rel-auto)

**lemma unrest-lift-rcond [unrest]:**
x ⩾ [b]< \Rightarrow x ⩾ [b]_<
by (simp add: lift-rcond-def)

**lemma unrest-ino-var [unrest]:**
\[
\left[ \text{mwb-lens } x; \text{ino } ≧ (P :: (\alpha, (\alpha × \beta)) \text{ uexpr}) \right] \Rightarrow ≧ P
\]
by (rel-auto)

**lemma unrest-outo-var [unrest]:**
\[
\left[ \text{mwb-lens } x; \text{outo } ≧ (P :: (\alpha, (\alpha × \beta)) \text{ uexpr}) \right] \Rightarrow ≧ P'
\]
by (rel-auto)

**lemma unrest-pre-outo [unrest]:** outo ≧ [b]<
by (transfer, auto simp add: outo-def)

**lemma unrest-post-ino [unrest]:** ino ≧ [b]>
by (transfer, auto simp add: ino-def)

**lemma unrest-pre-in-var [unrest]:**
x ⩾ p1 \Rightarrow ≧ [p1]<
by (transfer, simp)

**lemma unrest-post-out-var [unrest]:**
x ⩾ p1 \Rightarrow ≧ p1>
by (transfer, simp)
lemma unrest-convr-out [unrest]:
\[ \text{in} \alpha \downharpoonright p \implies \text{out} \alpha \downharpoonright p^- \]
by (transfer, auto simp add: lens-defs)

lemma unrest-convr-in [unrest]:
\[ \text{out} \alpha \downharpoonright p \implies \text{in} \alpha \downharpoonright p^- \]
by (transfer, auto simp add: lens-defs)

lemma unrest-in-rel-var-res [unrest]:
\[ \text{vwb-lens} \ x \implies x \upharpoonright (P |\alpha x) \]
by (simp add: rel-var-res-def unrest)

lemma unrest-out-rel-var-res [unrest]:
\[ \text{vwb-lens} \ x \implies x' \upharpoonright (P |\alpha x) \]
by (simp add: rel-var-res-def unrest)

lemma unrest-out-alpha-usubst-rel-lift [unrest]:
\[ \text{out} \alpha \upharpoonright \sigma \implies \sigma \upharpoonright (P \oplus_r a) \]
by (rel-auto)

lemma unrest-in-rel-aext [unrest]:
\[ x \triangleright y \implies y \upharpoonright (P \oplus_r a) \]
by (simp add: rel-aext-def unrest-aext-indep)

lemma unrest-out-rel-aext [unrest]:
\[ x \triangleright y \implies y' \upharpoonright (P \oplus_r a) \]
by (simp add: rel-aext-def unrest-aext-indep)

lemma rel-aext-false [alpha]:
\[ \text{false} \oplus_r a = \text{false} \]
by (pred-auto)

lemma rel-aext-seq [alpha]:
\[ \text{weak-lens} \ a \implies (P ;; Q) \oplus_r a = (P \oplus_r a ;; Q \oplus_r a) \]
apply (rel-auto)
apply (rename_tac aa b y)
apply (rule-tac x=\text{create}_a y in exI)
apply (simp)
done

lemma rel-aext-cond [alpha]:
\[ (P \triangleright b \triangleright_r Q) \oplus_r a = (P \oplus_r a \triangleright b \triangleright_p a \triangleright_r Q \oplus_r a) \]
by (rel-auto)

15.7 Substitution laws

lemma subst-seq-left [usubst]:
\[ \text{out} \alpha \downharpoonright \sigma \implies \sigma \upharpoonright (P ;; Q) = (\sigma \upharpoonright P) ;; Q \]
by (rel-simp, (metis (no-types, lifting) Pair-inject surjective-pairing)+)

lemma subst-seq-right [usubst]:
\[ \text{in} \alpha \downharpoonright \sigma \implies \sigma \upharpoonright (P ;; Q) = P ;; (\sigma \upharpoonright Q) \]
by (rel-simp, (metis (no-types, lifting) Pair-inject surjective-pairing)+)

The following laws support substitution in heterogeneous relations for polymorphically typed literal expressions. These cannot be supported more generically due to limitations in HOL’s type system. The laws are presented in a slightly strange way so as to be as general as possible.
lemma bool-seqr-laws [usubst]:

fixes $x :: (\text{bool} \Rightarrow 'a)$

shows
\[ \begin{align*}
&P \cdot Q \cdot \sigma(x \mapsto\_, \text{true}) \triangledown (P :: \sigma[\text{true}/\sigma x] :: Q) \\
&P \cdot Q \cdot \sigma(x \mapsto\_, \text{false}) \triangledown (P :: \sigma[\text{false}/\sigma x] :: Q) \\
&P \cdot Q \cdot \sigma(x' \mapsto\_, \text{true}) \triangledown (P :: \sigma[\text{true}/\sigma x'] :: Q) \\
&P \cdot Q \cdot \sigma(x' \mapsto\_, \text{false}) \triangledown (P :: \sigma[\text{false}/\sigma x']) \\
\end{align*}\]

by (rel-auto)+

lemma zero-one-seqr-laws [usubst]:

fixes $x :: (\text{false} \Rightarrow 'a)$

shows
\[ \begin{align*}
&P \cdot Q \cdot \sigma(x \mapsto\_, \text{false}) \triangledown (P :: \sigma[\text{false}/\sigma x] :: Q) \\
&P \cdot Q \cdot \sigma(x' \mapsto\_, \text{false}) \triangledown (P :: \sigma[\text{false}/\sigma x'] :: Q) \\
\end{align*}\]

by (rel-auto)+

lemma numeral-seqr-laws [usubst]:

fixes $x :: (\text{numeral} n \Rightarrow 'a)$

shows
\[ \begin{align*}
&P \cdot Q \cdot \sigma(x \mapsto\_, \text{numeral} n) \triangledown (P :: \sigma[\text{numeral} n/\sigma x] :: Q) \\
&P \cdot Q \cdot \sigma(x' \mapsto\_, \text{numeral} n) \triangledown (P :: \sigma[\text{numeral} n/\sigma x'] :: Q) \\
\end{align*}\]

by (rel-auto)+

lemma usubst-condr [usubst]:

$\sigma \triangledown (P \triangleleft\triangleleft \sigma \triangleleft\triangleleft b \triangleleft\triangleleft Q) = (\sigma \triangledown P \triangleleft\triangleleft b \triangleleft\triangleleft \sigma \triangleleft\triangleleft Q)$

by (rel-auto)

lemma subst-skip-r [usubst]:

$\sigma \triangleleft\triangleleft \text{out} \triangleleft\triangleleft \text{in} \triangleleft\triangleleft \sigma \triangleleft\triangleleft \text{II} = (\langle \sigma \rangle)_{\text{a}}$

by (rel-auto)

lemma subst-pre-skip [usubst]: $[\sigma]_{\text{s}} \triangleleft\triangleleft \text{II} = \langle \sigma \rangle_{\text{s}}$

by (rel-auto)

lemma subst-rel-lift-seq [usubst]:

$[\sigma]_{\text{s}} \triangleleft\triangleleft (P :: Q) = ([\sigma]_{\text{s}} \triangleleft\triangleleft P) :: Q$

by (rel-auto)

lemma subst-rel-lift-comp [usubst]:

$[\sigma]_{\text{s}} \circ \langle q \rangle_{\text{s}} = [\sigma \circ q]_{\text{s}}$

by (rel-auto)

lemma usubst-upd-in-comp [usubst]:

$\sigma(\text{kin} \triangleleft\triangleleft \text{in} \triangleleft\triangleleft \sigma \triangleleft\triangleleft x \mapsto\_, \_ \triangleleft\triangleleft v) = \sigma(x \mapsto\_, \_ \triangleleft\triangleleft v)$

by (simp add: pr-var-def fst-lens-def ina-def in-var-def)

lemma usubst-upd-out-comp [usubst]:

$\sigma(\text{out} \triangleleft\triangleleft \text{out} \triangleleft\triangleleft \sigma \triangleleft\triangleleft x \mapsto\_, \_ \triangleleft\triangleleft v) = \sigma(x' \mapsto\_, \_ \triangleleft\triangleleft v)$

by (simp add: pr-var-def outa-def out-var-def snd-lens-def)

lemma subst-lift-upd [alpha]:

fixes $x :: (\text{('a} \Rightarrow \text{'a})$

shows $[\sigma(x \mapsto\_, \_ \triangleleft\triangleleft v)]_{\text{s}} = [\sigma]_{\text{s}}(\_ \mapsto\_, \_ \triangleleft\triangleleft \_ <)$

98
by (simp add: alpha usubst, simp add: pr-var-def fst-lens-def inα-def in-var-def)

lemma subst-drop-upd [alpha]:
  fixes x :: ('a ⇒ 'α)
  shows \[\sigma(s x \mapsto v)_s = \sigma(x \mapsto_s [v])\]
  by pred-simp

lemma subst-lift-pre [usubst]: \[\sigma s \uparrow [b] = \sigma \uparrow [b] <\]
  by (metis apply-subst-ext fst-vwb-lens inα-def)

lemma unrest-usubst-lift-in [unrest]:
  \[x \not\in P \Longrightarrow \{x \mapsto \} [P]_s\]
  by pred-simp

lemma unrest-usubst-lift-out [unrest]:
  fixes x :: ('a ⇒ 'α)
  shows \$x \not\in [P]_s\]
  by pred-simp

lemma subst-lift-cond [usubst]: \[\sigma s \uparrow s \leftarrow = \sigma \uparrow s \leftarrow\]
  by (rel-auto)

lemma msubst-seq [usubst]: \((P ;; Q) \leftarrow \{a \mapsto v\} = ((P \leftarrow \{a \mapsto v\}) ;; (Q \leftarrow \{a \mapsto v\}))\]
  by (rel-auto)

15.8 Alphabet laws

lemma aext-cond [alpha]:
  \((P \leftarrow b \leftarrow Q) \oplus p a = ((P \oplus_p a) \leftarrow (b \oplus_p a) \leftarrow (Q \oplus_p a))\]
  by (rel-auto)

lemma aext-seq [alpha]:
  \((P ;; Q) \leftarrow r a = ((P \leftarrow r a) ;; (Q \leftarrow r a))\]
  by (rel-simp, metis wb-lens-weak weak-lens.put-get)

lemma rcond-lift-true [simp]:
  \[true \leftarrow = true\]
  by rel-auto

lemma rcond-lift-false [simp]:
  \[false \leftarrow = false\]
  by rel-auto

lemma rel-ares-aext [alpha]:
  \(\{a \mapsto\} \leftarrow (P \oplus_r a) \leftarrow r a = P\)
  by (rel-auto)

lemma rel-aext-ares [alpha]:
  \(\{a \mapsto\} \leftarrow (P \oplus_r a) \leftarrow r a \leftarrow a = P\)
  by (rel-auto)

lemma rel-aext-uses [unrest]:
  \(\{a \mapsto\} \leftarrow (P \oplus_r a)\]
  by (rel-auto)
15.9 Relational unrestriction

Relational unrestriction states that a variable is both unchanged by a relation, and is not "read" by the relation.

**Definition** $RID : (\alpha \rightarrow \alpha) \Rightarrow \alpha \text{ hrel} \Rightarrow \alpha \text{ hrel}

where $RID x P = ((\exists \ x \cdot \exists \ x' \cdot P) \land \ x' = \ u \ x$)

**Declare** $RID-def$ [urel-defs]

**Lemma** $RID1$: $\text{vwb-lens } x \Rightarrow (\forall \ x : = \ <v> \ ; \ P = P \ ; \ x : = \ <v>) \Rightarrow RID(x)(P) = P$

apply (rel-auto)
apply (metis vwb-lens.put-eq)
apply (metis vwb-lens-wb wb-lens.get-put wb-lens-weak weak-lens.put-get)
done

**Lemma** $RID2$: $\text{vwb-lens } x \Rightarrow x := <v> ; ; \ P = RID(x)(P) ; ; x := <v>$

apply (rel-auto)
apply blast
done

**Lemma** $RID-assign-commute$:
$\text{vwb-lens } x \Rightarrow P = RID(x)(P) \leftrightarrow (\forall \ x : = \ <v> \ ; \ P = P \ ; \ x := <v>)$

by (metis RID1 RID2)

**Lemma** $RID-idem$:
$\text{mwb-lens } x \Rightarrow RID(x)(RID(x)(P)) = RID(x)(P)$

by (rel-auto)

**Lemma** $RID-mono$:
$P \subseteq Q \Rightarrow RID(x)(P) \subseteq RID(x)(Q)$

by (rel-auto)

**Lemma** $RID-pr-var$ [simp]:
$RID(\text{pr-var } x) = RID x$

by (simp add: pr-var-def)

**Lemma** $RID-skip-r$:
$\text{vwb-lens } x \Rightarrow RID(x)(II) = II$

apply (rel-auto) using vwb-lens.put-eq by fastforce

**Lemma** $\text{skip-r-RID}$ [closure]: $\text{vwb-lens } x \Rightarrow II \text{ is RID}(x)$

by (simp add: Healthy-def RID-skip-r)

**Lemma** $RID-disj$:
$RID(x)(P \lor Q) = (RID(x)(P) \lor RID(x)(Q))$

by (rel-auto)

**Lemma** $\text{disj-RID}$ [closure]: $[ P \text{ is } RID(x); Q \text{ is } RID(x) ] \Rightarrow (P \lor Q) \text{ is } RID(x)$

by (simp add: Healthy-def RID-disj)

**Lemma** $RID-conj$:
$\text{vwb-lens } x \Rightarrow RID(x)(RID(x)(P) \land RID(x)(Q)) = (RID(x)(P) \land RID(x)(Q))$

by (rel-auto)
lemma conj-RID [closure]: \([\text{vwb-lens } x; \ P \text{ is } \text{RID}(x); \ Q \text{ is } \text{RID}(x)] \implies (P \land Q) \text{ is } \text{RID}(x)\]
   by (metis Healthy-if Healthy-intro RID-conj)

lemma RID-assigns-r-diff:
\[
[\text{vwb-lens } x; \ x \notin \sigma] \implies \text{RID}(x)(\langle \sigma \rangle_a) = (\sigma)_a
\]
   apply (rel-auto)
   apply (metis vwb-lens.put-eq)
   apply (metis vwb-lens-vwb wb-lens.get-put wb-lens-weak weak-lens.put-get)
   done

lemma assigns-r-RID [closure]: \([\text{vwb-lens } x; \ x \notin \sigma] \implies (\sigma)_a \text{ is } \text{RID}(x)\]
   by (simp add: Healthy-def RID-assigns-r-diff)

lemma RID-assigns-r-same:
\[
\text{vwb-lens } x \implies \text{RID}(x)(x := v) = \Pi
\]
   apply (rel-auto)
   using vwb-lens.put-eq apply fastforce
   done

lemma RID-seq-left:
   assumes \text{vwb-lens } x
   shows \text{RID}(x)(\text{RID}(x)(P) ; Q) = (\text{RID}(x)(P) ; \text{RID}(x)(Q))
   proof
   have \text{RID}(x)(\text{RID}(x)(P) ; Q) = ((\exists \ x. \exists \ x' \cdot ((\exists \ x \cdot \exists \ x' \cdot P) \land x' =_u x) ; Q) \land x' =_u x)
      by (simp add: RID-def usubst)
   also from \text{assms have} \ldots = (((\exists \ x \cdot \exists \ x' \cdot P) ; (\exists \ x \cdot \exists \ x' \cdot Q)) \land x' =_u x)
      by (rel-auto)
   also from \text{assms have} \ldots = ((\exists \ x \\cdot \exists \ x' \cdot P) ; (\exists \ x \cdot \exists \ x' \cdot Q)) \land x' =_u x)
      apply (rel-auto)
      apply (metis mwb-lens.put-put vwb-lens-mwb)
      done
   also from \text{assms have} \ldots = (((\exists \ x \cdot \exists \ x' \cdot P) \land x' =_u x) ; (\exists \ x \cdot \exists \ x' \cdot Q) \land x' =_u x)
      by (rel-simp, metis (full-types) mwb-lens.put-put vwb-lens-def wb-lens-weak weak-lens.put-get)
   also have \ldots = (((\exists \ x \cdot \exists \ x' \cdot P) \land x' =_u x) ; (\exists \ x \cdot \exists \ x' \cdot Q) \land x' =_u x))
      by (rel-auto)
   finally show \ldots
   qed

lemma RID-seq-right:
   assumes \text{vwb-lens } x
   shows \text{RID}(x)(P ; \text{RID}(x)(Q)) = (\text{RID}(x)(P) ; \text{RID}(x)(Q))
   proof
   have \text{RID}(x)(P ; \text{RID}(x)(Q)) = ((\exists \ x \cdot \exists \ x' \cdot P ; ((\exists \ x \cdot \exists \ x' \cdot Q) \land x' =_u x)) \land x' =_u x)
      by (simp add: RID-def usubst)
   also from \text{assms have} \ldots = ((\exists \ x \cdot P ; (\exists \ x \cdot \exists \ x' \cdot Q) \land (\exists \ x' \cdot x' =_u x)) \land x' =_u x)}
by (rel-auto)
also from asms have ... = (((∀ x. ∃ x' P) ∧ (∃ x. ∃ x' Q)) ∧ x' = u x)
  apply (rel-auto)
  apply (metis vwb-lens.put-eq)
  apply (metis mwb-lens.put-put vwb-lens-mwb)
done
also from asms have ... = (((∃ x. ∃ x' P) ∧ x' = u x) ;; (∃ x. ∃ x' Q) ∧ x' = u x)
  by (rel-simp robust, metis (full-types) mwb-lens.put-put vwb-lens-def wb-lens-weak weak-lens.put-get)
also have ... = (RID x P ;; RID x Q)
  by (rel-auto)
finally show ?thesis .

qed

lemma seq-RID-closed [closure]: [ vwb-lens x; P is RID x; Q is RID x ] MODP P ;; Q is RID x
  by (metis Healthy-def RID-seq-right)

definition unrest-relation :: ('a MODP 'a hrel MODP bool (infix "\#\#") 20)
where (x \#\# P) MODP (P is RID x)

declare unrest-relation-def [urels]

lemma runrest-assign-commute:
  [ vwb-lens x; x \#\# P ] MODP x := v ;; P = P ;; x := v
  by (metis RID2 Healthy-def unrest-relation-def)

lemma runrest-ident-var:
  assumes x \#\# P
  shows (x = P) = (P ∧ x)
proof
  have P = (x' = u x ∧ P)
    by (metis RID-def asms Healthy-def unrest-relation-def utp-pred-laws.inf.cobounded2 utp-pred-laws.inf.absorb2)
  moreover have (x' = u x ∧ (x ∧ P)) = (x' = u x ∧ (P ∧ x'))
    by (rel-auto)
  ultimately show ?thesis
    by (metis utp-pred-laws.inf.assoc utp-pred-laws.inf.left-commute)
qed

lemma skip-r-runrest [unrest]:
  vwb-lens x MODP x \#\# II
  by (simp add: unrest-relation-def closure)

lemma assigns-r-runrest:
  [ vwb-lens x; x \#\# σ ] MODP x \#\# (σ)u
  by (simp add: unrest-relation-def closure)

lemma seq-r-runrest [unrest]:
  assumes vwb-lens x x \#\# P x \#\# Q
  shows x \#\# (P ;; Q)
  using asms by (simp add: unrest-relation-def closure)
16 Fixed-points and Recursion

theory utp-recursion
  imports
    utp-pred-laws
    utp-rel
begin

16.1 Fixed-point Laws

lemma mu-id: (μ X · X) = true
  by (simp add: antisym gfp-upperbound)

lemma mu-const: (μ X · P) = P
  by (simp add: gfp-const)

lemma nu-id: (ν X · X) = false
  by (meson lfp-lowerbound utp-pred-laws.bot.extremum-unique)

lemma nu-const: (ν X · P) = P
  by (simp add: lfp-const)

lemma mu-refine-intro:
  assumes (C ⇒ S) ⊑ F(C ⇒ S) (C ∧ μ F) = (C ∧ ν F)
  shows (C ⇒ S) ⊑ μ F
proof –
  from assms have (C ⇒ S) ⊑ ν F
    by (simp add: lfp-lowerbound)
  with assms show ?thesis
    by (pred-auto)
qed

16.2 Obtaining Unique Fixed-points

Obtaining termination proofs via approximation chains. Theorems and proofs adapted from
Chapter 2, page 63 of the UTP book [22].

type-synonym 'a chain = nat ⇒ 'a upred

definition chain :: 'a chain ⇒ bool where
  chain Y = ((Y 0 = false) ∧ (∀ i. Y (Suc i) ⊑ Y i))

lemma chain0 [simp]: chain Y ⇒ Y 0 = false
  by (simp add: chain-def)
lemma chainI:
  assumes \( Y 0 = \text{false} \land i. \ (Suc\ i) \subseteq Y i \)
  shows chain \( Y \)
  using assms by (auto simp add: chain-def)

lemma chainE:
  assumes chain \( Y \) \( \land i. \ [ Y 0 = \text{false}; \ (Suc\ i) \subseteq Y i ] \implies P \)
  shows \( P \)
  using assms by (simp add: chain-def)

lemma L274:
  assumes \( \forall n. \ ((E\ n \land Y) = (E\ n \land Y)) \)
  shows \( \prod (\text{range}\ E) \land X) = (\prod (\text{range}\ E) \land Y) \)
  using assms by (pred-auto)

Constructive chains
definition constr ::
  \('a upred \Rightarrow 'a upred \Rightarrow bool\)
where
  constr \( F\ E \leftarrow\rightarrow \) chain \( E \) \( \land (\forall X\ n. \ ((F\ X) \land E\ (n + 1)) = (F\ X \land E\ n) \land E\ (n + 1)))\)

lemma constrI:
  assumes chain \( E \) \( \land X\ n. \ ((F\ X) \land E\ (n + 1)) = (F\ X \land E\ n) \land E\ (n + 1)))\)
  shows constr \( F\ E \)
  using assms by (auto simp add: constr-def)

This lemma gives a way of showing that there is a unique fixed-point when the predicate function can be built using a constructive function \( F \) over an approximation chain \( E \)

lemma chain-pred-terminates:
  assumes constr \( F\ E\ mono\ F \)
  shows \( \prod \) (\text{range}\ E) \land \( \mu F \) = \( \prod \) (\text{range}\ E) \land \( \nu F \)
proof –
  from assms have \( \forall n. \ (E\ n \land \mu F) = (E\ n \land \nu F) \)
proof (rule-tac allI)
  fix \( n \)
  from assms show \( (E\ n \land \mu F) = (E\ n \land \nu F) \)
proof (induct \( n \))
    case 0 thus \(?case\ by\ (simp\ add: constr-def)\)
  next
    case (Suc \( n \))
    note hyp = this
    thus \(?case\)
    proof –
      have \( (E\ (n + 1) \land \mu F) = (E\ (n + 1) \land F\ (\mu F)) \)
      using gfp-unfold[OF hyp(3)], THEN sym| by (simp add: constr-def)
      also from hyp have \( \ldots = (E\ (n + 1) \land F\ (E\ n \land \mu F)) \)
      by (metis conj-comm constr-def)
      also from hyp have \( \ldots = (E\ (n + 1) \land F\ (E\ n \land \nu F)) \)
      by simp
      also from hyp have \( \ldots = (E\ (n + 1) \land \nu F) \)
      by (metis (no-types, lifting) conj-comm constr-def lfp-unfold)
      ultimately show \(?thesis\)
      by simp
    qed
  qed
The next lemma shows that using substitution also works. However, it is not that generic nor practical for proof automation...

