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 (Unpublished)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
Unless indicated otherwise, fulltext items are protected by copyright with all rights reserved. The copyright exception in section 29 of the Copyright, Designs and Patents Act 1988 allows the making of a single copy solely for the purpose of non-commercial research or private study within the limits of fair dealing. The publisher or other rights-holder may allow further reproduction and re-use of this version - refer to the White Rose Research Online record for this item. Where records identify the publisher as the copyright holder, users can verify any specific terms of use on the publisher’s website.

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
6 Substitution

6.1 Substitution definitions ........................................... 33
6.2 Syntax translations .................................................. 34
6.3 Substitution Application Laws .................................... 35
6.4 Substitution laws ..................................................... 38
6.5 Ordering substitutions ............................................... 40
6.6 Unrestriction laws ................................................... 40
6.7 Conditional Substitution Laws .................................... 41
6.8 Parallel Substitution Laws ......................................... 41

7 UTP Tactics

7.1 Theorem Attributes .................................................. 42
7.2 Generic Methods ..................................................... 42
7.3 Transfer Tactics ..................................................... 43
  7.3.1 Robust Transfer ................................................ 43
  7.3.2 Faster Transfer ................................................ 43
7.4 Interpretation ....................................................... 44
7.5 User Tactics ........................................................ 44

8 Meta-level Substitution ................................................ 46

9 Alphabetised Predicates .............................................. 47

9.1 Predicate type and syntax ......................................... 47
9.2 Predicate operators ................................................ 48
9.3 Unrestriction Laws .................................................. 53
9.4 Used-by laws ........................................................ 55
9.5 Substitution Laws ................................................... 55

10 UTP Events

10.1 Events .............................................................. 58
10.2 Channels ............................................................ 58
  10.2.1 Operators ....................................................... 58

11 Alphabet Manipulation .............................................. 59

11.1 Preliminaries ....................................................... 59
11.2 Alphabet Extrusion ................................................ 59
11.3 Expression Alphabet Restriction ................................ 62
11.4 Predicate Alphabet Restriction .................................. 63
11.5 Alphabet Lens Laws ............................................... 63
11.6 Substitution Alphabet Extension ................................ 64
11.7 Substitution Alphabet Restriction ................................ 64

12 Lifting Expressions .................................................. 65

12.1 Lifting definitions ................................................ 65
12.2 Lifting Laws ......................................................... 65
12.3 Substitution Laws .................................................. 66
12.4 Unrestriction laws ................................................ 66
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.14</td>
<td>Omega</td>
<td>122</td>
</tr>
<tr>
<td>18.15</td>
<td>Relation Algebra Laws</td>
<td>122</td>
</tr>
<tr>
<td>18.16</td>
<td>Kleene Algebra Laws</td>
<td>122</td>
</tr>
<tr>
<td>18.17</td>
<td>Omega Algebra Laws</td>
<td>124</td>
</tr>
<tr>
<td>18.18</td>
<td>Refinement Laws</td>
<td>124</td>
</tr>
<tr>
<td>18.19</td>
<td>Domain and Range Laws</td>
<td>124</td>
</tr>
<tr>
<td>19.1</td>
<td>Complete lattice of predicates</td>
<td>125</td>
</tr>
<tr>
<td>19.2</td>
<td>UTP theories hierarchy</td>
<td>126</td>
</tr>
<tr>
<td>19.3</td>
<td>UTP theory hierarchy</td>
<td>128</td>
</tr>
<tr>
<td>19.4</td>
<td>Theory of relations</td>
<td>136</td>
</tr>
<tr>
<td>19.5</td>
<td>Theory links</td>
<td>137</td>
</tr>
<tr>
<td>20.1</td>
<td>Hoare Triple Definitions and Tactics</td>
<td>138</td>
</tr>
<tr>
<td>20.2</td>
<td>Basic Laws</td>
<td>138</td>
</tr>
<tr>
<td>20.3</td>
<td>Assignment Laws</td>
<td>139</td>
</tr>
<tr>
<td>20.4</td>
<td>Sequence Laws</td>
<td>139</td>
</tr>
<tr>
<td>20.5</td>
<td>Conditional Laws</td>
<td>139</td>
</tr>
<tr>
<td>20.6</td>
<td>Recursion Laws</td>
<td>140</td>
</tr>
<tr>
<td>20.7</td>
<td>Iteration Rules</td>
<td>140</td>
</tr>
<tr>
<td>20.8</td>
<td>Frame Rules</td>
<td>142</td>
</tr>
<tr>
<td>21.1</td>
<td>Weakest Precondition Calculus</td>
<td>143</td>
</tr>
<tr>
<td>22.1</td>
<td>Strong Postcondition Calculus</td>
<td>144</td>
</tr>
<tr>
<td>23.1</td>
<td>Variable Renamings</td>
<td>146</td>
</tr>
<tr>
<td>23.2</td>
<td>Merge Predicates</td>
<td>147</td>
</tr>
<tr>
<td>23.3</td>
<td>Separating Simulations</td>
<td>148</td>
</tr>
<tr>
<td>23.4</td>
<td>Associative Merges</td>
<td>149</td>
</tr>
<tr>
<td>23.5</td>
<td>Parallel Operators</td>
<td>150</td>
</tr>
<tr>
<td>23.6</td>
<td>Unrestriction Laws</td>
<td>150</td>
</tr>
<tr>
<td>23.7</td>
<td>Substitution laws</td>
<td>150</td>
</tr>
<tr>
<td>23.8</td>
<td>Parallel-by-merge laws</td>
<td>151</td>
</tr>
<tr>
<td>23.9</td>
<td>Example: Simple State-Space Division</td>
<td>153</td>
</tr>
<tr>
<td>24.1</td>
<td>Relational Operational Semantics</td>
<td>155</td>
</tr>
<tr>
<td>25.1</td>
<td>Preliminaries</td>
<td>156</td>
</tr>
<tr>
<td>25.2</td>
<td>State Primitives</td>
<td>156</td>
</tr>
<tr>
<td>25.3</td>
<td>Relational State Spaces</td>
<td>157</td>
</tr>
<tr>
<td>26.1</td>
<td>Meta-theory for the Standard Core</td>
<td>160</td>
</tr>
</tbody>
</table>
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 unrestricted, 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
  imports ../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 ⊑ 50) and
  Lattice.sup (⨆) [90] 90 and
  Lattice.inf (⨅) [90] 90 and
  Lattice.join (infixl ⊔ 65) and
  Lattice.meet (infixl ⊓ 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 ‘α, 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 ⇒ ‘α, where the view type ‘a is the variable type, and the source type ‘α 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** in-var :: ('a \rightarrow 'a) \Rightarrow ('a \rightarrow 'a × 'b) where

[lens-defs]: in-var x = x :_L fst_L

**definition** out-var :: ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'a × 'b) where

[lens-defs]: out-var x = x :_L snd_L

Variables can also be used to effectively define sets of variables. Here we define the the universal alphabet (Σ) 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** (input) univ-alpha :: ('a => 'a) (Σ) where univ-alpha ≡ 1_L

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

**definition** pr-var :: ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) where

[lens-defs]: 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** in-var-weak-lens [simp]: weak-lens x \Rightarrow weak-lens (in-var x)  
by (simp add: comp-weak-lens in-var-def)

**lemma** in-var-semi-uvar [simp]: mwb-lens x \Rightarrow mwb-lens (in-var x)  
by (simp add: comp-mwb-lens in-var-def)

**lemma** pr-var-weak-lens [simp]: weak-lens x \Rightarrow weak-lens (pr-var x)  
by (simp add: pr-var-def)

**lemma** pr-var-mwb-lens [simp]: mwb-lens x \Rightarrow mwb-lens (pr-var x)  
by (simp add: pr-var-def)

**lemma** pr-var-vwb-lens [simp]: vwb-lens x \Rightarrow vwb-lens (pr-var x)  
by (simp add: pr-var-def)

**lemma** in-var-uvar [simp]: vwb-lens x \Rightarrow vwb-lens (in-var x)  
by (simp add: in-var-def)

**lemma** out-var-weak-lens [simp]: weak-lens x \Rightarrow weak-lens (out-var x)  
by (simp add: comp-weak-lens out-var-def)

**lemma** out-var-semi-uvar [simp]: mwb-lens x \Rightarrow mwb-lens (out-var x)
by (simp add: comp-mwb-lens out-var-def)

lemma out-var-uvar [simp]:
  \( \text{vwb-lens } x \implies \text{vwb-lens } (\text{out-var } x) \)
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]:
  \( \text{in-var } x \bowtie \text{out-var } y \)
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]:
  \( \text{out-var } x \bowtie \text{in-var } y \)
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]:
  \( x \bowtie y \implies \text{in-var } x \bowtie \text{in-var } y \)
by (simp add: in-var-def out-var-def)

lemma out-var-indep [simp]:
  \( x \bowtie y \implies \text{out-var } x \bowtie \text{out-var } y \)
by (simp add: out-var-def)

lemma pr-var-indeps [simp]:
  \( x \bowtie y \implies \text{pr-var } x \bowtie \text{pr-var } y \)
by (simp-all add: pr-var-def)

lemma prod-lens-indep-in-var [simp]:
  \( a \bowtie x \implies a \times_L b \bowtie \text{in-var } x \)
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]:
  \( b \bowtie x \implies a \times_L b \bowtie \text{out-var } x \)
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]:
  \( \text{in-var } (\text{pr-var } x) = \text{in-var } x \)
by (simp add: pr-var-def)

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

lemma pr-var-idem [simp]:
  \( \text{pr-var } (\text{pr-var } x) = \text{pr-var } x \)
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 snd-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 :: \(\text{id} \Rightarrow \text{svid} \ (- [999] 999)\)
  -svid-unit :: \(\text{svid} \Rightarrow \text{svids} \ (-)\)
  -svid-list :: \(\text{svid} \Rightarrow \text{svids} \Rightarrow \text{svids} \ (-,/-)\)
  -svid-alpha :: \(\text{svid} \ (v)\)
  -svid-dot :: \(\text{svid} \Rightarrow \text{svid} \Rightarrow \text{svid} \ (- [998,999] 998)\)
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 (infixl ; 75)`
- `salphacompp :: 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 (\{\})\alpha`
- `svar :: svar => logic (\'\{-\})\nu`

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

-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
-salba-all = 1_L
-salba-none = 0_L

— 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 len sum. The translation rules are carefully crafted to ensure both parsing and pretty printing.

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

syntax
-aivar-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 -aivar-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 \(\alpha\) to the expression’s type \(\tau\). 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 \(\langle\tau,\alpha\rangle\) uexpr = UNIV:: (\(\alpha\) \Rightarrow \tau) set..

setup-lifting type-definition-uexpr

notation Rep-uexpr (\[\[\cdot\]\])

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

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

named-theorems ueval and lit-simps and lit-norm

3.2 Core expression constructs

A variable expression corresponds to the lens 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 :: \(\tau \Rightarrow \alpha\) \Rightarrow \langle\tau, \alpha\rangle\) uexpr is lens-get.

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

lift-definition lit :: \(\tau \Rightarrow \langle\tau, \alpha\rangle\) uexpr is \(\lambda v b. v\).

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 :: \(\alpha \Rightarrow \beta\) \Rightarrow \langle\alpha, \alpha\rangle\) uexpr \Rightarrow \langle\beta, \alpha\rangle\) uexpr
is \(\lambda f e b. f (e b)\).

lift-definition bop ::
\(\alpha \Rightarrow \beta \Rightarrow \gamma\) \Rightarrow \langle\alpha, \alpha\rangle\) uexpr \Rightarrow \langle\beta, \alpha\rangle\) uexpr \Rightarrow \langle\gamma, \alpha\rangle\) uexpr
\[ \lambda f u v b. f (u b) (v b) . \]

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

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

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

**lift-definition ulambda ::**
\[(a \Rightarrow (b, \alpha) \text{ uexpr}) \Rightarrow (a \Rightarrow b, \alpha) \text{ uexpr} \]
\[\lambda f A x. f x A .\]

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

**abbreviation cond ::**
\[(a, \alpha) \text{ uexpr} \Rightarrow (\text{bool}, \alpha) \text{ uexpr} \Rightarrow (a, \alpha) \text{ uexpr} \Rightarrow (a, \alpha) \text{ uexpr} \]
\[((\_ - \_ - \_ / \_ ) [-52,0,53]) 52 \]
**where** 
\[P \triangleq b \triangleright Q \equiv \text{ trop If } b P Q \]

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

**definition eq-upred ::**
\[(a, \alpha) \text{ uexpr} \Rightarrow (a, \alpha) \text{ uexpr} \Rightarrow (a, \alpha) \text{ uexpr} \]
\[\lambda \text{ eq-upred x y } = \text{ bop HOL.eq x y} \]

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**
\[\text{ ulit :: } \tau \Rightarrow \tau, \text{ and } \text{ ueq :: } \tau \Rightarrow \tau \Rightarrow \text{ bool } \]
\[\lambda \text{ eq-upred }\]
A literal is the expression \(\ll v \gg\), 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**
\[-uvar :: \text{ svar } \Rightarrow \text{ logic } (\sim) \]

**translations**
\[-uvar x == \text{ const var } x \]

Since we already have a parser for variables, we can directly reuse it and simply apply the \textit{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.

```
imstantiation uexpr :: (zero, type) zero
begin
  definition zero-uexpr-def: 0 = lit 0
instance ..
end
```

```
imstantiation uexpr :: (one, type) one
begin
  definition one-uexpr-def: 1 = lit 1
instance ..
end
```

```
imstantiation uexpr :: (plus, type) plus
begin
  definition plus-uexpr-def: u + v = bop (op +) u v
instance ..
end

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

```
imstantiation uexpr :: (uminus, type) uminus
begin
  definition uminus-uexpr-def: − u = uop uminus u
instance ..
end
```

```
imstantiation uexpr :: (minus, type) minus
begin
  definition minus-uexpr-def: u − v = bop (op −) u v
instance ..
end
```

```
imstantiation uexpr :: (times, type) times
begin
  definition times-uexpr-def: u ∗ v = bop (op ∗) u v
instance ..
end
```

```
instance uexpr :: (Rings.dvd, type) Rings.dvd ..
```

```
imstantiation uexpr :: (divide, type) divide
begin
  definition divide-uexpr :: ('a, 'b) uexpr ⇒ ('a, 'b) uexpr ⇒ ('a, 'b) uexpr where
divide-uexpr u v = bop divide u v
instance ..
end
```

```
imstantiation uexpr :: (inverse, type) inverse
begin
  definition inverse-uexpr :: ('a, 'b) uexpr ⇒ ('a, 'b) uexpr where
  inverse-uexpr u = uop 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: add.commute)+

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

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

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

instance $uexpr :: (ab-group-add, type) ab-group-add$
  by (intro-classes)
    (simp add: plus-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 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 \( op \leq \) and \( op \leq \) return a \( \text{bool} \) 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 λ P Q. (∀ A. P A \leq Q A).
  definition less-uexpr :: (′a, ′b) uexpr ⇒ (′a, ′b) uexpr ⇒ bool
    where less-uexpr P Q = (P \leq Q ∧ ¬ Q \leq P)
instance ..
end

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

instance uexpr :: (order, type) order
proof
  fix x y z :: (′a, ′b) uexpr
  show \( x < y \) = \( x \leq y ∧ ¬ y \leq x \) by (simp add: less-uexpr-def)
  show \( x \leq x \) by (transfer, auto)
  show \( x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z \)
  by (transfer, blast intro:order.trans)
  show \( x \leq y \Longrightarrow y \leq x \Longrightarrow x = y \)
  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-ineq)+
  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 representations (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 numeral-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 t n] = (λ b. [P] t 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 :: ('a::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 :: ('a::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 fun-apply :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
where fun-apply f x = f x

declare fun-apply-def [simp]

definition ffun-entries :: 'k set ⇒ ('k ⇒ 'v) ⇒ ('k, 'v)
ffun-entries d f = graph-ffun \{(k, f k) \mid k. k \in d\}

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

adhoc-overloading
uempty 0 and
wapply fun-apply and wapply nth and wapply pfun-app and
wupd pfun-upd and wupd ffun-upd and wupd list-augment and
udom Domain and udom pdom and udom fdom and udom seq-dom and
udom Range and uran prun and uran fran and uran set and
udomres pdom-res and udomres fdom-res and
uranres pran-res and udomres fran-res and
ucard card and ucard pcard and ucard length and
usums list-sum and usums Sum and usums pfun-sum and
uentries pfun-entries and uentries 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 (input) ulens-override x f g ≡ lens-override f g x

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

definition set-of :: 'a itself ⇒ 'a set
set-of t = UNIV

translations
0 <= CONST uempty — We have to do this so we don’t see uempty. Is there a better way of printing?

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

nonterminal atuple-args and umaplet and umaplets

syntax — Core expression constructs
-ucoerce :: logic ⇒ type ⇒ logic (infix :_ 50)
-ulambda :: pttrn ⇒ logic ⇒ logic (\lambda \cdot \cdot [0, 10] 10)
-ulens-ovrd :: logic ⇒ logic ⇒ salpha ⇒ logic (\oplus \cdot on - [85, 0, 86] 86)
-ulens-get :: logic ⇒ svar ⇒ logic (\langle· [900,901] 901)

translations
\lambda x \cdot p == CONST ulambda (\lambda x. p)
x :_ a == x :: ('a, -) uexpr
-ulens-ovrd f g a => CONST bop (CONST ulens-override a) f g
-ulens-ovrd f g a <= CONST bop (\lambda x y. CONST lens-override x1 y1 a) f g
-ulens-get x y == CONST uop (CONST lens-get y) x

syntax — Tuples
-utuple :: ('a, 'a) uexpr ⇒ utuple-args ⇒ ('a * 'b, 'a) uexpr ((I ('/-' /)_)u)
-utuple-arg :: ('a, 'a) uexpr ⇒ utuple-args (-)
-utuple-args :: ('a, 'a) uexpr ⇒> utuple-args ⇒ utuple-args (-/-)
-umin :: ('a, 'a) uexpr ('(')u)
-afst :: ('a × 'b, 'a) uexpr ⇒ ('a, 'a) uexpr (π₁(')')
-usnd :: ('a × 'b, 'a) uexpr ⇒ ('b, 'a) uexpr (π₂(')')

translations
( )u == <(>
(x, y)u == CONST bop (CONST Pair) x y
-utuple x (-utuple-args y z) == -utuple x (-utuple-arg -utuple-args y z)
π₁(x) == CONST uop CONST fst x
π₂(x) == CONST uop CONST snd x

syntax — Polymorphic constructs
-undef :: logic (⊥u)
-unmap-empty :: logic ([]u)
-unmap :: logic ⇒ logic ([-']u) [999.0] 999
-unmaplet :: [logic, logic] ⇒> umaplet (-/⇒/ -)
-unmaplets :: [umaplet, umaplets] ⇒> umaplets (-/ -)
-UMaplets :: [umaplet, umaplets] ⇒> umaplets (-/ -)
-UMap :: [umaplets ⇒ logic ([I[-]u])]
-ucard :: logic ⇒ logic (#_u('')
-uless :: logic ⇒ logic ⇒ logic (infix <_u 50)
-uleq :: logic ⇒ logic ⇒ logic (infix ≤_u 50)
-ugreat :: logic ⇒ logic ⇒ logic (infix >_u 50)
-ugeq :: logic ⇒ logic ⇒ logic (infix ≥_u 50)
-uceil :: logic ⇒ logic ([]-]u)
-ufloor :: logic ⇒ logic ([-|]u)
-udom :: logic ⇒ logic (dom_u(''))
-uran :: logic ⇒ logic (ran_u(''))
-usum :: logic ⇒ logic (sum_u(''))
-udom-res :: logic ⇒ logic ⇒ logic (infix1 <u 85)
-uran-res :: logic ⇒ logic ⇒ logic (infix1 >u 85)
-umin :: logic ⇒ logic ⇒ logic (min_u(''-']'))
-umax :: logic ⇒ logic ⇒ logic (max_u(''-']'))
-ugcd :: logic ⇒ logic ⇒ logic (gdc_u('''''')
-uentries :: logic ⇒ logic ⇒ logic (entr_u(''''''))

translations
— Pretty printing for adhoc-overloaded constructs
f(x)u <= CONST uapp f x
dom_u(f) <= CONST udom f
ran_u(f) <= CONST uran f
A <u f <= CONST udomres A f
f >u A <= CONST uranres f A
#_u(f) <= CONST ucard f
f(k ⇀ v)u <= CONST uappd f k v

— Overloaded construct translations
f(x,y,z,u)u == CONST bop CONST uapp f (x,y,z,u)u
f(x,y,z)u == CONST bop CONST uapp f (x,y,z)u
f(x,y)u == CONST bop CONST uapp f (x,y)u
f(x)u == CONST bop CONST uapp f x

20
\#_u(xs) == CONST uop CONST ucard xs

\sum_u(A) == CONST uop CONST usums A

dom_u(f) == CONST uop CONST udom f

ran_u(f) == CONST uop CONST uran f

[]_u == \langle\text{CONST uempty}\rangle

⊥_u == \langle\text{CONST undefined}\rangle

A <_u f == CONST bop (CONST udomres) A f

f \triangleright_u A == CONST bop (CONST uranres) f A

\text{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{UMap}(\text{UMaplet}(x, y)) == CONST trop CONST usums A

\text{UMap}(ms) == \text{UMap}\,[\text{UMap}(ms)]

\text{UMaplets}(ms1, ms2) <= \text{UMap}(\text{UMap}(ms1) ms2)

\text{UMaplets}(ms1, ms2, ms3) <= \text{UMaplets}(\text{UMap}(ms1) ms2) ms3

— Type-class polymorphic constructs

\(x <_u y\) == CONST bop (op \(\leq\)) x y

\(x \leq_u y\) == CONST bop (op \(\leq\)) x y

\(x \geq_u y\) == \(\triangleright\) y \leq x

\(x \geq_u y\) == \(\triangleright\) y \leq x

\text{min}_u(x, y) == CONST bop (CONST \text{min}) x y

\text{max}_u(x, y) == CONST bop (CONST \text{max}) x y

\text{gcd}_u(x, y) == CONST bop (CONST \text{gcd}) x y

\(x\) |\_u == CONST uop CONST ceiling x

\[x\] \_u == CONST uop CONST floor x

\textbf{syntax} — Lists / Sequences

-\textbf{unil} :: \langle'a list, \alpha\rangle uexpr \langle\rangle

-\textbf{ulist} :: \langle'a list, \alpha\rangle uexpr \langle\langle\rangle\rangle

-\textbf{uappend} :: \langle'a list, \alpha\rangle uexpr \rightarrow \langle'a list, \alpha\rangle uexpr \rightarrow \langle'a list, \alpha\rangle uexpr (\text{infixr} \_u 80)

-\textbf{ulast} :: \langle'a list, \alpha\rangle uexpr \rightarrow \langle'a list, \alpha\rangle uexpr (\text{last}_u \langle\rangle)

-\textbf{ufront} :: \langle'a list, \alpha\rangle uexpr \rightarrow \langle'a list, \alpha\rangle uexpr (\text{front}_u \langle\rangle)

-\textbf{uhead} :: \langle'a list, \alpha\rangle uexpr \rightarrow \langle'a, \alpha\rangle uexpr (\text{head}_u \langle\rangle)

-\textbf{uextract} :: \langle'a set, \alpha\rangle uexpr \rightarrow \langle'a list, \alpha\rangle uexpr \rightarrow \langle'a list, \alpha\rangle uexpr (\text{infixl} \_u 75)

-\textbf{ucats} :: \langle'a set, \alpha\rangle uexpr \rightarrow \langle'a list, \alpha\rangle uexpr \rightarrow \langle'a list, \alpha\rangle uexpr (\text{infixl} \_u 75)

-\textbf{uelems} :: \langle'a list, \alpha\rangle uexpr \rightarrow \langle'a set, \alpha\rangle uexpr (\text{elems}_u \langle\rangle)

-\textbf{usorted} :: \langle'a list, \alpha\rangle uexpr \rightarrow \langle\text{bool}, \alpha\rangle uexpr (\text{sorted}_u \langle\rangle)

-\textbf{udistinct} :: \langle'a list, \alpha\rangle uexpr \rightarrow \langle\text{bool}, \alpha\rangle uexpr (\text{distinct}_u \langle\rangle)

-\textbf{uupto} :: \text{logic} \rightarrow \text{logic} \rightarrow \text{logic} (\langle\text{--}\rangle)

-\textbf{uupt} :: \text{logic} \rightarrow \text{logic} \rightarrow \text{logic} (\langle\text{--}\rangle)

-\textbf{uzip} :: \text{logic} \rightarrow \text{logic} \rightarrow \text{logic} (\text{zip}_u)

-\textbf{utr-iter} :: \text{logic} \rightarrow \text{logic} \rightarrow \text{logic} (\text{iter}[-\langle\rangle\langle\rangle])

\textbf{translations}

\langle\rangle == \langle\text{[]}\rangle

\langle x, xs\rangle == CONST bop (op \#) x \langle xs\rangle

\langle x\rangle == CONST bop (op \#) x \langle\text{[]}\rangle

x \_u y == CONST bop (op \@) x y

\text{last}_u(xs) == CONST uop CONST last xs

\text{front}_u(xs) == CONST uop CONST butlast xs

\text{head}_u(xs) == CONST uop CONST hd xs

21
tail_a(xs) == CONST uop CONST tl xs
drop_a(n, xs) == CONST bop CONST drop n xs
take_a(n, xs) == CONST bop CONST take n xs
elems_a(xs) == CONST uop CONST set xs
sorted_a(xs) == CONST uop CONST sorted xs
distinct_a(xs) == CONST uop CONST distinct xs
xs | u A == CONST bop CONST seq-filter xs A
A \ u n xs == CONST bop (op \ i) A xs
\(n..k\) == CONST bop CONST upto n k
\(n..<k\) == CONST bop CONST upto n k
map_a f xs == CONST bop CONST map f xs
zip_a xs ys == CONST bop CONST zip xs ys
iter[n](P) == CONST uop (CONST tr-iter n) P

**syntax** — Sets

- **-ufinite**: logic ⇒ logic (finite_a(''))
- **-ufinset**: ('a set, 'a) uexpr {()_a}
- **-uset**: args ⇒ ('a set, 'a) uexpr {()_a}
- **-union**: ('a set, 'a) uexpr ⇒ ('a set, 'a) uexpr ⇒ ('a set, 'a) uexpr (infixl \_u 65)
- **-inter**: ('a set, 'a) uexpr ⇒ ('a set, 'a) uexpr ⇒ ('a set, 'a) uexpr (infixl \_i 70)
- **-umem**: ('a, 'a) uexpr ⇒ ('a set, 'a) uexpr ⇒ (bool, 'a) uexpr (infix \_mem 50)
- **-usubset**: ('a set, 'a) uexpr ⇒ ('a set, 'a) uexpr ⇒ (bool, 'a) uexpr (infix \_c 50)
- **-usubseteq**: ('a set, 'a) uexpr ⇒ ('a set, 'a) uexpr ⇒ (bool, 'a) uexpr (infix \_c 50)
- **-ucarrier**: type ⇒ logic (\[-\]_T [1000] 999)
- **-uid**: type ⇒ logic (id[_])
- **-uproduct**: logic ⇒ logic ⇒ logic (infixr \times_a 80)
- **-urelcomp**: logic ⇒ logic ⇒ logic (infixr \_u 75)

**translations**

finite_a(x) == CONST uop (CONST finite) x
\{()\}_a == {}\[\]
\{x, xs\}_a == CONST bop (CONST insert) x {xs}_a
\{()\}_a == CONST bop (CONST insert) x <{}>
A \_u B == CONST bop (op \ u) A B
A \_i B == CONST bop (op \ i) A B
x \_e_a A == CONST bop (op \ e) x A
A \_c_a B == CONST bop (op \ c) A B
f \_c_a g <= CONST bop (op \ c) f g
f \_c_a g <= CONST bop (op \ c) f g
A \_c_a B == CONST bop (op \ c) A B
f \_c_a g <= CONST bop (op \ c) f g
f \_c_a g <= CONST bop (op \ c) f g
P^' == CONST uop CONST converse P
\[\_\]_T == <CONST set-of TYPE(\_a)>
\_id_a == <CONST Id-on (CONST set-of TYPE(\_a))>
A \_x_a B == CONST bop CONST Product-Type.Times A B
A \_i_a B == CONST bop CONST relcomp A B

**syntax** — Partial functions

- **-umap-plus**: logic ⇒ logic ⇒ logic (infixl \_u 85)
- **-umap-minus**: logic ⇒ logic ⇒ logic (infixl \_u 85)

**translations**

f \_u g => (f :: ((-, -) pfun, -) uexpr) + g
We also lift the following functions on topological spaces for taking function limits, and describing continuity.

**Definition** \( \text{ulim-left} :: \text{a::t2-space where} \)

\[ \text{ulim-left} = (\lambda p. \text{Lim } (\text{at-left } p) f) \]

**Definition** \( \text{ulim-right} :: \text{a::t2-space where} \)

\[ \text{ulim-right} = (\lambda p. \text{Lim } (\text{at-right } p) f) \]

**Definition** \( \text{ucont-on} :: \text{a::topological-space where} \)

\[ \text{ucont-on} = (\lambda f. \text{continuous-on } A f) \]

**Syntax**

- \( \text{ulim-left} :: \text{id } \Rightarrow \text{logic } \Rightarrow \text{logic } \Rightarrow \text{logic } (\text{lim}_a (\text{\_ } \Rightarrow \text{\_}) (\text{\_})) \)
- \( \text{ulim-right} :: \text{id } \Rightarrow \text{logic } \Rightarrow \text{logic } \Rightarrow \text{logic } (\text{lim}_a (\text{\_ } \Rightarrow \text{\_}) (\text{\_})) \)
- \( \text{ucont-on} :: \text{logic } \Rightarrow \text{logic } \Rightarrow \text{logic } (\text{infix } \text{cont-on}_a 90) \)

**Translations**

\[ \text{lim}_a (x \Rightarrow p^\ast) (e) = \text{CONST } bop \text{ CONST } \text{ulim-left } p \ (\lambda x \cdot e) \]

\[ \text{lim}_a (x \Rightarrow p^\ast) (e) = \text{CONST } bop \text{ CONST } \text{ulim-right } p \ (\lambda x \cdot e) \]

\[ f \text{ cont-on}_a A = \text{CONST } bop \text{ CONST } \text{continuous-on } A f \]
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`
- `times-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]`: `[[x]]_e b = x`

by `(transfer, simp)`

**lemma** `var-ueval [ueval]`: `[[x]]_e b = get_x b`

by `(transfer, simp)`

**lemma** `uop-ueval [ueval]`: `[[uop f x]]_e b = f ([[x]]_e b)`

by `(transfer, simp)`

**lemma** `bop-ueval [ueval]`: `[[bop f x y]]_e b = f ([[x]]_e b) ([[y]]_e b)`

by `(transfer, simp)`

**lemma** `trop-ueval [ueval]`: `[[trop f x y z]]_e b = f ([[x]]_e b) ([[y]]_e b) ([[z]]_e b)`

by `(transfer, simp)`

**lemma** `qtop-ueval [ueval]`: `[[qtop f x y z w]]_e b = f ([[x]]_e b) ([[y]]_e b) ([[z]]_e b) ([[w]]_e b)`

by `(transfer, simp)`

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)`
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]: \(<\text{numeral } n> = \text{numeral } n \) 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]: \(<x> \times <y> = <x> \times <y> \) 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 ueval-eq-iff)
lemma lit-plus-appl [lit-norm]: $\langle op \rangle_a(x) = x + y$ by (simp add: ueval, transfer, simp)
lemma lit-minus-appl [lit-norm]: $\langle op \rangle_a(x) = x - y$ by (simp add: ueval, transfer, simp)
lemma lit-mul-appl [lit-norm]: $\langle op \rangle_a(x) = x * y$ by (simp add: ueval, transfer, simp)
lemma lit-divide-apply [lit-norm]: $\langle op \rangle_a(x) = x / y$ by (simp add: ueval, 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 $\theta = [\ ]_u$

$$1 = \langle i \rangle :: \theta'$$
$$\theta + \theta = \theta \quad \theta \quad \theta$$
$$\theta - \theta = \theta \quad \theta \quad \theta$$
$$\theta * \theta = \theta \quad \theta \quad \theta$$
$$\text{inverse } \theta = \theta \quad \theta \quad \theta$$
$$\theta \div \theta = \theta \quad \theta \quad \theta$$
$$\text{sgn } \theta = \theta \quad \theta \quad \theta$$

$$\theta = \text{Abs-uexpr} (\lambda b. \text{numeral} \theta \theta)$$
$$\theta = \text{numeral} \theta$$

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 (1::(int, 'a) uexpr) + "2" = 4 ↔ "3" = 4
apply (uexpr-simp) oops

26
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 $fv$ function.

