This is a repository copy of *Isabelle/UTP: Mechanised Theory Engineering for the UTP*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/129358/

---

**Monograph:**
Foster, Simon David orcid.org/0000-0002-9889-9514, Zeyda, Frank, Nemouchi, Yakoub et al. (2 more authors) (2018) *Isabelle/UTP: Mechanised Theory Engineering for the UTP*. Working Paper. (Unpublished)

---

**Reuse**
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

**Takedown**
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.
Isabelle/UTP: Mechanised Theory Engineering for the UTP

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

April 4, 2018

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 6

2 UTP Variables 7
   2.1 Initial syntax setup ............................................. 7
   2.2 Variable foundations .............................................. 7
   2.3 Variable lens properties ......................................... 8
   2.4 Lens simplifications ............................................. 10
   2.5 Syntax translations ............................................. 10

3 UTP Expressions 13
   3.1 Expression type .................................................. 13
   3.2 Core expression constructs ..................................... 13
   3.3 Type class instantiations ....................................... 14
   3.4 Overloaded expression constructors .............................. 18
   3.5 Syntax translations ............................................. 19
   3.6 Lifting set collectors ........................................... 23
   3.7 Lifting limits .................................................... 23
   3.8 Evaluation laws for expressions ................................ 24
   3.9 Misc laws .......................................................... 24
   3.10 Literalise tactics ............................................... 25

4 Unrestriction 27
   4.1 Definitions and Core Syntax .................................. 27
   4.2 Unrestriction laws ............................................... 28

5 Used-by 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Substitution</td>
<td>32</td>
</tr>
<tr>
<td>6.1 Substitution definitions</td>
<td>33</td>
</tr>
<tr>
<td>6.2 Syntax translations</td>
<td>34</td>
</tr>
<tr>
<td>6.3 Substitution Application Laws</td>
<td>35</td>
</tr>
<tr>
<td>6.4 Substitution laws</td>
<td>38</td>
</tr>
<tr>
<td>6.5 Ordering substitutions</td>
<td>40</td>
</tr>
<tr>
<td>6.6 Unrestriction laws</td>
<td>40</td>
</tr>
<tr>
<td>6.7 Conditional Substitution Laws</td>
<td>41</td>
</tr>
<tr>
<td>6.8 Parallel Substitution Laws</td>
<td>41</td>
</tr>
<tr>
<td>7 UTP Tactics</td>
<td>42</td>
</tr>
<tr>
<td>7.1 Theorem Attributes</td>
<td>42</td>
</tr>
<tr>
<td>7.2 Generic Methods</td>
<td>42</td>
</tr>
<tr>
<td>7.3 Transfer Tactics</td>
<td>43</td>
</tr>
<tr>
<td>7.3.1 Robust Transfer</td>
<td>43</td>
</tr>
<tr>
<td>7.3.2 Faster Transfer</td>
<td>43</td>
</tr>
<tr>
<td>7.4 Interpretation</td>
<td>44</td>
</tr>
<tr>
<td>7.5 User Tactics</td>
<td>44</td>
</tr>
<tr>
<td>8 Meta-level Substitution</td>
<td>46</td>
</tr>
<tr>
<td>9 Alphabetised Predicates</td>
<td>47</td>
</tr>
<tr>
<td>9.1 Predicate type and syntax</td>
<td>47</td>
</tr>
<tr>
<td>9.2 Predicate operators</td>
<td>48</td>
</tr>
<tr>
<td>9.3 Unrestriction Laws</td>
<td>53</td>
</tr>
<tr>
<td>9.4 Used-by laws</td>
<td>55</td>
</tr>
<tr>
<td>9.5 Substitution Laws</td>
<td>55</td>
</tr>
<tr>
<td>10 UTP Events</td>
<td>58</td>
</tr>
<tr>
<td>10.1 Events</td>
<td>58</td>
</tr>
<tr>
<td>10.2 Channels</td>
<td>58</td>
</tr>
<tr>
<td>10.2.1 Operators</td>
<td>58</td>
</tr>
<tr>
<td>11 Alphabet Manipulation</td>
<td>59</td>
</tr>
<tr>
<td>11.1 Preliminaries</td>
<td>59</td>
</tr>
<tr>
<td>11.2 Alphabet Extrusion</td>
<td>59</td>
</tr>
<tr>
<td>11.3 Expression Alphabet Restriction</td>
<td>62</td>
</tr>
<tr>
<td>11.4 Predicate Alphabet Restriction</td>
<td>63</td>
</tr>
<tr>
<td>11.5 Alphabet Lens Laws</td>
<td>63</td>
</tr>
<tr>
<td>11.6 Substitution Alphabet Extension</td>
<td>64</td>
</tr>
<tr>
<td>11.7 Substitution Alphabet Restriction</td>
<td>64</td>
</tr>
<tr>
<td>12 Lifting Expressions</td>
<td>65</td>
</tr>
<tr>
<td>12.1 Lifting definitions</td>
<td>65</td>
</tr>
<tr>
<td>12.2 Lifting Laws</td>
<td>65</td>
</tr>
<tr>
<td>12.3 Substitution Laws</td>
<td>66</td>
</tr>
<tr>
<td>12.4 Unrestriction laws</td>
<td>66</td>
</tr>
</tbody>
</table>
13 Predicate Calculus Laws
  13.1 Propositional Logic ........................................ 66
  13.2 Lattice laws .................................................. 70
  13.3 Equality laws .................................................. 75
  13.4 HOL Variable Quantifiers .................................... 76
  13.5 Case Splitting ................................................ 77
  13.6 UTP Quantifiers ............................................... 78
  13.7 Variable Restriction .......................................... 79
  13.8 Conditional laws ............................................. 80
  13.9 Additional Expression Laws .................................. 81
  13.10 Refinement By Observation ................................. 82
  13.11 Cylindric Algebra ............................................ 83

14 Healthiness Conditions ........................................... 83
  14.1 Main Definitions ............................................... 84
  14.2 Properties of Healthiness Conditions ..................... 85

15 Alphabetised Relations ........................................... 89
  15.1 Relational Alphabets .......................................... 89
  15.2 Relational Types and Operators .............................. 90
  15.3 Syntax Translations .......................................... 93
  15.4 Relation Properties ......................................... 94
  15.5 Introduction laws ............................................ 95
  15.6 Unrestriction Laws ........................................... 95
  15.7 Substitution laws ............................................. 96
  15.8 Alphabet laws ................................................ 98
  15.9 Relational unrestriction ..................................... 99

16 Fixed-points and Recursion ...................................... 102
  16.1 Fixed-point Laws ............................................. 102
  16.2 Obtaining Unique Fixed-points .............................. 102
  16.3 Noetherian Induction Instantiation ......................... 104

17 UTP Deduction Tactic ............................................. 106

18 Relational Calculus Laws ........................................ 108
  18.1 Conditional Laws ............................................. 108
  18.2 Precondition and Postcondition Laws ....................... 108
  18.3 Sequential Composition Laws ................................ 109
  18.4 Iterated Sequential Composition Laws ....................... 112
  18.5 Quantale Laws ................................................ 112
  18.6 Skip Laws ................................................... 113
  18.7 Assignment Laws ............................................. 113
  18.8 Converse Laws .............................................. 115
  18.9 Assertion and Assumption Laws .............................. 116
  18.10 Frame and Anti-frame Laws ................................. 116
  18.11 While Loop Laws ........................................... 118
  18.12 Algebraic Properties ....................................... 119
      18.12.1 Kleene Star ............................................ 121
  18.13 Kleene Plus ................................................ 121
### 18.14 Omega

- Omega Algebra Laws

### 18.15 Relation Algebra Laws

- Relation Algebra Laws

### 18.16 Kleene Algebra Laws

- Kleene Algebra Laws

### 18.17 Omega Algebra Laws

- Omega Algebra Laws

### 18.18 Refinement Laws

- Refinement Laws

### 18.19 Domain and Range Laws

- Domain and Range Laws

### 19 UTP Theories

- Complete lattice of predicates
- UTP theories hierarchy
- UTP theory hierarchy
- Theory of relations
- Theory links

### 20 Relational Hoare calculus

- Hoare Triple Definitions and Tactics
- Basic Laws
- Assignment Laws
- Sequence Laws
- Conditional Laws
- Recursion Laws
- Iteration Rules
- Frame Rules

### 21 Weakest Precondition Calculus

- Weakest Precondition Calculus

### 22 Strong Postcondition Calculus

- Strong Postcondition Calculus

### 23 Concurrent Programming

- Variable Renamings
- Merge Predicates
- Separating Simulations
- Associative Merges
- Parallel Operators
- Unrestriction Laws
- Substitution laws
- Parallel-by-merge laws
- Example: Simple State-Space Division

### 24 Relational Operational Semantics

- Relational Operational Semantics

### 25 Local Variables

- Preliminaries
- State Primitives
- Relational State Spaces

### 26 Meta-theory for the Standard Core

- Meta-theory for the Standard Core
1 Introduction

This document contains the description of our mechanisation of Hoare and He’s *Unifying Theories of Programming* [14, 7] (UTP) in Isabelle/HOL. UTP uses the “programs-as-predicate” approach 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\(^1\) [13, 20, 12] 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 itself [14].

The Isabelle/UTP core mechanises most of definitions and theorems from chapters 1, 2, 4, and 7, 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 [14, 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 [13, 11], which gives a formal semantics to state spaces and variables. This, in turn, builds on a previous version of Isabelle/UTP [8, 9], 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 [10, 13] 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 limitations on Isabelle/UTP. 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. 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 [13]. For a detailed discussion of semantic embedding approaches, please see [20].

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 [13];
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;

\(^1\)Isabelle/UTP website: https://www.cs.york.ac.uk/~simonf/utp-isabelle/
6. an implementation of local variables using stacks;
7. proof tactics for the above based on interpretation [15];
8. a formalisation of UTP theories using locales [4] and building on HOL-Algebra [5];
9. Hoare logic;
10. weakest precondition and strongest postcondition calculi;
11. concurrent programming with parallel-by-merge;
12. relational operational semantics.

2 UTP Variables

theory utp-var
import
../
toolkit/utp-toolkit
utp-parser-utils
begin

In this first UTP theory we set up variables, which are are built on lenses [10, 13]. 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 ([infixl \sqcup\ ] 90 \sqcup\ 90) and
  Lattice.inf ([infixl \sqcap\ ] 90 \sqcap\ 90) and
  Lattice.join (infixl \sqcup\ ] 65 \sqcup\ 65) and
  Lattice.meet (infixl \sqcap\ ] 70 \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]
declare lens-indep-left-ext [simp]
declare lens-indep-right-ext [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 [8, 9] in this shallow model are simply represented as types 'a, 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 \to\ 'a, where the view type 'a is the variable type, and the source type 'a 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.

**definition** \( \text{in-var} :: \left('a \Rightarrow '\alpha, 'a \Rightarrow '\alpha \times '\beta \right) \)

[lens-defs]: \( \text{in-var} x = x \ ; L \ \text{fst}_L \)

**definition** \( \text{out-var} :: \left('a \Rightarrow '\beta \right) \Rightarrow \left('a \Rightarrow '\alpha \times '\beta \right) \)

[lens-defs]: \( \text{out-var} x = x \ ; L \ \text{snd}_L \)

Variables can also be used to effectively define sets of variables. Here we define 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.

**abbreviation** \((\text{input})\) \( \text{univ-alpha} :: \left('\alpha \Rightarrow '\alpha \right) \ (\Sigma) \)

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.

**definition** \( \text{pr-var} :: \left('a \Rightarrow '\beta \right) \Rightarrow \left('a \Rightarrow '\beta \right) \)

[lens-defs]: \( \text{pr-var} x = x \)

### 2.3 Variable lens properties

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

**lemma** \( \text{in-var-weak-lens} \) [simp]:

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

by (simp add: comp-weak-lens in-var-def)

**lemma** \( \text{in-var-semi-uvar} \) [simp]:

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

by (simp add: comp-mwb-lens in-var-def)

**lemma** \( \text{pr-var-weak-lens} \) [simp]:

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

by (simp add: pr-var-def)

**lemma** \( \text{pr-var-mwb-lens} \) [simp]:

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

by (simp add: pr-var-def)

**lemma** \( \text{pr-var-vwb-lens} \) [simp]:

vwb-lens \( x \Rightarrow \text{vwb-lens} \ (\text{pr-var} \ x) \)

by (simp add: pr-var-def)

**lemma** \( \text{in-var-uvar} \) [simp]:

vwb-lens \( x \Rightarrow \text{vwb-lens} \ (\text{in-var} \ x) \)

by (simp add: in-var-def)

**lemma** \( \text{out-var-weak-lens} \) [simp]:

weak-lens \( x \Rightarrow \text{weak-lens} \ (\text{out-var} \ x) \)

by (simp add: comp-weak-lens out-var-def)

**lemma** \( \text{out-var-semi-uvar} \) [simp]:

mwb-lens \( x \Rightarrow \text{mwb-lens} \ (\text{out-var} \ x) \)
by (simp add: comp-mwb-lens out-var-def)

**lemma out-var-uvar [simp]:**

\[ \text{vwb-lens } x \implies \text{vwb-lens (out } \ x) \]

by (simp add: out-var-def)

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

**lemma in-out-indep [simp]:**

\[ \text{in-var } x \bowtie \in-out \ y \]

by (simp add: lens-indep-def in-var-def out-var-def fst-lens-def snd-lens-def lens-comp-def)

**lemma out-in-indep [simp]:**

\[ \text{out-var } x \bowtie \in-out \ y \]

by (simp add: lens-indep-def in-var-def out-var-def fst-lens-def snd-lens-def lens-comp-def)

**lemma in-var-indep [simp]:**

\[ x \bowtie \in-out \ y \implies \text{in-var } x \bowtie \in-out \ y \]

by (simp add: in-var-def out-var-def)

**lemma out-var-indep [simp]:**

\[ x \bowtie \in-out \ y = \implies \text{out-var } x \bowtie \in-out \ y \]

by (simp add: out-var-def)

**lemma pr-var-indeps [simp]:**

\[ x \bowtie \in-out \ y \implies \text{pr-var } x \bowtie \in-out \ y \]

by (simp add: out-var-def pr-var-def)

**lemma prod-lens-indep-in-var [simp]:**

\[ a \bowtie \in-out \ x \implies a \times L b \bowtie \in-out \ y \]

by (metis in-var-def in-var-indep out-in-indep out-var-def plus-pres-lens-indep prod-as-plus)

**lemma prod-lens-indep-out-var [simp]:**

\[ b \bowtie \in-out \ x \implies a \times L b \bowtie \in-out \ y \]

by (metis in-out-indep in-var-def out-var-def out-var-indep plus-pres-lens-indep prod-as-plus)

**lemma in-var-pr-var [simp]:**

\[ \text{in-var (pr-var } x ) = \in-var x \]

by (simp add: pr-var-def)

**lemma out-var-pr-var [simp]:**

\[ \text{out-var (pr-var } x ) = \out-var x \]

by (simp add: pr-var-def)

**lemma pr-var-idem [simp]:**

\[ \text{pr-var (pr-var } x ) = \text{pr-var } x \]

by (simp add: pr-var-def)

**lemma pr-var-lens-plus [simp]:**

\[ \text{pr-var } (x +_L y ) = (x +_L y ) \]

by (simp add: pr-var-def)

**lemma pr-var-lens-comp-1 [simp]:**

\[ \text{pr-var } x ;_L y = \text{pr-var } (x ;_L y ) \]
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 plus-lens-distr)

lemma out-var-plus [simp]: out-var \((x +_L y)\) = out-var \(x +_L\) out-var \(y\)
by (simp add: out-var-def plus-lens-distr)

Similar properties follow for sublens

lemma in-var-sublens [simp]:
y \subseteq L x \implies \text{in-var} y \subseteq L \text{in-var} x
by (metis (no-types, hide-lams) in-var-def lens-comp-assoc sublens-def)

lemma out-var-sublens [simp]:
y \subseteq L x \implies \text{out-var} y \subseteq L \text{out-var} x
by (metis (no-types, hide-lams) out-var-def lens-comp-assoc sublens-def)

lemma pr-var-sublens [simp]:
y \subseteq L x \implies \text{pr-var} y \subseteq L \text{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)

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 \\Rightarrow svid (\cdot [999] 999)
- svid-unit :: svid \\Rightarrow svids (\cdot)
- svid-list :: svid \Rightarrow svids \Rightarrow svids (\cdot, / \cdot)
- svid-alpha :: svid (v)
- svid-dot :: svid \Rightarrow svid \Rightarrow svid (\cdot:: [998,999] 998)
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` (infixr ; 75)
- `salphacomp` :: `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` (\{\})
- `svar` :: `svar ⇒ logic` (´)

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**

- `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

— Decorations
- spvar Σ ↦ CONST svar CONST id-lens
- sinvar Σ ↦ CONST ivar 1L
- soutvar Σ ↦ CONST ovar 1L
- 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) z ↦ -svid-dot (CONST lens-comp y x) z

- svar x = CONST svar x
- sinvar x = CONST ivar x
- soutvar x = CONST ovar x

— Alphabets
- salphaparen a ↦ a
- salphaid x ↦ x
- salphacomp 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 = 1L
- salpha-none = 0L

— Quotations
- ualpha-set A → A
- svar x ↦ x

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 lens 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.

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
}
3 UTP Expressions

theory utp-expr
imports utp-var
begin

3.1 Expression type

Purge-notation BNF-Def.convol (\langle (-,/ -) \rangle)

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 'a. 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 \[15\], 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]

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

The term \( [\[ e \] e b \)

The term \( [\[ e \] e b \)

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.

lift-definition var :: ('t ⇒ 'α) ⇒ ('t, 'α) uexpr is \texttt{lens-get}.

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

lift-definition lit :: 't ⇒ ('t, 'α) uexpr is λ v b. 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.

lift-definition uop :: ('a ⇒ 'b) ⇒ ('a, 'α) uexpr ⇒ ('b, 'α) uexpr is λ f e b. f (e b).

lift-definition bop ::
  ('a ⇒ 'b ⇒ 'c) ⇒ ('a, 'α) uexpr ⇒ ('b, 'α) uexpr ⇒ ('c, 'α) uexpr
is \( \lambda f \ u \ v \ b \).  

**lift-definition** trop ::
\[
(a \Rightarrow b \Rightarrow c \Rightarrow d) \Rightarrow (a, \alpha \ uexpr \Rightarrow (b, \alpha \ uexpr \Rightarrow (c, \alpha \ uexpr \Rightarrow (d, \alpha \ uexpr
\]
is \( \lambda f \ u \ v \ w \ b \).  

**lift-definition** qtop ::
\[
(a \Rightarrow b \Rightarrow c \Rightarrow d \Rightarrow e) \Rightarrow (a, \alpha \ uexpr \Rightarrow (b, \alpha \ uexpr \Rightarrow (c, \alpha \ uexpr \Rightarrow (d, \alpha \ uexpr \Rightarrow (e, \alpha \ uexpr
\]
is \( \lambda f \ u \ v \ w \ x \ b \).  

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

**lift-definition** ulambda ::
\[
(a \Rightarrow b \Rightarrow c \Rightarrow d \Rightarrow e) \Rightarrow (a \Rightarrow b, \alpha \ uexpr \Rightarrow (c \Rightarrow d, \alpha \ uexpr \Rightarrow (e, \alpha \ uexpr
\]
is \( \lambda f \ A \ x \).  

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

**abbreviation** cond ::
\[
(a, \alpha \ uexpr \Rightarrow (boolean, \alpha \ uexpr \Rightarrow (a, \alpha \ uexpr \Rightarrow (a, \alpha \ uexpr
\]

**definition** eq-upred ::
\[
(a, \alpha \ uexpr \Rightarrow (a, \alpha \ uexpr \Rightarrow (true, \alpha \ uexpr
\]

We define syntax for expressions using adhoc-overloading – this allows us to later define operators on different types if necessary (e.g. when adding types for new UTP theories).

**consts**
\[
lit :: t \Rightarrow e (\approx \rightarrow)
\]
\[
eq :: a \Rightarrow a \Rightarrow boolean (\infixl = 50)
\]

**adhoc-overloading**
\[
lit \lit \ and
\]
\[
eq \equpred
\]

A literal is the expression \( \langle v \rangle \), where \( v \) 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.

**syntax**
\[
-uuvar :: svar \Rightarrow logic (-)
\]

**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.

\textbf{instantiation \textit{uexpr} :: (zero, type) zero}
begin
\textbf{definition} zero-uexpr-def: \(0 = \text{lit} \ 0\)
instance ..
end

\textbf{instantiation \textit{uexpr} :: (one, type) one}
begin
\textbf{definition} one-uexpr-def: \(1 = \text{lit} \ 1\)
instance ..
end

\textbf{instantiation \textit{uexpr} :: (plus, type) plus}
begin
\textbf{definition} plus-uexpr-def: \(u + v = \text{bop} \ (\text{op} +) \ u \ v\)
instance ..
end

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

\textbf{instantiation \textit{uexpr} :: (uminus, type) uminus}
begin
\textbf{definition} uminus-uexpr-def: \(−u = \text{uop} \ \text{uminus} \ u\)
instance ..
end

\textbf{instantiation \textit{uexpr} :: (minus, type) minus}
begin
\textbf{definition} minus-uexpr-def: \(u - v = \text{bop} \ (\text{op} -) \ u \ v\)
instance ..
end

\textbf{instantiation \textit{uexpr} :: (times, type) times}
begin
\textbf{definition} times-uexpr-def: \(u \ast v = \text{bop} \ (\text{op} \ast) \ u \ v\)
instance ..
end

\textbf{instance \textit{uexpr} :: (Rings.dvd, type) Rings.dvd ..}

\textbf{instantiation \textit{uexpr} :: (divide, type) divide}
begin
\textbf{definition} divide-uexpr :: (\texttt{a}, \texttt{b}) \textit{uexpr} ⇒ (\texttt{a}, \texttt{b}) \textit{uexpr} ⇒ (\texttt{a}, \texttt{b}) \textit{uexpr} \ \textbf{where}
\hspace{1cm} divide-uexpr \ u \ v = \text{bop} \ \text{divide} \ u \ v
instance ..
end

\textbf{instantiation \textit{uexpr} :: (inverse, type) inverse}
begin
\textbf{definition} inverse-uexpr :: (\texttt{a}, \texttt{b}) \textit{uexpr} ⇒ (\texttt{a}, \texttt{b}) \textit{uexpr} \ \textbf{where}
\hspace{1cm} inverse-uexpr \ u = \text{uop} \ \text{inverse} \ u

instance ..
end

instantiation $uexpr$ :: (modulo, type) modulo
begin
definition mod-$uexpr$-def: $u \mod v = \text{bop\ (op\ mod)}\ u\ v$
instance ..
end

instantiation $uexpr$ :: (sgn, type) sgn
begin
definition sgn-$uexpr$-def: $\text{sgn}\ u = \text{uop\ sgn}\ u$
instance ..
end

instantiation $uexpr$ :: (abs, type) abs
begin
definition abs-$uexpr$-def: $\text{abs}\ u = \text{uop\ abs}\ u$
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 $uexpr$ :: (semigroup-add, type) semigroup-add
  by (intro-classes) (simp add: plus-$uexpr$-def zero-$uexpr$-def, transfer, simp add: add.assoc)+

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: fun-eq-iff)+

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

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

We can also define the order relation on expressions. Now, unlike the previous group and ring
constructs, the order relations $\leq$ and $\leq$ return a $\textit{bool}$ 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.

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-uexpr :: ('a, 'b) uexpr ⇒ ('a, 'b) uexpr ⇒ bool
where less-uexpr P Q = (P ≤ Q ∧ ¬ Q ≤ 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 :: ('a, 'b) uexpr
  show $((x < y) = (x \leq y \land \neg y \leq x)$ by (simp add: less-uexpr-def)
  show $x \leq x$ 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

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-uexpr-def, transfer, simp)

instance uexpr :: (ordered-ab-group-add-abs, type) ordered-ab-group-add-abs
apply (intro-classes)
  apply (simp add: abs-uexpr-def zero-uexpr-def plus-uexpr-def minus-uexpr-def times-uexpr-def
  transfer, simp add: abs-ge-self abs-le-iff abs-triangle-inq)+
apply (metis ab-group-add-class ab-diff-cone-add-uminus abs-ge-minus-self abs-ge-self add-mono-thms-
linordered-semiring)+
done

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

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

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: $[[\text{numeral } x]]_e \ b = \text{numeral } x$
apply (induct x)
  apply (simp add: lit.rep-eq one-uexpr-def)
  apply (simp add: bop.rep-eq numeral-Bit0 plus-uexpr-def)
  apply (simp add: bop.rep-eq lit.rep-eq numeral-code(3) one-uexpr-def plus-uexpr-def)
done

lemma numeral-uexpr-simp: numeral x = ≪numeral x≫
  by (simp add: uexpr-eq-iff numeral-uexpr-rep-eq lit.rep-eq)

The next theorem lifts powers.

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

We can also lift a few trace properties from the class instantiations above using transfer.

lemma uexpr-diff-zero [simp]:
  fixes a :: ('α::trace, 'a) uexpr
  shows a - 0 = a
  by (simp add: minus-uexpr-def zero-uexpr-def, transfer, auto)

lemma uexpr-add-diff-cancel-left [simp]:
  fixes a b :: ('α::trace, 'a) uexpr
  shows (a + b) - a = b
  by (simp add: minus-uexpr-def plus-uexpr-def, transfer, auto)

3.4 Overloaded expression constructors

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** \( \text{fun-apply} :: (\text{'}a \Rightarrow \text{'}b) \Rightarrow (\text{'}a \Rightarrow \text{'}b) \)

**Where** \( \text{fun-apply} \ f \ x = f \ x \)

**Declare** \( \text{fun-apply-def} \ [\text{simp}] \)

**Definition** \( \text{ffun-entries} :: (\text{'}k \set \Rightarrow (\text{'}k \Rightarrow \text{'}v)) \Rightarrow (\text{'}k \set, \text{'}v) \)

**Where** \( \text{ffun-entries} \ d \ f = \text{graph-ffun} \ \{(k, f k) \mid k \in d\} \)

We then set up the overloading for a number of useful constructs for various collections.

**Adhoc-overloading**

- \( \text{uempty} 0 \) **and**
- \( \text{uapply} \ \text{fun-apply} \) **and**
- \( \text{uapply} \ \text{nth} \) **and**
- \( \text{uapply} \ \text{pfun-app} \) **and**
- \( \text{uapply} \ \text{ffun-app} \) **and**
- \( \text{uupd} \ \text{pfun-upd} \) **and**
- \( \text{uupd} \ \text{ffun-upd} \) **and**
- \( \text{uupd} \ \text{list-augment} \) **and**
- \( \text{udom} \ \text{Domain} \) **and**
- \( \text{udom} \ \text{pdom} \) **and**
- \( \text{udom} \ \text{fdom} \) **and**
- \( \text{udom} \ \text{seq-dom} \) **and**
- \( \text{urannres} \ \text{pran-res} \) **and**
- \( \text{urannres} \ \text{udomres} \) **and**
- \( \text{urannres} \ \text{fran-res} \) **and**
- \( \text{ucard} \ \text{card} \) **and**
- \( \text{ucard} \ \text{pcard} \) **and**
- \( \text{ucard} \ \text{length} \) **and**
- \( \text{usums} \ \text{list-sum} \) **and**
- \( \text{usums} \ \text{Sum} \) **and**
- \( \text{usums} \ \text{pfun-sum} \) **and**
- \( \text{uentries} \ \text{pfun-entries} \) **and**
- \( \text{uentries} \ \text{ffun-entries} \)

### 3.5 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** \( \text{(input)} \ \text{ulens-override} \ x \ f \ g \equiv \text{lens-override} \ f \ g \ x \)

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

**Definition** \( \text{set-of} :: \text{'}a \ \text{itself} \Rightarrow \text{'}a \ \text{set} \)

**Where** \( \text{set-of} \ t = \text{UNIV} \)

**Translations**

\( 0 \leq \ CONST \ \text{uempty} \) — We have to do this so we don’t see \text{uempty}. Is there a better way of printing?

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

**Nonterminal** \( \text{atuple-args} \) **and** \( \text{umaplet} \) **and** \( \text{umaplets} \)

**Syntax** — Core expression constructs

- \( \text{ucoerce} :: \text{logic} \Rightarrow \text{type} \Rightarrow \text{logic} \) **(infix** \( :_u \ 50)\)
- \( \text{ulambda} :: \text{pttrn} \Rightarrow \text{logic} \Rightarrow \text{logic} \) **(\( \lambda \cdot \cdot \cdot [0, 10] 10\))\)
- \( \text{ulens-ovrd} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{salpha} \Rightarrow \text{logic} \) **(\( \circ \cdot \cdot \cdot \text{on} \cdot [85, 0, 86] 86\))\)
- \( \text{ulens-get} :: \text{logic} \Rightarrow \text{svar} \Rightarrow \text{logic} \) **(\( \cdot :- [900,901] 901\))\)

**Translations**

\( \lambda \ x \cdot p \equiv \ CONST \ \text{ulambda} (\lambda \ x. p) \)
\( x :_u a == x :: (\text{',', } a) \) **uexpr**
\( \text{ulens-ovrd} \ f \ g \ a \Rightarrow \ CONST \ \text{bop} (\CONST \ \text{ulens-override} \ a) \ f \ g \)
\( \text{ulens-ovrd} \ f \ g \ a \subset \ CONST \ \text{bop} (\lambda x \ y. \ \CONST \ \text{lens-override} \ x1 \ y1 \ a) \ f \ g \)
\( \text{ulens-get} \ x \ y == \ CONST \ \text{wop} (\CONST \ \text{lens-get} \ y) \ x \)

**Syntax** — Tuples
translations

\( ()_u \mapsto \text{CONST bop (CONST Pair) x y} \)

-utuple \( (x, y)_u \mapsto \text{CONST bop (CONST Pair) x y} \)

\( \pi_1(x) \mapsto \text{CONST uop CONST fst x} \)

\( \pi_2(x) \mapsto \text{CONST uop CONST snd x} \)

translations — Pretty printing for adhoc-overloaded constructs

\( f(x)_a \mapsto \text{CONST uapply } f \ x \)

\( \text{dom}_u(f) \mapsto \text{CONST udom } f \)

\( \text{ran}_u(f) \mapsto \text{CONST uran } f \)

\( A \prec_u f \mapsto \text{CONST udomres A } f \)

\( f \succ_u A \mapsto \text{CONST uranres } f \ A \)

\( \#_u(f) \mapsto \text{CONST ucard } f \)

\( f(k \mapsto v)_u \mapsto \text{CONST uupdate } f \ k \ v \)

— Overloaded construct translations

\( f(x,y,z,u)_u \mapsto \text{CONST bop CONST uapply } f \ (x,y,z,u)_u \)

\( f(x,y,z)_u \mapsto \text{CONST bop CONST uapply } f \ (x,y,z)_u \)

\( f(x,y)_u \mapsto \text{CONST bop CONST uapply } f \ (x,y)_u \)

\( f(x)_u \mapsto \text{CONST bop CONST uapply } f \ x \)
#_u(xs) == CONST uop CONST uward xs
sum_u(A) == CONST uop CONST usums A
dom_u(f) == CONST uop CONST udom f
run_u(f) == CONST uop CONST uran f
\[ \square_u == \langle \text{CONST empty} \rangle \]
\[ \perp_u == \langle \text{CONST undefined} \rangle \]
A ⪯_u f == CONST bop (CONST udomres) A f
f ⪰_u A == CONST bop (CONST uranres) f A
entr_u(d,f) == CONST bop CONST uentries d \langle f \rangle
-\text{UMapUpd} m (-\text{UMaplets} xy ms) == -\text{UMapUpd} (-\text{UMapUpd} m xy) ms
-\text{UMapUpd} m (-\text{umaplet} x y) == \text{CONST trop} CONST uwdp m x y
-\text{UMap} ms == -\text{UMapUpd} \square_u ms
-\text{UMap} (-\text{UMaplets} ms1 ms2) <= -\text{UMapUpd} (-\text{UMap} ms1) ms2
-\text{UMaplets} ms1 (-\text{UMaplets} ms1 ms3) <= -\text{UMaplets} (-\text{UMaplets} ms1 ms2) ms3

— Type-class polymorphic constructs
\[ x <_u y == \text{CONST bop} (\text{op} <) x y \]
\[ x \leq_u y == \text{CONST bop} (\text{op} \leq) x y \]
\[ x >_u y == \Rightarrow y <_u x \]
\[ x \geq_u y == \Rightarrow y \leq_u x \]
\[ \text{min}_u(x, y) == \text{CONST bop} (\text{CONST min}) x y \]
\[ \text{max}_u(x, y) == \text{CONST bop} (\text{CONST max}) x y \]
\[ \text{gcd}_u(x, y) == \text{CONST bop} (\text{CONST gcd}) x y \]
\[ \lceil x \rceil_u == \text{CONST uop} \text{CONST ceiling} x \]
\[ \lfloor x \rfloor_u == \text{CONST uop} \text{CONST floor} x \]

**syntax** — Lists / Sequences

-\text{unil} :: ('a list, 'a) uexpr (\langle \rangle)
-\text{ulist} :: args => ('a list, 'a) uexpr ((\langle\langle\rangle\rangle))
-\text{uappend} :: ('a list, 'a) uexpr => ('a list, 'a) uexpr => ('a list, 'a) uexpr (infixr \_u \_80)
-\text{ulast} :: ('a list, 'a) uexpr => ('a, 'a) uexpr (last_u(''))
-\text{ufront} :: ('a list, 'a) uexpr => ('a list, 'a) uexpr (front_u(''))
-\text{uhead} :: ('a list, 'a) uexpr => ('a, 'a) uexpr (head_u(''))
-\text{utail} :: ('a list, 'a) uexpr => ('a list, 'a) uexpr (tail_u(''))
-\text{uuptake} :: (nat, 'a) uexpr => ('a list, 'a) uexpr => ('a list, 'a) uexpr (take_u(''-\'\'-'))
-\text{udrop} :: (nat, 'a) uexpr => ('a list, 'a) uexpr => ('a list, 'a) uexpr (drop_u(''-\'\'-'))
-\text{ufilter} :: ('a list, 'a) uexpr => ('a set, 'a) uexpr => ('a list, 'a) uexpr (infixl \_u \_75)
-\text{uextract} :: ('a set, 'a) uexpr => ('a list, 'a) uexpr => ('a list, 'a) uexpr (infixl \_u \_75)
-\text{uelems} :: ('a list, 'a) uexpr => ('a set, 'a) uexpr (elems_u(''))
-\text{usorted} :: ('a list, 'a) uexpr => (bool, 'a) uexpr (sorted_u(''))
-\text{udistinct} :: ('a list, 'a) uexpr => (bool, 'a) uexpr (distinct_u(''))
-\text{uupto} :: logic => logic => logic ((\langle\langle\langle\rangle\rangle\rangle))
-\text{uupt} :: logic => logic => logic ((\langle\langle\langle\rangle\rangle\rangle\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle))
-\text{umap} :: logic => logic => logic (map_u)
-\text{uzip} :: logic => logic => logic (zip_u)
-\text{utr-iter} :: logic => logic => logic (iter[-\langle\langle\langle\rangle\rangle\rangle\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle\langle\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle\rangle\langle\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle\rangle\langle\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle\rangle\langle\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle\rangle))

**translations**
\[ \langle \rangle == \langle \rangle\]
\[ \langle x, xs \rangle == \text{CONST bop} (\text{op #}) x \langle xs \rangle \]
\[ \langle x \rangle == \text{CONST bop} (\text{op #}) x \langle \rangle\]
\[ x \_u \_y == \text{CONST bop} (\text{op @}) x \_y \]
\[ \text{last}_u(xs) == \text{CONST uop} \text{CONST last} xs \]
\[ \text{front}_u(xs) == \text{CONST uop} \text{CONST butlast} xs \]
\[ \text{head}_u(xs) == \text{CONST uop} \text{CONST hd} xs \]
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 (op |_i) A xs
⟨n..k⟩ == CONST bop CONST upto n k
⟨n..<k⟩ == CONST bop CONST upto n k
map_u f xs == CONST bop CONST map f xs
zip_u xs ys == CONST bop CONST zip xs ys
\text{iter}[n](P) == CONST uop (CONST \text{tr-iter } n) P

\text{syntax} — Sets
- _\text{ufinite} :: \text{logic} \Rightarrow \text{logic} (\text{finite}_u(\cdot))
- _\text{uempset} :: (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} (\{\cdot\}) \rangle) \text{ uexpr}
- _\text{uset} :: \text{args} \Rightarrow (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} (\{\cdot\}) \rangle)
- _\text{uunion} :: (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} \Rightarrow (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} \Rightarrow (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} (\text{infixl} \cup_u 65))))
- _\text{uinter} :: (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} \Rightarrow (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} \Rightarrow (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} (\text{infixl} \cap_u 70))))
- _\text{umem} :: (\langle \text{a}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} \Rightarrow (\langle \text{a}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} \Rightarrow (\text{bool}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} (\text{infix} \subseteq_u 50))))
- _\text{usubseteq} :: (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} \Rightarrow (\langle \text{a set}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} \Rightarrow (\text{bool}, \langle \text{\cdot}, \text{\cdot} \rangle \text{ uexpr} (\text{infix} \subseteq_u 50))))
- _\text{uconverse} :: \text{logic} \Rightarrow \text{logic} ([\cdot]_T [1000] 999)
- _\text{uoid} :: \text{type} \Rightarrow \text{logic} (\text{iden}[\cdot])
- _\text{uproduct} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} (\text{infixr} \times_u 80)
- _\text{urelcomp} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} (\text{infixr} :_u 75)

\text{translations}
\text{finite}_u(x) == CONST uop (CONST \text{finite}) x
\{\cdot\}_u == \langle \{\cdot\} \rangle
\{x, xs\}_u == CONST bop (CONST \text{insert}) x \{xs\}_u
\{x\}_u == CONST bop (CONST \text{insert}) x \langle \{\cdot\} \rangle
A \cup_u B == CONST bop (op \cup) A B
A \cap_u B == CONST bop (op \cap) A B
x \in_u A == CONST bop (op \in) x A
A \subseteq_u B == CONST bop (op \subseteq) A B
f \subseteq_u g == CONST bop (op \subseteq_p) f g
A \subseteq_u B == CONST bop (op \subseteq) A B
P'' == CONST uop CONST converse P
\text{\textquoteleft}a\textquoteright]_T == \langle \text{CONSTR set-of \text{TYPE}(\textquoteleft}a\textquoteright) \rangle
\text{\textquoteleft}u\textquoteright]_T == \langle \text{CONSTR \text{Id-on} (CONST set-of \text{TYPE}(\textquoteleft}a\textquoteright)) \rangle
A \times_u B == CONST bop CONST Product-Type.Times A B
A ::_u B == CONST bop CONST relcomp A B

\text{syntax} — Partial functions
- _\text{umap-plus} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} (\text{infixl} \oplus_u 85)
- _\text{umap-minus} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} (\text{infixl} \ominus_u 85)

\text{translations}
f \ominus_u g =\Rightarrow (f :: (\langle \cdot, \cdot \rangle pfun, \cdot) \text{ uexpr}) + g
3.6 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**

- `uset-atLeastAtMost :: (a, α) uexpr ⇒ (a, α) uexpr ⇒ (a set, α) uexpr ((I{⋯}u))`
- `uset-atLeastLessThan :: (a, α) uexpr ⇒ (a, α) uexpr ⇒ (a set, α) uexpr ((I{⋯<}u))`
- `uset-compr :: pttrn ⇒ (a set, α) uexpr ⇒ (bool, α) uexpr ⇒ (b, α) uexpr ⇒ (b set, α) uexpr ((I{⋯|⋯/⋯}u))`
- `uset-compr-nset :: pttrn ⇒ (bool, α) uexpr ⇒ (b, α) uexpr ⇒ (b set, α) uexpr ((I{⋯|⋯/⋯}u))`

**lift-definition**

**ZedSetCompr ::**

\[(\lambda x . P x . b | x . x \in A \land \text{fst} (PF x) b)\].