**lemma** refine-usubst-to-ueq:

- **assumes** constr_F_E mono F \( \prod (\text{range } E) = C \)
- **shows** \( \text{assms}(1) \text{ assms}(2) \text{ assms}(3) \text{ chain-pred-terminates by blast} \)

16.3 Noetherian Induction Instantiation

Contribution from Yakoub Nemouchi. The following generalization was used by Tobias Nipkow and Peter Lammich in *Refine, Monadic*.

**lemma** wf-fixp-uniq:

- **assumes** fixp-unfold: \( fp \ B = B \ (fp \ B) \)
- **and** \( \text{WF: wf } R \)
- **and** \( \text{induct-step:} \)

\[
\forall f \ st. \ \forall st'. (st', st) \in R \implies (((\text{Pre} \land [e]_<_u st') \implies \text{Post}) \subseteq f) \\
\implies fp \ B = f \implies ((\text{Pre} \land [e]_<_u st) \implies \text{Post}) \subseteq (B \ f)
\]

**shows** \( (\text{Pre} \implies \text{Post}) \subseteq (fp \ B) \)

**proof**

\[
\begin{align*}
\{ & \text{ fix } st \\
& \text{ have } ((\text{Pre} \land [e]_<_u st') \implies \text{Post}) \subseteq (fp \ B) \\
& \text{ using } \text{WF proof (induction rule: wf-induct-rule)} \\
& \quad \text{ case (less } x) \\
& \quad \text{ hence } (\text{Pre} \land [e]_<_u x \implies \text{Post}) \subseteq B \ (fp \ B) \\
& \quad \text{ by (rule induct-step, rel-blast, simp)} \\
& \quad \text{ then show } ?\text{case} \\
& \quad \quad \text{ using fixp-unfold by auto} \\
& \} \\
\text{ thus } ?\text{thesis} \\
\text{ by pred-simp} \\
\text{ qed}
\end{align*}
\]

The next lemma shows that using substitution also work. However it is not that generic nor practical for proof automation ...

**lemma** refine-usubst-to-ueq:

\[
\text{vwb-lens } E \implies (\text{Pre} \implies \text{Post}) [\text{st'}/E] \subseteq f [\text{st'}/E] = ((\text{Pre} \land E = u \text{ st'}) \implies \text{Post}) \subseteq f
\]

**by** (rel-auto, metis vwb-lens-vb wb-lens.get-put)

By instantiation of \( [[?fp ?B = ?B (\text{ families })]; \ (\forall f \ st. \ (\forall st'. (st', st) \in ?R \implies ((\text{Pre} \land [?e]_<_u st') \implies \text{Post}) \subseteq f) \land (?fp ?B) \implies (?\text{Pre} \land [?e]_<_u st) \implies (?\text{Post}) \subseteq ?B \ f)] \)

**implies** \( (?\text{Pre} \implies ?\text{Post}) \subseteq (?fp ?B) \text{ with } \mu \text{ and lifting of the well-founded relation we have } ...

**lemma** mu-rec-total-pure-rule:

- **assumes** \( \text{WF: wf } R \)
- **and** \( M: \text{mono } B \)
- **and** \( \text{induct-step:} \)

\[
\forall f \ st. \ ((\text{Pre} \land ([e]_<_u st'))_u \in R_\Rightarrow \text{Post}) \subseteq f \\
\implies \mu \ B = f \implies (\text{Pre} \land [e]_<_u st) \Rightarrow \text{Post}) \subseteq (B \ f)
\]

**shows** \( (\text{Pre} \Rightarrow \text{Post}) \subseteq \mu \ B \)

**proof** (rule wf-fixp-uniq-pure-ueq-gen[where \( fp=\mu \text{ and } \text{Pre}=\text{Pre} \text{ and } B=B \text{ and } R=R \text{ and } e=e \])

**show** \( \mu \ B = B \ (\mu \ B) \)
proof

(\text{nu-rec-total-utp-rule})

\begin{align*}
\text{qed}
\end{align*}

\begin{lemma}
\text{nu-rec-total-pure-rule:}
\end{lemma}

\begin{proof}
\text{wf R}
\end{proof}

\begin{lemma}
\text{mu-rec-total-utp-rule:}
\end{lemma}

\begin{proof}
\text{wf R}
\end{proof}

\begin{lemma}
\text{nu-rec-total-utp-rule:}
\end{lemma}

\begin{proof}
\text{wf R}
\end{proof}

\text{end}
17  Sequent Calculus

theory utp-sequent
  imports utp-pred-laws
begin

  definition sequent :: 'α upred ⇒ 'α upred ⇒ bool (infixr ⊢) where
      [upred-defs]: sequent P Q = (Q ⊑ P)

  abbreviation sequent-triv (⊢ [15] 15) where ⊢ P ≡ (true ⊢ P)

  translations
      ⊢ P ≡ true ⊢ P

  lemma sTrue: P ⊢ true
      by pred-auto

  lemma sAx: P ⊢ P
      by pred-auto

  lemma sNotI: Γ ∧ P ⊢ false ⇒ Γ ⊢ ¬ P
      by pred-auto

  lemma sConjI: [ Γ ⊢ P; Γ ⊢ Q ] ⇒ Γ ⊢ P ∧ Q
      by pred-auto

  lemma sImplI: [ (Γ ∧ P) ⊢ Q ] ⇒ Γ ⊢ (P ⇒ Q)
      by pred-auto

end

18  Relational Calculus Laws

theory utp-rel-laws
  imports utp-rel
  utp-recursion
begin

18.1  Conditional Laws

  lemma comp-cond-left-distr:
      ((P ≪ b ⊳ r, Q) ;; R) = ((P ;; R) ≪ b ⊳ r (Q ;; R))
      by (rel-auto)

  lemma cond-seq-left-distr:
      outa ≻ b ⇒ ((P ≪ b ⊳ Q) ;; R) = ((P ;; R) ≪ b ⊳ (Q ;; R))
      by (rel-auto)

  lemma cond-seq-right-distr:
      ina ≻ b ⇒ (P ;; (Q ≪ b ⊳ R)) = ((P ;; Q) ≪ b ⊳ (P ;; R))
      by (rel-auto)

  Alternative expression of conditional using assumptions and choice

  lemma rcond-rassume-expand: P ≪ b ⊳ r Q = ([b]T ;; P) ∩ ([¬ b][T] ;; Q)
18.2 Precondition and Postcondition Laws

**theorem** precond-equiv:
\[
P = (P ;; true) \iff (\text{out} \alpha \not\in P)
\]
by (rel-auto)

**theorem** postcond-equiv:
\[
P = (true ;; P) \iff (\text{in} \alpha \not\in P)
\]
by (rel-auto)

**lemma** precond-right-unit: out \alpha \not\in p \implies (p ;; true) = p
by (metis precond-equiv)

**lemma** postcond-left-unit: in \alpha \not\in p \implies (true ;; p) = p
by (metis postcond-equiv)

**theorem** precond-left-zero:
assumes out \alpha \not\in p p \neq false
shows (true ;; p) = true
using assms by (rel-auto)

**theorem** feasible-iff-true-right-zero:
\[
P ;; true = true \iff \exists \text{out} \alpha \cdot P
\]
by (rel-auto)

18.3 Sequential Composition Laws

**lemma** seqr-assoc: (P ;; Q) ;; R = P ;; (Q ;; R)
by (rel-auto)

**lemma** seqr-left-unit [simp]:
\[
I ;; P = P
\]
by (rel-auto)

**lemma** seqr-right-unit [simp]:
\[
P ;; I = P
\]
by (rel-auto)

**lemma** seqr-left-zero [simp]:
\[
false ;; P = false
\]
by pred-auto

**lemma** seqr-right-zero [simp]:
\[
P ;; false = false
\]
by pred-auto

**lemma** impl-seqr-mono: [ ‘P \Rightarrow Q’; ‘R \Rightarrow S’ ] \Rightarrow ‘(P ;; R) \Rightarrow (Q ;; S)’
by (pred-blast)

**lemma** seqr-mono:
\[
[P_1 \subseteq P_2; Q_1 \subseteq Q_2] \Rightarrow (P_1 ;; Q_1) \subseteq (P_2 ;; Q_2)
\]
by (rel-blast)

**lemma** seqr-monotonic:
\[ \text{mono } P; \text{ mono } Q \implies \text{mono } (\lambda X. \text{ P } X \text{ ; } Q \text{ X}) \]

by (simp add: mono-def, rel-blast)

**lemma** Monotonic-seqr-tail [closure]:

assumes Monotonic F

shows Monotonic (\lambda X. P ;; F(X))

by (simp add: assms monoD monoI seqr-mono)

**lemma** seqr-exists-left:

\((\exists x \cdot \text{ P } x) ;; Q) = (\exists x \cdot (\text{ P } ;; Q))\)

by (rel-auto)

**lemma** seqr-exists-right:

\((P ;; (\exists x \cdot Q)) = (\exists x \cdot (P ;; Q))\)

by (rel-auto)

**lemma** seqr-or-distl:

\((P \lor Q) ;; R) = ((P ;; R) \lor (Q ;; R))\)

by (rel-auto)

**lemma** seqr-or-distr:

\((P ;; (Q \lor R)) = ((P ;; Q) \lor (P ;; R))\)

by (rel-auto)

**lemma** seqr-inf-distl:

\((P \sqcap Q) ;; R) = ((P ;; R) \sqcap (Q ;; R))\)

by (rel-auto)

**lemma** seqr-inf-distr:

\((P ;; (Q \sqcap R)) = ((P ;; Q) \sqcap (P ;; R))\)

by (rel-auto)

**lemma** seqr-and-distr-ufunc:

ufunctional P \implies (P ;; (Q \land R)) = ((P ;; Q) \land (P ;; R))

by (rel-auto)

**lemma** seqr-and-distl-uinj:

uinj R \implies ((P \land Q) ;; R) = ((P ;; R) \land (Q ;; R))

by (rel-auto)

**lemma** seqr-unfold:

\((P ;; Q) = (\exists v \cdot P[<\text{v}>/\$v\cdot] \land Q[<\text{v}>/\$v])\)

by (rel-auto)

**lemma** seqr-middle:

assumes vwb-lens x

shows \((P ;; Q) = (\exists v \cdot P[<\text{v}>/\$x\cdot] ;; Q[<\text{v}>/\$x])\)

using assms

by (rel-auto', metis vwb-lens-wb wb-lens.source-stability)

**lemma** seqr-left-one-point:

assumes vwb-lens x

shows \((P \land x = u \cdot <v>) ;; Q) = (P[<\text{v}>/\$x\cdot] ;; Q[<\text{v}>/\$x])\)

using assms

by (rel-auto, metis vwb-lens-wb wb-lens.get-put)
lemma seqr-right-one-point:
  assumes vwb-lens x
  shows \((P \land (x \land Q)) = (P[\text{true}/x'] \land Q[\text{true}/x])\)
  using assms
  by (rel-auto, metis vwb-lens-wb wb-lens.get-put)

lemma seqr-left-one-point-true:
  assumes vwb-lens x
  shows \((P \land \neg x') \land Q) = (P[\text{false}/x'] \land Q[\text{false}/x])\)
  by (metis assms seqr-left-one-point true-alt-def upred-eq-true)

lemma seqr-left-one-point-false:
  assumes vwb-lens x
  shows \((P \land \neg x') \land Q) = (P[\text{false}/x'] \land Q[\text{false}/x])\)
  by (metis assms false-alt-def seqr-left-one-point upred-eq-false)

lemma seqr-right-one-point-true:
  assumes vwb-lens x
  shows \((P \land (x \land Q)) = (P[\text{true}/x'] \land Q[\text{true}/x])\)
  by (metis assms seqr-right-one-point true-alt-def upred-eq-true)

lemma seqr-right-one-point-false:
  assumes vwb-lens x
  shows \((P \land (x \land Q)) = (P[\text{true}/x'] \land Q[\text{true}/x])\)
  by (metis assms seqr-right-one-point true-alt-def upred-eq-true)

lemma seqr-insert-ident-left:
  assumes vwb-lens x \(x' \neq P \land Q\)
  shows \(((x' =_u \land x \land Q)) = (P \land Q)\)
  using assms
  by (rel-simp, meson vwb-lens-wb wb-lens-weak weak-lens,put-get)

lemma seqr-insert-ident-right:
  assumes vwb-lens x \(x' \neq P \land Q\)
  shows \(((x' =_u \land x \land Q)) = (P \land Q)\)
  using assms
  by (rel-simp, metis (no-types, hide-lams) vwb-lens-def wb-lens-def weak-lens,put-get)

lemma seq-var-ident-lift:
  assumes vwb-lens x \(x' \neq P \land Q\)
  shows \(((x' =_u \land x \land (P \land Q))) = (\land x' =_u \land x \land (P \land Q))\)
  using assms by (rel-auto', metis (no-types, lifting) vwb-lens-wb wb-lens-weak weak-lens,put-get)

lemma seqr-bool-split:
  assumes vwb-lens x
  shows \(P \lor Q \land (P[\text{true}/x'] \lor Q[\text{false}/x'] \lor R[\text{false}/x] = R[\text{false}/x])\)
  using assms
  by (subst seqr-middle[of x], simp-all)

lemma cond-inter-var-middle:
  assumes vwb-lens x
  shows \((P \land \neg x' \land Q) \lor (R[\text{true}/x'] \lor R[\text{false}/x] = R[\text{false}/x])\)
  proof
    have \((P \land \neg x' \land Q) \lor (R[\text{true}/x'] \lor R[\text{false}/x] = R)\)

by (simp add: cond-def seqr-or-distl)
also have \( y = ((P \land \neg x) ; R \lor (Q \land \neg x) ; R) \)
by (rel-auto)
also have \( y = (P[\text{true}/x] ; R[\text{true}/x] \lor Q[\text{false}/x] ; R[\text{false}/x]) \)
by (simp add: seqr-left-one-point-true seqr-left-one-point-false assms)
finally show \(?thesis\).
qed

theorem seqr-pre-transfer: \( \alpha \# q \implies ((P \land q) ; R) = (P ; (q \land R)) \)
by (rel-auto)

theorem seqr-pre-transfer':
\( ((P \land [q]_{<}) ; R) = (P ; ([q]_{<} \land R)) \)
by (rel-auto)

theorem seqr-post-out: \( \alpha \# r \implies (P ; (Q \land r)) = ((P ; Q) \land r) \)
by (rel-blast)

lemma seqr-post-var-out:
fixes \( x :: (\text{bool} \implies 'a) \)
shows \( (P ; (Q \land \neg x)) = ((P ; Q) \land \neg x) \)
by (rel-auto)

theorem seqr-post-transfer: \( \alpha \# q \implies (P ; (Q \land R)) = ((P \land Q) ; R) \)
by (rel-auto)

lemma seqr-pre-out: \( \alpha \# p \implies ((p \land Q) ; R) = (p \land (Q ; R)) \)
by (rel-blast)

lemma seqr-pre-var-out:
fixes \( x :: (\text{bool} \implies 'a) \)
shows \( ((Q \land P) ; X) = (Q \land (P ; X)) \)
by (rel-auto)

lemma seqr-true-lemma:
\( (P = (\neg ((\neg P) ; \text{true}))) = (P = (P ; \text{true})) \)
by (rel-auto)

lemma seqr-to-conj: \[ \text{out} \alpha \# P ; \text{in} \alpha \# Q \] \implies (P ; Q) = (P \land Q)
by (metis postcond-left-unit seqr-pre-out utp-pred-laws.inf-top.right-neutral)

lemma shEx-lift-seq-1 [uquant-lift]:
\( \exists x \cdot P x ; Q = \exists x \cdot (P x ; Q) \)
by rel-auto

lemma shEx-mem-lift-seq-1 [uquant-lift]:
assumes \( \text{out} \alpha \# A \)
shows \( \exists x \in A \cdot P x ; Q = \exists x \in A \cdot (P x ; Q) \)
using assms by rel-blast

lemma shEx-lift-seq-2 [uquant-lift]:
\( P ; \exists x \cdot Q x \) = \( \exists x \cdot (P ; Q x) \)
by rel-auto

lemma shEx-mem-lift-seq-2 [uquant-lift]:
assumes \( \alpha \not
in A \)
shows \( (P ;; (\exists x \in A \cdot Q x)) = (\exists x \in A \cdot (P ;; Q x)) \)
using assms by rel-blast

18.4 Iterated Sequential Composition Laws

lemma iter-seq-nil [simp]: \( i : [] \cdot P(i) = II \)
by (simp add: seqr-iter-def)

lemma iter-seq-cons [simp]: \( i : (x \# xs) \cdot P(i) = P(x) ;; (i : xs \cdot P(i)) \)
by (simp add: seqr-iter-def)

18.5 Quantale Laws

lemma seq-Sup-distl \( P ;; (\prod A \cdot P Q x) = (\prod A \cdot P ;; Q x) \)
by (transfer, auto)

lemma seq-Sup-distr \( (\prod A) ;; Q = (\prod P \cdot P ;; Q) \)
by (transfer, auto)

lemma seq-UINF-distl \( P ;; (\prod Q \cdot F(Q)) = (\prod Q \cdot P ;; F(Q)) \)
by (simp add: UINF-as-Sup-collect seq-Sup-distl)

lemma seq-UINF-distl' \( (\prod Q \cdot F(Q)) ;; Q = (\prod P \cdot F(P) ;; Q) \)
by (metis UINF-mem-UNIV seq-UINF-distl)

lemma seq-SUP-distl \( P ;; (\prod i \cdot F(i)) = (\prod i \cdot P ;; F(i)) \)
by (simp add: seq-Sup-distr)

lemma seq-SUP-distl' \( (\prod i \cdot P(i)) ;; Q = (\prod i \cdot P ;; Q) \)
by (simp add: seq-Sup-distr)

18.6 Skip Laws

lemma cond-skip: \( \alpha \not
in b \implies (b \land II) = (II \land b) \)
by (rel-auto)

lemma pre-skip-post: \( (b \land II) = (II \land \alpha) \)
by (rel-auto)

lemma skip-var:
\begin{itemize}
  \item fixes \( x :: \bullet \alpha \)
  \item shows \( (\land x \land II) = (II \land \alpha) \)
  \item by (rel-auto)
\end{itemize}

lemma skip-r-unfold:
\begin{itemize}
  \item fixes \( x :: \bullet \alpha \)
  \item shows \( (\land x \land II) = (II \land \alpha) \)
  \item by (rel-auto)
\end{itemize}

lemma skip-r-alpha-eq:
\( II = (\$v \leftarrow u \$v) \)
by (rel-auto)

**lemma skip-ra-unfold:**
\( \Pi x_{; y} = (\$x \leftarrow u \$x \land \Pi y) \)
by (rel-auto)

**lemma skip-res-as-ra:**
\[
\llbracket \text{vwb-lens } y ; x +_{L} y \approx_{L} I_{L} ; x \sqsubseteq y \rrbracket \implies \Pi x_{; y} = \Pi y
\]
apply (rel-auto)
apply (metis (no-types, lifting) lens-indep-def)
apply (metis vwb-lens, put-eq)
done

### 18.7 Assignment Laws

**lemma assigns-subst [usubst]:**
\[
[\sigma]_{s} \mapsto (g)_{a} = (g \circ \sigma)_{a}
\]
by (rel-auto)

**lemma assigns-r-comp: \((\sigma)_{a} ;; P) = ([\sigma]_{s} \mapsto P)\)**
by (rel-auto)

**lemma assigns-r-feasible:**
\((\sigma)_{a} ;; \text{true}) = \text{true} \)
by (rel-auto)

**lemma assign-subst [usubst]:**
\[
\llbracket \text{mwb-lens } x ; \text{mwb-lens } y \rrbracket \implies [\$x \mapsto u]_{\prec} \mapsto (y := v) = (x, y) := (u, [x \mapsto u] \mapsto v)
\]
by (rel-auto)

**lemma assign-vacuous-skip:**
assumes vwb-lens x
shows \((x := &x) = \Pi \)
using assms by rel-auto

The following law shows the case for the above law when \(x\) is only mainly-well behaved. We require that the state is one of those in which \(x\) is well defined using and assumption.