Unrestriction was first defined in the work of Marcel Oliveira [19, 18] in his UTP mechanisation in 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 unrestricted, as several concepts will have this defined.

consts
  unrest :: 'a ⇒ 'b ⇒ bool

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

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

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 unrestricted conjectures using a simplification set.

named-theorems unrest
method unrest-tac = (simp add: 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 unrest-ucexpr :: ('a ⇒ 'α) ⇒ ('b, 'α) ucexpr ⇒ bool
is λ x e. ∀ b v. e (putx b v) = e b .

adhoc-overloading
  unrest unrest-ucexpr

lemma unrest-expr-alt-def:
  weak-lens x ⇒ (x ♯ P) = (∀ b b'. [P]_e (b ⊕L b' on x) = [P]_e b)
by (transfer, metis lens-override-def weak-lens.put-get)
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{lemma}
\textsf{unrest-var-comp [unrest]}:
\[ x \varnothing P; y \varnothing P \implies x; y \varnothing P \]
by (transfer, simp add: lens-defs)
\end{lemma}

\begin{lemma}
\textsf{unrest-svar [unrest]}: \(\& x \varnothing P \iff (x \varnothing P)\)
by (transfer, simp add: lens-defs)
\end{lemma}

No lens is restricted by a literal, since it returns the same value for any state binding.

\begin{lemma}
\textsf{unrest-lit [unrest]}: \(x \varnothing \ll v\)
by (transfer, simp)
\end{lemma}

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

\begin{lemma}
\textsf{unrest-sublens}:
\begin{proof}
\item \textbf{fixes} \(P :: (\alpha, \alpha)\) \textbf{uexpr}
\item \textbf{assumes} \(x \varnothing P \subseteq L x\)
\item \textbf{shows} \(y \varnothing P\)
\item \textbf{using} \textit{assms}
by (transfer, metis \textit{lens-equiv-def sublens-pres-mwb subset-sublens-unrest-uexpr.rep-eq})
\end{proof}
\end{lemma}

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

\begin{lemma}
\textsf{unrest-equiv}:
\begin{proof}
\item \textbf{fixes} \(P :: (\alpha, \alpha)\) \textbf{uexpr}
\item \textbf{assumes} \textit{mwb-lens} \(y x \approx_L x \varnothing P\)
\item \textbf{shows} \(y \varnothing P\)
by (metis \textit{assms lens-equiv-def sublens-pres-mwb sublens-put-put unrest-uexpr.rep-eq})
\end{proof}
\end{lemma}

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

\begin{lemma}
\textsf{bij-lens-unrest-all}:
\begin{proof}
\item \textbf{fixes} \(P :: (\alpha, \alpha)\) \textbf{uexpr}
\item \textbf{assumes} \textit{bij-lens} \(X X \varnothing P\)
\item \textbf{shows} \(\Sigma \varnothing P\)
\item \textbf{using} \textit{assms}
by blast
\end{proof}
\end{lemma}

\begin{lemma}
\textsf{bij-lens-unrest-all-eq}:
\begin{proof}
\item \textbf{fixes} \(P :: (\alpha, \alpha)\) \textbf{uexpr}
\item \textbf{assumes} \textit{bij-lens} \(X\)
\item \textbf{shows} \(\Sigma \varnothing P \iff (X \varnothing P)\)
by (meson \textit{assms lens-equiv-def sublens-equiv-id unrest-sublens})
\end{proof}
\end{lemma}

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

\begin{lemma}
\textsf{unrest-all-var}:
\begin{proof}
\item \textbf{fixes} \(e :: (\alpha, \alpha)\) \textbf{uexpr}
\end{proof}
\end{lemma}
We can split an unrestriction composed by lens plus

**Lemma** unrestr-plus-split:

fixes $P :: (\alpha', \alpha)$

assumes $x \triangleleft y \ \text{wlb-lens} \ x \ \text{wlb-lens} \ y$

shows unrestr $(x +_L y) \iff (x \# P) \land (y \# P)$

using assms

by (meson lens-plus-right-sublens lens-plus-unsublens unrest-sublens unrest-comp wlb-lens-unsublens)

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** unrestr-var [unrest!]: $[\ [ \text{wlb-lens} \ x; \ x \triangleright y \ ] \implies y \# \text{var} \ x$

by (transfer, auto)

**Lemma** unrestr-iuvar [unrest!]: $[ \ [ \text{wlb-lens} \ x; \ x \triangleright y \ ] \implies $\ y' \# \ $x$'$

by (simp add: unrestr-var)

**Lemma** unrestr-ouvar [unrest!]: $[ \ [ \text{wlb-lens} \ x; \ x \triangleright y \ ] \implies $\ y \# \ $x$'$

by (simp add: unrestr-var)

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

**Lemma** unrestr-iuvar-ouvar [unrest!]:

fixes $x :: (\alpha' \implies \alpha)$

assumes wlb-lens $y$

shows $\ [ \ [ x \# u; \ x \# v ] \implies x \# bop \ f \ u \ v$

by (metis prod.collapse unrest-uxpr.rep-eq var.rep-eq var-lookup-out var-update-in)

**Lemma** unrestr-ouvar-iuvar [unrest!]:

fixes $x :: (\alpha' \implies \alpha)$

assumes wlb-lens $y$

shows $\ [ \ [ x' \# y ; \ x \# y ] \implies x' \# \ y$

by (metis prod.collapse unrest-uxpr.rep-eq var.rep-eq var-lookup-in var-update-out)

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

**Lemma** unrestr-uop [unrest!]: $x \# e \implies x \# uop \ f \ e$

by (transfer, simp)

**Lemma** unrestr-bop [unrest!]: $[ x \# u; x \# v ] \implies x \# bop \ f \ u \ v$

by (transfer, simp)

**Lemma** unrestr-trop [unrest!]: $[ x \# u; x \# v; x \# w ] \implies x \# trop \ f \ u \ v \ w$

by (transfer, simp)

**Lemma** unrestr-qtop [unrest!]: $[ x \# u; x \# v; x \# w; x \# y ] \implies x \# qtop \ f \ u \ v \ w \ y$

by (transfer, simp)

For convenience, we also prove unrestriction rules for the bespoke operators on equality, numbers, arithmetic etc.
lemma unrest-eq [unrest]: \[ x \not< u; x \not< v ] \implies x \not< u = v \\
by (simp add: eq-upred-def, transfer, simp)

lemma unrest-zero [unrest]: x \not< 0 \\
by (simp add: unrest-lit zero-uxpr-def)

lemma unrest-one [unrest]: x \not< 1 \\
by (simp add: one-uxpr-def unrest-lit)

lemma unrest-numeral [unrest]: x \not< \text{numeral} n \\
by (simp add: numeral-uxpr-simp unrest-lit)

lemma unrest-sgn [unrest]: x \not< u \implies x \not< \text{sgn} u \\
by (simp add: sgn-uxpr-def unrest-uop)

lemma unrest-abs [unrest]: x \not< u \implies x \not< \text{abs} u \\
by (simp add: abs-uxpr-def unrest-uop)

lemma unrest-plus [unrest]: \[ x \not< u; x \not< v ] \implies x \not< u + v \\
by (simp add: plus-uxpr-def unrest)

lemma unrest-uminus [unrest]: x \not< u \implies x \not< - u \\
by (simp add: uminus-uxpr-def unrest)

lemma unrest-minus [unrest]: \[ x \not< u; x \not< v ] \implies x \not< u - v \\
by (simp add: minus-uxpr-def unrest)

lemma unrest-times [unrest]: \[ x \not< u; x \not< v ] \implies x \not< u \times v \\
by (simp add: times-uxpr-def unrest)

lemma unrest-divide [unrest]: \[ x \not< u; x \not< v ] \implies x \not< u / v \\
by (simp add: divide-uxpr-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]: \\
\[ \forall x. v \not< F x \] \implies v \not< (\lambda x. 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 => ‘b => bool
syntax

- \text{usedBy} :: \text{salpha} \Rightarrow \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \text{ (infix \, \sharp \, 20)}

translations

- \text{usedBy} \, x \, p = \text{CONST} \, \text{usedBy} \, x \, p
- \text{usedBy} \, (-\text{salphase} \, (-\text{salphamk} \, (x +_L \, y))) \, P \leq \text{usedBy} \, (x +_L \, y) \, P

lift-definition \text{usedBy-uxexpr} :: ('b \Rightarrow \alpha) \Rightarrow ('a, \alpha) \, \text{uxexpr} \Rightarrow \text{bool}

is \lambda \, e. \, (\forall \, b \, b'. \, e \, (b' +_L \, b \, \text{on} \, x) = e \, b).

adhoc-overloading \text{usedBy} \, \text{usedBy-uxexpr}

lemma \text{usedBy-lit} \, [\text{unrest}]: \, x \, \sharp \, \{v\}

by \, (\text{transfer, simp})

lemma \text{usedBy-sublens}:

- fixes \, P :: ('a, \alpha) \, \text{uxexpr}
- assumes \, x \, \sharp \, P \, x \subseteq_L \, y \, \text{vwb-lens} \, y
- shows \, y \, \sharp \, P

using \, \text{assms}

by \, (\text{transfer, auto, metis \text{lens-override-def} \text{lens-override-idem} \text{sublens-obs-get} \text{vwb-lens-mwb}})

lemma \text{usedBy-var} \, [\text{unrest}]: \, \sharp \, \text{\{\{\}\}} \, P \Rightarrow \& \, x \, \sharp \, P

by \, (\text{transfer, simp add: lens defs})

lemma \text{usedBy-lens-plus-1} \, [\text{unrest}]: \, x \, \sharp \, P \Rightarrow \, x; \, y \, \sharp \, P

by \, (\text{transfer, simp add: lens defs})

lemma \text{usedBy-lens-plus-2} \, [\text{unrest}]: \, \sharp \, \{x \, \triangleright \, y; \, y \, \sharp \, P \} \Rightarrow \, x; \, y \, \sharp \, P

by \, (\text{transfer, auto simp add: lens defs lens indep comm})

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

lemma \text{usedBy-indep-uses}:

- fixes \, P :: ('a, \alpha) \, \text{uxexpr}
- assumes \, x \, \sharp \, P \, x \triangleright \, y
- shows \, y \, \sharp \, P

using \, \text{assms by} \, (\text{transfer, auto, metis \text{lens indep get} \text{lens override def}})

lemma \text{usedBy-var} \, [\text{unrest}]:

- assumes \, \text{vwb-lens} \, x \, y \subseteq_L \, x
- shows \, x \, \sharp \, \text{var} \, y

using \, \text{assms}

by \, (\text{transfer, simp add: uexpr defs pr var def})

\quad (\text{metis \text{lens override def} \text{lens override idem} \text{sublens obs get} \text{vwb lens mwb}})

lemma \text{usedBy-uop} \, [\text{unrest}]: \, x \, \sharp \, e \Rightarrow \, x \, \sharp \, \text{uop} \, f \, e

by \, (\text{transfer, simp})

lemma \text{usedBy-bop} \, [\text{unrest}]: \, \sharp \, \{x \, \uop \, u; \, x \, \sharp \, v \} \Rightarrow \, x \, \sharp \, \text{bop} \, f \, u \, v

by \, (\text{transfer, simp})

lemma \text{usedBy-trop} \, [\text{unrest}]: \, \sharp \, \{x \, \uop \, u; \, x \, \sharp \, v; \, x \, \sharp \, w \} \Rightarrow \, x \, \sharp \, \text{trop} \, f \, u \, v \, w

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

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

lemma usedBy-eq [unrest]: \[
\begin{align*}
[x \sharp u; x \sharp v] \implies x \sharp u =_u v
\end{align*}
\]
by (simp add: eq-upred-def, transfer, simp)

lemma usedBy-zero [unrest]: \[
\begin{align*}
x \sharp 0
\end{align*}
\]
by (simp add: usedBy-lit zero-ueexpr-def)

lemma usedBy-one [unrest]: \[
\begin{align*}
x \sharp 1
\end{align*}
\]
by (simp add: one-ueexpr-def usedBy-lit)

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

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

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

lemma usedBy-plus [unrest]: \[
\begin{align*}
[x \sharp u; x \sharp v] \implies x \sharp u + v
\end{align*}
\]
by (simp add: plus-ueexpr-def unrest)

lemma usedBy-uminus [unrest]: \[
\begin{align*}
x \sharp u \implies x \sharp - u
\end{align*}
\]
by (simp add: uminus-ueexpr-def unrest)

lemma usedBy-minus [unrest]: \[
\begin{align*}
[x \sharp u; x \sharp v] \implies x \sharp u - v
\end{align*}
\]
by (simp add: minus-ueexpr-def unrest)

lemma usedBy-times [unrest]: \[
\begin{align*}
[x \sharp u; x \sharp v] \implies x \sharp u * v
\end{align*}
\]
by (simp add: times-ueexpr-def unrest)

lemma usedBy-divide [unrest]: \[
\begin{align*}
[x \sharp u; x \sharp v] \implies x \sharp u / v
\end{align*}
\]
by (simp add: divide-ueexpr-def unrest)

lemma usedBy-ulambda [unrest]: \[
\begin{align*}
\lambda x \cdot (\lambda x \cdot F x)
\end{align*}
\]
by (transfer, simp)

lemma unrest-var-sep [unrest]: \[
\begin{align*}
\forall x. v \sharp F x \implies v \sharp (\lambda x \cdot F x)
\end{align*}
\]
by (transfer, simp add: lens-defs)

end

6 Substitution

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

end

32
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 \( \sigma \vdash e \). We introduce a polymorphic constant that will be used to represent application of a substitution, and also a set of theorems to represent laws.

\[
\text{consts} \\
\text{usubst} :: 's \Rightarrow 'a \Rightarrow 'b \\
\text{(infixr } \uparrow \text{ 80)}
\]

\[
\text{named-theorems usubst}
\]

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.

\[
\text{type-synonym } (\alpha, \beta) psubst = \alpha \Rightarrow \beta
\]

\[
\text{type-synonym } \alpha usubst = \alpha \Rightarrow \alpha
\]

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 \( e \).

\[
\text{lift-definition subst} :: (\alpha, \beta) psubst \Rightarrow (\alpha, \beta) uexpr \Rightarrow (\alpha, \alpha) uexpr \text{ is } \\
\lambda \sigma e b. e (\sigma b) .
\]

\[
\text{adhoc-overloading}
\]

\[
\text{usubst subst}
\]

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

\[
\text{consts subst-upd} :: (\alpha, \beta) psubst \Rightarrow 'v \Rightarrow (\alpha, \alpha) uexpr \Rightarrow (\alpha, \beta) psubst
\]

The following function takes a substitution form state-space \( \alpha \) to \( \beta \), a lens with source \( \beta \) and view \( 'a \), and an expression over \( \alpha \) 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.

\[
\text{definition subst-upd-uvar} :: (\alpha, \beta) psubst \Rightarrow (\alpha, \beta) uexpr \Rightarrow (\alpha, \alpha) uexpr \Rightarrow (\alpha, \beta) psubst \text{ where } \\
\text{subst-upd-uvar } \sigma x v = (\lambda b. \text{put}_x (\sigma b) ([v]_e b))
\]

\[
\text{adhoc-overloading}
\]

\[
\text{subst-upd subst-upd-uvar}
\]

The next function looks up the expression associated with a variable in a substitution by use of the \texttt{get} lens function.

\[
\text{lift-definition usubst-lookup} :: (\alpha, \beta) psubst \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\alpha, \alpha) uexpr \Rightarrow (\alpha, \beta)
\text{ is } \\
\lambda \sigma x b. \text{get}_x (\sigma b) .
\]

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 \( \text{put} \) and the substitution commute.
**Definition**  
unrest-usubst :: ('a → 'a) ⇒ 'α usubst ⇒ bool  

**Where**  
unrest-usubst x σ = (∀ v. σ (putx g v) = putx (σ g) v)

**Adhoc-overloading**  
unrest-unrest-usubst

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

**Definition**  
cond-subst :: 'α usubst ⇒ (bool, 'α) uexpr ⇒ 'α usubst ⇒ 'α usubst  

**Where**  
cond-subst σ b g = (λ s. if [b]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 :: 'α usubst ⇒ ('a ⇒ 'a) ⇒ ('b ⇒ 'a) ⇒ 'α usubst ⇒ 'α usubst  
par-subst σ₁ A B 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  
  (- /⇒, /-)
- smaplets :: [smaplet, smaplets] ⇒ smaplets (-,-)
- SubstUpd :: ['m usubst, smaplets] ⇒ 'm usubst (-/-/-) [900,0] 900
- Subst :: smaplets ⇒ 'a → 'b  
  (λ [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 ms1 ms2) <= -SubstUpd (-Subst ms1) ms2
- SMaplets ms1 (-SMaplets ms2 ms3) <= -SMaplets (-SMaplets ms1 ms2) ms3
- subst P es vs => CONST subst (-psubst (CONST id) vs es) P
Thus we can write things like $\sigma(x \mapsto_s v)$ to update a variable $x$ in $\sigma$ with expression $v$, $[x \mapsto_s e, y \mapsto_s f]$ to construct a substitution with two variables, and finally $P[[v/x]]$, the traditional syntax.

We can now express deletion of a substitution maplet.

definition subst-del :: "'a usubst ⇒ ('a ⇒ 'a) ⇒ 'a usubst" where
subst-del $\sigma x = \sigma(x \mapsto_s s \& x)

6.3 Substitution Application Laws

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

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

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

lemma usubst-lookup-id [usubst]: $\langle id \rangle_x x = \text{var} x$
  by (transfer, simp)

lemma subst-upd-id-lam [usubst]: $\text{subst-upd} (\lambda x. x) x v = \text{subst-upd} id x v$
  by (simp add: id-def)

A substitution update naturally yields the given expression.

lemma usubst-lookup-upd [usubst]:
  assumes weak-lens $x$
  shows $\langle \sigma(x \mapsto_s s v) \rangle_x x = v$
  using assms
  by (simp add: subst-upd-uvar-def assms comp-def)

lemma usubst-lookup-upd-pr-var [usubst]:
  assumes weak-lens $x$
  shows $\langle \sigma(x \mapsto_s s v) \rangle_x (\text{pr-var} x) = v$
  using assms
  by (simp add: subst-upd-uvar-def pr-var-def assms comp-def)

Substitution update is idempotent.

lemma usubst-upd-idem [usubst]:
  assumes mwb-lens $x$
  shows $\sigma(x \mapsto_s s u, x \mapsto_s s v) = \sigma(x \mapsto_s s v)$
  by (simp add: subst-upd-uvar-def assms comp-def)

Substitution updates commute when the lenses are independent.

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

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

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

lemma swap-usubst-inj:
fixes \( x \, y :: ('a \mapsto 'a) \)
assumes \( \text{vwb-lens } x \, \text{vwb-lens } y \, x \mapsto y \)
shows \( \text{inj } [x \mapsto_s &y, y \mapsto_s &x] \)
proof (rule injI)
fix \( b_1 :: 'a \, \text{and } b_2 :: 'a \)
assume \( [x \mapsto_s &y, y \mapsto_s &x] \, b_1 = [x \mapsto_s &y, y \mapsto_s &x] \, b_2 \)
hence \( \forall a \, b. \, \text{put}_{y} (\text{put}_{x} b_1 ([&y]_e b_1)) ([&x]_e b_1) = \text{put}_{y} (\text{put}_{x} b_2 ([&y]_e b_2)) ([&x]_e b_2) \)
  by (auto simp add: subst-upd-uar-def)
then have \( (\forall a \, b. \, \text{get}_{x} (\text{put}_{y} a \, b) = \text{get}_{y} a) \wedge (\forall a \, b. \, \text{get}_{y} (\text{put}_{x} a \, b) = \text{get}_{y} a) \)
  by (simp add: assms(3) lens-indep_lens-put-irrev2 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]:
\( \text{vwb-lens } x \mapsto x \mapsto_s \text{var } x \)
apply (simp add: subst-upd-var-def)
apply (transfer)
apply (rule ext)
apply (auto)
done

lemma usubst-upd-pr-var-id [usubst]:
\( \text{vwb-lens } x \mapsto x \mapsto_s \text{var } (pr-var \, x) \)
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 \( x :: ('a \mapsto 'a) \)
shows \( \sigma(\$x' \mapsto_s v, \$x \mapsto_s u) = \sigma(\$x \mapsto_s u, \$x' \mapsto_s v) \)
using out-indep usubst-upd-comm by blast

lemma subst-upd-lens-plus [usubst]:
\( \text{subst-upd } \sigma (x \mapsto_L y) <(u,v)> = \sigma(y \mapsto_s <v>, x \mapsto_s <u>) \)
by (simp add: lens-defs uexpr-defs subst-upd-var-def, transfer, auto)

lemma subst-upd-in-lens-plus [usubst]:
\( \text{subst-upd } \sigma (\text{ivar } (x \mapsto_L y)) <(u,v)> = \sigma(\$y \mapsto_s <v>, \$x \mapsto_s <u>) \)
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]:
\[ \text{subst-upd } \sigma \ (\text{ovar } (x +_L y)) \langle (u, v) \rangle = \sigma(\{y' \mapsto_s v', x' \mapsto_s u\}) \]
by (simp add: lens-defs uexpr-defs subst-upd-uvar-def, transfer, auto simp add: prod.case-eq-if)

lemma usubst-lookup-upd-indep [usubst]:
assumes mwb-lens \( x \propto y \)
shows \( (\sigma(y \mapsto_s v))_x = (\sigma)_x \)
using assms
by (simp add: subst-upd-uvar-def, transfer, simp)

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

lemma usubst-apply-unrest [usubst]:
\[
\text{If a variable is unrestricted in an expression, then any substitution of that variable has no effect on the expression .}
\]

There follows various laws about deleting variables from a substitution.

lemma subst-del-id [usubst]:
\[
vwb-lens \ x \implies \text{id}_x = \text{id}
\]
by (simp add: subst-del-def subst-upd-uvar-def lens-indep-comm)

lemma subst-del-upd-same [usubst]:
\[
mwb-lens \ x \implies \sigma(x \mapsto_s v)_x = \sigma \_x
\]
by (simp add: subst-del-def subst-upd-uvar-def pr-var-def)

lemma subst-del-upd-diff [usubst]:
\[
\sigma(x \mapsto_s v)_x = (\sigma \_x)(y \mapsto_s v)
\]
by (simp add: subst-del-def subst-upd-uvar-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]: \( x \not\propto P \implies \sigma(x \mapsto_s v) \uparrow P = \sigma \uparrow P \)
by (simp add: subst-upd-uvar-def, transfer, auto)

lemma subst-unrest-2 [usubst]:
\[
\text{fixes } P :: ('a, 'a) uexpr
\text{assumes } x \not\propto P \text{ x } x \propto y \text{ v}
\text{shows } \sigma(x \mapsto u, y \mapsto_v) \uparrow P = \sigma(y \mapsto_v) \uparrow P
\text{using assms}
by (simp add: subst-upd-uvar-def, transfer, auto, metis lens-indep.lens-put-comm)

lemma subst-unrest-3 [usubst]:
\[
\text{fixes } P :: ('a, 'a) uexpr
\text{assumes } x \not\propto P \text{ x } x \propto y \text{ x } y \propto z \text{ w}
\text{shows } \sigma(x \mapsto u, y \mapsto_v, z \mapsto_w) \uparrow P = \sigma(y \mapsto_v, z \mapsto_w) \uparrow P
\text{using assms}
by (simp add: subst-upd-uvar-def, transfer, auto, metis (no-types, hide-lams) lens-indep-comm)

lemma subst-unrest-4 [usubst]:
\[
\text{fixes } P :: ('a, 'a) uexpr
\text{assumes } x \not\propto P \text{ x } x \propto y \text{ x } z \propto u \text{ w}
\text{shows } \sigma(x \mapsto e, y \mapsto f, z \mapsto g, u \mapsto h) \uparrow P = \sigma(y \mapsto f, z \mapsto g, u \mapsto h) \uparrow P
\text{using assms}
by (simp add: subst-upd-uvar-def, transfer, auto, metis (no-types, hide-lams) lens-indep-comm)

37
lemma subst-unrest-5 [usubst]:
fixes P :: ('a, 'u) uexpr
assumes x ≠ P x ≡ y ≡ z ≡ u ≡ v
shows σ(x → y e, y → z f, z → u g, u → v h, v → i) ⊢ P = σ(y → z f, z → u g, u → v h, v → i) ⊢ P
using assumptions
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 o g(x → y v) = (σ o g)(x → y v)
by (simp add: subst-upd-uvar_def, transfer, auto simp add: unrest-usubst_def)

Any substitution is a monotonic function.

lemma subst-mono: mono (subst 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 ⊢ ϭ v = ϭ v
by (transfer, simp)

lemma subst-var [usubst]: σ ⊢ var x = (σ)x x
by (transfer, simp)

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

lemma unrest-usubst-del [unrest]: [ vwb-lens x; x ≠ (σ)x x; x ≠ σ - x ] ⊢ x ≠ (σ ⊢ P)
by (simp add: subst-del_def subst-upd-uvar_def unrest-uexpr_def unrest-usubst_def subst_rep_eq usubst_lookup_rep_eq
(metis vwb_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]: \( \sigma \uparrow (x + y) = \sigma \uparrow x + \sigma \uparrow y \)
by (simp add: plus-uexpr-def subst-bop)

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

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

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

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

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

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

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

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

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

lemma subst-eq-upred [usubst]: \( \sigma \uparrow (x =_u y) = (\sigma \uparrow x =_u \sigma \uparrow y) \)
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]: \( \sigma \uparrow \rho \uparrow e = (\rho \circ \sigma) \uparrow e \)
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 \( \varphi(x \mapsto_s v) \circ \sigma = (\varphi \circ \sigma)(x \mapsto_s \sigma \uparrow v) \)
by (rule ext, simp add: uexpr-defs subst-upd-uvar-def, transfer, simp)

lemma subst-singleton:
  fixes \( x :: ('a \Rightarrow 'a) \)
  assumes \( x \not\in \sigma \)
  shows \( \sigma(x \mapsto_s v) \uparrow P = (\sigma \uparrow P)[v/x] \)
using assms
by (simp add: usubst)

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

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-ntp]: var-name-ord x y = True

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

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

A fact of the form \( x \prec_v y \) 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 \( x \triangleright y \prec_v x \)
shows \( \sigma(x \mapsto u, y \mapsto v) = \sigma(y \mapsto v, x \mapsto u) \)
by (simp add: assms(1) usubst-upd-comm)

**lemma** var-name-order-comp-outer [usubst]: \( x \prec_v y \implies x:a \prec_v y:b \)
by (simp add: var-name-ord-def)

**lemma** var-name-order-comp-inner [usubst]: \( a \prec_v b \implies x:a \prec_v x:b \)
by (simp add: var-name-ord-def)

**lemma** var-name-ord-pr-var-1 [usubst]: \( x \prec_v y \implies &x \prec_v y \)
by (simp add: var-name-ord-def)

**lemma** var-name-ord-pr-var-2 [usubst]: \( x \prec_v y \implies x \prec_v &y \)
by (simp add: var-name-ord-def)

6.6 Unrestriction laws

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

**lemma** unrest-usubst-single [unrest]:
\( mwb-lens x; x \not\triangleright P \[x/v\] \implies x \not\triangleright P[\sigma/x] \)
by (simp add: transfer auto simp add: subst-upd-uvar-def unrest-uexpr-def)

**lemma** unrest-usubst-id [unrest]:
\( mwb-lens x \implies x \not\triangleright id \)
by (simp add: unrest-usubst-def)

**lemma** unrest-usubst-upd [unrest]:
\( mwb-lens x; x \not\triangleright \sigma; x \not\triangleright v \implies x \not\triangleright \sigma(y \mapsto v) \)

**lemma** unrest-subst [unrest]:
\( mwb-lens x; x \not\triangleright \sigma \implies x \not\triangleright (\sigma \dagger P) \)
by (transfer simp add: unrest-usubst-def)
6.7 Conditional Substitution Laws

**Lemma usubst-cond-upd-1 [usubst]**:
\[
\sigma(x \mapsto u) \prec b \triangleright_{\gamma} g(x \mapsto v) = (\sigma \prec b \triangleright_{\gamma} g)(x \mapsto, u \prec b \triangleright v)
\]
by (simp add: cond-subst-def subst-upd-uvar-def, transfer, auto)

**Lemma usubst-cond-upd-2 [usubst]**:
\[
\llbracket \text{vwb-lens } x; x \nleq \sigma \rrbracket \Rightarrow \sigma(x \mapsto, u) \prec b \triangleright_{\gamma} g(x \mapsto v) = (\sigma \prec b \triangleright_{\gamma} g)(x \mapsto, u \prec b \triangleright \& x)
\]
by (simp add: cond-subst-def subst-upd-uvar-def unrest-usubst-def, transfer)
\[
\text{metis (full-types, hide-lams) id-apply pr-var-def subst-upd-uvar-def subst-upd-pr-var-id var.rep-eq}
\]

**Lemma usubst-cond-upd-3 [usubst]**:
\[
\llbracket \text{vwb-lens } x; x \nleq \sigma \rrbracket \Rightarrow \sigma \prec b \triangleright_{\gamma} g(x \mapsto v) = (\sigma \prec b \triangleright_{\gamma} g)(x \mapsto \& x \prec b \triangleright v)
\]
by (simp add: cond-subst-def subst-upd-uvar-def unrest-usubst-def, transfer)
\[
\text{metis (full-types, hide-lams) id-apply pr-var-def subst-upd-uvar-def subst-upd-pr-var-id var.rep-eq}
\]

**Lemma usubst-cond-id [usubst]**:
\[
id \prec b \triangleright_{\gamma} id = id
\]
by (auto simp add: cond-subst-def)

6.8 Parallel Substitution Laws

**Lemma par-subst-id [usubst]**:
\[
\llbracket \text{vwb-lens } A; \text{vwb-lens } B \rrbracket \Rightarrow id \llbracket A|B \rrbracket = id
\]
by (simp add: par-subst-def lens-overide-idem id-def)

**Lemma par-subst-left-empty [usubst]**:
\[
\llbracket \text{vwb-lens } A \rrbracket \Rightarrow \sigma \llbracket \emptyset|A \rrbracket = id \llbracket \emptyset|A \rrbracket
\]
by (simp add: par-subst-def pr-var-def)

**Lemma par-subst-right-empty [usubst]**:
\[
\llbracket \text{vwb-lens } A \rrbracket \Rightarrow \sigma \llbracket A|\emptyset \rrbracket = \sigma \llbracket A|\emptyset \rrbracket id
\]
by (simp add: par-subst-def pr-var-def)

**Lemma par-subst-comm**:
\[
\llbracket A \bowtie B \rrbracket \Rightarrow \sigma \llbracket A|B \rrbracket = \sigma \llbracket B|A \rrbracket
\]
by (simp add: par-subst-def lens-overide-def lens-indep-comm)

**Lemma par-subst-upd-left-in [usubst]**:
\[
\llbracket \text{vwb-lens } A; A \bowtie B; \llbracket x \nsubseteq L A \rrbracket \Rightarrow \sigma(x \mapsto v) \llbracket A|B \rrbracket = (\sigma \llbracket A|B \rrbracket)(x \mapsto v)
\]
by (simp add: par-subst-def subst-upd-uvar-def lens-overide-put-right-in)
\[
\text{metis (full-types, hide-lams) lens-overide-def substlens-pres-indep}
\]

**Lemma par-subst-upd-left-out [usubst]**:
\[
\llbracket \text{vwb-lens } A; x \bowtie A \rrbracket \Rightarrow \sigma(x \mapsto v) \llbracket A|B \rrbracket = (\sigma \llbracket A|B \rrbracket)(x \mapsto v)
\]
by (simp add: par-subst-def subst-upd-uvar-def lens-overide-put-right-out)

**Lemma par-subst-upd-right-in [usubst]**:
\[
\llbracket \text{vwb-lens } B; A \bowtie B; x \nsubseteq L B \rrbracket \Rightarrow \sigma \llbracket A|B \rrbracket = (\sigma \llbracket A|B \rrbracket)(x \mapsto v)
\]
using lens-indep-sym par-subst-comm par-subst-upd-left-in by fastforce

**Lemma par-subst-upd-right-out [usubst]**:
\[
\llbracket \text{vwb-lens } B; A \bowtie B; x \nsubseteq L B \rrbracket \Rightarrow \sigma \llbracket A|B \rrbracket = (\sigma \llbracket A|B \rrbracket)(x \mapsto v)
\]
by (simp add: par-subst-comm par-subst-upd-left-out)

end
7 UTP Tactics

theory utp-tactics
  imports utp-expr utp-unrest utp-usedby
keywords update-uexpr-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])?;
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**

\[\text{uexpr-transfer-laws uexpr transfer laws}\]

**declare**

\[\text{uexpr-eq-iff [uexpr-transfer-laws]}\]

**named-theorems**

\[\text{uexpr-transfer-extra extra simplifications for uexpr transfer}\]

**declare**

\[\text{unrest-uexpr rep-eq [uexpr-transfer-extra]}\]

\[\text{utp-expr.numeral-uexpr-rep-eq [uexpr-transfer-extra]}\]

\[\text{utp-expr.less-eq-uexpr-rep-eq [uexpr-transfer-extra]}\]

\[\text{Abs-uexpr-inverse [simplified, uexpr-transfer-extra]}\]

\[\text{Rep-uexpr-inverse [uexpr-transfer-extra]}\]

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

**method**

\[\text{fast-uexpr-transfer = (simp add: uexpr-transfer-laws uexpr-rep-eq-thms uexpr-transfer-extra)}\]

**7.4 Interpretation**

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

**method**

\[\text{uexpr-interp-tac = (simp add: lens-interp-laws)}\]

**7.5 User Tactics**

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

**method**

\[\text{utp-simp-tac = (clarsimp)}\]

**method**

\[\text{utp-auto-tac = (((clarsimp); auto)}\]

**method**

\[\text{utp-blast-tac = (((clarsimp); blast)}\]

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

**ML-file**

\texttt{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**

\[\text{pred-simp} = \langle\langle (Scan.lift UTP-Tactics.scan-args) >> (fn args => fn ctx => let val prove-tac = Basic-Tactics.utp-simp-tac in (UTP-Tactics.inst-gen-pred-tac args prove-tac ctx) end); \rangle\rangle\]

**method-setup**

\[\text{rel-simp} = \langle\langle\rangle\rangle\]
(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)u] = (P x y)[x→e][y→f]
  by (rel-auto)

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

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

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

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

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

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

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

lemma msubst-bop-2 [usubst]: (bop f (v x y) (w x y))[(x,y)→u] = bop f ((v x y)[(x,y)→u]) ((w x y)[(x,y)→u])
  by (pred-simp, pred-simp)
lemma msubst-var [usubst]:
(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]: [ \( \forall v. x \not\in P(v); x \not\in k \) ] \( \Rightarrow \)
\( x \not\in P(v)[v \mapsto k] \)
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)
We set up two versions of each of the quantifiers: \texttt{uex} / \texttt{uall} and \texttt{ushEx} / \texttt{ushAll}. The former pair allows quantification of UTP variables, whilst the latter allows quantification of HOL variables in concert with the literal expression constructor \texttt{≪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).

### 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 \texttt{refine} that will add the refinement operator syntax to the HOL partial order class.

```plaintext
class refine = order

abbreviation refineBy :: 'a::refine ⇒ 'a ⇒ bool (infix ⊑ 50) where
```

---

**Syntax**

- \texttt{idt-el :: idt ⇒ idt-list (-)}
- \texttt{idt-list :: idt ⇒ idt-list ⇒ idt-list ((-/-) [0, 1])}
- \texttt{uex :: salpha ⇒ logic ⇒ logic (∃ - · [0, 10] 10)}
- \texttt{uall :: salpha ⇒ logic ⇒ logic (∀ - · [0, 10] 10)}
- \texttt{ushEx :: pttrn ⇒ logic ⇒ logic (∃ - · [0, 10] 10)}
- \texttt{ushAll :: pttrn ⇒ logic ⇒ logic (∀ - · [0, 10] 10)}
- \texttt{ushBEx :: pttrn ⇒ logic ⇒ logic (∃ - · [0, 0, 10] 10)}
- \texttt{ushBAll :: pttrn ⇒ logic ⇒ logic (∀ - · [0, 0, 10] 10)}
- \texttt{ushGAll :: pttrn ⇒ logic ⇒ logic (∀ - · [0, 0, 10] 10)}
- \texttt{ushLtAll :: pttrn ⇒ logic ⇒ logic (∀ - · [0, 0, 10] 10)}
- \texttt{uvar-res :: logic ⇒ salpha ⇒ logic (infix \_ \_ 90)}
\[P \subseteq Q \equiv \text{less-eq} Q P\]

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

```plaintext
purge-notation Lattices.inf (infixl \ \cap \ \ 70)
notation Lattices.inf (infixl \ \sqcup \ \ 70)
purge-notation Lattices.sup (infixl \ \cap \ \ 65)
notation Lattices.sup (infixl \ \sqcup \ \ 65)
```

We trivially instantiate our refinement class

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

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

```plaintext
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.

```plaintext
instantiation uexpr :: (lattice, type) lattice
begin
lift-definition sup-uexpr :: (\forall a \ b) uexpr \Rightarrow (\forall a \ b) uexpr \Rightarrow (\forall a \ b) uexpr
is \\land P A. Lattices.sup (P A) (Q A).

lift-definition inf-uexpr :: (\forall a \ b) uexpr \Rightarrow (\forall a \ b) uexpr \Rightarrow (\forall a \ b) uexpr
is \\land P A. Lattices.inf (P A) (Q A).

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

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

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

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

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

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

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

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

From the complete lattice, we can also define and give syntax for the fixed-point operators. Like the lattice operators, these are reversed in UTP.
syntax
  -mu :: pttrn ⇒ logic ⇒ logic (μ - • [0, 10] 10)
  -nu :: pttrn ⇒ logic ⇒ logic (ν - • [0, 10] 10)
notation gfp (μ)
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 (\A [-900] 900) where
\A A \equiv \bigcap A

**abbreviation** Disj-upred :: 'α upred set ⇒ 'α upred (\V [-900] 900) where
\V A \equiv \bigcup A

**notation**

- conj-upred (infixr \A p 35) and
- disj-upred (infixr \V p 30)

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

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

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

**syntax**

-USup :: ptrtn ⇒ logic ⇒ logic (\A \dots [0, 10] 10)
-USup :: ptrtn ⇒ logic ⇒ logic (\V \dots [0, 10] 10)
-USup-mem :: ptrtn ⇒ logic ⇒ logic ⇒ logic (\A \in \dots [0, 10] 10)
-USUP :: ptrtn ⇒ logic ⇒ logic ⇒ logic (\V \in \dots [0, 10] 10)
-UInf :: ptrtn ⇒ logic ⇒ logic (\P \dots [0, 10] 10)
-UInf-mem :: ptrtn ⇒ logic ⇒ logic ⇒ logic (\P \in \dots [0, 10] 10)
-UINF :: ptrtn ⇒ logic ⇒ logic ⇒ logic (\V \in \dots [0, 10] 10)
-UINF :: ptrtn ⇒ logic ⇒ logic ⇒ logic (\P \in \dots [0, 10] 10)

**translations**

\[ x \mid P \cdot F \Rightarrow \text{CONST UINF} \ (\lambda \ x. \ P) \ (\lambda \ x. \ F) \]
\[ x \cdot F \ = \equiv \ \bigcap x \mid \text{true} \cdot F \]
\[ x \cdot F \ = \equiv \ \bigcap x \mid \text{true} \cdot F \]
\[ x \in A \cdot F \Rightarrow \bigcap x \mid <x> \in_u <A> \cdot F \]
We also define the other predicate operators

\[ \begin{align*}
& \text{lift-definition } \text{impl} : \alpha \text{ upred } \Rightarrow \alpha \text{ upred } \Rightarrow \alpha \text{ upred is} \\
& \quad \lambda P Q A. P A \rightarrow Q A . \\
& \text{lift-definition } \text{iff-upred} : \alpha \text{ upred } \Rightarrow \alpha \text{ upred } \Rightarrow \alpha \text{ upred is} \\
& \quad \lambda P Q A. P A \leftrightarrow Q A . \\
& \text{lift-definition } \text{ex} : (\alpha \Rightarrow \alpha) \Rightarrow \alpha \text{ upred } \Rightarrow \alpha \text{ upred is} \\
& \quad \lambda x P b. (\exists v. P(\text{put} x b v)) . \\
& \text{lift-definition } \text{shEx} : \beta \Rightarrow \alpha \text{ upred } \Rightarrow \alpha \text{ upred is} \\
& \quad \lambda P A. \exists x . (P x) A . \\
& \text{lift-definition } \text{all} : (\alpha \Rightarrow \alpha) \Rightarrow \alpha \text{ upred } \Rightarrow \alpha \text{ upred is} \\
& \quad \lambda x P b. (\forall v. P(\text{put} x b v)) . \\
& \text{lift-definition } \text{shAll} : \beta \Rightarrow \alpha \text{ upred } \Rightarrow \alpha \text{ upred is} \\
& \quad \lambda P A. \forall x . (P x) A .
\end{align*} \]

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

\[ \begin{align*}
& \text{lift-definition } \text{var-res} : \alpha \text{ upred } \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha \text{ upred } \Rightarrow \alpha \text{ upred is} \\
& \quad \lambda P x b. \exists b \prime . P(b \prime \oplus L b \text{ on } x) .
\end{align*} \]

translations

- \text{uvar-res} P a \equiv \text{CONST var-res} P a

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

\[ \begin{align*}
& \text{lift-definition } \text{closure} : \alpha \text{ upred } \Rightarrow \alpha \text{ upred } ([\cdot]_u) \text{ is} \\
& \quad \lambda P A. \forall A'. P A' .
\end{align*} \]

\[ \text{lift-definition } \text{taut} : \alpha \text{ upred } \Rightarrow \text{bool } (\cdot') \]

is \( \lambda P. \forall A. P A .\)

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


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

adhoc-overloading

\[ \begin{align*}
& \text{uttrue true-upred and} \\
& \text{utfalse false-upred and} \\
& \text{unot not-upred and}
\end{align*} \]
uconj conj-upred and
udisj 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 ≠ₜ 50)
-umem :: ('a, 'a) uexpr ⇒ ('a set, 'a) uexpr ⇒ (bool, 'a) uexpr (infix ≠ₜ 50)

translations
x ≠ₜ y == CONST unot (x =ₜ y)
x /∈ₜ A == CONST unot (CONST bop (op ∈) 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 Unrestriction Laws

lemma unrest-allE:
[ Σ x; P; P = true ⇒ Q; P = false ⇒ Q ] ⇒ Q
by (pred-auto)
lemma unrest-true [unrest]; x /∈ₜ true
by (pred-auto)
lemma unrest-false [unrest]; x /∈ₜ false
by (pred-auto)
lemma unrest-conj [unrest]; [ x /∈ₜ (P :: 'a upred); x /∈ₜ Q ] ⇒ x /∈ₜ P ∧ Q
by (pred-auto)
lemma unrest-disj [unrest]; [ x /∈ₜ (P :: 'a upred); x /∈ₜ Q ] ⇒ x /∈ₜ P ∨ Q
by (pred-auto)
lemma unrest-UINF [unrest]:
\[ \llbracket (\bigwedge i. x \not\in P(i)); (\bigwedge i. x \not\in Q(i)) \rrbracket \implies x \not\in (\bigcap i. P(i) \land Q(i)) \]
by (pred-auto)

lemma unrest-USUP [unrest]:
\[ \llbracket (\bigwedge i. x \not\in P(i)); (\bigwedge i. x \not\in Q(i)) \rrbracket \implies x \not\in (\bigcup i. P(i) \land Q(i)) \]
by (pred-auto)

lemma unrest-UINF-mem [unrest]:
\[ \llbracket (\bigwedge i. i \in A \implies x \not\in P(i)) \rrbracket \implies x \not\in (\bigcap i \in A \cdot P(i)) \]
by (pred-simp, metis)

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

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

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

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

The sublens proviso can be thought of as membership below.

