

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

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

April 26, 2019

Abstract

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

Contents

1	Introduction	7
2	UTP Variables	8
2.1	Initial syntax setup	8
2.2	Variable foundations	9
2.3	Variable lens properties	9
2.4	Lens simplifications	11
2.5	Syntax translations	12
3	UTP Expressions	14
3.1	Expression type	14
3.2	Core expression constructs	15
3.3	Type class instantiations	16
3.4	Syntax translations	17
3.5	Evaluation laws for expressions	18
3.6	Misc laws	19
3.7	Literalise tactics	19
4	Expression Type Class Instantiations	20
4.1	Expression construction from HOL terms	23
4.2	Lifting set collectors	26
4.3	Lifting limits	26

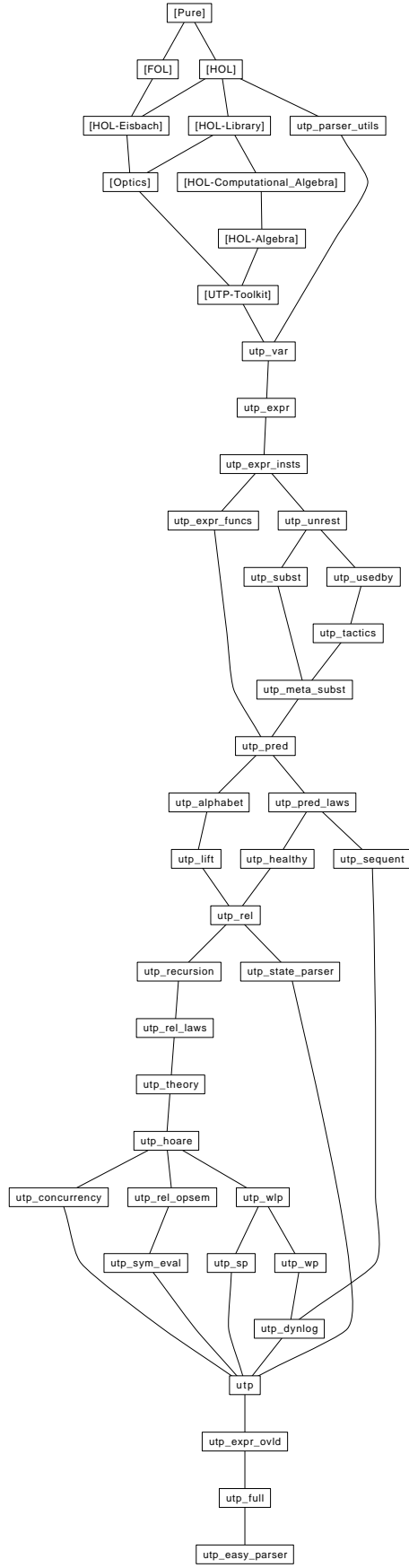
*Department of Computer Science, University of York. simon.foster@york.ac.uk

5	Unrestriction	27
5.1	Definitions and Core Syntax	27
5.2	Unrestriction laws	28
6	Used-by	31
7	Substitution	33
7.1	Substitution definitions	33
7.2	Syntax translations	34
7.3	Substitution Application Laws	35
7.4	Substitution laws	38
7.5	Ordering substitutions	40
7.6	Unrestriction laws	41
7.7	Conditional Substitution Laws	41
7.8	Parallel Substitution Laws	41
8	UTP Tactics	43
8.1	Theorem Attributes	43
8.2	Generic Methods	43
8.3	Transfer Tactics	44
8.3.1	Robust Transfer	44
8.3.2	Faster Transfer	44
8.4	Interpretation	45
8.5	User Tactics	45
9	Meta-level Substitution	47
10	Alphabetised Predicates	48
10.1	Predicate type and syntax	48
10.2	Predicate operators	49
10.3	Unrestriction Laws	54
10.4	Used-by laws	56
10.5	Substitution Laws	56
10.6	Sandbox for conjectures	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	64
11.6	Substitution Alphabet Extension	64
11.7	Substitution Alphabet Restriction	65
12	Lifting Expressions	65
12.1	Lifting definitions	65
12.2	Lifting Laws	66
12.3	Substitution Laws	66
12.4	Unrestriction laws	66

13 Predicate Calculus Laws	66
13.1 Propositional Logic	66
13.2 Lattice laws	70
13.3 Equality laws	75
13.4 HOL Variable Quantifiers	76
13.5 Case Splitting	77
13.6 UTP Quantifiers	78
13.7 Variable Restriction	80
13.8 Conditional laws	80
13.9 Additional Expression Laws	82
13.10Refinement By Observation	82
13.11Cylindric Algebra	83
14 Healthiness Conditions	83
14.1 Main Definitions	84
14.2 Properties of Healthiness Conditions	85
15 Alphabetised Relations	89
15.1 Relational Alphabets	90
15.2 Relational Types and Operators	91
15.3 Syntax Translations	94
15.4 Relation Properties	96
15.5 Introduction laws	96
15.6 Unrestriction Laws	96
15.7 Substitution laws	98
15.8 Alphabet laws	99
15.9 Relational unrestriction	100
16 Fixed-points and Recursion	103
16.1 Fixed-point Laws	103
16.2 Obtaining Unique Fixed-points	104
16.3 Noetherian Induction Instantiation	105
17 Sequent Calculus	107
18 Relational Calculus Laws	108
18.1 Conditional Laws	108
18.2 Precondition and Postcondition Laws	108
18.3 Sequential Composition Laws	108
18.4 Iterated Sequential Composition Laws	112
18.5 Quantale Laws	112
18.6 Skip Laws	113
18.7 Assignment Laws	113
18.8 Non-deterministic Assignment Laws	115
18.9 Converse Laws	115
18.10Assertion and Assumption Laws	116
18.11Frame and Antiframe Laws	116
18.12While Loop Laws	119
18.13Algebraic Properties	120
18.14Kleene Star	122

18.15 Kleene Plus	122
18.16 Omega	122
18.17 Relation Algebra Laws	122
18.18 Kleene Algebra Laws	122
18.19 Omega Algebra Laws	124
18.20 Refinement Laws	124
18.21 Domain and Range Laws	125
19 UTP Theories	126
19.1 Complete lattice of predicates	126
19.2 UTP theories hierarchy	127
19.3 UTP theory hierarchy	127
19.4 Theory of relations	135
19.5 Theory links	135
20 Relational Hoare calculus	136
20.1 Hoare Triple Definitions and Tactics	136
20.2 Basic Laws	137
20.3 Assignment Laws	137
20.4 Sequence Laws	138
20.5 Conditional Laws	138
20.6 Recursion Laws	138
20.7 Iteration Rules	139
20.8 Frame Rules	141
21 Weakest Liberal Precondition Calculus	142
22 Weakest Precondition Calculus	143
23 Dynamic Logic	144
23.1 Definitions	145
23.2 Box Laws	145
23.3 Diamond Laws	145
23.4 Sequent Laws	146
24 State Variable Declaration Parser	146
24.1 Examples	147
25 Relational Operational Semantics	147
26 Symbolic Evaluation of Relational Programs	149
27 Strongest Postcondition Calculus	150
28 Concurrent Programming	151
28.1 Variable Renamings	152
28.2 Merge Predicates	153
28.3 Separating Simulations	154
28.4 Associative Merges	155
28.5 Parallel Operators	156
28.6 Unrestriction Laws	156

28.7 Substitution laws	156
28.8 Parallel-by-merge laws	157
28.9 Example: Simple State-Space Division	159
29 Meta-theory for the Standard Core	161
30 Overloaded Expression Constructs	161
30.1 Overloadable Constants	161
30.2 Syntax Translations	162
30.3 Simplifications	163
30.4 Indexed Assignment	164
31 Meta-theory for the Standard Core with Overloaded Constructs	164
32 UTP Easy Expression Parser	164
32.1 Replacing the Expression Grammar	164
32.2 Expression Operators	165
32.3 Predicate Operators	165
32.4 Arithmetic Operators	166
32.5 Sets	166
32.6 Lists	167
32.7 Imperative Program Syntax	167



1 Introduction

This document contains the description of our mechanisation of Hoare and He’s *Unifying Theories of Programming* [22, 7] (UTP) in Isabelle/HOL. UTP uses the “programs-as-predicates” approach, pioneered by Hehner [20, 18, 19], to encode denotational semantics and facilitate reasoning about programs. It uses the alphabetised relational calculus, which combines predicate calculus and relation algebra, to denote programs as relations between initial variables (x) and their subsequent values (x'). Isabelle/UTP¹ [16, 28, 15] semantically embeds this relational calculus into Isabelle/HOL, which enables application of the latter’s proof facilities to program verification. For an introduction to UTP, we recommend two tutorials [6, 7], and also the UTP book [22].

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

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

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

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

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

¹Isabelle/UTP website: <https://www.cs.york.ac.uk/circus/isabelle-utp/>

We give a summary of the main contributions within the Isabelle/UTP core, which can all be seen in the table of contents.

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

2 UTP Variables

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

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

2.1 Initial syntax setup

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

```
purge-notation
  Order.le (infixl  $\sqsubseteq_1$  50) and
  Lattice.sup ( $\sqcup_1$ - [90] 90) and
  Lattice.inf ( $\sqcap_1$ - [90] 90) and
  Lattice.join (infixl  $\sqcup_1$  65) and
  Lattice.meet (infixl  $\sqcap_1$  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-inv-bij [simp]
declare id-bij-lens [simp]
declare lens-indep-left-ext [simp]
declare lens-indep-right-ext [simp]
declare lens-comp-quotient [simp]
declare plus-lens-distr [THEN sym, simp]
declare lens-comp-assoc [simp]

```

2.2 Variable foundations

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

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

definition *in-var* :: $('a \Longrightarrow '\alpha) \Rightarrow ('a \Longrightarrow '\alpha \times '\beta)$ **where**
[lens-defs]: in-var $x = x ;_L \text{fst}_L$

definition *out-var* :: $('a \Longrightarrow '\beta) \Rightarrow ('a \Longrightarrow '\alpha \times '\beta)$ **where**
[lens-defs]: out-var $x = x ;_L \text{snd}_L$

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* :: $('a \Longrightarrow '\alpha) (\Sigma)$ **where**
univ-alpha $\equiv 1_L$

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

definition *pr-var* :: $('a \Longrightarrow '\beta) \Rightarrow ('a \Longrightarrow '\beta)$ **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*]:
 $\text{weak-lens } x \Longrightarrow \text{weak-lens } (\text{in-var } x)$
by (*simp add: comp-weak-lens in-var-def*)

lemma *in-var-semi-uvar* [*simp*]:
 $\text{mwb-lens } x \Longrightarrow \text{mwb-lens } (\text{in-var } x)$
by (*simp add: comp-mwb-lens in-var-def*)

lemma *pr-var-weak-lens* [*simp*]:
 $\text{weak-lens } x \Longrightarrow \text{weak-lens } (\text{pr-var } x)$
by (*simp add: pr-var-def*)

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

mbw-lens $x \implies \text{mbw-lens } (\text{pr-var } x)$
by (*simp add: pr-var-def*)

lemma *pr-var-vwb-lens* [*simp*]:
 $\text{vwb-lens } x \implies \text{vwb-lens } (\text{pr-var } x)$
by (*simp add: pr-var-def*)

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

lemma *out-var-weak-lens* [*simp*]:
 $\text{weak-lens } x \implies \text{weak-lens } (\text{out-var } x)$
by (*simp add: comp-weak-lens out-var-def*)

lemma *out-var-semi-uvar* [*simp*]:
 $\text{mbw-lens } x \implies \text{mbw-lens } (\text{out-var } x)$
by (*simp add: comp-mbw-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*]:
 $x \bowtie y \implies \text{pr-var } x \bowtie y$
 $x \bowtie y \implies x \bowtie \text{pr-var } y$
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*]:

in-var (*pr-var* *x*) = *in-var* *x*
by (*simp add: pr-var-def*)

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

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

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

lemma *pr-var-lens-comp-1* [*simp*]:
pr-var *x* ;_L *y* = *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*)

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

Similar properties follow for sublens

lemma *in-var-sublens* [*simp*]:
 $y \subseteq_L x \implies \text{in-var } y \subseteq_L \text{in-var } x$
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-l* [*simp*]: $a \subseteq_L b \implies \text{pr-var } (a) \subseteq_L b$
by (*simp add: pr-var-def*)

lemma *pr-var-sublens-r* [*simp*]: $a \subseteq_L b \implies a \subseteq_L \text{pr-var } (b)$
by (*simp add: pr-var-def*)

2.4 Lens simplifications

We also define some lookup abstraction simplifications.