**translations**

\[\{x . y\} u == \text{CONST bop CONST atLeastAtMost } x y\]
\[\{x . <y\} u == \text{CONST bop CONST atLeastLessThan } x y\]
\[\{x : P \cdot F\} u == \text{CONST ZedSetCompr (CONST ulit CONST UNIV) } (\lambda x . (P, F))\]
\[\{x : A \mid P \cdot F\} u == \text{CONST ZedSetCompr A } (\lambda x . (P, F))\]

3.7 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::t2-space where ulim-left = (λ p f . Lim (at-left p) f)`

**definition**

`ulim-right :: 'a::order-topology ⇒ 'a ⇒ 'b::t2-space where ulim-right = (λ p f . Lim (at-right p) f)`

**definition**

`ucont-on :: ('a::topological-space ⇒ 'b::topological-space) ⇒ 'a set ⇒ bool where ucont-on = (λ f A . continuous-on A f)`

**syntax**

- `ulim-left :: id ⇒ logic ⇒ logic ⇒ logic (lim u (\cdot \to \cdot\cdot\cdot)(\cdot))`
- `ulim-right :: id ⇒ logic ⇒ logic ⇒ logic (lim u (\cdot \to \cdot\cdot\cdot)(\cdot))`
- `ucont-on :: logic ⇒ logic ⇒ logic (infix cont-on u 90)`

**translations**

\[\text{lim u}(x \to p^\cdot)(e) == \text{CONST bop CONST ulim-left p (λ x \cdot e)}\]
\[\text{lim u}(x \to p^\cdot)(e) == \text{CONST bop CONST ulim-right p (λ x \cdot e)}\]
\[f \text{ cont-on u A} == \text{CONST bop CONST continuous-on A f}\]
3.8 Evaluation laws for expressions

We now collect together all the definitional theorems for expression constructs, and use them to build an evaluation strategy for expressions that we will later use to construct proof tactics for UTP predicates.

lemmas uexpr-defs =
zero-uexpr-def
one-uexpr-def
plus-uexpr-def
uminus-uexpr-def
minus-uexpr-def	imes-uexpr-def
inverse-uexpr-def
divide-uexpr-def
sgn-uexpr-def
abs-uexpr-def
mod-uexpr-def
eq-upred-def
numeral-uexpr-simp
ulim-left-def
ulim-right-def
ucont-on-def

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]: 
\[ \var x b = \text{get}_x b \]
by (transfer, simp)

lemma uop-ueval [ueval]: 
\[ \uop f x_e b = f (\langle x \rangle_e b) \]
by (transfer, simp)

lemma bop-ueval [ueval]: 
\[ \bop f x y_e b = f (\langle x \rangle_e b) (\langle y \rangle_e b) \]
by (transfer, simp)

lemma trop-ueval [ueval]: 
\[ \trop f x y z_e b = f (\langle x \rangle_e b) (\langle y \rangle_e b) (\langle z \rangle_e b) \]
by (transfer, simp)

lemma qtop-ueval [ueval]: 
\[ \qtop f x y z w_e b = f (\langle x \rangle_e b) (\langle y \rangle_e b) (\langle z \rangle_e b) (\langle w \rangle_e b) \]
by (transfer, simp)

We also add all the definitional expressions to the evaluation theorem set.

declare uexpr-defs [ueval]

3.9 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 (λx y. y) u v = v`
by (transfer, simp)

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

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

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

lemma `uunion-empty-1` [simp]: `{}` u ∪ 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 `uset-minus-empty` [simp]: `x - `{}` = x
by (simp add: uexpr-defs, transfer, simp)

lemma `ulist-filter-empty` [simp]: `x |{}` = {}
by (transfer, simp)

lemma `tail-cons` [simp]: `tail (⟨x | u xs) = xs`
by (transfer, simp)

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

lemma `iter-0` [simp]: `iter[0](t) = ⟨⟩`
by (transfer, simp add: zero-list-def)

lemma `ufun-apply-lit` [simp]: 
`⟨f⟩(⟨x⟩) = ⟨f(x)⟩`
by (transfer, simp)

3.10 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-zero` [lit-simps]: `0 = 0`
by (simp add: ueval)

lemma `lit-one` [lit-simps]: `1 = 1`
by (simp add: ueval)

lemma `lit-numeral` [lit-simps]: `numeral n = numeral n`
by (simp add: ueval)

lemma `lit-uminus` [lit-simps]: `<− x> = −<x> by (simp add: ueval, transfer, simp)

lemma `lit-plus` [lit-simps]: `<x + y> = <x> + <y> by (simp add: ueval, transfer, simp)

lemma `lit-minus` [lit-simps]: `<x − y> = <x> − <y> by (simp add: ueval, transfer, simp)

lemma `lit-times` [lit-simps]: `<x * y> = <x> * <y>`
by (simp add: ueval, transfer, simp)

lemma `lit-divide` [lit-simps]: `<x / y> = <x> / <y>`
by (simp add: ueval, transfer, simp)

lemma `lit-div` [lit-simps]: `<x div y> = <x> div <y>`
by (simp add: ueval, transfer, simp)

lemma `lit-power` [lit-simps]: `<x ^ n> = <x> ^ n`
by (simp add: lit-rep-eq power-rep-eq uexpr-eq-iff)
lemma lit-plus-appl [lit-norm]: \( (\text{op})_a(x)(y)_a = x + y \) by (simp add: ueval, transfer, simp)
lemma lit-minus-appl [lit-norm]: \( (\text{op})_a(x)(y)_a = x - y \) by (simp add: ueval, transfer, simp)
lemma lit-mul-appl [lit-norm]: \( (\text{op})_a(x)(y)_a = x * y \) by (simp add: ueval, transfer, simp)
lemma lit-divide-apply [lit-norm]: \( (\text{op})_a(x)(y)_a = x / y \) by (simp add: ueval, transfer, simp)

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

In general unliteralising converts function applications to corresponding expression liftings. Since some operators, like + and *, have specific operators we also have to use \( \theta = \emptyset \)
\( I = \langle \i \ :: \ 'a \rangle \)
\( ?u + ?v = \text{bop } op + ?u \ ?v \)
\(- ?u = \text{uop } \text{uminus } ?u \)
\(- ?u - ?v = \text{bop } op - ?u \ ?v \)
\(- ?u * ?v = \text{bop } op * ?u \ ?v \)
\( \text{inverse } ?u = \text{uop } \text{inverse } ?u \)
\( ?u \text{ div } ?v = \text{bop } op \text{ div } ?u \ ?v \)
\( \text{sgn } ?u = \text{uop } \text{sgn } ?u \)
\( |?u| = \text{uop } \text{abs } ?u \)
\( ?u \text{ mod } ?v = \text{bop } op \text{ mod } ?u \ ?v \)
\( (?x =_a ?y) = \text{bop } op = ?x \ ?y \)
\( \text{numeral } ?x = \langle \text{numeral } ?x \rangle \)
\( \text{ulim-left} = (\lambda p. \text{Lim } (\text{at-left } p)) \)
\( \text{ulim-right} = (\lambda p. \text{Lim } (\text{at-right } p)) \)
\( \text{ucont-on} = (\lambda f A. \text{continuous-on } A f) \) 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.
lemma lit-numeral-1: \( \text{uop } \text{numeral } x = \text{Abs-uxpr } (\lambda b. \text{numeral } ([x]_c b)) \)
by (simp add: uop-def)
lemma lit-numeral-2: \( \text{Abs-uxpr } (\lambda b. \text{numeral } v) = \text{numeral } v \)
by (metis lit.abs-eq lit-numeral)

method literalise = (unfold lit-simps[THEN sym])
method unliteralise = (unfold lit-simps uexpr-defs[THEN sym];
(unfold lit-numeral-1 ; (unfold 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 uexpr-simp uses simps = ((literalise)?, simp add: lit-norm simps, (unliteralise)?)

lemma \( (\i :: (\text{int }, 'a) \text{ uexpr}) + <2> = 4 \leftrightarrow <3> = 4 \)
apply (uexpr-simp) oops
4 Unrestriction

theory utp-unrest
  imports utp-expr
begin

4.1 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 [19, 18] 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 [8] and Oliveira’s [18] 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.

consts
\( \textbf{unrest} :: 'a \Rightarrow 'b \Rightarrow \text{bool} \)

syntax
\( -\textbf{unrest} :: \text{salpha} \Rightarrow \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \) (infix \( \# \) 20)

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

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.

named-theorems \textbf{unrest}

method \textbf{unrest-tac} = (simp add: \textbf{unrest})?

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.

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

adhoc-overloading

\textbf{unrest \ textbf{unrest-uexpr}}

lemma \textbf{unrest-expr-alt-def}:
\( \text{weak-lens} \ x \Longrightarrow (x \# P) = (\forall \ b \ b'. [P]_e (b \oplus \text{L} b' \ on \ x) = [P]_e b) \)
by (transfer, metis \text{weak-lens-override-def} \text{weak-lens.put-get})
4.2 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 \textit{mwb-lens} and \textit{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]:
  \[[ x \# P; y \# P \] \implies x;y \# P
  by (transfer, simp add: lens-defs)
\end{verbatim}

\begin{verbatim}
lemma unrest-svar [unrest]: (\&x \# P) \longleftrightarrow (x \# 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.

\begin{verbatim}
lemma unrest-lit [unrest]: x \# \lll v \rrr
  by (transfer, simp)
\end{verbatim}

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

\begin{verbatim}
lemma unrest-sublens:
  fixes P :: ('a, 'a) uexpr
  assumes x \# P y \subseteq_L x
  shows y \# P
  using assms
  by (transfer, metis (no-types, lifting) lens.select-cones(2) lens-comp-def sublens-def)
\end{verbatim}

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

\begin{verbatim}
lemma unrest-equiv:
  fixes P :: ('a, 'a) uexpr
  assumes mwb-lens y x \approx_L y x \# P
  shows y \# P
  by (metis assms lens-equiv-def sublens-pres-mwb sublens-put-put unrest-uexpr.rep-eq)
\end{verbatim}

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

\begin{verbatim}
lemma bij-lens-unrest-all:
  fixes P :: ('a, 'a) uexpr
  assumes bij-lens X X \# P
  shows \Sigma \# P
  using assms
  by (blast)
\end{verbatim}

\begin{verbatim}
lemma bij-lens-unrest-all-eq:
  fixes P :: ('a, 'a) uexpr
  assumes bij-lens X
  shows (\Sigma \# P) \longleftrightarrow (X \# P)
  by (meson assms bij-lens-equiv-id lens-equiv-def unrest-sublens)
\end{verbatim}

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

\begin{verbatim}
lemma unrest-all-var:
  fixes e :: ('a, 'a) uexpr
\end{verbatim}
assumes $\Sigma \models e$
shows $x \not\models e$
by (metis assms id-lens-def lens.simps(2) unrest-uxpr.rep-eq)

We can split an unrestriction composed by lens plus

**lemma unrest-plus-split:**
fixes $P :: (\alpha, \alpha) \mathit{uxpr}$
assumes $x \not\gg y \mathit{vwb-lens} x \mathit{vwb-lens} y$
shows $\mathit{unrest} (x + L y) P \leftrightarrow (x \not\models P) \land (y \not\models P)$
using assms
by (meson lens-plus-right-sublens lens-plus-ub sublens-refl unrest-subl ens unrest-var-comp vwb-lens-wb)

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]:** $[\mathit{mwb-lens} x; x \not\gg y] \implies y \not\models \mathit{var} x$
by (transfer, auto)

**lemma unrest-iuvar [unrest]:** $[\mathit{mwb-lens} x; x \not\gg y] \implies \not\models y \mathit{\#} \mathit{x}$
by (simp add: unrest-var)

**lemma unrest-ouvar [unrest]:** $[\mathit{mwb-lens} x; x \not\gg y] \implies \not\models y' \mathit{\#} \mathit{x}'$
by (simp add: unrest-var)

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

**lemma unrest-iuvar-ouvar [unrest]:**
fixes $x :: (\alpha \rightarrow \alpha)$
assumes $\mathit{mwb-lens} y$
shows $\not\models x \mathit{\#} \mathit{y}'$
by (metis prod.collapse unrest-uxpr.rep-eq var.rep-eq var-lookup-out var-update-in)

**lemma unrest-ouvar-iuvar [unrest]:**
fixes $x :: (\alpha \rightarrow \alpha)$
assumes $\mathit{mwb-lens} y$
shows $\not\models x' \mathit{\#} \mathit{y}$
by (metis prod.collapse unrest-uxpr.rep-eq var.rep-eq var-lookup-in var-update-out)

Unrestrction 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]:** $x \not\models e \implies x \not\models \mathit{uop} f e$
by (transfer, simp)

**lemma unrest-bop [unrest]:** $[x \not\models u; x \not\models v] \implies x \not\models \mathit{bop} f u v$
by (transfer, simp)

**lemma unrest-trop [unrest]:** $[x \not\models u; x \not\models v; x \not\models w] \implies x \not\models \mathit{trop} f u v w$
by (transfer, simp)

**lemma unrest-qtop [unrest]:** $[x \not\models u; x \not\models v; x \not\models w; x \not\models y] \implies x \not\models \mathit{qtop} f u v w y$
by (transfer, simp)

For convenience, we also prove unrestriction rules for the bespoke operators on equality, numbers, arithmetic etc.
lemma unrest-eq [unrest]: \[ \langle x \# u; x \# v \rangle \Rightarrow x \# u = v \]
  by (simp add: eq-upred-def, transfer, simp)

lemma unrest-zero [unrest]: \[ x \# 0 \]
  by (simp add: unrest-lit zero-uexpr-def)

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

lemma unrest-numeral [unrest]: \[ x \# (numeral n) \]
  by (simp add: numeral-uexpr-simp unrest-lit)

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

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

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

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

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

lemma unrest-times [unrest]: \[ \langle x \# u; x \# v \rangle \Rightarrow x \# u \ast v \]
  by (simp add: times-uexpr-def unrest)

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

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]:
  \[ \langle \forall x. v \# F x \rangle \Rightarrow v \# \lambda x \cdot F x \]
  by (transfer, simp)

end

5 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 :: 'a \Rightarrow 'b \Rightarrow bool
syntax
-\textit{usedBy} :: \textit{salph} \Rightarrow \textit{logic} \Rightarrow \textit{logic} \Rightarrow \textit{logic} (\textbf{infix} \, \mathbin{\#} \, 20)

translations
-\textit{usedBy} \, x \, p \, == \, \text{CONST} \, \textit{usedBy} \, x \, p
-\textit{usedBy} \, ('\text{salpha} \, \textit{salph} \, x +_L \, y) \, P \, \text{<=} \, \textit{usedBy} \, (x +_L \, y) \, P

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

adhoc-overloading \textit{usedBy} \, \textit{usedBy-ueexpr}

lemma \textit{usedBy-lit} [unrest]: \, x \, \mathbin{\#} \, v
by (transfer, simp)

lemma \textit{usedBy-sublens}:
\textit{fixes} \, P :: ('a, 'a) \textit{ueexpr}
\textit{assumes} \, x \, \mathbin{\#} \, P \, x \subseteq_L \, y \, \textit{vwb-lens} \, y
\textit{shows} \, y \, \mathbin{\#} \, P
\textit{using} \, \textit{assms}
by (transfer, auto, metis lens-override-def lens-override-idem sublens-obs-get vwb-lens-mwb)

lemma \textit{usedBy-var} [unrest]: \, x \, \mathbin{\#} \, P \, \Rightarrow \, \& x \, \mathbin{\#} \, P
by (transfer, simp add: lens-defs)

lemma \textit{usedBy-lens-plus-1} [unrest]: \, x \, \mathbin{\#} \, P \, \Rightarrow \, x; \, y \, \mathbin{\#} \, P
by (transfer, simp add: lens-defs)

lemma \textit{usedBy-lens-plus-2} [unrest]: \, \{ x \, \mathbin{\triangle} \, y; \, y \, \mathbin{\#} \, P \, \} \, \Rightarrow \, x; \, y \, \mathbin{\#} \, P
by (transfer, auto simp add: lens-defs lens-indep-comm)

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

lemma \textit{usedBy-indep-uses}:
\textit{fixes} \, P :: ('a, 'a) \textit{ueexpr}
\textit{assumes} \, x \, \mathbin{\#} \, P \, x \, \mathbin{\triangle} \, y
\textit{shows} \, y \, \mathbin{\#} \, P
\textit{using} \, \textit{assms} \textbf{by} (transfer, auto, metis lens-indep-get lens-override-def)

lemma \textit{usedBy-var} [unrest]:
\textit{assumes} \, \textit{vwb-lens} \, x \, y \subseteq_L \, x
\textit{shows} \, x \, \mathbin{\#} \, \textit{var} \, y
\textit{using} \, \textit{assms}
by (transfer, simp add: ueexpr-defs pr-var-def)
\textit{(metis lens-override-def lens-override-idem sublens-obs-get vwb-lens-mwb)}

lemma \textit{usedBy-uop} [unrest]: \, x \, \mathbin{\#} \, e \, \Rightarrow \, x \, \mathbin{\#} \, \textit{uop} \, f \, e
by (transfer, simp)

lemma \textit{usedBy-bop} [unrest]: \, \{ x \, \mathbin{\#} \, u; \, x \, \mathbin{\#} \, v \, \} \, \Rightarrow \, x \, \mathbin{\#} \, \textit{bop} \, f \, u \, v
by (transfer, simp)

lemma \textit{usedBy-trop} [unrest]: \, \{ x \, \mathbin{\#} \, u; \, x \, \mathbin{\#} \, v; \, x \, \mathbin{\#} \, w \, \} \, \Rightarrow \, x \, \mathbin{\#} \, \textit{trop} \, f \, u \, v \, w
by (transfer, simp)
lemma usedBy-qtop [unrest]: \[ x \unrest u; x \unrest v; x \unrest w; x \unrest y \] \implies x \unrest \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]: \[ x \unrest u; x \unrest v \] \implies x \unrest u =_u v
by (simp add: eq-upred-def, transfer, simp)

lemma usedBy-zero [unrest]: x \unrest 0
by (simp add: usedBy-lit zero-ueexpr-def)

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

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

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

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

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

lemma usedBy-uminus [unrest]: x \unrest u \implies x \unrest - u
by (simp add: uminus-ueexpr-def unrest)

lemma usedBy-minus [unrest]: \[ x \unrest u; x \unrest v \] \implies x \unrest u - v
by (simp add: minus-ueexpr-def unrest)

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

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

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

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

end

6 Substitution

theory utp-subst
imports utp-expr utp-unrest
begin
6.1 Substitution definitions

Variable substitution, like unrestriction, 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 $\sigma \triangleright e$. We introduce a polymorphic constant that will be used to represent application of a substitution, and also a set of theorems to represent laws.

\begin{verbatim}
consts usubst :: 's ⇒ 'a ⇒ 'b (infixr † 80)

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

Application of a substitution simply applies the function $\sigma$ 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 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,′α) uexpr ⇒ (′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 'α 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 $\sigma$, 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 $\sigma$ $x$ $v$ = (λ $b$. 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 $\sigma$ does not restrict $x$ if application of $\sigma$ to an arbitrary state $\rho$ will not effect the valuation of $x$. Put another way, it requires that put and the substitution commute.
\end{verbatim}
Definition
unrest-usubst :: (a ⇒ α) ⇒ α usubst ⇒ bool
where
unrest-usubst x σ = (∀ g v. σ (put x g v) = put x (σ g) v)

Adhoc-overloading
unrest-unrest-usubst

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

definition cond-subst :: 'a usubst ⇒ (bool, 'a) uexpr ⇒ 'a usubst ⇒ 'a usubst ((3- α - 0) [927,0,324] 52) where
cond-subst σ b g = (λ s. if [b]e s then σ(s) else g(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 :: 'a usubst ⇒ (a ⇒ α) ⇒ (b ⇒ α) ⇒ 'a usubst ⇒ 'a usubst where
par-subst σ1 A σ2 = (λ s. (s ⊕L (σ1 s) on A) ⊕L (σ2 s) on B)

6.2 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 uexprs and salphas

Syntax

-smaplet :: [salpha, 'a] => smaplet (\ s => smaplets
:: smaplet => smaplets (\)
-SMaplets :: [smaplet, smaplets] => smaplets (\, \)
-SubstUpd :: [m usubst, smaplets] => m usubst ([\ ', ')] [900,0] 900)
-Subst :: smaplets => 'a => 'b \ ((1\[,\])
-psubst :: [logic, uexprs, uexprs] => logic
-subst :: logic => uexprs => salphas => logic ([\ ', ']\] [990,0,0] 991)
-uexprs :: [logic, uexprs] => uexprs (\, \)
:: logic => uexprs (\)
-salphas :: [salpha, salphas] => salphas (\, \)
:: salpha => salphas (\)
-par-subst :: logic => salpha => salpha => logic => logic ([\ ', ']\] [-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 ms == -SubstUpd (CONST id) ms
-Subst (-SMaplets m1 ms21) <= -SubstUpd (-Subst m1) ms21
-SMaplets m1 (-SMaplets m2 ms3) <= -SMaplets (-SMaplets m1 m2) m3
-subst P es vs => CONST subst (-psubst (CONST id) vs es) P

34
Thus we can write things like $\sigma(x \mapsto s v)$ to update a variable $x$ in $\sigma$ with expression $v$, $[x \mapsto e, y \mapsto f]$ 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

**Definition** $\mathrm{subst-del} :: \alpha \Rightarrow \mathrm{usubst} \Rightarrow (\alpha \Rightarrow \mathrm{usubst}) \Rightarrow \mathrm{usubst}$

Where

$\mathrm{subst-del} \sigma x = \sigma(x \mapsto s x)$

#### 6.3 Substitution Application Laws

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

**Method** $\mathrm{subst-tac} = (\mathrm{simp add: usubst unrest})$

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

**Lemma** $\mathrm{usubst-lookup-id}[\mathrm{usubst}]: \langle \mathrm{id} \rangle x = \mathrm{var} x$

By $(\mathrm{transfer}, \mathrm{simp})$

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

By $(\mathrm{simp add: id-def})$

A substitution update naturally yields the given expression.

**Lemma** $\mathrm{usubst-lookup-upd}[\mathrm{usubst}]:$

Assumes weak-lens $x$

Shows $\langle \sigma(x \mapsto s v) \rangle x = v$

Using $\mathrm{assms}$

By $(\mathrm{simp add: subst-upd-uvw-def, transfer}) (\mathrm{simp})$

**Lemma** $\mathrm{usubst-lookup-upd-pr-var}[\mathrm{usubst}]:$

Assumes weak-lens $x$

Shows $\langle \sigma(x \mapsto s v) \rangle (\mathrm{pr-var} x) = v$

Using $\mathrm{assms}$

By $(\mathrm{simp add: subst-upd-uvw-def pr-var-def, transfer}) (\mathrm{simp})$

Substitution update is idempotent.

**Lemma** $\mathrm{usubst-upd-idem}[\mathrm{usubst}]:$

Assumes $\mathrm{mwb-lens} x$

Shows $\sigma(x \mapsto s u, x \mapsto s v) = \sigma(x \mapsto s v)$

By $(\mathrm{simp add: subst-upd-uvw-def assms comp-def})$

Substitution updates commute when the lenses are independent.

**Lemma** $\mathrm{usubst-upd-comm}:$

Assumes $x \triangleright y$

Shows $\sigma(x \mapsto s u, y \mapsto s v) = \sigma(y \mapsto s v, x \mapsto s u)$

Using $\mathrm{assms}$

By $(\mathrm{rule-tac ext}, \mathrm{auto simp add: subst-upd-uvw-def assms comp-def lens-indep-comm})$
**lemma** usubst-upd-comm2:

assumes $z \bowtie y$

shows $\sigma(x \mapsto_y u, y \mapsto_v z, z \mapsto_y s) = \sigma(x \mapsto_y u, z \mapsto_y s, y \mapsto_v v)$

using assms

by (rule-tac ext, auto simp add: subst-upd-var-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 $x, y :: (\alpha \Rightarrow \alpha)$

assumes $\text{vwb-lens } x \text{ vwb-lens } y \bowtie y$

shows $\text{inj } [x \mapsto_s \&y, y \mapsto_s \&x]$

proof (rule inj)

fix $b_1 :: \alpha$ and $b_2 :: \alpha$

assume $[x \mapsto_s \&y, y \mapsto_s \&x] \ b_1 = [x \mapsto_s \&y, \ y \mapsto_s \&x] \ b_2$

hence $a \cdot \\text{put}_y (\text{put}_x b_1 ((\&y)_x b_1)) ((\&x)_x b_1) = \text{put}_y (\text{put}_x b_2 ((\&y)_x b_2)) ((\&x)_x b_2)$

by (auto simp add: subst-upd-var-def)

then have $(\forall a \ b. \ \text{put}_x (\text{put}_y a b) \ c = \text{put}_y (\text{put}_x a c) \ b) \land$

$(\forall a \ b. \ \text{get}_x (\text{put}_y a b) = \text{get}_x a) \land (\forall a \ b. \ \text{get}_y (\text{put}_x a b) = \text{get}_y a)$

by (simp add: assms(3) lens-indep.lens-put-irr2 lens-indep-comm)

then show $b_1 = b_2$

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]:

$vwb\text{-lens } x \Rightarrow [x \mapsto_s \text{var } x] = \text{id}$

apply (simp add: subst-upd-var-def)

apply (transfer)

apply (rule ext)

apply (auto)

done

**lemma** usubst-upd-pr-var-id [usubst]:

$vwb\text{-lens } x \Rightarrow [x \mapsto_s \text{var } (\text{pr-var } x)] = \text{id}$

apply (simp add: subst-upd-var-def pr-var-def)

apply (transfer)

apply (rule ext)

apply (auto)

done

**lemma** usubst-upd-comm-dash [usubst]:

fixes $x :: (\alpha \Rightarrow \alpha)$

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

using out-in-indep usubst-upd-comm by blast

**lemma** subst-upd-lens-plus [usubst]:

$\text{subst-upd } \sigma (x +_L y) \ <(u,v)> = \sigma(y \mapsto_s <v>, x \mapsto_s <u>)$

by (simp add: lens-defs uexpr-defs subst-upd-var-def, transfer, auto)

**lemma** subst-upd-in-lens-plus [usubst]:

$\text{subst-upd } \sigma (\text{ivar } (x +_L y)) \ <(u,v)> = \sigma(\$y \mapsto_s <v>, \$x \mapsto_s <u>)$

by (simp add: lens-defs uexpr-defs subst-upd-var-def, transfer, auto simp add: prod.case-eq-if)
lemma subst-upd-out-lens-plus [usubst]:
subt-upd σ (ovar (x +_L y)) ≪(u,v) = σ(σ(y →_s v), σ(x →_s u))
by (simp add: lens-defs uexpr-defs subst-upd-avar-def, transfer, auto simp add: prod.case-eq-if)

lemma usubst-lookup-upd-indep [usubst]:
assumes mwb-lens x x ≪ y
shows (σ(y →_s v))_s x = (σ)_s x
using assms
by (simp add: subst-upd-avar-def, transfer, simp)

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

lemma usubst-apply-unrest [usubst]:
fixes P :: ('a,'a) uexpr
assumes x ≫ P x ≫ y
shows σ(x →_s y,y →_s v) ≫ P = σ(y →_s v) ≫ P
using assms
by (simp add: subst-upd-avar-def, transfer, auto)

There follows various laws about deleting variables from a substitution.

lemma subst-del-id [usubst]:
vwb-lens x x ⊲ ⊳ id ⊲ x = id
by (simp add: subst-del-def subst-upd-avar-def lens-indep-comm)

lemma subst-del-upd-same [usubst]:
mwb-lens x x ⊲ ⊳ σ(x →_s v) ⊲ σ x = σ x
by (simp add: subst-del-def subst-upd-avar-def)

lemma subst-del-upd-diff [usubst]:
x ≫ y ⊲ σ(y →_s v) ⊲ σ x = (σ x)(y →_s v)
by (simp add: subst-del-def subst-upd-avar-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 ≫ P ⊲ σ(x →_s v) ≫ P = σ ≫ P
by (simp add: subst-upd-avar-def, transfer, auto)

lemma subst-unrest-2 [usubst]:
fixes P :: ('a,'a) uexpr
assumes x ≫ P x ≫ y
shows σ(x →_s u,y →_s v) ≫ P = σ(y →_s v) ≫ P
using assms
by (simp add: subst-upd-avar-def, transfer, auto, metis lens-indep.lens-put-comm)

lemma subst-unrest-3 [usubst]:
fixes P :: ('a,'a) uexpr
assumes x ≫ P x ≫ y x ≫ z
shows σ(x →_s u,y →_s v,z →_s w) ≫ P = σ(y →_s v,z →_s w) ≫ P
using assms
by (simp add: subst-upd-avar-def, transfer, auto, metis lens-indep.lens-put-comm)

lemma subst-unrest-4 [usubst]:
fixes P :: ('a,'a) uexpr
assumes x ≫ P x ≫ y x ≫ z x ≫ u
shows σ(x →_s e,y →_s f,z →_s g,u →_s h) ≫ P = σ(y →_s f,g →_s h) ≫ P
using assms
by (simp add: subst-upd-avar-def, transfer, auto, metis (no-types, hide-lams) lens-indep-comm)
lemma subst-unrest-5 [usubst]:
  fixes P :: ('a, 'a) uexpr
  assumes x ≠ P x ≡ y x ≡ z x ≡ u x ≡ v
  shows σ(x →ₗ e, y →ₗ f, z →ₗ g, u →ₗ h, v →ₗ i) † P = σ(y →ₗ f, z →ₗ g, u →ₗ h, v →ₗ i) † P
  using assms
  by (simp add: subst-upd-uvar-def, transfer, auto, metis (no-types, hide-lams) lens-indep-comm)

lemma subst-compose-upd [usubst]: x ≠ σ ⇒ σ ◦ σ(x →ₗ v) = (σ ◦ σ)(x →ₗ v)
  by (simp add: subst-upd-uvar-def, transfer, auto simp add: unrest-usubst-def subst.rep-eq)

Any substitution is a monotonic function.

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

6.4 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 usubst theorem attribute so that we can apply them using the substitution tactic.

lemma id-subst [usubst]: id † v = v
  by (transfer, simp)

lemma subst-lit [usubst]: σ † v= v
  by (transfer, simp)

lemma subst-var [usubst]: σ † var x = ⟨σ⟩ s x
  by (transfer, simp)

lemma usubst-ulambda [usubst]: σ † (λ x · P(x)) = (λ x · σ † P(x))
  by (transfer, simp)

lemma unrest-usubst-del [unrest]: [vw-lens x; x ≠ ((σ)s x); x ≠ σ † s x] ⇒ x ≠ (σ † P)
  (metis vw-lens.put-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 [usubst]: σ † uop f v = uop f (σ † v)
  by (transfer, simp)

lemma subst-bop [usubst]: σ † bop f u v = bop f (σ † u) (σ † v)
  by (transfer, simp)

lemma subst-trop [usubst]: σ † trop f u v w = trop f (σ † u) (σ † v) (σ † w)
  by (transfer, simp)

lemma subst-qtop [usubst]: σ † qtop f u v w x = qtop f (σ † u) (σ † v) (σ † w) (σ † x)
  by (transfer, simp)

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

38
by (simp add: case-prod-beta)

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

lemma subst-times [usubst]: \( \sigma \mapsto (x \times y) = \sigma \mapsto x \times \sigma \mapsto y \)
by (simp add: times-uexpr-def subst-bop)

lemma subst-mod [usubst]: \( \sigma \mapsto (x \mod y) = \sigma \mapsto x \mod \sigma \mapsto y \)
by (simp add: mod-uexpr-def usubst)

lemma subst-div [usubst]: \( \sigma \mapsto (x \div y) = \sigma \mapsto x \div \sigma \mapsto y \)
by (simp add: divide-uexpr-def usubst)

lemma subst-minus [usubst]: \( \sigma \mapsto (x - y) = \sigma \mapsto x - \sigma \mapsto y \)
by (simp add: minus-uexpr-def subst-bop)

lemma subst-uminus [usubst]: \( \sigma \mapsto (-x) = - (\sigma \mapsto x) \)
by (simp add: uminus-uexpr-def subst-uop)

lemma usubst-sgn [usubst]: \( \sigma \mapsto \text{sgn } x = \text{sgn } (\sigma \mapsto x) \)
by (simp add: sgn-uexpr-def subst-uop)

lemma usubst-abs [usubst]: \( \sigma \mapsto \text{abs } x = \text{abs } (\sigma \mapsto x) \)
by (simp add: abs-uexpr-def subst-uop)

lemma subst-zero [usubst]: \( \sigma \mapsto 0 = 0 \)
by (simp add: zero-uexpr-def subst-lit)

lemma subst-one [usubst]: \( \sigma \mapsto 1 = 1 \)
by (simp add: one-uexpr-def subst-lit)

lemma subst-eq-upred [usubst]: \( \sigma \mapsto (x = \_y) = (\sigma \mapsto x = \_y \sigma \mapsto y) \)
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]: \( \sigma \mapsto \_p \mapsto e = (\sigma \mapsto \_p \sigma) \mapsto e \)
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 \( x :: ('a \Rightarrow 'a) \)
  shows \( \rho(x \mapsto \_v \circ \sigma = (\rho \circ \sigma)(x \mapsto \_v \sigma \mapsto v) \)
by (rule ext, simp add: uexpr-defs subst-upd-uvw-def, transfer, simp)

lemma subst-singleton:
  fixes \( x :: ('a \Rightarrow 'a) \)
  assumes \( x \notin \sigma \)
  shows \( \sigma(x \mapsto \_v \mapsto P = (\sigma \mapsto P)[v/x]) \)
using assms
by (simp add: usubst)

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

39
6.5 Ordering substitutions

We set up a purely syntactic order on variable lenses which is useful for the substitution normal form.

definition var-name-ord :: (′a ⇒ ′α) ⇒ (′b ⇒ ′α) ⇒ bool where
[no-atp]: var-name-ord x y = True

syntax
-var-name-ord :: salpha ⇒ salpha ⇒ bool (infix ≺_v 65)

translations
-var-name-ord x y ::= CONST var-name-ord x y

A fact of the form \( x ≺_v y \) has no logical information; it simply exists to define a total order on named lenses that is useful for normalisation. The following theorem is simply an instance of the commutativity law for substitutions. However, that law could not be a simplification law as it would cause the simplifier to loop. Assuming that the variable order is a total order then this theorem will not loop.

lemma usubst-upd-comm-ord [usubst]:
assumes x ⪰ ⊳ y y ≺_v x
shows σ(x ↦-> s u, y ↦-> s v, x ↦-> s u) = σ(y ↦-> s v, x ↦-> s u)
by (simp add: assms(1) usubst-upd-comm)

lemma var-name-order-comp-outer [usubst]: x ≺_v y ⇒ x:a ≺_v y:b
by (simp add: var-name-ord-def)

lemma var-name-order-comp-inner [usubst]: a ≺_v b ⇒ x:a ≺_v x:b
by (simp add: var-name-ord-def)

lemma var-name-ord-pr-var-1 [usubst]: x ≺_v y ⇒ &x ≺_v y
by (simp add: var-name-ord-def)

lemma var-name-ord-pr-var-2 [usubst]: x ≺_v y ⇒ x ≺_v &y
by (simp add: var-name-ord-def)

6.6 Unrestriction laws

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

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

lemma unrest-usubst-id [unrest]:
mwb-lens x ⇒ x \sharp \: id
by (simp add: unrest-usubst-def)

lemma unrest-usubst-upd [unrest]:
\[ x \vartriangleright y; x \sharp \: \sigma; x \sharp \: v \] ⇒ \[ x \sharp \: \sigma(y \mapsto y_v v) \]

lemma unrest-subst [unrest]:
\[ x \sharp \: P; x \sharp \: \sigma \] ⇒ \[ x \sharp \: (\sigma \uparrow P) \]
by (transfer, simp add: unrest-usubst-def)
6.7 Conditional Substitution Laws

**lemma** usubst-cond-upd-1 [usubst]:
\[ \sigma(x \mapsto u) \circ b \triangleright_s g(x \mapsto_v v) = (\sigma \circ b \triangleright_s g)(x \mapsto_v u \circ b \triangleright_v v) \]
by (simp add: cond-subst-def subst-upd-unvar-def, transfer, auto)

**lemma** usubst-cond-upd-2 [usubst]:
\[ \llbracket vwb-lens x; x \ntriangleleft g \rrbracket \Rightarrow \sigma(x \mapsto_v u) \circ b \triangleright_s g = (\sigma \circ b \triangleright_s g)(x \mapsto_v u \circ b \triangleright_v v) \]
by (simp add: cond-subst-def subst-upd-unvar-def unrest-usubst-def, transfer)
  (metis (full-types, hide-lams) id-apply pr-var-def subst-upd-unvar-def usubst-upd-pr-var-id var.rep-eq)

**lemma** usubst-cond-upd-3 [usubst]:
\[ \llbracket vwb-lens x; x \ntriangleleft \sigma \rrbracket \Rightarrow \sigma \circ b \triangleright_s g(x \mapsto_v v) = (\sigma \circ b \triangleright_s g)(x \mapsto_v \ntriangleleft x \circ b \triangleright_v v) \]
by (simp add: cond-subst-def subst-upd-unvar-def unrest-usubst-def, transfer)
  (metis (full-types, hide-lams) id-apply pr-var-def subst-upd-unvar-def usubst-upd-pr-var-id var.rep-eq)

**lemma** usubst-cond-id [usubst]:
\[ id \circ b \triangleright_v id = id \]
by (auto simp add: cond-subst-def)

6.8 Parallel Substitution Laws

**lemma** par-subst-id [usubst]:
\[ \llbracket vwb-lens A; vwb-lens B \rrbracket \Rightarrow id [A | B]_s id = id \]
by (simp add: par-subst-def lens-override-idem id-def)

**lemma** par-subst-left-empty [usubst]:
\[ \llbracket vwb-lens A \rrbracket \Rightarrow \sigma [\emptyset | A]_s \circ g = id [\emptyset | A]_s \circ g \]
by (simp add: par-subst-def pr-var-def)

**lemma** par-subst-right-empty [usubst]:
\[ \llbracket vwb-lens A \rrbracket \Rightarrow \sigma [A | \emptyset]_s \circ g = \sigma [A | \emptyset]_s \circ id \]
by (simp add: par-subst-def pr-var-def)

**lemma** par-subst-comm:
\[ \llbracket A \bowtie B \rrbracket \Rightarrow \sigma [A | B]_s \circ g = \sigma [B | A]_s \circ g \]
by (simp add: par-subst-def lens-override-def lens-indep-comm)

**lemma** par-subst-upd-left-in [usubst]:
\[ \llbracket vwb-lens A; A \bowtie B; x \subseteq_L A \rrbracket \Rightarrow \sigma(x \mapsto_v v) [A | B]_s \circ g = (\sigma [A | B]_s \circ g)(x \mapsto_v v) \]
by (simp add: par-subst-def subst-upd-unvar-def lens-override-put-right-in)
  (simp add: lens-indep-comm lens-override-def sublens-pres-indep)

**lemma** par-subst-upd-left-out [usubst]:
\[ \llbracket vwb-lens A; A \bowtie B \rrbracket \Rightarrow \sigma(x \mapsto_v v) [A | B]_s \circ g = (\sigma [A | B]_s \circ g)(x \mapsto_v v) \]
by (simp add: par-subst-def subst-upd-unvar-def lens-override-put-right-out)

**lemma** par-subst-upd-right-in [usubst]:
\[ \llbracket vwb-lens B; A \bowtie B; x \subseteq_L B \rrbracket \Rightarrow \sigma [A | B]_s \circ g(x \mapsto_v v) = (\sigma [A | B]_s \circ g)(x \mapsto_v v) \]
using lens-indep-sym par-subst-comm par-subst-upd-left-in by fastforce

**lemma** par-subst-upd-right-out [usubst]:
\[ \llbracket vwb-lens B; A \bowtie B \rrbracket \Rightarrow \sigma [A | B]_s \circ g(x \mapsto_v v) = (\sigma [A | B]_s \circ g)(x \mapsto_v v) \]
by (simp add: par-subst-comm par-subst-upd-left-out)

end
7 UTP Tactics

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

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.

7.1 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

7.2 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 = (  
  ((unfold upred-defs urel-defs) [1])?;  
  

42
7.3 Transfer Tactics

Next, we define the component tactics used for transfer.

7.3.1 Robust Transfer

Robust transfer uses the transfer method of the lifting package.

method slow-uexpr-transfer = (transfer)

7.3.2 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.read-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** `uexpr-transfer-laws` `uexpr transfer laws`

**declare** `uexpr-eq-iff` `[uexpr-transfer-laws]`

**named-theorems** `uexpr-transfer-extra` extra simplifications for `uexpr` transfer

**declare** `unrest-uexpr.rep-eq` `[uexpr-transfer-extra]`

**usedBy** `uti-p expr` `numeral-uexpr-rep-eq` `[uexpr-transfer-extra]`

**usedBy** `uti-p expr` `less-eq-uexpr` `rep-eq` `[uexpr-transfer-extra]`

**Abs-uexpr-inverse** [simplified, `uexpr-transfer-extra`]

**Rep-uexpr-inverse** `[uexpr-transfer-extra]`

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

**method** `fast-uexpr-transfer` =

```
(simp add: `uexpr-transfer-laws` `uexpr-rep-eq-thms` `uexpr-transfer-extra`)
```

### 7.4 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** `uexpr-interp-tac` = `(simp add: `lens-interp-laws`)`?

### 7.5 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 ctx =>
  let val prove-tac = Basic-Tactics.utp-simp-tac in
    (UTP-Tactics.inst-gen-pred-tac args prove-tac ctx)
  end);
```

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

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

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

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

method-setup rel-blast = ⟨⟨
(Scan.lift UTP-Tactics.scan-args) >>
(fn args => fn ctx =>
  let val prove-tac = Basic-Tactics.utp-blast-tac in
  (UTP-Tactics.inst-gen-rel-tac args prove-tac ctx)
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
  = (rel-simp' simp; simp, blast intro: intro elim: elim dest: dest)
8 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)] = (P x y)[x→e][y→f]
  by (rel-auto)

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

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

end
lemma msubst-var [usubst];
    (utp-expr.var x)[[y→u]] = utp-expr.var x
by (pred-simp)

lemma msubst-var-2 [usubst];
    (utp-expr.var x)[[(y,z)→u]] = utp-expr.var x
by (pred-simp)+

lemma msubst-unrest [unrest]; [[ ∀ v. x ⋈ P(v); x ⋈ k ] ] \implies x ⋈ P(v)[v→k]
by (pred-auto)

end

9 Alphabetised Predicates

theory utp-pred
imports
    utp-expr
    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 [14].

9.1 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 ∨ 30) and
    Not (¬ - [40] 40)

consts
    utrue :: 'a (true)
    ufalse :: 'a (false)
    uconj :: 'a ⇒ 'a ⇒ 'a (infixr ∧ 35)
    udisj :: 'a ⇒ 'a ⇒ 'a (infixr ∨ 30)
    uimpl :: 'a ⇒ 'a ⇒ 'a (infixr ⇒ 25)
    uiff :: 'a ⇒ 'a ⇒ 'a (infixr ⇔ 25)
    unot :: 'a ⇒ 'a (¬ - [40] 40)
uex :: ('a ⇒ 'α) ⇒ 'p ⇒ 'p
uall :: ('a ⇒ 'α) ⇒ 'p ⇒ 'p
ushEx :: ['a ⇒ 'p] ⇒ 'p
ushAll :: ['a ⇒ 'p] ⇒ 'p

ad hoc overloading
uconj conj and
udisj disj and
unot Not

We set up two versions of each of the quantifiers: uex / uall and ushEx / ushAll. The former pair allows quantification of UTP variables, whilst the latter allows quantification of HOL variables in concert with the literal expression constructor ≪x≫. Both varieties will be needed at various points. Syntactically they are distinguished by a boldface quantifier for the HOL versions (achieved by the "bold" escape in Isabelle).

nonterminal idt-list

syntax
-idt-el :: idt ⇒ idt-list (-)
-idt-list :: idt ⇒ idt-list ⇒ idt-list ((/. -) [0, 1])
-uex :: salpha ⇒ logic ⇒ logic (∃ - · · [0, 10] 10)
-uall :: salpha ⇒ logic ⇒ logic (∀ - · · [0, 10] 10)
-ushEx :: ptrtn ⇒ logic ⇒ logic (∃ - · · [0, 10] 10)
-ushAll :: ptrtn ⇒ logic ⇒ logic (∀ - · · [0, 10] 10)
-ushBEx :: ptrtn ⇒ logic ⇒ logic ⇒ logic (∃ - - · · [0, 0, 10] 10)
-ushBALL :: ptrtn ⇒ logic ⇒ logic ⇒ logic ⇒ logic (∀ - - · · [0, 0, 10] 10)
-ushGLAll :: idt ⇒ logic ⇒ logic ⇒ logic (∀ - · · · · · [0, 0, 10] 10)
-ushLAll :: idt ⇒ logic ⇒ logic ⇒ logic (∀ - · · · · · [0, 0, 10] 10)
-uvar-res :: logic ⇒ salpha ⇒ logic (infixl [v 90])

translations
-uex x P == CONST uex x P
-uex (-salphaset (-salphamk (x +L y))) P <= -uex (x +L y) P
-uall x P == CONST uall x P
-uall (-salphaset (-salphamk (x +L y))) P <= -uall (x +L y) P
-ushEx x P == CONST ushEx (λ x. P)
∃ x ∈ A · P ⇒∃ x · ≪x≫ ∈u A ∧ P
-ushAll x P == CONST ushAll (λ x. P)
∀ x ∈ A · P ⇒∀ x · ≪x≫ ∈u A ⇒ P
∀ x · P · Q ⇒∀ x · P ⇒ Q
∀ x ≫ y · P ⇒∀ x · ≪x≫ ≫u y ⇒ P
∀ x ≪ y · P ⇒∀ x · ≪x≫ ≪u y ⇒ P

9.2 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
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 \cap 70)
notation Lattices.inf (infixl \cup 70)
purge-notation Lattices.sup (infixl \cap 65)
notation Lattices.sup (infixl \cup 65)
```

We trivially instantiate our refinement class
```
instance ueexpr :: (order, type) refine ..
```

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

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

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

```
instantiation ueexpr :: (lattice, type) lattice
begin
  lift-definition sup-ueexpr :: ('a, 'b) ueexpr \Rightarrow ('a, 'b) ueexpr \Rightarrow ('a, 'b) ueexpr
is \Lambda P Q A. Lattices.sup (P A) (Q A).
  lift-definition inf-ueexpr :: ('a, 'b) ueexpr \Rightarrow ('a, 'b) ueexpr \Rightarrow ('a, 'b) ueexpr
is \Lambda P Q A. Lattices.inf (P A) (Q A).
instance
  by (intro-classes) (transfer, auto)+
end
```
instance \textit{uexpr} :: (\textit{bounded-lattice}, \textit{type}) \textit{bounded-lattice}
begin
lift-definition \textit{bot-uexpr} :: ('a, 'b) \textit{uexpr} is \lambda A. \textit{Orderings.bot}.

lift-definition \textit{top-uexpr} :: ('a, 'b) \textit{uexpr} is \lambda A. \textit{Orderings.top}.

instance
by (intro-classes) (transfer, auto)+
end

lemma \textit{top-uexpr-rep-eq} [simp]:
\[(\textit{Orderings}.\\text{bot}) e b = \text{False}\]
by (transfer, auto)

lemma \textit{bot-uexpr-rep-eq} [simp]:
\[(\textit{Orderings}.\\text{top}) e b = \text{True}\]
by (transfer, auto)

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

instance \textit{uexpr} :: (\textit{complete-lattice}, \textit{type}) \textit{complete-lattice}
begin
lift-definition \textit{Inf-uexpr} :: ('a, 'b) \textit{uexpr set} \Rightarrow ('a, 'b) \textit{uexpr}
is \lambda \textit{PS} A. \textit{INF} P:PS. P(A).

lift-definition \textit{Sup-uexpr} :: ('a, 'b) \textit{uexpr set} \Rightarrow ('a, 'b) \textit{uexpr}
is \lambda \textit{PS} A. \textit{SUP} P:PS. P(A).

instance
by (intro-classes)
(transfer, auto intro: INF-lower SUP-upper simp add: INF-greatest SUP-least)+
end

instance \textit{uexpr} :: (\textit{complete-distrib-lattice}, \textit{type}) \textit{complete-distrib-lattice}
apply (intro-classes)
apply (transfer, rule ext, auto)
using sup-INF apply fastforce
apply (transfer, rule ext, auto)
using inf-SUP apply fastforce
done

instance \textit{uexpr} :: (\textit{complete-boolean-algebra}, \textit{type}) \textit{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
\[-\mu :: \textit{pttrn} \Rightarrow \textit{logic} \Rightarrow \textit{logic} (\mu - \cdot - [0, 10])\]
\[-\nu :: \textit{pttrn} \Rightarrow \textit{logic} \Rightarrow \textit{logic} (\nu - \cdot - [0, 10])\]

notation \textit{gfp} (\mu)
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)

**abbreviation** Conj-upred :: 'α upred set ⇒ 'α upred (⋁ [900] 900) where
\[ \land A \equiv \bigsqcap A \]

**abbreviation** Disj-upred :: 'α upred set ⇒ 'α upred (⋁ [900] 900) where
\[ \lor A \equiv \bigvee A \]

**notation**
conj-upred (infixr \(\land_p\) 35) and
disj-upred (infixr \(\lor_p\) 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 \(\lambda P F b. \sup \{[F x], b | x. [P x], b\}\).