**lemma assign-vacuous-assume:**
assumes mwb-lens x
shows \(((x := &x) = (y := v)) = (y := v)) \)
using assms by rel-auto

**lemma assign-simultaneous:**
assumes vwb-lens y x \(\sqsupseteq y\)
shows \((x, y) := (e, \&y) = (x := e) \)
by (simp add: assms usubst-upd-comm usubst-upd-var-id)

**lemma assigns-idem: mwb-lens x \implies (x, x) := (u, v) = (x := v)\)**
by (simp add: usubst)

**lemma assigns-comp: \((f)_{a} ;; (g)_{a}) = (g \circ f)_{a}\)**
by (simp add: assigns-r-comp usubst)

**lemma assigns-cond: \((f)_{a} \sqsubseteq b \triangleright_{r} (g)_{a}) = (f \sqsubseteq b \triangleright_{s} g)_{a}\)**
by (rel-auto)

lemma assigns-r-conv:
bij f \implies \langle f \rangle_a = \langle \inv f \rangle_a
by (rel-auto, simp-all add: bij-is-inj bij-is-surj surj-f-inv-f)

lemma assign-pred-transfer:
fixes x :: (\alpha \Rightarrow \alpha)
assumes $x \notin b \alpha \iff b$
shows (\beta \land x := v) = (x := v \land b)
using assms by (rel-blast)

lemma assign-r-comp:
fixes x :: (\alpha \Rightarrow \alpha)
shows (x := u ;; P) = (x := f)
by (simp add: assigns-r-comp usubst alpha)

lemma assign-test:
mwb-lens x = \implies (x := \ell u ;; x := \ell v) = (x := v)
by (simp add: assigns-comp usubst)

lemma assign-twice:
mwb-lens x \Longrightarrow (x := e ;; x := f) = (x := f)
by (simp add: assigns-comp usubst unrest)

lemma assign-commute:
assumes x \triangleright \bigtriangledown y x \triangleright f y \triangleright e
shows (x := e ;; (P \triangleleft b \triangleright Q)) = ((x := e ;; (P \triangleleft (b[e/x]\triangleleft \ell u) \triangleright (x := e ;; Q))
by (rel-auto)

lemma assign-cond:
fixes x :: (\alpha \Rightarrow \alpha)
assumes \theta b
shows (x := e ;; (P \triangleleft b \triangleright Q)) = \bigtriangleup Q
by (rel-auto)

lemma assign-r-alt-def:
fixes x :: (\alpha \Rightarrow \alpha)
shows \theta = H[\ell v] < \ell u
by (rel-auto)

lemma assigns-r-ufunc: ufunctional \langle f \rangle_a
by (rel-auto)

lemma assigns-r-uj: inj f \implies uj f
by (rel-simp, simp add: inj-eq)

lemma assigns-r-swap-uj:
mwb-lens x ;; mwb-lens y ;; x \triangleright y \Longrightarrow uj ((x,y) := (ky,kx))
by (metis assigns-r-uj pr-var-def swap-usubst-uj)

lemma assign-unfold:
mwb-lens x \Longrightarrow (x := v) = (u \uparrow \ell v < \ell H[\ell u] x)
apply (rel-auto, auto simp add: comp-def)
using vwb-lens.put-eq by fastforce

18.8 Non-deterministic Assignment Laws

lemma nd-assign-comp:
\[ x \bowtie y \Rightarrow x := \star ; y := \star, y := \star \]
apply (rel-auto) using lens-indep-comm by fastforce+

lemma nd-assign-assign:
\[ \text{⟦} \text{vwb-lens } x; x \# e \text{⟧} \Rightarrow x := \star ; x := e = x := e \]
by (rel-auto)

18.9 Converse Laws

lemma convr-invol [simp]: \( \neg \neg p = p \)
by pred-auto

lemma lit-convr [simp]: \( \langle v \rangle^{-} = \langle v \rangle \)
by pred-auto

lemma uivar-convr [simp]:
fixes \( x :: (\alpha \Rightarrow 'a) \)
shows \( \langle \$$x$$ \rangle^{-} = \$$x$$' \)
by pred-auto

lemma uovar-convr [simp]:
fixes \( x :: (\alpha \Rightarrow 'a) \)
shows \( \langle \$$x'$$ \rangle^{-} = \$$x \)
by pred-auto

lemma uop-convr [simp]: \( \text{uop } f \ u^{-} = \text{uop } f \ (u^{-}) \)
by (pred-auto)

lemma bop-convr [simp]: \( \text{bop } f \ u \ v^{-} = \text{bop } f \ (u^{-}) \ (v^{-}) \)
by (pred-auto)

lemma eq-convr [simp]: \( \text{p }=\text{u } q^{-} = (p^{-} =\text{u } q^{-}) \)
by (pred-auto)

lemma not-convr [simp]: \( \neg p^{-} = (\neg p^{-}) \)
by (pred-auto)

lemma disj-convr [simp]: \( p \lor q^{-} = (q^{-} \lor p^{-}) \)
by (pred-auto)

lemma conj-convr [simp]: \( p \land q^{-} = (q^{-} \land p^{-}) \)
by (pred-auto)

lemma seqr-convr [simp]: \( p \bowtie q^{-} = (q^{-} \bowtie p^{-}) \)
by (rel-auto)

lemma pre-convr [simp]: \( [p]_{<}^{-} = [p]_{>} \)
by (rel-auto)

lemma post-convr [simp]: \( [p]_{<}^{-} = [p]_{<} \)
18.10 Assertion and Assumption Laws

declare sublens-def [lens-defs del]

lemma assume-false: [false]\top = false
  by (rel-auto)

lemma assume-true: [true]\top = II
  by (rel-auto)

lemma assume-seq: [b]\top ;; [c]\top = [(b \land c)]\top
  by (rel-auto)

lemma assert-false: {false}⊥ = true
  by (rel-auto)

lemma assert-true: {true}⊥ = II
  by (rel-auto)

lemma assert-seq: {b}⊥ ;; {c}⊥ = {(b \land c)}⊥
  by (rel-auto)

18.11 Frame and Antiframe Laws

named-theorems frame

lemma frame-all [frame]: Σ:[P] = P
  by (rel-auto)

lemma frame-none [frame]:
  ∅:[P] = (P \land II)
  by (rel-auto)

lemma frame-commute:
  assumes $y \nz P \ni y \ni P \ni x \nz Q \ni x \ni Q x \ni Q x \ni Q y \ni Q
  shows x:[P] ;; y:[Q] = y:[Q] ;; x:[P]
  apply (insert assms)
  apply (rel-auto)
  apply (rename-tac s s' s_0)
  apply (subgoal-tac (s ⊔_L s' on y) ⊔_L s_0 on x = s_0 ⊔_L s' on y)
  apply (metis lens-indep-get lens-indep-sym lens-override-def)
  apply (simp add: lens-indep.lens-put-comm lens-override-def)
  apply (rename-tac s s' s_0)
  apply (subgoal-tac put_y (put_x s (get_x (put_x s_0 (get_x s')))) (get_y (put_y s (get_y s_0))))
  = put_x s_0 (get_x s'))
  apply (metis lens-override-get lens-indep-sym)
  apply (metis lens-indep.lens-put-comm)
  done

lemma frame-contract-RID:
  assumes vvwb-lens x P is RID(x) x ⊵ y
  shows (x;y):[P] = y:[P]
  proof –
  from assms(1,3) have (x;y):[RID(x)(P)] = y:[RID(x)(P)]
apply (rel-auto)
apply (simp add: lens-indep.lens-put-comm)
apply (metis (no-types) vwb-lens-wb wb-lens.get-put)
done
thus ?thesis
by (simp add: Healthy-if assms)
qed

lemma frame-miracle [simp]:
x:[false] = false
by (rel-auto)

lemma frame-skip [simp]:
vwb-lens x  \implies x:[II] = II
by (rel-auto)

lemma frame-assign-in [frame]:
\[[ vwb-lens a; x \subseteq L a ] \implies a:\{x := v\} = x := v
by (rel-auto, simp-all add: lens-get-put-quasi-commute lens-put-of-quotient)

lemma frame-conj-true [frame]:
\[[ \{x, x'\} \natural P; vwb-lens x \] \implies (P \land x:[true]) = x:[P]
by (rel-auto)

lemma frame-is-assign [frame]:
vwb-lens x  \implies x:[x' = u \lceil v \rceil < ] = x := v
by (rel-auto)

lemma frame-seq [frame]:
\[[ vwb-lens a; \{x, x'\} \natural P; \{x, x'\} \natural Q \] \implies x:[P ;; Q] = x:[P] ;; x:[Q]
apply (rel-auto)
apply (metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens-def weak-lens.put-get)
done

lemma frame-to-antiframe [frame]:
\[[ x \natural y; x + \subseteq L y = 1_L ] \implies x:[P] = y:[P]
by (rel-auto, metis lens-indep-def, metis lens-indep-def surj-pair)

lemma rel-frext-miracle [frame]:
a:[false]^+ = false
by (rel-auto)

lemma rel-frext-skip [frame]:
vwb-lens a  \implies a:[II]^+ = II
by (rel-auto)

lemma rel-frext-seq [frame]:
vwb-lens a  \implies a:[P ;; Q]^+ = (a:[P]^+ ;; a:[Q]^+)
apply (rel-auto)
apply (rename-tac s s' s_0)
apply (rule-tac x=put_a s s_0 in exI)
apply (auto)
apply (metis mwb-lens.put-put vwb-lens-mwb)
done

117
lemma rel-frext-assigns [frame]:
  vwb-lens a \implies a:\langle(\sigma)_a\rangle^+ = (\sigma \oplus\_\_ a)_a
by (rel-auto)

lemma rel-frext-rcond [frame]:
  a:\langle P \land b \triangleright_r Q \rangle^+ = (a:\langle P \rangle^+ \land b \triangleright P \land a\triangleright_r a:\langle Q \rangle^+)
by (rel-auto)

lemma rel-frext-commute:
  x \triangleright y \implies x:\langle P \rangle^+ ;; y:\langle Q \rangle^+ = y:\langle Q \rangle^+ ;; x:\langle P \rangle^+
apply (rel-auto)
  apply (rename-tac a c b)
  apply (subgoal-tac \(a\ land b\land get_y(put_x b a) = get_y b\))
  apply (metis (no-types, hide-lams) lens-indep-comm lens-indep-get)
  apply (simp add: lens-indep.lens-put-irr2)
  apply (metis vwb-lens-wb wb-lens-def weak-lens.put-get)
  apply (simp add: lens-indep.lens-put-irr2)
done

lemma antiframe-disj [frame]: (x:\langle P \rangle^+ \lor x:\langle Q \rangle^+) = x:\langle P \lor Q \rangle^+
by (rel-auto)

lemma antiframe-seq [frame]:
  \[ vwb-lens x; \# x\ C P; \# x\ C Q \] = \[
apply (rel-auto)
  apply (metis vwb-lens-wb wb-lens-def weak-lens.put-get)
  apply (simp add: lens-indep-lens-put-irr2)
done

lemma nameset-skip: vwb-lens x \implies (ns x \cdot II) = II x
by (rel-auto, meson vwb-lens-wb wb-lens.get-put)

lemma nameset-skip-ra: vwb-lens x \implies (ns x \cdot II x) = II x
by (rel-auto)

decclare sublens-def [lens-defs]

18.12 While Loop Laws

theorem while-unfold:
  while b do P od = ((P ;; while b do P od) \land b \triangleright_r II)
proof
  have m:mono \((\lambda X. (P ;; X) \land b \triangleright_r II)\)
    by (auto intro: monoI seqr-mono cond-mono)
  have (while b do P od) = (\nu X. (P ;; X) \land b \triangleright_r II)
    by (simp add: while-top-def)
  also have ... = (((\nu X. (P ;; X) \land b \triangleright_r II)) \land b \triangleright_r II)
    by (subgoal-tac subst lfp-unfold, simp-all add: m)
  also have ... = ((P ;; while b do P od) \land b \triangleright_r II)
    by (simp add: while-top-def)
  finally show ?thesis .
qed
**Theorem while-true**: while true do P od = false

**Proof**

apply (simp add: while-top-def alpha)
apply (rule antisym)
apply (rule simp-all)
apply (rule lfp-lowerbound)
apply (rule rel-auto)
done

**Theorem while-unfold**: while⊥ b do P od = ((P ;; while⊥ b do P od) < b bopp II)

**Proof**

have m:mono (λX. (P ;; X) < b bopp II)
  by (auto intro: monoI seqr-mono cond-mono)
have (while⊥ b do P od) = (µ X · (P ;; X) < b bopp II)
  by (simp add: while-bot-def)
also have ... = (((P ;; (µ X · (P ;; X) < b bopp II)) < b bopp II)
  by (simp add: while-bot-def)
also have ... = (((P ;; while⊥ b do P od) < b bopp II)
  by (simp add: while-bot-def)
finally show ?thesis.

qed

**Theorem while-bot-false**: while⊥ false do P od = II

**By** (simp add: while-bot-def mu-const alpha)

**Theorem while-bot-true**: while⊥ true do P od = (µ X · P ;; X)

**By** (simp add: while-bot-def alpha)

An infinite loop with a feasible body corresponds to a program error (non-termination).

**Theorem while-infinite**: P ;; trueb = true ⇒ while⊥ true do P od = true

**Apply** (simp add: while-bot-true)
apply (rule antisym)
apply (rule simp)
apply (rule gfp-upperbound)
apply (simp)
done

### 18.13 Algebraic Properties

**Interpretation** upred-semiring: semiring-I

where times = seqr and one = skip-r and zero = falseb and plus = Lattices.sup

by (unfold-locales, (rel-auto)+)

**Declare** upred-semiring.power-Suc [simp del]

We introduce the power syntax derived from semirings

**Abbreviation** upower :: 'a hrel ⇒ nat ⇒ 'a hrel (infixr "^") where

upower P n ≡ upred-semiring.power P n

**Translations**

P ^ i <= CONST power.power II op ;; P i
P ^ i <= (CONST power.power II op ;; P) i
Set up transfer tactic for powers

**Lemma upower-rep-eq:**

\[
\llbracket P \cdot i \rrbracket_e = (\lambda b. b \in \{\{p. \llbracket P \rrbracket_e p\} \cdot^i\})
\]

**Proof** (induct \(i\) arbitrary: \(P\))

- **Case 0**
  - then show \(?\)case
    - by (auto, rel-auto)

**Next**

- **Case (Suc \(i\))**
  - show \(?\)case
    - by (simp add: Suc seqr rep-eq relpow-commute upred-semiring power-Suc)

**QED**

**Lemma upower-rep-eq-alt:**

\[
\llbracket power \cdot power \cdot (id) \rrbracket_a (;;) P \cdot i \rrbracket_e = (\lambda b. b \in \{\{p. \llbracket P \rrbracket_e p\} \cdot^i\})
\]

by (metis skip-r-def upower-rep-eq)

**Update-uxpr-rep-eq-thms**

**Lemma Sup-power-expand:**

fixes \(P :\) nat \Rightarrow 'a::complete-lattice

shows \(P(0) \cap (\bigsqcup i. P(i+1)) = (\bigsqcup i. P(i))\)

**Proof**

- have \(UNIV = insert (0::nat) \{1..\}\)
  - by auto

moreover have \((\bigsqcup i. P(i)) = \bigsqcup (P \cdot UNIV)\)
  - by (blast)

moreover have \(\bigsqcup (P \cdot insert 0 \{1..\}) = P(0) \cap SUPREMUM \{1..\} P\)
  - by (simp)

moreover have \(SUPREMUM \{1..\} P = (\bigsqcup i. P(i+1))\)
  - by (simp add: atLeast-Suc-greaterThan greaterThan-0)

ultimately show \(?\)thesis
  - by (simp only:)

**QED**

**Lemma Sup-upto-Suc:**

\((\bigsqcup i\in\{0..Suc n\}. P \cdot i) = (\bigsqcup i\in\{0..n\}. P \cdot i) \cap P \cdot Suc n\)

**Proof**

- have \((\bigsqcup i\in\{0..Suc n\}. P \cdot i) = (\bigsqcup i\in insert (Suc n) \{0..n\}. P \cdot i)\)
  - by (simp add: atLeast0-atMost-Suc)

also have \(\ldots = P \cdot Suc n \cap (\bigsqcup i\in\{0..n\}. P \cdot i)\)
  - by (simp)

finally show \(?\)thesis
  - by (simp add: Lattices.sup-commute)

**QED**

The following two proofs are adapted from the AFP entry *Kleene Algebra*. See also [2, 1].

**Lemma upower-inductl:**

\(Q \subseteq ((P ;;; Q) \cap R) \Rightarrow Q \subseteq P \cdot n ;;; R\)

**Proof** (induct \(n\))

- **Case 0**
  - then show \(?\)case by (auto)

**Next**

- **Case (Suc \(n\))**
  - then show \(?\)case
    - by (auto simp add: upred-semiring.power-Suc, metis (no-types, hide-lams) dual-order.trans order-refl seqr-assoc seqr-mono)

120
lemma \textit{upower-inductr}:
\begin{itemize}
\item assumes \( Q \subseteq R \cap (Q :: P) \)
\item shows \( Q \subseteq R :: (P ^ n) \)
\end{itemize}
using \textit{assms} proof (induct \( n \))
\begin{itemize}
\item case \( 0 \)
\item then show \( \textit{?case} \) by \textit{auto}
\end{itemize}
next
\begin{itemize}
\item case \( (\text{Suc} \ n) \)
\item have \( R :: P ^ \text{Suc} \ n = (R :: P ^ n) :: P \)
\item by (metis \textit{seqr-assoc upred-semiring.power-Suc2})
\item also have \( Q :: P \subseteq \ldots \)
\item by (meson \textit{Suc.hyps assms eq-iff seqr-mono})
\item also have \( Q \subseteq \ldots \)
\item using \textit{assms} by \textit{auto}
\item finally show \( \textit{?case} \).
\end{itemize}
qed

lemma \textit{SUP-atLeastAtMost-first}:
\begin{itemize}
\item fixes \( P :: \text{nat} \Rightarrow 'a::complete-lattice \)
\item assumes \( m \leq n \)
\item shows \( \{ i \in \{m..n\} . P(i) \} = P(m) \cap (\prod i \in \{Suc m..n\}. P(i)) \)
\end{itemize}
by (metis \textit{SUP-insert assms atLeastAtMost-insertL})

lemma \textit{upower-seqr-iter}:
\begin{align*}
P ^ n &= (:: \text{replicate n P} \cdot Q) \\
\end{align*}
by (induct \( n \), simp-all add: \textit{upred-semiring.power-Suc})

lemma \textit{assigns-power}:
\( \langle f \rangle ^ a ^ \text{``} n = \langle f ^ \text{``} n \rangle ^ a \)
by (induct \( n \), rel-auto+)

18.14 Kleene Star

\textit{definition} \textit{ustar} :: \( 'a \ \text{hrel} \Rightarrow 'a \ \text{hrel} \text{ (\textit{''} [999] 999)} \) \textit{where}
\( P ^ * = (\prod i \in \{0..\} \cdot P ^ i) \)

\textit{lemma} \textit{ustar-rep-eq}:
\begin{align*}
[P ^ *] _ a &= (\lambda b . b \in \{ \langle p . [P] _ a p \rangle ^ * \})) \\
\end{align*}
by (simp add: \textit{ustar-def}, rel-auto, simp-all add: \textit{relpow-imp-rtrancl rtrancl-imp-relpow})

update-uexpr-rep-eq-thms

18.15 Kleene Plus

\textit{purge-notation} \textit{trancl} ((\textit{\textit{+}}) [1000] 999)

\textit{definition} \textit{uplus} :: \( 'a \ \text{hrel} \Rightarrow 'a \ \text{hrel} \text{ (\textit{''} [999] 999)} \) \textit{where}
\[ P ^ + = (\prod i \cdot P ^ \text{Suc i}) \]

\textit{lemma} \textit{uplus-power-def}:
\begin{align*}
P ^ + &= (\prod i \cdot P ^ \text{Suc i}) \\
\end{align*}
by (simp add: \textit{uplus-def ustar-def seq-UINF-distr' UINF-atLeast-Suc upred-semiring.power-Suc})

18.16 Omega

\textit{definition} \textit{uomega} :: \( 'a \ \text{hrel} \Rightarrow 'a \ \text{hrel} \text{ (\textit{''} [999] 999)} \) \textit{where}
\( P ^ \omega = (\mu X . P :: X) \)
18.17 Relation Algebra Laws