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

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

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

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

lemma *get-lens-plus* [*simp*]: $get_x +_L y \ s = (get_x \ s, get_y \ s)$
by (*simp add: lens-defs*)

2.5 Syntax translations

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

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

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

syntax — Identifiers

-*svid* :: $id \Rightarrow svid$ (- [999] 999)
 -*sulongid* :: $longid \Rightarrow svid$ (- [999] 999)
 -*svid-unit* :: $svid \Rightarrow svids$ (-)
 -*svid-list* :: $svid \Rightarrow svids \Rightarrow svids$ (-, / -)
 -*svid-alpha* :: $svid$ (**v**)
 -*svid-dot* :: $svid \Rightarrow svid \Rightarrow svid$ (-:- [998,999] 998)
 -*mk-svid-list* :: $svids \Rightarrow logic$ — Helper function for summing a list of identifiers

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

syntax — Decorations

-*svar* :: $svid \Rightarrow svar$ (&- [990] 990)
 -*sinvar* :: $svid \Rightarrow svar$ (\$- [990] 990)
 -*soutvar* :: $svid \Rightarrow svar$ (\$-' [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* :: $svid \Rightarrow salpha$ (- [990] 990)
 -*salphavar* :: $svar \Rightarrow salpha$ (- [990] 990)
 -*salphaparen* :: $salpha \Rightarrow salpha$ ('(-'))
 -*salphacomp* :: $salpha \Rightarrow salpha \Rightarrow salpha$ (**infixr** ; 75)
 -*salphaprod* :: $salpha \Rightarrow salpha \Rightarrow salpha$ (**infixr** \times 85)
 -*salpha-all* :: $salpha$ (Σ)
 -*salpha-none* :: $salpha$ (\emptyset)
 -*svar-nil* :: $svar \Rightarrow svars$ (-)
 -*svar-cons* :: $svar \Rightarrow svars \Rightarrow svars$ (-, / -)
 -*salphaset* :: $svars \Rightarrow salpha$ ({-})
 -*salphamk* :: $logic \Rightarrow salpha$

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

$\text{-ualpha-set} :: \text{svars} \Rightarrow \text{logic } (\{-\}_\alpha)$
 $\text{-svar} :: \text{svar} \Rightarrow \text{logic } ('(-)_v)$

For various reasons, the syntax constructors above all yield specific grammar categories and will not parse 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

$\text{svar} :: 'v \Rightarrow 'e$
 $\text{ivar} :: 'v \Rightarrow 'e$
 $\text{ovar} :: 'v \Rightarrow 'e$

ad hoc-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

$\text{-svid } x \rightarrow x$
 $\text{-svlongid } x \rightarrow x$
 $\text{-svid-alpha} \Rightarrow \Sigma$
 $\text{-svid-dot } x \ y \rightarrow y ;_L x$
 $\text{-mk-svid-list } (\text{-svid-unit } x) \rightarrow x$
 $\text{-mk-svid-list } (\text{-svid-list } x \ xs) \rightarrow x +_L \text{-mk-svid-list } xs$

— Decorations

$\text{-spvar } \Sigma \leftarrow \text{CONST svar CONST id-lens}$
 $\text{-sinvar } \Sigma \leftarrow \text{CONST ivar } 1_L$
 $\text{-soutvar } \Sigma \leftarrow \text{CONST ovar } 1_L$
 $\text{-spvar } (\text{-svid-dot } x \ y) \leftarrow \text{CONST svar } (\text{CONST lens-comp } y \ x)$
 $\text{-sinvar } (\text{-svid-dot } x \ y) \leftarrow \text{CONST ivar } (\text{CONST lens-comp } y \ x)$
 $\text{-soutvar } (\text{-svid-dot } x \ y) \leftarrow \text{CONST ovar } (\text{CONST lens-comp } y \ x)$
 $\text{-svid-dot } (\text{-svid-dot } x \ y) \ z \leftarrow \text{-svid-dot } (\text{CONST lens-comp } y \ x) \ z$

$\text{-spvar } x \Rightarrow \text{CONST svar } x$
 $\text{-sinvar } x \Rightarrow \text{CONST ivar } x$
 $\text{-soutvar } x \Rightarrow \text{CONST ovar } x$

— Alphabets

$\text{-salphaparen } a \rightarrow a$
 $\text{-salphaid } x \rightarrow x$
 $\text{-salphacomp } x \ y \rightarrow x +_L y$
 $\text{-salphaprod } a \ b \Rightarrow a \times_L b$
 $\text{-salphavar } x \rightarrow x$

```

-svar-nil  $x \rightarrow x$ 
-svar-cons  $x \ xs \rightarrow x +_L \ xs$ 
-salphaset  $A \rightarrow A$ 
(-svar-cons  $x \ (-salphamk \ y)) \leftarrow -salphamk \ (x +_L \ y)$ 
 $x \leftarrow -salphamk \ x$ 
-salpha-all  $\Rightarrow 1_L$ 
-salpha-none  $\Rightarrow 0_L$ 

— Quotations
-ualpha-set  $A \rightarrow A$ 
-svar  $x \rightarrow x$ 

```

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

```
-uvar-ty      :: type  $\Rightarrow$  type  $\Rightarrow$  type
```

parse-translation (

let

```

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

```

```

in [(@{syntax-const -uvar-ty}, K uvar-ty-tr)] end

```

)

end

3 UTP Expressions

theory *utp-expr*

imports

utp-var

begin

3.1 Expression type

purge-notation *BNF-Def.convolve* ((-, / -))

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 $'a$. This general model will allow us to unify all constructions under one type. The majority definitions in the file are given using the *lifting* package [23], which allows us to reuse much of the existing library of HOL functions.

```
typedef ('t, 'α) uexpr = UNIV :: ('α  $\Rightarrow$  't) set ..
```

setup-lifting *type-definition-uexpr*

notation *Rep-uexpr* ($\llbracket - \rrbracket_e$)

notation *Abs-uexpr* (mk_e)

$$e = f \iff (\forall b. \llbracket e \rrbracket_e b = \llbracket f \rrbracket_e b)$$

using *Rep-expr-inject*[*of e f, THEN sym*] **by** (*auto*)

named-theorems *uexpr-defs* and *ueval* and *lit-simps* and *lit-norm*

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.

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

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.

$$\begin{aligned} & ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e) \Rightarrow \\ & ('a, ' \alpha) \text{ uexpr} \Rightarrow ('b, ' \alpha) \text{ uexpr} \Rightarrow ('c, ' \alpha) \text{ uexpr} \Rightarrow ('d, ' \alpha) \text{ uexpr} \Rightarrow \\ & ('e, ' \alpha) \text{ uexpr} \\ \text{is } & \lambda f u v w x b. f (u b) (v b) (w b) (x b) . \end{aligned}$$

where $P \triangleleft b \triangleright Q \equiv \text{trop } u \text{ If } b \text{ } P \text{ } Q$

UTP expression is equality is simply HOL equality lifted using the *bop* binary expression constructor.

definition *eq-upred* :: ('a, 'α) *uexpr* ⇒ ('a, 'α) *uexpr* ⇒ (bool, 'α) *uexpr* (**infixl** =_u 50)
where [*uexpr-defs*]: *eq-upred* *x y* = *bop HOL.eq x y*

A literal is the expression «*v*», where *v* is any HOL term. Actually, the literal construct is very versatile and also allows us to refer to HOL variables within UTP expressions, and has a variety of other uses. It can therefore also be considered as a kind of quotation mechanism.

We also set up syntax for UTP variable expressions.

syntax

-*uuvar* :: *svar* ⇒ *logic* (-)

translations

-*uuvar* *x* == *CONST var x*

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

3.3 Type class instantiations

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

instantiation *uexpr* :: (zero, *type*) zero

begin

definition *zero-uexpr-def* [*uexpr-defs*]: 0 = *lit* 0

instance ..

end

instantiation *uexpr* :: (one, *type*) one

begin

definition *one-uexpr-def* [*uexpr-defs*]: 1 = *lit* 1

instance ..

end

instantiation *uexpr* :: (*plus*, *type*) *plus*

begin

definition *plus-uexpr-def* [*uexpr-defs*]: *u* + *v* = *bop* (+) *u v*

instance ..

end

instance *uexpr* :: (*semigroup-add*, *type*) *semigroup-add*

by (*intro-classes*) (*simp add*: *plus-uexpr-def zero-uexpr-def*, *transfer*, *simp add*: *add.assoc*) +

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

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

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

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

```
instantiation uexpr :: (ord, type) ord
begin
  lift-definition less-eq-uexpr :: ('a', 'b') uexpr  $\Rightarrow$  ('a', 'b') uexpr  $\Rightarrow$  bool
  is  $\lambda$  P Q. ( $\forall$  A. P A  $\leq$  Q A) .
  definition less-uexpr :: ('a', 'b') uexpr  $\Rightarrow$  ('a', 'b') uexpr  $\Rightarrow$  bool
  where [uexpr-defs]: less-uexpr P Q = (P  $\leq$  Q  $\wedge$   $\neg$  Q  $\leq$  P)
instance ..
end
```

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

```
instance uexpr :: (order, type) order
proof
  fix x y z :: ('a', 'b') uexpr
  show (x < y) = (x  $\leq$  y  $\wedge$   $\neg$  y  $\leq$  x) by (simp add: less-uexpr-def)
  show x  $\leq$  x by (transfer, auto)
  show x  $\leq$  y  $\Rightarrow$  y  $\leq$  z  $\Rightarrow$  x  $\leq$  z
    by (transfer, blast intro: order.trans)
  show x  $\leq$  y  $\Rightarrow$  y  $\leq$  x  $\Rightarrow$  x = y
    by (transfer, rule ext, simp add: eq-iff)
qed
```

3.4 Syntax translations

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

abbreviation (*input*) *ulens-override x f g* \equiv *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* \Rightarrow '*a* *set* **where**
 [*uexpr-defs*]: *set-of t* = *UNIV*

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

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

syntax — Core expression constructs

```
-ucoerce    :: logic  $\Rightarrow$  type  $\Rightarrow$  logic (infix :u 50)
-ulambda    :: pttrn  $\Rightarrow$  logic  $\Rightarrow$  logic ( $\lambda$  - · - [0, 10] 10)
-ulens-ovrd :: logic  $\Rightarrow$  logic  $\Rightarrow$  salph  $\Rightarrow$  logic (-  $\oplus$  - on - [85, 0, 86] 86)
-ulens-get  :: logic  $\Rightarrow$  svar  $\Rightarrow$  logic (-:- [900, 901] 901)
-umem       :: ('a', 'α') uexpr  $\Rightarrow$  ('a set', 'α') uexpr  $\Rightarrow$  (bool, 'α') uexpr (infix  $\in_u$  50)
-uNone      :: logic (Noneu)
-uSome      :: logic  $\Rightarrow$  logic (Someu'(-'))
-uthe       :: logic  $\Rightarrow$  logic (theu'(-'))
```

translations

$\lambda x . p == \text{CONST ulambda } (\lambda x . p)$
 $x :_u 'a == x :: ('a, -) \text{ uexpr}$
 $\text{-ulens-ovrd } f g a \Rightarrow \text{CONST bop } (\text{CONST ulens-override } a) f g$
 $\text{-ulens-ovrd } f g a \leq \text{CONST bop } (\lambda x y . \text{CONST lens-override } x1 y1 a) f g$
 $\text{-ulens-get } x y == \text{CONST uop } (\text{CONST lens-get } y) x$
 $x \in_u A == \text{CONST bop } (\in) x A$
 $\text{-uNone} == \text{CONST lit } \text{CONST None}$
 $\text{-uSome } e == \text{CONST uop } \text{CONST Some } e$
 $\text{-uthe } x == \text{CONST uop } (\text{CONST the}) x$

syntax — Tuples

$\text{-utuple} :: ('a, 'α) \text{ uexpr} \Rightarrow \text{utuple-args} \Rightarrow ('a * 'b, 'α) \text{ uexpr} ((1'(-, -')_u))$
 $\text{-utuple-arg} :: ('a, 'α) \text{ uexpr} \Rightarrow \text{utuple-args } (-)$
 $\text{-utuple-args} :: ('a, 'α) \text{ uexpr} \Rightarrow \text{utuple-args} \Rightarrow \text{utuple-args} \quad (-, / -)$
 $\text{-uunit} :: ('a, 'α) \text{ uexpr } ('()_u)$
 $\text{-ufst} :: ('a \times 'b, 'α) \text{ uexpr} \Rightarrow ('a, 'α) \text{ uexpr } (\pi_1'(-))$
 $\text{-usnd} :: ('a \times 'b, 'α) \text{ uexpr} \Rightarrow ('b, 'α) \text{ uexpr } (\pi_2'(-))$

translations

$()_u == \langle\langle() \rangle\rangle$
 $(x, y)_u == \text{CONST bop } (\text{CONST Pair}) x y$
 $\text{-utuple } x (\text{-utuple-args } y z) == \text{-utuple } x (\text{-utuple-arg } (\text{-utuple } y z))$
 $\pi_1(x) == \text{CONST uop } \text{CONST fst } x$
 $\pi_2(x) == \text{CONST uop } \text{CONST snd } x$

syntax — Orders

$\text{-ules} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\mathbf{infix} <_u 50)$
 $\text{-uleq} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\mathbf{infix} \leq_u 50)$
 $\text{-ugreat} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\mathbf{infix} >_u 50)$
 $\text{-ugeq} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\mathbf{infix} \geq_u 50)$

translations

$x <_u y == \text{CONST bop } (<) x y$
 $x \leq_u y == \text{CONST bop } (\leq) x y$
 $x >_u y \Rightarrow y <_u x$
 $x \geq_u y \Rightarrow y \leq_u x$

Overloaded power syntax

overloading

$\text{uexprpow} \equiv \text{compow} :: \text{nat} \Rightarrow ('a, 's) \text{ uexpr} \Rightarrow ('a, 's) \text{ uexpr}$

begin

definition $\text{uexprpow} :: \text{nat} \Rightarrow ('a, 's) \text{ uexpr} \Rightarrow ('a, 's) \text{ uexpr}$ **where**

$[\text{uexpr-defs}]: \text{uexprpow } n e = \text{uop } (\text{compow } n) e$

end

3.5 Evaluation laws for expressions

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

lemma *lit-ueval* $[\text{ueval}]: \llbracket x \rrbracket_e b = x$

by (transfer, simp)

lemma var-ueval [ueval]: $\llbracket \text{var } x \rrbracket_e b = \text{get}_x b$
 by (transfer, simp)

lemma uop-ueval [ueval]: $\llbracket \text{uop } f \ x \rrbracket_e b = f (\llbracket x \rrbracket_e b)$
 by (transfer, simp)

lemma bop-ueval [ueval]: $\llbracket \text{bop } f \ x \ y \rrbracket_e b = f (\llbracket x \rrbracket_e b) (\llbracket y \rrbracket_e b)$
 by (transfer, simp)

lemma trop-ueval [ueval]: $\llbracket \text{trop } f \ x \ y \ z \rrbracket_e b = f (\llbracket x \rrbracket_e b) (\llbracket y \rrbracket_e b) (\llbracket z \rrbracket_e b)$
 by (transfer, simp)

lemma qtop-ueval [ueval]: $\llbracket \text{qtop } f \ x \ y \ z \ w \rrbracket_e b = f (\llbracket x \rrbracket_e b) (\llbracket y \rrbracket_e b) (\llbracket z \rrbracket_e b) (\llbracket w \rrbracket_e b)$
 by (transfer, simp)