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

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

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

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

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

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

**lemma** unrest-var-res-in [unrest]:
assumes \( \text{mwb-lens } x y \subseteq L \) \( x y \not\in P \)
shows \( y \not\in (P \mid_v x) \)
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 \( \forall y. x \not\in P(y) \)
shows \( x \not\in (\exists y \cdot P(y)) \)
using assms by (pred-auto)

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

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

### 9.4 Used-by laws

**lemma** usedBy-not [unrest]:
\( \lceil x \not\in P \rceil \Rightarrow x \not\in (\lnot P) \)
by (pred-simp)

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

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

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

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

### 9.5 Substitution Laws

Substitution is monotone

**lemma** subst-mono: \( P \subseteq Q \Rightarrow (\sigma \uparrow P) \subseteq (\sigma \uparrow Q) \)
by (pred-auto)
lemma subst-true [usubst]: $\sigma \uparrow \text{true} = \text{true}$
by (pred-auto)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma subst-ex-unrest [usubst]:
\[ x \not\in \sigma \implies \sigma \vdash (\exists x \cdot P) = (\exists x \cdot \sigma \vdash P) \]
by (pred-auto)

lemma subst-all-same [usubst]:
\[ \text{subst-ex-unrest unrest-all-in} \]
by (simp add: id-subst subst-unrest unrest-all-in)

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

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

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

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

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

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

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

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

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

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

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

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

lemma msubst-shAll-2 [usubst]:
\( (\forall x \cdot P x y z)[(y,z)\rightarrow v] = (\forall x \cdot (P x y z)[(y,z)\rightarrow v]) \)

57
by (pred-auto)+

end

10 UTP Events

theory utp-event
imports utp-pred
begin

10.1 Events

Events of some type ′ϑ are just the elements of that type.

type-synonym ′ϑ event = ′ϑ

10.2 Channels

Typed channels are modelled as functions. Below, ′a determines the channel type and ′ϑ 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?

type-synonym (′a, ′ϑ) chan = ′a ⇒ ′ϑ event

A downside of the approach is that the event type ′ϑ 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 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 [?])

definition chan-type :: (′a, ′ϑ) chan ⇒ ′a set (δ_u) where
[upred-defs]: δ_u c = UNIV

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

definition chan-apply ::
(′a, ′ϑ) chan ⇒ (′a, ′a) uexpr ⇒ (′ϑ event, ′a) uexpr ((′-/-·)u) where
[upred-defs]: (c·e)_u = <c·(e)>_a

lemma unrest-chan-apply [unrest]: x # e ⇒ x # (c·e)_u
by (rel-auto)

lemma usubst-chan-apply [usubst]: σ † (c·v)_u = (c·σ † v)_u
by (rel-auto)
lemma msubst-event [usubst]:
\((c \cdot v \, x)\, u\, [x \mapsto u] = (c \cdot (v \, x)\, [x \mapsto u] )\, u\)
by (pred-simp)

lemma msubst-event-2 [usubst]:
\((c \cdot v \, x \, y)\, u\, [(x, y) \mapsto u] = (c \cdot (v \, x \, y)\, [(x, y) \mapsto u] )\, u\)
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 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, 'beta) uexpr \Rightarrow ('beta, 'alpha) lens \Rightarrow ('a, 'alpha) uexpr (infixr \oplus_p 95)
is \lambda P \, x. \, P \, (get_x \, 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: \((P \oplus_p a) \oplus_p b = P \oplus_p (a \, \circ_L \, b)\)
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 [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 [simp]:** $\langle v \rangle \oplus_p a = \langle v \rangle$

by (pred-auto)

**Lemma aext-zero [simp]:** $0 \oplus_p a = 0$

by (pred-auto)

**Lemma aext-one [simp]:** $1 \oplus_p a = 1$

by (pred-auto)

**Lemma aext-numeral [simp]:** numeral $n \oplus_p a = numeral n$

by (pred-auto)

**Lemma aext-true [simp]:** true $\oplus_p a = true$

by (pred-auto)

**Lemma aext-false [simp]:** false $\oplus_p a = false$

by (pred-auto)

**Lemma aext-not [alpha]:** $(\neg P) \oplus_p x = (\neg (P \oplus_p x))$

by (pred-auto)

**Lemma aext-and [alpha]:** $(P \land Q) \oplus_p x = (P \oplus_p x \land Q \oplus_p x)$

by (pred-auto)

**Lemma aext-or [alpha]:** $(P \lor Q) \oplus_p x = (P \oplus_p x \lor Q \oplus_p x)$

by (pred-auto)

**Lemma aext-imp [alpha]:** $(P \Rightarrow Q) \oplus_p x = (P \oplus_p x \Rightarrow Q \oplus_p x)$

by (pred-auto)

**Lemma aext-iff [alpha]:** $(P \Leftrightarrow Q) \oplus_p x = (P \oplus_p x \Leftrightarrow Q \oplus_p x)$

by (pred-auto)

**Lemma aext-shAll [alpha]:** $(\forall x \cdot P(x)) \oplus_p a = (\forall x \cdot P(x) \oplus_p a)$

by (pred-auto)

**Lemma aext-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]:** $uop f u \oplus_p a = uop f (u \oplus_p a)$

by (pred-auto)

**Lemma aext-bop [alpha]:** $bop f u v \oplus_p a = bop f (u \oplus_p a) (v \oplus_p a)$

by (pred-auto)

**Lemma aext-trop [alpha]:** $trop f u v w \oplus_p a = trop f (u \oplus_p a) (v \oplus_p a) (w \oplus_p a)$

by (pred-auto)
Lemma aext-qtop [alpha]:  
\[ qtop f u v w x \oplus p = qtop f (u \oplus_p a) (v \oplus_p a) (w \oplus_p a) (x \oplus_p a) \]
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]:  
\[ (x \ast y) \oplus_p a = (x \oplus_p a) \ast (y \oplus_p a) \]
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]:  
\[ (\lambda x \cdot P(x)) \oplus_p a = (\lambda x \cdot P(x) \oplus_p a) \]
by (pred-auto)

Alphabet extension is monotonic and continuous.

Lemma aext-mono:  
\[ P \sqsubseteq Q \Rightarrow P \oplus_p a \sqsubseteq Q \oplus_p a \]
by (pred-auto)

Lemma aext-cont [alpha]:  
\[ vwb-lens a \Rightarrow (\bigcap A) \oplus_p a = (\bigcap P \in A. P \oplus_p a) \]
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]:  
\[ \text{mwb-lens a; } x \not\in p \Rightarrow \text{unrest } (x ; L a) (p \oplus_p a) \]
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]:  
\[ a \not\gg b \Rightarrow b \not\in (p \oplus_p a) \]
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.

\[ \text{lift-definition arestr :: } ('a, 'a) \text{ uexpr } \Rightarrow ('\beta, '\alpha) \text{ lens } \Rightarrow ('a, '\beta) \text{ uexpr (infixr } \varepsilon) \]

\[ \text{is } \lambda P \times b. \ P \ (\text{create}_x \ b) . \]

update-uexpr-rep-eq-thms

\[ \text{lemma arestr-id [simp]: } P \varepsilon 1_L = P \]
by (pred-auto)

\[ \text{lemma arestr-aext [simp]: mwb-lens a } \Rightarrow (\varepsilon (P \oplus_p a)) \varepsilon a = P \]
by (pred-auto)

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

\[ \text{lemma aext-arestr [alpha]:} \]
assumes mwb-lens a bij-lens (a +L b) a \bowtie b \not\bowtie P
shows (P \varepsilon a) \oplus_p a = P
proof -
from assms(2) have 1_L \subseteq_L a +L b
by (simp add: bij-lens-eq-id lens-eq-id)
with assms(1,3,4) show ?thesis
apply (auto simp add: id-lens-eq lens-plus-eq lens-comp-eq prod.case-eq-if)
apply (pred simp)
done
qed

\[ \text{lemma arestr-lit [simp]: } \langle v \rangle \varepsilon a = \langle v \rangle \]
by (pred-auto)

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

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

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

\[ \text{lemma arestr-var [alpha]:} \]
var x \varepsilon a = var (x /L a)
by (pred-auto)

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

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

\[ \text{lemma arestr-not [alpha]: } (\neg P) \varepsilon a = (\neg (P \varepsilon a)) \]
by (pred-auto)
lemma arestr-and [alpha]: \((P \land Q)|_x = (P|_x \land Q|_x)\)
by (pred-auto)

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

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

11.4 Predicate Alphabet Restriction

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

definition upred-ares :: 'α upred ⇒ ('β ⇒ 'α) ⇒ 'β upred
where [upred-defs]: upred-ares P a = (P|v a)↾e a

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

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

lemma upred-aext-ares [alpha]:
vw-lens a ⇒ P ⊠p a ↦p a = P
by (pred-auto)

lemma upred-ares-aext [alpha]:
a ♭ P ⇒ (P ↦p a) ⊠p a = P
by (pred-auto)

lemma upred-arestr-lit [simp]: ≪v≫ ↦p a = ≪v≫
by (pred-auto)

lemma upred-arestr-true [simp]: true ↦p a = true
by (pred-auto)

lemma upred-arestr-false [simp]: false ↦p a = false
by (pred-auto)

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

11.5 Alphabet Lens Laws

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

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

lemma in-var-prod-lens [alpha]:

63
wb-lens $Y \implies \text{in-var } x :_L (X \times_L Y) = \text{in-var } (x :_L X)$
by (simp add: in-var-def prod-as-plus lens-comp-assoc fst-lens-plus)

\textbf{Lemma} out-var-prod-lens [alpha]:
wb-lens $X \implies \text{out-var } x :_L (X \times_L Y) = \text{out-var } (x :_L Y)$
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

\textbf{11.6 Substitution Alphabet Extension}

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

definition subst-ext :: $\alpha$ subst $\Rightarrow$ $\beta$ subst (infix $\oplus$ 65) where
\[ \text{upred-defs}: \sigma \oplus_s x = (\lambda s. \text{put}_x s (\sigma (\text{get}_x s))) \]

\textbf{Lemma} id-subst-ext [usubst]:
mwb-lens $x \implies \text{id} \oplus_s x = \text{id}$
by pred-auto

\textbf{Lemma} upd-subst-ext [alpha]:
mwb-lens $x \implies (\sigma \uparrow c) \oplus_p x = (\sigma \oplus_s x) \uparrow (c \oplus_p x)$
by (pred-auto)

\textbf{Lemma} apply-subst-ext [alpha]:
mwb-lens $x \implies (\sigma \uparrow e) \oplus_p x = (\sigma \uparrow e) \oplus_p x$
by (pred-auto)

\textbf{Lemma} aext-upred-eq [alpha]:
\[ ((e =_u f) \oplus_p a) = ((e \oplus_p a) =_u (f \oplus_p a)) \]
by (pred-auto)

\textbf{Lemma} subst-aext-comp [usubst]:
mwb-lens $a \implies (\sigma \oplus a) \circ (\rho \oplus a) = (\sigma \circ \rho) \oplus a$
by pred-auto

\textbf{11.7 Substitution Alphabet Restriction}

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

definition subst-res :: $\alpha$ subst $\Rightarrow$ $\beta$ subst (infix $\smallfrown$ 65) where
\[ \text{upred-defs}: \sigma \smallfrown_s x = (\lambda s. \text{get}_x (\sigma (\text{create}_x s))) \]

\textbf{Lemma} id-subst-res [usubst]:
mwb-lens $x \implies \text{id} \smallfrown_s x = \text{id}$
by pred-auto

\textbf{Lemma} upd-subst-res [alpha]:
mwb-lens $x \implies \sigma(\&x:y \mapsto_s v) \smallfrown_s x = (\sigma \smallfrown_s x)(\&y :_s v \smallfrown c x)$
by (pred-auto)

\textbf{Lemma} subst-ext-res [usubst]:
mwb-lens $x \implies (\sigma \oplus_s x) \smallfrown_s x = \sigma$
by (pred-auto)

lemma unrest-subst-alpha-ext [unrest]:
\[ x \bowtie y \Rightarrow x \not\bowtie (P \oplus_s y) \]
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 \((\alpha, \alpha \times \beta)\) uexpr \((\llfloor\cdot\rrfloor\))
where \([P]\llfloor \equiv P \oplus_p \text{fst}_L\)

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

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 :: \((\alpha, \beta)\) uexpr \Rightarrow \((\alpha, \alpha \times \beta)\) uexpr \((\llfloor\cdot\rrfloor\))
where \([P]\llfloor \equiv P \oplus_p \text{snd}_L\)

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

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

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

**lemma** pre-var-subst [usubst]:
\[ \sigma(x \mapsto_v v) \uparrow [P]_\prec = \sigma \uparrow [P[v/v]/\&x]_\prec \]
by (pred-simp)

12.4 Unrestriction laws

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

**lemma** unrest-dash-var-pre [unrest]:

**fixes** x :: ('a \Rightarrow 'a)

**shows** $x \not\in [p]_\prec$
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 ≤ op <
  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 \land B' = ('A' \land 'B')
by (rel-auto)

**lemma** taut-conj-elim [elim!]:
\[ [ [A \land B]; [A'; 'B'] ] \Rightarrow P \] \Rightarrow P
by (rel-auto)

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

**lemma** taut-shEx-elim:
\[ [ \exists x \cdot P x]; \land x. \Sigma \not\in P x; \land x. 'P x' \Rightarrow Q ] \Rightarrow Q
by (rel-blast)

Linking refinement and HOL implication

**lemma** refine-prop-intro:
assumes $\Sigma \not\vdash P \, \Sigma \not\vdash Q \quad \Rightarrow \quad \neg P$
shows $P \subseteq Q$
using $\text{assms}$
by ($\text{pred-auto}$)

**Lemma taut-not:** $\Sigma \not\vdash P \quad \Rightarrow \quad \neg P$
by ($\text{rel-auto}$)

**Lemma taut-shAll-intro:**
$orall x. \neg P x \quad \Rightarrow \quad \forall x. P x$
by ($\text{rel-auto}$)

**Lemma taut-shAll-intro-2:**
$orall x y. \neg P x y \quad \Rightarrow \quad \forall (x, y). P x y$
by ($\text{rel-auto}$)

**Lemma taut-impl-intro:**
$[\Sigma \not\vdash P; \quad \neg Q \quad \Rightarrow \quad P]$ 
$\quad \Rightarrow \quad \neg P$
by ($\text{rel-auto}$)

**Lemma upred-eval-taut:**
$\left[ [ \left[ a \right] \downarrow \backslash b / \& v \right] \right] [ P ]_b$
by ($\text{pred-auto}$)

**Lemma refBy-order:** $P \subseteq Q \quad \Rightarrow \quad P$
by ($\text{pred-auto}$)

**Lemma conj-idem [simp]:** $((P::\alpha \text{ upred}) \land P) = P$
by ($\text{pred-auto}$)

**Lemma disj-idem [simp]:** $((P::\alpha \text{ upred}) \lor P) = P$
by ($\text{pred-auto}$)

**Lemma conj-comm:** $((P::\alpha \text{ upred}) \land Q) = (Q \land P)$
by ($\text{pred-auto}$)

**Lemma disj-comm:** $((P::\alpha \text{ upred}) \lor Q) = (Q \lor P)$
by ($\text{pred-auto}$)

**Lemma conj-subst:** $P = R \quad \Rightarrow \quad ((P::\alpha \text{ upred}) \land Q) = (R \land Q)$
by ($\text{pred-auto}$)

**Lemma disj-subst:** $P = R \quad \Rightarrow \quad ((P::\alpha \text{ upred}) \lor Q) = (R \lor Q)$
by ($\text{pred-auto}$)

**Lemma conj-assoc:** $((P::\alpha \text{ upred}) \land Q) \land S) = (P \land (Q \land S))$
by ($\text{pred-auto}$)

**Lemma disj-assoc:** $((P::\alpha \text{ upred}) \lor Q) \lor S) = (P \lor (Q \lor S))$
by ($\text{pred-auto}$)

**Lemma conj-disj-abs:** $((P::\alpha \text{ upred}) \land (P \lor Q)) = P$
by ($\text{pred-auto}$)

**Lemma disj-conj-abs:** $((P::\alpha \text{ upred}) \lor (P \land Q)) = P$
by (pred-auto)

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

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

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

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

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

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

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

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

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

**lemma** impl-refine-intro:
\[Q \subseteq (P \land R) \Rightarrow (P \Rightarrow Q) \subseteq R\]
by (pred-auto)

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

**lemma** impl-disjI:
\[\{P \Rightarrow R', \; Q \Rightarrow R'\} \Rightarrow \alpha (P \lor Q) \Rightarrow R'\]
by (rel-auto)

**lemma** conditional-iff:
\[(P \Rightarrow Q) = (P \Rightarrow R) \iff \alpha (Q \Leftrightarrow R)\]
by (pred-auto)

68
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]: \((\neg(\langle P::\alpha \upred \rangle \land Q)) = ((\neg P) \lor (\neg Q))\)
by (pred-auto)

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

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

lemma subsumption1:
\(\neg P \Rightarrow Q \Rightarrow Q = P\)
by (pred-auto)

lemma subsumption2:
\(P \Rightarrow \neg P \Rightarrow P\)
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: \((\langle P\rangle_u \land \langle Q\rangle_u) = \langle P \land Q\rangle_u\)
by (pred-auto)

lemma closure-imp-distr: \(\langle P \Rightarrow Q\rangle_u \Rightarrow \langle P\rangle_u \Rightarrow \langle Q\rangle_u\)
by (pred-auto)

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

lemma taut-iff-eq:
\(P \Leftrightarrow Q \Leftrightarrow (P = Q)\)

69
by (pred-auto)

lemma impl-alt-def: \((P \Rightarrow Q) = (\neg P \vee Q)\)
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]: \((\biguplus i \cdot F(P) \cdot true) = true\)
by (pred-auto)

lemma UINF-mem-UNIV [simp]: \((\bigsqcap x \in \text{UNIV} \cdot P(x)) = \bigsqcap x \cdot P(x)\)
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]: \(I \neq \{\} \implies (\bigsqcup i \in I \cdot false) = false\)
by (rel-simp)

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

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

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

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

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

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

lemma UINF-cong-eq:
\[ \land x. P_1(x) = P_2(x); \land x. 'P_1(x) \Rightarrow Q_1(x) =_u Q_2(x)' \] \[ \Rightarrow (\prod x \mid P_1(x) \cdot Q_1(x)) = (\prod x \mid P_2(x) \cdot Q_2(x)) \]

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

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

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

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

lemma UINF-as-Sup-image: \((\prod P \mid <P> \in_u <A> \cdot f(P)) = \prod (f' A)\)
apply (simp add: upred-defs bop rep-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 rep-eq lit rep-eq Inf uexpr-def)
apply (pred-simp)
apply (rule cong[of Inf])
apply (auto)
done

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

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

lemma USUP-as-Inf-image: \((\bigsqcup P \in P \cdot f(P)) = (\bigsqcup f : P)\)
apply (simp add: upred-defs bop rep-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 ∈ A → f i) = (⨆ i ∈ A · g(f(i)))
  by (pred-simp, rule-tac cong[of USUP Inf], auto)

lemma UINF-image-eq [simp]: UINF (λi. i ∈ A → f i) = (⨅ i ∈ A · g(f(i)))
  by (pred-simp, rule-tac cong[of USUP Inf], auto)

lemma subst-continuous [usubst]: σ † (⨅ i ∈ A · P(i)) = (⨅ i ∈ A → P(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 Inf-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 USUP Inf])
  apply (auto)
  done

lemma USUP-atLeast-first:
\((P(n) \land \bigcup i \in \{\text{Suc } n..\} \cdot P(i)) = (\bigcup i \in \{n..\} \cdot P(i))\)

**proof**

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

**qed**

**lemma** USUP-atLeast-Suc:
\((\bigcup i \in \{\text{Suc } m..\} \cdot P(i)) = (\bigcup i \in \{m..\} \cdot P(\text{Suc } i))\)
by (rel-simp, metis (full-types) Suc-le-D not-less-eq-eq)

**lemma** conj-UINF-dist:
\((P \land (\prod Q \in S \cdot F(Q))) = (\prod Q \in S \cdot P \land F(Q))\)
by (simp add: upred-defs bop.eq lit.eq, pred-auto)

**lemma** conj-UINF-ind-dist:
\((P \land (\prod Q \cdot F(Q))) = (\prod Q \cdot P \land F(Q))\)
by pred-auto

**lemma** disj-UINF-dist:
\(S \neq \{\} \implies (P \lor (\prod Q \in S \cdot F(Q))) = (\prod Q \in S \cdot P \lor F(Q))\)
by (simp add: upred-defs bop.eq lit.eq, pred-auto)

**lemma** UINF-conj-UINF [simp]:
\((\prod i \in I \cdot P(i)) \lor (\prod i \in I \cdot Q(i)) = (\prod i \in I \cdot P(i) \lor Q(i))\)
by (rel-auto)

**lemma** conj-USUP-dist:
\(S \neq \{\} \implies (P \land (\bigcup Q \in S \cdot F(Q))) = (\bigcup Q \in S \cdot P \land F(Q))\)

**lemma** USUP-conj-USUP [simp]:
\((\bigcup P \in A \cdot F(P)) \land (\bigcup P \in A \cdot G(P)) = (\bigcup P \in A \cdot F(P) \land G(P))\)
by (simp add: upred-defs bop.eq lit.eq, pred-auto)

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

**lemma** UINF-cong:
- assumes \( \land P. P \in A \implies F(P) = G(P)\)
- shows \( (\prod P \in A \cdot F(P)) = (\prod P \in A \cdot G(P))\)
by (simp add: UINF-as-Sup-collect assms)

**lemma** USUP-all-cong:
- assumes \( \land P. F(P) = G(P)\)
- shows \( (\bigcup P \cdot F(P)) = (\bigcup P \cdot G(P))\)
by (simp add: assms)

**lemma** USUP-cong:
- assumes \( \land P. P \in A \implies F(P) = G(P)\)
- shows \( (\bigcup P \in A \cdot F(P)) = (\bigcup P \in A \cdot G(P))\)
by (simp add: USUP-as-Inf-collect assms)
lemma \(\text{UINF-subset-mono}: A \subseteq B \implies (\bigsqcup P \in A \cdot F(P)) \subseteq (\bigsqcup P \in A \cdot F(P))\)
by (simp add: SUP-subset-mono UINF-as-Sup-collect)

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

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

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

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

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

lemma \(\text{UINF-all-nats \[simp\]}:
\begin{align*}
\text{fixes } P &:: \text{nat} \Rightarrow 'a \text{ upred} \\
\text{shows } (\bigsqcap n \cdot \bigsqcap i \in \{0..n\} \cdot P(i)) = (\bigsqcap n \cdot P(n))
\end{align*}
by (pred-auto)

lemma \(\text{USUP-all-nats \[simp\]}:
\begin{align*}
\text{fixes } P &:: \text{nat} \Rightarrow 'a \text{ upred} \\
\text{shows } (\bigsqcap n \cdot \bigsqcap i \in \{0..n\} \cdot P(i)) = (\bigsqcap n \cdot P(n))
\end{align*}
by (pred-auto)

lemma \(\text{UINF-upto-expand-first}:
\begin{align*}
(\bigsqcap i \in \{0..<\text{Suc}(n)\} \cdot P(i)) = (P(0) \lor (\bigsqcap i \in \{1..<\text{Suc}(n)\} \cdot P(i))
\end{align*}
apply (rel-auto)
using not-less by auto

lemma \(\text{UINF-upto-expand-last}:
\begin{align*}
(\bigsqcap i \in \{0..<\text{Suc}(n)\} \cdot P(i)) = ((\bigsqcap i \in \{0..<n\} \cdot P(i)) \lor P(n))
\end{align*}
apply (rel-auto)
using less-SucE by blast

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

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

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

lemma \(\text{UINF-list-conv}:
\begin{align*}
\end{align*}

\[(\prod i \in \{0..<\text{length}(xs)\} \cdot f (xs \cdot i)) = \text{foldr } op \lor (\text{map } f \hspace{1em} xs) \hspace{1em} \text{false}\]

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

lemma USUP-list-conv:
\[(\bigsqcap i \in \{0..<\text{length}(xs)\} \cdot f (xs \cdot i)) = \text{foldr } op \land (\text{map } f \hspace{1em} xs) \hspace{1em} \text{true}\]

apply (induct xs)
apply (rel-auto)
apply (simp add: USUP-upto-expand-first USUP-Suc-shift)
done

lemma UINF-refines':
assumes \( \bigwedge i. P \sqsubseteq Q(i) \)
shows \( P \sqsubseteq (\prod i \cdot Q(i)) \)
using asms
apply (rel-auto) using Sup-le-iff by fastforce

lemma UINF-pred-ueq [simp]:
\( (\bigsqcap x | \langle x \rangle =_u v \cdot P(x)) = (P x)[x\mapsto v] \)
by (pred-auto)

lemma UINF-pred-lit-eq [simp]:
\( (\bigsqcap x | \langle x \rangle = v \cdot P(x)) = (P v) \)
by (pred-auto)

13.3 Equality laws

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

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

lemma eq-cong-left:
assumes vwb-lens x $x \# Q$ $x \# R$ $x \# R$
shows \( (\langle x \rangle =_u x \amp Q) = (\langle x \rangle =_u x \amp R) \iff (Q = R) \)
using asms
by (pred-simp, (meson mwb-lens-def vwb-lens-mwb weak-lens-def)+)

lemma conj-eq-in-var-subst:
fixes x :: \( 'a \Rightarrow 'a \)
assumes vwb-lens x
shows \( P \land \langle x \rangle =_u v) = (P[x \mapsto v] \land \langle x \rangle =_u v) \)
using asms
by (pred-simp, (metis vwb-lens-wb wb-lens.get-put)+)

lemma conj-eq-out-var-subst:
fixes x :: \( 'a \Rightarrow 'a \)
assumes vwb-lens x
shows \( P \land \langle x \rangle =_u v) = (P[x \mapsto v] \land \langle x \rangle =_u v) \)
using asms
by (pred-simp, (metis vwb-lens-wb wb-lens.get-put)+)

lemma conj-pos-var-subst:

75
assumes \(\text{vwb-lens } x\)
shows \((\$x \land Q) = (\$x \land Q[\text{true}/\$x])\)
using \text{assms}
by \((\text{pred-auto, metis (full-types) vwb-lens-wb wb-lens.get-pat, metis (full-types) vwb-lens-wb wb-lens.get-pat})\)

lemma \texttt{conj-neg-var-subst}:
assumes \(\text{vwb-lens } x\)
shows \((\neg \$x \land Q) = (\neg \$x \land Q[\text{false}/\$x])\)
using \text{assms}
by \((\text{pred-auto, metis (full-types) vwb-lens-wb wb-lens.get-pat, metis (full-types) vwb-lens-wb wb-lens.get-pat})\)

lemma \texttt{upred-eq-true} [\texttt{simp}]: \((p =_{u} \text{true}) = p\)
by \((\text{pred-auto})\)

lemma \texttt{upred-eq-false} [\texttt{simp}]: \((p =_{u} \text{false}) = (\neg p)\)
by \((\text{pred-auto})\)

lemma \texttt{upred-true-eq} [\texttt{simp}]: \((\text{true} =_{u} p) = p\)
by \((\text{pred-auto})\)

lemma \texttt{upred-false-eq} [\texttt{simp}]: \((\text{false} =_{u} p) = (\neg p)\)
by \((\text{pred-auto})\)

lemma \texttt{conj-var-subst}:
assumes \(\text{vwb-lens } x\)
shows \((P \land \text{var } x =_{u} v) = (P[v/x] \land \text{var } x =_{u} v)\)
using \text{assms}
by \((\text{pred-simp, (metis (full-types) vwb-lens-def wb-lens.get-pat))})\)

13.4 HOL Variable Quantifiers

lemma \texttt{shEx-unbound} [\texttt{simp}]: \((\exists \ x \cdot P) = P\)
by \((\text{pred-auto})\)

lemma \texttt{shEx-bool} [\texttt{simp}]: \(\text{shEx } P = (P \text{ True } \lor P \text{ False})\)
by \((\text{pred-simp, metis (full-types)})\)

lemma \texttt{shEx-commute}: \((\exists \ x \cdot \exists \ y \cdot P \ x \ y) = (\exists \ y \cdot \exists \ x \cdot P \ x \ y)\)
by \((\text{pred-auto})\)

lemma \texttt{shEx-cong}: \([ \land \ x. \ P \ x = Q \ x ] \implies \text{shEx } P = \text{shEx } Q\)
by \((\text{pred-auto})\)

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

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

lemma \texttt{shAll-cong}: \([ \land \ x. \ P \ x = Q \ x ] \implies \text{shAll } P = \text{shAll } Q\)
by \((\text{pred-auto})\)

Quantifier lifting

named-theorems \texttt{uquant-lift}

lemma \texttt{shEx-lift-conj-1} [\texttt{uquant-lift}]:

76
(∃ 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 wvb-lens x
shows (P = Q) ≡ (∀ v. P[v/x] = Q[v/x])
using assms
by (pred-auto, metis wvb-lens-wb wb-lens, source-stability)

**lemma** eq-split-substI:
assumes wvb-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 wvb-lens x
shows \('P' ≡ (∀ v. 'P[v/x]' \)
using assms
by (pred-auto, metis wvb-lens-wb wb-lens, source-stability)

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

**lemma** bool-eq-splitI:
assumes wvb-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 wvb-lens x
shows \('P' ≡ ('P[false/x] ∧ P[true/x]') \)
proof -
from assms have \('P' = (∀ v. 'P[v/x]' \)
by (subt 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 x v
shows (\exists x \cdot P \wedge \text{var } x =_u v) = P[v/x]
using assms
by (pred-auto)

lemma exists-twice: mwb-lens x \Rightarrow (\exists x \cdot \exists x \cdot P) = (\exists x \cdot P)
by (pred-auto)

lemma all-twice: mwb-lens x \Rightarrow (\forall x \cdot \forall x \cdot P) = (\forall x \cdot P)
by (pred-auto)

lemma exists-sub: [ mwb-lens y; x \subseteq L y ] \Rightarrow (\exists x \cdot \exists y \cdot P) = (\exists y \cdot P)
by (pred-auto)

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

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

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

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

\textbf{lemma} \textit{unrest-as-exists}:
\[\text{vwb-lens } x \implies (x \not\in P) \iff (\exists \ x \cdot P) = P\]
\textit{by} (pred-simp, metis vwb-lens.put-eq)

\textbf{lemma} \textit{ex-mono}:
\[(P \subseteq Q) = \implies (\exists \ x \cdot P) \subseteq (\exists \ x \cdot Q)\]
\textit{by} (pred-auto)

\textbf{lemma} \textit{ex-weakens}:
\[(\exists \ x \cdot P) \subseteq P\]
\textit{by} (pred-simp, metis wb-lens.get-put)

\textbf{lemma} \textit{all-mono}:
\[(P \subseteq Q) = \implies (\forall \ x \cdot P) \subseteq (\forall \ x \cdot Q)\]
\textit{by} (pred-auto)

\textbf{lemma} \textit{all-strengthens}:
\[(\forall \ x \cdot P) \subseteq P\]
\textit{by} (pred-simp, metis wb-lens.get-put)

\textbf{lemma} \textit{ex-unrest}:
\[x \not\in P \implies (\exists \ x \cdot P) = P\]
\textit{by} (pred-auto)

\textbf{lemma} \textit{all-unrest}:
\[x \not\in P \implies (\forall \ x \cdot P) = P\]
\textit{by} (pred-auto)

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

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

\textbf{lemma} \textit{ex-conj-contr-left}:
\[x \not\in P \implies (\exists \ x \cdot P \land Q) = (P \land (\exists \ x \cdot Q))\]
\textit{by} (pred-auto)

\textbf{lemma} \textit{ex-conj-contr-right}:
\[x \not\in Q \implies (\exists \ x \cdot P \land Q) = ((\exists \ x \cdot P) \land Q)\]
\textit{by} (pred-auto)

13.7 Variable Restriction

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

\textbf{lemma} \textit{var-res-twice}:
\[\text{mwb-lens } x \implies P \mid_v \Sigma \mid_v x = P \mid_v x\]
\textit{by} (pred-auto)
13.8 Conditional laws

**lemma** cond-def:

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

by (pred-auto)

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

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

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

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

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

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

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

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

by (rel-auto)

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

\[ ((P \land b) \lor (Q \land b \Rightarrow R)) = ((P \lor Q) \land b \Rightarrow R) \]

by (pred-auto)

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

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

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

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

**lemma** cond-imp-distr:

\[ ((P \Rightarrow Q) \land b \Rightarrow (R \Rightarrow S)) = ((P \land b \Rightarrow R) \Rightarrow (Q \land b \Rightarrow S)) \] by (pred-auto)

**lemma** cond-eq-distr:

\[ ((P \equiv Q) \land b \Rightarrow (R \equiv S)) = ((P \land b \Rightarrow R) \equiv (Q \land b \Rightarrow S)) \] by (pred-auto)

**lemma** cond-cond-j conj-

\[ ((P \land (Q \land b \Rightarrow S)) = ((P \land Q) \land b \Rightarrow (P \land S)) \] by (pred-auto)

**lemma** cond-disj-distr: \( (P \lor (Q \land b \Rightarrow S)) = ((P \lor Q) \land b \Rightarrow (P \lor S)) \) by (pred-auto)

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

**lemma** cond-conj: \( P \land b \land c \Rightarrow Q = (P \land c \Rightarrow Q) \land b \Rightarrow Q \)

by (pred-auto)

**lemma** spec-cond-dist: \( (P \Rightarrow (Q \land b \Rightarrow R)) = ((P \Rightarrow Q) \land b \Rightarrow (P \Rightarrow R)) \)

by (pred-auto)

**lemma** cond-USUP dist: \( [\% P \in S \cdot F(P)] \land b \Rightarrow [\% P \in S \cdot G(P)] = [\% P \in S \cdot F(P) \land b \Rightarrow G(P)] \)

by (pred-auto)
13.9 Additional Expression Laws

**Lemma** \(\text{cond-UINF-dist}: (\prod P \in S \cdot F(P)) \triangleleft b \triangleright (\prod P \in S \cdot G(P)) = (\prod P \in S \cdot F(P) \triangleleft b \triangleright G(P))\)
by (pred-auto)