**lift-definition** USUP :: ('a ⇒ 'α upred) ⇒ ('a ⇒ ('b::complete-lattice, 'α) uexpr) ⇒ ('b, 'α) uexpr
is \(\lambda P F b. \inf \{[F x], b | x. [P x], b\}\).

**syntax**
-USup :: pttrn ⇒ logic ⇒ logic \((\land - \cdots [0, 10] 10)\)
-USup :: pttrn ⇒ logic ⇒ logic \((\lor - \cdots [0, 10] 10)\)
-USup-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic \((\land - \cdot | - \cdots [0, 10] 10)\)
-USUP :: pttrn ⇒ logic ⇒ logic ⇒ logic \((\lor - | - \cdots [0, 10] 10)\)
-USup :: pttrn ⇒ logic ⇒ logic ⇒ logic \((\land - | - \cdots [0, 10] 10)\)
-USUP :: pttrn ⇒ logic ⇒ logic ⇒ logic \((\lor - | - \cdots [0, 10] 10)\)
-UInf :: pttrn ⇒ logic ⇒ logic \((\land - \cdot [0, 10] 10)\)
-UInf :: pttrn ⇒ logic ⇒ logic \((\lor - \cdot [0, 10] 10)\)
-UInf-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic \((\land - \cdot | - \cdots [0, 10] 10)\)
-UInf-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic \((\lor - | - \cdots [0, 10] 10)\)
-UINF :: pttrn ⇒ logic ⇒ logic ⇒ logic \((\land - | - \cdots [0, 10] 10)\)
-UINF :: pttrn ⇒ logic ⇒ logic ⇒ logic \((\lor - | - \cdots [0, 10] 10)\)

**translations**
- \(\square x \mid P \cdot F \Rightarrow \text{CONST UINF } (\lambda x. P) (\lambda x. F)\)
- \(\square x \cdot F \Rightarrow \square x \mid \text{true} \cdot F\)
- \(\square x \cdot F \Rightarrow \square x \mid \text{true} \cdot F\)
- \(\square x \in A \cdot F \Rightarrow \square x \mid \langle x \rangle \in_u \langle A \rangle \cdot F\)
\[ x \in A \cdot F \subseteq \bigsqcap \{ x \mid \langle y \rangle \in_u \langle A \rangle \cdot F \} \]

\[ x \mid P \cdot F \subseteq \text{CONST UINF} (\lambda y. P) (\lambda x. F) \]

\[ x \mid P \cdot F(x) \subseteq \text{CONST UINF} (\lambda x. P) F \]

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

\[ x \mid F \quad \Rightarrow \quad \bigcup \{ x \mid \text{true} \cdot F \} \]

\[ x \in A \cdot F \Rightarrow \bigcup \{ x \mid \langle x \rangle \in_u \langle A \rangle \cdot F \} \]

\[ x \in A \cdot F \subseteq \bigcup \{ x \mid \langle y \rangle \in_u \langle A \rangle \cdot F \} \]

\[ x \mid P \cdot F \subseteq \text{CONST USUP} (\lambda y. P) (\lambda x. F) \]

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

We also define the other predicate operators

**lift-definition impl:** \( \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \)

\( \lambda P \quad Q \quad A \quad. \quad P \quad A \quad \Rightarrow \quad Q \quad A \quad . \)

**lift-definition iff-upred:** \( \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \)

\( \lambda P \quad Q \quad A \quad. \quad P \quad A \quad \Leftrightarrow \quad Q \quad A \quad . \)

**lift-definition ex:** \( \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \)

\( \lambda x \quad P \quad b \quad. \quad (\exists v. P(\text{put} x \quad b \quad v)) \).

**lift-definition shEx:** \( \text{\textquote{'} } \beta \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \)

\( \lambda P \quad A \quad. \quad \exists x. (P \quad x \quad A) \).

**lift-definition all:** \( \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \)

\( \lambda x \quad P \quad b \quad. \quad (\forall v. P(\text{put} x \quad b \quad v)) \).

**lift-definition shAll:** \( \text{\textquote{'} } \beta \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \)

\( \lambda P \quad A \quad. \quad \forall x. (P \quad x \quad A) \).

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:** \( \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \)

\( \lambda P \quad x \quad b \quad. \quad \exists b'. P(b' \oplus L \quad b \quad \text{on} \quad x) \).

**translations**

\( -uvar-res \quad P \quad a \quad \Rightarrow \quad \text{CONST \ var-res} \quad P \quad a \)

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:** \( \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \Rightarrow (\square u) \)

\( \lambda P \quad A \quad. \quad \forall A'. P \quad A' \).

**lift-definition taut:** \( \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \Rightarrow \text{\textquote{'} } \alpha \text{ upred } \)

\( \lambda P \quad A \quad. \quad \forall A'. P \quad A' \).

— Configuration for UTP tactics (see utp-tactics).

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

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

**adhoc-overloading**

\( \text{utrue true-upred and} \)

\( \text{ufalse false-upred and} \)

\( \text{unot not-upred and} \)
uconj conj-upred and
udisz disj-upred and
uimpl impl and
uiff iff-upred and
uex ex and
uall all and
ushEx shEx and
ushAll shAll

syntax
-uneq :: logic ⇒ logic ⇒ logic (infixl \neq 50)
-ummem :: ('a, 'a) uexpr ⇒ ('a set, 'a) uexpr ⇒ (bool, 'a) uexpr (infix \notin 50)

translations
x \neq_y y == CONST unot (x =_y y)
x \notin_A A == CONST unot (CONST bop (op \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-upd-uvar-def [upred-defs]
declare cond-subst-def [upred-defs]
declare subst-del-def [upred-defs]
declare unrest-usubst-def [upred-defs]
declare uexpr-defs [upred-defs]

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

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

declare true-alt-def[THEN sym,lit-simps]
declare false-alt-def[THEN sym,lit-simps]

9.3 Unrestiction Laws

lemma unrest-allE:
[ Σ x P; P = true ⇒ Q; P = false ⇒ Q ] ⇒ Q
by (pred-auto)

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

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

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

lemma unrest-disj [unrest]: [ x \notin (P :: 'a upred); x \notin Q ] ⇒ x \notin P \lor Q
by (pred-auto)
lemma unrest-UINF [unrest]:
\[ \forall (\bigwedge i \cdot x \not\in P(i)) \land (\bigwedge i \cdot x \not\in Q(i)) \implies x \not\in (\prod i \mid P(i) \cdot Q(i)) \]
by (pred-auto)

lemma unrest-USUP [unrest]:
\[ \forall (\bigwedge i \cdot x \not\in P(i)) \land (\bigwedge i \cdot x \not\in Q(i)) \implies x \not\in (\bigvee i \mid P(i) \cdot Q(i)) \]
by (pred-auto)

lemma unrest-UINF-mem [unrest]:
\[ \forall (\bigwedge i \cdot i \in A \implies x \not\in P(i)) \implies x \not\in (\prod i \mid P(i)) \]
by (pred-simp, metis)

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

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

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

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

The sublens proviso can be thought of as membership below.

lemma unrest-ex-in [unrest]:
\[ \forall \forall \exists \text{mwb-lens } y \mid x \subseteq L \implies x \not\in (\exists y \mid P) \]
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]:
\[ \forall \forall \exists \text{mwb-lens } y \mid x \not\in y \implies y \not\in (\forall x \mid P) \]
using assms lens-indep-comm
by (rel-simp', fastforce)

lemma unrest-all-in [unrest]:
\[ \forall \forall \exists \text{mwb-lens } y \mid x \subseteq L \implies x \not\in (\forall y \mid P) \]
by (pred-auto)

lemma unrest-all-diff [unrest]:
\[ \forall \forall \exists \text{mwb-lens } y \mid x \not\in y \implies y \not\in (\forall x \mid P) \]
using assms
by (pred-simp, simp-all add: lens-indep-comm)

lemma unrest-var-res-diff [unrest]:

54
assumes $x \sqsubseteq y$
shows $y \not\sqsubseteq (P \mid_v x)$
using assms by (pred-auto)

**Lemma unrest-var-res-in** [unrest]:
assumes mwb-lens $x y \subseteq_L x y \not\sqsubseteq P$
shows $y \not\sqsubseteq (P \mid_v x)$
using assms
apply (pred-auto)
apply fastforce
apply (metis (no-types, lifting) mwb-lens-weak weak-lens.put-get)
done

**Lemma unrest-shEx** [unrest]:
assumes $\land_y. x \not\sqsubseteq P(y)$
shows $x \not\sqsubseteq (\exists y \cdot P(y))$
using assms by (pred-auto)

**Lemma unrest-shAll** [unrest]:
assumes $\land_y. x \not\sqsubseteq P(y)$
shows $x \not\sqsubseteq (\forall y \cdot P(y))$
using assms by (pred-auto)

**Lemma unrest-closure** [unrest]:
$x \not\sqsubseteq [P]_u$
by (pred-auto)

### 9.4 Used-by laws

**Lemma usedBy-not** [unrest]:
$\llbracket x \not\sqsubseteq P \rrbracket \implies x \not\sqsubseteq (\neg P)$
by (pred-simp)

**Lemma usedBy-conj** [unrest]:
$\llbracket x \not\sqsubseteq P; x \not\sqsubseteq Q \rrbracket \implies x \not\sqsubseteq (P \land Q)$
by (pred-simp)

**Lemma usedBy-disj** [unrest]:
$\llbracket x \not\sqsubseteq P; x \not\sqsubseteq Q \rrbracket \implies x \not\sqsubseteq (P \lor Q)$
by (pred-simp)

**Lemma usedBy-impl** [unrest]:
$\llbracket x \not\sqsubseteq P; x \not\sqsubseteq Q \rrbracket \implies x \not\sqsubseteq (P \Rightarrow Q)$
by (pred-simp)

**Lemma usedBy-iff** [unrest]:
$\llbracket x \not\sqsubseteq P; x \not\sqsubseteq Q \rrbracket \implies x \not\sqsubseteq (P \Leftrightarrow Q)$
by (pred-simp)

### 9.5 Substitution Laws

Substitution is monotone

**Lemma subst-mono**: $P \subseteq Q \implies (\sigma \vdash P) \subseteq (\sigma \vdash Q)$
by (pred-auto)

55
lemma subst-true [usubst]: $\sigma \vdash \text{true} = \text{true}$
  by (pred-auto)

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

lemma subst-not [usubst]: $\sigma \vdash (\neg P) = (\neg \sigma \vdash P)$
  by (pred-auto)

lemma subst-impl [usubst]: $\sigma \vdash (P \Rightarrow Q) = (\sigma \vdash P \Rightarrow \sigma \vdash Q)$
  by (pred-auto)

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

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

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

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

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

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

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

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

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

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

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

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

lemma subst-ex-same' [usubst]:
  mwb-lens $x \Longrightarrow \sigma(x := v) \vdash (\exists &x \cdot P(x)) = \sigma \vdash (\exists &x \cdot P(x))$
  by (pred-auto)

lemma subst-ex-indep [usubst]:
  assumes $x \not\equiv y \not\equiv v$
  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 ∉ σ ⇔ σ † (∃ x · P) = (∃ x · σ † P)
by (pred-auto)

lemma subst-all-same [usubst]:
msubst x ⇒ σ(x ↔ y) † (∀ x · P) = σ † (∀ x · P)
by (simp add: id-subst subst-unrest unrest-all-in)

lemma subst-all-indep [usubst]:
assumes x ≈ y y ∉ v
shows (∀ y · P)[v/x] = (∀ y · P[v/x])
by (pred-simp, simp-all add: lens-indep-comm)

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

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

lemma msubst-not [usubst]: (∃ P(x))[x→v] = (∃ (P x)[x→v])
by (pred-auto)

lemma msubst-not-2 [usubst]: (∃ P x y)(x,y→v) = (∃ ((P x y))[x→v])
by (pred-auto+)

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

lemma msubst-disj-2 [usubst]: (P x y ∨ Q x y)(x,y→v) = ((P x y)[x→v] ∨ (Q x y)[x→y])
by (pred-auto+)

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

lemma msubst-conj-2 [usubst]: (P x y ∧ Q x y)(x,y→v) = ((P x y)[x→v] ∧ (Q x y)[x→v])
by (pred-auto+)

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

lemma msubst-implies-2 [usubst]:
(P x y ⇒ Q x y)(x,y→v) = ((P x y)[x→y] ⇒ (Q x y)[x,y→v])
by (pred-auto+)

lemma msubst-shAll [usubst]:
(∀ x · P x y)[y→v] = (∀ x · (P x y)[y→v])
by (pred-auto)

lemma msubst-shAll-2 [usubst]:
(∀ x · P x y z)[z→v] = (∀ x · (P x y z)[z→v])

57
10 UTP Events

theory utp-event
imports utp-pred
begin

10.1 Events

Events of some type '$\emptyset$' are just the elements of that type.

type-synonym '$\emptyset$ event' = '$\emptyset$

10.2 Channels

Typed channels are modelled as functions. Below, '$a$' determines the channel type and '$\emptyset$' the underlying event type. As with values, it is difficult to introduce channels as monomorphic types due to the fact that they can have arbitrary parametrisations in term of '$a$'. Applying a channel to an element of its type yields an event, as we may expect. Though this is not formalised here, we may also sensibly assume that all channel-representing functions are injective. Note: is there benefit in formalising this here?

\textbf{type-synonym} $(a, '\emptyset) \text{ chan} = a \Rightarrow '\emptyset \text{ event}$

A downside of the approach is that the event type '$\emptyset$' must be able to encode all events of a process model, and hence cannot be fixed upfront for a single channel or channel set. To do so, we actually require a notion of 'extensible' datatypes, in analogy to extensible record types. Another solution is to encode a notion of channel scoping that namely uses \textit{sum} types to lift channel types into extensible ones, that is using channel-set specific scoping operators. This is a current work in progress.

10.2.1 Operators

The Z type of a channel corresponds to the entire carrier of the underlying HOL type of that channel. Strictly, the function is redundant but was added to mirror the mathematical account in [?]. (TODO: Ask Simon Foster for [?])

\textbf{definition} \textit{chan-type} :: $(a, '\emptyset) \text{ chan} \Rightarrow a \text{ set } (\delta_a)$ \textbf{where}\n
\textit{upred-defs}: $\delta_a = \text{UNIV}$

The next lifted function creates an expression that yields a channel event, from an expression on the channel type '$a$.'

\textbf{definition} \textit{chan-apply} :: $(a, '\emptyset) \text{ chan} \Rightarrow (a, 'a) \text{ uexpr} \Rightarrow (\emptyset \text{ event}, 'a) \text{ uexpr} \Rightarrow (\emptyset / -) \text{ u}$ \textbf{where}\n
\textit{upred-defs}: $(c \cdot e) u = \langle c \rangle (e) u$

\textbf{lemma} unrest-chan-apply [unrest]: $x \# e \iff x \# (c \cdot e) u$
by (rel-auto)

\textbf{lemma} usubst-chan-apply [usubst]: $\sigma \uplus (c \cdot v) u = (c \cdot \sigma \uplus v) u$
by (rel-auto)

58
lemma msubst-event [usubst]:
\((c \cdot v x)_u[x \to u] = (c \cdot (v x)[x \to u])_u\)
by (pred-simp)

lemma msubst-event-2 [usubst]:
\((c \cdot v x y)_{u[(x, y) \to u]} = (c \cdot (v x y)[(x, y) \to u])_u\)
by (pred-simp)+

end

11 Alphabet Manipulation

theory utp-alphabet
  imports utp-pred utp-event
begin

11.1 Preliminaries

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.

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 \(\text{get}\) 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.

lift-definition aext :: \('a \times ('\beta \cdot \alpha)\) uexpr \Rightarrow (\'\beta \cdot '\alpha) uexpr (infixr \oplus 75)
is \(\lambda P \cdot P (\text{get}_\alpha b).

update-uexpr-rep-eq-thms

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

lemma aext-twice: \((P \oplus_p a) \oplus_p b = P \oplus_p (a \oplus_p b)\)
by (pred-auto)
The bijective $\Sigma$ lens identifies the source and view types. Thus an alphabet extension using this has no effect.

**Lemma aext-id** \[ \text{simp}: P \oplus_p I_L = P \]
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.

**Lemma aext-lit** \[ \text{simp}: \langle v \rangle \oplus_p a = \langle v \rangle \]
by (pred-auto)

**Lemma aext-zero** \[ \text{simp}: 0 \oplus_p a = 0 \]
by (pred-auto)

**Lemma aext-one** \[ \text{simp}: 1 \oplus_p a = 1 \]
by (pred-auto)

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

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

**Lemma aext-false** \[ \text{simp}: \text{false} \oplus_p a = \text{false} \]
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** \[ \alpha: (P \iff Q) \oplus_p x = (P \oplus_p x \iff Q \oplus_p x) \]
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-event** \[ \alpha: (c \cdot v)_u \oplus_p a = (c \cdot v \oplus_p a)_u \]
by (pred-auto)

Alphabet extension distributes through the function liftings.

**Lemma aext-uop** \[ \alpha: \text{uop } f u \oplus_p a = \text{uop } f (u \oplus_p a) \]
by (pred-auto)

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

**Lemma aext-trop** \[ \alpha: \text{trop } f u v w \oplus_p a = \text{trop } f (u \oplus_p a) (v \oplus_p a) (w \oplus_p a) \]
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]:
\( (x + y) \oplus_p a = (x \oplus_p a) + (y \oplus_p a) \)
by (pred-auto)

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

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

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

lemma aext-divide [alpha]:
\( (x / y) \oplus_p a = (x \oplus_p a) / (y \oplus_p a) \)
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]:
\( \text{var} x \oplus_p a = \text{var} (x ; L_a) \)
by (pred-auto)

lemma aext-ulambda [alpha]:
\( (\lambda x . P(x)) \oplus_p a = (\lambda x . P(x) \oplus_p a) \)
by (pred-auto)

Alphabet extension is monotonic and continuous.

lemma aext-mono: \( P \subseteq Q \Rightarrow P \oplus_p a \subseteq Q \oplus_p a \)
by (pred-auto)

lemma aext-cont [alpha]: \( \forall A. P \in A \Rightarrow \bigcap P \subseteq p \oplus_p a \)
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]:
\[ \text{mwb-lens a} ; x \neq p \] \( \Rightarrow \text{unrest} (x ; L_a) (p \oplus_p a) \)
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]:
\( a \not\bowtie b \Rightarrow b \neq (p \oplus_p a) \)
by pred-auto
11.3 Expression Alphabet Restriction

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

\[
\text{lift-definition} \quad \text{arestr :: } (\forall a, \forall \alpha \text{ uexpr } \Rightarrow (\forall \beta, \forall \alpha \text{ lens } \Rightarrow (\forall a, \forall \beta \text{ uexpr } \text{ infixr } e) 90)
\]
\[
\text{is } \lambda P \times b. \ P \ (\text{create}_x b).
\]

update-uxpr-rep-eq-thms

\text{lemma arestr-id [simp]}: \ P \ |_e 1 \ L = P \\
\quad \text{by (pred-auto)}

\text{lemma arestr-aext [simp]}: \text{ mwb-lens a } \implies \ (P \oplus_p a) \ |_e a = P \\
\quad \text{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.

\text{lemma aext-arestr [alpha]}:
\begin{align*}
\text{assumes } \text{ mwb-lens a } \text{ bij-lens } (a +_L b) \ a \bowtie b \ b \not\bowtie P \\
\text{shows } (P \ |_e a) \oplus_p a = P \\
\text{proof} - \\
\quad \text{from } \text{assms}(2) \text{ have } 1 \ L \subseteq_L a +_L b \\
\quad \text{by (simp add: bij-lens-equeq-id lens-equeq-def)}
\end{align*}
\begin{align*}
\quad \text{with } \text{assms}(1,3,4) \text{ show } \text{thesis} \\
\text{ apply (auto simp add: id-lens-def lens-plus-def sublens-def lens-comp-def prod.case-eq-if)}
\end{align*}
\begin{align*}
\quad \text{apply (predsimp)}
\quad \text{apply (metis lens-indep-comm mwb-lens-weak weak-lens.put-get)}
\end{align*}
\text{done}
qed

\text{lemma arestr-lit [simp]}: \ll v \gg \ |_e a = \ll v \gg \\
\quad \text{by (pred-auto)}

\text{lemma arestr-zero [simp]}: 0 \ |_e a = 0 \\
\quad \text{by (pred-auto)}

\text{lemma arestr-one [simp]}: 1 \ |_e a = 1 \\
\quad \text{by (pred-auto)}

\text{lemma arestr-numeral [simp]}: \text{ numeral n } \ |_e a = \text{ numeral n} \\
\quad \text{by (pred-auto)}

\text{lemma arestr-var [alpha]}:
\begin{align*}
\text{var x } \ |_e a = \text{ var } (x /_L a) \\
\quad \text{by (pred-auto)}
\end{align*}

\text{lemma arestr-true [simp]}: \text{ true } \ |_e a = \text{ true} \\
\quad \text{by (pred-auto)}

\text{lemma arestr-false [simp]}: \text{ false } \ |_e a = \text{ false} \\
\quad \text{by (pred-auto)}

\text{lemma arestr-not [alpha]}: (\neg P) \ |_e a = (\neg (P \ |_e a)) \\
\quad \text{by (pred-auto)}
Lemma \textit{arestr-and} [\alpha]: (P \land Q) \mathord{\left| e \right.} x = (P \mathord{\left| e \right.} x \land Q \mathord{\left| e \right.} x)
by (pred-auto)

Lemma \textit{arestr-or} [\alpha]: (P \lor Q) \mathord{\left| e \right.} x = (P \mathord{\left| e \right.} x \lor Q \mathord{\left| e \right.} x)
by (pred-auto)

Lemma \textit{arestr-imp} [\alpha]: (P \Rightarrow Q) \mathord{\left| e \right.} x = (P \mathord{\left| e \right.} x \Rightarrow Q \mathord{\left| e \right.} 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 \textit{upred-ares} :: \alpha \upred \Rightarrow \beta \Rightarrow \alpha \upred
where [\textit{upred-defs}]: upred-ares P a = (P \mathord{\left| e \right.} x) \mathord{\left| e \right.} a

Syntax
-\textit{-upred-ares} :: logic \Rightarrow salpha \Rightarrow logic (infixl \mathord{\left| p \right.} 90)

Translations
-\textit{-upred-ares} P a == CONST upred-ares P a

Lemma \textit{upred-aext-ares} [\alpha]:
\textit{vwb-lens} a = (P \mathord{\left| p \right.} a)\mathord{\left| p \right.} a = P
by (pred-auto)

Lemma \textit{upred-ares-aext} [\alpha]:
a \mathord{\left| p \right.} P = (P \mathord{\left| p \right.} a)\mathord{\left| p \right.} a = P
by (pred-auto)

Lemma \textit{upred-arestr-lit} [simp]: \textit{\langle v \rangle} \mathord{\left| p \right.} a = \textit{\langle v \rangle}
by (pred-auto)

Lemma \textit{upred-arestr-true} [simp]: true \mathord{\left| p \right.} a = true
by (pred-auto)

Lemma \textit{upred-arestr-false} [simp]: false \mathord{\left| p \right.} a = false
by (pred-auto)

Lemma \textit{upred-arestr-or} [\alpha]: (P \lor Q) \mathord{\left| p \right.} x = (P \mathord{\left| p \right.} x \lor Q \mathord{\left| p \right.} x)
by (pred-auto)

11.5 Alphabet Lens Laws
Lemma \textit{alpha-in-var} [\alpha]: x :L \textit{fst}_L = \textit{in-var} x
by (simp add: in-var-def)

Lemma \textit{alpha-out-var} [\alpha]: x :L \textit{snd}_L = \textit{out-var} x
by (simp add: out-var-def)

Lemma \textit{in-var-prod-lens} [\alpha]:
\[ \text{wb-lens } Y \Rightarrow \text{in-var } x : L (X \times L Y) = \text{in-var } (x : L X) \]

by (simp add: in-var-def prod-as-plus lens-comp-assoc fst-lens-plus)

**lemma out-var-prod-lens [alpha]:**
\[ \text{wb-lens } X \Rightarrow \text{out-var } x : L (X \times L Y) = \text{out-var } (x : L Y) \]

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 :: \( \alpha \Rightarrow \beta \Rightarrow \beta \Rightarrow \alpha \) usubst where**

\[ \text{upred-defs: } \sigma \oplus x = (\lambda s. \text{put}_x s (\sigma \text{ get}_x s)) \]

**lemma id-subst-ext [usubst]:**
\[ \text{wb-lens } x \Rightarrow \text{id } \oplus_s x = \text{id} \]

by pred-auto

**lemma upd-subst-ext [alpha]:**
\[ \text{vwb-lens } x \Rightarrow (\sigma \uparrow e) \oplus_p x = (\sigma \oplus_s x) \uparrow (e \oplus_p x) \]

by (pred-auto)

**lemma apply-subst-ext [alpha]:**
\[ \text{vwb-lens } x \Rightarrow (\sigma \uparrow e) \oplus_p x = (\sigma \uparrow e \oplus_p x) \)

by (pred-auto)

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

by (pred-auto)

**lemma subst-aext-comp [usubst]:**
\[ \text{vwb-lens } a \Rightarrow (\sigma \oplus_a a) \circ (\rho \oplus_s a) = (\sigma \circ \rho) \oplus_s a \]

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 :: \( \alpha \Rightarrow \beta \Rightarrow \beta \Rightarrow \alpha \) usubst where**

\[ \text{upred-defs: } \sigma |_x x = (\lambda s. \text{get}_x (\sigma \text{ create}_x s)) \]

**lemma id-subst-res [usubst]:**
\[ \text{mwb-lens } x \Rightarrow \text{id } |_s x = \text{id} \]

by pred-auto

**lemma upd-subst-res [alpha]:**
\[ \text{mwb-lens } x \Rightarrow (\sigma \uparrow e) |_s x = (\sigma |_s x) \uparrow (e |_s x) \]

by (pred-auto)

**lemma subst-ext-res [usubst]:**
\[ \text{mwb-lens } x \Rightarrow (\sigma \oplus_s x) |_s x = \sigma \]

by pred-auto
by (pred-auto)

lemma unrest-subst-alpha-ext [unrest]:
  \( x \bowtie y \implies x \not\bowtie (P \oplus_y y) \)
by (pred-simp robust, metis lens-indep-def)
end

12 Lifting Expressions

theory utp-lift
  imports utp-alphabet
begin

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 \([P]\) with some subscript to denote lifting an expression into a larger alphabet, and \([P]\) 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 :: ('\(\alpha\)', '\(\alpha\) uexpr \Rightarrow ('\(\alpha\) \times ' \beta) uexpr ([\(-\)]<)
where \([P]< \equiv P \oplus_p \text{fst}_L\)

abbreviation drop-pre :: ('\(\alpha\) \times ' \beta) uexpr \Rightarrow ('\(\alpha\)', '\(\alpha\) uexpr ([\(-\)]<)
where \([P]< \equiv P \mid_e \text{fst}_L\)

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 :: ('\(\alpha\)' \times ' \beta) uexpr \Rightarrow ('\(\alpha\)' \times ' \beta) uexpr ([\(-\)]>)
where \([P]> \equiv P \oplus_p \text{snd}_L\)

abbreviation drop-post :: ('\(\alpha\)' \times ' \beta) uexpr \Rightarrow ('\(\alpha\)' \times ' \beta) uexpr ([\(-\)]>)
where \([P]> \equiv P \mid_e \text{snd}_L\)

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]:
  \([\text{var } x]< = \$x\)
by (alpha-tac)

lemma lift-post-var [simp]:
  \([\text{var } x]> = \$x'\)
by (alpha-tac)
12.3 Substitution Laws

**lemma** pre-var-subst [usubst]:
\[ \sigma(x \mapsto s \langle v \rangle) \upharpoonright [P] < = \sigma \upharpoonright [P \langle v \rangle / \& x] < \]
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]:
\[ \text{fixes } x :: (\prime a \Rightarrow \prime \alpha) \]
\[ \text{shows } \$x \# \left[ P \right] < \]
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 op \( \leq \) op \(<\)
\[ \text{disj-upred false-upred true-upred} \]
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!]:
\[ [ A \& B \& A \& B ] \Rightarrow P \] \Rightarrow P
by (rel-auto)

**lemma** taut-refine-impl: [ Q \subseteq P; ‘P’ ] \Rightarrow ‘Q’
by (rel-auto)

**lemma** taut-shEx-elim:
\[ [ \exists x . \& P x \& \& x . \& P x \Rightarrow Q ] \Rightarrow Q \]
by (rel-blast)

Linking refinement and HOL implication

**lemma** refine-prop-intro:
assumes $\Sigma \not\vdash P \Sigma \not\vdash Q \ 'Q' \implies 'P'$
shows $P \sqsubseteq Q$
using assms
by (pred-auto)

lemma taut-not: $\Sigma \not\vdash P \implies \lnot 'P' = 'Q'$
by (rel-auto)

lemma taut-shAll-intro:
$\forall x. 'P x' \implies \forall x \cdot P x'$
by (rel-auto)

lemma taut-shAll-intro-2:
$\forall x y. 'P x y' \implies \forall (x, y) \cdot P x y'$
by (rel-auto)

lemma taut-impl-intro:
$[ [ \Sigma \not\vdash P; 'P' \implies 'Q' ] ] \implies 'P \implies Q'$
by (rel-auto)

lemma upred-eval-taut:
$'P[\ll b\gg]/\&v] = [P]_v b$
by (pred-auto)

lemma refBy-order: $P \sqsubseteq Q = 'Q \Rightarrow P'$
by (pred-auto)

lemma conj-idem [simp]: $(P::'α upred) \land P) = P$
by (pred-auto)

lemma disj-idem [simp]: $(P::'α upred) \lor P) = P$
by (pred-auto)

lemma conj-comm: $(P::'α upred) \land Q) = (Q \land P)$
by (pred-auto)

lemma disj-comm: $(P::'α upred) \lor Q) = (Q \lor P)$
by (pred-auto)

lemma conj-subst: $P = R \implies ((P::'α upred) \land Q) = (R \land Q)$
by (pred-auto)

lemma disj-subst: $P = R \implies ((P::'α upred) \lor Q) = (R \lor Q)$
by (pred-auto)

lemma conj-assoc:$(((P::'α upred) \land Q) \land S) = (P \land (Q \land S))$
by (pred-auto)

lemma disj-assoc:$(((P::'α upred) \lor Q) \lor S) = (P \lor (Q \lor S))$
by (pred-auto)

lemma conj-disj-abs:$(P::'α upred) \land (P \lor Q)) = P$
by (pred-auto)

lemma disj-conj-abs:$(P::'α upred) \lor (P \land Q)) = P$
by (pred-auto)

lemma conj-disj-distr:((P::'α upred) ∧ (Q ∨ R)) = ((P ∧ Q) ∨ (P ∧ R))
by (pred-auto)

lemma disj-conj-distr:((P::'α upred) ∨ (Q ∧ R)) = ((P ∨ Q) ∧ (P ∨ R))
by (pred-auto)

lemma true-disj-zero [simp]:
(P ∨ true) = true (true ∨ P) = true
by (pred-auto)+

lemma true-conj-zero [simp]:
(P ∧ false) = false (false ∧ P) = false
by (pred-auto)+

lemma false-sup [simp]: false ⊓ P = P false = P
by (pred-auto)+

lemma true-inf [simp]: true ⊔ P = P true = P
by (pred-auto)+

lemma imp-vacuous [simp]: (false ⇒ u) = true
by (pred-auto)

lemma imp-true [simp]: (p ⇒ true) = true
by (pred-auto)

lemma true-imp [simp]: (true ⇒ p) = p
by (pred-auto)

lemma impl-mp1 [simp]: (P ∧ (P ⇒ Q)) = (P ∧ Q)
by (pred-auto)

lemma impl-mp2 [simp]: ((P ⇒ Q) ∧ P) = (Q ∧ P)
by (pred-auto)

lemma impl-adjoin: ((P ⇒ Q) ∧ R) = ((P ∧ R ⇒ Q ∧ R) ∧ R)
by (pred-auto)

lemma impl-refine-intro:
\[[Q_1 ⊆ P_1; P_2 ⊆ (P_1 ∧ Q_2)] \implies (P_1 ⇒ P_2) ⊆ (Q_1 ⇒ Q_2)\]
by (pred-auto)

lemma spec-refine:
Q ⊆ (P ∧ R) \implies (P ⇒ Q) ⊆ R
by (rel-auto)

lemma impl-disjI: \[\{P ⇒ R'; Q ⇒ R'\} \implies \{P ∨ Q ⇒ R'\}\]
by (rel-auto)

lemma conditional-iff:
(P ⇒ Q) = (P ⇒ R) \iff \{P ⇒ (Q ⇔ R)\}
by (pred-auto)
lemma p-and-not-p [simp]: \((P \land \neg P) = false\)
  by (pred-auto)

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

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

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

lemma p-imp-false [simp]: \((P \Rightarrow false) = (\neg P)\)
  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:
  \(\neg P \Rightarrow Q \Rightarrow (P \lor Q) = Q\)
  by (pred-auto)

lemma subsumption2:
  \(\neg Q \Rightarrow P \Rightarrow (P \lor Q) = P\)
  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]: true \neq false false \neq true
  by (pred-auto)

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

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

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

lemma taut-iff-eq:
  \(\neg P \Leftrightarrow Q \iff (P = Q)\)
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:
  \((\bigsqcap i | A(i) \cdot P(i)) = (\bigsqcap i \cdot A(i) \land P(i))\)
by (rel-auto)

lemma USUP-true [simp]: \((\biguplus P | F(P) \cdot true) = true\)
by (pred-auto)

lemma UINF-mem-UNIV [simp]: \((\bigsqcap x \in \text{UNIV} \cdot P(x)) = (\bigcap x \cdot P(x))\)
by (pred-auto)

lemma USUP-mem-UNIV [simp]: \((\biguplus x \in \text{UNIV} \cdot P(x)) = (\biguplus x \cdot P(x))\)
by (pred-auto)

lemma USUP-false [simp]: \((\biguplus i \cdot false) = false\)
by (pred-simp)

lemma USUP-mem-false [simp]: \(I \neq \{\} \implies (\biguplus i \in I \cdot false) = false\)
by (rel-simp)

lemma USUP-where-false [simp]: \((\biguplus i | false \cdot P(i)) = true\)
by (rel-auto)

lemma UINF-true [simp]: \((\bigcap i \cdot true) = true\)
by (pred-simp)

lemma UINF-ind-const [simp]:
  \((\bigcap i \cdot P) = P\)
by (rel-auto)

lemma UINF-mem-true [simp]: \(A \neq \{\} \implies (\bigcap i \in A \cdot true) = true\)
by (pred-auto)

lemma UINF-false [simp]: \((\bigcap i | P(i) \cdot false) = false\)
by (pred-auto)

lemma UINF-where-false [simp]: \((\bigcap i | false \cdot P(i)) = false\)
by (rel-auto)

lemma UINF-cong-eq:
\[ \forall x. P_1(x) = P_2(x); \forall x. Q_1(x) \Rightarrow Q_2(x) \Rightarrow \left( \prod x \mid P_1(x) \cdot Q_1(x) = (\prod x \mid P_2(x) \cdot Q_2(x) \right) \]

by (unfold UNINF-def, pred-simp, metis)

lemma UNINF-as-Sup: \( (\prod P \in P \cdot P) = \prod P \)
apply (simp add: upred-defs bop reps eq lit rep eq Sup uexpr-def)
apply (pred-simp)
apply (rule cong[of Sup]
apply (auto)
done

lemma UNINF-as-Sup-collect: \( (\prod P \in A \cdot f(P)) = (\prod f(P)) \)
apply (simp add: upred-defs bop reps eq lit rep eq Sup uexpr-def)
apply (pred-simp)
apply (simp add: Setcompr eq-image)
done

lemma UNINF-as-Sup-collect': \( (\prod f(P)) = (\prod P \cdot f(P)) \)
apply (simp add: upred-defs bop reps eq lit rep eq Sup uexpr-def)
apply (pred-simp)
apply (simp add: full Setcompr eq)
done

lemma UNINF-as-Sup-image: \( (\prod P \mid \langle P \rangle \in \langle A \rangle \cdot f(P)) = \prod (f \cdot A) \)
apply (simp add: upred-defs bop reps eq lit rep eq Sup uexpr-def)
apply (pred-simp)
apply (rule cong[of Sup]
apply (auto)
done

lemma USUP-as-Inf: \( (\bigsqcup P \in P \cdot P) = \bigsqcup P \)
apply (simp add: upred-defs bop reps eq lit rep eq Inf uexpr-def)
apply (pred-simp)
apply (rule cong[of Inf]
apply (auto)
done

lemma USUP-as-Inf-collect: \( (\bigsqcup P \in A \cdot f(P)) = (\bigsqcup f(P)) \)
apply (pred-simp)
apply (simp add: Setcompr eq-image)
done

lemma USUP-as-Inf-collect': \( (\bigsqcup f(P)) = (\bigsqcup P \cdot f(P)) \)
apply (simp add: upred-defs bop reps eq lit rep eq Sup uexpr-def)
apply (pred-simp)
apply (simp add: full Setcompr eq)
done

lemma USUP-as-Inf-image: \( (\bigsqcup P \in P \cdot f(P)) = \bigsqcup (f \cdot P) \)
apply (simp add: upred-defs bop reps eq lit rep eq Inf uexpr-def)
apply (pred-simp)
apply (rule cong[of Inf]
apply (auto)
done

71
lemma USUP-image-eq [simp]: USUP (λi. ≪i ≈ f. λu. λf. g(i))) = (⋃ i∈A. g(f(i)))
  by (pred-simp, rule-tac cong[of Inf Inf], auto)

lemma UINF-image-eq [simp]: UINF (λi. ≪i ≈ f. λu. λf. g(i))) = (⨆ i∈A. g(f(i)))
  by (pred-simp, rule-tac cong[of Sup Sup], auto)

lemma subst-continuous usubst: σ† (⨆ i∈A. g(i)) = (⨆ i∈A. g(i))
  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∈A. P(i))) = (⋂ i∈A. ¬ P(i))
  by (pred-auto)

lemma not-USUP-ind: (¬ (⋃ i∈A. P(i))) = (⋂ i∈A. ¬ 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:
\[(P(n) \land (\bigsqcup i \in \{\text{Suc } n..\} \cdot P(i))) = (\bigsqcup i \in \{n..\} \cdot P(i))\]

**proof**

- **have** \[
\{\text{Suc } n..\} = \{n..\}\]
  - by (auto)
- **thus** \text{?thesis}
  - by (metis USUP-insert conj-upred-def)

**qed**

**lemma** \text{USUP-atLeast-Suc}:

\[
(\bigsqcup i \in \{\text{Suc } m..\} \cdot P(i)) = (\bigsqcup i \in \{m..\} \cdot P(\text{Suc } i))
\]
by (rel-simp, metis (full-types) Suc-le-D not-less-eq-eq)

**lemma** \text{conj-UINF-dist}:

\[
(P \land (\bigsqcap Q \in S \cdot F(Q))) = (\bigsqcap Q \in S \cdot P \land F(Q))
\]
by (simp add: upred-defs bop.eq lit.rep-eq, pred-auto)

**lemma** \text{conj-UINF-ind-dist}:

\[
(P \land (\bigsqcap Q \cdot F(Q))) = (\bigsqcap Q \cdot P \land F(Q))
\]
by (auto)

**lemma** \text{disj-UINF-dist}:

\[
S \neq \{} \Rightarrow (P \lor (\bigsqcap Q \in S \cdot F(Q))) = (\bigsqcap Q \in S \cdot P \lor F(Q))
\]
by (simp add: upred-defs bop.eq lit.rep-eq, pred-auto)

**lemma** \text{UINF-conj-UINF [simp]}:

\[
(\bigsqcap i \in I \cdot P(i)) \lor (\bigsqcap i \in I \cdot Q(i)) = (\bigsqcap i \in I \cdot P(i) \lor Q(i))
\]
by (auto)

**lemma** \text{conj-USUP-dist}:

\[
S \neq \{} \Rightarrow (P \land (\bigsqcup Q \in S \cdot F(Q))) = (\bigsqcup Q \in S \cdot P \land F(Q))
\]

**lemma** \text{USUP-conj-USUP [simp]}:

\[
(\bigsqcup P \in A \cdot F(P)) \land (\bigsqcup P \in A \cdot G(P)) = (\bigsqcup P \in A \cdot F(P) \land G(P))
\]
by (simp add: upred-defs bop.eq lit.rep-eq, pred-auto)