**Theorem RA1:** \((P :: (Q :: R)) = ((P :: Q) :: R)\)
by (simp add: seqr-assoc)

**Theorem RA2:** \((P :: II) = P (II :: P) = P\)
by simp-all

**Theorem RA3:** \(P^{--} = P\)
by simp

**Theorem RA4:** \((P :: Q)^-- = (Q^-- :: P^--)\)
by simp

**Theorem RA5:** \((P \lor Q)^-- = (P^-- \lor Q^--)\)
by (rel-auto)

**Theorem RA6:** \(((P \lor Q) :: R) = (P :: R \lor Q :: R)\)
using seqr-or-distl by blast

18.18 Kleene Algebra Laws

**Lemma ustar-alt-def:** \(P^* = (\prod i \cdot P^* i)\)
by (simp add: ustar-def)

**Theorem ustar-sub-unfoldl:** \(P^* \sqsubseteq II \sqcap (P :: P^*)\)
by (rel-simp, simp add: rtrancl-into-trancl2 trancl-into-rtrancl)

**Theorem ustar-inductl:**
- assumes \(Q \sqsubseteq R Q \sqsubseteq P :: Q\)
- shows \(Q \sqsubseteq P^* :: R\)
proof –
- have \(P^* :: R = (\prod i \cdot P^* i :: R)\)
  by (simp add: ustar-def UINF-as-Sup-collect seq-SUP-distr)
- also have \(Q \sqsubseteq ...\)
  by (simp add: SUP-least assms upower-inductl)
finally show \(?thesis\).
qed

**Theorem ustar-inductr:**
- assumes \(Q \sqsubseteq R Q \sqsubseteq Q :: P^*\)
- shows \(Q \sqsubseteq R :: P^*\)
proof –
- have \(R :: P^* = (\prod i \cdot R :: P^* i)\)
  by (simp add: ustar-def UINF-as-Sup-collect seq-SUP-distl)
- also have \(Q \sqsubseteq ...\)
  by (simp add: SUP-least assms upower-inductr)
finally show \(?thesis\).
qed

**Lemma ustar-refines-nu:** \((\nu X \cdot (P :: X) \sqcap II) \sqsubseteq P^*\)
by (metis (no-types, lifting) lfp-greatest semilattice-sup-class.le-sup-iff
semilattice-sup-class.idem upred-semiring.mult-2-right)
lemma ustar-as-nu: \( P^* = (\nu X \cdot (P ;; X) \cap II) \)

proof (rule antisym)

\[
\begin{align*}
&\text{show} \ (\nu X \cdot (P ;; X) \cap II) \subseteq P^* \\
&\quad \text{by (simp add: ustar-refines-nu)} \\
&\text{show} \ P^* \subseteq (\nu X \cdot (P ;; X) \cap II) \\
&\quad \text{by (metis lfp-lowerbound upred-semiring.add-commute ustar-sub-unfoldl)}
\end{align*}
\]

qed

lemma ustar-unfoldl: \( P^* = II \cap (P ;; P^*) \)

apply (simp add: ustar-as-nu)

apply (subst lfp-unfold)

apply (rule monoI)

apply (rel-auto)+

done

While loop can be expressed using Kleene star

lemma while-star-form:
\[
\text{while } b \text{ do } P \od = (P \triangleleft b \triangleright_r II)^* ;; [\neg b]^T
\]

proof

have 1: Continuous \( (\lambda X. \ P ;; X \triangleleft b \triangleright_r II) \)

by (rel-auto)

have while b do P od = \( (\prod i. (\lambda X. \ P ;; X \triangleleft b \triangleright_r II)^*) \) false

by (simp add: 1 false-upred-def sup-continuous-Continuous sup-continuous-lfp while-top-def)

also have \( \cdots = (\prod i. (\lambda X. \ P ;; X \triangleleft b \triangleright_r II)^* \) (false \( \cap \prod i. (\lambda X. \ P ;; X \triangleleft b \triangleright_r II)^* ) \) false)

by (subst Sup-power-expand, simp)

also have \( \cdots = (\prod i. (\lambda X. \ P ;; X \triangleleft b \triangleright_r II)^* (i+1) \) false)

by (simp)

also have \( \cdots = (\prod i. (P \triangleleft b \triangleright_r II)^* i ;; (\text{false} \triangleleft b \triangleright_r II)) \)

proof (rule SUP-cong, simp-all)

fix i

show \( P ;; (\lambda X. \ P ;; X \triangleleft b \triangleright_r II)^* i ;; (\text{false} \triangleleft b \triangleright_r II) = (P \triangleleft b \triangleright_r II)^* i ;; (\text{false} \triangleleft b \triangleright_r II) \)

proof (induct i)

case 0

then show \( \text{case by simp} \)

next

case (Suc i)

then show \( \text{case} \)

by (simp add: upred-semiring.power-Suc)

(metis (no-types, lifting) RA1 comp-cond-left-distr cond-L6 upred-semiring.mult.left-neutral)

qed

\[ \]

\[
\text{also have} \ \cdots = (\prod i \in \{0..\} \cdot (P \triangleleft b \triangleright_r II)^* i ;; [\neg b]^{T})
\]

by (rel-auto)

\[
\text{also have} \ \cdots = (P \triangleleft b \triangleright_r II)^* ;; [\neg b]^T
\]

by (metis seq-UINF-distr ustar-def)

finally show \( \text{thesis} \).

qed

18.19 Omega Algebra Laws

lemma uomega-induct:
\( P ;; P^\omega \subseteq P^\omega \)

by (simp add: uomega-def, metis eq-refl gfp-unfold monoI seqr-mono)
18.20 Refinement Laws

lemma skip-r-refine:
\((p \Rightarrow p) \subseteq H\)
by pred-blast

lemma conj-refine-left:
\((Q \Rightarrow P) \subseteq R \Longrightarrow P \subseteq (Q \land R)\)
by (rel-auto)

lemma pre-weak-rel:
assumes 'Pre ⇒ I'
and \((I ⇒ Post) \subseteq P\)
shows \((Pre ⇒ Post) \subseteq P\)
using assms by (rel-auto)

lemma cond-refine-rel:
assumes \(S \subseteq ([\neg b]_< \land Q)\)
shows \(S \subseteq P \triangleleft b \triangleright, Q\)
by (metis actx-not assms(1) assms(2) cond-def lift-rcond-def utp-pred-laws.le-sup-iff)

lemma seq-refine-pred:
assumes \([b]_< \Rightarrow [s]_> \subseteq P\) and \(([s]_< \Rightarrow [c]_>) \subseteq Q\)
shows \([b]_< \Rightarrow [c]_>) \subseteq (P ;; Q)\)
using assms by rel-auto

lemma seq-refine-unrest:
assumes outa \notin b inoa \notin c
assumes \((b \Rightarrow [s]_>) \subseteq P\) and \(([s]_< \Rightarrow c) \subseteq Q\)
shows \((b \Rightarrow c) \subseteq (P ;; Q)\)
using assms by rel-blast

18.21 Domain and Range Laws

lemma Dom-conv-Ran:
\(Dom(P^\top) = Ran(P)\)
by (rel-auto)

lemma Ran-conv-Dom:
\(Ran(P^\top) = Dom(P)\)
by (rel-auto)

lemma Dom-skip:
\(Dom(\bot) = true\)
by (rel-auto)

lemma Dom-assigns:
\(Dom(\langle \sigma \rangle_a) = true\)
by (rel-auto)

lemma Dom-miracle:
\(Dom(false) = false\)
by (rel-auto)

lemma Dom-assume:
\(Dom([b]^\top) = b\)
by (rel-auto)

lemma Dom-seq:
\[ \text{Dom}(P ; Q) = \text{Dom}(P ; \text{Dom}(Q)^\top) \]
by (rel-auto)

lemma Dom-disj:
\[ \text{Dom}(P \lor Q) = (\text{Dom}(P) \lor \text{Dom}(Q)) \]
by (rel-auto)

lemma Dom-inf:
\[ \text{Dom}(P \land Q) = (\text{Dom}(P) \lor \text{Dom}(Q)) \]
by (rel-auto)

lemma assume-Dom:
\[ [\text{Dom}(P)]^\top :: P = P \]
by (rel-auto)

end

19 UTP Theories

theory utp-theory
imports utp-rel-laws
begin

Here, we mechanise a representation of UTP theories using locales [4]. We also link them to the HOL-Algebra library [5], which allows us to import properties from complete lattices and Galois connections.

19.1 Complete lattice of predicates

definition upred-lattice :: (α upred) gorder (P) where
upred-lattice = (carrier = UNIV, eq = (=), le = (⊆) )

\( P \) is the complete lattice of alphabetised predicates. All other theories will be defined relative to it.

interpretation upred-lattice: complete-lattice \( P \)
proof (unfold-locales, simp-all add: upred-lattice-def)
fix A :: 'α upred set
show \( \exists s. \text{is-lub} (\lambda \text{carrier} = \text{UNIV}, eq = (=), le = (⊆)) s A \) using
apply (rule-tac x=\( A \) in exI)
apply (rule least-UpperI)
apply (auto intro: Inf-greatest simp add: Inf-lower Upper-def)
done
show \( \exists i. \text{is-glb} (\lambda \text{carrier} = \text{UNIV}, eq = (=), le = (⊆)) i A \) using
apply (rule-tac x=\( A \) in exI)
apply (rule greatest-LowerI)
apply (auto intro: Sup-least simp add: Sup-upper Lower-def)
done
qed

lemma upred-weak-complete-lattice [simp]: weak-complete-lattice \( P \)
by (simp add: upred-lattice.weak.weak-complete-lattice-axioms)
lemma upred-lattice-eq [simp]:
(= P) = (=)
by (simp add: upred-lattice-def)

lemma upred-lattice-le [simp]:
le P Q = (P ⊑ Q)
by (simp add: upred-lattice-def)

lemma upred-lattice-carrier [simp]:
carrier P = UNIV
by (simp add: upred-lattice-def)

lemma Healthy-fixed-points [simp]:
fps P H = [H]
by (simp add: fps-def upred-lattice-def Healthy-def)

lemma upred-lattice-Idempotent [simp]:
Idem P H = Idempotent H
using upred-lattice.weak-partial-order-axioms by (auto simp add: idempotent-def Idempotent-def)

lemma upred-lattice-Monotonic [simp]:
Mono P H = Monotonic H
using upred-lattice.weak-partial-order-axioms by (auto simp add: isotone-def mono-def)

19.2 UTP theories hierarchy

definition utp-order :: ('a × 'a) health ⇒ 'a hrel gorder where
utp-order H = (carrier = (P. P is H), eq = (=), le = (≤))

Constant utp-order obtains the order structure associated with a UTP theory. Its carrier is the set of healthy predicates, equality is HOL equality, and the order is refinement.

lemma utp-order-carrier [simp]:
carrier (utp-order T) = [H]
by (simp add: utp-order-def)

lemma utp-order-eq [simp]:
eq (utp-order T) = (=)
by (simp add: utp-order-def)

lemma utp-order-le [simp]:
le (utp-order T) = (≤)
by (simp add: utp-order-def)

lemma utp-partial-order: partial-order (utp-order T)
by (unfold-locales, simp-all add: utp-order-def)

lemma utp-weak-partial-order: weak-partial-order (utp-order T)
by (unfold-locales, simp-all add: utp-order-def)

lemma mono-Monotone-utp-order:
mono f ⇒ Monotone (utp-order T) f
apply (auto simp add: isotone-def)
apply (metis partial-order-def utp-partial-order)
apply (metis monoD)
done

lemma isotone-utp-orderI: Monotonic H ⇒ isotone (utp-order X) (utp-order Y) H
by (auto simp add: mono-def isotone-def utp-weak-partial-order)

lemma Mono-utp-orderI:
\[ \forall P Q. P \subseteq Q \land P \in H \land Q \in H \Rightarrow F(P) \subseteq F(Q) \] \Rightarrow \text{Mono}_{\text{utp-order}} H F
by (auto simp add: isotone-def utp-weak-partial-order)

The UTP order can equivalently be characterised as the fixed point lattice, \( \text{fpl} \).

lemma utp-order-fpl: utp-order H = \( \text{fpl} \setminus P \leq H \)
by (auto simp add: utp-order-def upred-lattice-def fps-def Healthy-def)

19.3 UTP theory hierarchy

We next define a hierarchy of locales that characterise different classes of UTP theory. Minimally we require that a UTP theory’s healthiness condition is idempotent.

locale utp-theory =
fixes hcond :: \( 'a \ hrel \Rightarrow 'a \ hrel \) (H)
assumes HCond-Idem: \( H(H(P)) = H(P) \)
begin
abbreviation thy-order :: \( 'a \ hrel \) gorder where
thy-order \equiv utp-order H

lemma HCond-Idempotent: \( \text{Idempotent } H \)
by (simp add: Idempotent-def HCond-Idem)

sublocale utp-po: partial-order utp-order H
by (unfold-locales, simp-all add: utp-order-def)

We need to remove some transitivity rules to stop them being applied in calculations

declare utp-po.trans [trans del]

eend

locale utp-theory-lattice = utp-theory +
assumes uthy-lattice: \( \text{complete-lattice } \) (utp-order H)
begin

sublocale complete-lattice utp-order H
by (simp add: uthy-lattice)

declare top-closed [simp del]
declare bottom-closed [simp del]

The healthiness conditions of a UTP theory lattice form a complete lattice, and allows us to make use of complete lattice results from HOL-Algebra [5], such as the Knaster-Tarski theorem. We can also retrieve lattice operators as below.

abbreviation utp-top (\( \top \))
where utp-top \equiv top (utp-order H)

abbreviation utp-bottom (\( \bot \))
where utp-bottom \equiv bottom (utp-order H)

abbreviation utp-join (\( \sqcup \)) where
utp-join \equiv join (utp-order H)
abbreviation utp-meet (infixl \( \cap \)) where
\( \text{utp-meet} \equiv \text{meet} (\text{utp-order } H) \)

abbreviation utp-sup (\([\pmb{-} [90] 90]) where
\( \text{utp-sup} \equiv \text{Lattice.sup } (\text{utp-order } H) \)

abbreviation utp-inf (\([\pmb{-} [90] 90]) where
\( \text{utp-inf} \equiv \text{Lattice.inf } (\text{utp-order } H) \)

abbreviation utp-gfp (\(\nu\)) where
\( \text{utp-gfp} \equiv \text{GREATEST-FP } (\text{utp-order } H) \)

abbreviation utp-lfp (\(\mu\)) where
\( \text{utp-lfp} \equiv \text{LEAST-FP } (\text{utp-order } H) \)

end

syntax
-\(\text{tnu} :: \log \Rightarrow \text{pttrn} \Rightarrow \log \Rightarrow \log (\mu \cdot \cdot [0, 10])\)
-\(\text{tnu} :: \log \Rightarrow \text{pttrn} \Rightarrow \log \Rightarrow \log (\nu \cdot \cdot [0, 10])\)

notation \(\text{gfp } (\mu)\)
notation \(\text{lfp } (\nu)\)

translations
\(\mu_H X \cdot P \equiv \text{CONST LEAST-FP } (\text{CONST utp-order } H) (\lambda X. P)\)
\(\nu_H X \cdot P \equiv \text{CONST GREATEST-FP } (\text{CONST utp-order } H) (\lambda X. P)\)

lemma upred-lattice-inf:
\(\text{Lattice.inf } P A = \bigcap A\)
by (metis Sup-least Sup-upper UNIV-I antisym-cone subsetI upred-lattice.weak.inf-greatest upred-lattice.weak.inf-lower upred-lattice-carrier upred-lattice-le)

We can then derive a number of properties about these operators, as below.

context utp-theory-lattice
begin

lemma LFP-healthy-comp: \(\mu F = \mu (F \circ H)\)
proof
  have \(\{P. (P \text{ is } H) \land F P \subseteq P\} = \{P. (P \text{ is } H) \land F (H P) \subseteq P\}\)
  by (auto simp add: Healthy-def)
  thus \(?\)thesis
  by (simp add: LEAST-FP-def)
qed

lemma GFP-healthy-comp: \(\nu F = \nu (F \circ H)\)
proof
  have \(\{P. (P \text{ is } H) \land F P \subseteq P\} = \{P. (P \text{ is } H) \land P \subseteq F (H P)\}\)
  by (auto simp add: Healthy-def)
  thus \(?\)thesis
  by (simp add: GREATEST-FP-def)
qed

lemma top-healthy [closure]: \(\top \text{ is } H\)

128
using weak.top-closed by auto

lemma bottom-healthy [closure]: ⊥ is H
using weak.bottom-closed by auto

lemma utp-top: P is H ⇒ P ⊆ T
using weak.top-higher by auto

lemma utp-bottom: P is H ⇒ ⊥ ⊆ P
using weak.bottom-lower by auto

end

lemma upred-top: ⊤ P = false
using ball-UNIV greatest-def by fastforce

lemma upred-bottom: ⊥ P = true
by fastforce

One way of obtaining a complete lattice is showing that the healthiness conditions are monotone, which the below locale characterises.

locale utp-theory-mono = utp-theory +
assumes HCond-Mono [closure, intro]: Monotonic H

sublocale utp-theory-mono ⊆ utp-theory-lattice
proof –
interpret weak-complete-lattice fpl P H
by (rule Knaster-Tarski, auto)

have complete-lattice (fpl P H)
by (unfold-locales, simp add: fps-def sup-exists, (blast intro: sup-exists inf-exists)+)

hence complete-lattice (utp-order H)
by (simp add: utp-order-def, simp add: upred-lattice-def)

thus utp-theory-lattice H
by (simp add: utp-theory-axioms utp-theory-lattice.intro utp-theory-lattice-axioms.intro)
qed

In a monotone theory, the top and bottom can always be obtained by applying the healthiness condition to the predicate top and bottom, respectively.

context utp-theory-mono
begin

lemma healthy-top: T = H(false)
proof –
have T = T fpl P H
by (simp add: utp-order-fpl)
also have ... = H ⊤ P
using Knaster-Tarski-idem-extremes(1)[of P H]
by (simp add: HCond-Idempotent HCond-Mono)
also have ... = H false
by (simp add: upred-top)
finally show ?thesis .
qed
lemma healthy-bottom: \( \bot = \mathcal{H}(\text{true}) \)
proof
  have \( \bot = \bot_{fp} \mathcal{P} \mathcal{H} \)
    by (simp add: utp-order-fpl)
  also have \( \ldots = \mathcal{H} \bot_{\mathcal{P}} \)
    using Knaster-Tarski-idem-extremes(2)[of \( \mathcal{P} \) \( \mathcal{H} \)]
    by (simp add: HCond-Idempotent HCond-Mono)
  also have \( \ldots = \mathcal{H} \text{true} \)
    by (simp add: upred-bottom)
  finally show \( \text{thesis} \).
qed

lemma healthy-inf:
  assumes \( A \subseteq [\mathcal{H}]_\mathcal{H} \)
  shows \( \bigcap A = \mathcal{H} (\bigcap A) \)
  using Knaster-Tarski-idem-inf-eq[OF upred-weak-complete-lattice, of \( \mathcal{H} \)]
  by (simp, metis HCond-Idempotent HCond-Mono assms partial-object.simps(3) upred-lattice-def upred-lattice-inf utp-order-def)
end

locale utp-theory-continuous = utp-theory +
  assumes HCond-Cont[closure, intro]: Continuous \( \mathcal{H} \)
sublocale utp-theory-continuous \subseteq utp-theory-mono
proof
  show Monotonic \( \mathcal{H} \)
    by (simp add: Continuous-Monotonic HCond-Cont)
qed

context utp-theory-continuous
begin

lemma healthy-inf-cont:
  assumes \( A \subseteq [\mathcal{H}]_\mathcal{H} \) \( A \neq \{\} \)
  shows \( \bigcap A = \bigcap A \)
proof
  have \( \bigcap A = \bigcap (\mathcal{H} \mathcal{F} A) \)
    using Continuous-def HCond-Cont assms(1) assms(2) healthy-inf by auto
  also have \( \ldots = \bigcap A \)
    by (unfold Healthy-carrier-image[OF assms(1), simp])
  finally show \( \text{thesis} \).
qed

lemma healthy-inf-def:
  assumes \( A \subseteq [\mathcal{H}]_\mathcal{H} \)
  shows \( \bigcap A = (\text{if } (A = \{\}) \text{ then } \top \text{ else } (\bigcap A)) \)
  using assms healthy-inf-cont weak.weak-inf-empty by auto

lemma healthy-meet-cont:
  assumes \( P \text{ is } \mathcal{H} \) \( Q \text{ is } \mathcal{H} \)
  shows \( P \cap Q = P \cap Q \)
  using healthy-inf-cont[of \{\( P \), \( Q \)\} assms]
  by (simp add: Healthy-if meet-def)
lemma meet-is-healthy [closure]:
assumes $P$ is $H$
shows $P \cap Q$ is $H$
bymetis Continuous-Disjunctuous Disjunctuous-def HCond-Cont Healthy-def' assms(1) assms(2))

lemma meet-bottom [simp]:
assumes $P$ is $H$
shows $P \cap \bot = \bot$
bysimp add: assms semilattice-sup-class.sup-absorb2 utp-bottom

lemma meet-top [simp]:
assumes $P$ is $H$
shows $P \cap \top = P$
bysimp add: assms semilattice-sup-class.sup-absorb1 utp-top

The UTP theory lfp operator can be rewritten to the alphabetised predicate lfp when in a continuous context.