**Lemma** \(\text{cond-var-subst-left}:\)
- assumes \(\text{vwb-lens } x\)
- shows \((P[\text{true}/x] \triangleleft \text{var } x \triangleright Q) = (P \triangleleft \text{var } x \triangleright Q)\)
- using \(\text{assms by (pred-auto,metis (full-types) vwb-lens-wb wb-lens.get-put)}\)

**Lemma** \(\text{cond-var-subst-right}:\)
- assumes \(\text{vwb-lens } x\)
- shows \((P \triangleleft \text{var } x \triangleright Q[\text{false}/x]) = (P \triangleleft \text{var } x \triangleright Q)\)
- using \(\text{assms by (pred-auto,metis (full-types) vwb-lens.put-eq)}\)

**Lemma** \(\text{cond-split}:\)
- \(\text{vwb-lens } x \Longrightarrow (P[\text{true}/x] \triangleleft \text{var } x \triangleright P[\text{false}/x]) = P\)
by (rel-simp, (metis (full-types) vwb-lens.put-eq)+)

**Lemma** \(\text{cond-assign-subst}:\)
- \(\text{vwb-lens } x \Longrightarrow (P \triangleleft \text{utp-expr.var } x =_u v \triangleright Q) = (P[v/x] \triangleleft \text{utp-expr.var } x =_u v \triangleright Q)\)
apply (rel-simp) using \(\text{vwb-lens.put-eq by force}\)

**Lemma** \(\text{conj-conds}:\)
\((P_1 \triangleleft b \triangleright Q_1 \land P_2 \triangleleft b \triangleright Q_2) = (P_1 \land P_2) \triangleleft b \triangleright (Q_1 \land Q_2)\)
by pred-auto

**Lemma** \(\text{disj-conds}:\)
\((P_1 \triangleleft b \triangleright Q_1 \lor P_2 \triangleleft b \triangleright Q_2) = (P_1 \lor P_2) \triangleleft b \triangleright (Q_1 \lor Q_2)\)
by pred-auto

**Lemma** \(\text{cond-mono}:\)
\[(P_1 \subseteq P_2; Q_1 \subseteq Q_2) \Longrightarrow (P_1 \triangleleft b \triangleright Q_1) \subseteq (P_2 \triangleleft b \triangleright Q_2)\]
by (rel-auto)

**Lemma** \(\text{cond-mono-monic}:\)
\[(\text{mono } P; \text{mono } Q) \Longrightarrow \text{mono } (\lambda X. P X \triangleleft b \triangleright Q X)\]
by (simp add: mono-def, rel-blast)

Sometimes Isabelle desugars conditionals, and the following law undoes this

**Lemma** \(\text{resugar-cond}: \text{trop } (\lambda b P Q, (b \rightarrow P) \land (\neg b \rightarrow Q)) b P Q = \text{cond } P b Q\)
by (transfer, auto simp add: fun-eq-iff)

13.9 Additional Expression Laws
shows (numeral i :: ('a, 'a) uexpr) ≤_u numeral j = true
using assms by (pred-auto)

lemma unumeral-le-2 [simp]:
assumes (numeral i :: 'a::{numeral,linorder}) > numeral j
shows (numeral i :: ('a, 'a) uexpr) ≤_u numeral j = false
using assms by (pred-auto)

lemma uset-laws [simp]:
x ∈_u {}_u = false
x ∈_u {m..n}_u = (m ≤_u x ∧ x ≤_u n)
by (pred-auto)+

lemma pfun-entries-apply [simp]:
(entr_u(d,f) :: (('k, 'v) pfun, 'a) uexpr)(i)_a = ((f ›_a i) ⊲_a m(i)_a)
by (pred-auto)

lemma urit-eq [simp]:
x = y =⇒ (≪_a x ≫_a =_u ≫_a y) = true
by (rel-auto)

lemma urit-neq [simp]:
x ≠ y =⇒ (≪_a x ≫_a =_u ≫_a y) = false
by (rel-auto)

lemma uset-mems [simp]:
x ∈_u {y}_u = (x =_u y)
x ∈_u A ∪_u B = (x ∈_u A ∨ x ∈_u B)
x ∈_u A ∩_u B = (x ∈_u A ∧ x ∈_u B)
by (rel-auto)+

13.10 Refinement By Observation

Function to obtain the set of observations of a predicate

definition obs-apred :: 'a upred ⇒ 'a set ([·]_o)
where [upred-defs]: [P]_o = {b. [P]_o b}

lemma obs-apred-refine-iff:
P ⊆ Q =⇒ [Q]_o ⊆ [P]_o
by (pred-auto)

A refinement can be demonstrated by considering only the observations of the predicates which are relevant, i.e. not unrestricted, for them. In other words, if the alphabet can be split into two disjoint segments, x and y, and neither predicate refers to y then only x need be considered when checking for observations.

lemma refine-by-obs:
\begin{align*}
\text{assumes } & x \triangleright y \text{ bij-lens } (x + L \ y) y \# P y \# Q \{ v . \ P[<v>/x]\} \subseteq \{ v . \ Q[<v>/x]\} \\
\text{shows } & Q \subseteq P \\
\text{using } & \text{assms}(3-5) \\
\text{apply } & \text{(simp add: obs-upred-refine-iff subset-eq)} \\
\text{apply } & \text{(pred-simp)} \\
\text{apply } & \text{(rename-tac b)} \\
\text{apply } & \text{(drule-tac } x = \text{get } x \text{ in spec)} \\
\text{apply } & \text{(auto simp add: assms)} \\
\text{apply } & \text{(metis assms(1) assms(2) bij-lens.axioms(2) bij-lens-axioms-def lens-override-def lens-override-plus)} + \\
\text{done}
\end{align*}

13.11 Cylindric Algebra

\begin{align*}
\text{lemma } C1 & : (\exists \ x \cdot \text{false}) = \text{false} \\
& \text{by (pred-auto)} \\
\text{lemma } C2 & : wb-lens x \Rightarrow 'P \Rightarrow (\exists \ x \cdot P) \\
& \text{by (pred-simp, metis wb-lens.get-put)} \\
\text{lemma } C3 & : mwb-lens x \Rightarrow (\exists \ x \cdot (P \land (\exists \ y \cdot Q))) = ((\exists \ x \cdot P) \land (\exists \ x \cdot Q)) \\
& \text{by (pred-auto)} \\
\text{lemma } C4a & : x \approx_L y \Rightarrow ((\exists \ x \cdot (P \land (\exists \ y \cdot Q))) = ((\exists \ x \cdot P) \land (\exists \ x \cdot Q)) \\
& \text{by (pred-auto)} \\
\text{lemma } C4b & : x \triangleright \triangleright y \Rightarrow ((\exists \ x \cdot \exists \ y \cdot P) = (\exists \ y \cdot \exists \ x \cdot P) \\
& \text{using ex-commute by blast} \\
\text{lemma } C5 & : \text{fixes } x :: ('a \Rightarrow 'a) \\
& \text{shows } (\&x =_u \&x) = \text{true} \\
& \text{by (pred-auto)} \\
\text{lemma } C6 & : \\
& \text{assumes } wb-lens x x \triangleright z \\
& \text{shows } (\&y =_u \&z) = ((\exists \ x \cdot \&y =_u \&x \land \&x =_u \&z) \\
& \text{using assms} \\
& \text{by (pred-simp, (metis lens-indep-def)} + \\
\text{lemma } C7 & : \\
& \text{assumes weak-lens x x \triangleright y} \\
& \text{shows } ((\exists \ x \cdot \&x =_u \&y \land P) \land (\exists \ x \cdot \&x =_u \&y \land \neg P)) = \text{false} \\
& \text{using assms} \\
& \text{by (pred-simp, simp add: lens-indep-sym)} \\
\end{align*}

end

14 Healthiness Conditions

theory utp-healthy
imports utp-pred-laws
begin
14.1 Main Definitions

We collect closure laws for healthiness conditions in the following theorem attribute.

named-theorems closure

type-synonym 'α health = 'α upred ⇒ 'α upred

A predicate \( P \) is healthy, under healthiness function \( H \), if \( P \) is a fixed-point of \( H \).

definition Healthy :: 'α upred ⇒ 'α health ⇒ bool (infix is 30)
where \( P \) is \( H \equiv (H \ P = P) \)

lemma Healthy-def': \( P \) is \( H \) \( \iff \) \( (H \ P = P) \)
unfolding Healthy-def by auto

lemma Healthy-if: \( P \) is \( H \) \( \implies \) \( (H \ P = P) \)
unfolding Healthy-def by auto

lemma Healthy-intro: \( H(\ P = P \) \( \implies \) \( P \) is \( H \)
by (simp add: Healthy-def)

declare Healthy-def' [upred-defs]

abbreviation Healthy-carrier :: 'α health ⇒ 'α upred set ([H]_H)
where \( [H]_H \equiv \{P. \ P \in A\} \)

lemma Healthy-carrier-image:
\( A \subseteq [H]_H \implies \{H \mid P. \ P \in A\} \)
by (auto simp add: image-def, (metis Healthy-if mem-Collect-eq subsetCE)+)

lemma Healthy-carrier-Collect: \( A \subseteq [H]_H \implies \{H(\ P = P \mid P. \ P \in A\} \)
by (simp add: Healthy-carrier-image Setcompr-eq-image)

lemma Healthy-func:
\( [[F : [H_1]_H \to [H_2]_H; P is [H_1] \} \implies [H_2](F(\ P = P)) = F(\ P) \)
using Healthy-if by blast

lemma Healthy-comp:
\( [[P is [H_1]; P is [H_2] \} \implies [H_1 \circ [H_2] \)
by (simp add: Healthy-def)

lemma Healthy-apply-closed:
assumes \( F \in [H]_H \to [H]_H \ P is [H] \)
shows \( F(\ P = P \) \)
using assms(1) assms(2) by auto

lemma Healthy-set-image-member:
\( [[P \in F \ A; \ \bigwedge x. \ F x is [H] \} \implies [P is [H] \)
by blast

lemma Healthy-SUPREMUM:
\( A \subseteq [H]_H \implies SUPREMUM A H = \bigsqcup A \)
by (drule Healthy-carrier-image, presburger)

lemma Healthy-INFIMUM:
\( A \subseteq [H]_H \implies INFIMUM A H = \bigsqcap A \)
by (drule Healthy-carrier-image, presburger)

lemma Healthy-nu [closure]:
assumes mono F F ∈ [id]H → [H]H
shows ν F is H
by (metis (mono-tags) Healthy-def Healthy-func assms eq-id-iff lfp-unfold)

lemma Healthy-mu [closure]:
assumes mono F F ∈ [id]H → [H]H
shows µ F is H
by (metis (mono-tags) Healthy-def Healthy-func assms eq-id-iff gfp-unfold)

lemma Healthy-subset-member: [ A ⊆ [H]H; P ∈ A ] =⇒ H(P) = P
by (meson Ball-Collect Healthy-if)

lemma is-Healthy-subset-member: [ A ⊆ [H]H; P ∈ A ] =⇒ P is H
by blast

14.2 Properties of Healthiness Conditions

definition Idempotent :: 'a health ⇒ bool where
Idempotent(H) = (∀ P. H(H(P)) = H(P))

abbreviation Monotonic :: 'a health ⇒ bool where
Monotonic(H) ≡ mono H

definition IMH :: 'a health ⇒ bool where
IMH(H) = Idempotent(H) ∧ Monotonic(H)

definition Antitone :: 'a health ⇒ bool where
Antitone(H) = (∀ P Q. Q ⊑ P −→ (H(P) ⊑ H(Q)))

definition Conjunctive :: 'a health ⇒ bool where
Conjunctive(H) = (∃ Q. ∀ P. H(P) = (P ∧ Q))

definition FunctionalConjunctive :: 'a health ⇒ bool where
FunctionalConjunctive(H) = (∃ F. ∀ P. H(P) = (P ∧ F(P)) ∧ Monotonic(F))

definition WeakConjunctive :: 'a health ⇒ bool where
WeakConjunctive(H) = (∀ P. ∃ Q. H(P) = (P ∧ Q))

definition Disjunctuous :: 'a health ⇒ bool where
[upred-defs]: Disjunctuous H = (∀ P Q. H(P ∩ Q) = (H(P) ∩ H(Q)))

definition Continuous :: 'a health ⇒ bool where
[upred-defs]: Continuous H = (∀ A. A ≠ {} −→ H (∪ A) = ∪ (H ' A))

lemma Healthy-Idempotent [closure]:
Idempotent H =⇒ H(P) is H
by (simp add: Healthy-def Idempotent-def)

lemma Healthy-range: Idempotent H =⇒ range H = [H]H
by (auto simp add: image-def Healthy-if Healthy-Idempotent, metis Healthy-if)

lemma Idempotent-id [simp]: Idempotent id
by (simp add: Idempotent-def)
lemma Idempotent-comp [intro]:
    [[ Idempotent f; Idempotent g; \ f \circ g = g \circ f ]] \Rightarrow \ 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 (λ X. X)
by (simp add: monoI)

lemma Monotonic-const [closure]:
    Monotonic (λ x. c)
by (simp add: mono-def)

lemma Monotonic-comp [intro]:
    [[ Monotonic f; Monotonic g ]] \Rightarrow \ Monotonic \ (f \circ g)
by (simp add: mono-def)

lemma Monotonic-inf [closure]:
    assumes Monotonic P Monotonic Q
    shows Monotonic (λ 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 (λ 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)
    shows HC(P \land Q) = (HC(P) \land Q)
    using assms unfolding Conjunctive-def
    by (metis utp-pred-laws.inf.assoc utp-pred-laws.inf.commute)

lemma Conjunctive-distr-conj:
    assumes Conjunctive(HC)
    shows HC(P \land Q) = (HC(P) \land HC(Q))
    using assms unfolding Conjunctive-def
    by (metis Conjunctive-conj assms utp-pred-laws.inf.assoc utp-pred-laws.inf-right-idem)

lemma Conjunctive-distr-disj:
assumes $Conjunctive(HC)$
shows $HC(P \lor Q) = (HC(P) \lor HC(Q))$
using assms unfolding $Conjunctive$-def
using utp-pred-laws.inf-sup-distrib2 by fastforce

lemma $Conjunctive$-distr-cond:
assumes $Conjunctive(HC)$
shows $HC(P \triangleleft b \triangleright Q) = (HC(P) \triangleleft b \triangleright HC(Q))$
using assms unfolding $Conjunctive$-def
by (metis cond-conj-distr utp-pred-laws.inf-commute)

lemma $FunctionalConjunctive$-Monotonic:
$FunctionalConjunctive(H) \Longrightarrow Monotonic(H)$
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: $Disjunctuous\ H \Longrightarrow Monotonic\ H$
by (metis $Disjunctuous$-Monotonic mono-def semilattice-sup-class.le-iff-sup)

lemma $ContinuousD$ [dest]: $[\ Continuous\ H;\ A \neq \{\} \ ] \Longrightarrow H\ (\bigsqcap A) = (\bigsqcap P \in A.\ H(P))$
by (simp add: $Continuous$-def)

lemma $Continuous$-Disjunctuous: $Continuous\ H \Longrightarrow Disjunctuous\ H$
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]: $Continuous\ H \Longrightarrow Monotonic\ H$
by (simp add: $Continuous$-$Disjunctuous$ $Disjunctuous$-Monotonic)

lemma $Continuous$-comp [intro]:
$[\ Continuous\ f;\ Continuous\ g \ ] \Longrightarrow Continuous\ (f \circ g)$
by (simp add: $Continuous$-def)

lemma $Continuous$-const [closure]: $Continuous\ (\lambda\ X.\ P)$
by pred-auto

87
lemma Continuous-cond [closure]:
\[\text{assumes Continuous } F \text{ Continuous } G\]
\[\text{shows Continuous } (\lambda X. F(X) \triangleq b \triangleright G(X))\]
\[\text{using assms by (pred-auto)}\]

Closure laws derived from continuity

lemma Sup-Continuous-closed [closure]:
\[\text{Continuous } H; \bigwedge i, i \in A \implies P(i) = H; A \neq \{\} \implies (\bigcap i \in A. P(i)) = H\]
\[\text{by (drule ContinuousD[of } H P i A\text{]; simp add: UINF-mem-UNIV[THEN sym] UINF-as-Sup[THEN sym]}\]
\[\text{(metis (no-types, lifting) Healthy-def SUP cong image-image)}\]

lemma UINF-mem-Continuous-closed [closure]:
\[\text{Continuous } H; \bigwedge i, i \in A \implies P(i) = H; A \neq \{\} \implies (\bigcap i \in A \cdot P(i)) = H\]
\[\text{by (simp add: Sup-Continuous-closed UINF-as-Sup-collect)}\]

lemma UINF-mem-Continuous-closed-pair [closure]:
\[\text{assumes Continuous } H \bigwedge i, j. (i, j) \in A \implies P i j = H A \neq \{\}\]
\[\text{shows (\bigcap (i,j) \in A \cdot P i j) = H}\]
\[\text{proof –}\]
\[\text{have (\bigcap (i,j) \in A \cdot P i j) = (\bigcap x \in A \cdot P (fst x) \cdot (snd x))}\]
\[\text{by (rel-auto)}\]
\[\text{also have ... is } H\]
\[\text{by (metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse)}\]
\[\text{finally show ?thesis} .\]

qed

lemma UINF-mem-Continuous-closed-triple [closure]:
\[\text{assumes Continuous } H \bigwedge i, j, k. (i, j, k) \in A \implies P i j k = H A \neq \{\}\]
\[\text{shows (\bigcap (i,j,k) \in A \cdot P i j k) = H}\]
\[\text{proof –}\]
\[\text{have (\bigcap (i,j,k) \in A \cdot P i j k) = (\bigcap x \in A \cdot P (fst x) \cdot (fst (snd x)) \cdot (snd (snd x)))}\]
\[\text{by (rel-auto)}\]
\[\text{also have ... is } H\]
\[\text{by (metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse)}\]
\[\text{finally show ?thesis} .\]

qed

lemma UINF-Continuous-closed [closure]:
\[\text{Continuous } H; \bigwedge i. P(i) = H \implies (\bigcap i \cdot P(i)) = H\]
\[\text{using UINF-mem-Continuous-closed[of } H UNIV P\text{]}\]
\[\text{by (simp add: UINF-mem-UNIV)}\]

All continuous functions are also Scott-continuous

lemma sup-continuous-Continuous [closure]: Continuous \( F \) \implies sup-continuous \( F \)
\[\text{by (simp add: Continuous-def sup-continuous-def)}\]

lemma USUP-healthy: \( A \subseteq [H]_H \implies (\bigsqcup P \in A \cdot F(P)) = (\bigsqcup P \in A \cdot F(H(P)))\]
\[\text{by (rule USUP-cong, simp add: Healthy-subset-member)}\]

lemma UINF-healthy: \( A \subseteq [H]_H \implies (\bigsqcap P \in A \cdot F(P)) = (\bigsqcap P \in A \cdot F(H(P)))\]
\[\text{by (rule UINF-cong, simp add: Healthy-subset-member)}\]

end
15 Alphabetised Relations

theory utp-rel
imports
  utp-pred-laws
  utp-healthy
  utp-lift
  utp-tactics
begin

An alphabetised relation is simply a predicate whose state-space is a product type. In this theory we construct the core operators of the relational calculus, and prove a library of associated theorems, based on Chapters 2 and 5 of the UTP book [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 ina :: ('α \Rightarrow 'α × 'β) where
[lens-defs]: ina = \text{fst}_L

definition outa :: ('β \Rightarrow 'α × 'β) where
[lens-defs]: outa = \text{snd}_L

lemma ina-uvar [simp]: vwb-lens ina
by (unfold-locales, auto simp add: ina-def)

lemma outa-uvar [simp]: vwb-lens outa
by (unfold-locales, auto simp add: outa-def)

lemma var-in-alpha [simp]: \( x \, ;_L \, \text{ina} = \text{i-var} \, x \)
by (simp add: \text{fst-lens-def ina-def in-var-def})

lemma var-out-alpha [simp]: \( x \, ;_L \, \text{outa} = \text{o-var} \, x \)
by (simp add: outa-def out-var-def \text{snd-lens-def})

lemma drop-pre-inv [simp]: [ \text{outa} \; p ] = \ll p \rr = p
by (pred-simp)

lemma usubst-lookup-ivar-unrest [usubst]:
ina \; σ \Rightarrow \sigma_s (\text{i-var} \, x) = \$x
by (rel-simp, metis \text{fstI})

lemma usubst-lookup-ovar-unrest [usubst]:
outa \; σ \Rightarrow \sigma_s (\text{o-var} \, x) = \$x`
by (rel-simp, metis \text{sndI})

lemma out-alpha-in-indep [simp]:
outa \bowtie \text{in-var} \, x \text{ in-var} \, x \bowtie \text{outa}
by (simp-all add: \text{in-var-def outa-def lens-indep-def \text{fst-lens-def snd-lens-def lens-comp-def}})

lemma in-alpha-out-indep [simp]:
inα \bowtie \text{out-var} \, x \text{ out-var} \, x \bowtie \text{ina}
by (simp-all add: \text{in-var-def ina-def lens-indep-def \text{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
\[ [\sigma]_s \equiv \sigma \oplus_s \in\alpha \]

**abbreviation** usubst-rel-drop :: (′α × ′α) usubst ⇒ ′α usubst ([]-_s) where
\[ [\sigma]_s \equiv \sigma |_s \in\alpha \]

The alphabet of a relation then consists wholly of the input and output portions.

**lemma** alpha-in-out:
\[ \Sigma \approx \text{L in} + \text{L out} \alpha \]

by (simp add: fst-snd-id-lens in\ α -def lens-equiv-refl out\ α -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**

\[(\text{type}) (′α, ′β) \text{ urel} \leq (\text{type}) (′α \times ′β) \text{ upred}\]

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 ((-)_α)

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
\[ [\text{upred-defs}]: [b]_\rightarrow = [b]_< \]

**abbreviation** rcond :: (′α, ′β) urel ⇒ ′α cond ⇒ (′α, ′β) urel ⇒ (′α, ′β) urel
\[(\{\text{P} \rightarrow _\rightarrow \text{Q}\}, [52,0,53]) 52\]

**where** (P \text{ Q}) \equiv (P \triangleleft [b]_\rightarrow \triangleright \text{ 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.

**lift-definition** seqr::(′α, ′β) urel ⇒ (′β, ′γ) urel ⇒ (′α × ′γ) upred

is \(\lambda P Q. b \in \{(p. P \ p) \ O \ (q. Q \ q)\}\).

**adhoc-overloading**

useq seqr
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} :: \(\alpha \hrel \Rightarrow \alpha \hrel \Rightarrow \alpha \hrel\) (infixr \(h\)); \(\hrel\) where

\[\text{seqh } P \ Q \equiv (P \ ;; \ Q)\]

**abbreviation** \texttt{truer} :: \(\alpha \hrel\) where

\[\text{truer} \equiv \text{true}\]

**abbreviation** \texttt{falser} :: \(\alpha \hrel\) where

\[\text{falser} \equiv \text{false}\]

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} :: \((\alpha, \alpha \times \beta) \uexpr \Rightarrow (\alpha, \beta \times \alpha) \uexpr\) where \texttt{conv-r e} \(\equiv e \oplus p \text{swapL}\)

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 \(b\).

**definition** \texttt{assigns-r} :: \(\alpha \usubst \Rightarrow \alpha \hrel\)

\[\text{assigns-r } x \ v \equiv \begin{cases} \lambda \sigma \ (b, b') : b' = \sigma(b) \end{cases}\]

**adhoc-overloading**

\uassigns\texttt{assigns-r}

Relational identity, or skip, is then simply an assignment with the identity substitution: it simply identifies all variables.

**definition** \texttt{skip-r} :: \(\alpha \hrel\) where

\[\text{skip-r} = \text{assigns-r id}\]

**adhoc-overloading**

\uskip\texttt{skip-r}

We set up iterated sequential composition which iterates an indexed predicate over the elements of a list.

**definition** \texttt{seqr-iter} :: \(\alpha \ \text{list} \Rightarrow (\alpha \Rightarrow \beta \ hrel) \Rightarrow \beta \ hrel\) where

\[\text{seqr-iter } xs \ P \equiv \text{foldr} (\lambda \ i \ Q. \ P(i) ;; Q) \ xs \ II\]

A singleton assignment simply applies a singleton substitution function, and similarly for a double assignment.

**abbreviation** \texttt{assign-r} :: \((t \Rightarrow \alpha) \Rightarrow (t, \alpha) \uexpr \Rightarrow \alpha \hrel\)

\[\text{assign-r } x \ v \equiv \langle [x \mapsto s \ v] \rangle_a\]

**abbreviation** \texttt{assign-2-r} :: \((t \Rightarrow \alpha) \Rightarrow (t'2 \Rightarrow \alpha) \Rightarrow (t'2, \alpha) \uexpr \Rightarrow \alpha \hrel\)

\[\text{assign-2-r } x \ y \ v \ v \equiv \text{assigns-r } [x \mapsto a, y \mapsto s \ v]\]

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} :: \((\beta, \alpha) \ \text{lens} \Rightarrow \alpha \hrel\) where

\[\text{skip-ra } v = (\$v' =_a \$v)\]

Similarly, we define the alphabetised assignment operator.
definition assigns-ra :: 'a usubst ⇒ ('β, 'a) lens ⇒ 'a hrel (\rightarrow_) where  
\langle σ \rangle a = ((σ)_a \uparrow \text{skip-ra } a)

Assumptions (c^T) and assertions (c_\bot) 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 :: 'a upred ⇒ 'a hrel (\{\}^T) where  
\{urel-defs\}: rassume c = II \triangleright c \triangleright_r false

definition rassert :: 'a upred ⇒ 'a hrel (\{\}^\bot) where  
\{urel-defs\}: rassert c = II \triangleleft c \triangleright_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 :: 'a cond ⇒ 'a hrel (\{\}^T)  
where \{urel-defs\}: \langle b \rangle_T = ((\{\}^\bot)_T \triangleleft (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 :: 'a cond ⇒ 'a hrel ⇒ 'a hrel (\{\}^\bot - do - od) where  
\{urel-defs\}: while^\bot b do P od = (\nu X \cdot (P ;; X) \triangleleft b \triangleright_r II)

While loops with invariant decoration (cf. [1]) – partial correctness.

definition while-inv :: 'a cond ⇒ 'a cond ⇒ 'a hrel ⇒ 'a hrel (while - invr - do - od) where  
\{urel-defs\}: while invr p do S od = while b do S od

While loops with invariant decoration – total correctness.


definition while-inv-bot :: 'a cond ⇒ 'a cond ⇒ 'a hrel ⇒ 'a hrel (while_\bot - invr - do - od 71) where  
\{urel-defs\}: while_\bot b invr p do S od = while_\bot b do S od

While loops with invariant and variant decorations – total correctness.

definition while-vrt ::  
'b cond ⇒ 'a hrel ⇒ (nat, 'a) uexpr ⇒ 'a hrel ⇒ 'a hrel (while - invr - vrt - do - od) where  
\{urel-defs\}: while b invr p vrt v do S od = while_\bot 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 :: 'a hrel ⇒ ('a ⇒ 'a) ⇒ 'a hrel (infix \rightarrow_\alpha 80) where  
\{urel-defs\}: P \rightarrow_\alpha x = (\exists \$x \cdot \exists \$x' \cdot P)

Alphabet extension and restriction add additional variables by the given lens in both their primed and unprimed versions.


definition rel-aext :: 'b hrel ⇒ ('b ⇒ 'a) ⇒ 'a hrel  
where \{upred-defs\}: rel-aext P a = P oplus (a \times_L a)
definition rel-ares :: 'a hrel ⇒ ('β ⇒ 'α) ⇒ 'β hrel
where [upred-defs]: rel-ares P a = (P ↾_p (a × a))

We next describe frames and antiframes with the help of lenses. A frame states that P defines how variables in a changed, and all those outside of a remain the same. An antiframe describes the converse: all variables outside a are specified by P, and all those in remain the same. For more information please see [17].

definition frame :: ('a ⇒ 'α) ⇒ 'α hrel ⇒ 'α hrel
where [urel-defs]: frame a P = (P ∧ $v' =_a $v ⊕ $v' on &a)

definition antiframe :: ('a ⇒ 'α) ⇒ 'α hrel ⇒ 'α hrel
where [urel-defs]: antiframe a P = (P ∧ $v' =_a $v' ⊕ $v on &a)

Frame extension combines alphabet extension with the frame operator to both add additional variables and then frame those.

definition rel-frext :: ('β ⇒ 'α) ⇒ 'β hrel ⇒ 'α hrel
where [upred-defs]: rel-frext a P = frame a (rel-aext P a)

The nameset operator can be used to hide a portion of the after-state that lies outside the lens a. It can be useful to partition a relation’s variables in order to conjoin it with another relation.

definition nameset :: ('a ⇒ 'α) ⇒ 'α hrel ⇒ 'α hrel
where [urel-defs]: nameset a P = (P ↾_v {$v$ =_a$'v$})

15.3 Syntax Translations

syntax
— Alternative traditional conditional syntax
-upt-if :: logic ⇒ logic ⇒ logic ⇒ logic ((ifw (-)/ then (-)/ else (-)) [0, 0, 71] 71)
— Iterated sequential composition
-seqr-iter :: pttrn ⇒ 'a list ⇒ 'σ hrel ⇒ 'σ hrel ((3;; - · - / -) [0, 0, 10] 10)
— Single and multiple assignment
-assignment :: svids ⇒ uexprs ⇒ 'α hrel (‘(-) := ‘(-))
-assignment :: svids ⇒ uexprs ⇒ 'α hrel (infixr := 72)
— Indexed assignment
-assignment-upd :: 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 (infix1 ⊕_r 90)
— Relational Alphabet Restriction
-rel-ares :: logic ⇒ salpha ⇒ logic (infix1 |_r 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
The following code sets up pretty-printing for homogeneous relation algebra expressions. We cannot do this via the “translations” command as we only want the rule to apply when the input and output alphabet types are the same. The code has to deconstruct a \((a', a)\) `uexpr` type, determine that it is relational (product alphabet), and then checks if the types `alpha` and `beta` are the same. If they are, the type is printed as a `hexpr`. Otherwise, we have no match. We then set up a regular translation for the `hrel` type that uses this.

```plaintext
print-translation ⇨
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;
  end
in [[@{type-syntax uexpr},tr']]]
end
```

**Translations**

\[(type) 'a hrel <= (type) (bool, 'a) hexpr\]

### 15.4 Relation Properties

We describe some properties of relations, including functional and injective relations. We also provide operators for extracting the domain and range of a UTP relation.

**Definition** `ufunctional` :: \((a', b) urel \to bool\)
where [urel-defs]:
\[\text{ufunctional } R \iff II \subseteq R^{-}; R \]

**Definition** `uinj` :: \((a', b) urel \to bool\)
where [urel-defs]:
\[\text{uinj } R \iff II \subseteq R; R \]

**Definition** `Dom` :: \(\alpha hrel \to (\alpha upred\)
where [upred-defs]:
\[\text{Dom } P = [\exists v \cdot P] \]

**Definition** `Ran` :: \(\alpha hrel \to (\alpha upred\]

94
where \([\text{upred-defs}]: \text{Ran } P = [\exists v \cdot P]_\succ\)

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

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

### 15.5 Introduction laws

**lemma** urel-refine-ext:

\[
\lceil \land s \cdot s', P[\llangle s \rrangle,\llangle s' \rrangle/$v,v'] \subseteq Q[\llangle s \rrangle,\llangle s' \rrangle/$v,v'] \rceil \implies P \subseteq Q
\]

by (rel-auto)

**lemma** urel-eq-ext:

\[
\lceil \land s \cdot s', P[\llangle s \rrangle,\llangle s' \rrangle/$v,v'] = Q[\llangle s \rrangle,\llangle s' \rrangle/$v,v'] \rceil \implies P = Q
\]

by (rel-auto)

### 15.6 Unrestriction Laws

**lemma** unrest-iuvar [unrest]: out \(\alpha \sharp x\)

by (metis fst-snd-lens-indep lift-pre-var out \(\alpha \)-def unrest-aext-indep)

**lemma** unrest-ouvar [unrest]: in \(\alpha \sharp x'\)

by (metis in \(\alpha \)-def lift-post-var snd-fst-lens-indep unrest-aext-indep)

**lemma** unrest-semir-undash [unrest]:

fixes \(x::(\alpha \rightarrow (\alpha \times (\beta \times \gamma)))\)

assumes \(\sharp x \not\in P\)

shows \(\sharp x \not\in P \vee Q\)

using \(\text{assms}\) by (rel-auto)

**lemma** unrest-semir-dash [unrest]:

fixes \(x::(\alpha \rightarrow (\alpha \times (\beta \times \gamma)))\)

assumes \(\sharp x' \not\in Q\)

shows \(\sharp x' \not\in P \vee Q\)

using \(\text{assms}\) by (rel-auto)

**lemma** unrest-cond [unrest]:

\(\llceil x \not\in P; x \not\in b; x \not\in Q \rceil \implies x \not\in P \bowtie b \bowtie Q\)

by (rel-auto)

**lemma** unrest-lift-rcond [unrest]:

\(x \not\in (\llceil b \rceil)_< \implies x \not\in (\llceil b \rceil)_<\)

by (simp add: lift-rcond-def)

**lemma** unrest-ina-var [unrest]:

\(\llceil \text{mwb-lens } x; \text{ina } \sharp (P :: ('a, ('a \times 'b)) \text{ uexpr}) \rceil \implies \sharp x \not\in P\)

by (rel-auto)

**lemma** unrest-outa-var [unrest]:

\(\llceil \text{mwb-lens } x; \text{outa } \sharp (P :: ('a, ('a \times 'b)) \text{ uexpr}) \rceil \implies \sharp x' \not\in P\)

by (rel-auto)

**lemma** unrest-pre-outa [unrest]: outa \(\sharp (\llceil b \rceil)_<\)

by (transfer, auto simp add: outa-def)

**lemma** unrest-post-ina [unrest]: ina \(\sharp (\llceil b \rceil)_>\)
by (transfer, auto simp add: ina-def)

lemma unrest-pre-in-var [unrest]:
\[ x \sharp p \text{I} \Rightarrow \sigma x \sharp \lfloor p \text{I} \rfloor < \]
by (transfer, simp)

lemma unrest-post-out-var [unrest]:
\[ x \sharp p \text{I} \Rightarrow \sigma x' \sharp \lceil p \text{I} \rceil > \]
by (transfer, simp)

lemma unrest-conv-aoutα [unrest]:
\[ ina \sharp p \Rightarrow outα \sharp p^- \]
by (transfer, auto simp add: lens-defs)

lemma unrest-conv-ino [unrest]:
\[ outα \sharp p \Rightarrow inα \sharp p^- \]
by (transfer, auto simp add: lens-defs)

lemma unrest-in-rel-var-res [unrest]:
\[ vw-lens x \Rightarrow \sigma x \sharp (P \mid α x) \]
by (simp add: rel-var-res-def unrest)

lemma unrest-out-rel-var-res [unrest]:
\[ vw-lens x \Rightarrow \sigma x' \sharp (P \mid α x) \]
by (simp add: rel-var-res-def unrest)

lemma unrest-out-alpha-usubst-rel-lift [unrest]:
\[ outα \sharp [σ]_s \]
by (rel-auto)

lemma unrest-in-rel-aext [unrest]:
\[ x ≧ y \Rightarrow \sigma y \sharp P \oplus r x \]
by (simp add: rel-aext-def unrest-aext-indep)

lemma unrest-out-rel-aext [unrest]:
\[ x ≧ y \Rightarrow \sigma y' \sharp P \oplus r x \]
by (simp add: rel-aext-def unrest-aext-indep)

lemma rel-aext-seq [α]:
\[ weak-lens a \Rightarrow (P \; ; ; Q) \oplus r a = (P \oplus_r a \; ; ; Q \oplus_r a) \]
apply (rel-auto)
apply (rename-tac aa b y)
apply (rule-tac x=\alpha y in exI)
apply (simp)
done

lemma rel-aext-cond [α]:
\[ (P \triangleleft b \triangleright_r Q) \oplus_r a = (P \oplus_r a \triangleleft b \oplus_p a \triangleright_r Q \oplus_r a) \]
by (rel-auto)

15.7 Substitution laws

lemma subst-seq-left [usubst]:
\[ outα \sharp σ \Rightarrow σ \uparrow (P \; ; ; Q) = (σ \uparrow P) \; ; ; Q \]
by (rel-simp, (metis (no-types, lifting) Pair-inject surjective-pairing)+)

lemma subst-seq-right [usubst]:
\[ inα \sharp σ \Rightarrow σ \uparrow (P \; ; ; Q) = P \; ; ; (σ \uparrow Q) \]
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]`:

```plaintext
fixes x :: (bool ⇒ 'α)
shows
\(\land P Q \cdot \sigma (\$x \mapsto \_ \, true) \triangleright (P \triangleleft Q) = \sigma \triangleright (P[true/$x\!] \triangleleft Q)\)
\(\land P Q \cdot \sigma (\$x \mapsto \_ \, false) \triangleright (P \triangleleft Q) = \sigma \triangleright (P[false/$x\!] \triangleleft Q)\)
\(\land P Q \cdot \sigma (\$x' \mapsto \_ \, true) \triangleright (P \triangleleft Q) = \sigma \triangleright (P \triangleleft Q[true/$x'\!]\)
\(\land P Q \cdot \sigma (\$x' \mapsto \_ \, false) \triangleright (P \triangleleft Q) = \sigma \triangleright (P \triangleleft Q[false/$x'\!]\)
by (rel-auto)+
```