**lemma** \text{UINF-all-cong [cong]}:

\*assumes* \[
P \land F(P) = G(P)
\]
\*shows* \[
(\bigsqcap P \cdot F(P)) = (\bigsqcap P \cdot G(P))
\]
by (simp add: UINF-as-Sup-collect assms)

**lemma** \text{UINF-cong}:

\*assumes* \[
P \land P' \in A \Rightarrow F(P) = G(P)
\]
\*shows* \[
(\bigsqcap P \in A \cdot F(P)) = (\bigsqcap P \in A \cdot G(P))
\]
by (simp add: UINF-as-Sup-collect assms)

**lemma** \text{USUP-all-cong}:

\*assumes* \[
P \land F(P) = G(P)
\]
\*shows* \[
(\bigsqcup P \cdot F(P)) = (\bigsqcup P \cdot G(P))
\]
by (simp add: assms)

**lemma** \text{USUP-cong}:

\*assumes* \[
P \land P \in A \Rightarrow F(P) = G(P)
\]
\*shows* \[
(\bigsqcup P \in A \cdot F(P)) = (\bigsqcup P \in A \cdot G(P))
\]
by (simp add: USUP-as-Inf-collect assms)

73
lemma \( \text{UINF-subset-mono} \): \( A \subseteq B \implies (\bigcap P \in A \cdot F(P)) \subseteq (\bigcap P \in B \cdot F(P)) \)
by (simp add: SUP-subset-mono USUP-as-Inf-collect)

lemma \( \text{USUP-subset-mono} \): \( A \subseteq B \implies (\bigcup P \in A \cdot F(P)) \subseteq (\bigcup P \in B \cdot F(P)) \)
by (simp add: INF-superset-mono USUP-as-Sup-collect)

lemma \( \text{UINF-impl} \): \( (\bigcap P \in A \cdot F(P) \Rightarrow G(P)) = (\bigcup P \in A \cdot F(P)) \Rightarrow (\bigcap P \in A \cdot G(P)) \)
by (pred-simp)

lemma \( \text{USUP-is-forall} \): \( (\bigcup x \cdot P(x)) = (\forall x \cdot P(x)) \)
by (pred-simp)

lemma \( \text{USUP-ind-is-forall} \): \( (\bigcup x \in A \cdot P(x)) = (\forall x \in A \cdot P(x)) \)
by (pred-auto)

lemma \( \text{UINF-is-exists} \): \( (\bigcap x \cdot P(x)) = (\exists x \cdot P(x)) \)
by (pred-simp)

lemma \( \text{UINF-all-nats} \) [simp]:
fixes \( P :: \) nat \( \Rightarrow \) upred
shows \( (\bigcap n \cdot \bigcap i \in \{0..n\} \cdot P(i)) = (\bigcap n \cdot P(n)) \)
by (pred-auto)

lemma \( \text{USUP-all-nats} \) [simp]:
fixes \( P :: \) nat \( \Rightarrow \) upred
shows \( (\bigcup n \cdot \bigcup i \in \{0..n\} \cdot P(i)) = (\bigcup n \cdot P(n)) \)
by (pred-auto)

lemma \( \text{UINF-upto-expand-first} \):
\( (\bigcap i \in \{0..<\text{Suc}(n)\} \cdot P(i)) = (P(0) \vee (\bigcap i \in \{1..<\text{Suc}(n)\} \cdot P(i))) \)
apply (rel-auto)
using not-less by auto

lemma \( \text{UINF-upto-expand-last} \):
\( (\bigcap i \in \{0..<\text{Suc}(n)\} \cdot P(i)) = ((\bigcap i \in \{0..<n\} \cdot P(i)) \vee P(n)) \)
apply (rel-auto)
using less-SucE by blast

lemma \( \text{UINF-Suc-shift} \):
\( (\bigcap i \in \{\text{Suc} 0..<\text{Suc} n\} \cdot P(i)) = (\bigcap i \in \{0..<n\} \cdot P(\text{Suc} i)) \)
apply (rel-simp)
apply (rule cong[of Sup], auto)
using less-Suc-eq-0-disj by auto

lemma \( \text{USUP-upto-expand-first} \):
\( (\bigcup i \in \{0..<\text{Suc}(n)\} \cdot P(i)) = (P(0) \land (\bigcup i \in \{1..<\text{Suc}(n)\} \cdot P(i))) \)
apply (rel-auto)
using not-less by auto

lemma \( \text{USUP-Suc-shift} \):
\( (\bigcup i \in \{\text{Suc} 0..<\text{Suc} n\} \cdot P(i)) = (\bigcup i \in \{0..<n\} \cdot P(\text{Suc} i)) \)
apply (rel-simp)
apply (rule cong[of Inf], auto)
using less-Suc-eq-0-disj by auto

lemma \( \text{UINF-list-conv} \):
\[(\prod i \in \{0..<\text{length}(xs)\} \cdot f (xs ! i)) = \text{foldr} \lor (\text{map} f xs) \text{ false}\]

apply \text{(induct xs)}
apply \text{(rel-auto)}
apply \text{(simp add: UINF-upto-expand-first UINF-Suc-shift)}
done

lemma \text{USUP-list-conv}:
\[(\bigvee i \in \{0..<\text{length}(xs)\} \cdot f (xs ! i)) = \text{foldr} \land (\text{map} f xs) \text{ true}\]

apply \text{(induct xs)}
apply \text{(rel-auto)}
apply \text{(simp-all add: USUP-upto-expand-first USUP-Suc-shift)}
done

lemma \text{UINF-refines}:
assumes \(\forall i. P \sqsubseteq Q(i)\)
shows \(P \sqsubseteq (\bigwedge i \cdot Q(i))\)
using \text{assms}
apply \text{(rel-auto)} using \text{Sup-le-iff} by \text{fastforce}

lemma \text{UINF-pred-ueq [simp]}:
\((\bigwedge x | \ll x \gg = u \cdot P(x)) = (P x)[x\mapsto v]\)
by \text{(pred-auto)}

lemma \text{UINF-pred-lit-eq [simp]}:
\((\bigwedge x | \ll x = v \gg \cdot P(x)) = (P v)\)
by \text{(pred-auto)}

13.3 Equality laws

lemma \text{eq-upred-refl [simp]}: \((x =_u x) = \text{true}\)
by \text{(pred-auto)}

lemma \text{eq-upred-sym}: \((x =_u y) = (y =_u x)\)
by \text{(pred-auto)}

lemma \text{eq-cong-left}:
assumes \text{vwb-lens} x \$x \not\in Q \$x' \not\in Q \$x \not\in R \$x' \not\in R
shows \((\$x' =_u \$x \land Q) = (\$x =_u \$x \land R) \iff (Q = R)\)
using \text{assms}
by \text{(pred-simp, (meson mwb-lens-def vwb-lens-mwb weak-lens-def))}

lemma \text{conj-eg-in-var-subst}:
fixes \(x :: ('a \Rightarrow 'a)\)
assumes \text{vwb-lens} x
shows \((P \land \$x =_u v) = (P[v/$x] \land \$x =_u v)\)
using \text{assms}
by \text{(pred-simp, (metis vwb-lens-wb wb-lens.get-put))}

lemma \text{conj-eg-out-var-subst}:
fixes \(x :: ('a \Rightarrow 'a)\)
assumes \text{vwb-lens} x
shows \((P \land \$x' =_u v) = (P[v/$x'] \land \$x' =_u v)\)
using \text{assms}
by \text{(pred-simp, (metis vwb-lens-wb wb-lens.get-put))}

lemma \text{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-pat, metis (full-types) vwb-lens-wb wb-lens.get-pat)

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-pat, metis (full-types) vwb-lens-wb wb-lens.get-pat)

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 \text{var } x =_u v) = (P[v/x] \land \text{var } x =_u v)
using assms
by (pred-simp, (metis (full-types) vwb-lens-def wb-lens.get-pat)+)

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 ] \Longrightarrow shEx P = shEx Q
by (pred-auto)

lemma shAll-unbound [simp]: (\forall \ x \cdot P) = P
by (pred-auto)

lemma shAll-bool [simp]: shAll P = (P True \land P False)
by (pred-simp, metis (full-types))

lemma shAll-cong: [ \land \ x. P \ x = Q \ x ] \Longrightarrow shAll P = shAll Q
by (pred-auto)

Quantifier lifting

named-theorems uquant-lift

lemma shEx-lift-conj-1 [uquant-lift]:
((∃x·P(x)) ∧ Q) = (∃x·P(x) ∧ Q)
by (pred-auto)

lemma shEx-lift-conj-2 [uquant-lift]:
(P ∧ (∃x·Q(x))) = (∃x·P ∧ Q(x))
by (pred-auto)

13.5 Case Splitting

lemma eq-split-subst:
assumes vwb-lens x
shows (P = Q) ↔ (∀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 ∧ 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' ↔ (∀v·'P[v/x]')
using assms
by (pred-auto, metis vwb-lens-wb wb-lens.source-stability)

lemma eq-split:
assumes 'P ⇒ Q' 'Q ⇒ P'
shows P = Q
using assms
by (pred-auto)

lemma bool-eq-splitI:
assumes vwb-lens x P[true/x] = Q[true/x] 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] ∧ P[true/x]))'
proof -
from assms have 'P' = (∀v·'P[v/x]')
  by (subst taut-split-subst[of x], auto)
also have ... = ('P[True/x]' ∧ 'P[False/x]')
  by (metis (mono-tags, lifting))
also have ... = ('P[false/x] ∧ P[true/x])'
  by (pred-auto)
finally show ?thesis .
qed

lemma subst-eq-replace:
fixes x :: ('a :: 'a)
shows (p[u/x] ∧ u =_u v) = (p[v/x] ∧ u =_u v)
by (pred-auto)
13.6 UTP Quantifiers

lemma one-point:
  assumes mwb-lens $x \vartriangleleft y$
  shows $(\exists x \cdot x \cdot P \land \text{var } x = u \cdot v) = P[v/x]$
  using assms
  by (pred-auto)

lemma exists-twice: $\text{mwb-lens } x \implies (\exists x \cdot \exists x \cdot P) = (\exists x \cdot P)$
  by (pred-auto)

lemma all-twice: $\text{mwb-lens } x \implies (\forall x \cdot \forall x \cdot P) = (\forall x \cdot P)$
  by (pred-auto)

lemma exists-sub: $[\text{mwb-lens } y; x \subseteq L y] \implies (\exists x \cdot \exists y \cdot P) = (\exists y \cdot P)$
  by (pred-auto)

lemma all-sub: $[\text{mwb-lens } y; x \subseteq L y] \implies (\forall x \cdot \forall y \cdot P) = (\forall y \cdot P)$
  by (pred-auto)

lemma ex-commute:
  assumes $x \vartriangleright y$
  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 $x \vartriangleright y$
  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-equiv:
  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-equiv:
  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 \textit{ex-plus}:  
\((\exists\; y \cdot x \cdot P) = (\exists\; x \cdot \exists\; y \cdot P)\)  
by (pred-auto) 

lemma \textit{all-plus}:  
\((\forall\; y \cdot x \cdot P) = (\forall\; x \cdot \forall\; y \cdot P)\)  
by (pred-auto) 

lemma \textit{closure-all}:  
\([P]_u = (\forall\; \Sigma \cdot P)\)  
by (pred-auto) 

lemma \textit{unrest-as-exists}:  
\(vweb\text{-}lens\; x = \Rightarrow (x \nmid P) \iff (\exists\; x \cdot P) = P\)  
by (pred-simp, metis vweb-lens.put-eq) 

lemma \textit{ex-mono}:  
\(P \sqsubseteq Q = \Rightarrow (\exists\; x \cdot P) \sqsubseteq (\exists\; x \cdot Q)\)  
by (pred-auto) 

lemma \textit{ex-weakens}:  
\(wb\text{-}lens\; x = \Rightarrow (\exists\; x \cdot P) \sqsubseteq P\)  
by (pred-simp, metis wb-lens.get-put) 

lemma \textit{all-mono}:  
\(P \sqsubseteq Q = \Rightarrow (\forall\; x \cdot P) \sqsubseteq (\forall\; x \cdot Q)\)  
by (pred-auto) 

lemma \textit{all-strengthens}:  
\(wb\text{-}lens\; x = \Rightarrow P \sqsubseteq (\forall\; x \cdot P)\)  
by (pred-simp, metis wb-lens.get-put) 

lemma \textit{ex-unrest}:  
\(x \nmid P \Rightarrow (\exists\; x \cdot P) = P\)  
by (pred-auto) 

lemma \textit{all-unrest}:  
\(x \nmid P \Rightarrow (\forall\; x \cdot P) = P\)  
by (pred-auto) 

lemma \textit{not-ex-not}:  
\(\neg (\exists\; x \cdot \neg P) = (\forall\; x \cdot P)\)  
by (pred-auto) 

lemma \textit{not-all-not}:  
\(\neg (\forall\; x \cdot \neg P) = (\exists\; x \cdot P)\)  
by (pred-auto) 

lemma \textit{ex-conj-contr-left}:  
\(x \nmid P \Rightarrow (\exists\; x \cdot P \land Q) = (P \land (\exists\; x \cdot Q))\)  
by (pred-auto) 

lemma \textit{ex-conj-contr-right}:  
\(x \nmid Q \Rightarrow (\exists\; x \cdot P \land Q) = ((\exists\; x \cdot P) \land Q)\)  
by (pred-auto) 

13.7 Variable Restriction 

lemma \textit{var-res-all}:  
\(P \mid_v \Sigma = P\)  
by (rel-auto) 

lemma \textit{var-res-twice}:  
\(mwb\text{-}lens\; x = \Rightarrow P \mid_v x \mid_v x = P \mid_v x\)  
by (pred-auto)
13.8 Conditional laws

**lemma** cond-def:

\[(P \land b \triangleright Q) = ((b \land P) \lor ((\neg b) \land Q))\]

by (pred-auto)

**lemma** cond-idem [simp]: \((P \land b \triangleright P) = P\) by (pred-auto)

**lemma** cond-true-false [simp]: \(true \land b \triangleright false = b\) by (pred-auto)

**lemma** cond-symm: \((P \land b \triangleright Q) = (Q \land \neg b \triangleright P)\) by (pred-auto)

**lemma** cond-assoc: \(((P \land b \triangleright Q) \land c \triangleright (R \land S)) = ((P \land b \triangleright R) \land (Q \land b \triangleright S))\) by (pred-auto)

**lemma** cond-distr: \((P \land b \triangleright (Q \land c \triangleright R)) = ((P \land b \triangleright Q) \land c \triangleright (P \land b \triangleright R))\) by (pred-auto)

**lemma** cond-unit-T [simp]: \((P \land false \triangleright Q) = Q\) by (pred-auto)

**lemma** cond-unit-F [simp]: \((P \land false \triangleright Q) = Q\) by (pred-auto)

**lemma** cond-conj-not: \(((P \land b \triangleright Q) \land (\neg b)) = (Q \land (\neg b))\)

by (rel-auto)

**lemma** cond-and-T-integrate:

\(((P \land b) \lor (Q \land b \triangleright R)) = ((P \lor Q) \land b \triangleright R)\)

by (pred-auto)

**lemma** cond-L6: \((P \land b \triangleright (Q \land b \triangleright R)) = (P \land b \triangleright R)\) by (pred-auto)

**lemma** cond-L7: \((P \land b \triangleright (P \land c \triangleright Q)) = (P \land b \lor c \triangleright Q)\) by (pred-auto)

**lemma** cond-and-distr: \(((P \land Q) \land b \triangleright (R \land S)) = ((P \land b \triangleright R) \land (Q \land b \triangleright S))\) by (pred-auto)

**lemma** cond-or-distr: \(((P \lor Q) \land b \triangleright (R \lor S)) = ((P \land b \triangleright R) \lor (Q \land b \triangleright S))\) by (pred-auto)

**lemma** cond-imp-distr:

\(((P \Rightarrow Q) \land b \triangleright (R \Rightarrow S)) = ((P \land b \triangleright R) \Rightarrow (Q \land b \triangleright S))\) by (pred-auto)

**lemma** cond-eq-distr:

\(((P \iff Q) \land b \triangleright (R \iff S)) = ((P \land b \triangleright R) \iff (Q \land b \triangleright S))\) by (pred-auto)

**lemma** cond-conj-impl: \((P \land (Q \land b \triangleright S)) = ((P \land Q) \land b \triangleright (P \land S))\) by (pred-auto)

**lemma** cond-disj-impl: \((P \lor (Q \land b \triangleright S)) = ((P \lor Q) \land b \triangleright (P \lor S))\) by (pred-auto)

**lemma** cond-impl: \(\neg (P \land b \triangleright Q) = ((\neg P) \land b \triangleright (\neg Q))\) by (pred-auto)

**lemma** cond-conj: \(P \land b \land c \triangleright Q = (P \land c \triangleright Q) \land b \triangleright Q\)

by (pred-auto)

**lemma** spec-cond-distr: \((P \Rightarrow (Q \land b \triangleright R)) = ((P \Rightarrow Q) \land b \triangleright (P \Rightarrow R))\)

by (pred-auto)

**lemma** cond-USUP-dist: \((\bigsqcup P \in S \cdot F(P)) \land b \triangleright (\bigsqcup P \in S \cdot G(P)) = (\bigsqcup P \in S \cdot F(P) \land b \triangleright G(P))\)

by (pred-auto)

80
lemma cond-UNIF-dist: \((\prod P \in S \cdot F(P)) <\ b \triangleright (\prod P \in S \cdot G(P)) = (\prod P \in S \cdot F(P) <\ b \triangleright G(P))\)
by (pred-auto)

lemma cond-var-subst-left:
assumes vwb-lens x
shows \((P[\text{true}/x] <\ var x \triangleright Q) = (P <\ var x \triangleright Q)\)
using assms by (pred-auto, metis (full-types) vwb-lens-wb wb-lens.get-put)

lemma cond-var-subst-right:
assumes vwb-lens x
shows \((P <\ var x \triangleright Q[\text{false}/x]) = (P <\ var x \triangleright Q)\)
using assms by (pred-auto, metis (full-types) vwb-lens.put-eq)

lemma cond-var-split:
vwb-lens x \implies (P[true/x] <\ var x \triangleright P[false/x]) = P
by (rel-simp, metis (full-types) vwb-lens.put-eq)+

lemma cond-assign-subst:
vwb-lens x \implies (P <\ \text{utp-exp}\cdot\var x =_u v \triangleright Q) = (P[v/x] <\ \text{utp-exp}\cdot\var x =_u v \triangleright Q)
apply (rel-simp) using vwb-lens.put-eq by force

lemma conj-conds:
\((P1 <\ b \triangleright Q1 \land P2 <\ b \triangleright Q2) = (P1 \land P2) <\ b \triangleright (Q1 \land Q2)\)
by pred-auto

lemma disj-conds:
\((P1 <\ b \triangleright Q1 \lor P2 <\ b \triangleright Q2) = (P1 \lor P2) <\ b \triangleright (Q1 \lor Q2)\)
by pred-auto

lemma cond-mono:
\[(P1 \subseteq P2;\ Q1 \subseteq Q2) \implies (P1 <\ b \triangleright Q1) \subseteq (P2 <\ b \triangleright Q2)\]
by (rel-auto)

lemma cond-mono-antisym:
\[(\text{mono } P;\ \text{mono } Q) \implies \text{mono } (\lambda X.\ P X <\ b \triangleright Q X)\]
by (simp add: mono-def, rel-blast)

Sometimes Isabelle desugars conditionals, and the following law undoes this

lemma resugar-cond: \(\text{trop } (\lambda b\ P\ Q,\ (b \rightarrow P) \land (\neg b \rightarrow Q))\ b\ P\ Q = \text{cond } P\ b\ Q\)
by (transfer, auto simp add: fun-eq-iff)

13.9 Additional Expression Laws

lemma le-pred-refl [simp]:
fixes x :: ('a::preorder, 'a) uexpr
shows \((x \leq_u x) = \text{true}\)
by (pred-auto)

lemma uzero-le-laws [simp]:
(\emptyset :: ('a::{linordered-semidom}, 'a) uexpr) \leq_u \text{numeral } x = \text{true}
(1 :: ('a::{linordered-semidom}, 'a) uexpr) \leq_u \text{numeral } x = \text{true}
(\emptyset :: ('a::{linordered-semidom}, 'a) uexpr) \leq_u 1 = \text{true}
by (pred-simp)+

lemma unumeral-le-1 [simp]:
assumes \((\text{numeral } i :: ('a::{numeral,ord})) \leq \text{numeral } j\)
shows \((\text{numeral } i :: ('a, 'a) \text{ uexpr}) \leq_u \text{ numeral } j = \text{ true}\)
using assms by (pred-auto)

lemma unumeral-le-2 [simp]:
assumes \((\text{numeral } i :: 'a::\{\text{numeral,linorder}\}) > \text{ numeral } j\)
shows \((\text{numeral } i :: ('a, 'a) \text{ uexpr}) \leq_u \text{ numeral } j = \text{ false}\)
using assms by (pred-auto)

lemma useset-laws [simp]:
\[\begin{align*}
x &\in_u \{\}_u = \text{ false} \\
x &\in_u \{m..n\}_u = (m \leq_u x \land x \leq_u n)
\end{align*}\]
by (pred-auto)

lemma pfun-entries-apply [simp]:
\[\begin{align*}
\text{entr}_u (d,f) :: (('k, 'v) \text{ pfun}, 'a) \text{ uexpr} (i)_a &= ((f(i))_a \land i \in_u d \lor \bot_u)
\end{align*}\]
by (pred-auto)

lemma uapply-uupdate-pfun [simp]:
\[\begin{align*}
\text{fixes } m :: (('k, 'v) \text{ pfun}, 'a) \text{ uexpr} \\
\text{shows } m(k \mapsto v)_u(i) &= v \land i =_u k \land m(i)_a
\end{align*}\]
by (rel-auto)

lemma ulit-eq [simp]: \(x = y \Rightarrow (\ll x \gg_u =_u \ll y \gg_u) = \text{ true}\)
by (rel-auto)

lemma ulit-neq [simp]: \(x \neq y \Rightarrow (\ll x \gg_u \neq_\ll y \gg_u) = \text{ false}\)
by (rel-auto)

lemma useset-mems [simp]:
\[\begin{align*}
x &\in_u \{y\}_u = (x =_u y) \\
x &\in_u \{A \cup B\}_u = (x \in_u A \lor x \in_u B) \\
x &\in_u \{A \cap B\}_u = (x \in_u A \land x \in_u B)
\end{align*}\]
by (rel-auto)

13.10 Refinement By Observation

Function to obtain the set of observations of a predicate

definition obs-apred :: 'a \text{ upred} \Rightarrow 'a \text{ set} ([\_])_o
where [upred-defs]: \([P]_o = \{b. \ll [P]_u b\}\)

lemma obs-apred-refine-iff:
\(P \subseteq Q \iff [Q]_o \subseteq [P]_o\)
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 $x \triangleright y$ bij-lens $(x + L y) \triangleright y \
\not\in P y \not\in Q \{ v. \ 'P[<v>/x]\}' \subseteq \{ v. \ 'Q[<v>/x]\}'$
shows $Q \subseteq P$
using $\text{assms}(3-5)$
apply $(\text{simp add: obs-upred-refine-iff subset-eq})$
apply $(\text{pred-simp})$
apply $(\text{rename-tac } b)$
apply $(\text{drule-tac } x\Rightarrow \text{get } x \text{ in spec})$
apply $(\text{auto simp add: assms})$
apply $(\text{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 $C1$: $(\exists x \cdot \text{false}) = \text{false}$
by $(\text{pred-auto})$

lemma $C2$: $\text{wb-lens } x \Rightarrow 'P \Rightarrow (\exists x \cdot P)$
by $(\text{pred-simp, metis wb-lens.get-put})$

lemma $C3$: $\text{mbw-lens } x \Rightarrow (\exists x \cdot (P \land (\exists x \cdot Q))) = ((\exists x \cdot P) \land (\exists x \cdot Q))$
by $(\text{pred-auto})$

lemma $C4a$: $x \approx_L y \Rightarrow (\exists x \cdot \exists y \cdot P) = (\exists y \cdot \exists x \cdot P)$
by $(\text{pred-simp, metis (no-types, lifting) lens.select-convs(2)}+)$$+

lemma $C4b$: $x \triangleright y \Rightarrow (\exists x \cdot \exists y \cdot P) = (\exists y \cdot \exists x \cdot P)$
using $\text{ex-commute by blast}$

lemma $C5$: fixes $x :: ('a \Rightarrow 'a)$
shows $(\&x =_u &x) = \text{true}$
by $(\text{pred-auto})$

lemma $C6$: assumes $\text{wb-lens } x x \triangleright y x \triangleright z$
shows $(\&y =_u &z) = (\exists x \cdot &y =_u &x \land &x =_u &z)$
using $\text{assms}$
by $(\text{pred-simp, (metis lens-indep-def)+})$

lemma $C7$: assumes weak-lens $x x \triangleright y$
shows $((\exists x \cdot &x =_u &y \land P) \land (\exists x \cdot &x =_u &y \land \neg P)) = \text{false}$
using $\text{assms}$
by $(\text{pred-simp, simp add: lens-indep-sym})$

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.

defined as 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 $H$ ≡ ($H(P) = P$)

lemma Healthy-def' $P$ is $H$ ←→ ($H(P) = P$)
unfolding Healthy-def by auto

lemma Healthy-if: $P$ is $H$ =⇒ ($H(P) = P$)
unfolding Healthy-def by auto

lemma Healthy-intro: $H(P) = P$ =⇒ $P$ is $H$
by (simp add: Healthy-def)

declare Healthy-def' [upred-defs]

abbreviation Healthy-carrier :: 'α health ⇒ 'α upred set ([[]]H)
where $H$ ≡{$P$. $P$ is $H$}

lemma Healthy-carrier-image:
$A \subseteq [H]_H \Longrightarrow H A = A$
by (auto simp add: image-def, (metis Healthy-if mem-Collect-eq subsetCE)+)

lemma Healthy-carrier-Collect: $A \subseteq [H]_H \Longrightarrow A = \{H(P) \mid P. P \in A\}$
by (simp add: Healthy-carrier-image Setcompr-eq-image)

lemma Healthy-func:
$[F \in [H]_H \rightarrow [H]_H; P is H_1 ] \Longrightarrow H_2(F(P)) = F(P)$
using Healthy-if by blast

lemma Healthy-comp:
$[P is H_1; P is H_2 ] \Longrightarrow P is H_1 \circ H_2$
by (simp add: Healthy-def)

lemma Healthy-apply-closed:
assumes $F \in [H]_H \rightarrow [H]_H$ $P$ is $H$
shows $F(P)$ is $H$
using assms(1) assms(2) by auto

lemma Healthy-set-image-member:
$[P \in F \cdot A; \bigwedge x. F x is H ] \Longrightarrow P is H$
by blast

lemma Healthy-SUPREMUM:
$A \subseteq [H]_H \Longrightarrow SUPREMUM A H = \bigsqcap A$
by (drule Healthy-carrier-image, presburger)

lemma Healthy-INFIMUM:
$A \subseteq [H]_H \Longrightarrow INFIMUM A H = \bigsqcup A$
by (drule Healthy-carry-image, presburger)

lemma Healthy-\text{nu} [\text{closure}]:
  \text{assumes} \ mono \ F \ F \in [id]_H \rightarrow [H]_H
  \text{shows} \ \nu \ F \text{ is } H
  \text{by} (metis (mono-tags) Healthy-def Healthy-func assms eq-id-iff lfp-unfold)

lemma Healthy-\text{mu} [\text{closure}]:
  \text{assumes} \ mono \ F \ F \in [id]_H \rightarrow [H]_H
  \text{shows} \ \mu \ F \text{ is } H
  \text{by} (metis (mono-tags) Healthy-def Healthy-func assms eq-id-iff gfp-unfold)

lemma Healthy-subset-member: \ [ A \subseteq [H]_H; P \in A ] \Rightarrow H(P) = P
  \text{by} (meson Ball-Collect Healthy-if)

lemma is-Healthy-subset-member: \ [ A \subseteq [H]_H; P \in A ] \Rightarrow P \text{ is } H
  \text{by} blast

14.2 Properties of Healthiness Conditions

definition Idempotent :: 'a health \Rightarrow \text{bool} \text{ where}
  Idempotent(H) \leftrightarrow (\forall P. H(H(P)) = H(P))

abbreviation Monotonic :: 'a health \Rightarrow \text{bool} \text{ where}
  Monotonic(H) \equiv mono H

definition IMH :: 'a health \Rightarrow \text{bool} \text{ where}
  IMH(H) \leftrightarrow Idempotent(H) \land Monotonic(H)

definition Antitone :: 'a health \Rightarrow \text{bool} \text{ where}
  Antitone(H) \leftrightarrow (\forall P \ Q. Q \subseteq P \rightarrow (H(P) \subseteq H(Q)))

definition Conjunctive :: 'a health \Rightarrow \text{bool} \text{ where}
  Conjunctive(H) \leftrightarrow (\exists Q. \forall P. H(P) = (P \land Q))

definition FunctionalConjunctive :: 'a health \Rightarrow \text{bool} \text{ where}
  FunctionalConjunctive(H) \leftrightarrow (\exists F. \forall P. H(P) = (P \land F(P)) \land Monotonic(F))

definition WeakConjunctive :: 'a health \Rightarrow \text{bool} \text{ where}
  WeakConjunctive(H) \leftrightarrow (\forall P. \exists Q. H(P) = (P \land Q))

definition Disjunctuous :: 'a health \Rightarrow \text{bool} \text{ where}
  [\upred-defs]: Disjunctuous H = (\forall P \ Q. H(P \land Q) = (H(P) \land H(Q)))

definition Continuous :: 'a health \Rightarrow \text{bool} \text{ where}
  [\upred-defs]: Continuous H = (\forall A. A \neq \{\} \rightarrow H (\bigsqcap A) = \bigsqcap \{ H ' A \})

lemma Healthy-Idempotent [\text{closure}]:
  Idempotent H \Rightarrow H(P) \text{ is } H
  \text{by} (simp add: Healthy-def Idempotent-def)

lemma Healthy-range: Idempotent H \Rightarrow \text{range } H = [H]_H
  \text{by} (auto simp add: image-def Healthy-if Healthy-Idempotent, metis Healthy-if)

lemma Idempotent-id [simp]: Idempotent id
  \text{by} (simp add: Idempotent-def)

85
lemma Idempotent-comp [intro]:
[\[ \text{Idempotent } f; \text{Idempotent } g; f \circ g = g \circ f \] \Rightarrow \text{Idempotent } (f \circ g)]
by (auto simp add: Idempotent-def comp-def, metis)

lemma Idempotent-image: Idempotent f \Rightarrow f \cdot f \cdot A = f \cdot A
by (metis (mono-tags, lifting) Idempotent-def image-cong image-image)

lemma Monotonic-id [simp]: Monotonic id
by (simp add: monoI)

lemma Monotonic-id’ [closure]:
mono (\lambda 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) \triangleleft b \triangleright Q(X))
by (simp add: assms cond-monotonic)

lemma Conjunctive-Idempotent:
Conjunctive(H) \Rightarrow Idempotent(H)
by (auto simp add: Conjunctive-def Idempotent-def)

lemma Conjunctive-Monotonic:
Conjunctive(H) \Rightarrow Monotonic(H)
unfolding Conjunctive-def mono-def
using dual-order.trans by fastforce

lemma Conjunctive-conj:
assumes Conjunctive(HC)
s 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.phact)

lemma Conjunctive-distr-conj:
assumes Conjunctive(HC)
s 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

lemma Conjunctive-distr-cond:
assumes Conjunctive(HC)
shows \( HC(P \bowtie b \bowtie Q) = (HC(P) \bowtie b \bowtie HC(Q)) \)
using assms unfolding Conjunctive-def
by (metis cond-conj-distr utp-pred-laws.inf-commute)

lemma FunctionalConjunctive-Monotonic:
FunctionalConjunctive(H) \implies Monotonic(H)
unfolding FunctionalConjunctive-def by (metis mono-def utp-pred-laws.inf-mono)

lemma WeakConjunctive-Refinement:
assumes WeakConjunctive(HC)
shows \( P \subseteq HC(P) \)
using assms unfolding WeakConjunctive-def by (metis utp-pred-laws.inf.cobounded1)

lemma WeakConjunctive-Healthy-Refinement:
assumes WeakConjunctive(HC) and \( P \) is HC
shows \( HC(P) \subseteq P \)
using assms unfolding WeakConjunctive-def Healthy-def by simp

declare Conjunctive-def [upred-defs]
declare mono-def [upred-defs]

lemma Disjunctuous-Monotonic: Disjunctuous H \implies Monotonic H
by (metis Disjunctuous-def mono-def semilattice-sup-class.le-iff-sup)

lemma ContinuousD [dest]: \( [\text{Continuous } H; A \neq \{\} ] \implies H (\bigsqcap A) = (\bigsqcap \{P \in A. H(P)\}) \)
by (simp add: Continuous-def)

lemma Continuous-Disjunctuous: Continuous H \implies 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 \implies Monotonic H
by (simp add: Continuous-Disjunctuous Disjunctuous-Monotonic)

lemma Continuous-comp [intro]:
\( [\text{Continuous } f; \text{Continuous } g ] \implies \text{Continuous } (f \circ g) \)
by (simp add: Continuous-def)

lemma Continuous-const [closure]: Continuous (\lambda X. P)
by pred-auto

87
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]:
  [ Continuous H; \( i, i \in A \Rightarrow P(i) \) is H; A ≠ {} ] \( (\bigcap \ i \in A. P(i)) \) 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]:
  [ Continuous H; \( i, i \in A \Rightarrow P(i) \) is H; A ≠ {} ] \( (\bigcap \ i \in A \cdot P(i)) \) is H
  by (simp add: Sup-Continuous-closed UINF-as-Sup-collect)

lemma UINF-mem-Continuous-closed-pair [closure]:
  assumes Continuous H \( \bigwedge i, j. (i, j) \in A \Rightarrow P(i, j) \) is H A ≠ {}
  shows \( (\bigcap \ (i,j) \in A \cdot P(i, j)) \) is H
  proof –
    have \( (\bigcap \ i, j \in A \cdot P(i, j)) = (\bigcap \ x \in A \cdot P(fst x) (snd x)) \)
      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 \( \bigwedge i, j, k. (i, j, k) \in A \Rightarrow P(i, j, k) \) is H
  shows \( (\bigcap \ (i,j,k) \in A \cdot P(i,j,k)) \) is H
  proof –
    have \( (\bigcap \ i, j, k \in A \cdot P(i, j, k)) = (\bigcap \ x \in A \cdot P(fst x) (fst (snd x)) (snd (snd x))) \)
      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-Continuous-closed [closure]:
  [ Continuous H; \( i, P(i) \) is H ] \( (\bigcap \ i \cdot P(i)) \) is H
  using UINF-membr-Continuous-closed[of H UNIV P]
  by (simp add: UINF-mem-UNIV)

All continuous functions are also Scott-continuous

lemma sup-continuous-Continuous [closure]: Continuous F \( \Rightarrow \) sup-continuous F
  by (simp add: Continuous-def sup-continuous-def)

lemma USUP-healthy: A ⊆ [H]H \( \Rightarrow \) (\bigcap \ P \in A \cdot F(P)) = (\bigcap \ P \in A \cdot F(H(P)))
  by (rule USUP-cong, simp add: Healthy-subset-member)

lemma UINF-healthy: A ⊆ [H]H \( \Rightarrow \) (\bigcap \ P \in A \cdot F(P)) = (\bigcap \ P \in A \cdot F(H(P)))
  by (rule UINF-cong, simp add: Healthy-subset-member)
end
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 [14].

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 \( \text{fst}_L \) and \( \text{snd}_L \).

**definition** \( \text{in}_\alpha :: ('\alpha \Rightarrow '\alpha \times '\beta) \text{ where} \)

\[ \text{[lens-defs]: } \text{in}_\alpha = \text{fst}_L \]

**definition** \( \text{out}_\alpha :: ('\beta \Rightarrow '\alpha \times '\beta) \text{ where} \)

\[ \text{[lens-defs]: } \text{out}_\alpha = \text{snd}_L \]

**lemma** \( \text{in}_\alpha\text{-uvar} \text{ [simp]}: \text{vwb-lens } \text{in}_\alpha \)

by (unfold-locales, auto simp add: \( \text{in}_\alpha\text{-def} \))

**lemma** \( \text{out}_\alpha\text{-uvar} \text{ [simp]}: \text{vwb-lens } \text{out}_\alpha \)

by (unfold-locales, auto simp add: \( \text{out}_\alpha\text{-def} \))

**lemma** \( \text{var-in-alpha} \text{ [simp]}: x ;_L \text{in}_\alpha = \text{ivar } x \)

by (simp add: \( \text{fst-lens-def in}_\alpha\text{-def in-var-def} \))

**lemma** \( \text{var-out-alpha} \text{ [simp]}: x ;_L \text{out}_\alpha = \text{ovar } x \)

by (simp add: \( \text{out}_\alpha\text{-def out-var-def snd-lens-def} \))

**lemma** \( \text{drop-pre-inv} \text{ [simp]}: [ \text{out}_\alpha \not\subseteq \sigma \] \Rightarrow \llbracket \llbracket \sigma \rrbracket \rrbracket \sigma = p \)

by (pred-simp)

**lemma** \( \text{usubst-lookup-ivar-unrest} \text{ [usubst]}: \)

\[ \text{in}_\alpha \not\subseteq \sigma \Rightarrow \langle \sigma \rangle_s (\text{ivar } x) = \$x \]

by (rel-simp, metis \( \text{fstI} \))

**lemma** \( \text{usubst-lookup-ovar-unrest} \text{ [usubst]}: \)

\[ \text{out}_\alpha \not\subseteq \sigma \Rightarrow \langle \sigma \rangle_s (\text{ovar } x) = \$x' \]

by (rel-simp, metis \( \text{sndI} \))

**lemma** \( \text{out-alpha-in-indep} \text{ [simp]}: \)

\( \text{out}_\alpha \bowtie \text{in-var } x \text{ in-var } x \bowtie \text{out}_\alpha \)

by (simp-all add: \( \text{in-var-def out}_\alpha\text{-def lens-indep-def fst-lens-def snd-lens-def lens-comp-def} \))

**lemma** \( \text{in-alpha-out-indep} \text{ [simp]}: \)

\( \text{in}_\alpha \bowtie \text{out-var } x \text{ out-var } x \bowtie \text{in}_\alpha \)

by (simp-all add: \( \text{in-var-def in}_\alpha\text{-def lens-indep-def fst-lens-def lens-comp-def} \))
The following two functions lift a predicate substitution to a relational one.

**abbreviation usubst-rel-lift :: 'α usubst ⇒ ('α × 'β) usubst ([|]_s) where**

\[ [σ]_s ≡ σ ⊕_s inα \]

**abbreviation usubst-rel-drop :: ('α × 'α) usubst ⇒ 'α usubst ([|]_s) where**

\[ [σ]_s ≡ σ \restriction_s inα \]

The alphabet of a relation then consists wholly of the input and output portions.

**lemma alpha-in-out:**

\[ Σ ≈ L_{inα} + L_{outα} \]

by (simp add: fst-snd-id-lens inα-le-def lens-equiv-refl outα-le-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**

**type-synonym ('α, 'β) urel = ('α × 'β) upred**

**type-synonym 'α hrel = ('α × 'α) upred**

**type-synonym ('a, 'α) hexpr = ('a, 'α × 'α) uexpr**

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 ; 71)

uassigns :: 'a usubst ⇒ 'b ((\cdot)_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 ⇒ (\alpha, \beta) urel**

\((\alpha - a - \triangleright_r / -) [52,0,53] 52)\]

**where** \((P a b r Q) ≡ (P a [b]_< \triangleright Q)\)

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 (op O). Since this returns a set, the definition states that the state binding \(b\) is an element of this set.

**definition seqr ([|]_<) where**

\[ [upred-defs]: \ [b]_< = \ [b]_< \]

**abbreviation seqr :: ('α, 'β) urel ⇒ (\alpha, \gamma) urel ⇒ ('α × 'γ) upred**

\((\beta - a - \triangleright_r / -) [52,0,53] 52)\]

**where** \((P a b r Q) ≡ (P a [b]_< \triangleright Q)\)

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 ⇒ (\alpha, \beta) urel**

\((\alpha - a - \triangleright_r / -) [52,0,53] 52)\]

**where** \((P a b r Q) ≡ (P a [b]_< \triangleright Q)\)

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 (op O). Since this returns a set, the definition states that the state binding \(b\) is an element of this set.

**definition seqr ([|]_<) where**

\[ [upred-defs]: \ [b]_< = \ [b]_< \]

**abbreviation seqr :: ('α, 'β) urel ⇒ (\alpha, \gamma) urel ⇒ ('α × 'γ) upred**

\((\beta - a - \triangleright_r / -) [52,0,53] 52)\]

**where** \((P a b r Q) ≡ (P a [b]_< \triangleright Q)\)

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 (op O). Since this returns a set, the definition states that the state binding \(b\) is an element of this set.
We also set up a homogeneous sequential composition operator, and versions of \texttt{true} and \texttt{false} that are explicitly typed by a homogeneous alphabet.

**abbreviation** \texttt{seqh} :: \texttt{\'a hrel \Rightarrow \'a hrel (infix \_\_ 71)} where

\texttt{seqh P Q \equiv (P \_\_ Q)}

**abbreviation** \texttt{truer} :: \texttt{\'a hrel (true\_h)} where

\texttt{truer \equiv true}

**abbreviation** \texttt{falser} :: \texttt{\'a hrel (false\_h)} where

\texttt{falser \equiv false}

We define the relational converse operator as an alphabet extrusion on the bijective lens \texttt{swapL} that swaps the elements of the product state-space.

**abbreviation** \texttt{conv-r} :: \texttt{(\'a, \'a \times \'b hrel \Rightarrow \'a \times \'b hrel)}

\texttt{where conv-r e \equiv e \oplus_p swapL}

Assignment is defined using substitutions, where latter defines what each variable should map to. The definition of the operator identifies the after state binding, \(b'\), with the substitution function applied to the before state binding \(b\).

**lift-definition** \texttt{assigns-r} :: \texttt{\'a usubst \Rightarrow \'a hrel}

\is\lambda \sigma (b, b') \cdot b' = \sigma(b)

**adhoc-overloading**

\texttt{uassigns assigns-r}

Relational identity, or skip, is then simply an assignment with the identity substitution: it simply identifies all variables.

**definition** \texttt{skip-r} :: \texttt{\'a hrel}

[urel-defs]: \texttt{skip-r = assigns-r id}

**adhoc-overloading**

\texttt{uskip skip-r}

We set up iterated sequential composition which iterates an indexed predicate over the elements of a list.

**definition** \texttt{seqr-iter} :: \texttt{\'a list \Rightarrow (\'a \Rightarrow \'b hrel) \Rightarrow \'b hrel}

[urel-defs]: \texttt{seqr-iter xs P = foldr (\lambda i Q. P(i) \_\_ Q) xs II}

A singleton assignment simply applies a singleton substitution function, and similarly for a double assignment.