3.6 Misc laws

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

lemma uop-const [simp]: $\text{uop id } u = u$
 by (transfer, simp)

lemma bop-const-1 [simp]: $\text{bop } (\lambda x \ y. \ y) \ u \ v = v$
 by (transfer, simp)

lemma bop-const-2 [simp]: $\text{bop } (\lambda x \ y. \ x) \ u \ v = u$
 by (transfer, simp)

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

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

3.7 Literalise tactics

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

lemma lit-fun-simps [lit_simps]:
 $\llbracket i \ x \ y \ z \ u \rrbracket = \text{qtop } i \ \llbracket x \rrbracket \ \llbracket y \rrbracket \ \llbracket z \rrbracket \ \llbracket u \rrbracket$
 $\llbracket h \ x \ y \ z \rrbracket = \text{trop } h \ \llbracket x \rrbracket \ \llbracket y \rrbracket \ \llbracket z \rrbracket$
 $\llbracket g \ x \ y \rrbracket = \text{bop } g \ \llbracket x \rrbracket \ \llbracket y \rrbracket$
 $\llbracket f \ x \rrbracket = \text{uop } f \ \llbracket x \rrbracket$
 by (transfer, simp)+

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

lemma numeral-uexpr-rep-eq [ueval]: $\llbracket \text{numeral } x \rrbracket_e b = \text{numeral } x$
 apply (induct x)
 apply (simp add: lit.rep-eq one-uexpr-def)
 apply (simp add: bop.rep-eq numeral-Bit0 plus-uexpr-def)

```

apply (simp add: bop.rep-eq lit.rep-eq numeral-code(3) one-uepr-def plus-uepr-def)
done

```

```

lemma numeral-uepr-simp: numeral  $x = \llbracket \text{numeral } x \rrbracket$ 
by (simp add: uepr-eq-iff numeral-uepr-rep-eq lit.rep-eq)

```

```

lemma lit-zero [lit-simps]:  $\llbracket 0 \rrbracket = 0$  by (simp add: uepr-defs)

```

```

lemma lit-one [lit-simps]:  $\llbracket 1 \rrbracket = 1$  by (simp add: uepr-defs)

```

```

lemma lit-plus [lit-simps]:  $\llbracket x + y \rrbracket = \llbracket x \rrbracket + \llbracket y \rrbracket$  by (simp add: uepr-defs, transfer, simp)

```

```

lemma lit-numeral [lit-simps]:  $\llbracket \text{numeral } n \rrbracket = \text{numeral } n$  by (simp add: numeral-uepr-simp)

```

In general unliteralising converts function applications to corresponding expression liftings. Since some operators, like $+$ and $*$, have specific operators we also have to use $uIf = If$

```

( $?x =_u ?y$ ) = bop (=) ?x ?y

```

```

0 =  $\llbracket 0 \rrbracket$ 

```

```

1 =  $\llbracket 1 \rrbracket$ 

```

```

? $u + ?v$  = bop (+) ? $u$  ? $v$ 

```

```

( $?P < ?Q$ ) = ( $?P \leq ?Q \wedge \neg ?Q \leq ?P$ )

```

```

set-of ? $t$  = UNIV

```

$?e^{?n} = uop$ (compow ? n) ? e in reverse to correctly interpret these. Moreover, numerals must be handled separately by first simplifying them and then converting them into UTP expression numerals; hence the following two simplification rules.

```

lemma lit-numeral-1: uop numeral  $x = Abs-uepr (\lambda b. \text{numeral } (\llbracket x \rrbracket_e b))$ 
by (simp add: uop-def)

```

```

lemma lit-numeral-2: Abs-uepr ( $\lambda b. \text{numeral } v$ ) = numeral  $v$ 
by (metis lit.abs-eq lit-numeral)

```

```

method literalise = (unfold lit-simps[THEN sym])

```

```

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

```

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

```

method uepr-simp uses_simps = ((literalise)?, simp add: lit-norm_simps, (unliteralise)?)

```

```

lemma (1::( $int, 'a$ ) uepr) +  $\llbracket 2 \rrbracket = 4 \longleftrightarrow \llbracket 3 \rrbracket = 4$ 
apply (literalise)
apply (uepr-simp) oops

```

```

end

```

4 Expression Type Class Instantiations

```

theory utp-expr-insts
imports utp-expr
begin

```

It should be noted that instantiating the unary minus class, *uminus*, will also provide negation

UTP predicates later.

```

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

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

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

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

instantiation uexpr :: (divide, type) divide
begin
  definition divide-uexpr :: ('a, 'b) uexpr  $\Rightarrow$  ('a, 'b) uexpr  $\Rightarrow$  ('a, 'b) uexpr where
    [uexpr-defs]: divide-uexpr u v = bop divide u v
instance ..
end

instantiation uexpr :: (inverse, type) inverse
begin
  definition inverse-uexpr :: ('a, 'b) uexpr  $\Rightarrow$  ('a, 'b) uexpr
  where [uexpr-defs]: inverse-uexpr u = uop inverse u
instance ..
end

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

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

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

```

Once we've set up all the core constructs for arithmetic, we can also instantiate the type classes for various algebras, including groups and rings. The proofs are done by definitional expansion,

the *transfer* tactic, and then finally the theorems of the underlying HOL operators. This is mainly routine, so we don't comment further.

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

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

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

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

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

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

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

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

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

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

We 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 uminus-uexpr-def, transfer, simp
add: abs-ge-self abs-le-iff abs-triangle-ineq)+
  apply (metis ab-group-add-class.ab-diff-conv-add-uminus abs-ge-minus-self abs-ge-self add-mono-thms-linordered-semiri
done)
```

The next theorem lifts powers.

```
lemma power-rep-eq [ueval]:  $\llbracket P \wedge n \rrbracket_e = (\lambda b. \llbracket P \rrbracket_e b \wedge n)$ 
  by (induct n, simp-all add: lit.rep-eq one-uexpr-def bop.rep-eq times-uexpr-def)
```

```
lemma of-nat-uexpr-rep-eq [ueval]:  $\llbracket \text{of-nat } x \rrbracket_e b = \text{of-nat } x$ 
  by (induct x, simp-all add: uexpr-defs ueval)
```

```

lemma lit-uminus [lit-simps]:  $\llbracket - x \rrbracket = - \llbracket x \rrbracket$  by (simp add: uexpr-defs, transfer, simp)
lemma lit-minus [lit-simps]:  $\llbracket x - y \rrbracket = \llbracket x \rrbracket - \llbracket y \rrbracket$  by (simp add: uexpr-defs, transfer, simp)
lemma lit-times [lit-simps]:  $\llbracket x * y \rrbracket = \llbracket x \rrbracket * \llbracket y \rrbracket$  by (simp add: uexpr-defs, transfer, simp)
lemma lit-divide [lit-simps]:  $\llbracket x / y \rrbracket = \llbracket x \rrbracket / \llbracket y \rrbracket$  by (simp add: uexpr-defs, transfer, simp)
lemma lit-div [lit-simps]:  $\llbracket x \text{ div } y \rrbracket = \llbracket x \rrbracket \text{ div } \llbracket y \rrbracket$  by (simp add: uexpr-defs, transfer, simp)
lemma lit-power [lit-simps]:  $\llbracket x ^ n \rrbracket = \llbracket x \rrbracket ^ n$  by (simp add: lit.rep-eq power-rep-eq uexpr-eq-iff)

```

4.1 Expression construction from HOL terms

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

named-theorems *mkueexpr*

```

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

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

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

lemma mkueexpr-numeral [mkueexpr]:  $mk_e (\lambda s. \text{numeral } n) = \text{numeral } n$ 
using lit-numeral-2 by blast

lemma mkueexpr-lit [mkueexpr]:  $mk_e (\lambda s. k) = \llbracket k \rrbracket$ 
by (transfer, simp)

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

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

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

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

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

lemma mkueexpr-divide [mkueexpr]:  $mk_e (\lambda s. f s / g s) = mk_e f / mk_e g$ 
by (simp add: divide-uexpr-def, transfer, simp)

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

syntax — Polymorphic constructs
  -uceil      :: logic  $\Rightarrow$  logic ( $\lceil \cdot \rceil_u$ )
  -ufloor    :: logic  $\Rightarrow$  logic ( $\lfloor \cdot \rfloor_u$ )
  -umin      :: logic  $\Rightarrow$  logic  $\Rightarrow$  logic ( $\min_u'(-, -')$ )

```

$-umax \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\max_u'(-, -'))$
 $-ugcd \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\gcd_u'(-, -'))$

translations

— Type-class polymorphic constructs
 $\min_u(x, y) == \text{CONST bop} \ (\text{CONST min}) \ x \ y$
 $\max_u(x, y) == \text{CONST bop} \ (\text{CONST max}) \ x \ y$
 $\gcd_u(x, y) == \text{CONST bop} \ (\text{CONST gcd}) \ x \ y$
 $\lceil x \rceil_u == \text{CONST uop} \ \text{CONST ceiling} \ x$
 $\lfloor x \rfloor_u == \text{CONST uop} \ \text{CONST floor} \ x$

syntax — Lists / Sequences

$-ucons \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\text{infixr} \ \#_u \ 65)$
 $-unil \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \ (\langle \rangle)$
 $-ulist \quad :: \text{args} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \ (\langle (-) \rangle)$
 $-uappend \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \ (\text{infixr} \ \hat{\ }_u \ 80)$
 $-udconcat \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\text{infixr} \ \frown_u \ 90)$
 $-ulast \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a, 'a) \ \text{uexpr} \ (\text{last}_u'(-))$
 $-ufront \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \ (\text{front}_u'(-))$
 $-uhead \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a, 'a) \ \text{uexpr} \ (\text{head}_u'(-))$
 $-utail \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \ (\text{tail}_u'(-))$
 $-utake \quad :: (\text{nat}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \ (\text{take}_u'(-, -'))$
 $-udrop \quad :: (\text{nat}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \ (\text{drop}_u'(-, -'))$
 $-ufilter \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{set}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \ (\text{infixl} \ \downarrow_u \ 75)$
 $-uextract \quad :: ('a \ \text{set}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{list}, 'a) \ \text{uexpr} \ (\text{infixl} \ \uparrow_u \ 75)$
 $-uelems \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow ('a \ \text{set}, 'a) \ \text{uexpr} \ (\text{elems}_u'(-))$
 $-usorted \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow (\text{bool}, 'a) \ \text{uexpr} \ (\text{sorted}_u'(-))$
 $-udistinct \quad :: ('a \ \text{list}, 'a) \ \text{uexpr} \Rightarrow (\text{bool}, 'a) \ \text{uexpr} \ (\text{distinct}_u'(-))$
 $-uupto \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\langle \dots \rangle)$
 $-uupt \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\langle \dots < \rangle)$
 $-umap \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\text{map}_u)$
 $-uzip \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\text{zip}_u)$

translations

$x \ \#_u \ ys == \text{CONST bop} \ (\#) \ x \ ys$
 $\langle \rangle == \llbracket \rrbracket$
 $\langle x, xs \rangle == x \ \#_u \ \langle xs \rangle$
 $\langle x \rangle == x \ \#_u \ \llbracket \rrbracket$
 $x \ \hat{\ }_u \ y == \text{CONST bop} \ (@) \ x \ y$
 $A \ \frown_u \ B == \text{CONST bop} \ (\frown) \ A \ B$
 $\text{last}_u(xs) == \text{CONST uop} \ \text{CONST last} \ xs$
 $\text{front}_u(xs) == \text{CONST uop} \ \text{CONST butlast} \ xs$
 $\text{head}_u(xs) == \text{CONST uop} \ \text{CONST hd} \ xs$
 $\text{tail}_u(xs) == \text{CONST uop} \ \text{CONST tl} \ xs$
 $\text{drop}_u(n, xs) == \text{CONST bop} \ \text{CONST drop} \ n \ xs$
 $\text{take}_u(n, xs) == \text{CONST bop} \ \text{CONST take} \ n \ xs$
 $\text{elems}_u(xs) == \text{CONST uop} \ \text{CONST set} \ xs$
 $\text{sorted}_u(xs) == \text{CONST uop} \ \text{CONST sorted} \ xs$
 $\text{distinct}_u(xs) == \text{CONST uop} \ \text{CONST distinct} \ xs$
 $xs \ \downarrow_u \ A == \text{CONST bop} \ \text{CONST seq-filter} \ xs \ A$
 $A \ \uparrow_u \ xs == \text{CONST bop} \ (\uparrow_l) \ A \ xs$
 $\langle n..k \rangle == \text{CONST bop} \ \text{CONST upto} \ n \ k$
 $\langle n..<k \rangle == \text{CONST bop} \ \text{CONST upt} \ n \ k$
 $\text{map}_u f \ xs == \text{CONST bop} \ \text{CONST map} \ f \ xs$
 $\text{zip}_u \ xs \ ys == \text{CONST bop} \ \text{CONST zip} \ xs \ ys$

syntax — Sets

$-ufinite :: logic \Rightarrow logic (finite_u '(-))$
 $-uempset :: ('a\ set, '\alpha)\ uexpr (\{\}_u)$
 $-uset :: args \Rightarrow ('a\ set, '\alpha)\ uexpr (\{(-)\}_u)$
 $-uunion :: ('a\ set, '\alpha)\ uexpr \Rightarrow ('a\ set, '\alpha)\ uexpr \Rightarrow ('a\ set, '\alpha)\ uexpr (\mathbf{infixl}\ \cup_u\ 65)$
 $-uinter :: ('a\ set, '\alpha)\ uexpr \Rightarrow ('a\ set, '\alpha)\ uexpr \Rightarrow ('a\ set, '\alpha)\ uexpr (\mathbf{infixl}\ \cap_u\ 70)$
 $-uinsert :: logic \Rightarrow logic \Rightarrow logic (insert_u)$
 $-uimage :: logic \Rightarrow logic \Rightarrow logic (-\lceil-\rceil_u\ [10,0]\ 10)$
 $-usubset :: ('a\ set, '\alpha)\ uexpr \Rightarrow ('a\ set, '\alpha)\ uexpr \Rightarrow (bool, '\alpha)\ uexpr (\mathbf{infix}\ \subset_u\ 50)$
 $-usubseteq :: ('a\ set, '\alpha)\ uexpr \Rightarrow ('a\ set, '\alpha)\ uexpr \Rightarrow (bool, '\alpha)\ uexpr (\mathbf{infix}\ \subseteq_u\ 50)$
 $-uconverse :: logic \Rightarrow logic ((\sim) [1000]\ 999)$
 $-ucarrier :: type \Rightarrow logic (\lceil-\rceil_T)$
 $-uid :: type \Rightarrow logic (id[-])$
 $-uproduct :: logic \Rightarrow logic \Rightarrow logic (\mathbf{infixr}\ \times_u\ 80)$
 $-urelcomp :: logic \Rightarrow logic \Rightarrow logic (\mathbf{infixr}\ ;_u\ 75)$

translations

$finite_u(x) == CONST\ uop\ (CONST\ finite)\ x$
 $\{\}_u == \ll\{\}\gg$
 $insert_u\ x\ xs == CONST\ bop\ CONST\ insert\ x\ xs$
 $\{x, xs\}_u == insert_u\ x\ \{xs\}_u$
 $\{x\}_u == insert_u\ x\ \ll\{\}\gg$
 $A \cup_u B == CONST\ bop\ (\cup)\ A\ B$
 $A \cap_u B == CONST\ bop\ (\cap)\ A\ B$
 $f(\lceil A \rceil)_u == CONST\ bop\ CONST\ image\ f\ A$
 $A \subset_u B == CONST\ bop\ (\subset)\ A\ B$
 $f \subset_u g <= CONST\ bop\ (\subset_p)\ f\ g$
 $f \subset_u g <= CONST\ bop\ (\subset_f)\ f\ g$
 $A \subseteq_u B == CONST\ bop\ (\subseteq)\ A\ B$
 $f \subseteq_u g <= CONST\ bop\ (\subseteq_p)\ f\ g$
 $f \subseteq_u g <= CONST\ bop\ (\subseteq_f)\ f\ g$
 $P^\sim == CONST\ uop\ CONST\ converse\ P$
 $\lceil a \rceil_T == \ll CONST\ set-of\ TYPE('a) \gg$
 $id[\lceil a \rceil] == \ll CONST\ Id-on\ (CONST\ set-of\ TYPE('a)) \gg$
 $A \times_u B == CONST\ bop\ CONST\ Product-Type.Times\ A\ B$
 $A ;_u B == CONST\ bop\ CONST\ relcomp\ A\ B$

syntax — Partial functions

$-umap-plus :: logic \Rightarrow logic \Rightarrow logic (\mathbf{infixl}\ \oplus_u\ 85)$
 $-umap-minus :: logic \Rightarrow logic \Rightarrow logic (\mathbf{infixl}\ \ominus_u\ 85)$

translations

$f \oplus_u g \Rightarrow (f :: ((-, -)\ pfun, -)\ uexpr) + g$
 $f \ominus_u g \Rightarrow (f :: ((-, -)\ pfun, -)\ uexpr) - g$

syntax — Sum types

$-uinl :: logic \Rightarrow logic (inl_u '(-))$
 $-uinr :: logic \Rightarrow logic (inr_u '(-))$

translations

$inl_u(x) == CONST\ uop\ CONST\ Inl\ x$
 $inr_u(x) == CONST\ uop\ CONST\ Inr\ x$

4.2 Lifting set collectors

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

syntax