theorem utp-lfp-def:
assumes Monotonic $F$ $F \in [H]_H \to [H]_H$
shows $\mu F = (\mu X \cdot F(H(X)))$
proof (rule antisym)
  have ne: $\{ P. (P \in H) \land F P \sqsubseteq P \} \neq \{}$
  proof
    have $F \top \sqsubseteq \top$
      using assms(2) utp-top weak.top-closed by force
    thus $\?thesis$
      by (auto, rule-tac x$=\top$ in exI, auto simp add: top-healthy)
  qed
  show $\mu F \sqsubseteq (\mu X \cdot F(H X))$
  proof
    have $\{ P. (P \in H) \land F P \sqsubseteq P \} \subseteq \{ P. F(H(P)) \sqsubseteq P \}$
    proof
      have $1: \land P. F(H(P)) = H(F(H(P)))$
        by (metis HCond-Idem Healthy-def assms(2) funcset-mem mem-Collect-eq)
      show $\?thesis$
        proof (rule Sup-least, auto)
          fix $P$
          assume $a: F(H P) \sqsubseteq P$
          hence $F: (F(H P)) \sqsubseteq (H P)$
            by (metis 1 HCond-Mono mono-def)
          show $\bigcap \{ P. (P \in H) \land F P \sqsubseteq P \} \subseteq \{ P. F(H(P)) \sqsubseteq P \}$
            proof (rule Sup-upper2[of $F(H P)$])
              show $F(H P) \in \{ P. (P \in H) \land F P \sqsubseteq P \}$
                proof (auto)
                  show $F(H P)$ is $H$
                    by (metis 1 Healthy-def)
                  show $F(F(H P)) \sqsubseteq F(H P)$
                    using $F$ mono-def assms(1) by blast
                  qed
              show $F(H P) \subseteq P$
                by (simp add: $a$)
            qed
            qed
          qed
        qed
      qed
    qed
  qed
qed
with ne show ?thesis
  by (simp add: LEAST-FP-def gfp-def, subst healthy-inf-cont, auto simp add: lfp-def)
qed
from ne show (μ X · F (H X)) ⊆ μ F
  apply (simp add: LEAST-FP-def gfp-def, subst healthy-inf-cont, auto simp add: lfp-def)
  apply (rule Sup-least)
  apply (auto simp add: Healthy-def Sup-upper)
  done
qed

lemma UINF-ind-Healthy [closure]:
  assumes ∩ i. P(i) is H
  shows (∩ i · P(i)) is H
  by (simp add: closure assms)
end

In another direction, we can also characterise UTP theories that are relational. Minimally this requires that the healthiness condition is closed under sequential composition.

locale utp-theory-rel =
  utp-theory +
  assumes Healthy-Sequence [closure]: [ P is H; Q is H ] ⇒ (P ;; Q) is H
begin

lemma upower-Suc-Healthy [closure]:
  assumes P is H
  shows P ^ Suc n is H
  by (induct n, simp-all add: closure assms upred-semiring.power-Suc)
end

locale utp-theory-cont-rel = utp-theory-rel + utp-theory-continuous
begin

lemma seq-cont-Sup-distl:
  assumes P is H A ⊆ [H]H A ≠ {}
  shows P ;; (∩ A) = ∩ {P ;; Q | Q. Q ∈ A }
  proof
    have {P ;; Q | Q. Q ∈ A } ⊆ [H]H
      using Healthy-Sequence assms(1) assms(2) by (auto)
    thus ?thesis
    by (simp add: healthy-inf-cont seq-Sup-distl setcompr-eq-image assms)
  qed

lemma seq-cont-Sup-distr:
  assumes Q is H A ⊆ [H]H A ≠ {}
  shows (∩ A) ;; Q = ∩ {P ;; Q | P. P ∈ A }
  proof
    have {P ;; Q | P. P ∈ A } ⊆ [H]H
      using Healthy-Sequence assms(1) assms(2) by (auto)
    thus ?thesis
    by (simp add: healthy-inf-cont seq-Sup-distr setcompr-eq-image assms)
  qed
There also exist UTP theories with units. Not all theories have both a left and a right unit (e.g. H1-H2 designs) and so we split up the locale into two cases.

**locale utp-theory-units =**

**utp-theory-rel +**

**fixes utp-unit (\(\mathbb{I}\))**

**assumes Healthy-Unit [closure]: \(\mathbb{I}\) is \(\mathcal{H}\)**

**begin**

We can characterise the theory Kleene star by lifting the relational one.

**definition utp-star (\(-\star\) [999] 999) where**

**[upred-defs]: utp-star P = (P\(^\star\) ;; \(\mathbb{I}\))**

We can then characterise tests as refinements of units.

**definition utp-test :: 'a hrel ⇒ bool where**

**[upred-defs]: utp-test b = (\(\mathbb{I}\)\(\subseteq\) b)**

**end**

**locale utp-theory-left-unital =**

**utp-theory-units +**

**assumes Unit-Left: P is \(\mathcal{H}\) \implies (\(\mathbb{I}\) ;; P) = P**

**locale utp-theory-right-unital =**

**utp-theory-units +**

**assumes Unit-Right: P is \(\mathcal{H}\) \implies (P ;; \(\mathbb{I}\)) = P**

**locale utp-theory-unital =**

**utp-theory-left-unital + utp-theory-right-unital**

**begin**

**lemma Unit-self [simp]:**

\(\mathbb{I}\) ;; \(\mathbb{I}\) = \(\mathbb{I}\)

**by (simp add: Healthy-Unit Unit-Right)**

**lemma utest-intro:**

\(\mathbb{I}\)\(\subseteq\) P \implies utp-test P

**by (simp add: utp-test-def)**

**lemma utest-Unit [closure]:**

utp-test \(\mathbb{I}\)

**by (simp add: utp-test-def)**

**end**

**locale utp-theory-mono-unital = utp-theory-unital + utp-theory-mono**

**begin**

**lemma utest-Top [closure]: utp-test \(\top\)**
by (simp add: Healthy-Unit utp-test-def utp-top)

end

locale utp-theory-cont-unital = utp-theory-cont-rel + utp-theory-unital

sublocale utp-theory-cont-unital ⊆ utp-theory-mono-unital
  by (simp add: utp-theory-mono-axioms utp-theory-mono-unital-def utp-theory-unital-axioms)

locale utp-theory-unital-zerol =
  utp-theory-unital + utp-theory-lattice +
  assumes Top-Left-Zero: P is H ⇒ ⊤ ; P = ⊤

locale utp-theory-cont-unital-zerol =
  utp-theory-cont-unital + utp-theory-unital-zerol

begin

lemma Top-test-Right-Zero:
  assumes b is H utp-test b
  shows b ; ⊤ = ⊤
proof
  have b ∩ II = II
    by (meson assms(2) semilattice-sup-class.le-iff-sup utp-test-def)
  then show ?thesis
    by (metis (no-types) Top-Left-Zero Unit-Left assms(1) meet-top top-healthy upred-semiring.distrib-right)
qed

end

19.4 Theory of relations

interpretation rel-theory: utp-theory-mono-unital id skip-r
rewrites rel-theory.utp-top = false
and rel-theory.utp-bottom = true
and carrier (utp-order id) = UNIV
and (P is id) = True
proof
  show utp-theory-mono-unital id II
    by (unfold-locales, simp-all add: Healthy-def)
  then interpret utp-theory-mono-unital id skip-r
    by simp
  show utp-top = false utp-bottom = true
    by (simp-all add: healthy-top healthy-bottom)
  show carrier (utp-order id) = UNIV (P is id) = True
    by (auto simp add: utp-order-def Healthy-def)
qed

thm rel-theory.GFP-unfold

19.5 Theory links

We can also describe links between theories, such a Galois connections and retractions, using the following notation.

definition mk-conn (- ⇔(,·)⇒ : [90,0,0,91] 91) where
$H_1 \leftarrow (H_1, H_2) \Rightarrow H_2 \equiv (| \text{order}_A = \text{utp-order } H_1, \text{order}_B = \text{utp-order } H_2, \text{lower } = H_2, \text{upper } = H_1 |)$

**Lemma** $\text{mk-conn-orderA} [\text{simp}]: X \leftarrow (H_1, H_2) \Rightarrow H_2 = \text{utp-order } H_1$
  by (simp add: $\text{mk-conn-def}$)

**Lemma** $\text{mk-conn-orderB} [\text{simp}]: Y \leftarrow (H_1, H_2) \Rightarrow H_2 = \text{utp-order } H_2$
  by (simp add: $\text{mk-conn-def}$)

**Lemma** $\text{mk-conn-lower} [\text{simp}]: \pi_\star H_1 \leftarrow (H_1, H_2) \Rightarrow H_2 = H_1$
  by (simp add: $\text{mk-conn-def}$)

**Lemma** $\text{mk-conn-upper} [\text{simp}]: \pi^* H_1 \leftarrow (H_1, H_2) \Rightarrow H_2 = H_2$
  by (simp add: $\text{mk-conn-def}$)

**Lemma** $\text{galois-comp} : (H_2 \leftarrow (H_3, H_4) \Rightarrow H_3) \circ g (H_1 \leftarrow (H_1, H_2) \Rightarrow H_2) = (H_1 \leftarrow (H_1 \circ H_3, H_4 \circ H_2) \Rightarrow H_3$
  by (simp add: $\text{comp-galcon-def } \text{mk-conn-def}$)

Example Galois connection / retract: Existential quantification

**Lemma** $\text{Idempotent-ex} : \text{mwb-lens } x = \Rightarrow \text{Idempotent } (\text{ex } x)$
  by (simp add: $\text{Idempotent-def } \text{exists-twice}$)

**Lemma** $\text{Monotonic-ex} : \text{mwb-lens } x = \Rightarrow \text{Monotonic } (\text{ex } x)$
  by (simp add: $\text{mono-def } \text{ex-mono}$)

**Lemma** $\text{ex-closed-unrest} : \text{vwb-lens } x = \Rightarrow [ [ \text{ex } x ] ] H = \{ P. x \neq P \}$
  by (simp add: $\text{Healthy-def } \text{unrest-as-exists}$)

Any theory can be composed with an existential quantification to produce a Galois connection

**Theorem** $\text{ex-retract} :$
  **assumes** $\text{vwb-lens } x \text{ Idempotent } H \text{ ex } x \circ H = H \circ \text{ex } x$
  **shows** $\text{retract } ((\text{ex } x \circ H) \leftarrow (\text{ex } x, H) \Rightarrow H)$
  **proof** (unfold-locales, simp-all)
    **show** $H \in [ [ \text{ex } x \circ H ] ] H \Rightarrow [ H ] H$
      using $\text{Healthy-Idempotent } \text{assms}$ by blast
    **from** $\text{assms}(1)$ $\text{assms}(3)$ THEN sym show $\text{ex } x \in [ H ] H \Rightarrow [ [ \text{ex } x \circ H ] ] H$
      by (simp add: $\text{Pi-iff } \text{Healthy-def } \text{fun-eq-iff exists-twice}$)
    **fix** $P Q$
    **assume** $P$ is $[ [ \text{ex } x \circ H ] ] Q$ is $H$
    **thus** $[H P \subseteq Q] = (P \subseteq (\exists x. Q))$
      by (metis $\text{no-types, lifting } \text{Healthy-Idempotent } \text{Healthy-if } \text{assms } \text{comp-apply } \text{dual-order.trans } \text{ex-weakens}$ $\text{utp-pred-laws.ex-mon } \text{mwb-lens-ub}$)
  **next**
    **fix** $P$
    **assume** $P$ is $[ \text{ex } x \circ H ]$
    **thus** $(\exists x. H P) \subseteq P$
      by (simp add: $\text{Healthy-def}$)
  qed

**Corollary** $\text{ex-retract-id} :$
  **assumes** $\text{vwb-lens } x$
  **shows** $\text{retract } ((\text{ex } x \leftarrow (\text{ex } x, id) \Rightarrow id)$
  **using** $\text{assms } \text{ex-retract}[\text{where } H=id]$ by (auto)
end
20 Relational Hoare calculus

theory utp-hoare
  imports
    utp-rel-laws
    utp-theory
begin

20.1 Hoare Triple Definitions and Tactics

definition hoare-r :: 'α cond ⇒ 'α hrel ⇒ 'α cond ⇒ bool (⦃⦃⦅⦄⦄⦅⦄⦆⦅⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆⦆يئة
20.3 Assignment Laws

lemma assigns-hoare-r [hoare-safe]: ‘p ⇒ σ ⌜ q’ ⇒ {p}⟨σ⟩u {q}u
   by rel-auto

lemma assigns-backward-hoare-r:
   \[\{σ \uparrow p\}⟨σ⟩u {p}u\]
   by rel-auto

lemma assign-floyd-hoare-r:
   assumes \(\text{vwb-lens } x\)
   shows \(\{\text{assign } x e\} Q\)
   using assms
   rel-auto
   by (rel-auto, metis vwb-lens-wb wb-lens.get-put)

lemma skip-hoare-r [hoare-safe]: \{p\} II {p}u
   by rel-auto

lemma skip-hoare-impl-r [hoare-safe]: ‘p ⇒ q’ ⇒ \{p\} II {q}u
   by rel-auto

20.4 Sequence Laws

lemma seq-hoare-r: \[\{p\} Q_1 \{s\}u ; \{s\} Q_2 \{r\}u \]\n   ⇒ \{p\} Q_1 ; Q_2 \{r\}u
   by rel-auto

lemma seq-hoare-invariant [hoare-safe]: \[\{p\} Q_1 \{p\}u ; \{p\} Q_2 \{p\}u \]\n   ⇒ \{p\} Q_1 ; Q_2 \{p\}u
   by rel-auto

lemma seq-hoare-stronger-pre-1 [hoare-safe]:
   \[\{p \land q\} Q_1 \{p \land q\}u ; \{p \land q\} Q_2 \{q\}u \]\n   ⇒ \{p \land q\} Q_1 ; Q_2 \{q\}u
   by rel-auto

lemma seq-hoare-stronger-pre-2 [hoare-safe]:
   \[\{p \land q\} Q_1 \{p \land q\}u ; \{p \land q\} Q_2 \{p\}u \]\n   ⇒ \{p \land q\} Q_1 ; Q_2 \{p\}u
   by rel-auto

lemma seq-hoare-inv-r-2 [hoare]: \[\{p\} Q_1 \{q\}u ; \{q\} Q_2 \{q\}u \]\n   ⇒ \{p\} Q_1 ; Q_2 \{q\}u
   by rel-auto

lemma seq-hoare-inv-r-3 [hoare]: \[\{p\} Q_1 \{p\}u ; \{p\} Q_2 \{q\}u \]\n   ⇒ \{p\} Q_1 ; Q_2 \{q\}u
   by rel-auto

20.5 Conditional Laws

lemma cond-hoare-r [hoare-safe]: \[\{b \land p\} S \{q\}u ; \{\neg b \land p\} T \{q\}u \]\n   ⇒ \{p\} S a b ▷ r, T \{q\}u
   by rel-auto

lemma cond-hoare-r-wp:
   assumes \(\{p\} S \{q\}u\) and \(\{p’\} T \{q\}u\)
   shows \(\{b \land p\} \lor (\neg b \land p’)\) S a b ▷ r, T \{q\}u
   using assms by pred-simp
lemma cond-hoare-r-sp:
assumes $\langle \{ b \land p \}, S, \{ q \} \rangle_u$ and $\langle \neg b \land p \}, T, \{ q \} \rangle_u$
shows $\langle \{ p \}, S \triangleright b \triangleright_r T, \{ q \lor s \} \rangle_u$
using assms by pred-simp

20.6 Recursion Laws

lemma nu-hoare-r-partial:
assumes induct-step:
$\forall st. \langle \{ p \}, S, \{ q \} \rangle_u \Rightarrow \langle \{ p \}, F, \{ q \} \rangle_u$
shows $\langle \{ p \}, \nu F, \{ q \} \rangle_u \equiv \langle \{ p \}, S, \{ q \} \rangle_u$ using

unfolding hoare-r-def

proof (rule mu-rec-total-utp-rule [OF WF M, of - e], goal-cases)
  case (1 st)
  then show ?case using induct-step unfolded hoare-r-def, of $\langle [p], [e], [st] \rangle_u \in_R \Rightarrow [q] \rangle$ st
  by (simp add: alpha)

qed

lemma mu-hoare-r:
assumes WF: \(Q \)
assumes M: mono F
assumes induct-step:
$\forall st. \langle \{ p \}, S, \{ q \} \rangle_u \Rightarrow \langle \{ p \}, F, \{ q \} \rangle_u$
shows $\langle \{ p \}, \mu F, \{ q \} \rangle_u$

unfolding hoare-r-def

proof (rule mu-rec-total-utp-rule [OF WF M, of - e], goal-cases)
  case (1 st)
  then show ?case using induct-step unfolded hoare-r-def, of $\langle [p], [e], [st] \rangle_u \in_R \Rightarrow [q] \rangle$ st
  by (simp add: alpha)

qed

lemma mu-hoare-r':
assumes WF: \(Q \)
assumes M: mono F
assumes induct-step:
$\forall st. \langle \{ p \}, S, \{ q \} \rangle_u \Rightarrow \langle \{ p \}, F, \{ q \} \rangle_u$
shows $\langle \{ p \}, \mu F, \{ q \} \rangle_u$

by (meson I0 M WF induct-step mu-hoare-r)

20.7 Iteration Rules

lemma iter-hoare-r [hoare-safe]:
assumes $\{ p \} S \{ p \} \Rightarrow \{ p \} S \{ \} u$
by (rel-simp', metis (mono-tags, lifting) mem-Collect-eq rtrancl-induct)

lemma while-hoare-r [hoare-safe]:
assumes $\{ p \} \triangleright b \triangleright_r S, \{ p \} u$
shows $\{ p \} \triangleright b \triangleright_r \{ \neg b \land p \} \rangle_u$
using assms by (simp add: while-top-def hoare-r-def, rule-tac lfp-lowerbound)

lemma while-inv-hoare-r [hoare-safe]:
assumes $\{ p \} \triangleright b \triangleright_r S, \{ p \} u$
shows $\{ p \} \triangleright b \triangleright_r \{ \neg b \land p \} \rangle_u$
by (metis assms hoare-r-conseq while-hoare-r while-inv-def)

lemma while-r-minimal-partial:
assumes seq-step: $p \Rightarrow \triangleright_r c$
assumes induct-step: $\{ c \} \triangleright_r b \triangleright_r C, \{ c \} u$

138
shows $\{p\}$ while $b$ do $C$ od $\{\neg b \land \text{invar}\}$

using induct-step pre-str-hoare-r seq-step while-hoare-r by blast

lemma approx-chain:
$\prod n::\text{nat. } \{p \land v <u}<n\} < \{p\} <$
by (rel-auto)

Total correctness law for Hoare logic, based on constructive chains. This is limited to variants that have naturals numbers as their range.

lemma while-term-hoare-r:
assumes $\forall z::\text{nat. } \{p \land b \land v =u <z\} S \{p \land v <u <z\}$
shows $\{p\}$ while $\bot$ $b$ do $S$ od $\{\neg b \land p\}$
proof
have $\{p\} \Rightarrow (\forall i. \{p \land v <u <\text{Suc } i\} \Rightarrow \{p\} \land \forall i. \{p \land v <u <i\} <$
by (rel-auto)

show $\prod n. \forall E n = \{p\}$
by (rel-auto)

show mono $(\lambda X. S :: X <b \triangleright_r II)$
by (simp add: cond-mono monoI seqr-mono)

show constr $(\lambda X. S :: X <b \triangleright_r II)$ ?E
proof (rule constrI)

show chain ?E
proof (rule chainI)

show $\forall i. \{p \land v <u <\text{Suc } i\} < \forall i. \{p \land v <u <i\} <$
by (rel-auto)

qed

from assms
show $\prod X n. (S :: X <b \triangleright_r II \land \{p \land v <u <n + 1\}) = (S :: (X \land \{p \land v <u <n\} <b \triangleright_r II \land \{p \land v <u <n + 1\} <$
apply (rel-auto)
using less-antisym less-trans apply blast

qed

thus $\forall E$
by (simp add: hoare-r-def while-bot-def)

qed

lemma while-wrt-hoare-r [hoare-safe]:
assumes $\bigwedge z::\text{nat}. \{ p \land b \land v =_u \langle z \rangle \, S \{ p \land v <_u \langle z \rangle \} \}_u \ 'pre \Rightarrow p' \ '(\neg b \land p) \Rightarrow post'$
shows $\{ \text{pre} \}_u \text{while invr v rt v do S od} \{ \text{post} \}_u$ 
apply (rule hoare-r-conseq[OF assms(2) - assms(3)]) 
apply (simp add: while-ert-def) 
apply (rule while-term-hoare-r[where v=v, OF assms(1)]) 
done

General total correctness law based on well-founded induction

**lemma while-wf-hoare-r:**

assumes $\text{WF: wf } R$
assumes $\text{I0: 'pre } \Rightarrow p'$
assumes $\text{induct-step: } \forall \, st. \{ b \land p \land e =_u \langle st \rangle \} \{ p \land (e, \langle st \rangle) \}_u \in_u \{ R \}_u$
assumes $\text{PHI: '}(\neg b \land p) \Rightarrow p'$
shows $\{ \text{pre} \}_u \text{while } b \text{ invr } p \text{ do } Q \od \{ \text{post} \}_u$

unfolding hoare-r-def while-inv-bot-def while-bot-def

**proof** (rule pre-weak-rel[of - \{p\}_<])
from $\text{I0}$ show $\langle [p]_< \Rightarrow [p]_< \rangle$
  by (rel-auto)
show $([p]_< \Rightarrow [\text{post}]_>) \in (\mu X. Q ; X < b \triangleright_r \text{ II})$
  by (rule mu-rec-total-utp-rule[where $e=e$, OF $WF$])
proof (rule mono-rec-total-utp-rule)
  show $\text{Monotonic (XQ: X < b \triangleright_r \text{ II})}$
    by (simp add: closure)
  have $\text{induct-step: } \forall \, st. \{ b \land p \land e =_u \langle st \rangle \} \Rightarrow ([p \land (e, \langle st \rangle) \}_u \in_u \{ R \}_u ) \}
    \in Q$
  using $\text{induct-step by rel-auto}$
with $\text{PHI}$
show $\forall \, st. \{ [p]_< \land [e]_< =_u \langle st \rangle \Rightarrow [\text{post}]_> \} \in Q ; ; ([p]_< \land ([e]_<, \langle st \rangle)_u \in_u \{ R \}_u \Rightarrow [\text{post}]_>)$
\begin{itemize}
  \item $b \triangleright_r \text{ II}$
  \item \text{ by (rel-auto)}
\end{itemize}
qed

**20.8 Frame Rules**

Frame rule: If starting $S$ in a state satisfying $\text{pestablishesq}$ in the final state, then we can insert an invariant predicate $r$ when $S$ is framed by $a$, provided that $r$ does not refer to variables in the frame, and $q$ does not refer to variables outside the frame.