**abbreviation** \texttt{assign-r} :: \texttt{(\'t \Rightarrow \'a) \Rightarrow (\'t, \'a) uexpr \Rightarrow \'a hrel}

\texttt{where assign-r x v \equiv \langle x \mapsto_s v \rangle_a}

**abbreviation** \texttt{assign-2-r} ::

\texttt{(\'t1 \Rightarrow \'a) \Rightarrow (\'t2 \Rightarrow \'a) \Rightarrow (\'t1, \'a) uexpr \Rightarrow (\'t2, \'a) uexpr \Rightarrow \'a hrel}

\texttt{where assign-2-r x y u v \equiv assigns-r [x \mapsto_s u, y \mapsto_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** \texttt{skip-ra} :: \texttt{(\'\beta, \'\alpha) lens \Rightarrow \'a hrel}

[urel-defs]: \texttt{skip-ra v = (\$v' =_u \$v)}

Similarly, we define the alphabetised assignment operator.
definition assigns-ra :: 'α usubst ⇒ ('β, 'α) lens ⇒ 'α hrel (⊥→) where
(σ)α = (〔σ〕α ⊔ 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.

definition rassume :: 'α upred ⇒ 'α hrel (⊥→) where
[urel-defs]: rassume c = II ⊔ c ∨ r, false

definition rassert :: 'α upred ⇒ 'α hrel (⊥→) where
[urel-defs]: rassert c = II ⊓ c ∨ r, true

A test is like a precondition, except that it identifies to the postcondition, and is thus a refinement of II. It forms the basis for Kleene Algebra with Tests [16, 1] (KAT), which embeds a Boolean algebra into a Kleene algebra to represent conditions.

definition lift-test :: 'α cond ⇒ 'α hrel (⊥→) where
[urel-defs]: [b]t = ([b]t ∧ II)

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 :: 'α cond ⇒ 'α hrel ⇒ 'α hrel (while ⊤ - do - od) where
[urel-defs]: while ⊤ b do P od = (ν X · (P ;; X) ⊔ b ∨ r, II)

abbreviation while-top :: 'α cond ⇒ 'α hrel ⇒ 'α hrel (while - do - od) where
while b do P od ≡ while ⊤ b do P od

definition while-bot :: 'α cond ⇒ 'α hrel ⇒ 'α hrel (while ⊥ - do - od) where
[urel-defs]: while ⊥ b do P od = (μ X · (P ;; X) ⊔ b ∨ r, II)

While loops with invariant decoration (cf. [1]) – partial correctness.

definition while-inv :: 'α cond ⇒ 'α cond ⇒ 'α hrel ⇒ 'α hrel (while - invr - do - od) where
[urel-defs]: while b invr p do S od = while ⊥ b do S od

While loops with invariant decoration – total correctness.

definition while-inv-bot :: 'α cond ⇒ 'α cond ⇒ 'α hrel ⇒ 'α hrel (while ⊥ - invr - do - od 71) where
[urel-defs]: while ⊥ b invr p do S od = while ⊥ b do S od

While loops with invariant and variant decorations – total correctness.

definition while-vrt :: 'α cond ⇒ 'α cond ⇒ (nat, 'α) uexpr ⇒ 'α hrel ⇒ 'α hrel (while - invr - vrt - do - od) where
[urel-defs]: while b invr p vrt v do S od = while ⊥ b do S od

We implement a poor man’s version of alphabet restriction that hides a variable within a relation.

definition rel-var-res :: 'α hrel ⇒ (′a → 'a) ⇒ 'α hrel (infix |α 80) where
[urel-defs]: P |α x = (∃ $x ⊕ 3 $x′ · P)

Alphabet extension and restriction add additional variables by the given lens in both their primed and unprimed versions.

definition rel-aext :: 'β hrel ⇒ ('β → 'α) ⇒ 'α hrel
where [upred-defs]: rel-aext P a = P ⊔p (a ×L a)
The nameset operator can be used to hide a portion of the after-state that lies outside the lens nameset definition.

Frame extension combines alphabet extension with the frame operator to both add additional variables and then frame those.

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.

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 [17].

Frame extension combines alphabet extension with the frame operator to both add additional variables and then frame those.

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.

15.3 Syntax Translations

syntax

- seq-iter :: pttrn ⇒ 'a list ⇒ 'a hrel ⇒ 'a hrel ((3;; - : - • -) [0, 0, 10] 10)
- assignment :: svids ⇒ uexprs ⇒ 'α hrel (([-] := [-]) [0000, 0000] 100)
- assignment-upd :: svids ⇒ uexprs ⇒ 'α hrel (infixr := 72)
- indexed assignment
- assignment-apd :: svid ⇒ logic ⇒ logic ⇒ logic (([-] :=/-) [73, 0, 0] 72)
- Substitution constructor
- mk-usubst :: svids ⇒ uexprs ⇒ 'α 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

-upt-if b P Q =⇒ P ⊆ b ⊃, Q
\[ x : l \cdot P \equiv (\text{CONST segr-iter}) \ (\lambda x. \ P) \]

- \text{mk-usubst} \ \sigma (\text{-svid-unit} \ x) \ v \equiv \sigma(\&x \mapsto v)
- \text{mk-usubst} \ \sigma (\text{-svid-list} \ x \ xs) \ (\text{-uzexprs} \ v \ vs) \equiv (\text{-mk-usubst} \ (\sigma(\&x \mapsto v)) \ xs \ vs)
- \text{assignment} \ x \ v \leftarrow (\text{CONST uassigns} \ (\text{-mk-usubst} \ (\text{CONST id}) \ x \ v))
- \text{assignment} \ x \ v \leftarrow (\text{-assignment} \ (\text{-spvar} \ x) \ v)

\[ x, y := u, v \leftarrow (\text{CONST uassigns} \ (\text{CONST subst-upd} \ (\text{CONST id}) \ x \ v)) \]

— Indexed assignment uses the overloaded collection update function \text{uupd}.

\[ x[k] := v \mapsto x := \&x(k \mapsto v) \]

\[ \text{frame} \ (\text{-salphaset} \ (\text{-salphamk} \ x)) \ P \leftarrow (\text{CONST frame} \ x \ P) \]

\[ \text{antiframe} \ (\text{-salphaset} \ (\text{-salphamk} \ x)) \ P \leftarrow (\text{CONST antiframe} \ x \ P) \]

\[ \text{nameset} \ x \ P \equiv (\text{CONST nameset} \ x \ P) \]

\[ \text{rel-aext} \ P \ a \equiv (\text{CONST rel-aext} \ P \ a) \]

\[ \text{rel-ares} \ P \ a \equiv (\text{CONST rel-ares} \ P \ a) \]

\[ \text{rel-frext} \ a \ P \equiv (\text{CONST rel-frext} \ a \ P) \]

The following code sets up pretty-printing for homogeneous relation 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 \((\alpha, \beta)\) \text{uexpr} type, determine that it is relational (product alphabet), and then checks if the types \(\alpha\) and \(\beta\) are the same. If they are, then the type is printed as a \text{hexpr}. Otherwise, we have no match. We then set up a regular translation for the \text{hrel} type that uses this.

\text{print-translation} \.timestamp
\begin{verbatim}
let fun tr' ctx [ 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
\end{verbatim}

\text{translations}
\begin{verbatim}
  (type) \alpha \text{hrel} \leftarrow (type) \text{bool, \alpha} \text{hexpr}
\end{verbatim}

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.

\text{definition} \text{afunctional} :: (\alpha, \beta) \text{urel} \Rightarrow \text{bool}
\text{where} [\text{urel-defs}]: \text{afunctional} \ R \leftarrow H \sqsubseteq R^- ;; \ R

\text{definition} \text{ainj} :: (\alpha, \beta) \text{urel} \Rightarrow \text{bool}
\text{where} [\text{urel-defs}]: \text{ainj} \ R \leftarrow H \sqsubseteq R^- ;; \ R

\text{definition} \text{Dom} :: \alpha \text{hrel} \Rightarrow \alpha \text{upred}
\text{where} [\text{upred-defs}]: \text{Dom} \ P = [\exists \v' \cdot P]_<

\text{definition} \text{Ran} :: \alpha \text{hrel} \Rightarrow \alpha \text{upred}
where [upred-defs]: \( \text{Ran } P = [\exists v \cdot P] > \)

— Configuration for UTP tactics (see utp-tactics).


15.5 Introduction laws

**Lemma urel-refine-ext:**
\[ \left[ \bigwedge s s'. P[s,s'/v,v'] \subseteq Q[s,s'/v,v'] \right] \implies P \subseteq Q \]
by (rel-auto)

**Lemma urel-eq-ext:**
\[ \left[ \bigwedge s s'. P[s,s'/v,v'] = Q[s,s'/v,v'] \right] \implies P = Q \]
by (rel-auto)

15.6 Unrestriction Laws

**Lemma unrest-iuvar [unrest]:** \( \text{out } \alpha \sharp x \)
by (metis fst-snd-lens-indep lift-pre-var out \( \alpha \)-def unrest-aext-indep)

**Lemma unrest-ouvar [unrest]:** \( \text{in } \alpha \sharp x' \)
by (metis in \( \alpha \)-def lift-post-var snd-fst-lens-indep unrest-aext-indep)

**Lemma unrest-semir-undash [unrest]:**
\[ \text{fixes } x :: ('a \Rightarrow 'a) \]
\[ \text{assumes } \sharp x \not\in P \]
\[ \text{shows } \sharp x \not\in P ; ; Q \]
\[ \text{using asms by (rel-auto)} \]

**Lemma unrest-semir-dash [unrest]:**
\[ \text{fixes } x :: ('a \Rightarrow 'a) \]
\[ \text{assumes } \sharp x' \not\in Q \]
\[ \text{shows } \sharp x \not\in P ; ; Q \]
\[ \text{using asms by (rel-auto)} \]

**Lemma unrest-cond [unrest]:**
\[ \left[ \sharp x \not\in P; \ x \not\in b ; \ x \not\in Q \right] \implies x \not\in P < b \triangleright Q \]
by (rel-auto)

**Lemma unrest-lift-rcond [unrest]:**
\[ x \not\in [b] < \implies x \not\in [b] \]
by (simp add: lift-rcond-def)

**Lemma unrest-ina-var [unrest]:**
\[ \left[ \text{mwb-lens } x; \text{ ina } \sharp (P :: ('a, ('a \times 'b)) uexpr) \right] \implies \sharp x \not\in P \]
by (rel-auto)

**Lemma unrest-outa-var [unrest]:**
\[ \left[ \text{mwb-lens } x; \text{ outa } \sharp (P :: ('a, ('a \times 'b)) uexpr) \right] \implies \sharp x' \not\in P \]
by (rel-auto)

**Lemma unrest-pre-outa [unrest]:** \( \text{outa } \not\in [b] \)
by (transfer, auto simp add: outa-def)

**Lemma unrest-post-ina [unrest]:** \( \text{ina } \not\in [b] \)

95
by (transfer, auto simp add: ina-def)

lemma unrest-pre-in-var [unrest]:
\[ x \neq p l \Rightarrow \$x \neq [p l] < \]
by (transfer, simp)

lemma unrest-post-out-var [unrest]:
\[ x \neq p l \Rightarrow \$x' \neq [p l] > \]
by (transfer, simp)

lemma unrest-convr-out \[unrest\]:
in \[\alpha\] ♯ \[p\] = \\Rightarrow \$
\begin{align*}
\text{by (transfer, auto simp add: lens-defs)}
\end{align*}

lemma unrest-convr-in \[unrest\]:
out \[\alpha\] ♯ \[p\] = \\Rightarrow \$
\begin{align*}
\text{by (transfer, auto simp add: lens-defs)}
\end{align*}

lemma unrest-in-rel-var-res [unrest]:
vwb-lens \(x\) = \\Rightarrow \$
\begin{align*}
\text{by (simp add: rel-var-res-def unrest)}
\end{align*}

lemma unrest-out-rel-var-res [unrest]:
vwb-lens \(x\) = \\Rightarrow \$
\begin{align*}
\text{by (simp add: rel-var-res-def unrest)}
\end{align*}

lemma unrest-out-alpha-usubst-rel-lift [unrest]:
\(\text{out} \[\alpha\] ♯ \[\sigma\] s\)
by (rel-auto)

lemma unrest-in-rel-aext [unrest]:
\(\text{weak-lens} \, a\) \Rightarrow \(\text{(} P \,;;; Q \,\text{)} \oplus_r \, a = (P \oplus_r a \,;;\, Q \oplus_r a)\)
\begin{align*}
\text{apply (rel-auto)}
\end{align*}

lemma unrest-out-rel-aext [unrest]:
\(\text{weak-lens} \, a\) \Rightarrow \(\text{(} P \,;;; Q \,\text{)} \oplus_r \, x\)
by (simp add: rel-aext-def unrest-aext-indep)

lemma rel-aext-seq [alpha]:
weak-lens \(a\) \Rightarrow \(\text{(} P \,;;; Q \,\text{)} \oplus_r \, a = (P \oplus_r a \,;;\, Q \oplus_r a)\)
\begin{align*}
\text{apply (rename-lac aa b y)}
\end{align*}

lemma rel-aext-cond [alpha]:
\(\text{(} P \lhd b \rhd_r Q \,\text{)} \oplus_r \, a = (P \oplus_r a \lhd b \oplus_p a \rhd_r Q \oplus_r a)\)
by (rel-auto)

15.7 Substitution laws

lemma subst-seq-left [usubst]:
\(\text{out} \neq \sigma \Rightarrow \sigma \uparrow (P ~;;; Q) = (\sigma \uparrow P) :: Q\)
by (rel-simp, (metis (no-types, lifting) Pair-inject surjective-pairing)+)

lemma subst-seq-right [usubst]:
in \neq \sigma \Rightarrow \sigma \uparrow (P ~;;; Q) = P ~;;; (\sigma \uparrow 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 \( x :: \) (bool → 'a)

shows

\[
\begin{align*}
& \land P \cdot Q \cdot (x \mapsto s \true) \vdash (P :: Q) = \sigma \vdash (P[[true/x]] :: Q) \\
& \land P \cdot Q \cdot (x \mapsto s \false) \vdash (P :: Q) = \sigma \vdash (P[[false/x]] :: Q) \\
& \land P \cdot Q \cdot (x' \mapsto s \true) \vdash (P :: Q) = \sigma \vdash (P :: Q[[true/x']]) \\
& \land P \cdot Q \cdot (x' \mapsto s \false) \vdash (P :: Q) = \sigma \vdash (P :: Q[[false/x']])
\end{align*}
\]

by (rel-auto)+

**Lemma** zero-one-seqr-laws [usubst]:

fixes \( x :: (- \Rightarrow 'a) \)

shows

\[
\begin{align*}
& \land P \cdot Q \cdot (x \mapsto s \false) \vdash (P :: Q) = \sigma \vdash (P[0/x] :: Q) \\
& \land P \cdot Q \cdot (x \mapsto s \true) \vdash (P :: Q) = \sigma \vdash (P[1/x] :: Q) \\
& \land P \cdot Q \cdot (x' \mapsto s \false) \vdash (P :: Q) = \sigma \vdash (P :: Q[0/x']) \\
& \land P \cdot Q \cdot (x' \mapsto s \true) \vdash (P :: Q) = \sigma \vdash (P :: Q[1/x'])
\end{align*}
\]

by (rel-auto)+

**Lemma** numeral-seqr-laws [usubst]:

fixes \( x :: (- \Rightarrow 'a) \)

shows

\[
\begin{align*}
& \land P \cdot Q \cdot (x \mapsto s \text{ \textit{numeral} \( n \))} \vdash (P :: Q) = \sigma \vdash (P[\text{\textit{numeral} \( n \)/x}] :: Q) \\
& \land P \cdot Q \cdot (x' \mapsto s \text{ \textit{numeral} \( n \))} \vdash (P :: Q) = \sigma \vdash (P :: Q[\text{\textit{numeral} \( n \)/x']})
\end{align*}
\]

by (rel-auto)+

**Lemma** usubst-condr [usubst]:

\( \sigma \vdash (P \circ b \triangleright Q) = (\sigma \vdash P \circ \sigma \vdash b \triangleright \sigma \vdash Q) \)

by (rel-auto)

**Lemma** subst-skip-r [usubst]:

\( \text{outa} \vdash \sigma \Rightarrow \sigma \vdash H = (\langle \sigma \rangle \text{ a}) \)

by (rel-simp, (metis (mono-tags, lifting) prod.sel(1) sndI surjective-pairing)+)

**Lemma** subst-pre-skip [usubst]: \( [\sigma] \cdot \vdash H = \langle \sigma \rangle \text{ a} \)

by (rel-auto)

**Lemma** subst-rel-lift-seq [usubst]:

\( [\sigma] \cdot \vdash (P :: Q) = ([\sigma] \cdot \vdash P) :: Q \)

by (rel-auto)

**Lemma** subst-rel-lift-comp [usubst]:

\( [\sigma] \cdot \circ \circ \cdot = [\sigma \circ \circ \cdot] \cdot \)

by (rel-auto)

**Lemma** usubst-upd-in-comp [usubst]:

\( \sigma(\text{\textit{kina}x \mapsto s \( v \))} = \sigma(\text{\textit{x} \mapsto s \( v \))} \)

by (simp add: pr-var-def fst-lens-def ina-def in-var-def)

**Lemma** usubst-upd-out-comp [usubst]:

\( \sigma(\text{\textit{kouta}x \mapsto s \( v \))} = \sigma(\text{\textit{x} \mapsto s \( v \))} \)
by (simp add: pr-var-def outa-def out-var-def snd-lens-def)

lemma subst-lift-upd [alpha]:
  fixes x :: ('a ⇒ 'α)
  shows \[ \sigma(x \mapsto v) \] \(\sigma\)' = \[ \sigma \] \(x \mapsto v\)
  by (simp add: alpha usubst, simp add: pr-var-def fst-lens-def in α-def in-var-def)

lemma subst-drop-upd [alpha]:
  fixes x :: ('a ⇒ 'α)
  shows \[ \sigma(x \mapsto v) \] \(\sigma\)' = \[ \sigma \] \(x \mapsto s\) \(\sigma\)' \(v\)
  by pred-simp

lemma subst-lift-pre [usubst]: \[ \sigma \] \(\sigma\)' \(\sigma\)' \(\sigma\)
  by (metis apply-subst-ext fst-vwb-lens in α-def)

lemma unrest-usubst-lift-in [unrest]:
  fixes x :: ('a ⇒ 'α)
  shows \(x \mapsto v\) \(\sigma\)' \(\sigma\)
  by pred-simp

lemma unrest-usubst-lift-out [unrest]:
  fixes x :: ('a ⇒ 'α)
  shows \(x \mapsto v\) \(\sigma\)' \(\sigma\)
  by pred-simp

lemma subst-lift-cond [usubst]: \[ \sigma \] \(\sigma\)' \(\sigma\)' \(\sigma\)
  by (rel-auto)

lemma msubst-seq [usubst]: ((P x ; Q x) [x \mapsto v]) = ((P x) [x \mapsto v] ; (Q x) [x \mapsto v])
  by (rel-auto)

15.8 Alphabet laws

lemma aext-cond [alpha]:
  \(P \odot b \odot Q \odot p \odot a \odot (P \odot p \odot a \odot (b \odot p \odot a \odot (Q \odot p \odot a)) \odot (Q \odot p \odot a))
  by (rel-auto)

lemma aext-seq [alpha]:
  \(w b-lens a \odot ((P ; Q) \odot p \odot (a \times L a)) = ((P \odot p \odot (a \times L a)) ; (Q \odot p \odot (a \times L a)))\)
  by (rel-simp,metis wb-lens-weak weak-lens.put-get)

lemma rcond-lift-true [simp]:
  \[ true \] \(true\)
  by rel-auto

lemma rcond-lift-false [simp]:
  \[ false \] \(false\)
  by rel-auto

lemma rel-ares-aext [alpha]:
  \(v b-lens a \odot ((P \odot r a) \odot r a = P)\)
  by (rel-auto)

lemma rel-aext-ares [alpha]:
  \(\{ a, a \} \odot P \odot (a \odot r) \odot a = P\)
  by (rel-auto)
lemma rel-aext-uses [unrest]:
\[ vwb-lens a \implies (\forall x. a) \uplus (P \ominus_r a) \]
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 :: (\forall x. a) \implies (\forall \alpha. \alpha \ hrel) \implies (\forall \alpha. \alpha \ hrel)
where RID x P = ((\exists x \cdot \exists x'. \cdot P \land x' = \ u \ x) )
declare RID-def [urel-defs]

lemma RID1: vwb-lens x \implies (\forall v. x := v; (P \ominus x := v)) \implies RID(x)(P) = P
apply (rel-auto)
apply (metis vwb-lens.put-eq)
apply (metis vwb-lens-wb wb-lens.put-put wb-lens-weak weak-lens.put-get)
done

lemma RID2: vwb-lens x \implies x := v; (P \ominus v) = RID(x)(P) ; x := v
apply (rel-auto)
apply (metis mwblens.put-put vwb-lens-mwb wb-lens.get-put wb-lens-def weak-lens.put-get)
apply blast
done

lemma RID-assign-commute:
\[ vwb-lens x \implies P = RID(x)(P) \iff (\forall v. x := v; (P \ominus x := v)) \]
by (simp add: Healthy-def RID-skip-r)

lemma RID-idem:
mwb-lens x \implies RID(x)(RID(x)(P)) = RID(x)(P)
by (rel-auto)

lemma RID-mono:
P \cup Q \implies RID(x)(P) \cup Q
by (rel-auto)

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

lemma RID-skip-r:
\[ vwb-lens x \implies RID(x)(II) = II \]
apply (rel-auto) using vwb-lens.put-eq by fastforce

lemma skip-r-RID [closure]: vwb-lens x \implies II 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 disj-RID [closure]: \[ P is RID(x); Q is RID(x) \] \implies (P \lor Q) is RID(x)
by (simp add: Healthy-def RID-disj)
lemma RID-conj:
  \[ vwb-lens \, x \implies RID(x)(RID(x)(P) \land RID(x)(Q)) = (RID(x)(P) \land RID(x)(Q)) \]
by (rel-auto)

lemma conj-RID [closure]:
  \[ vwb-lens \, x; \; P \text{ is } RID(x); \; Q \text{ is } RID(x) \implies (P \land Q) \text{ is } RID(x) \]
by (metis Healthy-if Healthy-intro RID-conj)

lemma RID-assigns-r-diff:
  \[ vwb-lens \, x; \; x \notin \sigma \implies RID(x)((\sigma)_a) = (\sigma)_a \]
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 assigns-r-RID [closure]:
  \[ vwb-lens \, x; \; x \notin \sigma \implies (\sigma)_a \text{ is } RID(x) \]
by (simp add: Healthy-def RID-assigns-r-diff)

lemma RID-assign-r-same:
  \[ vwb-lens \, x \implies RID(x)(x := v) = \Pi \]
apply (rel-auto)
using vwb-lens.put-eq apply fastforce
done

lemma RID-seq-left:
  assumes vwb-lens \, x
  shows \[ RID(x)(RID(x)(P) ;; Q) = (RID(x)(P) ;; RID(x)(Q)) \]
proof
  have \[ RID(x)(RID(x)(P) ;; Q) = ((\exists \, \$x \cdot \exists \, \$x' \cdot (\exists \, \$x \cdot \exists \, \$x' \cdot P) \land \$x' =_u \$x) ;; Q) \land \$x' =_u \$x \]
  by (simp add: RID-def usubst)
  also from \text{ assms} have \[ ... = (((((\exists \, \$x \cdot \exists \, \$x' \cdot P) \land (\exists \, \$x \cdot \exists \, \$x' =_u \$x)) ;; (\exists \, \$x' \cdot Q)) \land \$x' =_u \$x) \]
  by (rel-auto)
  also from \text{ assms} have \[ ... = (((((\exists \, \$x \cdot \exists \, \$x' \cdot P) ;; (\exists \, \$x \cdot \exists \, \$x' \cdot Q)) \land \$x' =_u \$x) \]
  by (rel-auto)
  also from \text{ assms} have \[ ... = (((((\exists \, \$x \cdot \exists \, \$x' \cdot P) \land \$x' =_u \$x) ;; (\exists \, \$x \cdot \exists \, \$x' \cdot Q) \land \$x' =_u \$x)) \land \$x' =_u \$x) \]
  by (rel-simp, fastforce)
  also have \[ ... = (((((\exists \, \$x \cdot \exists \, \$x' \cdot P) \land \$x' =_u \$x) ;; (\exists \, \$x \cdot \exists \, \$x' \cdot Q) \land \$x' =_u \$x)) \]
  by (rel-auto)
  also have \[ ... = (RID(x)(P) ;; RID(x)(Q)) \]
  by (rel-auto)
  finally show \[ ?thesis \]
qed
\[
\begin{align*}
= & \ s x \\
& \text{by (simp add: RID-def asubst)} \\
& \text{also from assms have } ... = (((\exists \ x \cdot P) \land (\exists x' \cdot Q)) \land (\exists x' = u \ s x) \land \ s x' = u \ s x) \\
& \text{by (rel-auto)} \\
& \text{also from assms have } ... = (((\exists \ x \cdot \exists x' \cdot P) \land \ s x' = u \ s x) \land (\exists x' \cdot Q)) \land \ s x = u \ s x \\
& \text{by (metis wvb-lens.put-get)} \\
& \text{apply (metis wvb-lens, put-get wvblens-def wblens-weak)} \\
& \text{done} \\
& \text{also from assms have } ... = (((\exists \ x \cdot \exists x' \cdot P) \land \ s x' = u \ s x) \land (\exists x' \cdot Q)) \land \ s x = u \ s x \\
& \text{by (rel-auto)} \\
& \text{also have } ... = (\text{RID}(x)(P)) \iff (\text{RID}(x)(Q)) \\
& \text{by (rel-auto)} \\
& \text{finally show } \ ?\text{thesis .} \\
\end{align*}
\]

\text{lem}a \ \text{seqR-RID-closed} \ [\text{closure}]: \ [\text{wvb-lens} \ x; \ P \text{ is RID}(x); \ Q \text{ is RID}(x)] \implies P \iff Q \text{ is RID}(x) \\
\text{by (metis Healthy-def RID-seq-right)} \\

\text{def}inition \ \text{unrest-relation} :: \ ('a \Rightarrow \ 'a) \Rightarrow 'a \ hrel \Rightarrow \ 'a (\text{infix} \ \#\# \ 20) \\
\text{where } (x \#\# P) \iff (P \text{ is RID}(x)) \\

\text{declare} \ \text{unrest-relation-def} \ [\text{urel-defs}] \\

\text{lem}a \ \text{runrest-assign-commute}: \\
[\text{wvb-lens} \ x; \ x \#\# P] \implies x := v \iff P \iff x := v \\
\text{by (metis RID2 Healthy-def unrest-relation-def)} \\

\text{lem}a \ \text{runrest-ident-var}: \\
\text{assumes } x \#\# P \\
\text{shows } (\ s x \land P) = (P \land \ s x') \\
\text{proof} - \\
\text{have } P = (\ s x' = u \ s x \land P) \\
\text{by (metis RID-def assms Healthy-def unrest-relation-def utp-pred-laws.inf.cobounded2 utp-pred-laws.inf-absorb2)} \\
\text{moreover have } (\ s x' = u \ s x \land (\ s x \land P)) = (\ s x' = u \ s x \land (P \land \ s x')) \\
\text{by (rel-auto)} \\
\text{ultimately show } \ ?\text{thesis} \\
\text{by (metis utp-pred-laws.inf.assoc utp-pred-laws.inf-left-commute)} \\
\text{qed} \\

\text{lem}a \ \text{skip-r-runrest} \ [\text{unrest}]: \\
\text{wvb-lens} \ x \Rightarrow x \#\# H \\
\text{by (simp add: unrest-relation-def closure)} \\

\text{lem}a \ \text{assigns-r-runrest}: \\
[\text{wvb-lens} \ x; \ x \#\# \sigma] \Rightarrow x \#\# (\sigma)_a \\
\text{by (simp add: unrest-relation-def closure)} \\

\text{lem}a \ \text{seq-r-runrest} \ [\text{unrest}]:
assumes \( \text{vwb-lens } x \ x \ = \ P \ x \ Q \)
shows \( x \ = (P ; Q) \)
using assms by (simp add: unrest-relation-def closure )

lemma false-runrest [unrest]: \( x \ = \ \text{false} \)
by (rel-auto)

lemma and-runrest [unrest]: \( [ \text{vwb-lens } x ; x \ = \ P \ ; x \ = \ Q ] \implies x \ = (P \land Q) \)
by (metis RID-conj Healthy-def unrest-relation-def)

lemma or-runrest [unrest]: \([ x \ = \ P ; x \ = \ Q ] \implies x \ = (P \lor Q) \)
by (simp add: RID-disj Healthy-def unrest-relation-def)

end

16 Fixed-points and Recursion

theory utp-recursion
imports
  utp-pred-laws
  utp-rel
begin

16.1 Fixed-point Laws

lemma mu-id: \( (\mu X \cdot X) = \text{true} \)
by (simp add: antisym gfp-upperbound)

lemma mu-const: \( (\mu X \cdot P) = P \)
by (simp add: gfp-const)

lemma nu-id: \( (\nu X \cdot X) = \text{false} \)
by (meson lfp-lowerbound utp-pred-laws.bot.extremum-unique)

lemma nu-const: \( (\nu X \cdot P) = P \)
by (simp add: lfp-const)

lemma mu-refine-intro:
  assumes \( C \Rightarrow S \subseteq F(C \Rightarrow S) (C \land \mu F) = (C \land \nu F) \)
  shows \( C \Rightarrow S \subseteq \mu F \)
proof
  from assms have \( C \Rightarrow S \subseteq \nu F \)
  by (simp add: lfp-lowerbound)
  with assms show \( ? \text{thesis} \)
  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 [14].

type-synonym \'a chain = nat \Rightarrow \'a upred

definition chain :: \'a chain \Rightarrow bool where
chain \( Y = ((Y \ 0 = \ false) \land (\forall \ i. \ Y (Suc \ i) \subseteq Y \ i)) \)

**lemma** chain0 [simp]: \( chain \ Y \Rightarrow Y \ 0 = false \)
by (simp add: chain-def)

**lemma** chainI:
assumes \( Y \ 0 = false \land i. \ Y (Suc \ i) \subseteq Y \ i \)
shows \( chain \ Y \)
using assms by (auto simp add: chain-def)

**lemma** chainE:
assumes \( chain \ Y \land i. \ [Y \ 0 = false; Y (Suc \ i) \subseteq Y \ i] \Rightarrow P \)
shows \( P \)
using assms by (simp add: chain-def)

**lemma** L274:
assumes \( \forall \ n. \ ((E \ n \land Y) = (E \ n \land Y)) \)
shows \( \prod (range \ E) \land X = (\prod (range \ E) \land Y) \)
using assms by (pred-auto)

Constructive chains

**definition** constr ::
\( (\text{‘a upred} \to \text{‘a upred}) \to \text{‘a chain} \Rightarrow \text{bool} \)
where
constr \( F \ E \leftarrow \rightleftharpoons \text{chain} \ E \land (\forall \ X \ n. ((F(X) \land E(n + 1)) = (F(X \land E(n)) \land E (n + 1)))) \)

**lemma** constrI:
assumes \( chain \ E \land i. \ [F \ (X \land E(n) \land E (n + 1))) \)
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 \( \prod (range \ E \land \mu \ F) = (\prod (range \ E \land \nu \ F) \land \mu \ F) \)
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 \( ... = (E \ (n + 1) \land F (E \ n \land \mu \ F)) \)
  by (metis conj-comm constr-def)
  also from \( hyp \) have \( ... = (E \ (n + 1) \land F (E \ n \land \nu \ F)) \)
  by simp
  also from \( hyp \) have \( ... = (E \ (n + 1) \land \nu \ F) \)
  by (metis (no-types, lifting) conj-comm constr-def lfp-unfold)

103
ultimately show \( \text{thesis} \)
by simp
qed
qed
qed
thus \( \text{thesis} \)
by (auto intro: L274)
qed

theorem constr-fp-uniq:
assumes constr \( F \ E \) mono \( F \ \prod \ (\text{range } E) = C \)
shows \( (C \land \mu F) = (C \land \nu F) \)
using \( \text{assms(1) assms(2) assms(3) chain-pred-terminates by blast} \)

\[ \text{16.3 Noetherian Induction Instantiation} \]

Contribution from Yakoub Nemouchi. The following generalization was used by Tobias Nipkow and Peter Lammich in Refine_Monadic

\[ \text{lemma wf-fixp-uninduct-pure-ueq-gen:} \]
assumes fixp-unfold: \( \text{fp } B = B (\text{fp } B) \)
and WF: \( \text{wf } R \)
and induct-step:
\[ \forall f \ st. \left[ \left( \text{Pre} \land \left[ \epsilon \right] < =_u \left< st' \right> \Rightarrow \text{Post} \right) \subseteq f \right] \] \[ \Rightarrow \text{fp } B = f \Rightarrow (\text{Pre} \land \left[ \epsilon \right] < =_u \left< st \right> ) \Rightarrow \text{Post} \subseteq (B f) \]
shows \( ((\text{Pre} \Rightarrow \text{Post}) \subseteq \text{fp } B) \)
proof -
\{ fix \( st \) 
have \( (\text{Pre} \land \left[ \epsilon \right] < =_u \left< st \right> ) \Rightarrow \text{Post} \) \( \subseteq (\text{fp } B) \)
using WF proof (induction rule: wf-induct-rule)
  case (less \( x \))
  hence \( (\text{Pre} \land \left[ \epsilon \right] < =_u \left< x \right> ) \Rightarrow \text{Post} \) \( \subseteq B (\text{fp } B) \)
  by (rule induct-step, rel-blast, simp)
  then show \( ?\text{case} \)
    using fixp-unfold by auto
  qed
\}
thus \( ?\text{thesis} \)
by pred-simp
qed

The next lemma shows that using substitution also work. However it is not that generic nor practical for proof automation ...

\[ \text{lemma refine-usubst-to-ueq:} \]
\[ \text{vwb-lens } E \Rightarrow (\text{Pre} \Rightarrow \text{Post})[\text{st'}/\$E] \subseteq f[\text{st'}/\$E] = (((\text{Pre} \land \$E =_u \text{st'}) \Rightarrow \text{Post}) \subseteq f) \]
by (rel-auto, metis vwb-lens-vwb wb-lens-get-pat)

By instantiation of \( [[?\text{fp } ?B = ?B (?\text{fp } ?B); \text{wf } ?R; \prod f \ st. \left[ \left( \text{Pre} \land \left[ \epsilon \right] < =_u \left< st' \right> \Rightarrow \text{Post} \right) \subseteq f \right]; ?\text{fp } ?B = f]\] \Rightarrow (\text{Pre} \land \left[ \epsilon \right] < =_u \left< st \right> ) \Rightarrow ?\text{Post} \subseteq ?B f) \)
\[ \Rightarrow (\text{Pre} \Rightarrow ?\text{Post}) \subseteq ?\text{fp } ?B \text{ with } \mu \text{ and lifting of the well-founded relation we have ...} \]

\[ \text{lemma mu-rec-total-pure-rule:} \]
assumes WF: \( \text{wf } R \)
and mono \( B \)
and induct-step:
\[ \prod f \ st. \left[ (\text{Pre} \land (\left[ \epsilon, st \right] _u \subseteq R \Rightarrow \text{Post}) \subseteq f) \right] \]

104
\[ \implies \mu B = f \implies (\text{Pre} \land [e] <_u \prec s \text{t} \implies \text{Post}) \subseteq (B f) \]

**proof** (rule \texttt{wf-fzp-ainduct-pure-ueq-gen}[where \texttt{fp=mu} and \texttt{Pre=Pre} and \texttt{B=B} and \texttt{R=R} and \texttt{e=e}])

**show** \( \mu B = B (\mu B) \)

by (simp add: \texttt{M_def-lfp-unfold})

**show** \( \text{wf R} \)

by (fact \texttt{WF})

**show** \( \bigwedge f \text{ st. } (\bigwedge \text{st}', (\text{st}', \text{st}) \in R \implies (\text{Pre} \land [e] <_u \prec \text{st}' \implies \text{Post}) \subseteq f) \implies \mu B = f \implies (\text{Pre} \land [e] <_u \prec \text{st} \implies \text{Post}) \subseteq (B f) \)

by (rule \texttt{induct-step, rel-simp, simp})

qed

**lemma** \texttt{nu-rec-total-pure-rule}:

**assumes** \( \text{WF: wf R} \)

and \( M: \text{ mono B} \)

and \( \text{induct-step:} \)

\[ \bigwedge f \text{ st. } \bigwedge \text{st}', (\text{st}', \text{st}) \in R \implies (\text{Pre} \land [e] <_u \prec \text{st}' \implies \text{Post}) \subseteq f \implies \nu B = f \implies (\text{Pre} \land [e] <_u \prec \text{st} \implies \text{Post}) \subseteq (B f) \]

**shows** \( \text{(Pre \implies Post) \subseteq \nu B} \)

**proof** (rule \texttt{nf-pz-induct-pure-ueq-gen}[where \texttt{fp=nu} and \texttt{Pre=Pre} and \texttt{B=B} and \texttt{R=R} and \texttt{e=e}])

**show** \( \nu B = B (\nu B) \)

by (simp add: \texttt{M_def-lfp-unfold})

**show** \( \text{wf R} \)

by (fact \texttt{WF})

**show** \( \bigwedge f \text{ st. } (\bigwedge \text{st}', (\text{st}', \text{st}) \in R \implies (\text{Pre} \land [e] <_u \prec \text{st}' \implies \text{Post}) \subseteq f \implies \nu B = f \implies (\text{Pre} \land [e] <_u \prec \text{st} \implies \text{Post}) \subseteq (B f) \)

by (rule \texttt{induct-step, rel-simp, simp})

qed

Since \( B (\text{Pre} \land ([E] <_u \prec \text{st} _u \in \text{R} \implies \text{Post}) \subseteq B (\mu B) \) and \( \text{mono B} \), thus, \[ \text{wf ?R;} \text{ Monotonic ?B;} \bigwedge f \text{ st. } ([\text{Pre} \land ([?e] <_u \prec \text{st} _u \in \text{R} \implies ?\text{Post}) \subseteq f; \mu ?B = f] \implies (?\text{Pre} \land [?e] <_u \prec \text{st} \implies ?\text{Post}) \subseteq ?B f] \implies (?\text{Pre} \implies ?\text{Post}) \subseteq \mu ?B \] can be expressed as follows

**lemma** \texttt{mu-rec-total-utp-rule}:

**assumes** \( \text{WF: wf R} \)

and \( M: \text{ mono B} \)

and \( \text{induct-step:} \)

\[ \bigwedge \text{st. } (\text{Pre} \land [e] <_u \prec \text{st} \implies \text{Post}) \subseteq (B ((\text{Pre} \land ([e] <_u \prec \text{st} _u \in \text{R} \implies \text{Post}))) \)

**shows** \( \text{(Pre \implies Post) \subseteq \mu B} \)

**proof** (rule \texttt{nu-rec-total-utp-rule}[where \texttt{R=R} and \texttt{e=e}], \texttt{simp-all add: asms})

**show** \[ \bigwedge f \text{ st. } (\text{Pre} \land ([e] <_u \prec \text{st} _u \in \text{R} \implies \text{Post}) \subseteq f \implies \mu B = f \implies (\text{Pre} \land [e] <_u \prec \text{st} \implies \text{Post}) \subseteq (B f) \]

by (simp add: \texttt{M_induct-step monoD order-subst2})

qed

**lemma** \texttt{nu-rec-total-utp-rule}:

**assumes** \( \text{WF: wf R} \)

and \( M: \text{ mono B} \)

and \( \text{induct-step:} \)

\[ \bigwedge \text{st. } (\text{Pre} \land [e] <_u \prec \text{st} \implies \text{Post}) \subseteq (B ((\text{Pre} \land ([e] <_u \prec \text{st} _u \in \text{R} \implies \text{Post}))) \)

**shows** \( \text{(Pre \implies Post) \subseteq \nu B} \)

**proof** (rule \texttt{nu-rec-total-utp-rule}[where \texttt{R=R} and \texttt{e=e}], \texttt{simp-all add: asms})

**show** \[ \bigwedge f \text{ st. } (\text{Pre} \land ([e] <_u \prec \text{st} _u \in \text{R} \implies \text{Post}) \subseteq f \implies \nu B = f \implies (\text{Pre} \land [e] <_u \prec \text{st} \implies \text{Post}) \subseteq (B f) \]

105
by (simp add: M induct-step monoD order-subst2)

qed

end

17 UTP Deduction Tactic

theory utp-deduct
imports utp-pred
begin

named-theorems uintro
named-theorems uelim
named-theorems udest

lemma uttrueI [uintro]: \([\text{true}]_b\)
  by (pred-auto)

lemma uopI [uintro]: \(f (\text{[x]}_b) \Rightarrow \text{[uop f x]}_b\)
  by (pred-auto)

lemma bopI [uintro]: \(f (\text{[x]}_b) (\text{[y]}_b) \Rightarrow \text{[bop f x y]}_b\)
  by (pred-auto)

lemma tropI [uintro]: \(f (\text{[x]}_b) (\text{[y]}_b) (\text{[z]}_b) \Rightarrow \text{[trop f x y z]}_b\)
  by (pred-auto)

lemma uconjI [uintro]: \(\text{[p]}_b = \Rightarrow \text{[p \land q]}_b\)
  by (pred-auto)

lemma uconjE [uelim]: \(\text{[p \land q]}_b = \Rightarrow \text{P} \Rightarrow P\)
  by (pred-auto)

lemma uimpI [uintro]: \(\text{[p]}_b \Rightarrow \text{[q]}_b = \Rightarrow \text{[p \Rightarrow q]}_b\)
  by (pred-auto)

lemma uimpE [elim]: \(\text{[p \Rightarrow q]}_b = \Rightarrow \text{[p]}_b = \Rightarrow \text{P} \Rightarrow P\)
  by (pred-auto)

lemma ushAllI [uintro]: \(\text{[\forall x. p(x)}]_b \Rightarrow \text{[\forall x \cdot p(x)}]_b\)
  by pred-auto

lemma ushExI [uintro]: \(\text{[p(x)}]_b \Rightarrow \text{[\exists x \cdot p(x)}]_b\)
  by pred-auto

lemma udeduct-tautI [uintro]: \(\text{[\land b. [p]}]_b \Rightarrow \text{‘ p’}\)
  using taut.rep-eq by blast

lemma udeduct-refineI [uintro]: \(\text{[\land b. [q]}]_b \Rightarrow [p]_b \Rightarrow p \subseteq q\)
  by (pred-auto)

lemma udeduct-eqI [uintro]: \(\text{[\land b. [p]}]_b \Rightarrow [q]_b; \text{[\land b. [q]}]_b \Rightarrow [p]_b \Rightarrow p = q\)
  by (pred-auto)

Some of the following lemmas help backward reasoning with bindings
lemma conj-implies: \([ P \land Q \] e \ b \] \implies \([ P \] e \ b \land \[ Q \] e \ b \]
by pred-auto

lemma conj-implies2: \([ P \] e \ b \land \[ Q \] e \ b \] \implies \([ P \land Q \] e \ b \]
by pred-auto

lemma disj-eq: \([ P \] e \ b \lor \[ Q \] e \ b \] \implies \([ P \lor Q \] e \ b \]
by pred-auto

lemma disj-eq2: \([ P \lor Q \] e \ b \] \implies \([ P \] e \ b \lor \[ Q \] e \ b \]
by pred-auto

lemma conj-eq-subst: \([ P \land Q \] e \ b \land \[ P \] e \ b = \[ R \] e \ b \) = \([ P \land Q \] e \ b \land \[ P \] e \ b = \[ R \] e \ b \)
by pred-auto

lemma conj-imp-subst: \([ P \land Q \] e \ b \land \[ P \] e \ b \implies \([ P \] e \ b = \[ R \] e \ b \]) = \([ P \land Q \] e \ b \land \[ P \] e \ b \implies \([ P \] e \ b = \[ R \] e \ b \])
by pred-auto

lemma disj-imp-subst: \([ Q \land (P \lor S) \] e \ b \land \[ Q \] e \ b \implies \([ P \] e \ b = \[ R \] e \ b \]) = \([ Q \land (R \lor S) \] e \ b \land \[ Q \] e \ b \implies \([ P \] e \ b = \[ R \] e \ b \])
by pred-auto