```
-uset-atLeastAtMost :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ ('a set, 'α) uexpr ((1{-..-}u))
-uset-atLeastLessThan :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ ('a set, 'α) uexpr ((1{-..<-}u))
-uset-compr :: pttrn ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr ⇒ ('b, 'α) uexpr ⇒ ('b set, 'α) uexpr
((1{- :/ - | / - · / -}u))
-uset-compr-nset :: pttrn ⇒ (bool, 'α) uexpr ⇒ ('b, 'α) uexpr ⇒ ('b set, 'α) uexpr ((1{- | / - · / -}u))
```

lift-definition ZedSetCompr ::

```
('a set, 'α) uexpr ⇒ ('a ⇒ (bool, 'α) uexpr × ('b, 'α) uexpr) ⇒ ('b set, 'α) uexpr
is λ A PF b. { snd (PF x) b | x. x ∈ A b ∧ fst (PF x) b } .
```

translations

```
{x..y}u == CONST bop CONST atLeastAtMost x y
{x..<y}u == CONST bop CONST atLeastLessThan x y
{x | P · F}u == CONST ZedSetCompr (CONST lit CONST UNIV) (λ x. (P, F))
{x : A | P · F}u == CONST ZedSetCompr A (λ x. (P, F))
```

4.3 Lifting limits

We also lift the following functions on topological spaces for taking function limits, and describing continuity.

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

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

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

syntax

```
-ulim-left :: id ⇒ logic ⇒ logic ⇒ logic (lim_u'(- → --)'(-))
-ulim-right :: id ⇒ logic ⇒ logic ⇒ logic (lim_u'(- → -+)'(-))
-ucont-on :: logic ⇒ logic ⇒ logic (infix cont-on_u 90)
```

translations

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

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

lemma *winter-empty-1* [simp]: $x \cap_u \{\}_u = \{\}_u$
by (transfer, simp)

lemma *winter-empty-2* [simp]: $\{\}_u \cap_u x = \{\}_u$
by (transfer, simp)

lemma *uunion-empty-1* [simp]: $\{\}_u \cup_u x = x$

by (transfer, simp)

lemma union-insert [simp]: $(bop\ insert\ x\ A) \cup_u B = bop\ insert\ x\ (A \cup_u B)$
 by (transfer, simp)

lemma ulist-filter-empty [simp]: $x \vdash_u \{\} = \langle \rangle$
 by (transfer, simp)

lemma tail-cons [simp]: $tail_u(\langle x \rangle \hat{\ }_u xs) = xs$
 by (transfer, simp)

lemma uconcat-units [simp]: $\langle \rangle \hat{\ }_u xs = xs\ xs\ \hat{\ }_u \langle \rangle = xs$
 by (transfer, simp)+

end

5 Unrestriction

theory utp-unrest
imports utp-expr-ists
begin

5.1 Definitions and Core Syntax

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

Unrestriction was first defined in the work of Marcel Oliveira [27, 26] 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 [9] and Oliveira's [26] 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
 $unrest :: 'a \Rightarrow 'b \Rightarrow bool$

syntax
 $-unrest :: salpha \Rightarrow logic \Rightarrow logic \Rightarrow logic\ (\mathbf{infix}\ \# 20)$

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 $unrest-uepr :: ('a \implies 'α) \Rightarrow ('b, 'α) uepr \Rightarrow bool$
is $\lambda x e. \forall b v. e (put_x b v) = e b$.

ad hoc-overloading

$unrest\ unrest-uepr$

lemma $unrest-expr-alt-def$:

$weak-lens\ x \implies (x \# P) = (\forall b b'. \llbracket P \rrbracket_e (b \oplus_L b' \text{ on } x) = \llbracket P \rrbracket_e b)$
by $(transfer,metis\ lens-override-def\ weak-lens.put-get)$

5.2 Unrestriction laws

We now prove unrestricted laws for the key constructs of our expression model. Many of these depend on lens properties and so variously employ the assumptions $mwb-lens$ and $vwb-lens$, 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.

lemma $unrest-var-comp\ [unrest]$:

$\llbracket x \# P; y \# P \rrbracket \implies x;y \# P$
by $(transfer, simp\ add: lens-defs)$

lemma $unrest-svar\ [unrest]$: $(\&x \# P) \longleftrightarrow (x \# P)$

by $(transfer, simp\ add: lens-defs)$

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

lemma $unrest-lit\ [unrest]$: $x \# \langle v \rangle$

by $(transfer, simp)$

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

lemma $unrest-sublens$:

fixes $P :: ('a, 'α) uepr$

assumes $x \# P\ y \subseteq_L x$

shows $y \# P$

using $assms$

by $(transfer,metis\ (no-types, lifting)\ lens.select-convs(2)\ lens-comp-def\ sublens-def)$

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

lemma $unrest-equiv$:

fixes $P :: ('a, 'α) uepr$

assumes $mwb-lens\ y\ x \approx_L y\ x \# P$

shows $y \# P$

by $(metis\ assms\ lens-equiv-def\ sublens-pres-mwb\ sublens-put-put\ unrest-uepr.rep-eq)$

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

lemma *bij-lens-unrest-all*:
fixes $P :: ('a, 'α) uexpr$
assumes *bij-lens* $X X \# P$
shows $\Sigma \# P$
using *assms bij-lens-equiv-id lens-equiv-def unrest-sublens* **by** *blast*

lemma *bij-lens-unrest-all-eq*:
fixes $P :: ('a, 'α) uexpr$
assumes *bij-lens* X
shows $(\Sigma \# P) \longleftrightarrow (X \# P)$
by (*meson assms bij-lens-equiv-id lens-equiv-def unrest-sublens*)

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

lemma *unrest-all-var*:
fixes $e :: ('a, 'α) uexpr$
assumes $\Sigma \# e$
shows $x \# e$
by (*metis assms id-lens-def lens.simps(2) unrest-uexpr.rep-eq*)

We can split an unrestricted composed by lens plus

lemma *unrest-plus-split*:
fixes $P :: ('a, 'α) uexpr$
assumes $x \bowtie y$ *vwb-lens* x *vwb-lens* y
shows *unrest* $(x +_L y) P \longleftrightarrow (x \# P) \wedge (y \# P)$
using *assms*
by (*meson lens-plus-right-sublens lens-plus-ub sublens-refl unrest-sublens unrest-var-comp vwb-lens-wb*)

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

lemma *unrest-var* [*unrest*]: $\llbracket mwb\text{-}lens\ x; x \bowtie y \rrbracket \Longrightarrow y \# var\ x$
by (*transfer, auto*)

lemma *unrest-iuvar* [*unrest*]: $\llbracket mwb\text{-}lens\ x; x \bowtie y \rrbracket \Longrightarrow \$y \# \$x$
by (*simp add: unrest-var*)

lemma *unrest-ouvar* [*unrest*]: $\llbracket mwb\text{-}lens\ x; x \bowtie y \rrbracket \Longrightarrow \$y' \# \$x'$
by (*simp add: unrest-var*)

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

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

lemma *unrest-ouvar-iuvar* [*unrest*]:
fixes $x :: ('a \Longrightarrow 'α)$
assumes *mwb-lens* y
shows $\$x' \# \y
by (*metis prod.collapse unrest-uexpr.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 unrestrictedions for the majority of the expression language.

lemma *unrest-uop* [*unrest*]: $x \# e \implies x \# uop\ f\ e$
by (*transfer*, *simp*)

lemma *unrest-bop* [*unrest*]: $\llbracket x \# u; x \# v \rrbracket \implies x \# bop\ f\ u\ v$
by (*transfer*, *simp*)

lemma *unrest-trop* [*unrest*]: $\llbracket x \# u; x \# v; x \# w \rrbracket \implies x \# trop\ f\ u\ v\ w$
by (*transfer*, *simp*)

lemma *unrest-qtop* [*unrest*]: $\llbracket x \# u; x \# v; x \# w; x \# y \rrbracket \implies x \# qtop\ f\ u\ v\ w\ y$
by (*transfer*, *simp*)

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

lemma *unrest-eq* [*unrest*]: $\llbracket x \# u; x \# v \rrbracket \implies x \# u =_u v$
by (*simp add: eq-upred-def*, *transfer*, *simp*)

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

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

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

lemma *unrest-sgn* [*unrest*]: $x \# u \implies x \# sgn\ u$
by (*simp add: sgn-uepr-def unrest-uop*)

lemma *unrest-abs* [*unrest*]: $x \# u \implies x \# abs\ u$
by (*simp add: abs-uepr-def unrest-uop*)

lemma *unrest-plus* [*unrest*]: $\llbracket x \# u; x \# v \rrbracket \implies x \# u + v$
by (*simp add: plus-uepr-def unrest*)

lemma *unrest-uminus* [*unrest*]: $x \# u \implies x \# -\ u$
by (*simp add: uminus-uepr-def unrest*)

lemma *unrest-minus* [*unrest*]: $\llbracket x \# u; x \# v \rrbracket \implies x \# u - v$
by (*simp add: minus-uepr-def unrest*)

lemma *unrest-times* [*unrest*]: $\llbracket x \# u; x \# v \rrbracket \implies x \# u * v$
by (*simp add: times-uepr-def unrest*)

lemma *unrest-divide* [*unrest*]: $\llbracket x \# u; x \# v \rrbracket \implies x \# u / v$
by (*simp add: divide-uepr-def unrest*)

lemma *unrest-case-prod* [*unrest*]: $\llbracket \bigwedge i\ j. x \# P\ i\ j \rrbracket \implies x \# case\text{-}prod\ P\ v$
by (*simp add: prod.split-sel-asm*)

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

lemma *unrest-ulambda* [*unrest*]:
 $\llbracket \bigwedge x. v \# F\ x \rrbracket \implies v \# (\lambda x. F\ x)$
by (*transfer*, *simp*)

end

6 Used-by

```
theory utp-usedby
  imports utp-unrest
begin
```

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

```
consts
  usedBy :: 'a  $\Rightarrow$  'b  $\Rightarrow$  bool
```

```
syntax
  -usedBy :: salpha  $\Rightarrow$  logic  $\Rightarrow$  logic  $\Rightarrow$  logic (infix  $\mathbin{\lhd}$  20)
```

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