**lemma** `zero-one-seqr-laws [usubst]`:

```plaintext
fixes x :: (′α ⇒ ′α)
shows
\(\land P Q \cdot \sigma (\$x \mapsto _0) \triangleright (P \triangleleft Q) = \sigma \triangleright (P[0/$x\!] \triangleleft Q)\)
\(\land P Q \cdot \sigma (\$x \mapsto _1) \triangleright (P \triangleleft Q) = \sigma \triangleright (P[1/$x\!] \triangleleft Q)\)
\(\land P Q \cdot \sigma (\$x' \mapsto _0) \triangleright (P \triangleleft Q) = \sigma \triangleright (P \triangleleft Q[0/$x'\!]\)
\(\land P Q \cdot \sigma (\$x' \mapsto _1) \triangleright (P \triangleleft Q) = \sigma \triangleright (P \triangleleft Q[1/$x'\!]\)
by (rel-auto)+
```

**lemma** `numeral-seqr-laws [usubst]`:

```plaintext
fixes x :: (′α ⇒ ′α)
shows
\(\land P Q \cdot \sigma (\$x \mapsto \_ \, \text{numeral} n) \triangleright (P \triangleleft Q) = \sigma \triangleright (P[\text{numeral} n/$x\!] \triangleleft Q)\)
\(\land P Q \cdot \sigma (\$x' \mapsto \_ \, \text{numeral} n) \triangleright (P \triangleleft Q) = \sigma \triangleright (P \triangleleft Q[\text{numeral} n/$x'\!]\)
by (rel-auto)+
```

**lemma** `usubst-condr [usubst]`:

```plaintext
\(\sigma \triangleright (P \triangleleft b \triangleright Q) = (\sigma \triangleright P \triangleleft b \triangleright \sigma \triangleright Q)\)
by (rel-auto)
```

**lemma** `subst-skip-r [usubst]`:

```plaintext
\(out\alpha \not\in \sigma \Longrightarrow \sigma \triangleright H = (\sigma\rangle\rangle_{\alpha})\)
by (rel-simp, (metis (mono-tags, lifting) prod.sel(1) sndI surjective-pairing)+)
```

**lemma** `subpre-skip [usubst]`:

```plaintext
\([\sigma]\rangle\rangle H = (\sigma\rangle\rangle)\)
by (rel-auto)
```

**lemma** `subst-rel-lift-seq [usubst]`:

```plaintext
\([\sigma]\rangle\rangle (P \triangleleft Q) = ([\sigma]\rangle\rangle P \triangleleft Q)\)
by (rel-auto)
```

**lemma** `subst-rel-lift-comp [usubst]`:

```plaintext
\([\sigma]\rangle\rangle \circ [\vartheta] = [\sigma \circ \vartheta]\rangle\rangle\)
by (rel-auto)
```

**lemma** `usubst-upd-in-comp [usubst]`:

```plaintext
\(\sigma(kin\alpha:x \mapsto _v) = \sigma(\$x \mapsto _v)\)
by (simp add: pr-var-def fst-lens-def in\alpha-def in-var-def)
```

**lemma** `usubst-upd-out-comp [usubst]`:

```plaintext
\(\sigma(kout\alpha:x \mapsto _v) = \sigma(\$x' \mapsto _v)\)
```
by (simp add: pr-var-def outa-def out-var-def snd-lens-def)

lemma subst-lift-upd [alpha]:
fixes $x :: (\alpha \Rightarrow \alpha)$
shows $[\nu(v \mapsto \sigma(x \mapsto s))_s = [\sigma(u \mapsto \nu)_s]
by (simp add: alpha usubst, simp add: pr-var-def fst-lens-def in
alpha -def in-var-def)

lemma subst-drop-upd [alpha]:
fixes $x :: (\alpha \Rightarrow \alpha)$
shows $\lfloor \nu(v \mapsto \sigma(u \mapsto s))_s = \lfloor \nu(v)_s <
by (simp add: pre-simp)

lemma subst-lift-pre [usubst]: $[\nu(v \mapsto \sigma)_s < = [\nu(v)_s <
by (metis apply-subst-ext fst-vwb-lens in alpha -def)

lemma unrest-usubst-lift-out [unrest]:
fixes $x :: (\alpha \Rightarrow \alpha)$
shows $\nu(x \mapsto \sigma)_s < = \nu(x \mapsto \sigma)_s <
by (rel-auto)

lemma subst-lift-cond [usubst]: $[\nu(v \mapsto \sigma)_s < = [\nu(v)_s <
by (rel-auto)

15.8 Alphabet laws

lemma aext-cond [alpha]:
$(P \triangleleft_a Q) \oplus_a a = ((P \oplus_a a) \triangleleft_b (b \oplus_a a) \triangleright (Q \oplus_a a))
by (rel-auto)

lemma aext-seq [alpha]:
$((P \oplus_a a) \oplus_a a) \triangleleft_b (b \oplus_a a) \triangleright (P \oplus_a a)$
by (rel-simp, metis wb-lens-weak weak-lens,put-get)

lemma rel-ares-aext [alpha]:
$[true]_< = true$
by (rel-auto)

lemma rcond-lift-true [simp]:
$[false]_< = false$
by (rel-auto)

lemma rel-aext-ares [alpha]:
$([a, a'] \oplus_a a) \oplus_a a = P$
by (rel-auto)

lemma rel-aext-ares [alpha]:
$([a, a'] \oplus_a a) \oplus_a a = P$
by (rel-auto)
Lemma `rel-aext-uses [unrest]`:

\[
\text{vwb-lens } a \implies (\$a, \$a') \vdash (P \oplus_r a)
\]

by (rel-auto)

15.9 Relational unrestriction

Relational unrestriction states that a variable is both unchanged by a relation, and is not "read" by the relation.

Definition `RID :: (\'a \implies \'a) hrel \implies \'a 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 := \langle v \rangle ;; P = P ;; x := \langle v \rangle) \implies RID(x)(P) = P`

apply (rel-auto)
  apply (metis vwb-lens.put-eq)
  apply (metis vwb-lens-wb wb-lens.get-put wb-lens-weak weak-lens.put-get)
done

Lemma `RID2`: `vwb-lens x \implies x := \langle v \rangle ;; RID(x)(P) = RID(x)(P) ;; x := \langle v \rangle`

apply (rel-auto)
  apply blast
done

Lemma `RID-assgn-commute`:

\[
vwb-lens x \implies P = RID(x)(P) \iff (\forall v. x := \langle v \rangle ;; P = P ;; x := \langle v \rangle)
\]

by (metis RID1 RID2)

Lemma `RID-idem`:

\[
mwb-lens x \implies RID(x)(RID(x)(P)) = RID(x)(P)
\]

by (rel-auto)

Lemma `RID-mono`:

\[
P \sqsubseteq Q \implies RID(x)(P) \sqsubseteq RID(x)(Q)
\]

by (rel-auto)

Lemma `RID-pr-var [simp]`:

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

by (simp add: pr-var-def)

Lemma `RID-skip-r`:

\[
vwb-lens x \implies RID(x)(\text{II }) = \text{II}
\]

apply (rel-auto) using vwb-lens.put-eq by fastforce

Lemma `skip-r-RID [closure]`: `vwb-lens x \implies \text{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]`: \[ P \text{ is RID(x); Q is RID(x) } \implies (P \lor Q) \text{ is RID(x) } \]

by (simp add: Healthy-def RID-disj)
lemma RID-conj:
\[ \text{vwb-lens } x \implies \text{RID}(x)(\text{RID}(x)(P) \land \text{RID}(x)(Q)) = (\text{RID}(x)(P) \land \text{RID}(x)(Q)) \]
by (rel-auto)

lemma conj-RID [closure]: [ \text{vwb-lens } x; P \text{ is } \text{RID}(x); Q \text{ is } \text{RID}(x) ] \implies (P \land Q) \text{ is } \text{RID}(x)
by (metis \text{Healthy-if} \text{ Healthy-intro} \text{ RID-conj})

lemma RID-assigns-r-diff:
\[ \left[ \text{vwb-lens } x; x \notin \sigma \right] \implies \text{RID}(x)(\langle \sigma \rangle_a) = \langle \sigma \rangle_a \]
apply (rel-auto)
  apply (metis \text{vwb-lens}.put-eq)
  apply (metis \text{vwb-lens-wb} \text{ wb-lens.get-put} \text{ wb-lens-weak} \text{ weak-lens.put-get})
done

lemma assigns-r-RID [closure]: [ \text{vwb-lens } x; x \notin \sigma ] \implies \langle \sigma \rangle_a \text{ is } \text{RID}(x)
by (simp add: \text{Healthy-def RID-assigns-r-diff})

lemma RID-assign-r-same:
\[ \text{vwb-lens } x \implies \text{RID}(x)(x := v) = II \]
apply (rel-auto)
using \text{vwb-lens}.put-eq apply fastforce
done

lemma RID-seq-left:
assumes \text{vwb-lens } x
shows \text{RID}(x)(\text{RID}(x)(P) ;; Q) = (\text{RID}(x)(P) ;; \text{RID}(x)(Q))
proof –
  have \text{RID}(x)(\text{RID}(x)(P) ;; Q) = ((\exists \ x \cdot \exists x' \cdot (\exists x \cdot \exists x' :: P) \land x' =_u x) ;; Q) \land x' =_u x
  by (simp add: \text{RID-def usubst})
  also from \text{assms} have \ldots = (((\exists \ x \cdot \exists x' :: P) \land (\exists x \cdot \exists x' =_u x)) ;; (\exists x' :: Q)) \land x' =_u x
  by (rel-auto)
  also from \text{assms} have \ldots = (((\exists \ x :: (\exists x' :: P) \land x' =_u x) ;; (\exists x :: (\exists x' :: Q)) \land x' =_u x) \land x' =_u x
  by (rel-auto)
  also from \text{assms} have \ldots = (((\exists \ x :: (\exists x' :: P) \land x' =_u x) ;; (\exists x :: (\exists x' :: Q)) \land x' =_u x) \land x' =_u x
  by (rel-auto)
  also have \ldots = (((\exists \ x :: (\exists x' :: P) \land x' =_u x) ;; (\exists x :: (\exists x' :: Q)) \land x' =_u x))
  by (rel-auto)
  also have \ldots = (\text{RID}(x)(P) ;; \text{RID}(x)(Q))
  by (rel-auto)
finally show \ ?thesis .
qed

lemma RID-seq-right:
assumes \text{vwb-lens } x
shows \text{RID}(x)(P ;; \text{RID}(x)(Q)) = (\text{RID}(x)(P) ;; \text{RID}(x)(Q))
proof –
  have \text{RID}(x)(P ;; \text{RID}(x)(Q)) = ((\exists \ x :: \exists x' :: P ;; ((\exists x :: \exists x' :: Q) \land x' =_u x)) \land x' =_u x}
\(=_{u} \varsigma x\)

by (simp add: RID-def asubst)

also from assms have \(\ldots = (((\exists x \cdot P) ;; (\exists x \cdot \exists x' \cdot Q) \land (\exists x' \cdot \varsigma x' =_{u} x)) \land \varsigma x' =_{u} \varsigma x\)

by (rel-auto)

also from assms have \(\ldots = (((\exists x \cdot \exists x' \cdot P) ;; (\exists x \cdot \exists x' \cdot Q)) \land \varsigma x' =_{u} \varsigma x)\)

apply (rel-auto)

apply (metis vwb-lens.put-get)

apply (metis mwb-lens.put-put vwb-lens-mwb)

done

also from assms have \(\ldots = (((\exists x \cdot \exists x' \cdot P) \land \varsigma x' =_{u} \varsigma x) ;; (\exists x \cdot \exists x' \cdot Q) \land \varsigma x' =_{u} \varsigma x)) \land \varsigma x' =_{u} \varsigma x\)

by (rel-simp robust, metis (full-types) mwb-lens.put-put vwb-lens-def wb-lens-weak weak-lens.put-get)

also have \(\ldots = (RID(x)(P) ;; RID(x)(Q))\)

by (rel-auto)

finally show \(?thesis\).

qed

lemma seqr-RID-closed [closure]: \([ vwb-lens x; P \text{ is } RID(x); Q \text{ is } RID(x) ] \implies P ;; Q \text{ is } RID(x)\)

by (metis Healthy-def RID-seq-right)

definition unrest-relation :: ('a \(\implies\) 'a) \(\implies\) 'a \textit{ hrel} \(\implies\) bool (infix \(\varpi\) 20)

where \((x \varpi P) \leftrightarrow (P \text{ is } RID(x))\)

declare unrest-relation-def [urel-defs]

lemma runrest-assign-commute:

\([ vwb-lens x; x \varpi P ] \implies x := \langle v \rangle ;; P = P ;; x := \langle v \rangle\)

by (metis RID2 Healthy-def unrest-relation-def)

lemma runrest-ident-var:

assumes \(x \varpi P\)

shows \((\varsigma x \land P) = (P \land \varsigma x)\)

proof -

have \(P = (\varsigma x' =_{u} x \land P)\)

by (metis RID-def assms Healthy-def unrest-relation-def utp-pred-laws.inf.cobounded2 utp-pred-laws.inf-absorb2)

moreover have \((\varsigma x' =_{u} x \land (\varsigma x \land P)) = (\varsigma x' =_{u} x \land (P \land \varsigma x'))\)

by (rel-auto)

ultimately show \(?thesis\)

by (metis utp-pred-laws.inf.assoc utp-pred-laws.inf-left-commute)

qed

lemma skip-r-runrest [unrest]:

\(vwb-lens x \Rightarrow x \varpi I\)

by (simp add: unrest-relation-def closure)

lemma assigns-r-runrest:

\([ vwb-lens x; x \varpi \alpha ] \Rightarrow x \varpi (\alpha)_{u}\)

by (simp add: unrest-relation-def closure)

lemma seqr-r-runrest [unrest]:

101
assumes \( \text{vwb-lens } x \ x \ x \ x \ x \ x \ P \ x \ x \ x \ x \ x \ x \ x \ Q \)
shows \( x \ x \ x (P \ x \ x \ x \ x \ x \ x \ x \ Q) \)
using assms by (simp add: unrest-relation-def closure)

lemma false-runrest [unrest]: \( x \ x \ x \ x \ x \ x \ x \ x \ false \)
by (rel-auto)

lemma and-runrest [unrest]: \( [\text{vwb-lens } x \ x \ x \ x \ x \ x \ x \ P \ x \ x \ x \ x \ x \ x \ x \ Q] \rightarrow x \ x \ x (P \ x \ x \ x \ x \ x \ x \ x \ Q) \)
by (metis RID-conj Healthy-def unrest-relation-def)

lemma or-runrest [unrest]: \( [\text{vwb-lens } x \ x \ x \ x \ x \ x \ x \ P \ x \ x \ x \ x \ x \ x \ x \ Q] \rightarrow x \ x \ x (P \ x \ x \ x \ x \ x \ x \ x \ Q) \)
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-upperbound utp-pred-laws.bot.extremum-unique)

lemma nu-const: \((\nu X \cdot P) = P\)
by (simp add: lfp-const)

lemma mu-refine-intro:
assumes \((C \Rightarrow S) \subseteq F(C \Rightarrow S) \ (C \ x \ x \ x F) = (C \ x \ x \ x F)\)
shows \((C \Rightarrow S) \subseteq \mu F\)
proof
  from assms have \((C \Rightarrow S) \subseteq \nu F\)
  by (simp add: lfp-upperbound)
with assms show \(?thesis\)
  by (pred-auto)
qed

16.2 Obtaining Unique Fixed-points

Obtaining termination proofs via approximation chains. Theorems and proofs adapted from Chapter 2, page 63 of the UTP book [14].

type-synonym 'a chain = nat ⇒ 'a upred

definition chain :: 'a chain ⇒ bool where
\[
\text{chain } Y = (Y \ 0 = \text{false} \land (\forall \ i. \ Y (\text{Suc } i) \subseteq Y i))
\]

**Lemma** \(\text{chain0 [simp]}\):
\[
\text{chain } Y \implies Y \ 0 = \text{false}
\]
by (simp add: chain-def)

**Lemma** \(\text{chainI}\):
\[
\begin{align*}
&\text{assumes } Y \ 0 = \text{false} \land i. \ Y (\text{Suc } i) \subseteq Y i \\
&\text{shows } \text{chain } Y
\end{align*}
\]
using assms by (auto simp add: chain-def)

**Lemma** \(\text{chainE}\):
\[
\begin{align*}
&\text{assumes } \text{chain } Y \land i. \ [[ Y \ 0 = \text{false}; Y (\text{Suc } i) \subseteq Y i ] \implies P \\
&\text{shows } P
\end{align*}
\]
using assms by (simp add: chain-def)

**Lemma** \(\text{L274}\):
\[
\begin{align*}
&\text{assumes } \forall n. ((E n \land Y) = (E n \land Y)) \\
&\text{shows } (\prod (\text{range } E) \land Y) = (\prod (\text{range } E) \land Y) \\
&\text{using } \text{assms by (pred-auto)}
\end{align*}
\]

Constructive chains

**Definition** \(\text{constr} ::\)
\[
(\text{'a upred} \Rightarrow \text{'a upred}) \Rightarrow \text{'a chain} \Rightarrow \text{bool}
\]
where
\[
\text{constr } F \ E \leftarrow\rightarrow \text{chain } E \land (\forall X n. \ ((F(X) \land E(n + 1)) = (F(X \land E(n)) \land E (n + 1))))
\]

**Lemma** \(\text{constrI}\):
\[
\begin{align*}
&\text{assumes } \text{chain } E \land X n. \ ((F(X) \land E(n + 1)) = (F(X \land E(n)) \land E (n + 1))) \\
&\text{shows } \text{constr } F \ E \\
&\text{using } \text{assms by (auto simp add: constr-def)}
\end{align*}
\]

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** \(\text{chain-pred-terminates}\):
\[
\begin{align*}
&\text{assumes } \text{constr } F \ E \ \text{mono } F \\
&\text{shows } (\prod (\text{range } E) \land \mu F) = (\prod (\text{range } E) \land \nu F) \\
&\text{proof} -- \text{from assms have } \forall n. \ (E n \land \mu F) = (E n \land \nu F) \\
&\text{proof (rule-tac allI)} \\
&\text{fix } n \\
&\text{from assms show } (E n \land \mu F) = (E n \land \nu F) \\
&\text{proof (induct n)} \\
&\text{case } 0 \ \text{thus } ?\text{case by (simp add: constr-def)}
\end{align*}
\]

next
\[
\begin{align*}
&\text{case } (\text{Suc } n) \\
&\text{note hyp = this} \\
&\text{thus } ?\text{case}
\end{align*}
\]

proof --
\[
\begin{align*}
&\text{have } (E (n + 1) \land \mu F) = (E (n + 1) \land F (\mu F)) \\
&\text{using gfp-unfold[OF hyp(3)], THEN sym] by (simp add: constr-def)} \\
&\text{also from hyp have } \ldots = (E (n + 1) \land F (E n \land \mu F)) \\
&\text{by (metis conj-comm constr-def)} \\
&\text{also from hyp have } \ldots = (E (n + 1) \land F (E n \land \nu F)) \\
&\text{by simp} \\
&\text{also from hyp have } \ldots = (E (n + 1) \land \nu F) \\
&\text{by (metis (no-types, lifting) conj-comm constr-def lfp-unfold)}
\end{align*}
\]
ultimately show \( \text{thesis} \)
by \( \text{simp} \)
qed
qed
qed
thus \( \text{thesis} \)
by (auto intro: L274)
qed

**Theorem constr-fp-uniq:**

assumes \( \text{constr } F \ E \ \text{mono } F \prod (\text{range } E) = C \)
shows \( (C \land \mu F) = (C \land \nu F) \)
using \( \text{assms(1) assms(2) assms(3) chain-pred-terminates by blast} \)

### 16.3 Noetherian Induction Instantiation

Contribution from Yakoub Nemouchi. The following generalization was used by Tobias Nipkow and Peter Lammich in *Refine Monadic*.

**Lemma wf-fixp-unduct-pure-ueq-gen:**

assumes \( \text{fixp-unfold: } \text{fp } B = \text{B (fp } B) \)
and \( \text{WF: } \text{wf } R \)
and \( \text{induct-step:} \)
\( \forall f \ st. \left( \left[ \forall \text{st'. (st', st)} \in R \implies ((\text{Pre } \land \left[ \text{e} \right] \left[ \left[ \text{st'} \right] \right] \implies \text{Post}) \subseteq f) \right] \right) \)

\( \implies \text{fp } B = f \implies ((\text{Pre } \land \left[ \text{e} \right] \left[ \left[ \text{st} \right] \right] \implies \text{Post}) \subseteq (B \ f) \)
shows \( ((\text{Pre } \implies \text{Post}) \subseteq \text{fp } B) \)

**Proof**

\{ fix \( \text{st} \)
have \( ((\text{Pre } \land \left[ \text{e} \right] \left[ \left[ \text{st} \right] \right] \implies \text{Post}) \subseteq (\text{fp } B) \)
using \( \text{WF } \text{proof (induction rule: wf-induct-rule)} \)
case (less \( \text{x} \))
  hence \( (\text{Pre } \land \left[ \text{e} \right] \left[ \left[ \text{x} \right] \right] \implies \text{Post}) \subseteq B (\text{fp } B) \)
  by (rule induct-step, rel-blast, simp)
  then show ?case
    using \( \text{fixp-unfold by auto} \)
  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 ...

**Lemma refine-usubst-to-ueq:**

\( \text{vwb-lens } E \implies (\text{Pre } \Rightarrow \text{Post})[\text{st'}]/E \subseteq f[\text{st'}/E] = (((\text{Pre } \land \text{E } \in \text{st'}) \Rightarrow \text{Post}) \subseteq f) \)
by (rel-auto, metis \( \text{vwb-lens-wb wb-lens.get-pat} \))

By instantiation of \( (\text{?fp } \Rightarrow \text{?B } = \text{?B (?fp } \Rightarrow \text{?B)}) \): \( \forall f \ st. \left[ \forall \text{st', (st', st)} \in ?R \implies (\text{?Pre } \land \left[ \text{?e} \right] \left[ \left[ \text{st} \right] \right] \implies \text{?Post}) \subseteq f \right] \implies \text{?fp } \Rightarrow \text{?B } = f \right] \implies (\text{?Pre } \land \left[ \text{?e} \right] \left[ \left[ \text{st} \right] \right] \Rightarrow \text{?Post}) \subseteq ?B f \)
\( \Rightarrow (\text{?Pre } \Rightarrow ?Post) \subseteq \text{?fp } \Rightarrow \text{?B} \) with \( \mu \) and lifting of the well-founded relation we have ...

**Lemma mu-rec-total-pure-rule:**

assumes \( \text{WF: } \text{wf } R \)
and \( \text{mono } B \)
and \( \text{induct-step:} \)
\( \forall f \ st. \left[ (\text{Pre } \land \left[ \text{e}, \left[ \text{st} \right] \right] \in \text{E } \Rightarrow \text{Post}) \subseteq f \right] \)
proof  
\[\mu B = f \implies \left( \text{Pre} \land \left[ e \right] \prec_u \left[ \text{st} \right] \implies \text{Post} \right) \subseteq (Bf) \]

\[\text{shows (Pre }\implies \text{Post) } \subseteq \mu B \]

proof (rule wf-fixp-induct-pure-ucr-gen[where fp=\mu \text{ and Pre=}\text{Pre} \text{ and } B=B \text{ and } R=R \text{ and } e=e])

\[\text{show } \mu B = B \ (\mu B) \]

by (simp add: M def-gfp-unfold)

show \(\text{wf R}\)

by (fact WF)

\[\text{show } \bigwedge \text{f st. } \left( \bigwedge \text{st'} \right) \subseteq \left( \text{Pre} \land \left[ e \right] \prec_u \left[ \text{st} \right] \implies \text{Post} \right) \subseteq f \implies \mu B = f \]

\[\text{shows (Pre }\implies \text{Post) } \subseteq \nu B \]

proof (rule wf-fixp-induct-pure-ucr-gen[where fp=\nu \text{ and Pre=}\text{Pre} \text{ and } B=B \text{ and } R=R \text{ and } e=e])

\[\text{show } \nu B = B \ (\nu B) \]

by (simp add: M def-lfp-unfold)

show \(\text{wf R}\)

by (fact WF)

\[\text{show } \bigwedge \text{f st. } \left( \bigwedge \text{st'} \right) \subseteq \left( \text{Pre} \land \left[ e \right] \prec_u \left[ \text{st} \right] \implies \text{Post} \right) \subseteq f \implies \nu B = f \]

\[\text{by (rule induct-step, rel-simp, simp) \quad qed} \]

lemma nu-rec-total-pure-rule:

\[\text{assumes WF: \text{wf R}} \]

\[\text{and } \ M: \text{ mono B} \]

\[\text{and \quad induct-step:} \]

\[\bigwedge \text{f st. } \left( \bigwedge \text{st'} \right) \subseteq \left( \text{Pre} \land \left[ e \right] \prec_u \left[ \text{st} \right] \implies \text{Post} \right) \subseteq f \implies \nu B = f \]

\[\text{shows (Pre }\implies \text{Post) } \subseteq \nu B \]

proof (rule nu-rec-total-pure-rule[where R=R \text{ and } e=e], simp-all add: Jssm)

\[\text{show } \bigwedge \text{f st. } \left( \bigwedge \text{st'} \right) \subseteq \left( \text{Pre} \land \left[ e \right] \prec_u \left[ \text{st} \right] \implies \text{Post} \right) \subseteq f \implies \mu B = f \]

\[\text{by (simp add: M induct-step monoD order-subst2) \quad qed} \]

lemma nu-rec-total-utp-rule:

\[\text{assumes WF: \text{wf R}} \]

\[\text{and } \ M: \text{ mono B} \]

\[\text{and \quad induct-step:} \]

\[\bigwedge \text{f st. } \left( \bigwedge \text{st'} \right) \subseteq \left( \text{Pre} \land \left[ e \right] \prec_u \left[ \text{st} \right] \implies \text{Post} \right) \subseteq f \implies \nu B = f \]

\[\text{shows (Pre }\implies \text{Post) } \subseteq \nu B \]

proof (rule nu-rec-total-pure-rule[where R=R \text{ and } e=e], simp-all add: Jssm)

\[\text{show } \bigwedge \text{f st. } \left( \bigwedge \text{st'} \right) \subseteq \left( \text{Pre} \land \left[ e \right] \prec_u \left[ \text{st} \right] \implies \text{Post} \right) \subseteq f \implies \mu B = f \]

\[\text{by (simp add: M induct-step monoD order-subst2) \quad qed} \]
17 UTP Deduction Tactic

theory utp-deduct
imports utp-pred
begin

named-theorems uintro
named-theorems uelim
named-theorems udest

lemma uttrueI [uintro]: \([\text{true}]_e b\)
by (pred-auto)

lemma uopI [uintro]: \(f ([x]_e b) \Rightarrow [uop f x]_e b\)
by (pred-auto)

lemma bopI [uintro]: \(f ([x]_e b) ([y]_e b) \Rightarrow [bop f x y]_e b\)
by (pred-auto)

lemma tropI [uintro]: \(f ([x]_e b) ([y]_e b) ([z]_e b) \Rightarrow [trop f x y z]_e b\)
by (pred-auto)

lemma uconjI [uintro]: \([p]_e b; [q]_e b \Rightarrow [p \land q]_e b\)
by (pred-auto)

lemma uconjE [uelim]: \([p \land q]_e b; [p]_e b \Rightarrow P \] \Rightarrow P
by (pred-auto)

lemma uimpI [uintro]: \([p]_e b \Rightarrow [q]_e b \Rightarrow [p \Rightarrow q]_e b\)
by (pred-auto)

lemma uimpE [elim]: \([p \Rightarrow q]_e b; ([p]_e b \Rightarrow [q]_e b) \Rightarrow P \] \Rightarrow P
by (pred-auto)

lemma ushAllI [uintro]: \(\forall x \cdot [p(x)]_e b \Rightarrow [\forall x \cdot p(x)]_e b\)
by pred-auto

lemma ushExI [uintro]: \([p(x)]_e b \Rightarrow [\exists x \cdot p(x)]_e b\)
by pred-auto

lemma udeduct-tautI [uintro]: \([\land b. [p]_e b \Rightarrow 'p'\)
using taut.rep-eq by blast

lemma udeduct-refineI [uintro]: \([\land b. [q]_e b \Rightarrow [p]_e b \Rightarrow p \subseteq q\)
by pred-auto

lemma udeduct-eqI [uintro]: \([\land b. [p]_e b \Rightarrow [q]_e b; \land b. [q]_e b \Rightarrow [p]_e b \Rightarrow p = q\)
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: \((P \land Q)_e b \land [P]_e b = [R]_e b \implies ([P]_e b \land [P]_e b = [R]_e b)\)
  by pred-auto

lemma conj-imp-subst: \((P \land Q)_e b \land ([P]_e b \implies ([P]_e b = [R]_e b))) \implies ([P]_e b \land ([P]_e b \implies ([P]_e b = [R]_e b)))\)
  by pred-auto

lemma disj-imp-subst: \((Q \land (P \lor S))_e b \land ([Q]_e b \implies ([P]_e b = [R]_e b))) \implies ([Q \land (R \lor S)]_e b \land ([Q]_e b \implies ([P]_e b = [R]_e b)))\)
  by pred-auto

Simplifications on value equality

lemma ueq-expr: \([e_0]_e b = [e_1]_e b \implies [e_0 = u]_e b\)
  by pred-auto

lemma ueq-expr-trans: \((P \land e_0 = u e_1)_e b \land ([P]_e b \implies [e_1 = u e_2]_e b) \implies ([P]_e b \land ([P]_e b \implies [e_1 = u e_2]_e b))\)
  by pred-auto

lemma ueq-expr-trans2: \((P \land e_0 = u e_1)_e b \land ([P]_e b \implies [e_1 = u e_2]_e b) \implies ([P]_e b \land ([P]_e b \implies [e_1 = u e_2]_e b))\)
  by pred-auto

lemma uequality: \([Q]_e b \implies [P \land Q]_e b \implies [P \land R]_e b \implies [P \land Q]_e b\)
  by pred-auto

lemma ueq1: \([P]_e b \implies ([Q]_e b = [R]_e b) \implies ([P \land R]_e b \implies [P \land Q]_e b)\)
  by pred-auto

lemma ueq2: \([P]_e b \implies ([Q]_e b = [R]_e b) \land [Q]_e b = [R \land P]_e b \implies ([P]_e b \implies ([Q]_e b = [R]_e b))\)
  by pred-auto

lemma ueq3: \([P]_e b \implies ([Q]_e b = [R]_e b) \implies ([R \land P]_e b = [Q \land P]_e b)\)
  by pred-auto

The following allows simplifying the equality if \(P \Rightarrow Q = R\)

lemma ueq3-imp: \((\land b. [P]_e b \implies ([Q]_e b = [R]_e b)) \implies ((R \land P) = (Q \land P))\)
  by pred-auto

lemma ueq3-imp2: \((\land b. [P]_e b \implies ([Q]_e b = [R]_e b)) \implies ((P \land Q) = (P \land R))\)
  by pred-auto
lemma \text{ueqe3-imp2}: \[(\bigwedge b. [P_0 \land P_1]_e b \implies [Q]_e b \implies [R]_e b = [S]_e b)] \implies ((P_0 \land P_1 \land (Q \Rightarrow R)) = (P_0 \land P_1 \land (Q \Rightarrow S))\]
by \text{pred-auto}

The following can introduce the binding notation into predicates

lemma \text{conj-bind-dist}: [P \land Q]_e b = ([P]_e b \land [Q]_e b)
by \text{pred-auto}

lemma \text{disj-bind-dist}: [P \lor Q]_e b = ([P]_e b \lor [Q]_e b)
by \text{pred-auto}

lemma \text{imp-bind-dist}: [P \Rightarrow Q]_e b = ([P]_e b \rightarrow [Q]_e b)
by \text{pred-auto}

end

18 Relational Calculus Laws

thory \text{utp-rel-laws}

imports
\text{utp-rel}
\text{utp-recursion}

begin

18.1 Conditional Laws

lemma \text{comp-cond-left-distr}:
(P @ b \triangleright_r Q) ;; R = (P ;; R) @ b \triangleright_r (Q ;; R)
by \text{rel-auto}

lemma \text{cond-seq-left-distr}:
out \alpha \neq b \implies ((P @ b \triangleright Q) ;; R) = (P ;; R) @ b \triangleright (Q ;; R)
by \text{rel-auto}

lemma \text{cond-seq-right-distr}:
in \alpha \neq b \implies (P ;; (Q @ b \triangleright R)) = ((P ;; Q) @ b \triangleright (P ;; R))
by \text{rel-auto}

Alternative expression of conditional using assumptions and choice

lemma \text{rcond-rassume-expand}: P @ b \triangleright_r Q = ([b]^\top ;; P) \sqcap ([\neg b]^\top ;; Q)
by \text{rel-auto}

18.2 Precondition and Postcondition Laws

theorem \text{precond-equiv}:
P = (P ;; \text{true}) \iff (\text{out} \alpha \neq P)
by \text{rel-auto}

theorem \text{postcond-equiv}:
P = (\text{true} ;; P) \iff (\text{in} \alpha \neq P)
by \text{rel-auto}

lemma \text{precond-right-unit}:
out \alpha \neq p \implies (p ;; \text{true}) = p
by \text{metis \text{precond-equiv}}

108
lemma postcond-left-unit: inα ♯ p =⇒ (true ;; p) = p
by (metis postcond-equiv)

theorem precond-left-zero:
assumes outα ♯ p p ≠ false
shows (true ;; p) = true
using assms by (rel-auto)

theorem feasibile-iff-true-right-zero:
P ;; true = true ⇐⇒ ∃ outα · P
by (rel-auto)