Simplifications on value equality

lemma uexpr-eq: \([ [e_0]_e = [e_1]_e \ b \] \) = \([ e_0 =_u e_1 \] e \ b \)
by pred-auto

lemma uexpr-trans: \([ [P \land e_0 =_u e_1]_e \ b \land \[ P \] e \ b \implies \[ e_1 =_u e_2 \] e \ b \]) = \([ [P \land e_0 =_u e_1]_e \ b \land \[ P \] e \ b \implies \[ e_1 =_u e_2 \] e \ b \])
by (pred-auto)

lemma uexpr-trans2: \([ [P \land Q \land e_0 =_u e_1]_e \ b \land \[ P \] e \ b \implies \[ e_1 =_u e_2 \] e \ b \]) = \([ [P \land Q \land e_0 =_u e_1]_e \ b \land \[ P \] e \ b \implies \[ e_1 =_u e_2 \] e \ b \])
by (pred-auto)

lemma uequality: \([ [R]_e = [Q]_e \ b \] \implies \([ P \land R]_e \ b = \[ P \land Q]_e \ b \)
by pred-auto

lemma ueqc1: \([ \[ P \] e \ b \implies \[ R \] e \ b = \[ Q \] e \ b \] \] \implies \([ P \land R]_e \ b \implies \[ P \land Q]_e \ b \)
by pred-auto

lemma ueqc2: \([ \[ P \] e \ b \implies \[ Q \] e \ b = \[ R \] e \ b \land \[ Q \land P]_e \ b = \[ R \land P]_e \ b \] \implies \([ P \] e \ b \implies \[ Q]_e \ b = \[ R]_e \ b)\]
by pred-auto

lemma ueqc3: \([ \[ P \] e \ b \implies \[ Q]_e \ b = \[ R]_e \ b \] \] \implies \([ R \land P]_e \ b = \[ Q \land P]_e \ b \)
by pred-auto

The following allows simplifying the equality if \( P \rightarrow Q = R \)

lemma ueqc3-imp: \([ (\land b. \[ P \] e \ b \implies \[ Q]_e \ b = \[ R]_e \ b) \implies (R \lor P) = (Q \land P) \]
by pred-auto

lemma ueqc3-imp2: \([ (\land b. \[ P \] e \ b \implies \[ Q]_e \ b = \[ R]_e \ b) \implies ((P \land Q) = (P \land R)) \]
by pred-auto
lemma *ueq3-imp2*: 
\[
(\forall b. ([P0 \land P1]_e b \implies [Q]_e b \implies [R]_e b = [S]_e b)) \implies ((P0 \land P1 \land (Q \implies R)) = (P0 \land P1 \land (Q \implies S)))
\]
by *pred-auto*

The following can introduce the binding notation into predicates

lemma *conj-bind-dist*: 
\[
[P \land Q]_e b = ([P]_e b \land [Q]_e b)
\]
by *pred-auto*

lemma *disj-bind-dist*: 
\[
[P \lor Q]_e b = ([P]_e b \lor [Q]_e b)
\]
by *pred-auto*

lemma *imp-bind-dist*: 
\[
[P \implies Q]_e b = ([P]_e b \rightarrow [Q]_e b)
\]
by *pred-auto*
**Lemma** postcond-left-unit: \( \text{in} \  \alpha \  \# \  p \  \Rightarrow \  (\text{true} \  ; \  p) = p \)
by (metis postcond-equiv)

**Theorem** precond-left-zero:
assumes \( \text{out} \  \alpha \  \# \  p \  \neq \  \text{false} \)
shows \( (\text{true} \  ; \  p) = \text{true} \)
using assms by (rel-auto)

**Theorem** feasible-iff-true-right-zero:
\( P \  ; \  \text{true} = \text{true} \longleftrightarrow \exists \  \text{out} \  \alpha \cdot P' \)
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]:
\( II ; ; P = P \)
by (rel-auto)

**Lemma** seqr-right-unit [simp]:
\( P ; ; II = P \)
by (rel-auto)

**Lemma** seqr-left-zero [simp]:
\( \text{false} ; ; P = \text{false} \)
by pred-auto

**Lemma** seqr-right-zero [simp]:
\( P ; ; \text{false} = \text{false} \)
by pred-auto

**Lemma** impl-seqr-mono: \[ (P \Rightarrow Q ; ; R \Rightarrow S) \] = \( (P ; ; R) \Rightarrow (Q ; ; S) \)
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:
\[ [\text{mono} P ; ; \text{mono} Q ] \Rightarrow \text{mono} (\lambda X. P X ; ; Q X) \]
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:
\( (\exists x \cdot P) ; ; Q = (\exists x \cdot (P ; ; Q)) \)
by (rel-auto)

**Lemma** seqr-exists-right:
\( (P ; ; (\exists x \cdot Q)) = (\exists x \cdot (P ; ; Q)) \)
by (rel-auto)
lemma seqr-or-distl:
\[(\langle P \lor Q \rangle ; R) = (\langle P ; R \rangle \lor (Q ; R))\]
by (rel-auto)

lemma seqr-or-distr:
\[(P ; (Q \lor R)) = ((P ; Q) \lor (P ; R))\]
by (rel-auto)

lemma seqr-and-distr-ufunc:
\[ufunctional P \Rightarrow (\langle P \land Q \rangle ; R) = (\langle P ; R \rangle \lor (Q ; R))\]
by (rel-auto)

lemma seqr-and-distl-uinj:
\[uinj R \Rightarrow ((P \land Q) ; R) = (\langle P ; R \rangle \land (Q ; R))\]
by (rel-auto)

lemma seqr-unfold:
\[(\langle P \rangle ; Q) = (\exists v \cdot P[\langle v \rangle] \land Q[\langle v \rangle])\]
by (rel-auto)

lemma seqr-middle:
assumes vwb-lens x
shows \[(P ; Q) = (\exists v \cdot P[\langle v \rangle] ; Q[\langle v \rangle])\]
using assms
by (rel-auto', metis vwb-lens-wb wb-lens source-stability)

lemma seqr-left-one-point:
assumes vwb-lens x
shows \[(\langle P \rangle ; \langle Q \rangle) = (P[\langle v \rangle] ; Q[\langle v \rangle])\]
using assms
by (rel-auto, metis vwb-lens-wb wb-lens get-put)

lemma seqr-right-one-point:
assumes vwb-lens x
shows \[(\langle P \rangle ; (\langle Q \rangle \land \langle R \rangle)) = (P[\langle v \rangle] ; Q[\langle v \rangle])\]
using assms
by (rel-auto, metis vwb-lens-wb wb-lens.get-put)

lemma seqr-left-one-point-true:
assumes vwb-lens x
shows \[(\langle P \rangle \land \langle x \rangle = \langle u \rangle \land \langle v \rangle) ; Q) = (P[\langle v \rangle ; Q[\langle v \rangle])\]
using assms
by (rel-auto, metis vwb-lens-wb wb-lens.get-put)

lemma seqr-left-one-point-true:
assumes vwb-lens x
shows \[(\langle P \rangle \land \langle x \rangle = \langle u \rangle \land \langle v \rangle) ; Q) = (P[\langle v \rangle ; Q[\langle v \rangle])\]
using assms
by (rel-auto, metis vwb-lens-wb wb-lens.get-put)

lemma seqr-left-one-point-false:
assumes vwb-lens x
shows \[(\langle P \rangle \land \langle x \rangle = \langle u \rangle \land \langle v \rangle) ; Q) = (P[\langle v \rangle ; Q[\langle v \rangle])\]
using assms
by (rel-auto, metis vwb-lens-wb wb-lens.get-put)

lemma seqr-right-one-point-false:
assumes vwb-lens x
shows \[(\langle P \rangle ; (\langle x \rangle \land \langle Q \rangle)) = (P[\langle v \rangle ; Q[\langle v \rangle])\]
using assms
by (rel-auto, metis vwb-lens-wb wb-lens.get-put)

lemma seqr-right-one-point-false:
assumes vwb-lens x
shows \[(\langle P \rangle ; (\langle x \rangle \land \langle Q \rangle)) = (P[\langle v \rangle ; Q[\langle v \rangle])\]
using assms
by (rel-auto, metis vwb-lens-wb wb-lens.get-put)
assumes \( vwb-lens \ x \)
shows \( (P \;;; \neg \$x \land Q) = (P[false/\$x\,] ;; Q[false/\$x\,]) \)
by (meson assms false-alt-def seqr-right-one-point upred-eq-false)

lemma \( \text{seqr-insert-ident-left} \):
assumes \( vwb-lens \ x \ $x' \not\in P \ $x \not\in Q \)
shows \((\$(x' =_{u} $x \land P) ;; Q) = (P ;; Q) \)
using assms
by (rel-simp, meson vwb-lens-wb wb-lens-weak weak-lens.put-get)

lemma \( \text{seqr-insert-ident-right} \):
assumes \( vwb-lens \ x \ $x' \not\in P \ $x \not\in Q \)
shows \((P ;; (x' =_{u} $x \land Q)) = (P ;; Q) \)
using assms
by (rel-simp, metis (no-types, hide-lams) vwb-lens-def wb-lens-def weak-lens.put-get)

lemma \( \text{seqr-var-ident-left} \):
assumes \( vwb-lens \ x \ $x' \not\in P \ $x \not\in Q \)
shows \((\$(x' =_{u} $x \land P) ;; (x' =_{u} $x \land Q)) = (\$(x' =_{u} $x \land P ;; Q) \)
using assms by (rel-auto', metis (no-types, lifting) vwb-lens-wb wb-lens-weak weak-lens.put-get)

lemma \( \text{seqr-bool-split} \):
assumes \( vwb-lens \ x \)
shows \( P ;; Q = (P[true/\$x\,] ;; Q[true/\$x\,] \lor P[false/\$x\,] ;; Q[false/\$x\,]) \)
using assms by (simp add: true-alt-def false-alt-def)

lemma \( \text{cond-inter-var-split} \):
assumes \( vwb-lens \ x \)
shows \( (P \land \$x' \Rightarrow Q) ;; R = (P[true/\$x\,] ;; R[true/\$x\,] \lor P[false/\$x\,] ;; R[false/\$x\,]) \)
proof –
have \( (P \land \$x' \Rightarrow Q) ;; R = ((\$x' \land P) ;; R \lor (\neg \$x' \land Q) ;; R) \)
  by (simp add: cond-def seqr-or-distl)
also have \( \ldots = ((\$x' \land P) ;; R \lor (Q \land \neg\$x') ;; R) \)
  by (rel-auto)
also have \( \ldots = (P[true/\$x\,] ;; R[true/\$x\,] \lor Q[false/\$x\,] ;; R[false/\$x\,]) \)
  by (simp add: seqr-left-one-point-true seqr-left-one-point-false assms)
finally show \( ?thesis \).
qed

theorem \( \text{seqr-pre-transfer} : \text{inl} \not\in q \Rightarrow ((P \land q) ;; R) = (P ;; (q \land R)) \)
by (rel-auto)

theorem \( \text{seqr-pre-transfer} : 
((P \land [q]_{<}) ;; R) = (P ;; ([q]_{<} \land R)) \)
by (rel-auto)

theorem \( \text{seqr-post-out} : \text{inl} \not\in r \Rightarrow (P ;; (Q \land r)) = ((P ;; Q) \land r) \)
by (rel-blast)

lemma \( \text{seqr-post-var-out} : 
fixes \ x :: \ (\text{bool} \Rightarrow 'a) \)
shows \( (P ;; (Q \land \$x')) = ((P ;; Q) \land \$x') \)
by (rel-auto)
**Theorem**  seqr-post-transfer: \( \text{out} \alpha \# q \implies (P ;; (q \land R)) = ((P \land q^\sim) ;; R) \)
by (rel-auto)

**Lemma**  seqr-pre-out: \( \text{out} \alpha \# p \implies ((p \land Q) ;; R) = (p \land (Q ;; R)) \)
by (rel-blast)

**Lemma**  seqr-pre-var-out:
fixes \( x :: (\text{bool} \implies 'a) \)
shows \( ((\exists x \cdot P x) ;; Q) = (\exists x \cdot (P ;; Q)) \)
by (rel-auto)

**Lemma**  seqr-true-lemma:
\( (P = (\neg ((\neg P) ;; \text{true}))) = (P = (P ;; \text{true})) \)
by (rel-auto)

**Lemma**  seqr-to-conj: \[ [ \text{out} \# P ; \text{ino} \# Q ] \implies (P ;; Q) = (P \land Q) \]
by (metis postcond-left-unit seqr-pre-out utp-pred-laws.inf-top.right-neutral)

**Lemma**  shEx-lift-seq-1 [uquant-lift]:
\( ((\exists x \cdot P x) ;; Q) = (\exists x \cdot (P ;; Q)) \)
by pred-auto

**Lemma**  shEx-lift-seq-2 [uquant-lift]:
\( (P ;; (\exists x \cdot Q x)) = (\exists x \cdot (P ;; Q x)) \)
by pred-auto

### 18.4 Iterated Sequential Composition Laws

**Lemma**  iter-seqr-nil [simp]: \( ;; [i :: [] \cdot P(i)] = II \)
by (simp add: seqr-iter-def)

**Lemma**  iter-seqr-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: \( P ;; (\bigsqcup A) = (\bigsqcup Q \in A. P ;; Q) \)
by (transfer, auto)

**Lemma**  seq-Sup-distr: \( (\bigsqcup A) ;; Q = (\bigsqcup P \in A. P ;; Q) \)
by (transfer, auto)

**Lemma**  seq-UINF-distl: \( P ;; (\Inf Q \in A \cdot F(Q)) = (\Inf Q \in A \cdot P ;; F(Q)) \)
by (simp add: UINF-as-Sup-collect seq-Sup-distl)

**Lemma**  seq-UINF-distl': \( P ;; (\forall Q \cdot F(Q)) = (\forall Q \cdot P ;; F(Q)) \)
by (metis UINF-mem-UNIV seq-UINF-distl)

**Lemma**  seq-UINF-distr: \( (\bigsqcup P \in A \cdot F(P)) ;; Q = (\bigsqcup P \in A \cdot F(P) ;; Q) \)
by (simp add: UINF-as-Sup-collect seq-Sup-distr)

**Lemma**  seq-UINF-distr': \( (\bigsqcup P \cdot F(P)) ;; Q = (\bigsqcup P \cdot F(P) ;; Q) \)
by (metis UINF-mem-UNIV seq-UINF-distr)

**Lemma**  seq-SUP-distl: \( P ;; (\bigsqcup i \in A. Q(i)) = (\bigsqcup i \in A. P ;; Q(i)) \)
by (metis image-image seq-Sup-distl)
lemma seq-SUP-distr: $(\coprod i \in A. P(i)) ;; Q = (\coprod i \in A. P(i) ;; Q)$
by (simp add: seq-Sup-distr)

18.6 Skip Laws

lemma cond-skip: $\alpha^\sharp \ b \Longrightarrow (b \land II) = (II \land b^-)$
by (rel-auto)

lemma pre-skip-post: $([b]_\prec \land II) = (II \land [b]_\succ)$
by (rel-auto)

lemma skip-var:
fixes $x :: (\text{bool} \Longrightarrow 'a)$
shows $(\sigma \land II) = (II \land (\sigma \cdot x'))$
by (rel-auto)

lemma skip-r-unfold:
$vwb-lens x \Longrightarrow II = ((\cdot x =^u \sigma \land II|_\alpha^\prec)$
by (rel-simp, metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens.get-put)

lemma skip-r-alpha-eq:
$I\!I = (\cdot v =^u \cdot v)$
by (rel-auto)

lemma skip-ra-unfold:
$I\!I x; y = ((\cdot x =^u \cdot x \land I\!I y)$
by (rel-auto)

lemma skip-res-as-ra:
$\begin{array}{c}
\llbracket vwb-lens y; x +_L y \approx_L 1; x \bowtie y \rrbracket \Longrightarrow I\!I|_\alpha^\prec x = I\!I y \\
apply (rel-auto) \\
apply (\text{metis (no-types, lifting) lens-indep-def}) \\
apply (\text{metis vwb-lens.put-eq}) \\
done
\end{array}$

18.7 Assignment Laws

lemma assigns-subst [usubst]:
$[\sigma]_a \uparrow (\cdot q)_a = (q \circ \sigma)_a$
by (rel-auto)

lemma assigns-r-comp: $((\sigma)_a ;; P) = ([\sigma]_a \uparrow P)$
by (rel-auto)

lemma assigns-r-feasible:
$((\sigma)_a ;; \text{true}) = \text{true}$
by (rel-auto)

lemma assign-subst [usubst]:
$\begin{array}{c}
\llbracket \text{mwb-lens x; mwb-lens y} \rrbracket \Longrightarrow [x \mapsto_s [u]_\prec] \uparrow (y ::= v) = (x, y) ::= (u, [x \mapsto_s u] \uparrow v)
\end{array}$
by (rel-auto)

lemma assign-vacuous-skip:
assumes $vwb-lens x$
shows $(x := \&x) = \text{II}$

113
using assms by rel-auto

lemma assign-simultaneous:
  assumes vwblens y x ▷ y
  shows (x,y) := (e, & y) = (x := e)
  by (simp add: assms usubst-upd-comm usubst-upd-var-id)

lemma assigns-idem: mwblens x \Rightarrow (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 \triangle b \triangleright g)_a = (f \triangle b \triangleright s g)_a
  by (rel-auto)

lemma assigns-r-conv:
  bij f \Rightarrow (f)_a = (inv f)_a
  by (rel-auto, simp-all add: bij-is-inj bij-is-surj surj-f-inv-f)

lemma assign-pred-transfer:
  fixes x :: (′a \Rightarrow ′a)
  assumes $x \# b out \alpha \# b$
  shows (b \land x := v) = (x := v \land b^{-})
  using assms by (rel-blast)

lemma assign-r-comp:
  x := u ;; P = P[⌈u⌉/x] \Rightarrow (x := e ;; x := f) = (x := f)
  by (simp add: assigns-r-comp usubst unrest)

lemma assign-commute:
  assumes x \triangle y x \triangle f y \triangle e
  shows (x := e ;; y := f) = (y := f ;; x := e)
  using assms by (rel-simp, simp-all add: lens-indep-comm)

lemma assign-cond:
  fixes x :: (′a \Rightarrow ′a)
  assumes outa \# b
  shows (x := e ;; (P \triangle b \triangleright Q)) = ((x := e ;; P) \triangle (b[⌈e⌉/x]] \triangleright (x := e ;; Q))
  by (rel-auto)

lemma assign-r-cond:
  fixes x :: (′a \Rightarrow ′a)
  shows (x := e ;; (P \triangle b \triangleright_r Q)) = ((x := e ;; P) \triangle (b[⌈e⌉/x]] \triangleright_r (x := e ;; Q))
  by (rel-auto)

lemma assign-r-alt-def:
  fixes x :: (′a \Rightarrow ′a)
  shows x := v = II[⌈v⌉/x]
by (rel-auto)

lemma assigns-r-ufunc: ufunctional $f_a$
  by (rel-auto)

lemma assigns-r-uinj: inj $f$ $\rightarrow$ uinj $f_a$
  by (rel-simp, simp add: inj-eq)

lemma assigns-r-swap-uinj:
\[ \text{vwb-lens } x; \text{vwb-lens } y; x \bowtie y \implies uinj ((x,y) := (k,y,kx)) \]
  by (metis assigns-r-uinj pr-var-def swap-usubst-inj)

lemma assign-unfold:
\[ \text{vwb-lens } x \implies (x := v) = (\text{v} x' = \text{v} x) \]
  by (rel-auto, auto simp add: comp-def)

using vwb-lens.put-eq by fastforce

18.8 Converse Laws

lemma convr-invol [simp]: $p^{-} = p$
  by pred-auto

lemma lit-convr [simp]: $\ll v \gg^{-} = \ll v \gg$
  by pred-auto

lemma uivar-convr [simp]:
  fixes $x :: (\text{'a } \implies \text{'a})$
  shows $(\text{v} x^{-}) = \text{v} x'$
  by pred-auto

lemma uovar-convr [simp]:
  fixes $x :: (\text{'a } \implies \text{'a})$
  shows $(\text{v} x')^{-} = \text{v} x$
  by pred-auto

lemma uop-convr [simp]: (uop $f$ $u$)$^{-} = uop f (u^{-})$
  by (pred-auto)

lemma bop-convr [simp]: (bop $f$ $u$ $v$)$^{-} = bop f (u^{-}) (v^{-})$
  by (pred-auto)

lemma eq-convr [simp]: ($p =_u q$)$^{-} = (p^{-} =_u q^{-})$
  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]: ($p ;; q$)$^{-} = (q^{-} ;; p^{-})$
  by (rel-auto)
lemma $\text{pre-convr}$ \[\text{simp}: \lceil p \rceil^- < \lceil p \rceil = \lceil p \rceil < \]
by (rel-auto)

lemma $\text{post-convr}$ \[\text{simp}: \lceil p \rceil > \lceil p \rceil = \lceil p \rceil < \]
by (rel-auto)

18.9 Assertion and Assumption Laws

declare \text{sublens-def} \text{lens-defs del}

lemma assume-false: \{false\} = false
by (rel-auto)

lemma assume-true: \{true\} = \Pi
by (rel-auto)

lemma assume-seq: \{b\} \{c\} = \{b \land c\}
by (rel-auto)

lemma assert-false: \{false\} = true
by (rel-auto)

lemma assert-true: \{true\} = \Pi
by (rel-auto)

lemma assert-seq: \{b\} \{c\} = \{b \land c\}
by (rel-auto)

18.10 Frame and Antiframe Laws

named-theorems frame

lemma frame-all \text{frame}: \Sigma;[P] = P
by (rel-auto)

lemma frame-none \text{frame}: 
\emptyset;[P] = (P \land \Pi)
by (rel-auto)

lemma frame-commute:
assumes \$y \notin P \; \$y' \notin P \; \$x \notin Q \; \$x' \notin Q \; x \bowtie y
shows \; x;[P] :: y;[Q] = y;[Q] :: x;[P]
apply \text{insert assms}
apply (rel-auto)
apply (rename-tac s s' s_0)
apply (subgoal-tac \text{s} \oplusL \text{s'} on y) \oplusL s_0 on x = s_0 \oplusL s' on y
apply (metis \text{lens-indep-get lens-indep-sym lens-override-def})
apply (simp add: \text{lens-indep.lens-put-comm lens-override-def})
apply (rename-tac \text{s} \text{s'} \text{s_0})
apply (subgoal-tac \text{put}_y (\text{put}_x \text{s} (\text{get}_x (\text{put}_x s_0 (\text{get}_x s'_0)))) (\text{get}_y (\text{put}_y s (\text{get}_y s_0)))
\text{put}_x s_0 (\text{get}_x s'_0))
apply (metis \text{lens-indep-get lens-indep-sym})
apply (metis \text{lens-indep.lens-put-comm})
done

lemma frame-contract-RID:
assumes \( \text{vwb-lens } x \ P \) is \( \text{RID}(x) \ x \bowtie y \)
shows \( (x;y);[P] = y;[P] \)
proof
- from \( \text{assms}(1,3) \) have \( (x;y);[\text{RID}(x)(P)] = y;[\text{RID}(x)(P)] \)
  apply (rel-auto)
  apply (simp add: lens-indep.lens-put-comm)
  apply (metis (no-types) \text{vwb-lens-wb wb-lens.get-put})
  done
thus \( \text{thesis} \)
  by (simp add: Healthy-if \( \text{assms} \))
qed

lemma frame-miracle [simp]:
\[ x;[\text{false}] = \text{false} \]
by (rel-auto)

lemma frame-skip [simp]:
\[ \text{vwb-lens } x \implies x;[\text{II}] = \text{II} \]
by (rel-auto)

lemma frame-assign-in [frame]:
\[ \{ \text{\$x,$x'} \} ; P ; \text{vwb-lens } x \implies \{ P \land x;[\text{true}] \} = x;[P] \]
by (rel-auto, simp-all add: lens-get-put-quasi-commute lens-put-of-quotient)

lemma frame-conj-true [frame]:
\[ \{ \{ \text{\$x,$x'} \} ; P ; \text{vwb-lens } x \} \implies \{ P \land x;[\text{true}] \} = x;[P] \]
by (rel-auto)

lemma frame-is-assign [frame]:
\[ \text{vwb-lens } x \implies x;[\{ \text{\$x'} = v \} \}_u = x;[v] \]
by (rel-auto)

lemma frame-seq [frame]:
\[ \{ \text{\$x,$x'} \} ; P ; \{ \text{\$x,$x'} \} ; Q \implies x;[P ; Q] = x;[P] ;; x;[Q] \]
apply (rel-auto)
apply (metis \text{mwb-lens.put-put \text{vwb-lens-mwb \text{vwb-lens-wb wb-lens-def weak-lens.put-get}}})
apply (metis \text{mwb-lens.put-put \text{vwb-lens-mwb}})
done

lemma frame-to-antiframe [frame]:
\[ \{ x \bowtie y ; x + L y = 1 L \} \implies x;[P] = y;[P] \]
by (rel-auto, \text{metis lens-indep-def, \text{metis lens-indep-def surj-pair}})

lemma rel-frext-miracle [frame]:
\[ a;[\text{false}]^+ = \text{false} \]
by (rel-auto)

lemma rel-frext-skip [frame]:
\[ \text{vwb-lens } a \implies a;[\text{II}]^+ = \text{II} \]
by (rel-auto)

lemma rel-frext-seq [frame]:
\[ \text{vwb-lens } a \implies a;[P] ^+ ; Q] ^+ = (a;[P] ^+ ;; a;[Q] ^+) \]
apply (rel-auto)
apply (rename-tac s s' so)
apply (rule-tac \(x=\text{put} \_a \_s \_s_0\) in exI)
apply (auto)
apply (metis vwb-lens.put-put vwb-lens-mwb)
done

lemma rel-frext-assigns [frame]:
\(\text{vwb-lens} \_a \implies a:\langle \sigma \_a \rangle^+ = \langle \sigma \_a \_s \_a \rangle_a\)
by (rel-auto)

lemma rel-frext-rcond [frame]:
\(a:[P \_a \_b \_r \_Q]^+ = (a:[P]+ \_a \_b \_r \_Q)^+\)
by (rel-auto)

lemma rel-frext-commute:
\(x \_a \_b \_y \implies x:[P] + y:[Q] = y:[Q] + x:[P]\)
apply (rename-tac \(a \_c \_b\))
apply (subgoal-tac \(\_\_b \_a \_a \_a \_y\))
apply (metis (no-types, hide-lams) lens-indep-comm lens-indep-get)
apply (simp add: lens-indep.lens-put-irr2)
apply (metis (no-types, hide-lams) lens-indep-comm lens-indep-get)
apply (simp-all add: lens-indep.lens-put-irr2)
done

lemma antiframe-disj [frame]:
\(x:[P] \lor x:[Q] = x:[P \lor Q]\)
by (rel-auto)

lemma antiframe-seq [frame]:
\(\text{vwb-lens} x \_x \_\_II \implies (x:[P] \_x \_\_II) = x:[P \_x \_\_II]\)
by (rel-auto)

declare sublens-def [lens-defs]

18.11 While Loop Laws

theorem while-unfold: 
\(\text{while } b \_d o P \_d o = ((P \_d o \_b \_d o P \_d o) \_b \_b \_r \_II)\)
proof –
have m:mono \((\_\_X. (P \_d o \_X) \_b \_b \_r \_II)\)
  by (auto intro: monoI seqr-mono cond-mono)
have \((\text{while } b \_d o P \_d o) = (\nu \_X. (P \_d o \_X) \_b \_b \_r \_II)\)
  by (simp add: while-def)
also have ... = ((P \_d o \_\_X. (P \_d o \_X) \_b \_b \_r \_II) \_b \_b \_r \_II)
  by (subst lfp-unfold, simp-all add: m)
also have ... = ((P \_d o \_\_while } b \_d o P \_d o) \_b \_b \_r \_II)
by (simp add: while-def)
finally show \(?\text{thesis} \).
qed

**theorem** while-false: while false do P od = II
by (subst while-unfold, rel-auto)

**theorem** while-true: while true do P od = false
apply (simp add: while-def alpha)
apply (rule antisym)
apply (simp-all)
apply (rule lfp-lowerbound)
apply (rel-auto)
done

**theorem** while-bot-unfold:
while \(\perp\) b do P od = ((P ;; while \(\perp\) b do P od) \(\triangleleft\) b \(\trianglerighteq\) II)

**proof**
have m:mono (\(\lambda\) X. (P ;; X) \(\triangleleft\) b \(\trianglerighteq\) II)
  by (auto intro: monol seqr-mono cond-mono)
have (while \(\perp\) b do P od) = (\(\mu\) X \(\cdot\) (P ;; X) \(\triangleleft\) b \(\trianglerighteq\) II)
  by (simp add: while-bot-def)
also have \(\ldots\) = ((P ;; (\(\mu\) X \(\cdot\) (P ;; X) \(\triangleleft\) b \(\trianglerighteq\) II)) \(\triangleleft\) b \(\trianglerighteq\) II)
  by (subst gfp-unfold, simp-all add: m)
also have \(\ldots\) = ((P ;; while \(\perp\) b do P od) \(\triangleleft\) b \(\trianglerighteq\) II)
  by (simp add: while-bot-def)
finally show \(?\text{thesis} \).
qed

**theorem** while-bot-false: while \(\perp\) false do P od = II
by (simp add: while-bot-def mu-const alpha)

**theorem** while-bot-true: while \(\perp\) true do P od = (\(\mu\) X \(\cdot\) 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 ;; true \(_h\) = true \(\Rightarrow\) while \(\perp\) true do P od = true
apply (simp add: while-bot-true)
apply (rule antisym)
apply (simp)
apply (rule gfp-upperbound)
apply (simp)
done

### 18.12 Algebraic Properties

**interpretation** upred-semiring: semiring-I
where times = seqr and one = skip-r and zero = false\(_h\) 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 \Rightarrow \text{nat} \Rightarrow 'a\ hrel\ (infixr \(^\_\))\) where
upower P n \equiv upred-semiring.power P n
translations

\[ P \uparrow i \leq \text{CONST power.power II op }; P i \]
\[ P \uparrow i \leq (\text{CONST power.power II op }; P) i \]

Set up transfer tactic for powers

lemma upower-rep-eq:
\[ [P \uparrow i]_e = (\lambda b. b \in ([p. [P]_e p] \uparrow i)) \]
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:
\[ [\text{power.power } \langle \text{id} \rangle_\alpha \text{ op }; P i]_e = (\lambda b. b \in ([p. [P]_e p] \uparrow i)) \]
by (metis skip-r-def upower-rep-eq)

update-uexpr-rep-eq-thms

lemma Sup-power-expand:
fixes P :: 'a::complete-lattice
shows \( P(0) \cap (\prod_i. P(i+1)) = (\prod_i. P(i)) \)
proof
  have UNIV = insert (0::nat) \{1..\}
    by auto
  moreover have \( (\prod_i. P(i)) = \prod \text{ (P ' UNIV) \} \)
    by (blast)
  moreover have \( \prod \text{ (P ' insert 0 \{1..\}) = P(0) \cap SUPREMUM \{1..\} P \}
    by (simp)
  moreover have \( SUPREMUM \{1..\} P = (\prod_i. P(i+1)) \)
    by (simp add: atLeast-Suc-greaterThan greaterThan-0)
  ultimately show ?thesis
    by (simp only:)
qed

lemma Sup-upto-Suc:
\( (\prod_i \in \{0..Suc n\}. P \uparrow i) = (\prod_i \in \{0..n\}. P \uparrow i) \cap P \uparrow Suc n \)
proof
  have \( (\prod_i \in \{0..Suc n\}. P \uparrow i) = (\prod_i \in \{0..n\}. P \uparrow i) \)
    by (simp add: atLeast0-atMost-Suc)
  also have ... = \( P \uparrow Suc n \cap (\prod_i \in \{0..n\}. P \uparrow i) \)
    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: \( Q \subseteq (P \sqcap Q \cap R) \Longrightarrow Q \subseteq P \uparrow n \sqcap R \)
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 segr-assoc segr-mono)
qed

lemma upower-inductr:
  assumes Q ⊑ (R ∩ Q ;; P)
  shows Q ⊑ R ;; (P ^ n)
  using assms proof (induct n)
    case 0
    then show ?case by auto
  next
    case (Suc n)
    have R ;; P ^ Suc n = (R ;; P ^ n) ;; P
    by (metis seqr-assoc upred-semiring.power-Suc2)
    also have Q ⊑ ...
    by (meson Suc.hyps assms eq-iff segr-mono)
    also have Q ⊑ ...
    using assms by auto
    finally show ?case .
qed

lemma SUP-atLeastAtMost-first:
  fixes P :: nat ⇒ 'a::complete-lattice
  assumes m ≤ n
  shows (⨆ i∈{m..n}. P(i)) = P(m) ∩ (⨆ i∈{Suc m..n}. P(i))
  by (metis SUP-insert assms atLeastAtMost-insertL)

lemma upower-seqr-iter: P ^ n = (;; Q : replicate n P · Q)
  by (induct n, simp-all add: upred-semiring.power-Suc)

lemma assigns-power: ⟨f⟩ ^ n = ⟨f ^ n⟩ ^ n
  by (induct n, rel-auto+)

18.12.1 Kleene Star

definition ustar :: 'a hrel ⇒ 'a hrel (⋅^* [999] 999) where
P ^* = (⨆ i∈{0..} · P ^ i)

lemma ustar-rep-epq:
  [P ^*]_e = (λb. b ∈ ([P]_e p)^*))
  by (simp add: ustar-def, rel-auto, simp-all add: relpow-imp-rtrancl rtrancl-imp-relpow)

update-uexprrep-epq-thms

18.13 Kleene Plus

purge-notation trancl ((⋅^+) [1000] 999)

definition uplus :: 'a hrel ⇒ 'a hrel (⋅^{+} [999] 999) where
[upred-defs]: P ^+ = P ;; P ^*

lemma uplus-power-def: P ^+ = (⨆ i · P ^ (Suc i))
  by (simp add: uplus-def ustar-def seq-UINF-distl UINF-atLeast-Suc upred-semiring.power-Suc)
18.14 Omega

\textbf{definition omega :: }'a hrel \Rightarrow 'a hrel (\cdot [999] 999) \textbf{ where}
\[ P^\omega = (\mu X \cdot P \; ; X) \]

18.15 Relation Algebra Laws

\textbf{theorem RA1: } (P ;; (Q ;; R)) = ((P ;; Q) ;; R)
\begin{itemize}
  \item \textbf{by (simp add: seqr-assoc)}
\end{itemize}

\textbf{theorem RA2: } (P ;; II) = P (II ;; P) = P
\begin{itemize}
  \item \textbf{by simp-all}
\end{itemize}

\textbf{theorem RA3: } P^\omega = P
\begin{itemize}
  \item \textbf{by simp}
\end{itemize}

\textbf{theorem RA4: } (P ;; Q)^\omega = (Q^\omega ;; P^\omega)
\begin{itemize}
  \item \textbf{by simp}
\end{itemize}

\textbf{theorem RA5: } (P \lor Q)^\omega = (P^\omega \lor Q^\omega)
\begin{itemize}
  \item \textbf{by (rel-auto)}
\end{itemize}

\textbf{theorem RA6: } ((P \lor Q) ;; R) = (P ;; R \lor Q ;; R)
\begin{itemize}
  \item \textbf{using seqr-or-distl by blast}
\end{itemize}

\textbf{theorem RA7: } ((P^\omega ;; (\neg(P ;; Q))) \lor (\neg Q)) = (\neg Q)
\begin{itemize}
  \item \textbf{by (rel-auto)}
\end{itemize}

18.16 Kleene Algebra Laws

\textbf{lemma ustar-alt-def: } P^\star = (\bigcap_i P^\omega \cdot i)
\begin{itemize}
  \item \textbf{by (simp add: ustar-def)}
\end{itemize}

\textbf{theorem ustar-sub-unfoldl: } P^\star \subseteq II \cap P ;; P^\star
\begin{itemize}
  \item \textbf{by (rel-simp, simp add: rtrancl-into-trancl2 trancl-into-rtrancl)}
\end{itemize}

\textbf{theorem ustar-inductl:}
\begin{itemize}
  \item \textbf{assumes } Q \subseteq R \; Q \subseteq P ;; Q
  \item \textbf{shows } Q \subseteq P^\star ;; R
\end{itemize}
\begin{itemize}
  \item \textbf{proof} –
  \item \textbf{have } P^\star ;; R = (\bigcap_i P^\omega \cdot i ;; R)
    \begin{itemize}
      \item \textbf{by (simp add: ustar-def UINF-as-Sup-collect seq-SUP-distr)}
    \end{itemize}
  \item \textbf{also have } Q \subseteq ... 
    \begin{itemize}
      \item \textbf{by (simp add: SUP-least assms upower-inductl)}
    \end{itemize}
\end{itemize}
\begin{itemize}
  \item \textbf{finally show } \textit{thesis} .
\end{itemize}
\textbf{qed}

\textbf{theorem ustar-inductr:}
\begin{itemize}
  \item \textbf{assumes } Q \subseteq R \; Q \subseteq Q ;; P
  \item \textbf{shows } Q \subseteq R ;; P^\star
\end{itemize}
\begin{itemize}
  \item \textbf{proof} –
  \item \textbf{have } R ;; P^\star = (\bigcap_i R ;; P^\omega \cdot i)
    \begin{itemize}
      \item \textbf{by (simp add: ustar-def UINF-as-Sup-collect seq-SUP-distl)}
    \end{itemize}
  \item \textbf{also have } Q \subseteq ...
    \begin{itemize}
      \item \textbf{by (simp add: SUP-least assms upower-inductr)}
    \end{itemize}
\end{itemize}
\begin{itemize}
  \item \textbf{finally show } \textit{thesis} .
\end{itemize}
lemma ustar-refines-nu: \((\nu X \cdot P :: X \sqcap I) \subseteq P^*\)
by (metis (no-types, lifting) lfp-greatest semilattice-sup-class.le-sup-iff
    semilattice-sup-class.sup-idem upred-semiring.multiplication-2-right
    upred-semiring.one-add-one ustar-induct))

lemma ustar-as-nu: \(P^* = (\nu X \cdot P :: X \sqcap I)\)
proof (rule antisym)
  show \((\nu X \cdot P :: X \sqcap I) \subseteq P^*\)
  by (simp add: ustar-refines-nu)
  show \(P^* \subseteq (\nu X \cdot P :: X \sqcap I)\)
    by (metis lfp-lowerbound upred-semiring.add-commute ustar-sub-unfold)
qed

lemma ustar-unfold: \(P^* = \Pi (P :: P^*)\)
apply (simp add: ustar-as-nu)
apply (subst lfp-unfold)
apply (rule monoid)
apply (rel-auto+)
done

While loop can be expressed using Kleene star

lemma while-star-form:
while b do P od = \((P \sqsubset b \triangleright_r II)^* :: [\neg b]\top\)
proof
  have 1: Continuous \((\lambda X. P :: X \sqsubset b \triangleright_r II)\)
    by (rel-auto)
  have while b do P od = \(\prod i. ((\lambda X. P :: X \sqsubset b \triangleright_r II) \triangleright (i+1))\ false)\)
    by (simp add: 1 false-upred-def sup-continuous-Continuous sup-continuous-lfp while-def)
  also have ... = \((\prod i. ((\lambda X. P :: X \sqsubset b \triangleright_r II) \triangleright (i+1))\ false)\)
    by (subst sup-power-expand, simp)
  also have ... = \((\prod i. (\lambda X. P :: X \sqsubset b \triangleright_r II) \triangleright (i+1))\ false)\)
    by (simp)
  also have ... = \(\prod i. (P \sqsubset b \triangleright_r II)^* i :: (false \sqsubset b \triangleright_r II)\)
proof (rule SUP-cong, simp-alt)
  fix i
  show P :: \((\lambda X. P :: X \sqsubset b \triangleright_r II) \triangleright i\ false \sqsubset b \triangleright_r II = (P \sqsubset b \triangleright_r II) \triangleright i :: (false \sqsubset b \triangleright_r II)\)
proof (induct i)
  case 0
    then show ?case by simp
next
  case (Suc i)
  then show ?case
    by (simp add: upred-semiring.power-Suc)
  (metis (no-types, lifting) RA1 comp-cond-left-distr cond-L6 resugar-cond upred-semiring.multiplication-left-neutral)
qed

qed

also have ... = \((\prod i \in \{0..\} \cdot (P \sqsubset b \triangleright_r II)^* i :: [\neg b]\top)\)
by (rel-auto)
also have ... = \((P \sqsubset b \triangleright_r II)^* :: [\neg b]\top)\)
    by (metis seq-UINF-distr ustar-def)
finally show ?thesis.
qed
18.17 Omega Algebra Laws

lemma uomega-induct:
\[ P ;\ P^\omega \sqsubseteq P^\omega \]
by (simp add: uomega-def, metis eq-refl gfp-unfold monoI seqr-mono)

18.18 Refinement Laws

lemma skip-r-refine:
\[ (p \Rightarrow p) \sqsubseteq II \]
by pred-blast

lemma conj-refine-left:
\[ (Q \Rightarrow P) \sqsubseteq R \implies P \sqsubseteq (Q \land R) \]
by (rel-auto)

lemma pre-weak-rel:
assumes ‘Pre ⇒ I’
and \( (I \Rightarrow Post) \sqsubseteq P \)
shows \( (Pre \Rightarrow Post) \sqsubseteq P \)
using assms by (rel-auto)

lemma cond-refine-rel:
assumes \( S \sqsubseteq ([b]_< \land P) \)
\( S \sqsubseteq ([\neg b]_< \land Q) \)
shows \( S \sqsubseteq P \triangleleft_b r Q \)
by (metis aext-not assms (1) assms (2) cond-def lift-rcond-def utp-pred-laws.le-sup-iff)

lemma seq-refine-pred:
assumes \( ([b]_< \Rightarrow [s]_>) \sqsubseteq P \) and \( ([s]_< \Rightarrow [c]_>) \sqsubseteq Q \)
shows \( ([b]_< \Rightarrow [c]_>) \sqsubseteq (P ;; Q) \)
using assms by rel-auto

lemma seq-refine-unrest:
assumes outa \( \frac{b}{c} \) ina \( \frac{b}{c} \)
assumes \( b \Rightarrow [s]_> \sqsubseteq P \) and \( ([s]_< \Rightarrow c) \sqsubseteq Q \)
shows \( b \Rightarrow c \sqsubseteq (P ;; Q) \)
using assms by rel-blast

18.19 Domain and Range Laws

lemma Dom-conv-Ran:
\( \text{Dom}(P^\omega) = \text{Ran}(P) \)
by (rel-auto)

lemma Ran-conv-Dom:
\( \text{Ran}(P^\omega) = \text{Dom}(P) \)
by (rel-auto)

lemma Dom-skip:
\( \text{Dom}(II) = \text{true} \)
by (rel-auto)

lemma Dom-assigns:
\( \text{Dom}((\sigma)_{\alpha}) = \text{true} \)
by (rel-auto)
lemma Dom-miracle:
  \( \text{Dom}(\text{false}) = \text{false} \)
by (rel-auto)

lemma Dom-assume:
  \( \text{Dom}(\text{[b]} \top) = b \)
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:
  \( \text{Dom}(P \sqcap Q) = (\text{Dom}(P) \lor \text{Dom}(Q)) \)
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 = (op =), le = op ⊆)

\( P \) 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 \( A :: ‘α \text{ upred set} \)

show \( \exists s . \text{is-lub} (\text{carrier} = \text{UNIV}, eq = \text{op =}, le = \text{op ⊆}) s A \)
  apply (rule-tac x=⨆ A in exI)
  apply (rule least-UpperI)
  apply (auto intro: Inf-greatest simp add: Inf-lower Upper-def)
  done

show \( \exists i . \text{is-glb} (\text{carrier} = \text{UNIV}, eq = \text{op =}, le = \text{op ⊆}) i A \)
  apply (rule-tac x=⨰ A in exI)
  apply (rule greatest-LowerI)

end
apply (auto intro: Sup-least simp add: Sup-upper Lower-def)
done
qed

lemma upred-weak-complete-lattice [simp]: weak-complete-lattice \( \mathcal{P} \)
by (simp add: upred-lattice.weak.weak-complete-lattice-axioms)

lemma upred-lattice-eq [simp]:
  \( \text{op} = \text{op} \)
by (simp add: upred-lattice-def)

lemma upred-lattice-le [simp]:
  \( \text{le} = (P \subseteq Q) \)
by (simp add: upred-lattice-def)

lemma upred-lattice-carrier [simp]:
  \( \text{carrier} = \text{UNIV} \)
by (simp add: upred-lattice-def)

lemma Healthy-fixed-points [simp]:
  \( \text{fps} = [H]_H \)
by (simp add: fps-def upred-lattice-def Healthy-def)

lemma upred-lattice-Idempotent [simp]:
  \( \text{Idem} = \text{Idempotent} \)
using upred-lattice.weak-partial-order-axioms by (auto simp add: idempotent-def Idempotent-def)

lemma upred-lattice-Monotonic [simp]:
  \( \text{Mono} = \text{Monotonic} \)
using upred-lattice.weak-partial-order-axioms by (auto simp add: isotone-def mono-def)

19.2 UTP theories hierarchy

typedef \( (\'T, \'a) \text{uthy} = \text{UNIV :: unit set} \)
by auto

We create a unitary parametric type to represent UTP theories. These are merely tags and contain no data other than to help the type-system resolve polymorphic definitions. The two parameters denote the name of the UTP theory – as a unique type – and the minimal alphabet that the UTP theory requires. We will then use Isabelle’s ad-hoc overloading mechanism to associate theory constructs, like healthiness conditions and units, with each of these types. This will allow the type system to retrieve definitions based on a particular theory context.

definition uthy :: \( ('a, 'b) uthy \) where
  uthy = Abs-uthy()

lemma uthy-eq [intro]:
  fixes x y :: \( ('a, 'b) uthy \)
  shows \( x = y \)
  by (cases x, cases y, simp)

syntax
  
-UTHY :: type \( \Rightarrow \) type \( \Rightarrow \) logic (UTHY('\_, '\_'))

translations
  UTHY('T, 'a) \( \Rightarrow \) CONST uthy :: \( ('T, 'a) uthy \)

We set up polymorphic constants to denote the healthiness conditions associated with a UTP theory. Unfortunately we can currently only characterise UTP theories of homogeneous rela-
tions; this is due to restrictions in the instantiation of Isabelle’s polymorphic constants which apparently cannot specialise types in this way.