**lemma frame-hoare-r:**

assumes $\text{vwb-lens } a \ a \ r \ a \ q \ \{ p \}_a P \{ q \}_u$
shows $\{ p \land r \} a : \{ P \}_u \{ q \land r \}_u$
using assms
by (rel-auto, metis)

**lemma frame-strong-hoare-r** [hoare-safe]:

assumes $\text{vwb-lens } a \ a \ r \ a \ q \ \{ p \land r \}_a S \{ q \}_u$
shows $\{ p \land r \} a : \{ S \}_u \{ q \}_u \{ r \land q \}_a$
using assms by (rel-auto, metis)

**lemma frame-hoare-r'** [hoare-safe]:

assumes $\text{vwb-lens } a \ a \ r \ a \ q \ \{ r \}_a S \{ q \}_u$
shows $\{ r \land p \} a : \{ S \}_u \{ r \land q \}_u$
using assms 
by (simp add: frame-strong-hoare-r utp-pred-laws.inf.commute)

**lemma antiframe-hoare-r:**

assumes $\text{vwb-lens } a \ a \ r \ a \ q \ \{ p \}_a P \{ q \}_u$
shows \{p \land r\} a : [P] \{q \land r\} u
using assms by (rel-auto, metis)

lemma antiframe-strong-hoare-r:
assumes wwb-lens a a \notin r a \notin q \{p \land r\} P\{q\} u
shows \{p \land r\} a : [P] \{q \land r\} u
using assms by (rel-auto, metis)
end

21 Weakest (Liberal) Precondition Calculus

theory utp-wp
imports utp-hoare
begin
A very quick implementation of wlp – more laws still needed!
named-theorems wp

method wp-tac = (simp add: wp)
consts
uwp :: 'a \Rightarrow 'b \Rightarrow 'c
syntax
-uwp :: logic \Rightarrow uexp \Rightarrow logic (infix wp 60)
translations
-uwp P b == CONST uwp P b

definition wp-upred :: ('\alpha, '\beta) urel \Rightarrow '\alpha cond \Rightarrow '\alpha cond where
wp-upred Q r = \lfloor \neg (Q ;; \neg [r]<) \rfloor :: ('\alpha, '\beta) urel<
adhoc-overloading
uwp wp-upred
declare wp-upred-def [urel-defs]

lemma wp-true [wp]: p wp true = true
by (rel-simp)

theorem wp-assigns-r [wp]:
\langle \sigma \rangle_a wp r = \sigma \uparrow r
by rel-auto

theorem wp-skip-r [wp]:
II wp r = r
by rel-auto

theorem wp-abort [wp]:
r \neq true \implies true wp r = false
by rel-auto

theorem wp-conj [wp]:
P wp (q \land r) = (P wp q \land P wp r)
by rel-auto

**Theorem wp-seq-r**: \( (P \cdot Q) \text{ wp } r = P \text{ wp } (Q \text{ wp } r) \)
by rel-auto

**Theorem wp-choice**: \( (P \sqcap Q) \text{ wp } R = (P \text{ wp } R \land Q \text{ wp } R) \)
by (rel-auto)

**Theorem wp-cond**: \( (P \triangleleft b \triangleright_r Q) \text{ wp } r = ((b \Rightarrow P \text{ wp } r) \land ((\neg b) \Rightarrow Q \text{ wp } r)) \)
by rel-auto

**Lemma wp-USUP-pre**: \( P \text{ wp } (\bigsqcup_{i \in \{0..n\}} Q(i)) = (\bigsqcup_{i \in \{0..n\}} P \text{ wp } Q(i)) \)
by (rel-auto)

**Theorem wp-hoare-link**: 
\[ \{p|Q\} r \longleftrightarrow (Q \text{ wp } r \sqsubseteq p) \]
by rel-auto

If two programs have the same weakest precondition for any postcondition then the programs are the same.

**Theorem wp-eq-intro**: \[ \{ r. P \text{ wp } r = Q \text{ wp } r \} \implies P = Q \]
by (rel-auto robust, fastforce+)

end

22 Dynamic Logic

theory utp-dynlog
  imports utp-sequent utp-wp
begin

22.1 Definitions

**Named-Theorems** dynlog-simp and dynlog-intro

**Definition dBox** :: 's hrel \( \Rightarrow \) 's upred \( \Rightarrow \) 's upred \( ([\cdot] \cdot \cdot [0,999] \cdot 999) \)
where [upred-defs]: dBox A \Phi = A \text{ wp } \Phi

**Definition dDia** :: 's hrel \( \Rightarrow \) 's upred \( \Rightarrow \) 's upred \( (<\cdot>) \cdot [0,999] \cdot 999) \)
where [upred-defs]: dDia A \Phi = (\neg [A] (\neg \Phi))

22.2 Box Laws

**Lemma dBox-false** [dynlog-simp]: [false]\Phi = true
by (rel-auto)

**Lemma dBox-skip** [dynlog-simp]: [II]\Phi = \Phi
by (rel-auto)

**Lemma dBox-assigns** [dynlog-simp]: [(\sigma)\sigma]\Phi = (\sigma \uparrow \Phi)
by (simp add: dBox-def wp-assigns-r)

**Lemma dBox-choice** [dynlog-simp]: \( [P \sqcap Q] \Phi = ([P] \Phi \land [Q] \Phi) \)
by (rel-auto)

**Lemma dBox-seq** [P \cdot Q] \Phi = [P][Q] \Phi
by (simp add: dBox-def wp-seq-r)

lemma dBox-star-unfold: \([P^*]\Phi = (\Phi \land [P][P^*]\Phi)\)
  by (metis dBox-choice dBox-seq dBox-skip ustar-unfoldl)

lemma dBox-star-induct: ‘(\Phi \land [P^*](\Phi \Rightarrow [P]\Phi)) \Rightarrow [P^*]\Phi’) 
  by (rel-simp, metis (mono-tags, lifting) mem-Collect-eq rtrancl-induct)

lemma dBox-test: \([?][p]\Phi = (p \Rightarrow \Phi)\)
  by (rel-auto)

22.3 Diamond Laws

lemma dDia-false [dynlog-simp]: <false>\Phi = false 
  by (simp add: dBox-false dDia-def)

lemma dDia-skip [dynlog-simp]: <II>\Phi = \Phi 
  by (simp add: dBox-skip dDia-def)

lemma dDia-assigns [dynlog-simp]: <\sigma>\Phi = (\sigma \dagger \Phi) 
  by (simp add: dBox-assigns dDia-def subst-not)

lemma dDia-choice: <P \sqcap Q>\Phi = (<P>\Phi \lor <Q>\Phi) 
  by (simp add: dBox-def dDia-def wp-choice)

lemma dDia-seq: <P ;; Q>\Phi = <P><Q>\Phi 
  by (simp add: dBox-def dDia-def wp-seq-r)

lemma dDia-test: <?[p]>\Phi = (p \land \Phi) 
  by (rel-auto)

22.4 Sequent Laws

lemma sBoxSeq [dynlog-simp]: \Gamma \vdash [P ;; Q]\Phi \equiv \Gamma \vdash [P][Q]\Phi 
  by (simp add: dBox-def wp-seq-r)

lemma sBoxTest [dynlog-intro]: \Gamma \vdash (b \Rightarrow \Psi) \implies \Gamma \vdash [?[b]]\Psi 
  by (rel-auto)

lemma sBoxAssignFwd [dynlog-simp]: [vwb-lens x; x := v; x \in \Gamma] \implies (\Gamma \vdash [x := v]\Phi) = ((\&x =_u v \land \Gamma) \vdash \Phi) 
  by (rel-auto, metis vwb-lens-wb wb-lens.get-put)

lemma sBoxIndStar: \vdash [\Phi \Rightarrow [P]\Phi]_u \equiv \Phi \vdash [P^*]\Phi 
  by (rel-auto, metis mono-tags, lifting) mem-Collect-eq rtrancl-induct)

lemma hoare-as-dynlog: \lfloor p \rfloor Q \lfloor r \rfloor_u = (p \vdash [Q]r) 
  by (rel-auto)

end

23 State Variable Declaration Parser

theory utp-state-parser
  imports utp-rel
This theory sets up a parser for state blocks, as an alternative way of providing lenses to a predicate. A program with local variables can be represented by a predicate indexed by a tuple of lenses, where each lens represents a variable. These lenses must then be supplied with respect to a suitable state space. Instead of creating a type to represent this alphabet, we can create a product type for the state space, with an entry for each variable. Then each variable becomes a composition of the $\text{fst}_L$ and $\text{snd}_L$ lenses to index the correct position in the variable vector.

We first creation a vacuous definition that will mark when an indexed predicate denotes a state block.

**definition** state-block :: $'v \Rightarrow 'p \Rightarrow 'v \Rightarrow 'p$ where

[upred-defs]:: state-block f x = f x

We declare a number of syntax translations to produce lens and product types, to obtain a type for the overall state space, to construct a tuple that denotes the lens vector parameter, to construct the vector itself, and finally to construct the state declaration.

**translations**

```
(type) PAIRTYPE('a, 'b) => (type) 'a × 'b  
(type) LENSTYPE('a, 'b) => (type) 'a ⇒ 'b  

-state-type (-constrain x t) => t  
-state-tuple st (-constrain x t) => -constrain x (-lensT t st)  
-state-tuple st (CONST Pair (-constrain x t) vs) => 
  CONST Product-Type.Pair (-constrain x (-lensT t st)) (-state-tuple st vs)  

-state-decl vs P =>  
  CONST state-block (-abs (-state-tuple (-state-type vs) vs) P) (-state-lenses vs)  
-state-decl vs P <= CONST state-block (-abs vs P) k
```

**parse-translation**

```
let
  open HOLogic;
  val lens-comp = Const (@{const-syntax lens-comp}, dummyT);
  val fst-lens = Const (@{const-syntax fst-lens}, dummyT);
  val snd-lens = Const (@{const-syntax snd-lens}, dummyT);
  val id-lens = Const (@{const-syntax id-lens}, dummyT);
  (* Construct a tuple of lenses for each of the possible locally declared variables *)
  fun
    state-lenses n st =
      if (n = 1)
        then st
        else pair-const dummyT dummyT $(lens-comp $ fst-lens $ st) $(state-lenses (n - 1) (lens-comp $ snd-lens $ st));
  fun
```
(∗ Add up the number of variable declarations in the tuple ∗)
var-decl-num (Const (♯{const-syntax Product-Type.Pair}.const-syntax Product-Type.Pair),.const-syntax Product-Type.Pair) $ - $ vs = var-decl-num vs + 1 |
var-decl-num - = 1;

fun state-lens ctxt [vs] = state-lenses (var-decl-num vs) id-lens ;
in [[-state-lenses, state-lens]]
end

23.1 Examples

term LOCAL (x::int, y::real, z::int) · x := (&x + &z)

lemma LOCAL p · II = II
  by (rel-auto)
end

24 Relational Operational Semantics

theory utp-rel-opsem
  imports
    utp-rel-laws
    utp-hoare
begin

This theory uses the laws of relational calculus to create a basic operational semantics. It is
based on Chapter 10 of the UTP book [22].

fun trel :: 'α usubst × 'α hrel ⇒ 'α usubst × 'α hrel ⇒ bool (infix → u 85) where
(σ, P) →_u (ϱ, Q) <-> (⟨σ⟩a ;; P) ⊑ (⟨ϱ⟩a ;; Q)

lemma trans-trel:
[ [ (σ, P) →_u (ϱ, Q); (ϱ, Q) →_u (φ, R) ] ] ⇒ (σ, P) →_u (φ, R)
by auto

lemma skip-trel: (σ, II) →_u (σ, II)
  by simp

lemma assigns-trel: (σ, ⟨ψ⟩a) →_u (ψ o σ, II)
  by (simp add: assigns-comp)

lemma assign-trel:
(σ, x := v) →_u (σ (&x := σ ⊢ v), II)
by (simp add: assigns-comp usubst)

lemma seq-trel:
  assumes (σ, P) →_u (ψ, Q)
  shows (σ, P ;; R) →_u (ψ, Q ;; R)
by (metis (no-types, lifting) assms order-refl seqr-assoc seqr-mono trel.simps)

lemma seq-skip-trel:
(σ, II ;; P) →_u (σ, P)
Theorem linking Hoare calculus and operational semantics. If we start $Q$ in a state $\sigma_0$ satisfying $p$, and $Q$ reaches final state $\sigma_1$ then $r$ holds in this final state.

\begin{quote}
\textbf{Theorem} \textit{hoare-opsem-link}:

\begin{align*}
\{p\} Q \{r\} = (\forall \sigma_0 \sigma_1. \ ' \sigma_0 \uparrow p' \land (\sigma_0, Q) \rightarrow_u (\sigma_1, II) \rightarrow ' \sigma_1 \uparrow r')
\end{align*}
\end{quote}

apply (rel-auto)
apply (rename-tac a b)
apply (drule-tac x=\lambda -. a in spec, simp)
apply (drule-tac x=\lambda -. b in spec, simp)
done

declare trel.simps [simp del]
definition utp-sym-eval :: 's usubst ⇒ 's hrel ⇒ 's hrel (infixr |= 55) where 
[upred-defs]: utp-sym-eval Γ P = ((Γ)ₐ ;; P)

named-theorems symeval

lemma seq-symeval [symeval]: Γ |= P ;; Q = (Γ |= P) ;; Q 
by (rel-auto)

lemma assigns-symeval [symeval]: Γ |= (σ)ₐ = (σ o Γ) |= II 
by (rel-auto)

lemma term-symeval [symeval]: (Γ |= II) ;; P = Γ |= P 
by (rel-auto)

lemma if-true-symeval [symeval]: [ Γ † b = true ] ⇒ Γ |= (P <<< b ⊞ r Q) = Γ |= P 
by (simp add: utp-sym-eval-def usubst assigns-r-comp)

lemma if-false-symeval [symeval]: [ Γ † b = false ] ⇒ Γ |= (P <<< b ⊞ r Q) = Γ |= Q 
by (simp add: utp-sym-eval-def usubst assigns-r-comp)

lemma while-true-symeval [symeval]: [ Γ † b = true ] ⇒ Γ |= while b do P od = Γ |= (P ;; while b do P od) 
by (subst while-unfold, simp add: symeval)

lemma while-false-symeval [symeval]: [ Γ † b = false ] ⇒ Γ |= while b do P od = Γ |= II 
by (subst while-unfold, simp add: symeval)

lemma while-inv-true-symeval [symeval]: [ Γ † b = true ] ⇒ Γ |= while b invr S do P od = Γ |= (P ;; while b do P od) 
by (metis while-inv-def while-true-symeval)

lemma while-inv-false-symeval [symeval]: [ Γ † b = false ] ⇒ Γ |= while b invr S do P od = Γ |= II 
by (metis while-false-symeval while-inv-def)

method sym-eval = (simp add: symeval usubst lit-simps[THEN sym]), (simp del: One-nat-def add: One-nat-def[THEN sym])?

syntax
-terminated :: logic ⇒ logic (terminated: - [999] 999)

translations
   terminated: Γ == Γ |= II

end

26 Strong Postcondition Calculus

theory utp-sp
imports utp-wp
begin

named-theorems sp

method sp-tac = (simp add: sp)
consts
  usp :: 'a ⇒ 'b ⇒ 'c (infix sp 60)

definition sp-upred :: 'a cond ⇒ ('α, 'β) urel ⇒ 'β cond where
  sp-upred p Q = ⌈(⌈p⌉ > ; Q) :: ('α, 'β) urel⌉

adhoc-overloading
  usp sp-upred

declare sp-upred-def [upred-defs]

lemma sp-false [sp]: p sp false = false
  by (rel-simp)

lemma sp-true [sp]: q ≠ false ==> q sp true = true
  by (rel-auto)

lemma sp-assigns-r [sp]:
  vwb-lens x =⇒ (p sp x := e) = (∃ v. p<e[x/x]> ∧ &x =:: e<e[x/x]>)
  by (rel-auto, metis vwb-lens-wb wb-lens.get-put, metis vwb-lens.put-eq)

lemma sp-it-is-post-condition:
  ⌈p⌉ C [p sp C] ᵃ
  by rel-blast

lemma sp-it-is-the-strongest-post:
  'p sp C ⇒ Q' =⇒ ⌈p⌉ C [Q] ᵃ
  by rel-blast

lemma sp-so:
  'p sp C ⇒ Q' = ⌈p⌉ C [Q] ᵃ
  by rel-blast

theorem sp-hoare-link:
  ⌈p⌉ Q [r] ᵃ ←⇒ (r ⊑ p sp Q)
  by rel-auto

lemma sp-while-r [sp]:
  assumes ('pre ⇒ I') and ([I ∧ b] C [I'] ᵃ) and ('I' ⇒ I')
  shows (pre sp invar I while⊥ b do C od) = (~b ∧ I)
  unfolding sp-upred-def
  oops

theorem sp-eq-intro: ⌜∧ r. r sp P = r sp Q} ⇒ P = Q
  by (rel-auto robust, fastforce+)

lemma wp-sp-sym:
  'prog wp (true sp prog)'
  by rel-auto

lemma it-is-pre-condition:
  ⌈C wp Q⌉ C [Q] ᵃ
  by rel-blast

lemma it-is-the-weakest-pre:
  P ⇒ C wp Q' = ⌈P⌉ C [Q] ᵃ
  by rel-blast
lemma s-pre: 'P ⇒ C wp Q' = {P} C {Q} u by rel-blast
end

27 Concurrent Programming

case theory utp-concurrency
  imports
    utp-hoare
    utp-rel
    utp-tactics
    utp-theory
begin

In this theory we describe the UTP scheme for concurrency, parallel-by-merge, which provides a general parallel operator parametrised by a “merge predicate” that explains how to merge the after states of the composed predicates. It can thus be applied to many languages and concurrency schemes, with this theory providing a number of generic laws. The operator is explained in more detail in Chapter 7 of the UTP book [22].

27.1 Variable Renamings

In parallel-by-merge constructions, a merge predicate defines the behaviour following execution of parallel processes, \( P \parallel Q \), as a relation that merges the output of \( P \) and \( Q \). In order to achieve this we need to separate the variable values output from \( P \) and \( Q \), and in addition the variable values before execution. The following three constructs do these separations. The initial state-space before execution is \( \alpha' \), the final state-space after the first parallel process is \( \beta_0 \), and the final state-space for the second is \( \beta_1 \). These three functions lift variables on these three state-spaces, respectively.

alphabet \( (\alpha', \beta_0, \beta_1) \) mrg =
  mrg-prior :: 'a
  mrg-left :: 'beta
  mrg-right :: 'beta1

definition pre-uvar :: ('a ⇒ 'a) ⇒ ('a ⇒ ('a, 'beta_0, 'beta_1) mrg) where
  upred-defs: pre-uvar x = x ;_L mrg-prior

definition left-uvar :: ('a ⇒ 'beta_0) ⇒ ('a ⇒ ('a, 'beta_0, 'beta_1) mrg) where
  upred-defs: left-uvar x = x ;_L mrg-left

definition right-uvar :: ('a ⇒ 'beta_1) ⇒ ('a ⇒ ('a, 'beta_0, 'beta_1) mrg) where
  upred-defs: right-uvar x = x ;_L mrg-right

We set up syntax for the three variable classes using a subscript <, 0-x, and 1-x, respectively.

syntax
  -svarpre :: svid ⇒ svid (< [995] 995)
  -svarleft :: svid ⇒ svid (0-- [995] 995)
  -svarright :: svid ⇒ svid (1-- [995] 995)

translations
  -svarpre x == CONST pre-uvar x
We proved behavedness closure properties about the lenses.

lemma left-uvar [simp]: \( \text{vwb-lens } x \implies \text{vwb-lens } (\text{left-uvar } x) \)
by (simp add: left-uvar-def)

lemma right-uvar [simp]: \( \text{vwb-lens } x \implies \text{vwb-lens } (\text{right-uvar } x) \)
by (simp add: right-uvar-def)

lemma pre-uvar [simp]: \( \text{vwb-lens } x \implies \text{vwb-lens } (\text{pre-uvar } x) \)
by (simp add: pre-uvar-def)

lemma left-uvar-mwb [simp]: \( \text{mwb-lens } x \implies \text{mwb-lens } (\text{left-uvar } x) \)
by (simp add: left-uvar-def)

lemma right-uvar-mwb [simp]: \( \text{mwb-lens } x \implies \text{mwb-lens } (\text{right-uvar } x) \)
by (simp add: right-uvar-def)

lemma pre-uvar-mwb [simp]: \( \text{mwb-lens } x \implies \text{mwb-lens } (\text{pre-uvar } x) \)
by (simp add: pre-uvar-def)

We prove various independence laws about the variable classes.

lemma left-uvar-indep-right-uvar [simp]:
\( \text{left-uvar } x \indep \text{right-uvar } y \)
by (simp add: left-uvar-def right-uvar-def lens-comp-assoc[THEN sym])

lemma left-uvar-indep-pre-uvar [simp]:
\( \text{left-uvar } x \indep \text{pre-uvar } y \)
by (simp add: left-uvar-def pre-uvar-def)

lemma left-uvar-indep-left-uvar [simp]:
\( x \indep y \implies \text{left-uvar } x \indep \text{left-uvar } y \)
by (simp add: left-uvar-def)

lemma right-uvar-indep-left-uvar [simp]:
\( \text{right-uvar } x \indep \text{left-uvar } y \)
by (simp add: lens-indep-sym)

lemma right-uvar-indep-pre-uvar [simp]:
\( \text{right-uvar } x \indep \text{pre-uvar } y \)
by (simp add: right-uvar-def pre-uvar-def)

lemma right-uvar-indep-right-uvar [simp]:
\( x \indep y \implies \text{right-uvar } x \indep \text{right-uvar } y \)
by (simp add: right-uvar-def)

lemma pre-uvar-indep-left-uvar [simp]:
\( \text{pre-uvar } x \indep \text{left-uvar } y \)
by (simp add: lens-indep-sym)

lemma pre-uvar-indep-right-uvar [simp]:

pre-uvar \( x \bowtie \) right-uvar \( y \)
by \((\text{simp add: lens-indep-sym})\)

lemma \(\text{pre-uvar-indep-pre-uvar} [\text{simp}]:\)
\( x \bowtie y \Rightarrow \text{pre-uvar} x \bowtie \text{pre-uvar} y \)
by \((\text{simp add: pre-uvar-def})\)

### 27.2 Merge Predicates

A merge predicate is a relation whose input has three parts: the prior variables, the output variables of the left predicate, and the output of the right predicate.