18.3 Sequential Composition Laws

lemma seqr-associ: (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]:
false ;; P = false
by pred-auto

lemma seqr-right-zero [simp]:
P ;; false = false
by pred-auto

lemma impl-seqr-mono: [P ⇒ Q; R ⇒ S] =⇒ (P ;; R) ⇒ (Q ;; S)
by (pred-blast)

lemma seqr-mono:
[ P1 ⊑ P2; Q1 ⊑ Q2 ] =⇒ (P1 ;; Q1) ⊑ (P2 ;; Q2)
by (rel-blast)

lemma seqr-monotonic:
[ mono P; mono Q ] =⇒ mono (λ X. P X ;; Q X)
by (simp add: mono-def, rel-blast)

lemma Monotonic-seqr-tail [closure]:
assumes Monotonic F
shows Monotonic (λ X. P ;; F(X))
by (simp add: assms monoD monoI seqr-mono)

lemma seqr-exists-left: ((∃ $x · P) ;; Q) = (∃ $x · (P ;; Q))
by (rel-auto)

lemma seqr-exists-right: (P ;; (∃ $x · Q)) = (∃ $x · (P ;; Q))
by (rel-auto)
lemma seqr-or-distl:
((P ∨ Q) ;; R) = ((P ;; R) ∨ (Q ;; R))
by (rel-auto)

lemma seqr-or-distr:
(P ;; (Q ∨ R)) = ((P ;; Q) ∨ (P ;; R))
by (rel-auto)

lemma seqr-and-distr-ufreq:
ufuncional P =⇒ ((P ;; Q) ∧ (R ;; (Q ∧ R)))
by (rel-auto)

lemma seqr-and-distl-uinj:
uinj R =⇒ ((P ∧ Q) ;; R) = ((P ;; R) ∧ (Q ;; R))
by (rel-auto)

lemma seqr-unfold:
((P ;; Q) = (∃ v · P[≪v≫/v] ∧ Q[≪v≫/v]))
by (rel-auto)

lemma seqr-middle:
assumes vwb-lens x
shows (P ;; (Q ;; R)) = ((∃ v · P[≪v≫/x] ;; Q[≪v≫/x])
using assms
by (rel-auto', metis vwb-lens-wb wb-lens, source-stability)

lemma seqr-left-one-point:
assumes vwb-lens x
shows ((P ∧ $x' = u ≪v≫) ;; Q) = (P[≪v≫/x] ;; Q[≪v≫/x])
using assms
by (rel-auto, metis vwb-lens-wb wb-lens.get-pat)

lemma seqr-right-one-point:
assumes vwb-lens x
shows (P ;; ($x = u ≪v≫ ∧ Q)) = (P[≪v≫/x] ;; Q[≪v≫/x])
using assms
by (rel-auto, metis vwb-lens-wb wb-lens.get-pat)

lemma seqr-left-one-point-true:
assumes vwb-lens x
shows ((P ∧ $x') ;; Q) = (P[true/sx'] ;; Q[true/sx])
by (metis assms seqr-left-one-point true-alt-def upred-eq-true)

lemma seqr-left-one-point-false:
assumes vwb-lens x
shows ((P ∧ ¬$x') ;; Q) = (P[false/sx'] ;; Q[false/sx])
by (metis assms false-alt-def seqr-left-one-point upred-eq-false)

lemma seqr-right-one-point-true:
assumes vwb-lens x
shows (P ;; ($x ∧ Q)) = (P[true/sx] ;; Q[true/sx])
by (metis assms seqr-right-one-point true-alt-def upred-eq-true)

lemma seqr-right-one-point-false:
assumes vwb-lens x
shows (P ;; (∼$x \land Q)) = (P[false/$x \mapsto]$ ;; Q[false/$x])
by (metis assms false-all-def seqr-right-one-point upred-eq-false)

lemma seqr-insert-ident-left:
assumes vwb-lens x $x' \not\in P \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 seqr-insert-ident-right:
assumes vwb-lens x $x' \not\in P \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 seqr-bool-split:
assumes vwb-lens x $x' \not\in Q
shows P ; Q = (P[true/$x \mapsto] ;; Q[true/$x] \lor P[false/$x'] ;; Q[false/$x])
using assms
by (subt seqr-middle[of x], simp-all add: true-alt-def false-alt-def)

lemma cond-inter-var-split:
assumes vwb-lens x
shows (P < $x' > Q) ;; R = (P[true/$x \mapsto] ;; R[true/$x] \lor Q[false/$x'] ;; R[false/$x])
proof –
  have (P < $x' > Q) ;; R = ((x' \land P) ;; R \lor (∼ x' \land Q) ;; R)
    by (simp add: cond-def seqr-or-distl)
  also have "... = (((P \land x') ;; R \lor (Q \land ∼ x) ;; R)
    by (rel-auto)
  also have "... = (P[true/$x \mapsto] ;; 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 seqr-pre-transfer: inα α q \to (P \land q) ;; R = (P ;; (q ∼ \land R))
by (rel-auto)

theorem seqr-pre-transfer':
((P \land [q]_>) ;; R) = (P ;; ([q]_< \land R))
by (rel-auto)

theorem seqr-post-out: inα α r \to (P ;; (Q \land r)) = ((P ;; Q) \land r)
by (rel-blast)

lemma seqr-post-var-out:
fixes x :: (bool \to 'a)
shows (P ;; (Q \land $x')) = ((P ;; Q) \land $x')
by (rel-auto)

111
**Theorem** seqr-post-transfer: \( \text{out} \alpha \# q \implies (P :: (q \land R)) = ((P \land q^-) :: R) \)
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 \((\exists x \cdot P x) ;; Q\) = \(\exists x :: (P ;; Q)\)
by (rel-auto)

**Lemma** seqr-true-lemma:
\( (P = (\neg ((\neg P) :: \text{true}))) = (P = (P :: \text{true})) \)
by (rel-auto)

**Lemma** seqr-to-conj: \[ \text{out} \alpha \# P; \text{in} \alpha \# Q \] = \( (P ;; Q) \)
by (metis postcond-left-unit seqr-pre-out utp-pred-laws inf-top right-neutral)

**Lemma** shEx-lift-seq-1 [uquant-lift]:
\( (\exists x \cdot P x) ;; Q\) = \(\exists x :: (P ;; Q)\)
by pred-auto

**Lemma** shEx-lift-seq-2 [uquant-lift]:
\( P ;; (\exists x \cdot Q x)\) = \(\exists x :: (P ;; Q x)\)
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-distr: \( P ;; (\bigcap A) = (\bigcap Q \in A \cdot P ;; Q) \)
by (transfer, auto)

**Lemma** seq-Sup-distr: \( \bigcap A :: Q = (\bigcap P \in A \cdot P :: Q) \)
by (transfer, auto)

**Lemma** seq-UINF-distl: \( P ;; (\bigcap Q \in A \cdot F(Q)) = (\bigcap Q \in A \cdot P ;; F(Q)) \)
by (simp add: UINF-as-Sup-collect seq-Sup-distl)

**Lemma** seq-UINF-distl': \( P ;; (\bigcap Q \cdot F(Q)) = (\bigcap Q \cdot P ;; F(Q)) \)
by (metis UINF-mem-UNIV seq-UINF-distl)

**Lemma** seq-UINF-distr: \( \bigcap P \in A \cdot F(P) :: Q = (\bigcap P \in A \cdot F(P) :: Q) \)
by (simp add: UINF-as-Sup-collect seq-Sup-distr)

**Lemma** seq-UINF-distr': \( \bigcap P \cdot F(P) ;; Q = (\bigcap P \cdot F(P) ;; Q) \)
by (metis UINF-mem-UNIV seq-UINF-distr)

**Lemma** seq-SUP-distl: \( P ;; (\bigcap i \in A. Q(i)) = (\bigcap i \in A. P ;; Q(i)) \)
by (metis image-image seq-Sup-distl)

112
lemma seq-SUP-distr: \( (\bigcap i \in A. P(i)) ;; Q = (\bigcap i \in A. P(i) ;; Q) \)
by (simp add: seq-Sup-distr)

18.6 Skip Laws

lemma cond-skip: out \( a \triangleleft b \rightarrow (b \land II) = (II \land b^-) \)
by (rel-auto)

lemma pre-skip-post: \( ([b]_* \land II) = (II \land [b]_*) \)
by (rel-auto)

lemma skip-var:
  fixes \( x :: (bool \rightarrow 'a) \)
  shows \( (\$x \land II) = (II \land \$x') \)
by (rel-auto)

lemma skip-r-unfold: \( \langle \sigma \rangle a = II \Rightarrow (\langle \sigma \circ \sigma \rangle a) = II \)
by (rel-simp, metis mwb-lens.put-put vwb-lens-wb wb-lens.get-put)

lemma skip-r-alpha-eq: \( II = (\$v' =_u \$v) \)
by (rel-auto)

lemma skip-ra-unfold: \( II x y = (\langle \sigma \circ \sigma \rangle a) = II y \)
by (rel-auto)

lemma skip-res-as-ra:
  \( \langle \sigma \rangle a = II \Rightarrow \langle \sigma \rangle a x = II y \)
apply (rel-auto)
  apply (metis (no-types, lifting) lens-indep-def)
done

18.7 Assignment Laws

lemma assigns-subst [usubst]: \( [\sigma]_s \triangleright (\varphi)_a = (\varphi \circ \sigma)_a \)
by (rel-auto)

lemma assigns-r-comp: \( ((\sigma)_a ;; P) = ([\sigma]_s \triangleright P) \)
by (rel-auto)

lemma assigns-r-feasible: \( ((\sigma)_a ;; \text{true}) = \text{true} \)
by (rel-auto)

lemma assign-subst [usubst]:
  \( \langle \text{mwb-lens } x, \text{mwb-lens } y \rangle \Rightarrow \langle \text{mwb-lens } x, \text{mwb-lens } y \rangle = [u \mapsto (x := u) \mapsto y] = (x := u, [x := u] \triangleright v) \)
by (rel-auto)

lemma assign-vacuous-skip:
  assumes \( \text{vwb-lens } x \)
  shows \( x := \& x = II \)
using assms by rel-auto

**Lemma assign-simultaneous:**

**Assumes** \( \mathrm{wblens} ~ y ~ x \bowtie y \)**

**Shows** \( (x; y) := (e, \& y) = (x := e) \)**

by (**simp add**: assms usubst-upd-comm usubst-upd-var-id)

**Lemma assign-idem:** \( \mathrm{mwb-lens} ~ x \equiv \Rightarrow (x, x) := (u \bowtie v) = (x := v) \)**

by (**simp add**: usubst)

**Lemma assign-comp:** \( ((f)_a ; (g)_a) = (g \circ f)_a \)**

by (**simp add**: assigns-r-comp usubst)

**Lemma assign-cond:** \( ((f)_a \bowtie b \triangleright_{r} (g)_a) = (f \bowtie b \triangleright_{s} g)_a \)**

by (**rel-auto**)

**Lemma assign-r-comp:** \( \mathrm{bij} ~ f \equiv \Rightarrow (f)_a^{-1} = (\mathrm{inv} f)_a \)**

by (**rel-auto, simp-all add**: bij-is-inj bij-is-surj surj-f-inv-f)

**Lemma assign-pred-transfer:**

**Fixes** \( x :: \left ( \alpha'_a = \Rightarrow \alpha'_a \right ) \)**

**Assumes** \( x \bowtie b \bowtie_{\alpha} b \)**

**Shows** \( (x := v) = (x := v \wedge b) \)**

using assms by (**rel-blast**)

**Lemma assign-r-comp:** \( x := u ;; P = P[[u]/x]] \)**

by (**simp add**: assigns-r-comp usubst alpha)

**Lemma assign-test:** \( \mathrm{mwb-lens} ~ x \equiv \Rightarrow (x := u \bowtie v) = (x := v) \)**

by (**simp add**: assigns-comp usubst)

**Lemma assign-twice:** \( \mathrm{mwb-lens} ~ x ;; x \bowtie f \) \( \Rightarrow (x := e ;; x := f) = (x := f) \)**

by (**simp add**: assigns-comp usubst unrest)

**Lemma assign-commute:**

**Assumes** \( x \bowtie y ~ x \bowtie f \bowtie y \equiv e \)**

**Shows** \( (x := e ;; y := f) = (y := f ;; x := e) \)**

using assms by (**rel-simp, simp-all add**: lens-indep-comm)

**Lemma assign-cond:**

**Fixes** \( x :: \left ( \alpha'_a = \Rightarrow \alpha'_a \right ) \)**

**Assumes** \( outa \bowtie b \)**

**Shows** \( (x := e ;; (P \bowtie b \triangleright Q)) \equiv ((x := e ;; P) \bowtie (b[[e]/x]]) \diamond (x := e ;; Q))) \)**

by (**rel-auto**)

**Lemma assign-r-cond:**

**Fixes** \( x :: \left ( \alpha'_a = \Rightarrow \alpha'_a \right ) \)**

**Shows** \( (x := e ;; (P \bowtie b \triangleright_{r} Q)) \equiv ((x := e ;; P) \bowtie (b[[e]/x])) \triangleright_{r} (x := e ;; Q))) \)**

by (**rel-auto**)

**Lemma assign-r-alt-def:**

**Fixes** \( x :: \left ( \alpha'_a = \Rightarrow \alpha'_a \right ) \)**

**Shows** \( x := v = H[[v]/x]] \)**

114
by (rel-auto)

lemma assigns-r-ufunc: ufunctional \langle f \rangle_a
by (rel-auto)

lemma assigns-r-uinj: inj f \Longrightarrow uinj \langle f \rangle_a
by (rel-simp, simp add: inj-eq)

lemma assigns-r-swap-uinj:
\[ \lbrack \text{vwb-lens } x; \text{vwb-lens } y; x \bowtie y \rbrack \Longrightarrow uinj ((x,y) := (ky,kx)) \]
by (metis assigns-r-uinj pr-var-def swap-usubst-inj)

lemma assign-unfold:
\[ \text{vwb-lens } x \Longrightarrow (x := v) = (\$x = u [v] < \land II|_\alpha x) \]
apply (rel-auto, auto simp add: comp-def)
using vwb-lens.put-eq by fastforce

18.8 Converse Laws

lemma convr-invol [simp]: \( p \neg \neg = p \)
by pred-auto

lemma lit-convr [simp]: \( \ll v \gg \neg \neg = \ll v \gg \)
by pred-auto

lemma uivar-convr [simp]:
fixes \( x :: (a \Longrightarrow \alpha) \)
shows \( \ll x \gg \neg = \ll x \gg \)
by pred-auto

lemma uovar-convr [simp]:
fixes \( x :: (a \Longrightarrow \alpha) \)
shows \( \ll x \gg \neg = \ll x \gg \)
by pred-auto

lemma uop-convr [simp]: \( \text{uop } f u \neg = \text{uop } f (u \neg) \)
by (pred-auto)

lemma bop-convr [simp]: \( \text{bop } f u v \neg = \text{bop } f (u \neg) (v \neg) \)
by (pred-auto)

lemma eq-convr [simp]: \( p =_u q \neg = (p \neg =_u q \neg) \)
by (pred-auto)

lemma not-convr [simp]: \( \neg p \neg = (\neg p \neg) \)
by (pred-auto)

lemma disj-convr [simp]: \( p \lor q \neg = (q \neg \lor p \neg) \)
by (pred-auto)

lemma conj-convr [simp]: \( p \land q \neg = (q \neg \land p \neg) \)
by (pred-auto)

lemma seqr-convr [simp]: \( p :: q \neg = (q \neg :: p \neg) \)
by (rel-auto)
lemma pre-convr \[\text{simp}]: \lceil p \rceil < = \lceil p \rfloor \\
by (rel-auto)

lemma post-convr \[\text{simp}]: \lceil p \rfloor > = \lceil p \rceil \\
by (rel-auto)

18.9 Assertion and Assumption Laws

declare sublens-def [lens-defs del]

lemma assume-false: \{false\} = \top \\
by (rel-auto)

lemma assume-true: \{true\} = \top \\
by (rel-auto)

lemma assume-seq: \{b\} ;; \{c\} = \{b \land c\} \\
by (rel-auto)

lemma assert-false: \{false\} = \top \\
by (rel-auto)

lemma assert-true: \{true\} = \top \\
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 [frame]: \Sigma;[P] = P \\
by (rel-auto)

lemma frame-none [frame]: 
\emptyset;[P] = (P \land \top) \\
by (rel-auto)

lemma frame-commute: 
assumes $y$ $P$ $y$ $P$ $x$ $Q$ $x$ $Q$ $x \bowtie y$
shows $x;[P] ;; y;[Q] ;; x;[P]$
apply (insert assms)
apply (rel-auto)
apply (rename-tac s s' s0)
apply (subgoal-tac (s $\oplus_L$ s' on y) $\oplus_L$ s0 on x = s0 $\oplus_L$ s' on y)
apply (metis lens-indep-get lens-indep-sym lens-override-def)
apply (simp add: lens-indep.lens-put-comm lens-override-def)
apply (rename-tac s s' s0)
apply (subgoal-tac put y (put x s (get x (put x s0 (get x s')))) (get y (put y s (get y s0))))
\quad = put x s0 (get x s'))
apply (metis lens-indep-get lens-indep-sym)
apply (metis lens-indep.lens-put-comm)
done

lemma frame-contract-RID:
assumes \( \text{wb-lens } x \text{ } P \) is RID \( x \bowtie y \)
shows \((x;y);[P] = y;[P]\)

proof –
from 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{wb-lens-wb } \text{wb-lens} \text{get-put})
  done
thus \(?thesis\)
  by (simp add: Healthy-if assms)
qed

lemma frame-miracle [simp]:
\(x;[false] = false\)
by (rel-auto)

lemma frame-skip [simp]:
\(\text{wb-lens } x \implies x;[\text{II}] = \text{II}\)
by (rel-auto)

lemma frame-assign-in [frame]:
\(\llbracket \text{wb-lens } a;\ x \leq L a \rrbracket \implies a;[x := v] = x := v\)
by (rel-auto, simp-all add: lens-get-put-quasi-commute lens-put-of-quotient)

lemma frame-conj-true [frame]:
\(\llbracket \{x,x\} ; P; \text{wb-lens } x \rrbracket \implies (P \land x;[true]) = x;[P]\)
by (rel-auto)

lemma frame-is-assign [frame]:
\(\text{wb-lens } x \implies x;[x = u \lceil v \rceil <] = x := v\)
by (rel-auto)

lemma frame-seq [frame]:
\(\llbracket \text{wb-lens } a;\ \{x,x\} ; P; \{x,x\} ; Q \rrbracket \implies x;[P ;; Q] = x;[P] ;; x;[Q]\)
apply (rel-auto)
apply (metis \text{mwb-lens.put-put} \text{wb-lens-mwb} \text{wb-lens-def} \text{weak-lens.put-get})
apply (metis \text{mwb-lens.put-put} \text{wb-lens-mwb})
done

lemma frame-to-antiframe [frame]:
\(\llbracket x \bowtie y; x + L y = 1_L \rrbracket \implies x;[P] = y;[P]\)
by (rel-auto, metis \text{mwb-indep-def}, \text{metis lens-indep-def surj-pair})

lemma rel-frext-miracle [frame]:
\(a;[false]^+ = false\)
by (rel-auto)

lemma rel-frext-skip [frame]:
\(\text{wb-lens } a \implies a;[\text{II}]^+ = \text{II}\)
by (rel-auto)

lemma rel-frext-seq [frame]:
\(\text{wb-lens } a \implies a;[P ;; Q]^+ = (a;[P]^+ ;; a;[Q]^+)\)
apply (rel-auto)
apply (rename-tac s s' so)
apply (rule-tac \( x=\text{\textit{put\_a}} \ s \ s_0 \in \textit{exI} \))
apply (auto)
apply (metis mwb-lens.put-put vwb-lens-mwb)
done

lemma \textit{rel-frext-assigns} [frame]:
\[ \text{vwb-lens} \ a \implies a:\{(\sigma \_a)^+\} = (\sigma \_s \ a\_a) \]
by (rel-auto)

lemma \textit{rel-frext-rcond} [frame]:
\[ a:[\text{P} \& b \triangleright \text{Q}]^+ = (a:[\text{P}]^+ \& b \triangleright \text{P} \& a \triangleright \text{Q})^+ \]
by (rel-auto)

lemma \textit{rel-frext-commute}:
\[ x \triangleright y \implies x:[\text{P}]^+ ;; y:[\text{Q}]^+ = y:[\text{Q}]^+ ;; x:[\text{P}]^+ \]
apply (rename-tac a c b)
apply (subgoal-tac \( \& \text{a get}_y (\text{put}_x \text{b a}) = \text{get}_y \text{b} \))
apply (metis (no-types, hide-lams) lens-indep-comm lens-indep-get)
apply (simp add: lens-indep.lens-put-irr2)
apply (subst lfp-unfold, simp-all add: m)
also have \ldots \ldots = ((\text{P} ;; (\nu \text{X} \cdot (\text{P} ;; \text{X}) \& b \triangleright \text{P}) \& b \triangleright \text{P}) ) \& b \triangleright \text{P} \]
apply (simp-ffun-unfold, simp-all add: m)
also have \ldots \ldots = ((\text{P} ;; \text{while} b \text{do} \text{P} \text{od}) \& b \triangleright \text{P}) \]
done

lemma \textit{antiframe-disj} [frame]:
\[ (x:[\text{P}] \lor x:[\text{Q}]) = x:[\text{P} \lor \text{Q}] \]
by (rel-auto)

lemma \textit{antiframe-seq} [frame]:
\[ (\lceil \text{vwb-lens} \ x ; x \triangleright \text{P} ; x \triangleright \text{Q} \rceil ) = (x:[\text{P}] ;; x:[\text{Q}]) = x:[\text{P} ;; \text{Q}] \]
apply (rel-auto)
apply (metis vwb-lens-wb wb-lens-def weak-lens.put-get)
apply (metis vwb-lens-wb wb-lens.put-twice wb-lens-def weak-lens.put-get)
done

lemma \textit{nameset-skip}:
\[ \text{vwb-lens} \ x \implies (\text{ns} \ x \cdot \text{II}) = \text{II} \]
by (rel-auto, meson vwb-lens-wb wb-lens-get-put)

lemma \textit{nameset-skip-ra}:
\[ \text{vwb-lens} \ x \implies (\text{ns} \ x \cdot \text{II} \ x) = \text{II} \]
by (rel-auto)

declare \textit{sublens-def} [lens-defs]

\textbf{18.11 While Loop Laws}

\textbf{theorem} \textit{while-unfold}:
\[ \text{while} \ b \text{ do} \text{P} \text{ od} = ((\text{P} ;; \text{while} b \text{ do} \text{P} \text{ od}) \& b \triangleright \text{P}) \]

\textbf{proof} –
\begin{itemize}
  \item \textbf{have} \( m:\text{mono} (\lambda X. (\text{P} ;; X) \& b \triangleright \text{P}) \)
  \begin{itemize}
    \item \textbf{by} (auto intro: monoI seqr-mono cond-mono)
  \end{itemize}
  \item \textbf{have} \( \text{while} \ b \text{ do} \text{P} \text{ od} = (\nu \text{X} \cdot (\text{P} ;; \text{X}) \& b \triangleright \text{P}) \)
  \begin{itemize}
    \item \textbf{by} (simp add: while-def)
  \end{itemize}
  \item \textbf{also have} \ldots \ldots = ((\text{P} ;; (\nu \text{X} \cdot (\text{P} ;; \text{X}) \& b \triangleright \text{P}) \& b \triangleright \text{P}) ) \& b \triangleright \text{P} \]
  \begin{itemize}
    \item \textbf{by} (subst lfp-unfold, simp-all add: m)
  \end{itemize}
  \item \textbf{also have} \ldots \ldots = ((\text{P} ;; \text{while} b \text{ do} \text{P} \text{ od}) \& b \triangleright \text{P}) \]
\end{itemize}

by (simp add: while-def)
finally show \?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\⊥\ b do P od = ((P ;; while\⊥\ b do P od) \isasymand b \isasymRightarrow II)
proof -
have m:mono (\lambda X. (P ;; X) \isasymand b \isasymRightarrow II)
  by (auto intro: monol seqr-mono cond-mono)
have (while\⊥\ b do P od) = (\mu X \cdot (P ;; X) \isasymand b \isasymRightarrow II)
  by (simp add: while-bot-def)
also have ... = ((P ;; (\mu X \cdot (P ;; X) \isasymand b \isasymRightarrow II)) \isasymand b \isasymRightarrow II)
  by (subst gfp-unfold, simp-all add: m)
also have ... = ((P ;; while\⊥\ b do P od) \isasymand b \isasymRightarrow II)
  by (simp add: while-bot-def)
finally show \?thesis .
qed

theorem while-bot-false: while\⊥\ false do P od = II
by (simp add: while-bot-def mu-const alpha)

theorem while-bot-true: while\⊥\ true do P od = (\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\ b = true \imp while\⊥\ 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_R 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 :: `(\alpha hrel \Rightarrow nat \Rightarrow \alpha hrel) (infixr \^) 80) where
  upower P n \equiv upred-semiring.power P n
translations
\[ P \mapsto i \leftarrow CONST \ power.power II op ;; P i \quad P \mapsto i \leftarrow (CONST \ power.power II op ;; P) i \]

Set up transfer tactic for powers

**lemma** upower-rep-eq:
\[ [P \mapsto i]e = (\lambda b. \ b \in (\{p. [P]e p\} \mapsto i)) \]
**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:
\[ [\text{power.power} \langle id \rangle_a op ;; P i]e = (\lambda b. \ b \in (\{p. [P]e p\} \mapsto i)) \]
by (metis skip-r-def upower-rep-eq)

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

**lemma** Sup-power-expand:
fixes P :: nat ⇒ ′a::complete-lattice
shows \( P(0) \cap (\bigcap i. P(i+1)) = (\bigcap i. P(i)) \)
**proof**
  have UNIV = insert (0::nat) {1..}
    by auto
  moreover have \( (\bigcap i. P(i)) = \bigcap (P \mapsto \text{UNIV}) \)
    by (blast)
  moreover have \( \bigcap (P \mapsto \text{insert 0 \{1..\}}) = P(0) \cap \text{SUPREMUM \{1..\} P} \)
    by (simp)
  moreover have \( \text{SUPREMUM \{1..\} P} = (\bigcap i. P(i+1)) \)
    by (simp add: atLeast-Suc-greaterThan greaterThan-0)
  ultimately show ?thesis
    by (simp only:)
qed

**lemma** Sup-upto-Suc:
\( (\bigcap i\in\{0..\text{Suc \ n}\}. P \mapsto i) = (\bigcap i\in\{0..\text{n}\}. P \mapsto i) \cap P \mapsto \text{Suc \ n} \)
**proof**
  have \( (\bigcap i\in\{0..\text{Suc \ n}\}. P \mapsto i) = (\bigcap i\in\text{insert (Suc \ n) \{0..\text{n}\}. P \mapsto i} \)
    by (simp add: atLeast0-atMost-Suc)
  also have \( \ldots = P \mapsto \text{Suc \ n} \cap (\bigcap i\in\{0..\text{n}\}. P \mapsto i) \)
    by (simp)
  finally show ?thesis
    by (simp add: Lattices.sup-commute)
qed

The following two proofs are adapted from the AFP entry Kleene Algebra. See also [2, 1].

**lemma** upower-inductl:
\( Q \subseteq (P ;; Q \cap R) \Rightarrow Q \subseteq P \mapsto \text{n ;; R} \)
**proof**(induct n)
  case 0
  then show ?case by (auto)
next
  case (Suc n)
  then show ?case
    by (auto simp add: upred-semiring.power-Suc, metis (no-types, hide-lams) dual-order.trans order-refl seqr-assoc seqr-mono)
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 seqr-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 (\prod{i \in {m..n}}. P(i)) = P(m) ∩ (\prod{i \in {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⟩ a ^ n = ⟨f ^ ^ n⟩ a
by (induct n, rel-auto+)

18.12.1 Kleene Star

definition ustar :: 'a hrel ⇒ 'a hrel (-^ [999] 999) where
  P^* = (\prod{i \in {0..}} · P^ i)

lemma ustar-rep-eq:
  [P^*] e = (λb. b ∈ ([p. [P] e] p)^*)
by (simp add: ustar-def, rel-auto, simp-all add: relpow-imp-rtrancl rtrancl-imp-relpow)

update-uexpr-rep-eq-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^+ = (\prod{i} · P ^ (Suc i))
by (simp add: uplus-def ustar-def seq-UINF-distl' UINF-atLeast-Suc upred-semiring.power-Suc)
18.14 Omega

definition oomega :: 'a hrel ⇒ 'a hrel (ω [999] 999) where
Pω = (μ X ⋆ X)

18.15 Relation Algebra Laws

theorem RA1: (P ;; (Q ;; R)) = ((P ;; Q) ;; R)
by (simp add: seqr-assoc)

theorem RA2: (P ;; II) = P (II ;; P) = P
by simp-all

theorem RA3: P⁺⁺ = P
by simp

theorem RA4: (P ;; Q)⁻ = (Q⁻ ;; P⁻)
by simp

theorem RA5: (P ∨ Q)⁻ = (P⁻ ∨ Q⁻)
by (rel-auto)

theorem RA6: ((P ∨ Q) ;; R) = (P ;; R ∨ Q ;; R)
using seqr-or-distl by blast

theorem RA7: ((P⁻ ;; (¬(P ;; Q))) ∨ (¬Q)) = (¬Q)
by (rel-auto)

18.16 Kleene Algebra Laws

lemma ustar-alt-def: P⋆ = (∏ i ⋆ i)
by (simp add: ustar-def)

theorem ustar-sub-unfoldl: P⋆ ⊑ II ⊓ P ;; P⋆
by (rel-simp, simp add: rtrancl-into-trancl2 trancl-into-rtrancl)

theorem ustar-inductl:
  assumes Q ⊑ R Q ⊑ P ;; Q
  shows Q ⊑ P⋆ ;; R
proof –
  have P⋆ ;; R = (∏ i ⋆ i ;; R)
  by (simp add: ustar-def UINF-as-Sup-collect' seq-SUP-distr)
  also have Q ⊑ ...
  by (simp add: SUP-least assms upower-inductl)
finally show ?thesis .
qed

theorem ustar-inductr:
  assumes Q ⊑ R Q ⊑ Q ;; P
  shows Q ⊑ R ;; P⋆
proof –
  have R ;; P⋆ = (∏ i ⋆ i ;; P ⋆)
  by (simp add: ustar-def UINF-as-Sup-collect' seq-SUP-distl)
  also have Q ⊑ ...
  by (simp add: SUP-least assms upower-inductr)
finally show ?thesis .
lemma ustar-refines-nu: \((\nu X \cdot P ;; X \cap I) \subseteq P^*\)
by (metis (no-types, lifting) lfp-greatest semilattice-sup-class.le-sup-iff
  semilattice-sup-class.sup-idem upred-semiring.multiplication-lift
  upred-semiring.onediv-one ustar-induct)

lemma ustar-as-nu: \(P^* = (\nu X \cdot P ;; X \cap I)\)
proof (rule antisym)
  show \((\nu X \cdot P ;; X \cap I) \subseteq P^*\)
  by (simp add: ustar-refines-nu)
  show \(P^* \subseteq (\nu X \cdot P ;; X \cap I)\)
  by (metis lfp-lowerbound upred-semiring.add-commute ustar-sub-unfold)
qed

lemma ustar-unfoldl: \(P^* = \Pi (P ;; P^*)\)
apply (simp add: ustar-as-nu)
apply (subst lfp-unfold)
apply (rule monotonous)
apply (rel-auto)
done

While loop can be expressed using Kleene star

lemma while-star-form:
while \(b\) do \(P\) od = \((P \triangleleft b \triangleright_r \Pi) \star \top\)
proof
  have 1: Continuous \((\lambda X. P ;; X \triangleleft b \triangleright_r \Pi)\)
  by (rel-auto)
  have while \(b\) do \(P\) od = \(\bigcap\ i. (\lambda X. P ;; X \triangleleft b \triangleright_r \Pi) \star i \top\) false
  by (simp add: 1 false-upred-def sup-continuous-Continuous sup-continuous-lfp while-def)
  also have \(\ldots = (\lambda X. P ;; X \triangleleft b \triangleright_r \Pi) \star (i+1)\) false \(\cap (\bigcap\ i. (\lambda X. P ;; X \triangleleft b \triangleright_r \Pi) \star (i+1)\) false
  by (subst Sup-power-expand, simp)
  also have \(\ldots = (\bigcap\ i. (\lambda X. P ;; X \triangleleft b \triangleright_r \Pi) \star (i+1)\) false
  by (simp)
  also have \(\ldots = (\bigcap\ i. (P \triangleleft b \triangleright_r \Pi) \star i \top. (false \triangleleft b \triangleright_r \Pi)\)
proof (rule SUP-cong, simp-alt)
  fix \(i\)
  show \(P ;; (\lambda X. P ;; X \triangleleft b \triangleright_r \Pi) \star i \top. (false \triangleleft b \triangleright_r \Pi)\) = \((P \triangleleft b \triangleright_r \Pi) \star i \top. (false \triangleleft b \triangleright_r \Pi)\)
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.matmul-left-neutral)
qed
qed

also have \(\ldots = (\bigcap\ i \in \{0..\} \cdot (P \triangleleft b \triangleright_r \Pi) \star i \top. \top\) false\)
by (rel-auto)
also have \(\ldots = (P \triangleleft b \triangleright_r \Pi) \star \top. \top\) false
by (metis seq-UINF-distr ustar-def)
finally show \?thesis .
qed
18.17 Omega Algebra Laws

lemma \textit{omega-induct}:
\[ P \subseteq P' \quad \text{by (simp add: omega-def, metis eq-refl gfp-unfold monoI seqr-mono)} \]

18.18 Refinement Laws

lemma \textit{skip-r-refine}:
\[(p \Rightarrow p) \subseteq \textit{II} \quad \text{by pred-blast} \]

lemma \textit{conj-refine-left}:
\[(Q \Rightarrow P) \subseteq R \Rightarrow P \subseteq (Q \land R) \quad \text{by (rel-auto)} \]

lemma \textit{pre-weak-rel}:
\[\text{assumes } \text{Pre} \Rightarrow \text{I} \quad \text{and } (I \Rightarrow \text{Post}) \subseteq P \quad \text{shows } (\text{Pre} \Rightarrow \text{Post}) \subseteq P \quad \text{using assms by (rel-auto)} \]

lemma \textit{cond-refine-rel}:
\[\text{assumes } S \subseteq ([b]_< \land P) \quad S \subseteq ([\neg b]_< \land Q) \quad \text{shows } S \subseteq P < b \lor_r Q \quad \text{by (metis aext-not assms(1) assms(2) cond-def lift-rcond-def utp-pred-laws.le-sup-iff)} \]

lemma \textit{seq-refine-pred}:
\[\text{assumes } ([b]_< \Rightarrow [s]_> \subseteq P \quad (s]_< \Rightarrow [c]_>) \subseteq Q \quad \text{shows } ([b]_< \Rightarrow [c]_>) \subseteq (P ;; Q) \quad \text{using assms by rel-auto} \]

lemma \textit{seq-refine-unrest}:
\[\text{assumes } \text{outa} \not\subseteq b \quad \text{ina} \not\subseteq c \quad \text{assumes } (b \Rightarrow [s]_> \subseteq P \quad ([s]_< \Rightarrow c) \subseteq Q \quad \text{shows } (b \Rightarrow c) \subseteq (P ;; Q) \quad \text{using assms by rel-blast} \]

18.19 Domain and Range Laws

lemma \textit{Dom-conv-Ran}:
\[\text{Dom}(P^-) = \text{Ran}(P) \quad \text{by (rel-auto)} \]

lemma \textit{Ran-conv-Dom}:
\[\text{Ran}(P^-) = \text{Dom}(P) \quad \text{by (rel-auto)} \]

lemma \textit{Dom-skip}:
\[\text{Dom}(\text{II}) = \text{true} \quad \text{by (rel-auto)} \]

lemma \textit{Dom(assigns)}:
\[\text{Dom}((\sigma)_a) = \text{true} \quad \text{by (rel-auto)} \]

124
lemma Dom-miracle:
\( \text{Dom}(\text{false}) = \text{false} \)
by (rel-auto)

lemma Dom-assume:
\( \text{Dom}([b]^{\top}) = b \)
by (rel-auto)

lemma Dom-seq:
\( \text{Dom}(P ;; Q) = \text{Dom}(P ;; [\text{Dom}(Q)]^{\top}) \)
by (rel-auto)

lemma Dom-disj:
\( \text{Dom}(P \lor Q) = (\text{Dom}(P) \lor \text{Dom}(Q)) \)
by (rel-auto)

lemma Dom-inf:
\( \text{Dom}(P \sqcap Q) = (\text{Dom}(P) \lor \text{Dom}(Q)) \)
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 :: ('a 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 :: 'a upred set

show \( \exists s. \text{is-lub \{carrier = UNIV, eq = op =, le = op ≤\} s A} \)
apply (rule-tac x=∪ A in exI)
apply (rule least-UpperI)
apply (auto intro: Inf-greatest simp add: Inf-lower Upper-def)
done

show \( \exists i. \text{is-glb \{carrier = UNIV, eq = op =, le = op ≤\} i A} \)
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 \( P \)
by (simp add: upred-lattice.weak.weak-complete-lattice-axioms)

lemma upred-lattice-eq [simp]:
\( op \vdash \alpha = \alpha \)
by (simp add: upred-lattice-def)

lemma upred-lattice-le [simp]:
\( le \vdash P Q = (P \subseteq Q) \)
by (simp add: upred-lattice-def)

lemma upred-lattice-carrier [simp]:
carrier \( P = \text{UNIV} \)
by (simp add: upred-lattice-def)

lemma Healthy-fixed-points [simp]: fps \( P H = \{[H]\} \)
by (simp add: fps-def upred-lattice-def Healthy-def)

lemma upred-lattice-Idempotent [simp]: Idem \( P H = \text{Idempotent } H \)
using upred-lattice.weak-partial-order-axioms by (auto simp add: idempotent-def Idempotent-def)

lemma upred-lattice-Monotonic [simp]: Mono \( P H = \text{Monotonic } H \)
using upred-lattice.weak-partial-order-axioms by (auto simp add: isotone-def mono-def)

19.2 UTP theories hierarchy

typedef \( (\alpha, \beta) \text{ uthy} = \text{UNIV :: unit set} \)
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 :: \( (\alpha, \beta) \text{ uthy} \)
where
\( \text{uthy} = \text{Abs-uthy } \)

lemma uthy-eq [intro]:
fixes \( x y :: (\alpha, \beta) \text{ uthy} \)
shows \( x = y \)
by (cases \( x \), cases \( y \), simp)

syntax
\( -\text{UTHY} :: \text{type } \Rightarrow \text{type } \Rightarrow \text{logic } (\text{UTHY}(\cdot, \cdot)) \)

translations
\( \text{UTHY}(\alpha, \beta) = \text{CONST uthy :: (\alpha, \beta) uthy} \)

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
  utp-hcond :: ('T, 'α) uthy ⇒ ('α × 'α) health (H)

definition utp-order :: ('α × 'α) health ⇒ 'α hrel gorder where
  utp-order H = (\{ carrier = \{ P. P is H \}, eq = (op =), le = op ⊑ \})

abbreviation uthy-order T ≡ utp-order H T

Constant utp-order obtains the order structure associated with a UTP theory. Its carrier is the
set of healthy predicates, equality is HOL equality, and the order is refinement.