**consts**

\[\text{utp-hcond :: } (\tau , \alpha ) \text{ uthy } \Rightarrow (\alpha \times \alpha ) \text{ health } (H)\]

**definition**

\[\text{utp-order :: } (\alpha \times \alpha ) \text{ health } \Rightarrow \alpha \text{ hrel order where}\]

\[\text{utp-order } H = \{\text{carrier } = \{P. \ P \text{ is } H\}, \ eq = (op =), \ le = op \subseteq \}\]

**abbreviation**

\[\text{uthy-order } T \equiv \text{utp-order } H_T\]

Constant \text{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**

\[\text{utp-order-carrier } [\text{simpl}]:\]

\[\text{carrier } (\text{utp-order } H) = [H]_H\]

**by** (simp add: utp-order-def)

**lemma**

\[\text{utp-order-eq } [\text{simpl}]:\]

\[\text{eq } (\text{utp-order } T) = op =\]

**by** (simp add: utp-order-def)

**lemma**

\[\text{utp-order-le } [\text{simpl}]:\]

\[\text{le } (\text{utp-order } T) = op \subseteq\]

**by** (simp add: utp-order-def)

**lemma**

\[\text{utp-partial-order: partial-order } (\text{utp-order } T)\]

**by** (unfold-locales, simp-all add: utp-order-def)

**lemma**

\[\text{utp-weak-partial-order: weak-partial-order } (\text{utp-order } T)\]

**by** (unfold-locales, simp-all add: utp-order-def)

**lemma**

\[\text{mono-Monotone-utp-order:}\]

\[\text{mono } f \implies \text{Monotone } (\text{utp-order } T) f\]

**apply** (auto simp add: isotone-def)

**apply** (metis partial-order-def utp-partial-order)

**apply** (metis monoD)

**done**

**lemma**

\[\text{isotone-utp-orderI}: \text{Monotonic } H \implies \text{isotone } (\text{utp-order } X) (\text{utp-order } Y) H\]

**by** (auto simp add: mono-def isotone-def utp-weak-partial-order)

**lemma**

\[\text{Mono-utp-orderI}:\]

\[\equiv \land P Q. \equiv P \subseteq Q; \ P \text{ is } H; \ Q \text{ is } H \implies F(P) \subseteq F(Q) \implies \text{Mono}_{\text{utp-order } H} F\]

**by** (auto simp add: isotone-def utp-weak-partial-order)

The UTP order can equivalently be characterised as the fixed point lattice, fpl.

**lemma**

\[\text{utp-order-fpl: utp-order } H = \text{fpl } P H\]

**by** (auto simp add: utp-order-def upred-lattice-def fps-def Healthy-def)

**definition**

\[\text{uth-eq :: } (\tau_1 , \alpha ) \text{ uthy } \Rightarrow (\tau_2 , \alpha ) \text{ uthy } \Rightarrow \text{bool } (\text{infix } \approx_T 50) \text{ where}\]

\[\tau_1 \approx_T \tau_2 \iff [\text{utp-order } H]_{\tau_1} = [\text{utp-order } H]_{\tau_2}\]

**lemma**

\[\text{uth-eq-refl}: \tau \approx_T T\]

**by** (simp add: uth-eq-def)
lemma uth-eq-sym: \( T_1 \approx_T T_2 \leftrightarrow T_2 \approx_T T_1 \)
by (auto simp add: uth-eq-def)

lemma uth-eq-trans: \( [T_1 \approx_T T_2; T_2 \approx_T T_3] \implies T_1 \approx_T T_3 \)
by (auto simp add: uth-eq-def)

definition uthy-plus :: \( ('T_1, 'a) uthy \Rightarrow ('T_2, 'a) uthy \Rightarrow ('T_1 \times 'T_2, 'a) uthy \) (infixl +T 65) where
uthy-plus \( T_1 T_2 = uthy \)

overloading
prod-hcond == utp-hcond :: \( ('T_1 \times 'T_2, 'a) uthy \Rightarrow ('a \times 'a) upred \) where
prod-hcond \( T = H_{UTHY}('T_1, 'a) \circ H_{UTHY}('T_2, 'a) \)

begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition prod-hcond :: \( ('T_1 \times 'T_2, 'a) uthy \Rightarrow ('a \times 'a) upred \) where
prod-hcond \( T = H_{UTHY}('T_1, 'a) \circ H_{UTHY}('T_2, 'a) \)

end

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 \( T :: ('T_1, 'a) uthy \) (structure)
assumes HCond-Idem: \( H(H(P)) = H(P) \)
begin

lemma uthy-simp:
  \( uthy = T \)
by blast

A UTP theory fixes \( T \), the structural element denoting the UTP theory. All constants associated with UTP theories can then be resolved by the type system.

lemma HCond-Idempotent [closure, intro]: Idempotent \( H \)
by (simp add: Idempotent-def HCond-Idem)

sublocale partial-order uthy-order \( T \)
by (unfold-locales, simp-all add: utp-order-def)
end

Theory summation is commutative provided the healthiness conditions commute.

lemma uthy-plus-comm:
assumes \( H_{T_1} \circ H_{T_2} = H_{T_2} \circ H_{T_1} \)
shows \( T_1 +_T T_2 \approx_T T_2 +_T T_1 \)
proof
  have \( T_1 = uthy T_2 = uthy \)
  by blast
thus \( ?thesis \)
  using assms by (simp add: uth-eq-def prod-hcond-def)
qed

lemma uthy-plus-assoc: \( T_1 +_T (T_2 +_T T_3) \approx_T (T_1 +_T T_2) +_T T_3 \)
by (simp add: uth-eq-def prod-hcond-def comp-def)

lemma uthy-plus-idem: utp-theory T \implies T \plus T \approx T by (simp add: uth-eq-def prod-hcond-def Healthy-def utp-theory. HCond-Idem utp-theory. uthy-simp)

locale utp-theory-lattice = utp-theory T + complete-lattice uthy-order T for T :: ('T, 'α) uthy (structure)

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, such as the Knaster-Tarski theorem. We can also retrieve lattice operators as below.

abbreviation utp-top (\top)
where utp-top T \equiv top (uthy-order T)

abbreviation utp-bottom (\bot)
where utp-bottom T \equiv bottom (uthy-order T)

abbreviation utp-join (infixl \⊔ 65) where utp-join T \equiv join (uthy-order T)

abbreviation utp-meet (infixl \⊓ 70) where utp-meet T \equiv meet (uthy-order T)

abbreviation utp-sup (\bigvee [0,10]) where utp-sup T \equiv Lattice.sup (uthy-order T)

abbreviation utp-inf (\bigwedge [0,10]) where utp-inf T \equiv Lattice.inf (uthy-order T)

abbreviation utp-gfp (\nu) where utp-gfp T \equiv GREATEST-FP (uthy-order T)

abbreviation utp-lfp (\mu) where utp-lfp T \equiv LEAST-FP (uthy-order T)

syntax
-tnu :: logic \Rightarrow pttrn \Rightarrow logic \Rightarrow logic (\mu \cdot \cdot \cdot [0,10] 10)
-tnu :: logic \Rightarrow pttrn \Rightarrow logic \Rightarrow logic (\nu \cdot \cdot \cdot [0,10] 10)

notation \text{gfp} (\mu)
notation \text{lfp} (\nu)

translations
\mu(T \cdot P) \equiv CONST \text{utp-lfp} T (\lambda X. P)
\nu(T \cdot P) \equiv CONST \text{utp-gfp} T (\lambda X. P)

lemma upred-lattice-inf:
Lattice.inf \mathcal{P} \equiv \bigcap A

by (metis Sup-least Sup-upper UNIV-I antisym-conv 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.

classification utp-theory-lattice
begin

lemma LFP-healthy-comp: \mu F = \mu (F \circ \mathcal{H})
proof
  have \{ P. (P is \mathcal{H}) \land F P \subseteq P \} = \{ P. (P is \mathcal{H}) \land F (\mathcal{H} P) \subseteq P \}
  by (auto simp add: Healthy-def)
thus \textit{?thesis}
  by (simp add: LEAST-FP-def)
qed

lemma GFP-healthy-comp: \nu F = \nu (F \circ \mathcal{H})
proof
  have \{ P. (P is \mathcal{H}) \land P \subseteq F P \} = \{ P. (P is \mathcal{H}) \land P \subseteq F (\mathcal{H} P) \}
  by (auto simp add: Healthy-def)
thus \textit{?thesis}
  by (simp add: GREATEST-FP-def)
qed

lemma top-healthy [closure]: \top is \mathcal{H}
  using weak.top-closed by auto
lemma bottom-healthy [closure]: \bot is \mathcal{H}
  using weak.bottom-closed by auto
lemma utp-top: P is \mathcal{H} \implies P \subseteq \top
  using weak.top-higher by auto
lemma utp-bottom: P is \mathcal{H} \implies \bot \subseteq P
  using weak.bottom-lower by auto
end

lemma upred-top: \top P = \textit{false}
  using ball-UNIV greatest-def by fastforce
lemma upred-bottom: \bot P = \textit{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 \mathcal{H}

sublocale utp-theory-mono \subseteq utp-theory-lattice
proof
  We can then use the Knaster-Tarski theorem to obtain a complete lattice, and thus provide all the usual properties.

  interpret weak-complete-lattice fpl P \mathcal{H}
  by (rule Knaster-Tarski, auto simp add: upred-lattice.weak.weak-complete-lattice-axioms)

  have complete-lattice (fpl P \mathcal{H})
  by (unfold-locales, simp add: fps-def sup-exists, (blast intro: sup-exists inf-exists)+)

  hence complete-lattice (uthy-order T)
  by (simp add: utp-order-def, simp add: upred-lattice-def)

  thus utp-theory-lattice T
by (simp add: utp-theory-axioms utp-theory-lattice-def)

qed

context utp-theory-mono

begin

In a monotone theory, the top and bottom can always be obtained by applying the healthiness condition to the predicate top and bottom, respectively.

lemma healthy-top: \( \top = H(false) \)

proof –
  have \( \top = \top fpl P H \)
    by (simp add: utp-order-fpl)
  also have \( ... = H\top P \)
    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: \( \bot = H(true) \)

proof –
  have \( \bot = \bot fpl P H \)
    by (simp add: utp-order-fpl)
  also have \( ... = H\bot P \)
    using Knaster-Tarski-idem-extremes(2)[of \( P H \)]
    by (simp add: HCond-Idempotent HCond-Mono)
  also have \( ... = H true \)
    by (simp add: upred-bottom)
  finally show ?thesis .

qed

lemma healthy-inf:
  assumes \( A \subseteq [\[ H]\] \)
  shows \( \bigsqcap A = H(\bigsqcap A) \)

proof –
  have 1: weak-complete-lattice (uthy-order \( T \))
    by (simp add: weak.weak-complete-lattice-axioms)
  have 2: Mono_uthy-order \( P H \)
    by (simp add: HCond-Mono isotone-utp-orderI)
  have 3: Idem_uthy-order \( T H \)
    by (simp add: HCond-Idem idempotent-def)
  show ?thesis
    using Knaster-Tarski-idem-inf-eq[OF upred-weak-complete-lattice, of \( H \)]
    by (simp, metis HCond-Idempotent HCond-Mono assms partial-object.simps(3) upred-lattice-def
    upred-lattice-inf utp-order-def)

qed

end

locale utp-theory-continuous = utp-theory +
  assumes HCond-Cont [closure,intro]: Continuous \( H \)

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 $A \subseteq \llbracket \mathcal{H} \rrbracket_H \ A \neq \{\}$
  shows $\bigsqcap A = \bigsqcap \mathcal{H}A$
proof
  have $\bigsqcap A = \bigsqcap (\mathcal{H} \ A)$
    using Continuous-def HCond-Cont assms(1) assms(2) healthy-inf by auto
  also have $\ldots = \bigsqcap A$
    by (unfold Healthy-carrier-image[OF assms(1)], simp)
  finally show $?$thesis .
qed

lemma healthy-inf-def:
  assumes $A \subseteq \llbracket \mathcal{H} \rrbracket_H$
  shows $\bigsqcap A = (\text{if } (A = \{\}) \text{ then } T \text{ else } (\bigsqcap A))$
  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 $P \text{ is } \mathcal{H} \ Q \text{ is } \mathcal{H}$
  shows $P \cap Q = \mathcal{H}$
  by (metis Continuous-Disjunctuous Disjunctuous-def HCond-Cont Healthy-def' assms(1) assms(2))

lemma meet-bottom [simp]:
  assumes $P \text{ is } \mathcal{H}$
  shows $P \cap \bot = \bot$
  by (simp add: assms semilattice-sup-class.sup-absorb2 utp-bottom)

lemma meet-top [simp]:
  assumes $P \text{ is } \mathcal{H}$
  shows $P \cap \top = P$
  by (simp 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 $F \ F \in \llbracket \mathcal{H} \rrbracket_H \rightarrow \llbracket \mathcal{H} \rrbracket_H$
  shows $\mu F = (\mu X \cdot F(\mathcal{H}(X)))$
proof (rule antisym)
  have ne: $\{P. \ (P \text{ is } \mathcal{H}) \wedge F P \subseteq P\} \neq \{\}$
  proof
    have $F' \top \subseteq \top$
      using assms(2) utp-top weak.top-closed by force
    thus $?$thesis
  qed

132
by (auto, rule-tac $x=\top$ in exI, auto simp add: top-healthy)

qed

show $\mu F \subseteq (\mu X \cdot F (H X))$
proof
  have $\prod \{ P. (P is H) \land F(P) \subseteq P \} \subseteq \prod \{ P. F(H(P)) \subseteq P \}$
  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 $a: F (H P) \subseteq P$
      hence $F (F (H P)) \subseteq (H P)$
        by (metis 1 HCond-Mono mono_def)
      show $F (H P) \in \{ P. (P is H) \land F P \subseteq P \}$
      proof (auto)
        show $F (H P)$ is $H$
          by (metis 1 Healthy-def)
        show $F (F (H P)) \subseteq F (H P)$
          using $F$ mono_def assms(1) by blast
      qed
      show $F (H P) \subseteq P$
        by (simp add: $a$)
      qed
      qed
    qed
  qed

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 $(\mu X \cdot F (H X)) \subseteq \mu F$
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 $\prod i. P(i)$ is $H$
  shows $[ \prod i \cdot P(i) ]$ is $H$
  by (simp add: HCond-Cont UINF-Continuous-closed 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]: $[ P is H; Q is H ] \Rightarrow (P ;; Q)$ is $H$
begin

lemma upower-Suc-Healthy [closure]:
  assumes $P$ is $H$

133
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-continuous + utp-theory-rel
begin

lemma seq-cont-Sup-distl:
assumes \( P \) is \( H \)
\( A \subseteq [\H]_H \) \( A \neq \{\} \)
shows \( \bigcap \{ P ;; Q | Q. Q \in A \} \)
proof
have \( \{ P ;; Q | Q. Q \in A \} \subseteq [\H]_H \)
using Healthy-Sequence assms \( (1) \) \( \text{assms}(2) \) by (auto)
thus \( \text{thesis} \)
by (simp add: healthy-inf-cont seq-Sup-distl setcompr-eq-image assms)
qed

lemma seq-cont-Sup-distr:
assumes \( Q \) is \( H \)
\( A \subseteq [\H]_H \) \( A \neq \{\} \)
shows \( \bigcap \{ P ;; Q | P. P \in A \} \)
proof
have \( \{ P ;; Q | P. P \in A \} \subseteq [\H]_H \)
using Healthy-Sequence assms \( (1) \) \( \text{assms}(2) \) by (auto)
thus \( \text{thesis} \)
by (simp add: healthy-inf-cont seq-Sup-distr setcompr-eq-image assms)
qed

lemma uplus-healthy [closure]:
assumes \( P \) is \( H \)
shows \( P^+ \) is \( H \)
by (simp add: uplus-power-def closure assms)

end

There also exist UTP theories with units, and the following operator is a theory specific operator for them.

consts
\( \text{utp-unit} :: (\'T, \'α) \text{uthy} \Rightarrow \'α \text{ hrel} (II) \)

We can characterise the theory Kleene star by lifting the relational one.

definition \( \text{utp-star} \) (-\( \star \)) \( [999] \) \( 999 \) where
[upred-defs]: \( \text{utp-star} \ T \ P = (P^* ;; II\ T) \)

We can then characterise tests as refinements of units.

definition \( \text{utest} \) (-\( \star \)) \( \text{uthy} \Rightarrow \'α \text{ hrel} \Rightarrow \text{bool} \) where
[upred-defs]: \( \text{utest} \ T \ b = (II\ T \subseteq b) \)

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-left-unital =
\( \text{utp-theory-rel} + \)
assumes Healthy-Left-Unit [closure]: \( II \) is \( H \)
and Left-Unit: \( P \) is \( H \) \( \Rightarrow (II ;; P) = P \)

134
locale utp-theory-right-unital =
  utp-theory-rel +
assumes Healthy-Right-Unit [closure]: \( \Pi \) is \( \mathcal{H} \)
and Right-Unit: \( P \) is \( \mathcal{H} \Rightarrow (P \sqcup \Pi) = P \)

locale utp-theory-unital =
  utp-theory-rel +
assumes Healthy-Unit [closure]: \( \Pi \) is \( \mathcal{H} \)
and Unit-Left: \( P \) is \( \mathcal{H} \Rightarrow (\Pi \sqcup P) = P \)
and Unit-Right: \( P \) is \( \mathcal{H} \Rightarrow (P \sqcup \Pi) = P \)
begin

  lemma Unit-self [simp]:
  \( \Pi \sqcup \Pi = \Pi \)
  by (simp add: Healthy-Unit Unit-Right)

  lemma utest-intro:
  \( \Pi \sqsubseteq P \Rightarrow \text{utest } T P \)
  by (simp add: utest-def)

  lemma utest-Unit [closure]:
  \( \text{utest } T \, \Pi \)
  by (simp add: utest-def)

  end

sublocale utp-theory-unital \( \subseteq \) utp-theory-left-unital
  by (simp add: Healthy-Unit Unit-Left Healthy-Sequence utp-theory-rel-def utp-theory-axioms utp-theory-rel-axioms-def utp-theory-left-unital-axioms-def)

sublocale utp-theory-unital \( \subseteq \) utp-theory-right-unital
  by (simp add: Healthy-Unit Unit-Right Healthy-Sequence utp-theory-rel-def utp-theory-axioms utp-theory-rel-axioms-def utp-theory-right-unital-axioms-def)

locale utp-theory-mono-unital = utp-theory-mono + utp-theory-unital
begin

  lemma utest-Top [closure]:
  \( \text{utest } T \, T \)
  by (simp add: Healthy-Unit utest-def utp-top)

  end

locale utp-theory-cont-unital = utp-theory-cont-rel + utp-theory-unital

sublocale utp-theory-cont-unital \( \subseteq \) 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 +
assumes Top-Left-Zero: \( P \) is \( \mathcal{H} \Rightarrow T \sqcup P = T \)

locale utp-theory-cont-unital-zerol =
  utp-theory-cont-unital + utp-theory-unital-zerol
begin
lemma Top-test-Right-Zero:
  assumes b is H utest T b
  shows b ;; ⊤ = ⊤
proof –
  have b ⊓ II = II
    by (meson assms(2) semilattice-sup-class.le-iff-sup utest-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

We can exemplify the creation of a UTP theory with the theory of relations, a trivial theory.

typedec REL
abbreviation REL ≡ UTHY(REL, ′α)

We declare the type REL to be the tag for this theory. We need know nothing about this type (other than it’s non-empty), since it is merely a name. We also create the corresponding constant to refer to the theory. Then we can use it to instantiate the relevant polymorphic constants.

overloading
  rel-hcond == utp-hcond :: (REL, ′α) uthy ⇒ (′α × ′α) health
  rel-unit == utp-unit :: (REL, ′α) uthy ⇒ ′α hrel

begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition rel-hcond :: (REL, ′α) uthy ⇒ (′α × ′α) upred ⇒ (′α × ′α) upred where
  [upred-defs]: rel-hcond T = id

The unit of the theory is simply the relational unit.

definition rel-unit :: (REL, ′α) uthy ⇒ ′α hrel where
  [upred-defs]: rel-unit T = II

end

Finally we can show that relations are a monotone and unital theory using a locale interpretation, which requires that we prove all the relevant properties. It’s convenient to rewrite some of the theorems so that the provisos are more UTP like; e.g. that the carrier is the set of healthy predicates.

interpretation rel-theory: utp-theory-mono-unital REL
  rewrites carrier (uthy-order REL) = [id]_H
    by (unfold-locales, simp-all add: rel-hcond-def rel-unit-def Healthy-def)

We can then, for instance, determine what the top and bottom of our new theory is.

lemma REL-top: ⊤ _REL = false
  by (simp add: rel-theory.healthy-top, simp add: rel-hcond-def)

lemma REL-bottom: ⊥ _REL = true
  by (simp add: rel-theory.healthy-bottom, simp add: rel-hcond-def)
A number of theorems have been exported, such as the fixed point unfolding laws.

**thm** rel-theory.GFP-unfold

### 19.5 Theory links

We can also describe links between theories, such as Galois connections and retractions, using the following notation.

**definition** mk-conn (- ≜⟨-,-⟩⇒ -90,0,0,91) where

H1 ≜⟨H1,H2⟩⇒ H2 ≜ [0 | orderA = utp-order H1, orderB = utp-order H2, lower = H2, upper = H1 ]

**abbreviation** mk-conn' (- ≜⟨-,-⟩→ -90,0,0,91) where

T1 ≜⟨H1,H2⟩→ T2 ≜ H T1 ≜⟨H1,H2⟩⇒ H T2

**lemma** mk-conn-orderA [simp]: X H1 ≜⟨H1,H2⟩⇒ H2 = utp-order H1
  by (simp add: mk-conn-def)

**lemma** mk-conn-orderB [simp]: Y H1 ≜⟨H1,H2⟩⇒ H2 = utp-order H2
  by (simp add: mk-conn-def)

**lemma** mk-conn-lower [simp]: π ∗ H1 ≜⟨H1,H2⟩⇒ H2 = H1
  by (simp add: mk-conn-def)

**lemma** mk-conn-upper [simp]: π ∗ H1 ≜⟨H1,H2⟩⇒ H2 = H2
  by (simp add: mk-conn-def)

**lemma** galois-comp: (H2 ≜⟨H3,H4⟩⇒ H3) ◦ g (H1 ≜⟨H1,H2⟩⇒ H2) = H1 ≜⟨H1 ◦ H3,H4 ◦ H2⟩⇒ H3
  by (simp add: comp-galcon-def mk-conn-def)

Example Galois connection / retract: Existential quantification

**lemma** Idempotent-ex: mwb-lens x =⇒ Idempotent (ex x)
  by (simp add: Idempotent-def exists-twice)

**lemma** Monotonic-ex: mwb-lens x =⇒ Monotonic (ex x)
  by (simp add: mono-def ex-mono)

**lemma** ex-closed-unrest: vwb-lens x =⇒ [ex x]H = { P. x ∉ P }
  by (simp add: Healthy-def unrest-as-exists)

Any theory can be composed with an existential quantification to produce a Galois connection

**theorem** ex-retract:
  assumes vwb-lens x Idempotent H ex x o H = H o ex x
  shows retract ((ex x o H) ≜⟨ex x, H⟩⇒ H)
  proof (unfold-locales, simp-all)
    show H ∈ [ex x o H]H ⇒ [H]H
      using Healthy-Idempotent assms by blast
    from assms(1) assms(3) THEN sym show ex x ∈ [H]H ⇒ [ex x o H]H
      by (simp add: Pi-iff Healthy-def fun-eq-if exists-twice)
    fix P Q
    assume P is (ex x o H) Q is H
    thus (H P ⊆ Q) = (P ⊆ (∃ x · Q))
      by (metis (no-types, lifting) Healthy-Idempotent Healthy-if assms comp-apply dual-order.trans ex-weakens utp-pred-laws.ex-mono vwb-lens-ub)
next
fix \( P \)
assume \( P \) is \((\exists \ x \cdot H)\)
thus \((\exists \ x \cdot H \ P) \subseteq P\)
by \((\text{simp add: Healthy-def})\)
qed

corollary \textit{ex-retract-id}:
assumes \( \text{vwb-lens} \ x \)
shows \( \text{retract} \ (ex \ x \Leftarrow (ex \ x, \text{id}) \Rightarrow \text{id}) \)
using \( \text{assms \ ex-retract[where \ H=\text{id}]} \) by \((\text{auto})\)
end

20 Relational Hoare calculus

theory \textit{utp-hoare}
imports
  \textit{utp-rel-laws}
  \textit{utp-theory}
begin

20.1 Hoare Triple Definitions and Tactics

definition \textit{hoare-r} :: \( 'a \) \, cond \Rightarrow \( 'a \) \, hrel \Rightarrow \( 'a \) \, cond \Rightarrow bool \) \( (\{-\}/ -/ \{-\}_u) \) \textit{where}
\( \{p\}Q\{r\}_u = ((\{p\}_< \Rightarrow [r]_>) \subseteq Q) \)

declare \textit{hoare-r-def [upred-defs]}

named-theorems \textit{hoare} and \textit{hoare-safe}

method \textit{hoare-split} \textbf{uses} \( hr = \)
\((\text{intro add: assigns-r-comp usubst unrest})? , \) — Eliminate assignments where possible
\( (\text{auto}) \)
\( \text{intro: hoare intro! : hoare-safe hr} \)
\( \text{simp add: assigns-r-comp conj-comm conj-assoc usubst unrest})[1] \) — Apply Hoare logic laws

method \textit{hoare-auto} \textbf{uses} \( hr = (\text{hoare-split hr; hr; rel-auto}?) \)

20.2 Basic Laws

lemma \textit{hoare-r-conj} \textbf{[hoare-safe]}: \( \{p\}Q\{r\}_u; \{p\}Q\{s\}_u \) \( \Rightarrow \{p\}Q\{r \land s\}_u \)
by \( \text{rel-auto} \)

lemma \textit{hoare-r-weaken-pre} \textbf{[hoare]}:
\( \{p\}Q\{r\}_u \Rightarrow \{p \land q\}Q\{r\}_u \)
\( \{q\}Q\{r\}_u \Rightarrow \{p \land q\}Q\{r\}_u \)
by \( \text{rel-auto+} \)

lemma \textit{pre-str-hoare-r}:
assumes \( 'p_1 \Rightarrow p_2' \) \textbf{and} \( \{p_2\}C\{q\}_u \)
shows \( \{p_1\}C\{q\}_u \)
using \( \text{assms \ by \ rel-auto} \)

lemma \textit{post-weak-hoare-r}:
assumes \( \{p\}C\{q_2\}_u \) \textbf{and} \( 'q_2 \Rightarrow q_1' \)
shows $\{p\}C\{q_1\}_u$
using assms by rel-auto

lemma hoare-r-conseq: \[ p_1 \Rightarrow p_2 ; \{p_2\}S\{q_2\}_u ; \ q_2 \Rightarrow q_1 \ \] \[ \Rightarrow \{p_1\}S\{q_1\}_u \]
by rel-auto

20.3 Assignment Laws

lemma assigns-hoare-r [hoare-safe]: \[ p \Rightarrow \sigma \vdash q \] \[ \Rightarrow \{p\} (\sigma)_u \{q\}_u \]
by rel-auto

lemma assigns-backward-hoare-r:
\[ \{\sigma \vdash p\} (\sigma)_u \{p\}_u \]
by rel-auto

lemma assign-floyd-hoare-r:
assumes wb-lens $x$
shows $\{p\} \assign-r x e \Rightarrow \exists v \cdot p[\langle v/x\rangle] \land \delta x = u \cdot \epsilon[\langle v/x\rangle]_u$
using assms
by (rel-auto, metis wb-lens-wb wb-lens.get-put)

lemma skip-hoare-r [hoare-safe]: $\{p\} H \{p\}_u$
by rel-auto

lemma skip-hoare-impl-r [hoare-safe]: \[ p \Rightarrow q \] \[ \Rightarrow \{p\} H \{q\}_u \]
by rel-auto

20.4 Sequence Laws

lemma seq-hoare-r: \[ \{p\} Q_1 \{s\}_u ; \{s\} Q_2 \{r\}_u \] \[ \Rightarrow \{p\} Q_1 ; Q_2 \{r\}_u \]
by rel-auto

lemma seq-hoare-invariant [hoare-safe]: \[ \{p\} Q_1 \{p\}_u ; \{p\} Q_2 \{p\}_u \] \[ \Rightarrow \{p\} Q_1 ; Q_2 \{p\}_u \]
by rel-auto

lemma seq-hoare-stronger-pre-1 [hoare-safe]: \[ \{p \land q\} Q_1 \{p \land q\}_u ; \{p \land q\} Q_2 \{q\}_u \] \[ \Rightarrow \{p \land q\} Q_1 ; Q_2 \{q\}_u \]
by rel-auto

lemma seq-hoare-stronger-pre-2 [hoare-safe]: \[ \{p \land q\} Q_1 \{p \land q\}_u ; \{p \land q\} Q_2 \{p\}_u \] \[ \Rightarrow \{p \land q\} Q_1 ; Q_2 \{p\}_u \]
by rel-auto

lemma seq-hoare-inv-r-2 [hoare]: \[ \{p\} Q_1 \{q\}_u ; \{q\} Q_2 \{q\}_u \] \[ \Rightarrow \{p\} Q_1 ; Q_2 \{q\}_u \]
by rel-auto

lemma seq-hoare-inv-r-3 [hoare]: \[ \{p\} Q_1 \{p\}_u ; \{p\} Q_2 \{q\}_u \] \[ \Rightarrow \{p\} Q_1 ; Q_2 \{q\}_u \]
by rel-auto

20.5 Conditional Laws

lemma cond-hoare-r [hoare-safe]: \[ \{b \land p\} S \{q\}_u ; \{\neg b \land p\} T \{q\}_u \] \[ \Rightarrow \{p\} S \land b \lor T \{q\}_u \]
by rel-auto

lemma cond-hoare-r-up:
assumes $\{p\} S \{q\}_u$ and $\{p''\} T \{q\}_u$
shows \(\{b \land p'\} \lor (\neg b \land p'')\} S < b \triangleright_r T \{q\}_u\)
using assms by pred-simp

lemma cond-hoare-r-sp:
assumes \(\{b \land p\} S \{q\}_u\) and \(\{\neg b \land p\} T \{s\}_u\)
shows \(\{p\} S < b \triangleright_r T \{q \lor s\}_u\)
using assms by pred-simp

20.6 Recursion Laws

lemma nu-hoare-r-partial:
assumes induct-step: \(\land_{st} P. \{p\} P \{q\}_u \implies \{p\} F P \{q\}_u\)
shows \(\{p\} \nu F \{q\}_u\)
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 ((\[p\] < \(\{e\} <, \{\langle st\rangle\}_u \in_u \langle R\rangle \implies \[q\] >) st)]
by (simp add: alpha)
qed

lemma mu-hoare-r:\nassumes WF: wf R
assumes M: mono F
assumes induct-step: \(\land_{st} P. \{p\} \land (e, \langle st\rangle)_u \in_u \langle R\rangle \implies \{p\} e =_u \langle st\rangle \implies \{p\} P \{q\}_u\)
shows \(\{p\} \mu F \{q\}_u\)
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 ((\[p\] < \(\{e\} <, \{\langle st\rangle\}_u \in_u \langle R\rangle \implies \[q\] >) st)]
by (simp add: alpha)
qed

lemma mu-hoare-r:\nassumes WF: wf R
assumes M: mono F
assumes induct-step: \(\land_{st} P. \{p\} \land (e, \langle st\rangle)_u \in_u \langle R\rangle \implies \{p\} e =_u \langle st\rangle \implies \{p\} P \{q\}_u\)
shows \(\{p\} \mu F \{q\}_u\)
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 ((\[p\] < \(\{e\} <, \{\langle st\rangle\}_u \in_u \langle R\rangle \implies \[q\] >) st)]
by (simp add: alpha)
qed

20.7 Iteration Rules

lemma while-hoare-r [hoare-safe]:
assumes \(\{p \land b\} S \{p\}_u\)
shows \(\{p\} while b do S od \{\neg b \land p\}_u\)
using assms
by (simp add: while-def hoare-r-def, rule_tac lfp-lowerbound) (rel-auto)

lemma while-inv-hoare-r [hoare-safe]:
assumes \(\{p \land b\} S \{p\}_u\) 'pre \rightarrow p' '(!b \land p) \rightarrow post'
shows \(\{pre\} while b invr p do S od \{post\}_u\)
by (metis assms hoare-r-conseq while-hoare-r while-inv-def)

lemma while-r-minimal-partial:
assumes seq-step: 'p \Rightarrow invvar'
assumes induct-step: \(\invar \land b\} C \{\invar\}_u\)
shows \(\{p\} while b do C od \{!b \land invvar\}_u\)
using induct-step pre-str-hoare-r seq-step while-hoare-r by blast

lemma approx-chain:
\((\prod n::nat. [p \land v <_u \prec \prec n \succ]_<) = [p]_<\)
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 \(\prod z::nat. \{ p \land b \land v =_u \prec \prec z \succ \}\}_{u} S \{ p \land v <_u \prec \prec z \succ \}_{u}
shows \(\{ p \}\) while \(\bot b\) do \(S\) od \(\{ \sim b \land p \}\)\(u\)
proof (rule mu-refine-intro)
from assms show \(\{ p \}\Rightarrow \{ \sim b \land p \}\) \(\subseteq\) \(S\) \(\{ p \land v <_u \prec \prec z \succ \}_{u}\)\(u\)
by (rel-auto)

let \(?E = \lambda n. [p \land v <_u \prec \prec n \succ]_<\)
show \(\{ p \}_< \land (\mu X \cdot S ;; X \prec b \succ r \ II)) = (\{ p \}_< \land (\nu X \cdot S ;; X \prec b \succ r \ II))\)
proof (rule constr-fp-uniq[where \(E=\?E\)])
show \(\prod n. ?E(n) = [p]_<\)
by (rel-auto)

show mono \((\lambda X. S ;; X \prec b \succ r \ II)\)
by (simp add: cond-mono monoI seqr-mono)

show constr \((\lambda X. S ;; X \prec b \succ r \ II) \ ?E\)
proof (rule constrI)

show chain \(?E\)
proof (rule chainI)
show \([p \land v <_u \prec \prec 0 \succ]_< = false\)
by (rel-auto)
show \(\prod i. [p \land v <_u \prec \prec Suc i \succ]_< \subseteq [p \land v <_u \prec \prec i \succ]_<\)
by (rel-auto)
qed

from assms
show \(\prod X n. (S ;; X \prec b \succ r \ II \land [p \land v <_u \prec \prec n + 1 \succ]_<) = (S ;; (X \land [p \land v <_u \prec \prec n \succ]_<) \prec b \succ r \ II \land [p \land v <_u \prec \prec n + 1 \succ]_<)\)
apply (rel-auto)
using less-antisym less-trans apply blast
done
qed
qed

thus \(?thesis\)
by (simp add: hoare-r-def while-bot-def)
qed

lemma while-vrt-hoare-r [hoare-safe]:
assumes \(\prod z::nat. \{ p \land b \land v =_u \prec \prec z \succ \}\}_{u} S \{ p \land v <_u \prec \prec z \succ \}_{u} \ 'pre \Rightarrow p' \ '\sim b \land p \Rightarrow post'\)

\textbf{141}
General total correctness law based on well-founded induction

**Lemma** while-wf-hoare-r:

- Assumes `WF`: `wf R`
- Assumes `I0`: `'pre ⇒ p'`
- Assumes `induct-step`: `∀ st. {b ∧ p ∧ e = u <st>} Q{p ∧ (e, <st>) u ∈ u <R>} u`
- Assumes `PHI`: `(¬b ∧ p) ⇒ post`
- Shows `∀ st. {b ∧ p ∧ e = u <st>} Q {p ∧ (e, <st>) u ∈ u <R>} u`

**Proof** (rule `pre-weak-rel[of - [p]<]`)

From `I0` show `'pre< ⇒ [p]<'`

by `rel-auto`

Show `∀ st. {b <post>} (μ X ∗ Q ;; X < b ∨r II)`

Proof (rule `mu-rec-total-atp-rule[where e=e, OF WF]`)

Show `Monotonic (λX. Q ;; X < b ∨r II)`

by (simp add: closure)

Have `induct-step': `∀ st. ([b ∧ p ∧ e = u <st>] < ⇒ ([p ∧ (e, <st>) u ∈ u <R>] < ) ∈ Q`

using `induct-step` by `rel-auto`

with `PHI`

Show `∀ st. ([p]< ∧ [e]< = u <st> ⇒ [post]< ) ∈ Q ;; ([p]< ∧ ([e]<, <st>) u ∈ u <R> ⇒ [post]< ) `< b ∨r II`

by (rel-auto)

qed

20.8 Frame Rules

Frame rule: If starting `S` in a state satisfying `pestablishesq` in the final state, then we can insert an invariant predicate `p` 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 `vwb-lens a a ⊥ r a ≈ q {p} P {q} u`
- Shows `{p ∧ r} a:[P] {q ∧ r} u`

using `assms`

by (`rel-auto`, `metis`)

**Lemma** frame-strong-hoare-r [hoare-safe]:

- Assumes `vwb-lens a a ⊥ r a ≈ q {p ∧ r} S {q} u`
- Shows `{p ∧ r} a:[S] {q ∧ r} u`

using `assms by` (`rel-auto`, `metis`)

**Lemma** frame-hoare-r' [hoare-safe]:

- Assumes `vwb-lens a a ⊥ r a ≈ q {r ∧ p} S {q} u`
- Shows `{r ∧ p} a:[S] {r ∧ q} u`

using `assms by` (`simp add: `frame-strong-hoare-r utp-pred-laws.inf.commute`)

**Lemma** antiframe-hoare-r:

- Assumes `vwb-lens a a ⊥ r a ≈ q {p} P {q} u`
- Shows `{p ∧ r} a:[P] {q ∧ r} u`
using assms by (rel-auto, metis)

lemma antiframe-strong-hoare-r:
  assumes vwb-lens a a ⊨ r a ⊨ q {p ∧ r} P{q} u
  shows {p ∧ r} a: [P] {q ∧ r} u
  using assms by (rel-auto, metis)

lemma antiframe-intro:
  assumes vwb-lens g vwb-lens g' vwb-lens l l ⊲ ⊳ g g' ⊆ L g
  {&g', &l};[C] = C {p} C{q} u 'r ⇒ p'
  shows {r} l;[C] {(∃ l · q) ∧ (∃ g' · r)} u
  using assms
  apply (rel-auto, simp-all add: lens-defs)
  apply metis
  apply (rename-tac Z a b)
  apply (rule-tac x = get g' a in exI)
oops

end

21 Weakest Precondition Calculus

theory utp-wp import utp-hoare begin
A very quick implementation of wp – more laws still needed!
named-theorems wp

method wp-tac = (simp add: wp)

consts
  uwp :: 'a ⇒ 'b ⇒ 'c (infix wp 60)

definition wp-upred :: ('α, 'β) urel ⇒ 'β cond ⇒ 'α cond where
  wp-upred Q r = [¬ (Q ;; (¬ [r]<_)) :: ('α, 'β) 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]:
  ⟨σ⟩a wp r = σ ⊢ r
  by rel-auto

theorem wp-skip-r [wp]:
  II wp r = r
by rel-auto

**Theorem wp-abort [wp]:**
\[ r \neq \text{true} \implies \text{true wp } r = \text{false} \]
by rel-auto

**Theorem wp-conj [wp]:**
\[ P \text{ wp } (q \land r) = (P \text{ wp } q \land P \text{ wp } r) \]
by rel-auto

**Theorem wp-seq-r [wp]:**
\[ (P ;; Q) \text{ wp } r = P \text{ wp } (Q \text{ wp } r) \]
by rel-auto

**Theorem wp-cond [wp]:**
\[ (P \triangleleft b \triangleright r, Q) \text{ wp } r = ((b \implies P \text{ wp } r) \land ((\neg b) \implies Q \text{ wp } r)) \]
by rel-auto

**Lemma wp-USUP-pre [wp]:**
\[ P \text{ wp } (\bigsqcup_{i \in \{0..n\}} Q(i)) = (\bigsqcup_{i \in \{0..n\}} P \text{ wp } Q(i)) \]
by (rel-auto)

**Theorem wp-hoare-link:**
\[ \{p\} Q \{r\}_a \longleftrightarrow (Q \text{ wp } r \subseteq p) \]
by rel-auto

If two programs have the same weakest precondition for any postcondition then the programs are the same.