**type-synonym** \( \alpha \text{ merge } = ((\alpha, \alpha, \alpha) \text{ mrg, } \alpha) \text{ urel} \)

**skip** is the merge predicate which ignores the output of both parallel predicates

**definition** \(\text{skip}_m :: (\alpha \text{ merge where} \) \(\) [upred-defs]: \(\text{skip}_m = (\$v' = u \$v < ) \)

**swap** is a predicate that swaps the left and right indices; it is used to specify commutativity of the parallel operator

**definition** \(\text{swap}_m :: ((\alpha, \beta, \beta) \text{ mrg) hrel where} \)
[upred-defs]: \(\text{swap}_m = (0-v,1-v) := (\&1-v,\&0-v) \)

A symmetric merge is one for which swapping the order of the merged concurrent predicates has no effect. We represent this by the following healthiness condition that states that \(\text{swap}_m\) is a left-unit.

**abbreviation** \(\text{SymMerge} :: \alpha \text{ merge } \Rightarrow \alpha \text{ merge where} \)
\(\text{SymMerge}(M) \equiv (\text{swap}_m :: M) \)

### 27.3 Separating Simulations

\(U_0\) and \(U_1\) are relations modify the variables of the input state-space such that they become indexed with 0 and 1, respectively.

**definition** \(\text{U0} :: (\beta_0, (\alpha, \beta_0, \beta_1) \text{ mrg) urel where} \)
[upred-defs]: \(\text{U0} = (\$0-v' = u \$v) \)

**definition** \(\text{U1} :: (\beta_1, (\alpha, \beta_0, \beta_1) \text{ mrg) urel where} \)
[upred-defs]: \(\text{U1} = (\$1-v' = u \$v) \)

**lemma** \(\text{U0-swap}: (U0 :: \text{swap}_m) = U1 \)
by \((\text{rel-auto})\)

**lemma** \(\text{U1-swap}: (U1 :: \text{swap}_m) = U0 \)
by \((\text{rel-auto})\)

As shown below, separating simulations can also be expressed using the following two alphabet extrusions

**definition** \(\text{U0}\alpha \) where [upred-defs]: \(\text{U0}\alpha = (1_L \times_L \text{ mrg-left}) \)

**definition** \(\text{U1}\alpha \) where [upred-defs]: \(\text{U1}\alpha = (1_L \times_L \text{ mrg-right}) \)

We then create the following intuitive syntax for separating simulations.

**abbreviation** \(\text{U0-alpha-lift} ([\cdot]_0) \) where \([P]_0 \equiv P \oplus_p U0\alpha \)

151
abbreviation \( U1\)-alpha-lift \( ([\cdot]_1) \) where \([P]_1 \equiv P \oplus_{U1} \alpha \)

\([P]_0\) is predicate \( P\) where all variables are indexed by 0, and \([P]_1\) is where all variables are indexed by 1. We can thus equivalently express separating simulations using alphabet extrusion.

lemma U0-as-alpha: \((P ;; U0) = [P]_0\)
by (rel-auto)

lemma U1-as-alpha: \((P ;; U1) = [P]_1\)
by (rel-auto)

lemma U0-alpha-vwb-lens [simp]: \(vwb\)-lens \( U0\alpha \)
by (simp add: \( U0\alpha\)-def \( id\)-vwb-lens \( prod\)-vwb-lens)

lemma U1-alpha-vwb-lens [simp]: \(vwb\)-lens \( U1\alpha \)
by (simp add: \( U1\alpha\)-def \( id\)-vwb-lens \( prod\)-vwb-lens)

lemma U0-alpha-out-var [alpha]: \([x]_0 = \#0\-x\)'
by (rel-auto)

lemma U1-alpha-out-var [alpha]: \([x]_1 = \#1\-x\)'
by (rel-auto)

lemma U0-skip [alpha]: \([II]_0 = (\#0\-v\)\' =_a \$v\)
by (rel-auto)

lemma U1-skip [alpha]: \([II]_1 = (\#1\-v\)\' =_a \$v\)
by (rel-auto)

lemma U0-seqr [alpha]: \([P ;; Q]_0 = P ;; [Q]_0\)
by (rel-auto)

lemma U1-seqr [alpha]: \([P ;; Q]_1 = P ;; [Q]_1\)
by (rel-auto)

lemma U0alpha-comp-in-var [alpha]: \((in\-var \; x) \; ;_L \; U0\alpha = in\-var \; x\)
by (simp add: \( U0\alpha\)-def \( alpha\)-in-var \( in\-var\)-prod-lens \( pre\-wvar\)-def)
lemma \( U_0 \alpha\text{-comp-out-var} \[\alpha\] : (out-var \ x) \parallel L \ U_0 \alpha = \text{out-var} \ (\text{left-var} \ x) \)
by (simp add: \( \alpha \)-def alpha-out-var id-wb-lens left-var-def out-var-prod-lens)

lemma \( U_1 \alpha\text{-comp-in-var} \[\alpha\] : (in-var \ x) \parallel L \ U_1 \alpha = \text{in-var} \ x \)
by (simp add: \( \alpha \)-def alpha-in-var in-var-prod-lens pre-var-def)

lemma \( U_1 \alpha\text{-comp-out-var} \[\alpha\] : (out-var \ x) \parallel L \ U_1 \alpha = \text{out-var} \ (\text{right-var} \ x) \)
by (simp add: \( \alpha \)-def alpha-out-var id-wb-lens right-var-def out-var-prod-lens)

### 27.4 Associative Merges

Associativity of a merge means that if we construct a three way merge from a two way merge and then rotate the three inputs of the merge to the left, then we get exactly the same three way merge back.

We first construct the operator that constructs the three way merge by effectively wiring up the two way merge in an appropriate way.

definition ThreeWayMerge :: \( \alpha \text{ merge} \Rightarrow \) \((\alpha, \alpha, \alpha) \) merge \( \Rightarrow \) \((\alpha, \alpha, \alpha) \) merge where 
[upred-defs]: ThreeWayMerge \( M = ((\&0 \parallel v)^{-1} = u \&0 \parallel v \& \&1 \parallel v^{-1} = u \&1 \parallel 0 \parallel v \& \&v_{<'} = u \&v_{<'}) ; M :: U0 \parallel \&1 \parallel v^{-1} = u \&1 \parallel 0 \parallel v \& \&v_{<'} = u \&v_{<'}) ; M\]

The next definition rotates the inputs to a three way merge to the left one place.

abbreviation rotate\_m where rotate\_m \( \equiv (0 \parallel v, 1 \parallel 0 \parallel v, 1 \parallel 1 \parallel v) \Rightarrow (\&1 \parallel 0 \parallel v, \&1 \parallel 1 \parallel v, \&0 \parallel v)\)

Finally, a merge is associative if rotating the inputs does not effect the output.

definition AssocMerge :: \( \alpha \text{ merge} \Rightarrow \text{bool} \) where 
[upred-defs]: AssocMerge \( M = (\text{rotate}\_m \Rightarrow M M \parallel \gamma M) \)

### 27.5 Parallel Operators

We implement the following useful abbreviation for separating of two parallel processes and copying of the before variables, all to act as input to the merge predicate.

abbreviation par-sep (infixr \( \parallel \) 85) where 
\( P \parallel s Q \equiv (P \parallel U0) \& (Q \parallel U1) \& \&v_{<'} = u \&v\)

The following implementation of parallel by merge is less general than the book version, in that it does not properly partition the alphabet into two disjoint segments. We could actually achieve this specifying lenses into the larger alphabet, but this would complicate the definition of programs. May reconsider later.

definition par-by-merge :: \( \alpha \text{ merge} \Rightarrow \) \((\alpha, \beta) \) merge \( \Rightarrow \) \((\alpha, \beta) \) merge \( \Rightarrow \) \((\alpha, \beta) \) merge where 
[upred-defs]: \( P \parallel M Q = (P \parallel s Q \parallel M)\)

lemma par-by-merge-alt-def: \( P \parallel M Q = ([P]_0 \& [Q]_1 \& \&v_{<'}^{-1} = u \&v) ; M \)
by (simp add: par-by-merge-def U0-as-alpha U1-as-alpha)

lemma shEx-pbm-left: \((\exists x \cdot P \ x) \parallel M Q = (\exists x \cdot (P \ x \parallel M Q))\)
by (rel-auto)

lemma shEx-pbm-right: \((P \parallel M (\exists x \cdot Q \ x)) = (\exists x \cdot (P \parallel M Q x))\)
by (rel-auto)
27.6 Unrestriction Laws

**Lemma** unrest-in-par-by-merge [unrest]:
\[
\text{by (rel-auto, fastforce+)}
\]

**Lemma** unrest-out-par-by-merge [unrest]:
\[
\text{by (rel-auto)}
\]

27.7 Substitution laws

Substitution is a little tricky because when we push the expression through the composition operator the alphabet of the expression must also change. Consequently for now we only support literal substitution, though this could be generalised with suitable alphabet coercions. We need quite a number of variants to support this which are below.

**Lemma** U0-seq-subst: \((P :: U0)[<v>/$0-x'] = (P[<v>/$x'] :: U0)\)
\[
\text{by (rel-auto)}
\]

**Lemma** U1-seq-subst: \((P :: U1)[<v>/$1-x'] = (P[<v>/$x'] :: U1)\)
\[
\text{by (rel-auto)}
\]

**Lemma** lit-pbm-subst [usubst]:
\[
\text{fixes } x :: (- \implies 'a)
\]

**Lemma** bool-pbm-subst [usubst]:
\[
\text{fixes } x :: (- \implies 'a)
\]

**Lemma** zero-one-pbm-subst [usubst]:
\[
\text{fixes } x :: (- \implies 'a)
\]

**Lemma** numeral-pbm-subst [usubst]:
\[
\text{fixes } x :: (- \implies 'a)
\]
27.8 Parallel-by-merge laws

**Lemma** par-by-merge-false [simp]:
\[ P \parallel \text{false} Q = \text{false} \]
by (rel-auto)

**Lemma** par-by-merge-left-false [simp]:
\[ \text{false} \parallel M Q = \text{false} \]
by (rel-auto)

**Lemma** par-by-merge-right-false [simp]:
\[ P \parallel M \text{false} = \text{false} \]
by (rel-auto)

**Lemma** par-by-merge-seq-add: \((P \parallel M Q) ;; R) = (P ;; M ;; R Q)\)
by (simp add: par-by-merge-def seqr-assoc)

A skip parallel-by-merge yields a skip whenever the parallel predicates are both feasible.

**Lemma** par-by-merge-skip:
assumes \(P ;; \text{true} = \text{true} Q ;; \text{true} = \text{true}\)
shows \(P \parallel \text{skip}_m Q = \text{II}\)
using assms by (rel-auto)

**Lemma** skip-merge-swap: \(\text{swap}_m ;; \text{skip}_m = \text{skip}_m\)
by (rel-auto)

**Lemma** par-sep-swap: \(P ||_s Q ;; \text{swap}_m = Q ||_s P\)
by (rel-auto)

Parallel-by-merge commutes when the merge predicate is unchanged by swap

**Lemma** par-by-merge-commute-swap:
shows \(P \parallel M Q = Q \parallel \text{swap}_m ;; M P\)
proof –
have \(Q \parallel \text{swap}_m ;; M P = (((Q ;; U0) \land (P ;; U1) \land \$v < v' = \_ \$v) ;; \text{swap}_m) ;; M)\)
by (simp add: par-by-merge-def seqr-assoc)
also have \(...) = (((Q ;; U0 ;; \text{swap}_m) \land (P ;; U1 ;; \text{swap}_m) \land \$v < v' = \_ \$v) ;; M)\)
by (rel-auto)
also have \(...) = (((Q ;; U1) \land (P ;; U0) \land \$v < v' = \_ \$v) ;; M)\)
by (simp add: U0-swap U1-swap)
also have \(...) = P \parallel M Q\)
by (simp add: par-by-merge-def utp-pred-laws.inf.left-commute)
finally show \(?\text{thesis} \) .
qed

**Theorem** par-by-merge-commute:
assumes \(M \text{ is SymMerge}\)
shows \(P \parallel M Q = Q \parallel_M P\)
by (metis Healthy-if assms par-by-merge-commute-swap)

**Lemma** par-by-merge-mono-1:
assumes \(P_1 \subseteq P_2\)
shows \(P_1 \parallel_M Q \subseteq P_2 \parallel_M Q\)
using assms by (rel-auto)
lemma par-by-merge mono-2:
assumes $Q_1 \sqsubseteq Q_2$
shows $\langle P \parallel M Q_1 \rangle \sqsubseteq \langle P \parallel M Q_2 \rangle$
using assms by (rel-blast)

lemma par-by-merge mono:
assumes $P_1 \sqsubseteq P_2$ $Q_1 \sqsubseteq Q_2$
shows $\langle P_1 \parallel M Q_1 \rangle \sqsubseteq \langle P_2 \parallel M Q_2 \rangle$
by (meson assms dual-order.trans par-by-merge mono-1 par-by-merge mono-2)

theorem par-by-merge assoc:
assumes $M$ is SymMerge AssocMerge $M$
shows $\langle P \parallel M Q \parallel M R \rangle = \langle P \parallel M (Q \parallel M R) \rangle$
proof -
  have $\langle P \parallel M Q \parallel M R \rangle = \langle P \parallel M (Q \parallel M R) \rangle$
    by (rel-blast)
  also have $\langle P \parallel M Q \parallel M R \rangle = \langle P \parallel M (Q \parallel M R) \rangle$
    by (rel-blast)
  also have $\langle P \parallel M Q \parallel M R \rangle = \langle P \parallel M (Q \parallel M R) \rangle$
    by (rel-blast)
  also have $\langle P \parallel M Q \parallel M R \rangle = \langle P \parallel M (Q \parallel M R) \rangle$
    by (rel-blast)
  finally show $\langle P \parallel M Q \parallel M R \rangle = \langle P \parallel M (Q \parallel M R) \rangle$
qed

theorem par-by-merge choice left:
$\langle P \cap Q \parallel M R \rangle = \langle P \parallel M R \cap (Q \parallel M R) \rangle$
by (rel-auto)

theorem par-by-merge choice right:
$\langle P \parallel M (Q \cap R) \rangle = \langle P \parallel M (Q \parallel M R) \rangle$
by (rel-auto)

theorem par-by-merge or left:
$\langle P \lor Q \parallel M R \rangle = \langle P \parallel M (R \parallel Q \parallel M R) \rangle$
by (rel-auto)

theorem par-by-merge or right:
$\langle P \parallel M (Q \lor R) \rangle = \langle P \parallel M (Q \parallel R \parallel M R) \rangle$
by (rel-auto)

theorem par-by-merge-USUP mem left:
$\langle \bigcap \{ i \in I \cdot P(i) \parallel M \} Q \parallel M (\bigcap \{ i \in I \cdot P(i) \parallel M \}) \rangle$
by (rel-auto)

theorem par-by-merge-USUP ind left:
$\langle \bigcap \{ i \cdot P(i) \parallel M \} Q \parallel M (\bigcap \{ i \cdot P(i) \parallel M \}) \rangle$
by (rel-auto)

theorem par-by-merge-USUP mem right:
$\langle P \parallel M (\bigcap \{ i \in I \cdot Q(i) \parallel M \}) \rangle = \langle \bigcap \{ i \in I \cdot P(i) \parallel M \} \rangle$
by (rel-auto)
27.9 Example: Simple State-Space Division

The following merge predicate divides the state space using a pair of independent lenses.

**definition** StateMerge :: ('a α⇒ 'a) ⇒ ('b α⇒ 'a) ⇒ α merge (M[-]-|)-α where

[upred-defs]: M[a|b]|σ = (|$v| =_u ($v< ≔ $θ(v) on &a) ⊕ $I − v on &b)

**lemma** swap-StateMerge: a b ⇒ (swap_m :: M[a|b]|σ) = M[b|a]|σ

by (rel-auto, simp-all add: lens-indep-comm)

**abbreviation** StateParallel :: 'a hrel ⇒ ('a α⇒ 'a) ⇒ ('b α⇒ 'a) ⇒ 'a hrel ⇒ 'a hrel (- |)-α - [85.0,0.86] 86

where P | a|b|σ Q = P |M[a|b]|σ Q

**lemma** StateParallel-commute: a b ⇒ P | a|b|σ Q = Q | b|a|σ P

by (metis par-by-merge-commute-swap swap-StateMerge)

**lemma** StateParallel-form:

P | a|b|σ Q = (∃ (st₀, st₁) · P |!st₀>/$v| ≦ Q |!st₁>/$v|) ∧ $v| =_u ($v + $st₀ on &a) ⊕ $st₁ on &b)

by (rel-auto)

**lemma** StateParallel-form':

assumes web-lens a web-lens b a b ⇒

shows P | a|b|σ Q = {&a.&b}:[(P |v (|$v| !a|) ∧ (Q |v (|$v| !b|)))

using assms

apply (simp add: StateParallel-form, rel-auto)

apply (metis web-lens-wb wb-lens-axioms-def wb-lens-def)

apply (metis web-lens-wb wb-lens-get-put)

apply (simp add: lens-indep-comm)

apply (metis (no-types, hide-lams) lens-indep-comm web-lens-wb wb-lens-def weak-lens.put-get)

done

We can frame all the variables that the parallel operator refers to

**lemma** StateParallel-frame:

assumes web-lens a web-lens b a b ⇒

shows {&a.&b}:P | a|b|σ Q = P | a|b|σ Q

using assms

apply (simp add: StateParallel-form, rel-auto)

using lens-indep-comm apply fastforce+

done

Parallel Hoare logic rule. This employs something similar to separating conjunction in the postcondition, but we explicitly require that the two conjuncts only refer to variables on the left and right of the parallel composition explicitly.