lemma utp-order-carrier [simp]:
  carrier (utp-order H) = [H] H
  by (simp add: utp-order-def)

lemma utp-order-eq [simp]:
  eq (utp-order T) = op =
  by (simp add: utp-order-def)

lemma utp-order-le [simp]:
  le (utp-order T) = op ⊑
  by (simp add: utp-order-def)

lemma utp-partial-order: partial-order (utp-order T)
  by (unfold-locales, simp-all add: utp-order-def)

lemma utp-weak-partial-order: weak-partial-order (utp-order T)
  by (unfold-locales, simp-all add: utp-order-def)

lemma mono-Monotone-utp-order:
  mono f ⇒ Monotone (utp-order T) f
  apply (auto simp add: isotone-def)
  apply (metis partial-order-def utp-partial-order)
  apply (metis monoD)
  done

lemma isotone-utp-orderI: Monotonic H ⇒ isotone (utp-order X) (utp-order Y) H
  by (auto simp add: mono-def isotone-def utp-weak-partial-order)

lemma Mono-utp-orderI:
  \[ ∧ P Q. \{ P ⊆ Q; P is H; Q is H \} ⇒ F(P) ⊆ F(Q) \] ⇒ Mono 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 utp-order-fpl: utp-order H = fpl P H
  by (auto simp add: utp-order-def upred-lattice-def fps-def Healthy-def)

definition uth-eq :: ('T, 'α) uthy ⇒ ('T, 'α) uthy ⇒ bool (infix ≈ T 50) where
  T1 ≈T T2 ≜ [H T1] H = [H T2] H

lemma uth-eq-refl: T ≈T T
  by (simp add: uth-eq-def)
lemma uth-eq-sym: $T_1 \approx_T T_2 \iff T_2 \approx_T T_1$
  by (auto simp add: uth-eq-def)

lemma uth-eq-trans: $\left( T_1 \approx_T T_2 ; T_2 \approx_T T_3 \right) \Rightarrow T_1 \approx_T T_3$
  by (auto simp add: uth-eq-def)

definition uthy-plus :: $\left( T_1, '\alpha \right)$ uthy $\Rightarrow$ $\left( T_2, '\alpha \right)$ uthy (infixl $+^T$ 65) where
  uthy-plus $T_1 T_2 = uthy$

overloading
  prod-hcond $\Leftarrow=$ utp-hcond :: $\left( T_1 \times T_2, '\alpha \right)$ uthy $\Rightarrow$ $\left( '\alpha \times '\alpha \right)$ upred

begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition prod-hcond :: $\left( T_1 \times T_2, '\alpha \right)$ uthy $\Rightarrow$ $\left( '\alpha \times '\alpha \right)$ upred where
  prod-hcond $T = H_{UTHY('T_1, '\alpha)} \circ H_{UTHY('T_2, '\alpha)}$

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 $T :: \left( T_1, '\alpha \right)$ 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 ==> T +_T T T \approx T 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 (⊤₁)
where utp-top T ≡ top (uthy-order T)

abbreviation utp-bottom (⊥₁)
where utp-bottom T ≡ bottom (uthy-order T)

abbreviation utp-join (infixl ⊔ 65) where
utp-join T ≡ join (uthy-order T)

abbreviation utp-meet (infixl ⊓ 70) where
utp-meet T ≡ meet (uthy-order T)

abbreviation utp-sup (⨆-\textsubscript{[0,10])} where
utp-sup T ≡ Lattice.sup (uthy-order T)

abbreviation utp-inf (⨅\textsubscript{[0,10])} where
utp-inf T ≡ Lattice.inf (uthy-order T)

abbreviation utp-gfp (ν₁) where
utp-gfp T ≡ GREATEST-FP (uthy-order T)

abbreviation utp-lfp (µ₁) where
utp-lfp T ≡ LEAST-FP (uthy-order T)

syntax
-tnu :: logic ⇒ pttrn ⇒ logic ⇒ logic (µ₁ - · - [0, 10])

-tnu :: logic ⇒ pttrn ⇒ logic ⇒ logic (ν₁ - · - [0, 10])

notation gfp (µ)
notation lfp (ν)

translations
µ_T X \cdot P == CONST utp-lfp T (λ X. P)
ν_T X \cdot P == CONST utp-gfp T (λ X. P)

lemma upred-lattice-inf:
Lattice.inf \mathcal{P} A = \sqcap A
by (metis Sup-least Sup-upper UNIV-I antisym-coneI upred-lattice.weak.inf-greatest upred-lattice.weak.inf-lower upred-lattice-carrier upred-lattice-le)

We can then derive a number of properties about these operators, as below.

context utp-theory-lattice
begin

lemma LFP-healthy-comp: µ F = µ (F o H)

129
proof  
  have \{ P. (P is H) \land F \downarrow P \} = \{ P. (P is H) \land F (H P) \downarrow P \}  
  by (auto simp add: Healthy-def)  
  thus \{thesis  
  by (simp add: LEAST-FP-def)  
qed  

lemma GFP-healthy-comp: \( \nu \circ F = \nu (F \circ H) \)  
proof  
  have \{ P. (P is H) \land P \downarrow F P \} = \{ P. (P is H) \land P \downarrow F (H P) \}  
  by (auto simp add: Healthy-def)  
  thus \{thesis  
  by (simp add: GREATEST-FP-def)  
qed  

lemma top-healthy [closure]: \( \top \) is H  
using weak.top-closed by auto  

lemma bottom-healthy [closure]: \( \bot \) is H  
using weak.bottom-closed by auto  

lemma utp-top: P is H \implies P \subseteq \top  
using weak.top-higher by auto  

lemma utp-bottom: P is H \implies \bot \subseteq P  
using weak.bottom-lower by auto  

end  

lemma upred-top: \( \top \ P = \text{false} \)  
using ball-UNIV greatest-def by fastforce  

lemma upred-bottom: \( \bot \ P = \text{true} \)  
by fastforce  

One way of obtaining a complete lattice is showing that the healthiness conditions are monotone, which the below locale characterises.  
locale utp-theory-mono = utp-theory +  
assumes HCond-Mono [closure, intro]: Monotonic H  

sublocale utp-theory-mono \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 H  
by (rule Knaster-Tarski, auto simp add: upred-lattice.weak.weak-complete-lattice-axioms)  

have complete-lattice (fpl P 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: \( T = \mathcal{H}(\text{false}) \)

proof
  have \( T = \top \) \text{fpl} \( \mathcal{P} \mathcal{H} \)  
    by (simp add: utp-order-fpl)
  also have \( \ldots = \mathcal{H} \top \) 
    using Knaster-Tarski-idem-extremes(1)[of \( \mathcal{P} \mathcal{H} \)] 
    by (simp add: HCond-Idempotent HCond-Mono)
  also have \( \ldots = \mathcal{H} \text{false} \) 
    by (simp add: upred-top)
  finally show \( \ldots \).

qed

lemma healthy-bottom: \( \bot = \mathcal{H}(\text{true}) \)

proof
  have \( \bot = \bot \) \text{fpl} \( \mathcal{P} \mathcal{H} \)  
    by (simp add: utp-order-fpl)
  also have \( \ldots = \mathcal{H} \bot \) 
    using Knaster-Tarski-idem-extremes(2)[of \( \mathcal{P} \mathcal{H} \)] 
    by (simp add: HCond-Idempotent HCond-Mono)
  also have \( \ldots = \mathcal{H} \text{true} \) 
    by (simp add: upred-bottom)
  finally show \( \ldots \).

qed

lemma healthy-inf:

assumes \( A \subseteq [\mathcal{H}]_H \)

shows \( \bigwedge A = \mathcal{H} (\bigwedge A) \)

proof
  have \( 1: \text{weak-complete-lattice} (\mathcal{U} \mathcal{H}) \)
    by (simp add: weak.weak-complete-lattice-axioms)
  have \( 2: \text{Mono}_{\mathcal{U} \mathcal{H}} \mathcal{H} \)
    by (simp add: HCond-Mono isotone-utp-orderI)
  have \( 3: \text{Idem}_{\mathcal{U} \mathcal{H}} \mathcal{H} \)
    by (simp add: HCond-Idem idempotent-def)
  show \( \ldots \)
    using Knaster-Tarski-idem-inf-eq[OF upred-weak-complete-lattice, of \( \mathcal{H} \)] 
    by (simp,metis HCond-Idempotent HCond-Mono assms partial-object.simps(3) upred-lattice-def
upred-lattice-inf utp-order-def)

qed

end

locale utp-theory-continuous = utp-theory +

assumes HCond-Cont [closure,intro]: Continuous \( \mathcal{H} \)

sublocale utp-theory-continuous \( \subseteq \) utp-theory-mono

proof

end
show Monotonic $\mathcal{H}$
by (simp add: Continuous-Monotonic HCond-Cont)
qed

context utp-theory-continuous
begin

lemma healthy-inf-cont:
assumes $A \subseteq \{H\} H A \neq \{\}$
shows $\bigsqcap A = \bigsqcap A$
proof
  have $\bigsqcap A = \bigsqcap (H' A)$
    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 $\text{thesis}$.
qed

lemma healthy-inf-def:
assumes $A \subseteq \{H\} H$
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 $P$ is $H$ $Q$ is $H$
shows $P \sqcap Q = P \sqcap Q$
using healthy-inf-cont[of $\{P, Q\}$] assms
by (simp add: Healthy-if meet-def)

lemma meet-is-healthy [closure]:
assumes $P$ is $H$ $Q$ is $H$
shows $P \sqcap Q$ is $H$
by (metis Continuous-Disjunctous Disjunctuous-def HCond-Cont Healthy-def' assms(1) assms(2))

lemma meet-bottom [simp]:
assumes $P$ is $H$
shows $P \sqcap \bot = \bot$
by (simp add: assms semilattice-sup-class.sup-absorb2 utp-bottom)

lemma meet-top [simp]:
assumes $P$ is $H$
shows $P \sqcap \top = P$
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 $F$ $F \in \{H\} H \rightarrow \{H\} H$
shows $\mu F = (\mu X \cdot F(H(X)))$
proof (rule antisym)
  have ne: $\{P. (P \text{ is } H) \land F P \subseteq P\} \neq \{\}$
  proof
    have $F' T \subseteq T$
      using assms(2) utp-top weak.top-closed by force
  thus $\text{thesis}$
  qed
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 \( \bigcap \{ P. (P is H) \land F(P) \subseteq P \} \subseteq \bigcap \{ P. F(H(P)) \subseteq P \} \)
  proof -
    have 1: \( \land P. F(H(P)) = H(F(H(P))) \)
      by (metis HCond-Idem Healthy-def assms)
  qed
  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

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

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]: \[ P is H; Q is H \] \implies (P ;; Q) is H
begin

lemma upower-Suc-Healthy [closure]:
  assumes \( P is H \)

end
shows $P \cdot \text{Suc} \ n$ is $\mathcal{H}$
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 $\mathcal{H}$ $A \subseteq [\mathcal{H}] \nobreak \ A \neq \{\}$
  shows $P :: (\prod A) = \prod \{P :: Q \mid Q. \ Q \in A\}$
proof -
  have $\{P :: Q \mid Q. \ Q \in A\} \subseteq [\mathcal{H}]$ \nobreak using Healthy-Sequence assms (1) assms (2) by (auto)
thus ?thesis
  by (simp add: healthy-inf-cont seq-Sup-distl setcompr-eq-image assms)
qed

lemma seq-cont-Sup-distr:
  assumes $Q$ is $\mathcal{H}$ $A \subseteq [\mathcal{H}]$ $A \neq \{\}$
  shows $(\prod A) :: Q = \prod \{P :: Q \mid P. \ P \in A\}$
proof -
  have $\{P :: Q \mid P. \ P \in A\} \subseteq [\mathcal{H}]$ \nobreak using Healthy-Sequence assms (1) assms (2) by (auto)
thus ?thesis
  by (simp add: healthy-inf-cont seq-Sup-distr setcompr-eq-image assms)
qed

lemma uplus-healthy [closure]:
  assumes $P$ is $\mathcal{H}$
  shows $P^+$ is $\mathcal{H}$
  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
  utp-unit :: $(\mathcal{T}, \mathcal{\alpha})$ uthy $\Rightarrow$ $\mathcal{\alpha}$ hrrel $(\mathcal{II})$

We can characterise the theory Kleene star by lifting the relational one.

definition utp-star $(\star)$ $[\mathcal{II}]$ $[999]$ $[999]$ where
\[\text{[upred-defs]}: \text{utp-star } \mathcal{T} \ P = (P^* :: II \mathcal{T})\]

We can then characterise tests as refinements of units.

definition utest $(\top, \mathcal{\alpha})$ uthy $\Rightarrow$ $\mathcal{\alpha}$ hrrel $\Rightarrow$ bool where
\[\text{[upred-defs]}: \text{utest } \mathcal{T} \ b = (\mathcal{II} \subseteq b)\]

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 =
  utp-theory-rel +
  assumes Healthy-Left-Unit [closure]: $\mathcal{II}$ is $\mathcal{H}$
  and Left-Unit: $P$ is $\mathcal{H} \Rightarrow (\mathcal{II} :: P) = P$

locale utp-theory-right-unital =
  utp-theory-rel +
assumes Healthy-Right-Unit [closure]: \( \mathcal{I} \) is \( \mathcal{H} \)
and Right-Unit: \( P \) is \( \mathcal{H} \) \( \Rightarrow (P ;; \mathcal{I}) = P \)
locale utp-theory-unital =
  utp-theory-rel +
assumes Healthy-Unit [closure]: \( \mathcal{I} \) is \( \mathcal{H} \)
and Unit-Left: \( P \) is \( \mathcal{H} \) \( \Rightarrow (\mathcal{I} ;; P) = P \)
and Unit-Right: \( P \) is \( \mathcal{H} \) \( \Rightarrow (P ;; \mathcal{I}) = P \)
begin
lemma Unit-self [simp]:
  \( \mathcal{I} ;; \mathcal{I} = \mathcal{I} \)
  by (simp add: Healthy-Unit Unit-Right)
lemma utest-intro:
  \( \mathcal{I} \subseteq P \Rightarrow \text{utest} \ \top \ P \)
  by (simp add: utest-def)
lemma utest-Unit [closure]:
  \( \text{utest} \ \top \mathcal{I} \)
  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 utp-theory-left-unital-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 utp-theory-right-unital-def)
locale utp-theory-mono-unital = utp-theory-mono + utp-theory-unital
begin
lemma utest-Top [closure]:
  \( \text{utest} \ \top \top \)
  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 \( \mathcal{H} \) \( \Rightarrow \top ;; P = \top \)
locale utp-theory-cont-unital-zerol =
  utp-theory-cont-unital + utp-theory-unital-zerol
begin

135
lemma Top-test-Right-Zero:
  assumes b is H utest T b
  shows b ;; T = T
proof –
  have b ⊓ I = I
    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 = I

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 (- \multimap (-,\cdot) \Rightarrow \cdot) where
H1 \multimap (H_1,H_2) \Rightarrow H2 \equiv (H_1,H_2) \Rightarrow \cdot H1, orderA = utp-order H1, orderB = utp-order H2, lower = H_2, upper = H_1

abbreviation mk-conn' (- \multimap (-,\cdot) \Rightarrow (\cdot)) where
T1 \multimap (H_1,H_2) \Rightarrow T2 \equiv (H_1,H_2) \Rightarrow (H_1,H_2)

lemma mk-conn-orderA [simp]: X \multimap (H_1,H_2) \Rightarrow H2 = utp-order H1 by (simp add: mk-conn-def)

lemma mk-conn-orderB [simp]: Y \multimap (H_1,H_2) \Rightarrow H2 = utp-order H2 by (simp add: mk-conn-def)

lemma mk-conn-lower [simp]: \pi \multimap (H_1,H_2) \Rightarrow H2 = H_1 by (simp add: mk-conn-def)

lemma mk-conn-upper [simp]: \pi \multimap (H_1,H_2) \Rightarrow H2 = H_2 by (simp add: mk-conn-def)

gleason comp: (H_2 \multimap (H_3,H_4) \Rightarrow H_3) \circ_g (H_1 \multimap (H_1,H_2) \Rightarrow H_2) = H_1 \multimap (H_1 \circ_g H_3,H_4 \circ_g H_2) \Rightarrow H_3 by (simp add: comp-galcon-def mk-conn-def)

Example Galois connection / retract: Existential quantification

lemma Idempotent-ex: mwb-lens x \Rightarrow Idempotent (ex x) by (simp add: Idempotent-def exists-twice)

lemma Monotonic-ex: mwb-lens x \Rightarrow Monotonic (ex x) by (simp add: mono-def ex-mono)

lemma ex-closed-unrest: vwb-lens x \Rightarrow \{ex x\}_H = \{P. x \notin 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 \circ H = H \circ ex x
shows retract ((ex x \circ H) \multimap (ex x,H) \Rightarrow H)
proof (unfold-locales, simp-all)
show H \in [ex x \circ H]_H \Rightarrow [H]_H
using Healthy-Idempotent assms by blast
from assms(1) assms(3) show ex x \in [H]_H \Rightarrow [ex x \circ H]_H
by (simp add: Pi-iff Healthy-def fun-eq-iff exists-twice)
fix P Q
assume P is (ex x \circ H) Q is H
thus (H P \subseteq Q) = (P \subseteq (\exists 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)
fix $P$
assume $P$ is $(\exists x \cdot H \land P) \subseteq P$
thus by (simp add: Healthy-def)
qed

corollary ex-retract-id:
assumes $vwb-lens x$
shows $\text{retract} (\exists x \leftarrow (\exists x, id) \Rightarrow id)$
using asms ex-retract[where $H=id$] by (auto)
end

20 Relational Hoare calculus

definition $hoare-r :: \alpha \Rightarrow \alpha$ where
\[
\{q\} Q \{r\}_u = (([p]_\leq \Rightarrow [r]_\geq) \subseteq Q)
\]
declare $hoare-r$-def [upred-defs]
named-theorems $hoare$ and $hoare-safe$

method $hoare-split$ uses $hr =$
\[
\text{(intro: hoare intro: hoare-safe hr \ simp add: assigns-r-comp conj-comm conj-assoc usubst unrest)[1]}\]
— Apply Hoare logic laws

method $hoare-auto$ uses $hr = (hoare-split hr; hr; rel-auto?)$

20.2 Basic Laws

lemma $hoare-r$-conj [hoare-safe]: \[
\{q\} Q \{r\}_u \Rightarrow \{p\} Q \{s\}_u \Rightarrow \{p\} Q \{r \land s\}_u
\]
by rel-auto

lemma $hoare-r$-weaken-pre [hoare]:
\[
\{q\} Q \{r\}_u \Rightarrow \{p \land q\} Q \{r\}_u
\]
\[
\{q\} Q \{r\}_u \Rightarrow \{p \land q\} Q \{r\}_u
\]
by rel-auto+

lemma pre-str-hoare-r:
assumes $p_1 \Rightarrow p_2$ and $\{p_2\} C \{q\}_u$
shows $\{p_1\} C \{q\}_u$
using asms by rel-auto

lemma post-weak-hoare-r:
assumes $\{p\} C \{q_2\}_u$ 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’ \( \implies \) \( \{ p_1 \} S \{ q_1 \}_u \)
by rel-auto

## 20.3 Assignment Laws

lemma assigns-hoare-r [hoare-safe]: \( p \Rightarrow \sigma \triangleright q \implies \{ p \} (\sigma)_u \{ q \}_u \)
by rel-auto

lemma assigns-backward-hoare-r:
\( \{ \sigma \triangleright p \} (\sigma)_u \{ p \}_u \)
by rel-auto

lemma assign-floyd-hoare-r:
assumes \( \text{wlv-lens } x \)
shows \( \{ p \} \text{assign-} x e \{ \exists v \cdot p[<v>/x] \land \& x =_u e[<v>/x]\}_u \)
using assms by (rel-auto, metis wlv-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]: \( p \Rightarrow q \implies \{ p \} H \{ q \}_u \)
by rel-auto

## 20.4 Sequence Laws

lemma seq-hoare-r: \( \{ \{ p \} Q_1 \{ s \}_u \; ; \; \{ s \} Q_2 \{ r \}_u \} \implies \{ p \} Q_1 ; \; Q_2 \{ r \}_u \)
by rel-auto

lemma seq-hoare-invariant [hoare-safe]: \( \{ \{ p \} Q_1 \{ p \}_u \; ; \; \{ p \} Q_2 \{ p \}_u \} \implies \{ p \} Q_1 ; \; Q_2 \{ p \}_u \)
by rel-auto

lemma seq-hoare-stronger-pre-1 [hoare-safe]:
\( \{ p \land q \} Q_1 \{ p \land q \}_u \; ; \; \{ p \land q \} Q_2 \{ q \}_u \} \implies \{ p \land q \} Q_1 ; \; Q_2 \{ q \}_u \)
by rel-auto

lemma seq-hoare-stronger-pre-2 [hoare-safe]:
\( \{ p \land q \} Q_1 \{ p \land q \}_u \; ; \; \{ p \land q \} Q_2 \{ p \}_u \} \implies \{ p \land q \} Q_1 ; \; Q_2 \{ p \}_u \)
by rel-auto

lemma seq-hoare-inv-r-2 [hoare]: \( \{ p \} Q_1 \{ q \}_u \; ; \; \{ q \} Q_2 \{ q \}_u \} \implies \{ p \} Q_1 ; \; Q_2 \{ q \}_u \)
by rel-auto

lemma seq-hoare-inv-r-3 [hoare]: \( \{ p \} Q_1 \{ p \}_u \; ; \; \{ p \} Q_2 \{ q \}_u \} \implies \{ p \} Q_1 ; \; Q_2 \{ q \}_u \)
by rel-auto

## 20.5 Conditional Laws

lemma cond-hoare-r [hoare-safe]: \( \{ b \land p \} S \{ q \}_u \; ; \; \{ \neg b \land p \} T \{ q \}_u \} \implies \{ p \} S \triangleleft b \triangleright T \{ q \}_u \)
by rel-auto

lemma cond-hoare-r-up:
assumes \( \{ p \} S \{ q \}_u \) and \( \{ p'' \} T \{ q \}_u \)
shows \( \langle \{ b \land p \} \rangle S \prec b \triangleright_r T \{ q \} \rangle_u \)

using assms by pred-simp

lemma cond-hoare-r-sp:
assumes \( \langle \{ b \land p \} \rangle S \prec \{ \neg b \land p' \} \rangle_u \) and \( \langle \{ \neg b \land p \} \rangle T \{ s \} \rangle_u \)
shows \( \langle \{ p \} \rangle S \prec b \triangleright_r T \{ q \lor s \} \rangle_u \)
using assms by pred-simp

20.6 Recursion Laws

lemma nu-hoare-r-partial:
assumes induct-step:
\( \bigwedge_{st} P. \{ p \} \square_{\{q\} \rangle_u \} \rightarrow \{ p \} F \{ q \} \rangle_u \)
shows \( \{ p \} \nu F \{ q \} \rangle_u \)

unfolding hoare-r-def
proof (rule mu-rec-total-utp-rule[WF M , of - e], goal-cases)
case (1 st)
then show ?case using induct-step[unfolded hoare-r-def, of (\{p\} \{e\} \{<\} \{\neg\} \{st\} \langle u < e < R \Rightarrow [q] \rangle st] by (simp add: alpha)
qed

lemma mu-hoare-r: 
assumes WF: wf R
assumes M: mono F
assumes induct-step:
\( \bigwedge_{st} P. \{ p \land (e, \langle st \rangle) \} u \in_y < R \Rightarrow \{ p \land e =_u \langle st \rangle \} F \{ q \} \rangle_u \)
shows \( \{ p \} \rho F \{ q \} \rangle_u \)

proof (rule rule-tac lfp-lowerbound order-refl)

lemma mu-hoare-r':
assumes WF: wf R
assumes M: mono F
assumes induct-step:
\( \bigwedge_{st} P. \{ p \land (e, \langle st \rangle) \} u \in_y < R \Rightarrow \{ p \land e =_u \langle st \rangle \} F \{ q \} \rangle_u \)
shows \( \{ p \} \rho' F \{ q \} \rangle_u \)

by (meson I0 M WF induct-step mu-hoare-r pre-str-hoare-r)

20.7 Iteration Rules

lemma while-hoare-r [hoare-safe]:
assumes \( \{ p \lor b \} S \{ p \} \rangle_u \)
shows \( \{ p \} \text{ while}\ b\ \text{ do } S \od \{ \neg b \lor p \} \rangle_u \)
using assms by (simp add: while-def hoare-r-def, rule-tac lfp-lowerbound) (rel-auto)

lemma while-invhoare-r [hoare-safe]:
assumes \( \{ p \lor b \} S \{ p \} \rangle_u \) 'pre \Rightarrow p' ' (\neg b \lor p) \Rightarrow post'
shows \( \{ pre \} \text{ while}\ b\ \text{ invr}\ p\ \text{ do } S \od \{ post \} \rangle_u \)
by (metis assms hoare-r-conseq while-hoare-r while-inv-def)

lemma while-r-minimal-partial:
assumes seq-step: 'p \Rightarrow invar'
assumes induct-step: \( \{ invar \land b \} \ C \{ invar \} \rangle_u \)
shows \( \{ p \} \text{ while}\ b\ \text{ do } C\ \text{ od}\ \{ \neg b \land invar \} \rangle_u \)
using induct-step pre-str-hoare-r seq-step while-hoare-r by blast

lemma approx-chain:
(\bigcap n::nat. [p \land v <_u <n>_u]_u) = [p]_u
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 \( \bigwedge z::nat. [p \land b \land v =_u <z>_u] S [p \land v <_u <z>_u]_u \)
shows \( [p] \text{while}_b S \text{od}_u [\neg b \land p]_u \)
proof
have \([p]_u \Rightarrow [\neg b \land p]_u \subseteq (\mu X \cdot S ;; X \triangleleft b \triangleright_r II)\)
proof (rule mu-refine-intro)
from assms show \([p]_u \Rightarrow [\neg b \land p]_u \subseteq (\mu X \cdot S ;; X \triangleleft b \triangleright_r II)\)
by (rel-auto)

let \(?E = \lambda n. [p \land v <_u <n>_u]_u\)
show \([p]_u \land (\mu X \cdot S ;; X \triangleleft b \triangleright_r II) = ([p]_u \land (\nu X \cdot S ;; X \triangleleft b \triangleright_r II))\)
proof (rule constr-fp-uniq[where \( E=?E \)])
show \((\bigcap n. ?E(n)) = [p]_u\)
by (rel-auto)

show mono \((\lambda X. S ;; X \triangleleft b \triangleright_r II)\)
by (simp add: cond-mono monoI seqr-mono)

show constr \((\lambda X. S ;; X \triangleleft b \triangleright_r II) ?E\)
proof (rule constrI)

show chain ?E
proof (rule chainI)
  show \([p \land v <_u <0>_u]_u = \text{false}\)
  by (rel-auto)
  show \(\bigwedge i. [p \land v <_u <\text{Suc } i>_u]_u \subseteq [p \land v <_u <i>_u]_u\)
  by (rel-auto)
qed

from assms
show \(\bigwedge X n. (S ;; X \triangleleft b \triangleright_r II \land [p \land v <_u <n + 1>_u]_u) =\)
  \((S ;; (X \land [p \land v <_u <n>_u]_u) \triangleleft b \triangleright_r II \land [p \land v <_u <n + 1>_u]_u)\)
apply (rel-auto)
using less-antisym less-trans apply blast
done
qed

thus \(?thesis\)
by (simp add: hoare-r-def while-bot-def)
qed

lemma while-vrt-hoare-r [hoare-safe]:
assumes \(\bigwedge z::nat. [p \land b \land v =_u <z>_u] S [p \land v <_u <z>_u]_u \text{pre } \Rightarrow \text{post}\)
proof (bl
shows $\{pre\}$ while $b$ inv $p$ vrt $v$ do $S$ od $\{post\}_u$
apply (rule hoare-r-conseq[$OF$ $assms(2)$ - $assms(3)$])
apply (simp add: while-vrt-def)
apply (rule while-term-hoare-r[where $v=v$, $OF$ $assms(1)$])
done

General total correctness law based on well-founded induction

lemma while-wf-hoare-r:
assumes WF: $wf$ $R$
assumes I0: ‘$pre$ $⇒$ $p$’
assumes indact-step: $\forall$ st. $\{b \land p \land e =_u <st>_u\} Q\{p \land (e, <st>_u) \in_u <R>_u\}$
assumes PHI: ‘$(\neg b \land p) \Rightarrow post’
shows $\{pre\}$ while $_b$ $b$ inv $p$ do $Q$ od $\{post\}_u$

proof
unfolding hoare-r-def while-inv-bot-def while-bot-def
from I0 show ‘$pre$ $⇒$ $[p]_<$’
  by rel-auto
  show $([p]_< [post]_> \subseteq (\mu X \cdot Q :: X < b \triangleright_r II)$
  proof (rule mu-rec-total-utp-rule[where $e=e$, $OF$ WF])
    show Monotonic $(\lambda X. Q :: X < b \triangleright_r II)$
    by (simp add: closure)
    have indact-step: $\forall$ st. $(\{b \land p \land e =_u <st>_u\} <_u \Rightarrow ([p \land (e, <st>_u) \in_u <R>_u]_>) \subseteq Q$
      using indact-step by rel-auto
    with PHI
    show $\forall$ st. $(\{p\}_<_u \land [e]_<_u =_u <st>_u \Rightarrow [post]_> \subseteq (\{p\}_<_u \land ([e]_<_u, <st>_u) \in_u <R>_u \Rightarrow [post]_>)$ $< b \triangleright_r II$
    by (rel-auto)
qed

20.8 Frame Rules

Frame rule: If starting $S$ in a state satisfying $p$establishes$eq$ in the final state, then we can insert an invariant predicate $r$ when $S$ is framed by $a$, provided that $r$ does not refer to variables in the frame, and $q$ does not refer to variables outside the frame.

lemma frame-hoare-r:
assumes vwb-lens $a$ $a$ $\triangleright_r a \triangleright_q q$ $\{p\}_u \sqcup q$
shows $\{p \land r\}_a:\{p\}_u \sqcup q \land r\}_u$
using $assms$
by (rel-auto, metis)

lemma frame-strong-hoare-r [hoare-safe]:
assumes vwb-lens $a$ $a$ $\triangleright_r a \triangleright_q q$ $\{p \land r\}_u \sqcup q$
shows $\{p \land r\}_a:\{S\}_a \sqcup q \land r\}_u$
using $assms$ by (rel-auto, metis)

lemma frame-hoare-r' [hoare-safe]:
assumes vwb-lens $a$ $a$ $\triangleright_r a \triangleright_q q$ $\{r \land p\}_u \sqcup q$
shows $\{r \land p\}_a:\{S\}_a \sqcup q \land r\}_u$
using $assms$ by (simp add: frame-strong-hoare-r utp-pred-laws.inf.commute)

lemma antiframe-hoare-r:
assumes vwb-lens $a$ $a$ $\triangleright_r a \triangleright_q q$ $\{p\}_u \sqcup q$
shows $\{p \land r\}_a:\{p\}_u \sqcup q \land r\}_u$

142
using assms by (rel-auto, metis)

lemma antiframe-strong-hoare-r:
  assumes vwblens 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 vwblens g vwblens g' vwblens l l ⊲ g 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
imports 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

143
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]:**
\[ (P \triangleleft b \trianglerighteq r) \text{ wp } r = ((b \implies P \text{ wp } r) \land ((\neg b) \implies Q \text{ wp } r)) \]
by rel-auto

**Lemma wp-USUP-pre [wp]:**
\[ \bigvee_{i \in \{0..n\}} \cdot Q(i) = \bigvee_{i \in \{0..n\}} \cdot P \text{ wp } Q(i) \]
by (rel-auto)

**Theorem wp-hoare-link:**
\[ \{ \lfloor p \rfloor \} Q \{ r \}_u \longleftrightarrow (Q \text{ wp } r \subseteq p) \]
by rel-auto

If two programs have the same weakest precondition for any postcondition then the programs are the same.

**Theorem wp-eq-intro:**
\[ \lceil \bigwedge_i \cdot P \text{ wp } r = Q \text{ wp } r \rceil \implies P = Q \]
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**

\[ \text{usp :: } 'a \Rightarrow 'b \Rightarrow 'c \text{ (infix sp 60)} \]

**Definition sp-upred :: 'α cond ⇒ ('α, 'β) urel ⇒ 'β cond where**
\[ sp-upred p Q = \lfloor \lfloor p \rfloor \rfloor ;; Q :: ('α, 'β) urel \rfloor \]

**Adhoc Overloading**

\[ \text{usp sp-upred} \]

**Declare sp-upred-def [upred-defs]**

**Lemma sp-false [sp]:**
\[ p \text{ sp } \text{false} = \text{false} \]
by (rel-simp)

**Lemma sp-true [sp]:**
\[ q \neq \text{false} \implies q \text{ sp } \text{true} = \text{true} \]
by (rel-auto)
lemma sp-assigns-r [sp]:
\( \forall x. (p \ sp x := e) = (\exists v \cdot p[x/v] \land x = e[x/v]) \)
by (rel-auto, metis wb-lens-wb wb-lens.get PUT, metis wb-lens.put-eq)

lemma sp-it-is-post-condition:
\( \{p\} \{ p \ sp C \} \)
by rel-blast

lemma sp-it-is-the-strongest-post:
\( p \ sp C \Rightarrow Q' \Rightarrow \{p\} \{Q\} \)
by rel-blast

lemma sp-so:
\( p \ sp C \Rightarrow Q' = \{p\} \{Q\} \)
by rel-blast

theorem sp-hoare-link:
\( \{p\} Q \{r\} \iff (r \sqsubseteq p \ sp Q) \)
by rel-auto

lemma sp-while-r [sp]:
\[ \begin{align*}
\text{assumes} & \quad \langle \text{pre} \Rightarrow I' \rangle \text{ and } \langle \{I \land b\} C \{I'\} \rangle \text{ and } \langle I' \Rightarrow I \rangle \\
\text{shows} & \quad (\text{pre sp invar I while } b \text{ do C od}) = (\neg b \land I) \\
\text{unfolding} & \quad \text{sp-upred-def} \\
\text{oops}
\end{align*} \]

theorem sp-eq-intro: \( \forall r. \ sp P = \ sp Q \Rightarrow P = Q \)
by (rel-auto robust, fastforce+)

lemma wp-sp-sym:
\( \text{prog wp (true sp prog)} \)
by rel-auto

lemma it-is-pre-condition:
\( \{C wp Q\} \{Q\} \)
by rel-blast

lemma it-is-the-weakest-pre:
\( P \Rightarrow C wp Q' = \{P\} \{Q\} \)
by rel-blast

lemma s-pre:
\( P \Rightarrow C wp Q' = \{P\} \{Q\} \)
by rel-blast

end

23 Concurrent Programming
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) \ mrg = \\
& \quad \text{mrg-prior :: } \alpha \\
& \quad \text{mrg-left :: } \beta_0 \\
& \quad \text{mrg-right :: } \beta_1
\end{align*}

definition \text{pre-uvar :: } (\alpha \Rightarrow (\alpha, \beta_0, \beta_1) \ mrg) \ where
\begin{align*}
\text{[upred-defs]:} & \quad \text{pre-uvar x = x ;L mrg-prior}
\end{align*}

definition \text{left-uvar :: } (\alpha \Rightarrow \beta_0) \Rightarrow ((\alpha, \beta_0, \beta_1) \ mrg) \ where
\begin{align*}
\text{[upred-defs]:} & \quad \text{left-uvar x = x ;L mrg-left}
\end{align*}

definition \text{right-uvar :: } (\alpha \Rightarrow \beta_1) \Rightarrow ((\alpha, \beta_0, \beta_1) \ mrg) \ where
\begin{align*}
\text{[upred-defs]:} & \quad \text{right-uvar x = x ;L mrg-right}
\end{align*}

We set up syntax for the three variable classes using a subscript $<, 0-x,$ and $1-x$, respectively.

\begin{align*}
\text{syntax} & \quad -svarpre :: \ svid \Rightarrow \ svid (\ < \ [995] 995) \\
& \quad -svarleft :: \ svid \Rightarrow \ svid (0- \ [995] 995) \\
& \quad -svarright :: \ svid \Rightarrow \ svid (1- \ [995] 995)
\end{align*}

\begin{align*}
\text{translations} & \quad -svarpre x == \ CONST \ pre-uvar x \\
& \quad -svarleft x == \ CONST \ left-uvar x \\
& \quad -svarright x == \ CONST \ right-uvar x \\
& \quad -svarpre \ Sigma <= \ CONST \ pre-uvar \ I_L \\
& \quad -svarleft \ Sigma <= \ CONST \ left-uvar \ I_L \\
& \quad -svarright \ Sigma <= \ CONST \ right-uvar \ I_L
\end{align*}

We proved behavedness closure properties about the lenses.

\begin{align*}
\text{lemma \ left-uvar [simp]:} & \quad \text{vwb-lens x \Rightarrow vwb-lens (left-uvar x)} \\
& \quad \text{by (simp add: left-uvar-def)}
\end{align*}

\begin{align*}
\text{lemma \ right-uvar [simp]:} & \quad \text{vwb-lens x \Rightarrow vwb-lens (right-uvar x)} \\
& \quad \text{by (simp add: right-uvar-def)}
\end{align*}

\begin{align*}
\text{lemma \ pre-uvar [simp]:} & \quad \text{vwb-lens x \Rightarrow vwb-lens (pre-uvar x)} \\
& \quad \text{by (simp add: pre-uvar-def)}
\end{align*}
lemma left-uvar-mwb [simp]: \( \text{mwb-lens } x \implies \text{mwb-lens } (\text{left-uvar } x) \)
by (simp add: left-uvar-def)

lemma right-uvar-mwb [simp]: \( \text{mwb-lens } x \implies \text{mwb-lens } (\text{right-uvar } x) \)
by (simp add: right-uvar-def)

lemma pre-uvar-mwb [simp]: \( \text{mwb-lens } x \implies \text{mwb-lens } (\text{pre-uvar } x) \)
by (simp add: pre-uvar-def)

We prove various independence laws about the variable classes.

lemma left-uvar-indep-right-uvar [simp]:
left-uvar \( x \bowtie \bowtie \) 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 \( x \bowtie \bowtie \) pre-uvar \( y \)
by (simp add: left-uvar-def pre-uvar-def)

lemma left-uvar-indep-left-uvar [simp]:
x \( \bowtie \bowtie \) \implies \text{left-uvar } x \bowtie \bowtie \text{left-uvar } y
by (simp add: left-uvar-def)

lemma right-uvar-indep-left-uvar [simp]:
right-uvar \( x \bowtie \bowtie \) left-uvar \( y \)
by (simp add: lens-indep-sym)

lemma right-uvar-indep-pre-uvar [simp]:
right-uvar \( x \bowtie \bowtie \) pre-uvar \( y \)
by (simp add: right-uvar-def pre-uvar-def)

lemma right-uvar-indep-right-uvar [simp]:
x \( \bowtie \bowtie \) \implies \text{right-uvar } x \bowtie \bowtie \text{right-uvar } y
by (simp add: right-uvar-def)

lemma pre-uvar-indep-left-uvar [simp]:
pre-uvar \( x \bowtie \bowtie \) left-uvar \( y \)
by (simp add: lens-indep-sym)

lemma pre-uvar-indep-right-uvar [simp]:
pre-uvar \( x \bowtie \bowtie \) right-uvar \( y \)
by (simp add: lens-indep-sym)

lemma pre-uvar-indep-pre-uvar [simp]:
x \( \bowtie \bowtie \) \implies \text{pre-uvar } x \bowtie \bowtie \text{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 \( 'a \text{ merge } = ('a, 'a, 'a) \text{ mrg, } 'a \text{ urel} \)

skip is the merge predicate which ignores the output of both parallel predicates

definition skip_\( m :: 'a \text{ merge } \text{ where} \)
swap is a predicate that swaps the left and right indices; it is used to specify commutativity of the parallel operator.