```
lift-definition usedBy-uepr :: ('b  $\Longrightarrow$  'α)  $\Rightarrow$  ('a, 'α) uepr  $\Rightarrow$  bool
is  $\lambda$  x e. ( $\forall$  b b'. e (b'  $\oplus_L$  b on x) = e b) .
```

```
adhoc-overloading usedBy usedBy-uepr
```

```
lemma usedBy-lit [unrest]: x  $\mathbin{\lhd}$   $\ll v \gg$ 
  by (transfer, simp)
```

```
lemma usedBy-sublens:
  fixes P :: ('a, 'α) uepr
  assumes x  $\mathbin{\lhd}$  P x  $\subseteq_L$  y vwb-lens y
  shows y  $\mathbin{\lhd}$  P
  using assms
  by (transfer, auto, metis Lens-Order.lens-override-idem lens-override-def sublens-obs-get vwb-lens-mwb)
```

```
lemma usedBy-svar [unrest]: x  $\mathbin{\lhd}$  P  $\Longrightarrow$  &x  $\mathbin{\lhd}$  P
  by (transfer, simp add: lens-defs)
```

```
lemma usedBy-lens-plus-1 [unrest]: x  $\mathbin{\lhd}$  P  $\Longrightarrow$  x;y  $\mathbin{\lhd}$  P
  by (transfer, simp add: lens-defs)
```

```
lemma usedBy-lens-plus-2 [unrest]:  $\ll x \bowtie y; y \mathbin{\lhd} P \gg \Longrightarrow x;y \mathbin{\lhd} P$ 
  by (transfer, auto simp add: lens-defs lens-indep-comm)
```

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

```
lemma usedBy-indep-uses:
  fixes P :: ('a, 'α) uepr
  assumes x  $\mathbin{\lhd}$  P x  $\bowtie$  y
  shows y  $\nmid$  P
  using assms by (transfer, auto, metis lens-indep-get lens-override-def)
```

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

lemma *usedBy-uop* [*unrest*]: $x \Vdash e \implies x \Vdash \text{uop } f\ e$
by (*transfer*, *simp*)

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

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

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

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

lemma *usedBy-eq* [*unrest*]: $\llbracket x \Vdash u; x \Vdash v \rrbracket \implies x \Vdash u =_u v$
by (*simp add: eq-upred-def, transfer, simp*)

lemma *usedBy-zero* [*unrest*]: $x \Vdash 0$
by (*simp add: usedBy-lit zero-uexpr-def*)

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

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

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

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

lemma *usedBy-plus* [*unrest*]: $\llbracket x \Vdash u; x \Vdash v \rrbracket \implies x \Vdash u + v$
by (*simp add: plus-uexpr-def unrest*)

lemma *usedBy-uminus* [*unrest*]: $x \Vdash u \implies x \Vdash -\ u$
by (*simp add: uminus-uexpr-def unrest*)

lemma *usedBy-minus* [*unrest*]: $\llbracket x \Vdash u; x \Vdash v \rrbracket \implies x \Vdash u - v$
by (*simp add: minus-uexpr-def unrest*)

lemma *usedBy-times* [*unrest*]: $\llbracket x \Vdash u; x \Vdash v \rrbracket \implies x \Vdash u * v$
by (*simp add: times-uexpr-def unrest*)

lemma *usedBy-divide* [*unrest*]: $\llbracket x \Vdash u; x \Vdash v \rrbracket \implies x \Vdash u / v$
by (*simp add: divide-uexpr-def unrest*)


```

lemma usedBy-ulambda [unrest]:
   $\llbracket \bigwedge x. v \Vdash F x \rrbracket \implies v \Vdash (\lambda x. F x)$ 
  by (transfer, simp)

```

```

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

```

```

end

```

7 Substitution

```

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

```

7.1 Substitution definitions

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

```

consts
  usubst :: 's  $\Rightarrow$  'a  $\Rightarrow$  'b (infixr  $\dagger$  80)

```

```

named-theorems usubst

```

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

```

type-synonym (' $\alpha$ , ' $\beta$ ) psubst = ' $\alpha \Rightarrow$  ' $\beta$ 
type-synonym ' $\alpha$  usubst = ' $\alpha \Rightarrow$  ' $\alpha$ 

```

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

```

lift-definition subst :: (' $\alpha$ , ' $\beta$ ) psubst  $\Rightarrow$  ('a, ' $\beta$ ) uexpr  $\Rightarrow$  ('a, ' $\alpha$ ) uexpr is
 $\lambda \sigma e b. e (\sigma b)$  .

```

```

adhoc-overloading
  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.

```

consts subst-upd :: (' $\alpha$ , ' $\beta$ ) psubst  $\Rightarrow$  'v  $\Rightarrow$  ('a, ' $\alpha$ ) uexpr  $\Rightarrow$  (' $\alpha$ , ' $\beta$ ) psubst

```

The following function takes a substitution from state-space ' α to ' β , a lens with source ' β and view "' a ", and an expression over ' α and returning a value of type "' a ", and produces an updated

substitution. It does this by constructing a substitution function that takes state binding b , and updates the state first by applying the original substitution σ , and then updating the part of the state associated with lens x with expression evaluated in the context of b . This effectively means that x is now associated with expression v . We add this definition to our overloaded constant.

definition $\text{subst-upd-uvar} :: ('a, 'b) \text{psubst} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a, 'a) \text{uexpr} \Rightarrow ('a, 'b) \text{psubst}$ **where**
 $\text{subst-upd-uvar } \sigma \ x \ v = (\lambda \ b. \text{put}_x (\sigma \ b) (\llbracket v \rrbracket_e b))$

ad hoc-overloading

$\text{subst-upd } \text{subst-upd-uvar}$

The next function looks up the expression associated with a variable in a substitution by use of the *get* lens function.

lift-definition $\text{usubst-lookup} :: ('a, 'b) \text{psubst} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a, 'a) \text{uexpr} (\langle - \rangle_s)$
is $\lambda \sigma \ x \ b. \text{get}_x (\sigma \ b)$.

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

definition $\text{unrest-usubst} :: ('a \Rightarrow 'a) \Rightarrow 'a \text{usubst} \Rightarrow \text{bool}$
where $\text{unrest-usubst } x \ \sigma = (\forall \ \rho \ v. \sigma (\text{put}_x \ \rho \ v) = \text{put}_x (\sigma \ \rho) \ v)$

ad hoc-overloading

$\text{unrest } \text{unrest-usubst}$

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 $\text{cond-subst} :: 'a \text{usubst} \Rightarrow (\text{bool}, 'a) \text{uexpr} \Rightarrow 'a \text{usubst} \Rightarrow 'a \text{usubst} ((\exists - \triangleleft - \triangleright_s / -) [52, 0, 53] \ 52)$ **where**
 $\text{cond-subst } \sigma \ b \ \rho = (\lambda \ s. \text{if } \llbracket b \rrbracket_e s \text{ then } \sigma(s) \text{ else } \rho(s))$

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 $\text{par-subst} :: 'a \text{usubst} \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a \text{usubst} \Rightarrow 'a \text{usubst}$ **where**
 $\text{par-subst } \sigma_1 \ A \ B \ \sigma_2 = (\lambda \ s. (s \oplus_L (\sigma_1 \ s) \text{ on } A) \oplus_L (\sigma_2 \ s) \text{ on } B)$

7.2 Syntax translations

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

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

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

syntax

```

-smaplet :: [salpha, 'a] => smaplet      (- /↦s/ -)
          :: smaplet => smaplets        (-)
-SMaplets :: [smaplet, smaplets] => smaplets (-,/ -)
-SubstUpd :: ['m usubst, smaplets] => 'm usubst (-/'(-') [900,0] 900)
-Subst    :: smaplets => 'a → 'b        ((1[-]))
-psubst   :: [logic, svars, uexprs] ⇒ logic
-subst    :: logic ⇒ uexprs ⇒ salphas ⇒ logic (([-'/-]) [990,0,0] 991)
-uexp-l   :: logic ⇒ uexp (- [64] 64)
-uexprs   :: [uexp, uexprs] => uexprs (-,/ -)
          :: uexp => uexprs (-)
-salphas  :: [salpha, salphas] => salphas (-,/ -)
          :: salpha => salphas (-)
-par-subst :: logic ⇒ salpha ⇒ salpha ⇒ logic ⇒ logic (- [-]ₛ - [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
-psubst m (-salphas x xs) (-uexprs v vs) => -psubst (-psubst m x v) xs vs
-psubst m x v => CONST subst-upd m x v
-subst P v x <= CONST usubst (CONST subst-upd (CONST id) x v) P
-subst P v x <= -subst P (-spvar x) v
-par-subst σ₁ A B σ₂ == CONST par-subst σ₁ A B σ₂
-uexp-l e => e

```

Thus we can write things like $\sigma(x \mapsto_s v)$ to update a variable x in σ 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 $\text{subst-del} :: 'a \text{ usubst} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \text{ usubst}$ (**infix** $-_s$ 85) **where**
 $\text{subst-del } \sigma \ x = \sigma(x \mapsto_s \&x)$

7.3 Substitution Application Laws

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

method $\text{subst-tac} = (\text{simp add: usubst unrest})?$

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]: $\langle \text{id} \rangle_s \ x = \text{var } x$
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 $\text{weak-lens } x$

shows $\langle \sigma(x \mapsto_s v) \rangle_s x = v$
using *assms*
by (*simp add: subst-upd-uvar-def, transfer*) (*simp*)

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

Substitution update is idempotent.

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

lemma *usubst-upd-idem-sub* [*usubst*]:
assumes $x \subseteq_L y$ *mwb-lens y*
shows $\sigma(x \mapsto_s u, y \mapsto_s v) = \sigma(y \mapsto_s v)$
by (*simp add: subst-upd-uvar-def assms comp-def fun-eq-iff sublens-put-put*)

Substitution updates commute when the lenses are independent.

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

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

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

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

lemma *swap-usubst-inj*:
fixes $x y :: ('a \Rightarrow 'a)$
assumes *vwb-lens x vwb-lens y* $x \bowtie y$
shows *inj* $[x \mapsto_s \&y, y \mapsto_s \&x]$
proof (*rule injI*)
fix $b_1 :: 'a$ **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 $a: \text{put}_y (\text{put}_x b_1 (\llbracket \&y \rrbracket_e b_1)) (\llbracket \&x \rrbracket_e b_1) = \text{put}_y (\text{put}_x b_2 (\llbracket \&y \rrbracket_e b_2)) (\llbracket \&x \rrbracket_e b_2)$
by (*auto simp add: subst-upd-uvar-def*)
then have $(\forall a b c. \text{put}_x (\text{put}_y a b) c = \text{put}_y (\text{put}_x a c) b) \wedge$
 $(\forall a b. \text{get}_x (\text{put}_y a b) = \text{get}_x a) \wedge (\forall a b. \text{get}_y (\text{put}_x a b) = \text{get}_y a)$
by (*simp add: assms(3) lens-indep.lens-put-irr2 lens-indep-comm*)
then show $b_1 = b_2$
by (*metis a assms(1) assms(2) pr-var-def var.rep-eq vwb-lens.source-determination vwb-lens-def wb-lens-def weak-lens.put-get*)
qed

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

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

lemma *usubst-upd-comm-dash* [*usubst*]:
fixes $x :: ('a \implies 'b)$
shows $\sigma(\$x' \mapsto_s v, \$x \mapsto_s u) = \sigma(\$x \mapsto_s u, \$x' \mapsto_s v)$
using *out-in-indep usubst-upd-comm* **by** *blast*

lemma *subst-upd-lens-plus* [*usubst*]:
 $subst\text{-}upd\ \sigma\ (x +_L y) \ll(u,v)\gg = \sigma(y \mapsto_s \ll v \gg, x \mapsto_s \ll u \gg)$
by (*simp add: lens-defs uexpr-defs subst-upd-uvar-def, transfer, auto*)

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

lemma *subst-upd-out-lens-plus* [*usubst*]:
 $subst\text{-}upd\ \sigma\ (ovar\ (x +_L y)) \ll(u,v)\gg = \sigma(\$y' \mapsto_s \ll v \gg, \$x' \mapsto_s \ll u \gg)$
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 $mwb\text{-}lens\ x\ x \bowtie y$
shows $\langle \sigma(y \mapsto_s v) \rangle_s x = \langle \sigma \rangle_s x$
using *assms*
by (*simp add: subst-upd-uvar-def, transfer, simp*)

lemma *subst-upd-plus* [*usubst*]:
 $x \bowtie y \implies subst\text{-}upd\ s\ (x +_L y)\ e = s(x \mapsto_s \pi_1(e), y \mapsto_s \pi_2(e))$
by (*simp add: subst-upd-uvar-def lens-defs, transfer, auto simp add: fun-eq-iff prod.case-eq-if lens-indep-comm*)

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

lemma *usubst-apply-unrest* [*usubst*]:
 $\ll vwb\text{-}lens\ x; x \nmid \sigma \gg \implies \langle \sigma \rangle_s x = var\ x$
by (*simp add: unrest-usubst-def, transfer, auto simp add: fun-eq-iff,metis vwb-lens-wb wb-lens.get-put wb-lens-weak weak-lens.put-get*)

There follows various laws about deleting variables from a substitution.

lemma *subst-del-id* [*usubst*]:
 $vwb\text{-}lens\ x \implies id -_s x = id$
by (*simp add: subst-del-def subst-upd-uvar-def pr-var-def, transfer, auto*)

lemma *subst-del-upd-same* [*usubst*]:

mw-lens $x \implies \sigma(x \mapsto_s v) -_s x = \sigma -_s x$
by (*simp add: subst-del-def subst-upd-uvar-def*)

lemma *subst-del-upd-diff* [*usubst*]:
 $x \boxtimes y \implies \sigma(y \mapsto_s v) -_s x = (\sigma -_s x)(y \mapsto_s v)$
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*]: $x \nmid P \implies \sigma(x \mapsto_s v) \dagger P = \sigma \dagger P$
by (*simp add: subst-upd-uvar-def, transfer, auto*)

lemma *subst-unrest-sublens* [*usubst*]: $\llbracket a \nmid P; x \subseteq_L a \rrbracket \implies \sigma(x \mapsto_s v) \dagger P = \sigma \dagger P$
by (*simp add: subst-upd-uvar-def, transfer, auto simp add: fun-eq-iff, metis (no-types, lifting) lens.select-convs(2) lens-comp-def sublens-def*)

lemma *subst-unrest-2* [*usubst*]:
fixes $P :: ('a, 'α) \text{uexpr}$
assumes $x \nmid P \ x \boxtimes y$
shows $\sigma(x \mapsto_s u, y \mapsto_s v) \dagger P = \sigma(y \mapsto_s v) \dagger P$
using *assms*
by (*simp add: subst-upd-uvar-def, transfer, auto, metis lens-indep.lens-put-comm*)

lemma *subst-unrest-3* [*usubst*]:
fixes $P :: ('a, 'α) \text{uexpr}$
assumes $x \nmid P \ x \boxtimes y \ x \boxtimes z$
shows $\sigma(x \mapsto_s u, y \mapsto_s v, z \mapsto_s w) \dagger P = \sigma(y \mapsto_s v, z \mapsto_s w) \dagger P$
using *assms*
by (*simp add: subst-upd-uvar-def, transfer, auto, metis (no-types, hide-lams) lens-indep-comm*)

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

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

lemma *subst-compose-upd* [*usubst*]: $x \nmid \sigma \implies \sigma \circ \varrho(x \mapsto_s v) = (\sigma \circ \varrho)(x \mapsto_s v)$
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*)

7.4 Substitution laws

We now prove the key laws that show how a substitution should be performed for every expression operator, including the core function operators, literals, variables, and the arithmetic

operators. They are all added to the *usubst* theorem attribute so that we can apply them using the substitution tactic.