**Theorem wp-eq-intro:**
\[ \{ \bigwedge_{r. \text{ P wp } r = Q \text{ wp } r} \} \implies P = Q \]
by (rel-auto robust, fastforce+)

end

## 22 Strong Postcondition Calculus

**Theory utp-sp**

**Imports utp-wp**

begin

**Named-Theorems sp**

**Method sp-tac** = (simp add: sp)

**Consts**

usp :: 'a \Rightarrow 'b \Rightarrow 'c (infix sp 60)

definition sp-upred :: 'a cond \Rightarrow ('a, 'b) urel \Rightarrow 'b cond where
sp-upred p Q = [([p]> ;; Q) :: ('a, 'b) urel]

**Adhoc-Overloading**

usp sp-upred

**Declare sp-upred-def [upred-defs]**

**Lemma sp-false [sp]:**
\[ p \text{ sp } false = false \]
by (rel-simp)

**Lemma sp-true [sp]:**
\[ q \neq false \implies q \text{ sp } true = true \]
by (rel-auto)
lemma *sp-assigns-r* [sp]:
\[
\forall vw-b-lens \; x \quad \Rightarrow \quad (p \; sp \; x \; := \; e) = (\exists \; v \; \cdot \; p[\langle v \rangle/x] \; \land \; &x = u \; e[\langle v \rangle/x])
\]
by (rel-auto, metis vw-b-lens-wb wb-lens.get-put, metis vw-b-lens.put-eq)

lemma *sp-it-is-post-condition*:
\[
\{p\} C \{p \; sp \; C\}_u
\]
by rel-blast

lemma *sp-it-is-the-strongest-post*:
\[
\forall p \; sp \; C \Rightarrow Q' \quad \Rightarrow \quad \{p\} C \{Q\}_u
\]
by rel-blast

lemma *sp-so*:
\[
\forall p \; sp \; C \Rightarrow Q' = \{p\} C \{Q\}_u
\]
by rel-blast

theorem *sp-hoare-link*:
\[
\{p\} Q \{r\}_u \longleftrightarrow (r \subseteq p \; sp \; Q)
\]
by rel-auto

lemma *sp-while-r* [sp]:
\[
\forall sp \; invar \; I \; while \bot \; b \; do \; C \; od = (-b \; \land \; I)
\]
unfolding sp-upred-def

oops

theorem *sp-eq-intro*:
\[
\forall r. \; r \; sp \; P = r \; sp \; Q \quad \Rightarrow \quad P = Q
\]
by (rel-auto robust, fastforce+)

lemma *wp-sp-sym*:
\[
\forall prog \; wp \; (true \; sp \; prog)
\]
by rel-auto

lemma *it-is-pre-condition*:
\[
\{C \; wp \; Q\} C \{Q\}_u
\]
by rel-blast

lemma *it-is-the-weakest-pre*:
\[
\forall P \Rightarrow C \; wp \; Q' = \{P\} C \{Q\}_u
\]
by rel-blast

lemma *s-pre*:
\[
\forall P \Rightarrow C \; wp \; Q' = \{P\} C \{Q\}_u
\]
by rel-blast

end

23 Concurrent Programming

theory *utp-concurrency*

imports
\[
\begin{align*}
\quad & \text{utp-hoare} \\
\quad & \text{utp-rel} \\
\quad & \text{utp-tactics} \\
\quad & \text{utp-theory}
\end{align*}
\]

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 [14].

23.1 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.

\[
\begin{align*}
\text{alphabet} \quad (\alpha', \beta_0', \beta_1') & \xrightarrow{\text{mrg}} \\
\text{mrg-prior} & \xrightarrow{\alpha} \\
\text{mrg-left} & \xrightarrow{\beta_0} \\
\text{mrg-right} & \xrightarrow{\beta_1}
\end{align*}
\]

**Definition**

**pre-uvar ::** \((\alpha \Rightarrow (\alpha', \beta_0, \beta_1') \xrightarrow{\text{mrg}})\) where 
\([\text{upred-defs}]: \text{pre-uvar } x = x\); 
\([\text{upred-defs}]: \text{left-uvar } x = x\); 
\([\text{upred-defs}]: \text{right-uvar } x = x\);

**left-uvar ::** \((\alpha' \Rightarrow \beta_0) \Rightarrow (\alpha \Rightarrow (\alpha', \beta_0, \beta_1') \xrightarrow{\text{mrg}})\) where 
\([\text{upred-defs}]: \text{left-uvar } x = x \; \vdash_L \; \text{mrg-left}\);

**right-uvar ::** \((\alpha' \Rightarrow \beta_1) \Rightarrow (\alpha \Rightarrow (\alpha', \beta_0, \beta_1') \xrightarrow{\text{mrg}})\) where 
\([\text{upred-defs}]: \text{right-uvar } x = x \; \vdash_L \; \text{mrg-right}\);

We set up syntax for the three variable classes using a subscript \(<1\), \(0-x\), and \(1-x\), respectively.

**Syntax**

- **svarpre ::** svid \( \Rightarrow \) svid \((\sim_< \{995\} 995)\)
- **svarleft ::** svid \( \Rightarrow \) svid \((0-\sim \{995\} 995)\)
- **svarright ::** svid \( \Rightarrow \) svid \((1-\sim \{995\} 995)\)

**Translations**

- **svarpre x ==** \text{CONST} \text{pre-uvar} x
- **svarleft x ==** \text{CONST} \text{left-uvar} x
- **svarright x ==** \text{CONST} \text{right-uvar} x
- **svarpre \Sigma ==** \text{CONST} \text{pre-uvar} 1_L
- **svarleft \Sigma ==** \text{CONST} \text{left-uvar} 1_L
- **svarright \Sigma ==** \text{CONST} \text{right-uvar} 1_L

We proved behavedness closure properties about the lenses.

**Lemma**

- **left-uvar [simp]:** \text{vwb-lens} x \( \Rightarrow \) \text{vwb-lens} (left-uvar x) 
  \text{by (simp add: left-uvar-def)}

- **right-uvar [simp]:** \text{vwb-lens} x \( \Rightarrow \) \text{vwb-lens} (right-uvar x) 
  \text{by (simp add: right-uvar-def)}

- **pre-uvar [simp]:** \text{vwb-lens} x \( \Rightarrow \) \text{vwb-lens} (pre-uvar x) 
  \text{by (simp add: pre-uvar-def)}
lemma left-uvar-mwb [simp]: mwb-lens x ⇒ mwb-lens (left-uvar x) 
  by (simp add: left-uvar-def )

lemma right-uvar-mwb [simp]: mwb-lens x ⇒ mwb-lens (right-uvar x) 
  by (simp add: right-uvar-def)

lemma pre-uvar-mwb [simp]: mwb-lens x ⇒ mwb-lens (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]:
  left-uvar x ⊲ ⊳ right-uvar y 
  by (simp add: left-uvar-def right-uvar-def lens-comp-assoc[THEN sym])

lemma left-uvar-indep-pre-uvar [simp]:
  left-uvar x ⊲ ⊳ pre-uvar y 
  by (simp add: left-uvar-def pre-uvar-def)

lemma left-uvar-indep-left-uvar [simp]:
  x ⊲ y ⇒ left-uvar x ⊲ left-uvar y 
  by (simp add: left-uvar-def)

lemma right-uvar-indep-left-uvar [simp]:
  right-uvar x ⊲ left-uvar y 
  by (simp add: lens-indep-sym)

lemma right-uvar-indep-pre-uvar [simp]:
  right-uvar x ⊲ pre-uvar y 
  by (simp add: right-uvar-def pre-uvar-def)

lemma right-uvar-indep-right-uvar [simp]:
  x ⊲ y ⇒ right-uvar x ⊲ right-uvar y 
  by (simp add: right-uvar-def)

lemma pre-uvar-indep-left-uvar [simp]:
  pre-uvar x ⊲ left-uvar y 
  by (simp add: lens-indep-sym)

lemma pre-uvar-indep-right-uvar [simp]:
  pre-uvar x ⊲ right-uvar y 
  by (simp add: lens-indep-sym)

lemma pre-uvar-indep-pre-uvar [simp]:
  x ⊲ y ⇒ pre-uvar x ⊲ pre-uvar y 
  by (simp add: pre-uvar-def)

23.2 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 'α merge = (('α, 'α, 'α) mrg, 'α) urel

skip is the merge predicate which ignores the output of both parallel predicates

definition skip_m :: 'α merge where
swap is a predicate that the swaps the left and right indices; it is used to specify commutativity of the parallel operator

definition swap_m :: (=('a, 'b, 'b) mrg) hrel where
[upred-defs]: swap_m = (0-\nu,1-\nu) := (1-\nu,\&0-\nu)

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 \( swap_m \) is a left-unit.

abbreviation SymMerge :: 'a merge \Rightarrow 'a merge where
SymMerge(M) \equiv \( swap_m ;; M \)

23.3 Separating Simulations

U0 and U1 are relations modify the variables of the input state-space such that they become indexed with 0 and 1, respectively.

definition U0 :: ('b_0, ('a, 'b_0, 'b_0) mrg) urel where
[upred-defs]: U0 = ($0-\nu' = u$\nu)

definition U1 :: ('b_1, ('a, 'b_0, 'b_1) mrg) urel where
[upred-defs]: U1 = ($1-\nu' = u$\nu)

lemma U0-swap: (U0 ;; swap_m) = U1
by (rel-auto)

lemma U1-swap: (U1 ;; swap_m) = U0
by (rel-auto)

As shown below, separating simulations can also be expressed using the following two alphabet extrusions
definition U0\alpha where [upred-defs]: U0\alpha = (1_L \times L mrg-left)
definition U1\alpha where [upred-defs]: U1\alpha = (1_L \times L mrg-right)

We then create the following intuitive syntax for separating simulations.

abbreviation U0-alpha-lift ([\-]_0) where \([P]_0 \equiv P \oplus_p U0\alpha\)

abbreviation U1-alpha-lift ([\-]_1) where \([P]_1 \equiv P \oplus_p U1\alpha\)

\([P]_0\) 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 U0alpha-vwb-lens [simp]: vwb-lens U0\alpha
by (simp add: U0alpha-def id-vwb-lens prod-vwb-lens)

lemma U1alpha-vwb-lens [simp]: vwb-lens U1\alpha
by (simp add: U1alpha-def id-vwb-lens prod-vwb-lens)
lemma $U0\alpha$-indep-right-uvar [simp]: \( vwb\text{-}lens \ x \Rightarrow U0\alpha \bowtie \text{out\text{-}var} \ (\text{right\text{-}uvar} \ x) \)
by (force intro: plus-pres-lens-indep fst-snd-lens-indep lens-indep-left-comp
simp add: $U0\alpha$-def right-uvar-def out-var-def prod-as-plus lens-comp-assoc[THEN sym])

lemma $U1\alpha$-indep-left-uvar [simp]: \( vwb\text{-}lens \ x \Rightarrow U1\alpha \bowtie \text{out\text{-}var} \ (\text{left\text{-}uvar} \ x) \)
by (force intro: plus-pres-lens-indep fst-snd-lens-indep lens-indep-left-comp
simp add: $U1\alpha$-def left-uvar-def out-var-def prod-as-plus lens-comp-assoc[THEN sym])

lemma $U0$-alpha-lift-bool-subst [usubst]:
\[
\sigma(0\rightarrow x'\mapsto s\text{true}) \upharpoonright \llbracket P \rrbracket \sigma(0\rightarrow x'\mapsto s\text{false}) \upharpoonright \llbracket P \rrbracket
\]
by (pred-auto+)

lemma $U1$-alpha-lift-bool-subst [usubst]:
\[
\sigma(1\rightarrow x'\mapsto s\text{true}) \upharpoonright \llbracket P \rrbracket \sigma(1\rightarrow x'\mapsto s\text{false}) \upharpoonright \llbracket P \rrbracket
\]
by (pred-auto+)

lemma $U0$-alpha-out-var [alpha]: \( \upharpoonright \llbracket x' \rrbracket \sigma - x' \)
by (rel-auto)

lemma $U1$-alpha-out-var [alpha]: \( \upharpoonright \llbracket x' \rrbracket = 1 - x' \)
by (rel-auto)

lemma $U0$-skip [alpha]: \( \llbracket II \rrbracket = (0\rightarrow v' = u \rightarrow v) \)
by (rel-auto)

lemma $U1$-skip [alpha]: \( \llbracket II \rrbracket = (1\rightarrow v' = u \rightarrow v) \)
by (rel-auto)

lemma $U0$-seqr [alpha]: \( P ; Q = P ; \llbracket Q \rrbracket \)
by (rel-auto)

lemma $U1$-seqr [alpha]: \( P ; Q = P ; \llbracket Q \rrbracket \)
by (rel-auto)

lemma $U0\alpha$-comp-in-var [alpha]: \( \text{in\text{-}var} \ x \bowtie L U0\alpha = \text{in\text{-}var} \ x \)
by (simp add: $U0\alpha$-def alpha-in-var in-var-prod-lens pre-uvar-def)

lemma $U0\alpha$-comp-out-var [alpha]: \( \text{out\text{-}var} \ x \bowtie L U0\alpha = \text{out\text{-}var} \ (\text{left\text{-}uvar} \ x) \)
by (simp add: $U0\alpha$-def alpha-out-var id-wb-lens left-uvar-def out-var-prod-lens)

lemma $U1\alpha$-comp-in-var [alpha]: \( \text{in\text{-}var} \ x \bowtie L U1\alpha = \text{in\text{-}var} \ x \)
by (simp add: $U1\alpha$-def alpha-in-var in-var-prod-lens pre-uvar-def)

lemma $U1\alpha$-comp-out-var [alpha]: \( \text{out\text{-}var} \ x \bowtie L U1\alpha = \text{out\text{-}var} \ (\text{right\text{-}uvar} \ x) \)
by (simp add: $U1\alpha$-def alpha-out-var id-wb-lens right-uvar-def out-var-prod-lens)

23.4 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 :: 'a merge ⇒ (('α, 'α, 'α, 'α) mrg) mrg, 'α) urel (M⇧(\cdot)) where

[upred-defs]: ThreeWayMerge M = ((\$0 - \$v\' = _u \$0 - \$v \& \$1 - \$v\' = _u \$1 - \$v \& \$v\'_\< = _u \$v\_<) ;; M ::
\$0 \& \$1 - \$v\' = _u \$1 - \$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 ≡ (\$0 - \$v, \$1 - \$v, \$1 - \$v) := (\$1 - \$v, \$0 - \$v, \$0 - \$v)

Finally, a merge is associative if rotating the inputs does not effect the output.

**Definition**: AssocMerge :: 'a merge ⇒ bool where

[upred-defs]: AssocMerge M = (rotate_m ;; M⇧(M) = M⇧(M))

### 23.5 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 ||s 85) where

\( P ||s Q \equiv (P ;; U0) \& (Q ;; U1) \& \$v\_<\' = _u \$v \)

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 :: ('a, 'β) urel ⇒ (('α, 'β, 'γ) mrg, 'δ) urel ⇒ ('α, 'γ) urel ⇒ ('α, 'δ) urel

\( (\cdot ||s \cdot [85,0,86]) 85 \)

**Where**: [upred-defs]: \( P ||_M Q = (P ||s Q ;; M) \)

**Lemma**: par-by-merge-alt-def: \( P ||_M Q = ([P]_0 \& [Q]_1 \& \$v\_<\' = _u \$v) ;; M \)

by (simp add: par-by-merge-def U0-as-alpha U1-as-alpha)

**Lemma**: shEx-pbm-left: \( ((\exists \ x \cdot P \ x) ||_M Q) = (\exists \ x \cdot (P \ x ||_M Q)) \)

by (rel-auto)

**Lemma**: shEx-pbm-right: \( (P ||_M (\exists \ x \cdot Q \ x)) = (\exists \ x \cdot (P ||_M Q \ x)) \)

by (rel-auto)

### 23.6 Unrestriction Laws

**Lemma**: unrest-in-par-by-merge [unrest]:

\[
\begin{align*}
\& \$x \not\in P; \$x\_< \not\in M; \$x \not\in Q \\
\Rightarrow \$x \not\in P ||_M Q
\end{align*}
\]

by (rel-auto, fastforce+)

**Lemma**: unrest-out-par-by-merge [unrest]:

\[
\begin{align*}
\& \$x\' \not\in M \\
\Rightarrow \$x\' \not\in P ||_M Q
\end{align*}
\]

by (rel-auto)

### 23.7 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: \( (P ;; U0)[<x>v]/<\$0-x^*> = (P[<x>v]/<\$x'>) ;; U0)\)
by (rel-auto)

lemma \(U1\)-seq-subst: \( (P ;; U1)[<x>v]/<\$1-x^*> = (P[<x>v]/<\$x'>) ;; U1)\)
by (rel-auto)

lemma lit-pbm-subst [usubst]:
fixes \(x::\) shows
\(\bigwedge\ P\ Q\ M\ \sigma.\ \sigma(x\mapsto_{\text{false}}) \vdash (P \parallel M\ Q) = \sigma \vdash ((P[<x>v]/<\$x']) \parallel \mathcal{M}_{<x>y}/<\$x'>(Q[<x>v]/<\$x>))\)
by (rel-auto)+

lemma bool-pbm-subst [usubst]:
fixes \(x::\) shows
\(\bigwedge\ P\ Q\ M\ \sigma.\ \sigma(x\mapsto_{\text{true}}) \vdash (P \parallel M\ Q) = \sigma \vdash ((P[<x>v]/<\$x']) \parallel \mathcal{M}_{<x>y}/<\$x'>(Q[<x>v]/<\$x>))\)
by (rel-auto)+

lemma zero-one-pbm-subst [usubst]:
fixes \(x::\) shows
\(\bigwedge\ P\ Q\ M\ \sigma.\ \sigma(x\mapsto_{\text{0}}) \vdash (P \parallel M\ Q) = \sigma \vdash ((P[<x>v]/<\$x']) \parallel \mathcal{M}_{<x>y}/<\$x'>(Q[<x>v]/<\$x>))\)
by (rel-auto)+

lemma numeral-pbm-subst [usubst]:
fixes \(x::\) shows
\(\bigwedge\ P\ Q\ M\ \sigma.\ \sigma(x\mapsto_{\text{numeral\ n}}) \vdash (P \parallel M\ Q) = \sigma \vdash ((P[<x>v]/<\$x']) \parallel \mathcal{M}_{<x>y}/<\$x'>(Q[<x>v]/<\$x>))\)
by (rel-auto)+

\(\begin{align*}
23.8 \quad \text{Parallel-by-merge laws} \\
\text{lemma par-by-merge-false [simp]:} \\
P \parallel \text{false} Q = \text{false} \\
\text{by (rel-auto)}
\end{align*}\)

\(\begin{align*}
\text{lemma par-by-merge-left-false [simp]:} \\
\text{false} \parallel M Q = \text{false} \\
\text{by (rel-auto)}
\end{align*}\)

\(\begin{align*}
\text{lemma par-by-merge-right-false [simp]:} \\
P \parallel M \text{false} = \text{false} \\
\text{by (rel-auto)}
\end{align*}\)

\(\begin{align*}
\text{lemma par-by-merge-seq-add:} \ (P \parallel M Q) ;; R = (P \parallel M ;; R Q)
\end{align*}\)
A skip parallel-by-merge yields a skip whenever the parallel predicates are both feasible.

**Lemma** par-by-merge-skip:
assumes $P ;; \text{true} = \text{true} \land Q ;; \text{true} = \text{true}$
shows $P \parallel_{\text{skip}} Q \equiv II$
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: $P \parallel_s Q ;; \text{swap}_m = Q \parallel_s P$
by (rel-auto)

Parallel-by-merge commutes when the merge predicate is unchanged by swap

**Lemma** par-by-merge-commute-swap:
shows $P \parallel M Q = Q \parallel M P$
proof –
have $Q \parallel_{\text{swap}_m} M P = (((Q ;; U0) \land (P ;; U1) \land \$v < \overline{u} = \$v) ;; \text{swap}_m) ;; M$
by (simp add: par-by-merge-def seqr-assoc)
also have $... = (((Q ;; U0) \land (P ;; U1) \land \$v < \overline{u} = \$v) ;; M)$
by (rel-auto)
also have $... = (((Q ;; U1) \land (P ;; U0) \land \$v < \overline{u} = \$v) ;; M)$
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 $M$ 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 $Q_1 \subseteq Q_2$
shows $(P \parallel_M Q_1) \subseteq (P \parallel_M Q_2)$
using assms by (rel-blast)

**Lemma** par-by-merge-mono:
assumes $P_1 \subseteq P_2 \land Q_1 \subseteq Q_2$
shows $(P_1 \parallel_M Q_1) \subseteq (P_2 \parallel_M Q_2)$
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 $(P \parallel_M Q) \parallel_M R = P \parallel_M (Q \parallel_M R)$
proof –
have $(P \parallel_M Q) \parallel_M R = ((P ;; U0) \land (Q ;; U0 ;; U1) \land (R ;; U1 ;; U1) \land \$v < \overline{u} \overline{u} = \$v) ;; M \&\& (M)$
by (rel-blast)
also have \( \ldots = (P :: U0) \land (Q :: U0 :: U1) \land (R :: U1 :: U1) \land \$v < ' = u \$v) :: \mathcal{M}3(M) \)
using AssocMerge-def assms(2) by force
also have \( \ldots = ((Q :: U0) \land (R :: U0 :: U1) \land (P :: U1 :: U1) \land \$v < ' = u \$v) :: \mathcal{M}3(M) \)
by (rel-blast)
also have \( \ldots = P \parallel_M (Q \parallel_M R) \),
by (simp add: assms(1) par-by-merge-commute)
finally show \(?thesis\)
qed

**Theorem** par-by-merge-choice-left:
\((P \cap Q) \parallel_M R = (P \parallel_M R) \cap (Q \parallel_M R)\)
by (rel-auto)

**Theorem** par-by-merge-choice-right:
\((P \parallel_M (Q \cap R)) = (P \parallel_M Q) \cap (P \parallel_M R)\)
by (rel-auto)

**Theorem** par-by-merge-or-left:
\((P \lor Q) \parallel_M R = (P \parallel_M R) \lor Q \parallel_M R\)
by (rel-auto)

**Theorem** par-by-merge-or-right:
\((P \parallel_M (Q \lor R)) = (P \parallel_M Q) \lor P \parallel_M R\)
by (rel-auto)

**Theorem** par-by-merge-USUP-mem-left:
\((\prod i : I \cdot P(i)) \parallel_M Q = (\prod i : I \cdot P(i) \parallel_M Q)\)
by (rel-auto)

**Theorem** par-by-merge-USUP-ind-left:
\((\prod i \cdot P(i)) \parallel_M Q = (\prod i \cdot P(i) \parallel_M Q)\)
by (rel-auto)

**Theorem** par-by-merge-USUP-mem-right:
\((\prod i : I \cdot Q(i)) = (\prod i : I \cdot P(i) \parallel_M Q(i))\)
by (rel-auto)

**Theorem** par-by-merge-USUP-ind-right:
\((\prod i \cdot Q(i)) = (\prod i \cdot P \parallel_M Q(i))\)
by (rel-auto)

### 23.9 Example: Simple State-Space Division

The following merge predicate divides the state space using a pair of independent lenses.

**Definition** StateMerge :: \( '(a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a \text{ merge } (M[\cdot]_{\sigma}) \text{ where} \)
upred-defs: \( M[a|b]_{\sigma} = (\$v < ' = u (\$v < \oplus \$0 - \$v \text{ on } a) \oplus \$1 - \$v \text{ on } a \& b) \)

**Lemma** swap-StateMerge: \( a \& b \Rightarrow (\text{swap}_m :: M[a|b]_{\sigma}) = M[b|a]_{\sigma} \)
by (rel-auto, simp-all add: lens-indep-comm)

**Abbreviation** StateParallel :: \( 'a \text{ hrel} \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel } (- | -)_{\sigma} \) - [85,0,0,86] 86
where \( P \parallel_M (a|b)_{\sigma} Q \equiv P \parallel_M |a|b_{\sigma} Q \)

153
Parallel Hoare logic rule. This employs something similar to separating conjunction in the theorem left and right of the parallel composition explicitly.

Postcondition, but we explicitly require that the two conjuncts only refer to variables on the lemma StateParallel-commute

\[ a \triangleright b \implies P |a|_\sigma Q = Q |b|_\sigma P \]

by (metis par-by-merge-commute-swap swap-StateMerge)

Specialised version of the above law where an invariant expression referring to variables outside the frame is preserved.

Parallelise the specification

We can frame all the variables that the parallel operator refers to

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]:

\[ (\exists (st_0, st_1) \cdot P[st_0] /\mathbb{v} \land P[st_1] /\mathbb{v}' \land Q[st_0] /\mathbb{v} \land Q[st_1] /\mathbb{v}' =_a (\mathbb{v} \oplus st_0 \lor a) \oplus st_1 \lor &b) \]

by (rel-auto)

Lemma StateParallel-form':

\[ (P |a|_\sigma Q = \{a,b\} : \{P |v|_\mathbb{v}, \mathbb{a} \} \land \{Q |v|_\mathbb{v}, \mathbb{b} \}) \]

using assms

apply (simp add: StateParallel-form, rel-auto)
  apply (metis vwb-lens-wb wb-lens-axioms-def wb-lens-def)
  apply (metis vwb-lens-wb wb-lens-get-put)
  apply (metis lens-indep-comm)
  apply (metis no-types, hide-lams lens-indep-comm vwb-lens-wb wb-lens-def weak-lens.put-get)
  done

Lemma StateParallel-frame:

\[ \{a\} P |a|_\sigma Q = \{a\} Q |a|_\sigma \]

using assms

apply simp add: StateParallel-form, rel-auto
  apply (metis lens-indep-comm)
  apply (fastforce+)
  done

Parallel Hoare rule by monotonicity of parallelism

Theorem StateParallel-frame-hoare [hoare]:

\[ (\exists (d_1, d_2) \cdot \{c\} P |a|_\sigma Q = \{c\} Q |d_1 \land d_2\} \]

Proof —
  — Parallellise the specification
  from assms(4,5)
  have 1: \[\{c\} \Rightarrow [d_1 \land d_2\} \subseteq \{c\} \Rightarrow [d_3\} \]
     \[|a|_\sigma ([c\} \Rightarrow [d_3\} \]
     by (simp add: StateParallel-form, rel-auto, metis assms(3) lens-indep-comm)
   — Prove Hoare rule by monotonicity of parallelism
   have 2: \[\exists (\mathbb{v}) \subseteq P \]
   proof (rule par-by-merge-mono)
     show \[\{c\} \Rightarrow [d_1\} \subseteq P \]
       using assms(1) hoare-r-def by auto
     show \[\{c\} \Rightarrow [d_2\} \subseteq Q \]
       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]:

154
assumes \( \text{vwb-lens} \ a \ \text{vwb-lens} \ b \ a \supseteq b \ a \not\subset d_1 \ b \not\subset d_2 \ a \not\subset c_1 \ b \not\subset c_1 \) \( P \ \parallel \) \( Q \) \( \sigma \) \( d_1 \) \( d_2 \) \( u \) \( \{ \sigma \land c_2 \} P \ | a | \ b | Q \ \{ \sigma \land d_1 \land d_2 \} u \\

proof -

\begin{align*}
\text{have} & \quad \{ \sigma \land c_2 \} \{ \& a, \& b \} : P \ | a | b | Q \ \{ \sigma \land d_1 \land d_2 \} u \\
& \quad \text{by (auto intro: frame-hoare-r StateParallel-hoare simp add: assms unrest plus-vwb-lens)}
\end{align*}

thus \( \sigma \).

\begin{align*}
\text{by (simp add: StateParallel-frame assms)}
\end{align*}

qed

end

## 24 Relational Operational Semantics

theory utp-rel-opsem
import 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 [14].

fun \( \text{tre} \) \( \alpha \) \( \text{usubst} \times \alpha \) \( \text{hrel} \) \( \Rightarrow \) \( \alpha \) \( \text{usubst} \times \alpha \) \( \text{hrel} \) \( \Rightarrow \) bool (infix \( \rightarrow_u \) 85) where

\begin{align*}
(\sigma, P) \rightarrow_u (\varphi, Q) \iff (\langle \sigma \rangle_a :: P \subseteq (\langle \varphi \rangle_a :: Q)
\end{align*}

\begin{itemize}
\item \textbf{lemma} \( \text{trans-trel} \):
\item \( [ (\sigma, P) \rightarrow_u (\varphi, Q) \mid (\varphi, Q) \rightarrow_u (\varphi, R) ] \Rightarrow (\sigma, P) \rightarrow_u (\varphi, R) \\
\item \text{by \( auto \)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \( \text{skip-trel} \): (\( \sigma, II \)) \( \rightarrow_u \) (\( \sigma, II \))
\item \text{by \( simp \)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \( \text{assigns-trel} \): (\( \sigma, (\varphi)_a \)) \( \rightarrow_u \) (\( \varphi \circ \sigma, II \))
\item \text{by \( simp \ add: \ assigns-comp \)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \( \text{assign-trel} \):
\item (\( \sigma, x := v \)) \( \rightarrow_u \) (\( \sigma(x := \sigma \uparrow v, II) \))
\item \text{by \( simp \ add: \ assigns-comp \ usubst \)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \( \text{seg-trel} \):
\item assumes (\( \sigma, P \)) \( \rightarrow_u \) (\( \varphi, Q \))
\item shows (\( \sigma, P \parallel R \)) \( \rightarrow_u \) (\( \varphi, Q \parallel R \))
\item \text{by \( (metis \ (no-types, lifting) \ \text{assms order-refl seqr-assoc seqr-mono trel.simps}) \)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \( \text{seg-skip-trel} \):
\item (\( \sigma, II \parallel P \)) \( \rightarrow_u \) (\( \sigma, P \))
\item \text{by \( simp \)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \( \text{nondet-left-trel} \):
\item (\( \sigma, P \cap Q \)) \( \rightarrow_u \) (\( \sigma, P \))
\item \text{by \( (metis \ (no-types, hide-lams) \ disj-comm \ disj-upred-def \ semilattice-sup-class.sup.absorb-iff1 \ semilattice-sup-class.sup.lifting \ seqr-or-distr trel.simps}) \)
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \( \text{nondet-right-trel} \):
\item (\( \sigma, P \cap Q \)) \( \rightarrow_u \) (\( \sigma, Q \))
\item \text{by \( (simp \ add: \ seqr-mono) \)}
\end{itemize}
lemma rcond-true-trel:
  assumes \( \sigma \uparrow b = true \)
  shows \( (\sigma, P \triangleleft b \triangleright Q) \rightarrow_u (\sigma, P) \)
  using assms
  by (simp add: assigns-r-comp usubst alpha cond-unit-T)

lemma rcond-false-trel:
  assumes \( \sigma \uparrow b = false \)
  shows \( (\sigma, P \triangleleft b \triangleright Q) \rightarrow_u (\sigma, Q) \)
  using assms
  by (simp add: assigns-r-comp usubst alpha cond-unit-F)

lemma while-true-trel:
  assumes \( \sigma \uparrow b = true \)
  shows \( (\sigma, \text{while } b \text{ do } P \text{ od}) \rightarrow_u (\sigma, P ;; \text{while } b \text{ do } P \text{ od}) \)
  by (metis assms rcond-true-trel while-unfold)

lemma while-false-trel:
  assumes \( \sigma \uparrow b = false \)
  shows \( (\sigma, \text{while } b \text{ do } P \text{ od}) \rightarrow_u (\sigma, II) \)
  by (metis assms rcond-false-trel while-unfold)

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.

theorem hoare-opsem-link:
\[ \{ p \} Q \{ r \}_u = (\forall \sigma_0 \sigma_1. \sigma_0 \uparrow p \land (\sigma_0, Q) \rightarrow_u (\sigma_1, II) \rightarrow (\sigma_1 \uparrow r)) \]
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]

end

25  Local Variables

theory utp-local
imports
  utp-rel-laws
  utp-meta-subst
  utp-theory
begin

25.1  Preliminaries

The following type is used to augment that state-space with a stack of local variables represented as a list in the special variable \textit{store}. Local variables will be represented by pushing variables onto the stack, and popping them off after use. The element type of the stack is \'u which corresponds to a suitable injection universe.

alphabet \'u local =
  store :: \'u list

156
State-space with a countable universe for local variables.

type-synonym ′a clocal = (nat, ′a) local-scheme

The following predicate wraps the relation with assumptions that the stack has a particular size before and after execution.

definition local-num where
local-num n P = [#u(store) =_u n] T ;; P ;; [#u(store) =_u n] T

declare inj-univ.from-univ-def [upred-defs]
declare inj-univ.to-univ-lens-def [upred-defs]
declare nat-inj-univ-def [upred-defs]

25.2 State Primitives

The following record is used to characterise the UTP theory specific operators we require in order to create the local variable operators.

record (′α, ′s) state-prim =

— The first field states where in the alphabet ′α the user state-space type is ′s is located with the form of a lens.

sstate :: ′s ⇒ ′α (si)

— The second field is the theory’s substitution operator. It takes a substitution over the state-space type and constructs a homogeneous assignment relation.

sassigns :: ′s usubst ⇒ ′α hrel (⦇⦈i)

syntax
-sstate :: logic ⇒ svid (si)

translations
-sstate T => CONST sstate T

The following record type adds an injection universe ′u to the above operators. This is needed because the stack has a homogeneous type into which we must inject type variable bindings. The universe can be any Isabelle type, but must satisfy the axioms of the locale inj-univ, which broadly shows the injectable values permitted.

record (′α, ′s, ′u, ′a) local-prim = (′α, (′u, ′s) local-scheme) state-prim +
inj-local :: (′a, ′u) inj-univ

The following locales give the assumptions required of the above signature types. The first gives the defining axioms for state-spaces. State-space lens s must be a very well-behaved lens, and sequential composition of assignments corresponds to functional composition of the underlying substitutions. TODO: We might also need operators to properly handle framing in the future.

locale utp-state =
fixes S (structure)
assumes vwb-lens s
and passigns-comp: (⦇⦈ ⦈) = (⦈ ◦ ⦈)

The next locale combines the axioms of a state-space and an injection universe structure. It then uses the required constructs to create the local variable operators.
locale utp-local-state = utp-state $S +$ inj-univ inj-local $S$ for $S :: ('a, 's, 'u::two, 'a)$ local-prim (structure)

begin

The following two operators represent opening and closing a variable scope, which is implemented by pushing an arbitrary initial value onto the stack, and popping it off, respectively.

definition var-open :: 'a hrel (open_u) where
var-open = ($\prod$ v $\cdot$ ([store $\mapsto$ s ($&$store $\vdash$ v $\mapsto$)[])])

definition var-close :: 'a hrel (close_u) where
var-close = ([store $\mapsto$ s front_u($&$store) $\triangleq$ $\#_u(\&$store) $>$ u $\triangleright$ &store])

The next operator is an expression that returns a lens pointing to the top of the stack. This is effectively a dynamic lens, since where it points to depends on the initial number of variables on the stack.

definition top-var :: ('a $\Rightarrow$ ('u, 'b) local-scheme, 'a) uexpr (top_u) where
top-var = $\langle\lambda$ l. to-univ-lens;$\delta$ list-lens l;$\delta$L store($\#_u(\&$store) $-$ 1)$\rangle_u$

Finally, we combine the above operators to represent variable scope. This is a kind of binder which takes a homogeneous relation, parametric over a lens, and returns a relation. It simply opens the variable scope, substitutes the top variable into the body, and then closes the scope afterwards.

definition var-scope :: (('a $\Rightarrow$ ('u, 's) local-scheme) $\Rightarrow$ 'a hrel) $\Rightarrow$ 'a hrel where
var-scope $f$ = open_u $; f(x)[x\mapsto[\text{top}$'] $; close_u$
end

notation utp-local-state.var-open (open[-$\cdot$])
notation utp-local-state.var-close (close[-$\cdot$])
notation utp-local-state.var-scope (V[-$\cdot$] [-$\cdot$])
notation utp-local-state.top-var (top[-$\cdot$])

syntax

-var-scope :: logic $\Rightarrow$ id $\Rightarrow$ logic $\Rightarrow$ logic (var[-$\cdot$] $\cdot$ $\cdot$ $\cdot$ [0, 0, 10] 10)
-var-scope-type :: logic $\Rightarrow$ id $\Rightarrow$ type $\Rightarrow$ logic (var[-$\cdot$] $\cdot$ $\cdot$ $\cdot$ [0, 0, 10] 10)

translations

-var-scope T x P == CONST utp-local-state.var-scope T (\lambda x. P)
-var-scope-type T x t P $\Rightarrow$ CONST utp-local-state.var-scope T (-abs (-constrain x (-uvar-ty t)) P)

Next, we prove a collection of important generic laws about variable scopes using the axioms defined above.

context utp-local-state
begin

lemma var-open-commute:
[ x $\bowtie$ store; store $\vdash$ v $\mapsto$ ] $\Rightarrow$ ([x $\mapsto$ s v]) $; open_u = open_v $; ([x $\mapsto$ s v])
by (simp add: var-open-def passigns-comp seq-UINF-distl' seq-UINF-distr' usubst unrest lens-indep-sym, simp add: usubst-upd-comm)

lemma var-close-commute:
[ x $\bowtie$ store; store $\vdash$ v $\mapsto$ ] $\Rightarrow$ ([x $\mapsto$ s v]) $; close_v = close_u $; ([x $\mapsto$ s v])
by (simp add: var-close-def passigns-comp seq-UINF-distl' seq-UINF-distr' usubst unrest lens-indep-sym, simp add: usubst-upd-comm)

158
lemma var-open-close-lemma:
\[ \begin{array}{c}
\text{store} \mapsto \text{front}_u (\& \text{store} \hat{u} \langle \ll v \gg \rangle) \triangleleft 0 < u \#_u (\& \text{store} \hat{u} \langle \ll v \gg \rangle) \triangleright \& \text{store} \hat{u} \langle \ll v \gg \rangle) = \text{id} \\
\text{by (rel-auto)}
\end{array} \]

lemma var-open-close: \( \text{open}_v ; \text{close}_v = \{ \text{id} \} \)
\text{by (simp add: var-open-def var-close-def seq-UINF-distr' passigns-comp usubst var-open-close-lemma)}

lemma var-scope-skip: \( (\text{var}[S] x \cdot \langle \text{id} \rangle) = \langle \text{id} \rangle \)
\text{by (simp add: var-scope-def var-open-def var-close-def seq-UINF-distr' passigns-comp var-open-close-lemma usubst)}

lemma var-scope-nonlocal-left:
\[ [x \triangleright \text{store} ; \text{store} \hat{u} v] \Rightarrow [x \mapsto v] \] ; (\text{var}[S] y \cdot P(y)) = (\text{var}[S] y \cdot (x \mapsto v)) ; P(y)

oops
end

declare utp-local-state.var-open-def [upred-defs]
declare utp-local-state.var-close-def [upred-defs]
declare utp-local-state.top-var-def [upred-defs]
declare utp-local-state.var-scope-def [upred-defs]

25.3 Relational State Spaces

To illustrate the above technique, we instantiate it for relations with a \( \text{nat} \) as the universe type. The following definition defines the state-space location, assignment operator, and injection universe for this.

definition rel-local-state ::
\( 'a::{\text{countable \ itself}} \Rightarrow ((\text{nat}, 's) \text{ local-scheme}, 's, \text{nat}, 'a::{\text{countable}}) \text{ local-prim} \) where
\( \text{rel-local-state} t = (| \text{sstate} = 1_L, \text{sassigns} = \text{assigns-r}, \text{inj-local} = \text{nat-inj-univ} |) \)

abbreviation rel-local \((\text{R}_l)\) where
\( \text{rel-local} \equiv \text{rel-local-state \ TYPE('a::{\text{countable}})} \)

syntax
\(-\text{rel-local-state-type} :: \text{type} \Rightarrow \text{logic} \ (\text{R}_l[-]) \)

translations
\(-\text{rel-local-state-type} \ a = \Rightarrow \text{CONST rel-local-state (-\text{TYPE} \ a)} \)

lemma get-rel-local [lens-defs]:
\( \text{get}_{\text{R}_l} = \text{id} \)
\text{by (simp add: rel-local-state-def lens-defs)}

lemma rel-local-state [simp]: utp-local-state \( R_l \)
\text{by (unfold-locales, simp-all add: upred-defs assigns-comp rel-local-state-def)}

lemma sassigns-rel-state [simp]: \( (\sigma)_R = (\sigma)_a \)
\text{by (simp add: rel-local-state-def)}

syntax
\(-\text{rel-var-scope} :: \text{id} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\text{var - \cdot \cdot [0, 10] 10}) \)
\(-\text{rel-var-scope-type} :: \text{id} \Rightarrow \text{type} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\text{var - \cdot \cdot \cdot [0, 0, 10] 10}) \)
translations
- rel-var-scope $x \rightarrow P$ => var-scope $R \rightarrow x \rightarrow P$
- rel-var-scope-type $x \rightarrow t \rightarrow P$ => var-scope-type ($rel-local-state-type t$) $x \rightarrow t \rightarrow P$

Next we prove some examples laws.

**Lemma** rel-var-ex-1: $(\text{var } x :: \text{string} \cdot II) = II$
  by (rel-auto')

**Lemma** rel-var-ex-2: $(\text{var } x \cdot x := 1) = II$
  by (rel-auto')

**Lemma** rel-var-ex-3: $x \triangleright store \Rightarrow x := 1 ;; \text{open}[R[\text{a} :: \text{countable}]] = \text{open}[R[\text{a}]] ;; x := 1$
  by (metis pr-var-def rel-local-state sassigns-rel-state unrest-one utp-local-state.var-open-commute)

**Lemma** rel-var-ex-4: $[x \triangleright store; \text{store } \triangleright v] \Rightarrow x := v ;; \text{open}[R[\text{a} :: \text{countable}]] = \text{open}[R[\text{a}]] ;; x := v$
  by (metis pr-var-def rel-local-state sassigns-rel-state utp-local-state.var-open-commute)

**Lemma** rel-var-ex-5: $[x \triangleright store; \text{store } \triangleright v] \Rightarrow x := v ;; (\text{var } y :: \text{int} \cdot P) = (\text{var } y :: \text{int} \cdot x := v ;; P)$

end

26 Meta-theory for the Standard Core

theory utp
imports
  utp-var
  utp-expr
  utp-unrest
  utp-usedby
  utp-subst
  utp-meta-subst
  utp-alphabet
  utp-lift
  utp-pred
  utp-pred-laws
  utp-recursion
  utp-deduct
  utp-rel
  utp-rel-laws
  utp-tactics
  utp-hoare
  utp-wp
  utp-sp
  utp-theory
  utp-concurrency
  utp-rel-opsem
  utp-local
  utp-event

begin end

160
References