**Definition** swap \( m \) :: \(((\alpha, \beta, \beta) \mathrm{mrg}) \mathrm{hrel}\) where

\[
\text{upred-defs: swap}_m = (0\!-\!v, 1\!-\!v) := (\&1\!-\!v, \&0\!-\!v)
\]

A symmetric merge is one for which swapping the order of the merged concurrent predicates has no effect. We represent this by the following healthiness condition that states that \( swap_m \) is a left-unit.

**Abbreviation** SymMerge :: \( \alpha \) merge \( \Rightarrow \) \( \alpha \) merge where

\[
\text{SymMerge}(M) \equiv (\text{swap}_m ;; M)
\]

### 23.3 Separating Simulations

U0 and U1 are relations that modify the variables of the input state-space such that they become indexed with 0 and 1, respectively.

**Definition** U0 :: \(((\beta_0, (\alpha, \beta_0, \beta_1) \mathrm{mrg}) \mathrm{urel})\) where

\[
\text{upred-defs: U0} = (0\!-\!v = u \& v)
\]

**Definition** U1 :: \(((\beta_1, (\alpha, \beta_0, \beta_1) \mathrm{mrg}) \mathrm{urel})\) where

\[
\text{upred-defs: U1} = (1\!-\!v = u \& v)
\]

**Lemma** U0-swap: \( U0 ;; \text{swap}_m \) = U1

by (rel-auto)

**Lemma** U1-swap: \( U1 ;; \text{swap}_m \) = 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]: \( U0\alpha = (1L \times L \mathrm{mrg-left}) \)

**Definition** U1\(\alpha\) where [upred-defs]: \( U1\alpha = (1L \times L \mathrm{mrg-right}) \)

We then create the following intuitive syntax for separating simulations.

**Abbreviation** U0-alpha-lift \((\lceil-\rceil_0)\) where \( [P]_0 \equiv P \oplus_p U0\alpha \)

**Abbreviation** U1-alpha-lift \((\lceil-\rceil_1)\) 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** U0\(\alpha\)-vwb-lens [simp]: vwb-lens U0\(\alpha\)

by (simp add: U0\(\alpha\)-def id-vwb-lens prod-vwb-lens)

**Lemma** U1\(\alpha\)-vwb-lens [simp]: vwb-lens U1\(\alpha\)

by (simp add: U1\(\alpha\)-def id-vwb-lens prod-vwb-lens)
lemma $U_0 \alpha$-indep-right-uvar [simp]: \( \text{vwb-lens } x \implies U_0 \alpha \propto \text{out-var } (\text{right-uvar } x) \)
by (force intro: plus-pres-lens-indep \( \text{fst-snd-lens-indep left-uvar-comp} \))
simp add: $U_0 \alpha$-def \( \text{right-uvar-def out-var-def prod-as-plus lens-comp-assoc} \)
THEN sym)

lemma $U_1 \alpha$-indep-left-uvar [simp]: \( \text{vwb-lens } x \implies U_1 \alpha \propto \text{out-var } (\text{left-uvar } x) \)
by (force intro: plus-pres-lens-indep \( \text{fst-snd-lens-indep left-uvar-comp} \))
simp add: $U_1 \alpha$-def \( \text{left-uvar-def out-var-def prod-as-plus lens-comp-assoc} \)
THEN sym)

lemma $U_0$-alpha-lift-bool-subst [usubst]:
\[
\sigma_0 (\sigma_0 \beta x ' \mapsto s \text{true}) \cap P_0 = \sigma_0 (\sigma_0 \beta x ' \mapsto s \text{false}) \cap P_0
\]
by (pred-auto+)

lemma $U_1$-alpha-lift-bool-subst [usubst]:
\[
\sigma_1 (\sigma_1 \beta x ' \mapsto s \text{true}) \cap P_1 = \sigma_1 (\sigma_1 \beta x ' \mapsto s \text{false}) \cap P_1
\]
by (pred-auto+)

lemma $U_0$-alpha-out-var [alpha]: $x_0 = $0−x' by (rel-auto)

lemma $U_1$-alpha-out-var [alpha]: $x_1 = $1−x' by (rel-auto)

lemma $U_0$-skip [alpha]: $P_0 = ($0−$v'$ =u $v$)
by (rel-auto)

lemma $U_1$-skip [alpha]: $P_1 = ($1−$v'$ =u $v$)
by (rel-auto)

lemma $U_0$-seqr [alpha]: $P_0 = P_0 ; Q_0$ by (rel-auto)

lemma $U_1$-seqr [alpha]: $P_1 = P_1 ; Q_1$ by (rel-auto)

lemma $U_0$-comp-in-var [alpha]: (in-var x) $L_0 U_0 \alpha = \text{in-var } x$
by (simp add: $U_0 \alpha$-def \( \text{alpha-in-var in-var-prod-lens pre-uvar-def} \))

lemma $U_0$-comp-out-var [alpha]: (out-var x) $L_0 U_0 \alpha = \text{out-var } (\text{left-uvar } x)$
by (simp add: $U_0 \alpha$-def \( \text{alpha-out-var id-wb-lens left-uvar-def out-var-prod-lens} \))

lemma $U_1$-comp-in-var [alpha]: (in-var x) $L_1 U_1 \alpha = \text{in-var } x$
by (simp add: $U_1 \alpha$-def \( \text{alpha-in-var in-var-prod-lens pre-uvar-def} \))

lemma $U_1$-comp-out-var [alpha]: (out-var x) $L_1 U_1 \alpha = \text{out-var } (\text{right-uvar } x)$
by (simp add: $U_1 \alpha$-def \( \text{alpha-out-var id-wb-lens right-uvar-def out-var-prod-lens} \))

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 ⇒ (('a, 'a, 'a) mrg) mrg, 'a) urel (M3(·)) where
[upred-defs]: ThreeWayMerge M = ((0−v’ =u $0−v ∧ $1−v’ =u $1−0−v ∧ $v’ =u $v<) :: M ::
U0 ∧ $1−v’ =u $1−1−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,1−0−v,1−1−v) := (⊥−0−v,⊥−1−v,⊥−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 :: M3(M) = M3(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 || 85) where
P || Q ≡ (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, 'b) urel ⇒ (('a, 'b, 'c) mrg, 'b) urel ⇒ ('a, 'c) urel ⇒ ('a, 'b) urel
(·, ·) − [85,0,86] 85
where [upred-defs]: P || M Q = (P || Q :: M)

lemma par-by-merge-alt-def: P || M Q = ([P]1 ∧ [Q]0 ∧ $v’ =u $v) :: M
by (simp add: par-by-merge-def U0-as-alpha U1-as-alpha)

lemma shEx-pbm-left: ((∃ x · P x) || M Q) = (∃ x · (P x || M Q))
by (rel-auto)

lemma shEx-pbm-right: (P || M (∃ x · Q x)) = (∃ x · (P || M Q x))
by (rel-auto)

23.6 Unrestriction Laws

lemma unrest-in-par-by-merge [unrest]:
[ $x ∉ P; $x< ∉ M; $x ∉ Q ] −→ $x ∉ P || M Q
by (rel-auto, fastforce+)

lemma unrest-out-par-by-merge [unrest]:
[ $x’ ∉ M ] −→ $x’ ∉ P || M Q
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 \texttt{U0-seq-subst}: \((P :: U0)[\texttt{v} \mapsto \texttt{x}'] = (P[\texttt{v} \mapsto \texttt{x}'] :: U0)

by (rel-auto)

lemma \texttt{U1-seq-subst}: \((P :: U1)[\texttt{v} \mapsto \texttt{x}'] = (P[\texttt{v} \mapsto \texttt{x}'] :: U1)

by (rel-auto)

lemma \texttt{lit-pbm-subst} [usubst):
\begin{align*}
\text{fixes } x :: (\rightarrow \Rightarrow \alpha) \\
\text{shows} \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x \mapsto, \texttt{false}) \uplus (P \parallel M \ Q) = \sigma \uplus (\sigma[\texttt{false} \mapsto \texttt{x}]) \parallel M[\texttt{false} \mapsto \texttt{x}] (Q[\texttt{false} \mapsto \texttt{x}]) \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x \mapsto, \texttt{true}) \uplus (P \parallel M \ Q) = \sigma \uplus (\sigma[\texttt{true} \mapsto \texttt{x}]) \parallel M[\texttt{true} \mapsto \texttt{x}] (Q[\texttt{true} \mapsto \texttt{x}]) \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x' \mapsto, \texttt{false}) \uplus (P \parallel M \ Q) = \sigma \uplus (P \parallel M[\texttt{false} \mapsto \texttt{x}']) (Q[\texttt{false} \mapsto \texttt{x}']) \\
\end{align*}

by (rel-auto)+

lemma \texttt{bool-pbm-subst} [usubst]:
\begin{align*}
\text{fixes } x :: (\rightarrow \Rightarrow \alpha) \\
\text{shows} \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x \mapsto, \texttt{false}) \uplus (P \parallel M \ Q) = \sigma \uplus (\sigma[\texttt{false} \mapsto \texttt{x}]) \parallel M[\texttt{false} \mapsto \texttt{x}] (Q[\texttt{false} \mapsto \texttt{x}]) \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x \mapsto, \texttt{true}) \uplus (P \parallel M \ Q) = \sigma \uplus (\sigma[\texttt{true} \mapsto \texttt{x}]) \parallel M[\texttt{true} \mapsto \texttt{x}] (Q[\texttt{true} \mapsto \texttt{x}]) \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x' \mapsto, \texttt{false}) \uplus (P \parallel M \ Q) = \sigma \uplus (P \parallel M[\texttt{false} \mapsto \texttt{x}']) (Q[\texttt{false} \mapsto \texttt{x}']) \\
\end{align*}

by (rel-auto)+

lemma \texttt{zero-one-pbm-subst} [usubst]:
\begin{align*}
\text{fixes } x :: (\rightarrow \Rightarrow \alpha) \\
\text{shows} \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x \mapsto, 0) \uplus (P \parallel M \ Q) = \sigma \uplus (\sigma[0 \mapsto \texttt{x}]) \parallel M[0 \mapsto \texttt{x}] (Q[0 \mapsto \texttt{x}]) \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x \mapsto, 1) \uplus (P \parallel M \ Q) = \sigma \uplus (\sigma[1 \mapsto \texttt{x}]) \parallel M[1 \mapsto \texttt{x}] (Q[1 \mapsto \texttt{x}]) \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x' \mapsto, 0) \uplus (P \parallel M \ Q) = \sigma \uplus (P \parallel M[0 \mapsto \texttt{x}']) (Q[0 \mapsto \texttt{x}']) \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x' \mapsto, 1) \uplus (P \parallel M \ Q) = \sigma \uplus (P \parallel M[1 \mapsto \texttt{x}']) (Q[1 \mapsto \texttt{x}']) \\
\end{align*}

by (rel-auto)+

lemma \texttt{numeral-pbm-subst} [usubst]:
\begin{align*}
\text{fixes } x :: (\rightarrow \Rightarrow \alpha) \\
\text{shows} \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x \mapsto, \texttt{numeral} \ n) \uplus (P \parallel M \ Q) = \sigma \uplus (\sigma[\texttt{numeral} \ n \mapsto \texttt{x}]) \parallel M[\texttt{numeral} \ n \mapsto \texttt{x}] (Q[\texttt{numeral} \ n \mapsto \texttt{x}]) \\
\quad \land \ P \ Q \ M \ \sigma. \ \sigma(\delta x' \mapsto, \texttt{numeral} \ n) \uplus (P \parallel M \ Q) = \sigma \uplus (P \parallel M[\texttt{numeral} \ n \mapsto \texttt{x}']) (Q[\texttt{numeral} \ n \mapsto \texttt{x}']) \\
\end{align*}

by (rel-auto)+

23.8 Parallel-by-merge laws

lemma \texttt{par-by-merge-false} [simp]:
\begin{align*}
P \parallel \texttt{false} \ Q = \texttt{false} \\
\text{by (rel-auto)}
\end{align*}

lemma \texttt{par-by-merge-left-false} [simp]:
\begin{align*}
\texttt{false} \parallel M \ Q = \texttt{false} \\
\text{by (rel-auto)}
\end{align*}

lemma \texttt{par-by-merge-right-false} [simp]:
\begin{align*}
P \parallel M \texttt{false} = \texttt{false} \\
\text{by (rel-auto)}
\end{align*}

lemma \texttt{par-by-merge-seq-add}: \((P \parallel M \ Q) ; \ R = (P \parallel M ; \ R \ Q)\)
A skip parallel-by-merge yields a skip whenever the parallel predicates are both feasible.

**Lemma** `par-by-merge-skip`:
- **Assumes**: \( P \text{ ; ; } \text{true} = \text{true} Q \text{ ; ; } \text{true} = \text{true} \)
- **Shows**: \( P \|_{\text{skip}} Q = \Pi \)
- **Using**: `assms` by `(rel-auto)

**Lemma** `skip-merge-swap`:
- **Assumes**: \( \text{swap}_m \text{ ; ; } \text{skip}_m = \text{skip}_m \)
- **By**: `(rel-auto)

**Lemma** `par-sep-swap`:
- **Assumes**: \( P \|_s Q \text{ ; ; } \text{swap}_m = Q \|_s P \)
- **By**: `(rel-auto)

Parallel-by-merge commutes when the merge predicate is unchanged by swap

**Lemma** `par-by-merge-commute-swap`:
- **Shows**: \( P \|_M Q = Q \|_{\text{swap}_m} M P \)
- **Proof** –
  - **Have**: \( Q \|_{\text{swap}_m} M P = (((Q \| U0) \land (P \| U1) \land v' = u v) ; ; \text{swap}_m) ; ; M) \)
  - **By**: `(simp add: par-by-merge-def seqr-assoc)
  - **Also have**: \( ... = (((Q \| U0) ; ; \text{swap}_m) \land (P \| U1 ; ; \text{swap}_m) \land v' = u v) ; ; M) \)
  - **By**: `(rel-auto)
  - **Also have**: \( ... = (((Q \| U1) \land (P \| U0) \land v' = u v) ; ; M) \)
    - **By**: `(simp add: U0-swap U1-swap)
  - **Also have**: \( ... = P \|_M Q \)
    - **By**: `(simp add: par-by-merge-def utp-pred-laws.inf.left-commute)
      - **Finally show**: thesis ..

**Theorem** `par-by-merge-commute`:
- **Assumes**: \( M \text{ is SymMerge} \)
- **Shows**: \( P \|_M Q = Q \|_M P \)
- **By**: `(metis Healthy-if assms par-by-merge-commute-swap)

**Lemma** `par-by-merge-mono-1`:
- **Assumes**: \( P_1 \sqsubseteq P_2 \)
- **Shows**: \( P_1 \|_M Q \sqsubseteq P_2 \|_M Q \)
- **Using**: `assms` by `(rel-auto)

**Lemma** `par-by-merge-mono-2`:
- **Assumes**: \( Q_1 \sqsubseteq Q_2 \)
- **Shows**: \( (P \|_M Q_1) \sqsubseteq (P \|_M Q_2) \)
- **Using**: `assms` by `(rel-blast)

**Lemma** `par-by-merge-mono`:
- **Assumes**: \( P_1 \sqsubseteq P_2 \quad Q_1 \sqsubseteq Q_2 \)
- **Shows**: \( P_1 \|_M Q_1 \sqsubseteq P_2 \|_M Q_2 \)
- **By**: `(meson assms dual-order.trans par-by-merge-mono-1 par-by-merge-mono-2)

**Theorem** `par-by-merge-assoc`:
- **Assumes**: \( M \text{ is SymMerge AssocMerge M} \)
- **Shows**: \( (P \|_M Q) \|_M R = P \|_M (Q \|_M R) \)
- **Proof** –
  - **Have**: \( (P \|_M Q) \|_M R = ((P \| U0) \land (Q \| U0) \land U1) \land (R \| U1) \land v' = u v) ; ; M \exists(M) \)
  - **By**: `(rel-blast)
also have \( \vdash \langle P \sqcap Q \rangle _M R = \langle P \rangle _M R \cap \langle Q \rangle _M R \) by (rel-auto)

theorem par-by-merge-choice-right:
\( \langle P \rangle _M (Q \cap R) = \langle P \cap Q \rangle _M R \cap \langle P \rangle _M R \)
by (rel-auto)

theorem par-by-merge-or-left:
\( \langle P \rangle _M (Q \cap R) = \langle P \rangle _M Q \cap \langle P \rangle _M R \)
by (rel-auto)

theorem par-by-merge-or-right:
\( \langle P \rangle _M (Q \cap R) = \langle P \cap Q \rangle _M R \cap \langle P \rangle _M R \)
by (rel-auto)

theorem par-by-merge-USUP-mem-left:
\( \langle \bigwedge i \cdot P(i) \rangle _M Q = \langle \bigwedge i \cdot P(i) \rangle _M Q \)
by (rel-auto)

theorem par-by-merge-USUP-ind-left:
\( \langle \bigwedge i \cdot P(i) \rangle _M Q = \langle \bigwedge i \cdot P(i) \rangle _M Q \)
by (rel-auto)

theorem par-by-merge-USUP-ind-right:
\( \langle \bigwedge i \cdot P(i) \rangle _M Q = \langle \bigwedge i \cdot P(i) \rangle _M Q \)
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 :: \( \langle \alpha \implies \alpha \rangle \implies \langle \beta \implies \alpha \rangle \implies \alpha \) merge \( (M[-1]_\alpha) \) where
[upred-defs]: \( M[a]_\alpha = (\$v' = u \cdot (\$v_0 + 0 = \mathbf{on} \& \mathbf{a}) \oplus \$t - \mathbf{v} \mathbf{on} \& \mathbf{a}) \)

lemma swap-StateMerge: \( a \parallel b \implies (\text{swap}_m :: M[a]_\beta) = M[b]_\alpha \)
by (rel-auto, simp-all add: lens-indep-comm)

abbreviation StateParallel :: \( \alpha \ hrel \implies \langle \alpha \implies \alpha \rangle \implies \langle \beta \implies \alpha \rangle \implies \alpha \ hrel \implies \alpha \ hrel \) \( (\cdot | - \cdot) \cdot \cdot [85, 0, 0.86] \)
where \( P \parallel \alpha \parallel Q \equiv P \parallel M[a]_\beta \parallel Q \)
lemma StateParallel-commute: \( a \parallel b \implies P \mid a \mid b \parallel Q = Q \mid b \mid a \parallel P \)
by (metis par-by-merge-commute-swap swap-StateMerge)

lemma StateParallel-form:
\[ P \mid a \mid b = Q = \exists (st_0, st_1) \cdot P[\langle st_0 \rangle/\$v\rangle] \land Q[\langle st_1 \rangle/\$v\rangle] \land \$v' =_a (\$v \oplus \langle st_0 \rangle \text{ on } \& \& b) \]
by (rel-auto)

lemma StateParallel-form':
assumes vwb-lens a vwb-lens b a \parallel b
shows \( P \mid a \mid b \parallel Q = \{ \& a, \& b \}: \{ (P \mid v \{ \$v, \$a' \}) \land (Q \mid v \{\$v, \$b'\}) \} \)
using assms
apply (simp add: StateParallel-form, rel-auto)
apply (metis vwb-lens-wb wb-lens-get-put)
apply (simp add: lens-indep-comm)
apply (metis (no-types, hide-lams) lens-indep-comm vwb-lens-wb wb-lens-def weak-lens.put-get)
done

We can frame all the variables that the parallel operator refers to

lemma StateParallel-frame:
assumes vwb-lens a vwb-lens b a \parallel b
shows \( \{ \& a, \& b \}: (P \mid a \mid b \parallel Q) = P \mid a \mid b \parallel Q \)
using assms
apply (simp add: StateParallel-form, rel-auto)
apply (metis vwb-lens-wb wb-lens-get-put)
apply (simp add: lens-indep-comm)
done

Parallel Hoare logic rule. This employs something similar to separating conjunction in the postcondition, but we explicitly require that the two conjuncts only refer to variables on the left and right of the parallel composition explicitly.

theorem StateParallel-hoare \{hoare\}:
assumes \{ \langle c \rangle \parallel d_1 \parallel d_2 \parallel a \parallel b \parallel d_1 \parallel d_2 \parallel \}
shows \{ \langle c \rangle \parallel P \mid a \mid b \parallel Q \mid d_1 \parallel d_2 \parallel \}
proof
— Parallelise the specification
from assms(4,5)
have 1:\( \{ \langle c \rangle \parallel [d_1 \parallel d_2 \parallel \} \subseteq \{ \langle c \rangle \parallel [d_1 \parallel d_2 \parallel \} \parallel (\text{is } \ ?lhs \subseteq ?rhs) \)
by (simp add: StateParallel-form, rel-auto, metis assms(3) lens-indep-comm)
— Prove Hoare rule by monotonicity of parallelism
have 2:\( ?rhs \subseteq P \mid a \mid b \parallel Q \)
proof (rule par-by-merge-mono)
show \( \{ \langle c \rangle \parallel [d_1 \parallel d_2 \parallel \} \subseteq P \)
using assms(1) hoare-r-def by auto
show \( \{ \langle c \rangle \parallel [d_2 \parallel d_2 \parallel \} \subseteq Q \)
using assms(2) hoare-r-def by auto
qed
show \( ?thesis \)
unfolding hoare-r-def using 1 2 order-trans by auto
qed

Specialised version of the above law where an invariant expression referring to variables outside the frame is preserved.

theorem StateParallel-frame-hoare \{hoare\}:
assumes \( vwb\text{-}lens\ a\ vwb\text{-}lens\ b\ a\conc\ b\ a\conc\ d_1\ b\conc\ d_2\ a\conc\ c_1\ b\conc\ c_1\ \{a\conc\ c_2\}P\{d_1\}u\ \{c_1\conc\ c_2\}Q\{d_2\}u\)

shows \( \{c_1\conc\ c_2\}P\{a\conc\ b\conc\ Q\}\{c_1\conc\ d_1\conc\ d_2\}u\)

proof

- have \( \{c_1\conc\ c_2\}\{\&\conc\ a\conc\ \&\conc\ b\conc\}:[P\{a\conc\ b\conc\ Q\}\{c_1\conc\ d_1\conc\ d_2\}u\)

  by (auto intro!: frame-hoare-r frame-parallel-hoare simp add: assms unrest plus-vwb-lens)

thus ?thesis

by (simp add: frame-parallel-frame assms)

qed

end

24 Relational Operational Semantics

theory utp-rel-opsem

imports
  utp-rel-laws
  utp-hoare

begin

This theory uses the laws of relational calculus to create a basic operational semantics. It is based on Chapter 10 of the UTP book [14].

fun \( \text{rel} :: \alpha\ \text{usubst} \times \alpha\ \text{hrel} \Rightarrow \alpha\ \text{usubst} \times \alpha\ \text{hrel} \Rightarrow \text{bool} \) (infix \( \rightarrow_u \) 85) where

\( (\sigma, P) \rightarrow_u (\varphi, Q) \iff (\langle\sigma\rangle_a \conc \; P) \subseteq (\langle\varphi\rangle_a \conc \; Q) \)

lemma trans-trel:
  \[ [ (\sigma, P) \rightarrow_u (\varphi, Q); (\varphi, Q) \rightarrow_u (\sigma, R) ] \implies (\sigma, P) \rightarrow_u (\varphi, R) \]
  by auto

lemma skip-trel: \( (\sigma, II) \rightarrow_u (\sigma, II) \)
  by simp

lemma assigns-trel: \( (\sigma, (\varphi)_a) \rightarrow_u (\varphi \circ \sigma, II) \)
  by (simp add: assigns-comp)

lemma assign-trel:
  \( (\sigma, x := v) \rightarrow_u (\sigma(\&x := \sigma \uparrow v), II) \)
  by (simp add: assigns-comp usubst)

lemma seq-trel:
  assumes \( (\sigma, P) \rightarrow_u (\varphi, Q) \)
  shows \( (\sigma, P \conc \; R) \rightarrow_u (\varphi, Q \conc \; R) \)
  by (metis (no-types, lifting) order-refl seqr-assoc seqr-mono trel.simps)

lemma seq-skip-trel:
  \( (\sigma, II \conc \; P) \rightarrow_u (\sigma, P) \)
  by simp

lemma nondet-left-trel:
  \( (\sigma, P \cap \; Q) \rightarrow_u (\sigma, P) \)
  by (metis (no-types, hide-lams) disj-comm disj-upred-def semilattice-sup-class.sup.absorb-iff1 semilattice-sup-class.sup.left-idem semilattice-sup-class.sup.absorb-iff1)

lemma nondet-right-trel:
  \( (\sigma, P \cap \; Q) \rightarrow_u (\sigma, Q) \)
  by (simp add: seqr-mono)

end
lemma rcond-true-trel:
  assumes σ † b = true
  shows (σ, P ◁ b ▷ Q) →u (σ, P)
  using assms
  by (simp add: assigns-r-comp usubst alpha cond-unit-T)

lemma rcond-false-trel:
  assumes σ † b = false
  shows (σ, P ◁ b ▷ Q) →u (σ, Q)
  using assms
  by (simp add: assigns-r-comp usubst alpha cond-unit-F)

lemma while-true-trel:
  assumes σ † b = true
  shows (σ, while b do P od) →u (σ, P ;; while b do P od)
  by (metis assms rcond-true-trel while-unfold)

lemma while-false-trel:
  assumes σ † b = false
  shows (σ, while b do P od) →u (σ, II)
  by (metis assms rcond-false-trel while-unfold)

Theorem linking Hoare calculus and operational semantics. If we start Q in a state σ₀ satisfying p, and Q reaches final state σ₁ then r holds in this final state.

theorem hoare-opsem-link:
  {p} Q {r} = (∀ σ₀ σ₁. ‘σ₀ † p’ ∧ (σ₀, Q) →u (σ₁, II) −→ ‘σ₁ † r’)
  apply (rel-auto)
  apply (rename-tac a b)
  apply (drule-tac x=λ - a in spec, simp)
  apply (drule-tac x=λ - 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 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
State-space with a countable universe for local variables.

\textbf{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.

\textbf{definition} local-num where
\texttt{local-num n P = [\#_u(&store) = u ≪ n≫]} \top \;; P \;;; \texttt{[\#_u(&store) = u ≪ n≫]} \top

\textbf{declare} inj-univ.from-univ-def [upred-defs]
\textbf{declare} inj-univ.to-univ-lens-def [upred-defs]
\textbf{declare} nat-inj-univ-def [upred-defs]

\section*{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.

\textbf{record} ('a, 's) state-prim =

— The first field states where in the alphabet 'a the user state-space type is 's is located with the form of a lens.

\texttt{sstate :: 's \implies 'a (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.

\texttt{sassigns :: 's usubst \Rightarrow 'a hrel (\langle \cdot \rangle_i)}

\textbf{syntax}
\texttt{-sstate :: logic \Rightarrow svid (si)}

\textbf{translations}
\texttt{-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.

\textbf{record} ('a, 's, 'u, 'a) local-prim = ('a, ('u, 's) local-scheme) state-prim +
\texttt{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 \texttt{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.

\textbf{locale} utp-state =
\textbf{fixes} S (structure)
\textbf{assumes} vwb-lens \texttt{s}
\textbf{and} passigns-comp: (\langle \sigma \rangle ;; \langle \varnothing \rangle) = (\varnothing \circ \sigma)

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 

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_v) where
var-open = (\(\prod_v \cdot ([\text{store} \mapsto s (\&\text{store} \cdot v \cdot (\&<v>))])))

**definition** var-close :: 'a hrel (\text{close}_v) where
var-close = ([\text{store} \mapsto s \cdot \text{front}_v (\&\text{store}) \cdot \#_{\text{store}} > u \cdot \&\text{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 => ('u, 'b) local-scheme, 'a) uexpr (top_v) where
top-var = \(\lambda l. \text{to-univ-lens} \cdot \text{list-lens} \cdot \text{l} \cdot \text{store} \cdot (\#_{\text{store}} - 1)_{a}

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 => ('u, 's) local-scheme) => 'a hrel) => 'a hrel where
var-scope \(f = \text{open}_v \cdot f(x)[x \mapsto [\text{top}_v]] \mapsto \text{close}_v\)

end

**notation** utp-local-state, var-open (open[\cdot])
**notation** utp-local-state, var-close (\text{close}[\cdot])
**notation** utp-local-state, var-scope (\text{Var}[\cdot])
**notation** utp-local-state, top-var (top[\cdot])

**syntax**
\(-\text{var-scope} :: \text{logic} \mapsto \text{id} \mapsto \text{logic} \mapsto \text{logic} (\text{var}[\cdot]) - \cdots - [0, 0, 10] \cdot 10\)
\(-\text{var-scope-type} :: \text{logic} \mapsto \text{id} \mapsto \text{type} \mapsto \text{logic} \mapsto \text{logic} (\text{var}[\cdot]) - \cdot \cdot - [0, 0, 10] \cdot 10\)

**translations**
\(-\text{var-scope} T \times P == \text{CONST utp-local-state, var-scope} T (\lambda x. P)\)
\(-\text{var-scope-type} T \times t P => \text{CONST utp-local-state, var-scope} T (-\text{abs} (-\text{constrain} x (-\text{var-ty} t)) P)\)

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:
\([x \mapsto \text{store}; \text{store} \cdot v \cdot] \Longrightarrow ([x \mapsto s v]) \mapsto \text{open}_v = \text{open}_v \mapsto ([x \mapsto s v])\)
\(\text{by (simp add: var-open-def passigns-comp seq-UINF-distl seq-UINF-distr usubst unrest lens-indep-sym, simp add: usubst-upd-comm)}\)

**lemma** var-close-commute:
\([x \mapsto \text{store}; \text{store} \cdot v \cdot] \Longrightarrow ([x \mapsto s v]) \mapsto \text{close}_v = \text{close}_v \mapsto ([x \mapsto s v])\)
\(\text{by (simp add: var-close-def passigns-comp seq-UINF-distl seq-UINF-distr usubst unrest lens-indep-sym, simp add: usubst-upd-comm)}\)
lemma \( \text{var-open-close-lemma} \):
\[
\begin{align*}
[s \mapsto s \times (\& (\text{store} u \langle \ll v \gg \rangle) \triangleright \& \text{store} u \langle \ll v \gg \rangle)] &= \text{id} \\
\text{by (rel-auto)}
\end{align*}
\]

lemma \( \text{var-open-close} \): \( \text{open}_v \); \( \text{close}_v = \{\text{id}\} \)
by (simp add: var-open-def var-close-def seq-UINF-distr' assigns-comp usubst var-open-close-lemma)

lemma \( \text{var-scope-skip} \): \( \text{var} [S] x \cdot \langle \text{id} \rangle \)
by (simp add: var-scope-def var-open-def var-close-def seq-UINF-distr' assigns-comp var-open-close-lemma usubst)

lemma \( \text{var-scope-nonlocal-left} \):
\[
\begin{align*}
\ll x \equiv \text{store} \& \text{store} \ll v \gg \rr \implies ([x \mapsto s \ll v \gg] \& (\var [S] y \cdot P(y))) &= (\var [S] y \cdot (\ll x \mapsto s \ll v \gg) \& P(y))
\end{align*}
\]

\( \text{oops} \)
\end{end}

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 \( \text{rel-local-state} :: \)
\[
'a::\text{countable} \Rightarrow ((\text{nat}, 's) \text{ local-scheme}, 's, \text{nat}, 'a::\text{countable}) \text{ local-prim where}
\]
\[
\text{rel-local-state} t = ([\text{sstate} = 1], \text{assigns} = \text{assigns-r}, \text{inj-local} = \text{nat-inj-univ})
\]

abbreviation \( \text{rel-local} (R_t) \) where
\( \text{rel-local} \equiv \text{rel-local-state TYPE('a::\text{countable})} \)

\[\text{syntax} \]
\( -\text{rel-local-state-type} :: \text{type} \Rightarrow \text{logic} (R_t[-]) \)
\( \text{translations} \)
\( -\text{rel-local-state-type} \ a = > \text{CONST rel-local-state (-TYPE a)} \)

\[\text{lemma get-rel-local} [\text{lens-defs}]: \]
\( \text{get}_{R_t} = \text{id} \)
\by (simp add: rel-local-state-def lens-defs)

\[\text{lemma rel-local-state} [\text{simp}]: \text{utp-local-state } R_t \]
\by (unfold-locales, simp-all add: upred-defs assigns-comp rel-local-state-def)

\[\text{lemma sassigns-rel-state} [\text{simp}]: (\sigma)_{R_t} = (\sigma)_a \]
\by (simp add: rel-local-state-def)

\[\text{syntax} \]
\( -\text{rel-var-scope} :: \text{id} \Rightarrow \text{logic} \Rightarrow \text{logic} (\text{var - - [0, 10] 10}) \)
\( -\text{rel-var-scope-type} :: \text{id} \Rightarrow \text{type} \Rightarrow \text{logic} \Rightarrow \text{logic} (\text{var - - - [0, 0, 10] 10}) \)
translations
- \(\text{rel-var-scope} \ x \ P \implies \text{var-scope} \ R_1 \ x \ P\)
- \(\text{rel-var-scope-type} \ x \ t \ P \implies \text{var-scope-type} \ (\text{rel-local-state-type} \ t) \ x \ t \ P\)

Next we prove some examples laws.

**lemma** rel-var-ex-1: \((\text{var} \ x :: \text{string} \cdot \Pi) = \Pi\)
by (rel-auto’)

**lemma** rel-var-ex-2: \((\text{var} \ x \cdot x := 1) = \Pi\)
by (rel-auto’)

**lemma** rel-var-ex-3: \(x \triangleright \text{store} \implies x := 1 \implies \text{open}[R_1[‘a::\text{countable}]] = \text{open}[R_1[‘a]] \implies x := 1\)
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 \text{store}; \text{store} \# v] \implies x := v \implies \text{open}[R_1[‘a::\text{countable}]] = \text{open}[R_1[‘a]] \implies 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 \text{store}; \text{store} \# v] \implies x := v \implies (\text{var} \ y :: \int \cdot P) = (\text{var} \ y :: \int \cdot x := v \implies P)\)

end

26 Meta-theory for the Standard Core
References