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

lemma *ident-subst* [*usubst*]: $(\lambda a. a) \uparrow p = p$
by (*simp add: subst.rep-eq uexpr-eq-iff*)

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

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

lemma *usubst-ulambda* [*usubst*]: $\sigma \uparrow (\lambda x. P(x)) = (\lambda x. \sigma \uparrow P(x))$
by (*transfer*, *simp*)

lemma *unrest-usubst-del* [*unrest*]: $\llbracket \text{vwb-lens } x; x \# (\langle \sigma \rangle_s x); x \# \sigma -_s x \rrbracket \implies x \# (\sigma \uparrow P)$
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*]: $\sigma \uparrow \text{uop } f v = \text{uop } f (\sigma \uparrow v)$
by (*transfer*, *simp*)

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

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

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

lemma *subst-case-prod* [*usubst*]:
fixes $P :: 'i \Rightarrow 'j \Rightarrow ('a, 'a) \text{ uexpr}$
shows $\sigma \uparrow \text{case-prod } (\lambda x y. P x y) v = \text{case-prod } (\lambda x y. \sigma \uparrow P x y) v$
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*]: $\sigma \uparrow (x * y) = \sigma \uparrow x * \sigma \uparrow y$
by (*simp add: times-uexpr-def subst-bop*)

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

lemma *subst-div* [*usubst*]: $\sigma \uparrow (x \text{ div } y) = \sigma \uparrow x \text{ div } \sigma \uparrow y$
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*]: $\sigma \dagger (-x) = -(\sigma \dagger x)$
by (*simp add: uminus-uepr-def subst-uop*)

lemma *usubst-sgn* [*usubst*]: $\sigma \dagger \text{sgn } x = \text{sgn } (\sigma \dagger x)$
by (*simp add: sgn-uepr-def subst-uop*)

lemma *usubst-abs* [*usubst*]: $\sigma \dagger \text{abs } x = \text{abs } (\sigma \dagger x)$
by (*simp add: abs-uepr-def subst-uop*)

lemma *subst-zero* [*usubst*]: $\sigma \dagger 0 = 0$
by (*simp add: zero-uepr-def subst-lit*)

lemma *subst-one* [*usubst*]: $\sigma \dagger 1 = 1$
by (*simp add: one-uepr-def subst-lit*)

lemma *subst-eq-upred* [*usubst*]: $\sigma \dagger (x =_u y) = (\sigma \dagger x =_u \sigma \dagger y)$
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*]: $\sigma \dagger \varrho \dagger e = (\varrho \circ \sigma) \dagger e$
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 $\varrho(x \mapsto_s v) \circ \sigma = (\varrho \circ \sigma)(x \mapsto_s \sigma \dagger v)$
by (*rule ext, simp add: uepr-defs subst-upd-uvar-def, transfer, simp*)

lemma *subst-singleton*:
fixes $x :: ('a \Rightarrow 'a)$
assumes $x \# \sigma$
shows $\sigma(x \mapsto_s v) \dagger P = (\sigma \dagger P)[[v/x]]$
using *assms*
by (*simp add: usubst*)

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

7.5 Ordering substitutions

A simplification procedure to reorder substitutions maplets lexicographically by variable syntax

simproc-setup *subst-order* (*subst-upd-uvar* (*subst-upd-uvar* σ x u) y v) =
 \ll ($fn - \Rightarrow fn \text{ ctx} \Rightarrow fn \text{ ct} \Rightarrow$
 $\text{case } (Thm.term-of \text{ ct}) \text{ of}$
 $\text{Const } (utp_subst.subst_upd_uvar, -) \$ (Const (utp_subst.subst_upd_uvar, -) \$ s \$ x \$ u) \$ y \$ v$
 $\Rightarrow \text{if } (YXML.content-of (Syntax.string-of-term \text{ ctx } x) > YXML.content-of (Syntax.string-of-term$
 $\text{ ctx } y))$
 $\text{then SOME (mk-meta-eq @\{thm usubst-upd-comm\})}$
 $\text{else NONE} \mid$
 $- \Rightarrow \text{NONE})$
 \gg

7.6 Unrestriction laws

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

lemma *unrest-usubst-single* [*unrest*]:
 $\llbracket \text{mwb-lens } x; x \# v \rrbracket \implies x \# P[v/x]$
by (*transfer*, *auto simp add: subst-upd-uvar-def unrest-uepr-def*)

lemma *unrest-usubst-id* [*unrest*]:
 $\text{mwb-lens } x \implies x \# \text{id}$
by (*simp add: unrest-usubst-def*)

lemma *unrest-usubst-upd* [*unrest*]:
 $\llbracket x \bowtie y; x \# \sigma; x \# v \rrbracket \implies x \# \sigma(y \mapsto_s v)$
by (*simp add: subst-upd-uvar-def unrest-usubst-def unrest-uepr.rep-eq lens-indep-comm*)

lemma *unrest-subst* [*unrest*]:
 $\llbracket x \# P; x \# \sigma \rrbracket \implies x \# (\sigma \dagger P)$
by (*transfer*, *simp add: unrest-usubst-def*)

7.7 Conditional Substitution Laws

lemma *usubst-cond-upd-1* [*usubst*]:
 $\sigma(x \mapsto_s u) \triangleleft b \triangleright_s \varrho(x \mapsto_s v) = (\sigma \triangleleft b \triangleright_s \varrho)(x \mapsto_s u \triangleleft b \triangleright v)$
by (*simp add: cond-subst-def subst-upd-uvar-def uepr-defs, transfer, auto*)

lemma *usubst-cond-upd-2* [*usubst*]:
 $\llbracket \text{vwb-lens } x; x \# \varrho \rrbracket \implies \sigma(x \mapsto_s u) \triangleleft b \triangleright_s \varrho = (\sigma \triangleleft b \triangleright_s \varrho)(x \mapsto_s u \triangleleft b \triangleright \&x)$
by (*simp add: cond-subst-def subst-upd-uvar-def unrest-usubst-def uepr-defs, transfer*)
(metis (full-types, hide-lams) id-apply pr-var-def subst-upd-uvar-def usubst-upd-pr-var-id var.rep-eq)

lemma *usubst-cond-upd-3* [*usubst*]:
 $\llbracket \text{vwb-lens } x; x \# \sigma \rrbracket \implies \sigma \triangleleft b \triangleright_s \varrho(x \mapsto_s v) = (\sigma \triangleleft b \triangleright_s \varrho)(x \mapsto_s \&x \triangleleft b \triangleright v)$
by (*simp add: cond-subst-def subst-upd-uvar-def unrest-usubst-def uepr-defs, transfer*)
(metis (full-types, hide-lams) id-apply pr-var-def subst-upd-uvar-def usubst-upd-pr-var-id var.rep-eq)

lemma *usubst-cond-id* [*usubst*]:
 $\sigma \triangleleft b \triangleright_s \sigma = \sigma$
by (*auto simp add: cond-subst-def*)

7.8 Parallel Substitution Laws

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

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

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

lemma *par-subst-comm*:

$\llbracket A \bowtie B \rrbracket \implies \sigma [A|B]_s \varrho = \varrho [B|A]_s \sigma$
by (*simp add: par-subst-def lens-override-def lens-indep-comm*)

lemma *par-subst-upd-left-in* [*usubst*]:
 $\llbracket \text{vwb-lens } A; A \bowtie B; x \subseteq_L A \rrbracket \implies \sigma(x \mapsto_s v) [A|B]_s \varrho = (\sigma [A|B]_s \varrho)(x \mapsto_s v)$
by (*simp add: par-subst-def subst-upd-uvar-def lens-override-put-right-in*)
(simp add: lens-indep-comm lens-override-def sublens-pres-indep)

lemma *par-subst-upd-left-out* [*usubst*]:
 $\llbracket \text{vwb-lens } A; x \bowtie A \rrbracket \implies \sigma(x \mapsto_s v) [A|B]_s \varrho = (\sigma [A|B]_s \varrho)$
by (*simp add: par-subst-def subst-upd-uvar-def lens-override-put-right-out*)

lemma *par-subst-upd-right-in* [*usubst*]:
 $\llbracket \text{vwb-lens } B; A \bowtie B; x \subseteq_L B \rrbracket \implies \sigma [A|B]_s \varrho(x \mapsto_s v) = (\sigma [A|B]_s \varrho)(x \mapsto_s v)$
using *lens-indep-sym par-subst-comm par-subst-upd-left-in* **by** *fastforce*

lemma *par-subst-upd-right-out* [*usubst*]:
 $\llbracket \text{vwb-lens } B; A \bowtie B; x \bowtie B \rrbracket \implies \sigma [A|B]_s \varrho(x \mapsto_s v) = (\sigma [A|B]_s \varrho)$
by (*simp add: par-subst-comm par-subst-upd-left-out*)

end

8 UTP Tactics

```
theory utp-tactics
imports
  utp-expr utp-unrest utp-usedby
keywords update-ueexpr-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.

8.1 Theorem Attributes

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

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

8.2 Generic Methods

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

Generic Predicate Tactics

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

Generic Relational Tactics

```
method gen-rel-tac methods transfer-tac interp-tac prove-tac = (
  ((unfold upred-defs urel-defs) [1])?;
```

```

(transfer-tac),
(simp add: fun-eq-iff relcomp-unfold OO-def
 lens-defs upred-defs alpha-splits Product-Type.split-beta)?,
(interp-tac)?);
(prove-tac)

```

8.3 Transfer Tactics

Next, we define the component tactics used for transfer.

8.3.1 Robust Transfer

Robust transfer uses the transfer method of the lifting package.

```
method slow-uexpr-transfer = (transfer)
```

8.3.2 Faster Transfer

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

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

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

ML-file *uexpr-rep-eq.ML*

```

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

```

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

```

ML (
  Outer-Syntax.command @{command-keyword update-uexpr-rep-eq-thms}
    reread and update content of the uexpr-rep-eq-thms attribute
    (Scan.succeed (Toplevel.theory uexpr-rep-eq.read-uexpr-rep-eq-thms));
)

```

update-uexpr-rep-eq-thms — Read *uexpr-rep-eq-thms* here.

Lastly, we require several named-theorem attributes to record the manual transfer laws and extra simplifications, so that the user can dynamically extend them in child theories.

named-theorems *ueexpr-transfer-laws ueexpr transfer laws*

declare *ueexpr-eq-iff* [*ueexpr-transfer-laws*]

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

declare *unrest-ueexpr.rep-eq* [*ueexpr-transfer-extra*]

usedBy-ueexpr.rep-eq [*ueexpr-transfer-extra*]

utp-expr.numeral-ueexpr.rep-eq [*ueexpr-transfer-extra*]

utp-expr.less-eq-ueexpr.rep-eq [*ueexpr-transfer-extra*]

Abs-ueexpr-inverse [*simplified, ueexpr-transfer-extra*]

Rep-ueexpr-inverse [*ueexpr-transfer-extra*]

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

method *fast-ueexpr-transfer* =

(*simp add: ueexpr-transfer-laws ueexpr-rep-eq-thms ueexpr-transfer-extra*)

8.4 Interpretation

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

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

8.5 User Tactics

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

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

method *utp-auto-tac* = ((*clarsimp*)?; *auto*)

method *utp-blast-tac* = ((*clarsimp*)?; *blast*)

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

ML-file *utp-tactics.ML*

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

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

```
method-setup rel-simp = ⟨
```

```

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

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

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

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

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

```

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