**theorem** StateParallel-hoare [hoare]:

assumes c P d₁ u c Q d₂ u a b d₁ b b d₂

shows c P | a|b|σ Q | d₁ d₂ u

proof —

— Parallelise the specification
from assms(4,5) have 1:\((c|c| \Rightarrow [d_1 \wedge d_2]|) \subseteq ([c]< \Rightarrow [d_1]|) \ |a|b|\sigma\ ([c]< \Rightarrow [d_2]|)\) (is ?lhs \subseteq ?rhs)
  by (simp add: StateParallel-form, rel-auto, metis assms(3) lens-indep-comm)
  — Prove Hoare rule by monotonicity of parallelism
have 2:?rhs \subseteq P \ |a|b|\sigma\ Q
proof (rule par-by-merge-mono)
  show \((c|c| \Rightarrow [d_1]|) \subseteq P\)
    using assms(1) hoare-r-def by auto
  show \((c|c| \Rightarrow [d_2]|) \subseteq Q\)
    using assms(2) hoare-r-def by auto
qed
show \?thesis
  unfolding hoare-r-def using 1 2 order-trans by auto
qed

Specialised version of the above law where an invariant expression referring to variables outside
the frame is preserved.

theorem StateParallel-frame-hoare [hoare]:
  assumes vwb-lens a vwb-lens b a \& b a  d_1 \& b \ b  d_2 a \& c_1 \ b \ c_1 \ & c_1 \ & c_2} P \ & d_1 \ & \{c_1 \ & c_2\} Q \ & d_2\u
  shows \{c_1 \ & c_2\} P \ & a|b|\sigma\ Q \ & c_1 \ & d_1 \ & d_2\u
proof –
  have \{c_1 \ & c_2\} \{&a,&b\}:\{P \ & a|b|\sigma\ Q\} \{c_1 \ & d_1 \ & d_2\u
    by (auto intro!: frame-hoare-r' StateParallel-hoare simp add: assms unrest plus-vwb-lens)
  thus \?thesis
    by (simp add: StateParallel-frame assms)
qed

end

28 Meta-theory for the Standard Core

theory utp
imports
  utp-var
  utp-expr
  utp-expr-insts
  utp-expr-funcs
  utp-unrest
  utp-usedby
  utp-subst
  utp-meta-subst
  utp-alphabet
  utp-lift
  utp-pred
  utp-pred-laws
  utp-recursion
  utp-dynlog
  utp-rel
  utp-rel-laws
  utp-sequent
  utp-state-parser
  utp-sgm-eval
  utp-tactics
  utp-hoare
  utp-wp

158
29 Overloaded Expression Constructs

theory utp-expr-ovld
imports utp
begin

29.1 Overloadable Constants

For convenience, we often want to utilise the same expression syntax for multiple constructs. This can be achieved using ad-hoc overloading. We create a number of polymorphic constants and then overload their definitions using appropriate implementations. In order for this to work, each collection must have its own unique type. Thus we do not use the HOL map type directly, but rather our own partial function type, for example.

consts
— Empty elements, for example empty set, nil list, 0...
  uempty :: 'f
— Function application, map application, list application...
  uapply :: 'f ⇒ 'k ⇒ 'v
— Function update, map update, list update...
  uupd :: 'f ⇒ 'k ⇒ 'v ⇒ 'f
— Domain of maps, lists...
  udom :: 'f ⇒ 'a set
— Range of maps, lists...
  uran :: 'f ⇒ 'b set
— Domain restriction
  udomres :: 'a set ⇒ 'f ⇒ 'f
— Range restriction
  uranres :: 'f ⇒ 'b set ⇒ 'f
— Collection cardinality
  ucard :: 'f ⇒ nat
— Collection summation
  usums :: 'f ⇒ 'a
— Construct a collection from a list of entries
  uentries :: 'k set ⇒ ('k ⇒ 'v) ⇒ 'f

We need a function corresponding to function application in order to overload.

definition fun-apply :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
where fun-apply f x = f x

declare fun-apply-def [simp]

definition ffun-entries :: 'k set ⇒ ('k ⇒ 'v) ⇒ ('k, 'v) ffun where
  ffun-entries d f = graph-ffun {(k, f k) | k. k ∈ d}

We then set up the overloading for a number of useful constructs for various collections.

adhoc-overloading
  uempty 0 and
29.2 Syntax Translations

syntax

- undef :: logic (⊥)
- umap-empty :: logic ([], )
- uapply :: ('a ⇒ 'b, 'a) uexpr ⇒ utuple-args ⇒ ('b, 'a) uexpr ('(-)')\[999,0\] 999
- umaplet :: [logic, logic] => umaplet (- / -)
  :: umaplet => umaplets ()
- UMaplets :: [umaplet, umaplets] => umaplets (- / -)
- UMapUpd :: [logic, umaplets] => logic (-'(-)\[900,0\] 900)
- UMap :: umaplets => logic ((1[⊥]))
- ucard :: logic ⇒ logic (#_\[(-)])
- udom :: logic ⇒ logic (dom\[(-)])
- uran :: logic ⇒ logic (ran\[(-)])
- usum :: logic ⇒ logic (sum\[(-)])
- udom-res :: logic ⇒ logic ⇒ logic (infixl \[\leq\]
- uran-res :: logic ⇒ logic ⇒ logic (infixl \[\geq\])
- uentries :: logic ⇒ logic ⇒ logic (entr\[(-)])

translations

— Pretty printing for adhoc-overloaded constructs

\[ \text{f}(\text{x})_\text{a} \leq \text{CONST} \text{uapply} \ \text{f} \ \text{x} \]
\[ \text{dom}_\text{a}(\text{f}) \leq \text{CONST} \text{udom} \ \text{f} \]
\[ \text{ran}_\text{a}(\text{f}) \leq \text{CONST} \text{uran} \ \text{f} \]
\[ \text{A} \leq \text{u} \text{f} \leq \text{CONST} \text{udomres} \ \text{A} \ \text{f} \]
\[ \text{f} \geq \text{u} \text{A} \leq \text{CONST} \text{wransres} \ \text{f} \ \text{A} \]
\[ \#_\text{u}(\text{f}) \leq \text{CONST} \text{uCARD} \ \text{f} \]
\[ \text{f}((\text{k} \mapsto \text{v}))_\text{a} \leq \text{CONST} \text{upld} \ \text{f} \ \text{k} \ \text{v} \]
\[ 0 \leq \text{CONST} \text{uempty} — \text{We have to do this so we don’t see uempty. Is there a better way of printing?} \]

— Overloaded construct translations

\[ \text{f}(\text{x},\text{y},\text{z},\text{u})_\text{a} \leq \text{CONST} \text{bop} \ \text{CONST} \text{uapply} \ \text{f} \ (\text{x},\text{y},\text{z},\text{u})_\text{u} \]
\[ \text{f}(\text{x},\text{y},\text{z})_\text{a} \leq \text{CONST} \text{bop} \ \text{CONST} \text{uapply} \ \text{f} \ (\text{x},\text{y},\text{z})_\text{u} \]
\[ \text{f}(\text{x})_\text{a} \leq \text{CONST} \text{bop} \ \text{CONST} \text{uapply} \ \text{f} \ \text{x} \]
\[ \#_\text{u}(\text{xs}) \leq \text{CONST} \text{uCARD} \ \text{xs} \]
\[ \text{sum}_\text{u}(\text{A}) \leq \text{CONST} \text{uCARD} \ \text{usums} \ \text{A} \]
\[ \text{dom}_\text{a}(\text{f}) \leq \text{CONST} \text{uCARD} \ \text{udom} \ \text{f} \]
\[ \text{ran}_\text{a}(\text{f}) \leq \text{CONST} \text{uCARD} \ \text{uran} \ \text{f} \]
\[ []_\text{u} \leq \langle \text{CONST} \text{uempty} \rangle \]
\[ \bot \leq \langle \text{CONST} \text{undefined} \rangle \]
\[ \text{A} \leq \text{u} \text{f} \leq \text{CONST} \text{bop} \ (\text{CONST} \ \text{udomres}) \ \text{A} \ \text{f} \]
\[ \text{f} \geq \text{u} \text{A} \leq \text{CONST} \text{bop} \ (\text{CONST} \ \text{wransres}) \ \text{f} \ \text{A} \]
\[ \text{entr}_\text{u}(\text{d},\text{f}) \leq \text{CONST} \text{bop} \ \text{CONST} \ \text{uentries} \ \text{d} \ \langle \text{f} \rangle \]
\[-\text{UMapUpd} \ \text{m} \ (-\text{UMaplets} \ \text{xy} \ \text{ms}) \leq -\text{UMapUpd} \ (-\text{UMapUpd} \ \text{m} \ \text{xy}) \ \text{ms} \]
29.3 Simplifications

**lemma ufun-apply-lit [simp]:**
\[ <f> \cdot (<x>) = <f(x)> \]
  by (transfer, simp)

**lemma lit-plus-appl [lit-norm]:**
\[ (<+) \cdot (<x>) = x + y \]
  by (simp add: uexpr-defs, transfer, simp)

**lemma lit-minus-appl [lit-norm]:**
\[ (<-) \cdot (<x>) = x - y \]
  by (simp add: uexpr-defs, transfer, simp)

**lemma lit-mult-appl [lit-norm]:**
\[ (<\times>) \cdot (<x>) = x * y \]
  by (simp add: uexpr-defs, transfer, simp)

**lemma lit-divide-appl [lit-norm]:**
\[ (<\div>) \cdot (<x>) = x / y \]
  by (simp add: uexpr-defs, transfer, simp)

**lemma pfun-entries-apply [simp]:**
\[ \text{entr}_{u}(d, f) :: ((k, v) \ pfun, 'a \ uexpr) \cdot (i) = (\langle f \cdot (i) \rangle) \cap i \in d \triangledown u \]
  by (pred-auto)

**lemma udom-update-pfun [simp]:**
\[ \text{fixes m :: ((k, v) \ pfun, 'a \ uexpr) \ shows dom}_{u}(m(k \mapsto v)_{u}) = \{k\}_{u} \cup \text{dom}_{u}(m) \]
  by (rel-auto)

**lemma uapply-update-pfun [simp]:**
\[ \text{fixes m :: ((k, v) \ pfun, 'a \ uexpr) \ shows (m(k \mapsto v)_{u} \cdot (i)_{a} = v \cap i =_{u} k \triangledown m(i)_{a} \]
  by (rel-auto)

29.4 Indexed Assignment

**syntax**

— Indexed assignment
\[-\text{assignment-upd} :: \text{svid} \Rightarrow \text{uexp} \Rightarrow \text{uexp} \Rightarrow \text{logic} (([-] := \cdot) [63, 0, 0] 62)\]

**translations**

— Indexed assignment uses the overloaded collection update function uupd.
\[-\text{assignment-upd} x k v = x := \& x(k \mapsto v)_{u} \]

end

30 Meta-theory for the Standard Core with Overloaded Constructs

**theory utp-full**
  imports utp utp-expr-ovld
begin end

31 UTP Easy Expression Parser

**theory utp-easy-parser**
  imports utp-easy-parser
begin
31.1 Replacing the Expression Grammar

The following theory provides an easy to use expression parser that is primarily targetted towards expressing programs. Unlike the built-in UTP expression syntax, this uses a closed grammar separate to the HOL logic nonterminal, that gives more freedom in what can be expressed. In particular, identifiers are interpreted as UTP variables rather than HOL variables and functions do not require subscripts and other strange decorations.

The first step is to remove the from the UTP parse the following grammar rule that uses arbitrary HOL logic to represent expressions. Instead, we will populate the uexp grammar manually.

\[\text{purge-syntax}\]
- \text{-uexp-l :: logic => uexp (- [64] 64)}

31.2 Expression Operators

\[\text{syntax}\]
- \text{-ue-quote :: uexp => logic ('\textbackslash'} e')
- \text{-ue-tuple :: uexprs => uexp ('\textbackslash'} e y z')
- \text{-ue-lit :: logic => uexp ('\textbackslash'} l')
- \text{-ue-var :: svid => uexp ('\textbackslash'} v')
- \text{-ue-eq :: uexp => uexp => uexp (infix \textbackslash'} =')
- \text{-ue-uop :: id => uexp => uexp (\textbackslash'} f x')
- \text{-ue-bop :: id => uexp => uexp => uexp (\textbackslash'} f x y')
- \text{-ue-trop :: id => uexp => uexp => uexp (\textbackslash'} f x y z')
- \text{-ue-apply :: uexp => uexp => uexp (\textbackslash'} f x')

\[\text{translations}\]
- \text{-ue-quote e => e}
- \text{-ue-tuple (-uexprs x (-uexprs y z)) => -ue-tuple (-uexprs x (-ue-tuple (-uexprs y z)))}
- \text{-ue-tuple (-uexprs x y) => CONST bop CONST Pair x y}
- \text{-ue-lit x => CONST lit x}
- \text{-ue-var x => CONST utp-expr.var (CONST pr-var x)}
- \text{-ue-apply f x => f(x)\textbackslash'}

31.3 Predicate Operators

\[\text{syntax}\]
- \text{-ue-true :: uexp (true)}
- \text{-ue-false :: uexp (false)}
- \text{-ue-not :: uexp => uexp (\textbar x \textbar) \textbar [40] 40)}
- \text{-ue-conj :: uexp => uexp => uexp (infixr \textbar)}
- \text{-ue-disj :: uexp => uexp => uexp (infixr \textbar)}
- \text{-ue-impl :: uexp => uexp => uexp (infixr \textbar)}
- \text{-ue-mem :: uexp => uexp => uexp (\textbar x \textbar') [151, 151] 150)}
- \text{-ue-nmem :: uexp => uexp => uexp (\textbar x \textbar') [151, 151] 150)}

\[\text{translations}\]
- \text{-ue-true => CONST true-upred}
31.4 Arithmetic Operators

syntax
- **ue-num** :: num-const ⇒ uexp (-)
- **ue-size** :: uexp ⇒ uexp (# [999] 999)
- **ue-eq** :: uexp ⇒ uexp ⇒ uexp (infix = 150)
- **ue-le** :: uexp ⇒ uexp ⇒ uexp (infix ≤ 150)
- **ue-ge** :: uexp ⇒ uexp ⇒ uexp (infix ≥ 150)
- **ue-ge** :: uexp ⇒ uexp ⇒ uexp (infix > 150)
- **ue-zero** :: uexp (0)
- **ue-one** :: uexp (1)
- **ue-plus** :: uexp ⇒ uexp ⇒ uexp (infixl + 165)
- **ue-uminus** :: uexp ⇒ uexp (− [181] 180)
- **ue-minus** :: uexp ⇒ uexp ⇒ uexp (infixl − 165)
- **ue-times** :: uexp ⇒ uexp ⇒ uexp (infixl * 170)
- **ue-div** :: uexp ⇒ uexp ⇒ uexp (infixl div 170)

translations
- **ue-num x** => -Numeral x
- **ue-size e** => # u(e)
- **ue-le x y** => x ≤ u y
- **ue-ge x y** => x ≥ u y
- **ue-ge x y** => x > u y
- **ue-zero** => 0
- **ue-one** => 1
- **ue-plus x y** => x + y
- **ue-uminus x** => - x
- **ue-minus x y** => x − y
- **ue-times x y** => x * y
- **ue-div x y** => CONST divide x y

31.5 Sets

syntax
- **ue-empset** :: uexp ({})
- **ue-setprod** :: uexp ⇒ uexp ⇒ uexp (infixr × 80)
- **ue-atLeastAtMost** :: uexp ⇒ uexp ⇒ uexp ((1 {-.-})
- **ue-atLeastLessThan** :: uexp ⇒ uexp ⇒ uexp ((1 {.-<})

translations
- **ue-empset** => {} u
- **ue-setprod e f** => CONST bop (CONST Product-Type TIMES) e f
- **ue-atLeastAtMost m n** => {m..n} u
- **ue-atLeastLessThan m n** => {m..<n} u
31.6  Imperative Program Syntax

syntax
-ue-if-then  :: uexp ⇒ logic ⇒ logic ⇒ logic (if - then - else - fi)
-ue-hoare   :: uexp ⇒ logic ⇒ uexp ⇒ logic ({{-}} / - / {{-}})
-ue-wp      :: logic ⇒ uexp ⇒ uexp (infix wp 60)

translations
-ue-if-then b P Q = > P ⊕ b ▷ Q
-ue-hoare b P c = {{b}}P{{c}}u
-ue-wp P b = > P wp b

32  Example: Summing a List

theory sum-list
  imports ../utp-easy-parser
begin
This theory exemplifies the use of the Isabelle/UTP Hoare logic verification component. We first create a state space with the variables the program needs.

alphabet st-sum-list =
  i :: nat
  xs :: int list
  ans :: int

Next, we define the program as by a homogeneous relation over the state-space type.

abbreviation Sum-List :: st-sum-list hrel where
  Sum-List ≡
  i := 0 ;;
  ans := 0 ;;
  while (i < #xs) invr (ans = list-sum(take(i, xs)))
  do
  ans := ans + xs[i] ;;
  i := i + 1
  od

Next, we symbolically evaluate some examples.

lemma TRY ([[&xs ⇒<4,3,7,1,12,8>]] |= Sum-List)
  apply (sym-eval) oops

Finally, we verify the program.

theorem Sum-List-sums:
  {{xs = <XS>}} Sum-List {{ans = list-sum(xs)}}
  by (hoare-auto, metis add.foldr-snoc take-Suc-conv-app-nth)
end

33  Simple UTP real-time theory

theory utp-simple-time imports ../utp begin
In this section we give a small example UTP theory, and show how Isabelle/UTP can be used to automate production of programming laws.

164
33.1 Observation Space and Signature

We first declare the observation space for our theory of timed relations. It consists of two variables, to denote time and the program state, respectively.

\[
\begin{align*}
\text{alphabet} & \quad \text{'s st-time} = \\
& \quad \text{clock :: nat st :: 's}
\end{align*}
\]

A timed relation is a homogeneous relation over the declared observation space.

\[
\begin{align*}
\text{type-synonym} & \quad \text{'s time-rel} = \text{'s st-time hrel}
\end{align*}
\]

We introduce the following operator for adding an \( n \) unit delay to a timed relation.

\[
\begin{align*}
\text{definition} & \quad \text{Wait :: nat} \Rightarrow \text{'s time-rel} \\
& \quad \text{where} \\
& \quad [\text{upred-defs}]: \text{Wait}(n) = (\text{clock'} =_u \text{clock} + \ll n \gg \land \text{st'} =_u \text{st})
\end{align*}
\]

33.2 UTP Theory

We define a single healthiness condition which ensures that the clock monotonically advances, and so forbids reverse time travel.

\[
\begin{align*}
\text{definition} & \quad \text{HT :: 's time-rel} \Rightarrow \text{'s time-rel} \\
& \quad \text{where} \\
& \quad [\text{upred-defs}]: \text{HT}(P) = (P \land \text{clock} \leq_u \text{clock'})
\end{align*}
\]

This healthiness condition is idempotent, monotonic, and also continuous, meaning it distributes through arbitrary non-empty infima.

\[
\begin{align*}
\text{theorem} & \quad \text{HT-idem: HT(HT(P)) = HT(P) by rel-auto} \\
\text{theorem} & \quad \text{HT-mono: P \sqsubseteq Q \Rightarrow HT(P) \sqsubseteq HT(Q) by rel-auto} \\
\text{theorem} & \quad \text{HT-continuous: Continuous HT by rel-auto}
\end{align*}
\]

We now create the UTP theory object for timed relations. This is done using a local interpretation \( \text{utp-theory-continuous HT} \). This raises the proof obligations that \( \text{HT} \) is both idempotent and continuous, which we have proved already. The result of this command is a collection of theorems that can be derived from these facts. Notably, we obtain a complete lattice of timed relations via the Knaster-Tarski theorem. We also apply some locale rewrites so that the theorems that are exports have a more intuitive form.

\[
\begin{align*}
\text{interpretation} & \quad \text{time-theory: utp-theory-continuous HT} \\
\text{rewrites} & \quad P \in \text{carrier time-theory.thy-order} \leftrightarrow P \text{ is HT} \\
\text{and} & \quad \text{carrier time-theory.thy-order} \rightarrow \text{carrier time-theory.thy-order} \equiv [\text{HT}]_H \rightarrow [\text{HT}]_H \\
\text{and} & \quad \text{le time-theory.thy-order} = (\subseteq) \\
\text{and} & \quad \text{eq time-theory.thy-order} = (=)
\end{align*}
\]

\[
\begin{align*}
\text{proof} & \quad \text{–} \\
\text{show} & \quad \text{utp-theory-continuous HT} \\
\text{proof} & \quad \text{–} \\
\text{show} & \quad \bigwedge P. \text{ HT } (\text{HT } P) = \text{ HT } P \\
& \quad \text{by (simp add: HT-idem)} \\
\text{show} & \quad \text{Continuous HT} \\
& \quad \text{by (simp add: HT-continuous)} \\
\text{qed}
\end{align*}
\]

\[
\text{proof (simp-all)}
\]

The object \( \text{time-theory} \) is a new namespace that contains both definitions and theorems. Since the theory forms a complete lattice, we obtain a top element, bottom element, and a least fixed-point constructor. We give all of these some intuitive syntax.
notation time-theory.utp-top (⊤₁)
notation time-theory.utp-bottom (⊥₁)
notation time-theory.utp-lfp (µ₁)

Below is a selection of theorems that have been exported by the locale interpretation.

thm time-theory.bottom-healthy
thm time-theory.top-higher
thm time-theory.meet-bottom
thm time-theory.LFP-unfold

33.3 Closure Laws

HT applied to Wait has no affect, since the latter always advances time.

lemma HT-Wait: HT(Wait(n)) = Wait(n) by (rel-auto)

lemma HT-Wait-closed [closure]: Wait(n) is HT
  by (simp add: HT-Wait Healthy-def)

Relational identity, II, is likewise HT-healthy.

lemma HT-skip-closed [closure]: II is HT
  by (rel-auto)

HT is closed under sequential composition, which can be shown by transitivity of (≤).

lemma HT-seqr-closed [closure]:
  [ [ P is HT; Q is HT ] ] ⇒ P ;; Q is HT
  by (rel-auto, meson dual-order.trans) — Sledgehammer required

Assignment is also healthy, provided that the clock variable is not assigned.

lemma HT-assign-closed [closure]: [ vwb-lens x; clock ⊲◁ x ] ⇒ x := v is HT
  by (rel-auto, metis (mono-tags, lifting) eq-iff lens.select-cones(1) lens-indep-get st-time.select-convs(1))

An alternative characterisation of the above is that x is within the state space lens.

lemma HT-assign-closed’ [closure]: [ vwb-lens x; x ⊆ₗ st ] ⇒ x := v is HT
  by (rel-auto)

33.4 Algebraic Laws

Finally, we prove some useful algebraic laws.

theorem Wait-skip: Wait(0) = II by (rel-auto)

theorem Wait-Wait: Wait(m) ;; Wait(n) = Wait (m + n) by (rel-auto)

theorem Wait-cond: Wait(m) ;; (P ⊲ b ⊳ₗ Q) = (Wait m ;; P) ⊲ b[&clock+<m>/&clock] ⊳ₗ (Wait m ;; Q)
  by (rel-auto)

end
Acknowledgements

This work is funded by the EPSRC projects CyPhyAssure\(^2\) (Grant EP/S001190/1), RoboCalc\(^3\) (Grant EP/M025756/1), and the Royal Academy of Engineering.

References


\(^2\)CyPhyAssure Project: https://www.cs.york.ac.uk/circus/CyPhyAssure/

\(^3\)RoboCalc Project: https://www.cs.york.ac.uk/circus/RoboCalc/