```

method rel-simp'
  uses simp
  = (simp add: upred-defs urel-defs lens-defs prod.case-eq-if relcomp-unfold uexpr-transfer-laws uexpr-transfer-extra
    uexpr-rep-eq-thms simp)

method rel-auto'
  uses simp intro elim dest
  = (auto intro: intro elim: elim dest: dest simp add: upred-defs urel-defs lens-defs relcomp-unfold
    uexpr-transfer-laws uexpr-transfer-extra uexpr-rep-eq-thms simp)

method rel-blast'
  uses simp intro elim dest
  = (rel-simp' simp: simp, blast intro: intro elim: elim dest: dest)

```

end

9 Meta-level Substitution

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

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

lift-definition $msubst :: ('b \Rightarrow ('a, 'a) uexpr) \Rightarrow ('b, 'a) uexpr \Rightarrow ('a, 'a) uexpr$
is $\lambda F v b. F (v b) b$.

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

syntax

$-msubst \quad :: \text{logic} \Rightarrow \text{pttrn} \Rightarrow \text{logic} \Rightarrow \text{logic} \ ((-\!\!\rightarrow\!-) \ [990,0,0] \ 991)$

translations

$-msubst \ P \ x \ v == \text{CONST } msubst \ (\lambda x. P) \ v$

lemma *msubst-lit* [usubst]: $\ll x \gg \ll x \rightarrow v \gg = v$
by (*pred-auto*)

lemma *msubst-const* [usubst]: $P \ll x \rightarrow v \gg = P$
by (*pred-auto*)

lemma *msubst-pair* [usubst]: $(P \ x \ y) \ll (x, y) \rightarrow (e, f)_u \gg = (P \ x \ y) \ll x \rightarrow e \gg \ll y \rightarrow f \gg$
by (*rel-auto*)

lemma *msubst-lit-2-1* [usubst]: $\ll x \gg \ll (x, y) \rightarrow (u, v)_u \gg = u$
by (*pred-auto*)

lemma *msubst-lit-2-2* [usubst]: $\ll y \gg \ll (x, y) \rightarrow (u, v)_u \gg = v$
by (*pred-auto*)

lemma *msubst-lit'* [usubst]: $\ll y \gg \ll x \rightarrow v \gg = \ll y \gg$
by (*pred-auto*)

lemma *msubst-lit'-2* [usubst]: $\ll z \gg \ll (x, y) \rightarrow v \gg = \ll z \gg$
by (*pred-auto*)

lemma *msubst-uop* [usubst]: $(uop \ f \ (v \ x)) \ll x \rightarrow u \gg = uop \ f \ ((v \ x) \ll x \rightarrow u \gg)$
by (*rel-auto*)

lemma *msubst-uop-2* [usubst]: $(uop \ f \ (v \ x \ y)) \ll (x, y) \rightarrow u \gg = uop \ f \ ((v \ x \ y) \ll (x, y) \rightarrow u \gg)$
by (*pred-simp, pred-simp*)

lemma *msubst-bop* [usubst]: $(bop \ f \ (v \ x) \ (w \ x)) \ll x \rightarrow u \gg = bop \ f \ ((v \ x) \ll x \rightarrow u \gg) \ ((w \ x) \ll x \rightarrow u \gg)$
by (*rel-auto*)

lemma *msubst-bop-2* [usubst]: $(bop \ f \ (v \ x \ y) \ (w \ x \ y)) \ll (x, y) \rightarrow u \gg = bop \ f \ ((v \ x \ y) \ll (x, y) \rightarrow u \gg) \ ((w \ x \ y) \ll (x, y) \rightarrow u \gg)$
by (*pred-simp, pred-simp*)

```

lemma msubst-var [usubst]:
  (utp-expr.var x) $\llbracket y \rightarrow u \rrbracket$  = utp-expr.var x
  by (pred-simp)

lemma msubst-var-2 [usubst]:
  (utp-expr.var x) $\llbracket (y, z) \rightarrow u \rrbracket$  = utp-expr.var x
  by (pred-simp)+

lemma msubst-unrest [unrest]:  $\llbracket \bigwedge v. x \# P(v); x \# k \rrbracket \Longrightarrow x \# P(v) \llbracket v \rightarrow k \rrbracket$ 
  by (pred-auto)

end

```

10 Alphabetised Predicates

```

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

```

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

10.1 Predicate type and syntax

An alphabetised predicate is simply a boolean valued expression.

```

type-synonym 'α upred = (bool, 'α) uexpr

```

```

translations
  (type) 'α upred <= (type) (bool, 'α) 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  $\wedge$  35) and
  disj (infixr  $\vee$  30) and
  Not ( $\neg$  - [40] 40)

consts
  uttrue :: 'a (true)
  ufalse :: 'a (false)
  uconj :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixr  $\wedge$  35)
  udisj :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixr  $\vee$  30)
  uimpl :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixr  $\Rightarrow$  25)
  uiff :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixr  $\Leftrightarrow$  25)
  unot :: 'a  $\Rightarrow$  'a ( $\neg$  - [40] 40)

```


$uex :: ('a \Rightarrow 'α) \Rightarrow 'p \Rightarrow 'p$
 $uall :: ('a \Rightarrow 'α) \Rightarrow 'p \Rightarrow 'p$
 $ushEx :: ['a \Rightarrow 'p] \Rightarrow 'p$
 $ushAll :: ['a \Rightarrow 'p] \Rightarrow 'p$

ad hoc-overloading

$uconj$ *conj* **and**
 $udisj$ *disj* **and**
 $unot$ *Not*

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

nonterminal *idt-list*

syntax

$-idt-el :: idt \Rightarrow idt-list \ (-)$
 $-idt-list :: idt \Rightarrow idt-list \Rightarrow idt-list \ ((-, / -) [0, 1])$
 $-uex :: salpha \Rightarrow logic \Rightarrow logic \ (\exists \ - \ - \ [0, 10] \ 10)$
 $-uall :: salpha \Rightarrow logic \Rightarrow logic \ (\forall \ - \ - \ [0, 10] \ 10)$
 $-ushEx :: pttrn \Rightarrow logic \Rightarrow logic \ (\exists \ - \ - \ [0, 10] \ 10)$
 $-ushAll :: pttrn \Rightarrow logic \Rightarrow logic \ (\forall \ - \ - \ [0, 10] \ 10)$
 $-ushBEx :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \ (\exists \ - \in \ - \ - \ [0, 0, 10] \ 10)$
 $-ushBAll :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \ (\forall \ - \in \ - \ - \ [0, 0, 10] \ 10)$
 $-ushGAll :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \ (\forall \ - \mid \ - \ - \ [0, 0, 10] \ 10)$
 $-ushGtAll :: idt \Rightarrow logic \Rightarrow logic \Rightarrow logic \ (\forall \ - \> \ - \ - \ [0, 0, 10] \ 10)$
 $-ushLtAll :: idt \Rightarrow logic \Rightarrow logic \Rightarrow logic \ (\forall \ - \< \ - \ - \ [0, 0, 10] \ 10)$
 $-uvar-res :: logic \Rightarrow salpha \Rightarrow logic \ (\mathbf{infixl} \ \vdash_v \ 90)$

translations

$-uex \ x \ P \quad == \ \mathbf{CONST} \ uex \ x \ P$
 $-uex \ (-salphaset \ (-salphamk \ (x +_L y))) \ P \leq -uex \ (x +_L y) \ P$
 $-uall \ x \ P \quad == \ \mathbf{CONST} \ uall \ x \ P$
 $-uall \ (-salphaset \ (-salphamk \ (x +_L y))) \ P \leq -uall \ (x +_L y) \ P$
 $-ushEx \ x \ P \quad == \ \mathbf{CONST} \ ushEx \ (\lambda x. P)$
 $\exists \ x \in A \cdot P \quad ==> \exists \ x \cdot \ll x \gg \in_u A \wedge P$
 $-ushAll \ x \ P \quad == \ \mathbf{CONST} \ ushAll \ (\lambda x. P)$
 $\forall \ x \in A \cdot P \quad ==> \forall \ x \cdot \ll x \gg \in_u A \Rightarrow P$
 $\forall \ x \mid P \cdot Q \quad ==> \forall \ x \cdot P \Rightarrow Q$
 $\forall \ x > y \cdot P \quad ==> \forall \ x \cdot \ll x \gg >_u y \Rightarrow P$
 $\forall \ x < y \cdot P \quad ==> \forall \ x \cdot \ll x \gg <_u y \Rightarrow P$

10.2 Predicate operators

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

class *refine* = *order*

abbreviation *refineBy* :: $'a :: refine \Rightarrow 'a \Rightarrow bool$ (**infix** \sqsubseteq 50) **where**

$P \sqsubseteq Q \equiv \text{less-eq } Q \ P$

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

purge-notation *Lattices.inf* (**infixl** \sqcap 70)
notation *Lattices.inf* (**infixl** \sqcup 70)
purge-notation *Lattices.sup* (**infixl** \sqcup 65)
notation *Lattices.sup* (**infixl** \sqcap 65)

purge-notation *Inf* (\sqcap - [900] 900)
notation *Inf* (\sqcup - [900] 900)
purge-notation *Sup* (\sqcup - [900] 900)
notation *Sup* (\sqcap - [900] 900)

purge-notation *Orderings.bot* (\perp)
notation *Orderings.bot* (\top)
purge-notation *Orderings.top* (\top)
notation *Orderings.top* (\perp)

purge-syntax

-INF1 :: *pttrns* \Rightarrow 'b \Rightarrow 'b (($\exists \sqcap$ -./ -) [0, 10] 10)
-INF :: *pttrn* \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b (($\exists \sqcap$ - \in -./ -) [0, 0, 10] 10)
-SUP1 :: *pttrns* \Rightarrow 'b \Rightarrow 'b (($\exists \sqcup$ -./ -) [0, 10] 10)
-SUP :: *pttrn* \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b (($\exists \sqcup$ - \in -./ -) [0, 0, 10] 10)

syntax

-INF1 :: *pttrns* \Rightarrow 'b \Rightarrow 'b (($\exists \sqcup$ -./ -) [0, 10] 10)
-INF :: *pttrn* \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b (($\exists \sqcup$ - \in -./ -) [0, 0, 10] 10)
-SUP1 :: *pttrns* \Rightarrow 'b \Rightarrow 'b (($\exists \sqcap$ -./ -) [0, 10] 10)
-SUP :: *pttrn* \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b (($\exists \sqcap$ - \in -./ -) [0, 0, 10] 10)

We trivially instantiate our refinement class

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

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

theorem *upred-ref-iff* [*uexpr-transfer-laws*]:

$(P \sqsubseteq Q) = (\forall b. \llbracket Q \rrbracket_e b \longrightarrow \llbracket P \rrbracket_e b)$

apply (*transfer*)

apply (*clarsimp*)

done

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

instantiation *uexpr* :: (*lattice*, *type*) *lattice*

begin

lift-definition *sup-uexpr* :: ('a, 'b) *uexpr* \Rightarrow ('a, 'b) *uexpr* \Rightarrow ('a, 'b) *uexpr*

is $\lambda P \ Q \ A. \text{Lattices.sup } (P \ A) \ (Q \ A) .$

lift-definition *inf-uexpr* :: ('a, 'b) *uexpr* \Rightarrow ('a, 'b) *uexpr* \Rightarrow ('a, 'b) *uexpr*

is $\lambda P \ Q \ A. \text{Lattices.inf } (P \ A) \ (Q \ A) .$

instance

by (*intro-classes*) (*transfer*, *auto*)+

end

```

instantiation uexpr :: (bounded-lattice, type) bounded-lattice
begin
  lift-definition bot-uexpr :: ('a, 'b) uexpr is  $\lambda A. \text{Orderings.bot}$  .
  lift-definition top-uexpr :: ('a, 'b) uexpr is  $\lambda A. \text{Orderings.top}$  .
instance
  by (intro-classes) (transfer, auto)+
end

```

```

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

```

```

lemma bot-uexpr-rep-eq [simp]:
   $\llbracket \text{Orderings.top} \rrbracket_e b = \text{True}$ 
  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  $\Rightarrow$  ('a, 'b) uexpr
  is  $\lambda PS A. \text{INF } P:PS. P(A)$  .
  lift-definition Sup-uexpr :: ('a, 'b) uexpr set  $\Rightarrow$  ('a, 'b) uexpr
  is  $\lambda PS A. \text{SUP } P:PS. P(A)$  .
instance
  by (intro-classes)
  (transfer, auto intro: INF-lower SUP-upper simp add: INF-greatest SUP-least)+
end

```

```

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

```

```

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

```

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

```

syntax
  -mu :: pttrn  $\Rightarrow$  logic  $\Rightarrow$  logic ( $\mu$  - - [0, 10] 10)
  -nu :: pttrn  $\Rightarrow$  logic  $\Rightarrow$  logic ( $\nu$  - - [0, 10] 10)

```

```

notation gfp ( $\mu$ )
notation lfp ( $\nu$ )

```

```

translations
   $\nu X \cdot P == \text{CONST lfp } (\lambda X. P)$ 
   $\mu X \cdot P == \text{CONST gfp } (\lambda X. P)$ 

```

With the lattice operators defined, we can proceed to give definitions for the standard predicate operators in terms of them.

definition $true-upred = (Orderings.top :: 'α upred)$

definition $false-upred = (Orderings.bot :: 'α upred)$

definition $conj-upred = (Lattices.inf :: 'α upred ⇒ 'α upred ⇒ 'α upred)$

definition $disj-upred = (Lattices.sup :: 'α upred ⇒ 'α upred ⇒ 'α upred)$

definition $not-upred = (uminus :: 'α upred ⇒ 'α upred)$

definition $diff-upred = (minus :: 'α upred ⇒ 'α upred ⇒ 'α upred)$

abbreviation $Conj-upred :: 'α upred set ⇒ 'α upred (\bigwedge - [900] 900)$ **where**
 $\bigwedge A \equiv \bigcap A$

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

notation

$conj-upred$ (**infixr** \wedge_p 35) **and**

$disj-upred$ (**infixr** \vee_p 30)

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

lift-definition $UINF :: ('a ⇒ 'α upred) ⇒ ('a ⇒ ('b::complete-lattice, 'α) uexpr) ⇒ ('b, 'α) uexpr$
is $\lambda P F b. Sup \{ \llbracket F x \rrbracket_e b \mid x. \llbracket P x \rrbracket_e b \} .$

lift-definition $USUP :: ('a ⇒ 'α upred) ⇒ ('a ⇒ ('b::complete-lattice, 'α) uexpr) ⇒ ('b, 'α) uexpr$
is $\lambda P F b. Inf \{ \llbracket F x \rrbracket_e b \mid x. \llbracket P x \rrbracket_e b \} .$

syntax

$-USup \quad :: pttrn \Rightarrow logic \Rightarrow logic \quad (\bigwedge - \cdot - [0, 10] 10)$
 $-USup \quad :: pttrn \Rightarrow logic \Rightarrow logic \quad (\bigcap - \cdot - [0, 10] 10)$
 $-USup-mem :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \quad (\bigwedge - \in \cdot - [0, 10] 10)$
 $-USup-mem :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \quad (\bigcap - \in \cdot - [0, 10] 10)$
 $-USUP \quad :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \quad (\bigwedge - | \cdot - [0, 0, 10] 10)$
 $-USUP \quad :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \quad (\bigcap - | \cdot - [0, 0, 10] 10)$
 $-UInf \quad :: pttrn \Rightarrow logic \Rightarrow logic \quad (\bigvee - \cdot - [0, 10] 10)$
 $-UInf \quad :: pttrn \Rightarrow logic \Rightarrow logic \quad (\bigcup - \cdot - [0, 10] 10)$
 $-UInf-mem :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \quad (\bigvee - \in \cdot - [0, 10] 10)$
 $-UInf-mem :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \quad (\bigcup - \in \cdot - [0, 10] 10)$
 $-UINF \quad :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \quad (\bigvee - | \cdot - [0, 10] 10)$
 $-UINF \quad :: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \quad (\bigcup - | \cdot - [0, 10] 10)$

translations

$\bigcap x \mid P \cdot F \Rightarrow CONST UINF (\lambda x. P) (\lambda x. F)$
 $\bigcap x \cdot F == \bigcap x \mid true \cdot F$
 $\bigcap x \cdot F == \bigcap x \mid true \cdot F$
 $\bigcap x \in A \cdot F \Rightarrow \bigcap x \mid \ll x \gg \in_u \ll A \gg \cdot F$
 $\bigcap x \in A \cdot F \Leftarrow \bigcap x \mid \ll y \gg \in_u \ll A \gg \cdot F$
 $\bigcap x \mid P \cdot F \Leftarrow CONST UINF (\lambda y. P) (\lambda x. F)$
 $\bigcap x \mid P \cdot F(x) \Leftarrow CONST UINF (\lambda x. P) F$
 $\bigcap x \mid P \cdot F \Rightarrow CONST USUP (\lambda x. P) (\lambda x. F)$
 $\bigcap x \cdot F == \bigcap x \mid true \cdot F$
 $\bigcap x \in A \cdot F \Rightarrow \bigcap x \mid \ll x \gg \in_u \ll A \gg \cdot F$
 $\bigcap x \in A \cdot F \Leftarrow \bigcap x \mid \ll y \gg \in_u \ll A \gg \cdot F$
 $\bigcap x \mid P \cdot F \Leftarrow CONST USUP (\lambda y. P) (\lambda x. F)$

$\sqcup \mid x \mid P \cdot F(x) \leq \text{CONST USUP } (\lambda x. P) F$

We also define the other predicate operators

lift-definition *impl* :: $'\alpha \text{ upred} \Rightarrow '\alpha \text{ upred} \Rightarrow '\alpha \text{ upred}$ **is**
 $\lambda P Q A. P A \longrightarrow Q A$.

lift-definition *iff-upred* :: $'\alpha \text{ upred} \Rightarrow '\alpha \text{ upred} \Rightarrow '\alpha \text{ upred}$ **is**
 $\lambda P Q A. P A \longleftrightarrow Q A$.

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

lift-definition *shEx* :: $['\beta \Rightarrow '\alpha \text{ upred}] \Rightarrow '\alpha \text{ upred}$ **is**
 $\lambda P A. \exists x. (P x) A$.

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

lift-definition *shAll* :: $['\beta \Rightarrow '\alpha \text{ upred}] \Rightarrow '\alpha \text{ upred}$ **is**
 $\lambda P A. \forall x. (P x) A$.

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

lift-definition *var-res* :: $'\alpha \text{ upred} \Rightarrow ('a \Longrightarrow '\alpha) \Rightarrow '\alpha \text{ upred}$ **is**
 $\lambda P x b. \exists b'. P(b' \oplus_L b \text{ on } x)$.

translations

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

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

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

lift-definition *taut* :: $'\alpha \text{ upred} \Rightarrow \text{bool } ('-')$
is $\lambda P. \forall A. P A$.

The following function extracts the characteristic set of a predicate

lift-definition *upred-set* :: $'a \text{ upred} \Rightarrow 'a \text{ set } ([\cdot]_p)$ **is**
 $\lambda P. \text{Collect } P$.

Configuration for UTP tactics

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

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

ad hoc overloading

uttrue true-upred **and**
ufalse false-upred **and**
unot not-upred **and**
uconj conj-upred **and**
udisj disj-upred **and**
uimpl impl **and**
wiff iff-upred **and**

$uex\ ex\ \mathbf{and}$
 $uall\ all\ \mathbf{and}$
 $ushEx\ shEx\ \mathbf{and}$
 $ushAll\ shAll$

syntax

$-uneq \quad ::\ logic \Rightarrow logic \Rightarrow logic\ (\mathbf{infixl}\ \neq_u\ 50)$
 $-unmem \quad ::\ ('a,\ ' \alpha)\ uexpr \Rightarrow ('a\ set,\ ' \alpha)\ uexpr \Rightarrow (bool,\ ' \alpha)\ uexpr\ (\mathbf{infix}\ \notin_u\ 50)$

translations

$x \neq_u y == CONST\ unot\ (x =_u y)$
 $x \notin_u A == CONST\ unot\ (CONST\ bop\ (\in)\ x\ A)$

$\mathbf{declare}\ true\text{-}upred\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ false\text{-}upred\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ conj\text{-}upred\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ disj\text{-}upred\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ not\text{-}upred\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ diff\text{-}upred\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ subst\text{-}upd\text{-}uvar\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ cond\text{-}subst\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ par\text{-}subst\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ subst\text{-}del\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ unrest\text{-}usubst\text{-}def\ [upred\text{-}defs]$
 $\mathbf{declare}\ uexpr\text{-}defs\ [upred\text{-}defs]$

$\mathbf{lemma}\ true\text{-}alt\text{-}def: true = \ll True \gg$
 $\mathbf{by}\ (pred\text{-}auto)$

$\mathbf{lemma}\ false\text{-}alt\text{-}def: false = \ll False \gg$
 $\mathbf{by}\ (pred\text{-}auto)$

$\mathbf{declare}\ true\text{-}alt\text{-}def[THEN\ sym,simp]$
 $\mathbf{declare}\ false\text{-}alt\text{-}def[THEN\ sym,simp]$

10.3 Unrestriction Laws

$\mathbf{lemma}\ unrest\text{-}allE:$
 $\ll \Sigma\ \# P; P = true \Longrightarrow Q; P = false \Longrightarrow Q \gg \Longrightarrow Q$
 $\mathbf{by}\ (pred\text{-}auto)$

$\mathbf{lemma}\ unrest\text{-}true\ [unrest]: x\ \# true$
 $\mathbf{by}\ (pred\text{-}auto)$

$\mathbf{lemma}\ unrest\text{-}false\ [unrest]: x\ \# false$
 $\mathbf{by}\ (pred\text{-}auto)$

$\mathbf{lemma}\ unrest\text{-}conj\ [unrest]: \ll x\ \# (P :: ' \alpha\ upred); x\ \# Q \gg \Longrightarrow x\ \# P \wedge Q$
 $\mathbf{by}\ (pred\text{-}auto)$

$\mathbf{lemma}\ unrest\text{-}disj\ [unrest]: \ll x\ \# (P :: ' \alpha\ upred); x\ \# Q \gg \Longrightarrow x\ \# P \vee Q$
 $\mathbf{by}\ (pred\text{-}auto)$

$\mathbf{lemma}\ unrest\text{-}UINF\ [unrest]:$
 $\ll (\bigwedge i. x\ \# P(i)); (\bigwedge i. x\ \# Q(i)) \gg \Longrightarrow x\ \# (\bigcap i\ |\ P(i) \cdot Q(i))$
 $\mathbf{by}\ (pred\text{-}auto)$

lemma *unrest-USUP* [*unrest*]:
 $\llbracket (\bigwedge i. x \# P(i)); (\bigwedge i. x \# Q(i)) \rrbracket \Longrightarrow x \# (\bigsqcup i \mid P(i) \cdot Q(i))$
by (*pred-auto*)

lemma *unrest-UINF-mem* [*unrest*]:
 $\llbracket (\bigwedge i. i \in A \Longrightarrow x \# P(i)) \rrbracket \Longrightarrow x \# (\prod_{i \in A} P(i))$
by (*pred-simp*, *metis*)

lemma *unrest-USUP-mem* [*unrest*]:
 $\llbracket (\bigwedge i. i \in A \Longrightarrow x \# P(i)) \rrbracket \Longrightarrow x \# (\bigsqcup_{i \in A} P(i))$
by (*pred-simp*, *metis*)

lemma *unrest-impl* [*unrest*]: $\llbracket x \# P; x \# Q \rrbracket \Longrightarrow x \# P \Rightarrow Q$
by (*pred-auto*)

lemma *unrest-iff* [*unrest*]: $\llbracket x \# P; x \# Q \rrbracket \Longrightarrow x \# P \Leftrightarrow Q$
by (*pred-auto*)

lemma *unrest-not* [*unrest*]: $x \# (P :: 'a \text{ upred}) \Longrightarrow x \# (\neg P)$
by (*pred-auto*)

The sublens proviso can be thought of as membership below.

lemma *unrest-ex-in* [*unrest*]:
 $\llbracket \text{mwb-lens } y; x \subseteq_L y \rrbracket \Longrightarrow x \# (\exists y \cdot P)$
by (*pred-auto*)

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

lemma *unrest-ex-diff* [*unrest*]:
assumes $x \bowtie y \ y \# P$
shows $y \# (\exists x \cdot P)$
using *assms lens-indep-comm*
by (*rel-simp'*, *fastforce*)

lemma *unrest-all-in* [*unrest*]:
 $\llbracket \text{mwb-lens } y; x \subseteq_L y \rrbracket \Longrightarrow x \# (\forall y \cdot P)$
by (*pred-auto*)

lemma *unrest-all-diff* [*unrest*]:
assumes $x \bowtie y \ y \# P$
shows $y \# (\forall x \cdot P)$
using *assms*
by (*pred-simp*, *simp-all add: lens-indep-comm*)

lemma *unrest-var-res-diff* [*unrest*]:
assumes $x \bowtie y$
shows $y \# (P \upharpoonright_v x)$
using *assms* **by** (*pred-auto*)

lemma *unrest-var-res-in* [*unrest*]:
assumes *mwb-lens* $x\ y \subseteq_L x\ y \# P$
shows $y \# (P \upharpoonright_v x)$
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 $\bigwedge y. x \# P(y)$
shows $x \# (\exists y. P(y))$
using *assms* **by** (*pred-auto*)

lemma *unrest-shAll* [*unrest*]:
assumes $\bigwedge y. x \# P(y)$
shows $x \# (\forall y. P(y))$
using *assms* **by** (*pred-auto*)

lemma *unrest-closure* [*unrest*]:
 $x \# [P]_u$
by (*pred-auto*)

10.4 Used-by laws

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

lemma *usedBy-conj* [*unrest*]:
 $\llbracket x \# P; x \# Q \rrbracket \implies x \# (P \wedge Q)$
by (*pred-simp*)

lemma *usedBy-disj* [*unrest*]:
 $\llbracket x \# P; x \# Q \rrbracket \implies x \# (P \vee Q)$
by (*pred-simp*)

lemma *usedBy-impl* [*unrest*]:
 $\llbracket x \# P; x \# Q \rrbracket \implies x \# (P \Rightarrow Q)$
by (*pred-simp*)

lemma *usedBy-iff* [*unrest*]:
 $\llbracket x \# P; x \# Q \rrbracket \implies x \# (P \Leftrightarrow Q)$
by (*pred-simp*)

10.5 Substitution Laws

Substitution is monotone

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

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

lemma *subst-false* [*usubst*]: $\sigma \dagger \text{false} = \text{false}$

by (pred-auto)

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

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

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

lemma *subst-disj* [usubst]: $\sigma \dagger (P \vee Q) = (\sigma \dagger P \vee \sigma \dagger Q)$
by (pred-auto)

lemma *subst-conj* [usubst]: $\sigma \dagger (P \wedge Q) = (\sigma \dagger P \wedge \sigma \dagger Q)$
by (pred-auto)

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

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

lemma *subst-UINF* [usubst]: $\sigma \dagger (\prod i \mid P(i) \cdot Q(i)) = (\prod i \mid (\sigma \dagger P(i)) \cdot (\sigma \dagger Q(i)))$
by (pred-auto)

lemma *subst-USUP* [usubst]: $\sigma \dagger (\bigsqcup i \mid P(i) \cdot Q(i)) = (\bigsqcup i \mid (\sigma \dagger P(i)) \cdot (\sigma \dagger Q(i)))$
by (pred-auto)

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

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

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

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

lemma *subst-ex-same* [usubst]:
mbw-lens $x \Rightarrow \sigma(x \mapsto_s v) \dagger (\exists x \cdot P) = \sigma \dagger (\exists x \cdot P)$
by (pred-auto)

lemma *subst-ex-same'* [usubst]:
mbw-lens $x \Rightarrow \sigma(x \mapsto_s v) \dagger (\exists \&x \cdot P) = \sigma \dagger (\exists \&x \cdot P)$
by (pred-auto)

lemma *subst-ex-indep* [usubst]:
assumes $x \bowtie y \ y \nparallel v$
shows $(\exists y \cdot P) \llbracket v/x \rrbracket = (\exists y \cdot P \llbracket v/x \rrbracket)$
using *assms*
apply (pred-auto)
using *lens-indep-comm* apply *fastforce* +
done

lemma *subst-ex-unrest* [*usubst*]:

$x \# \sigma \implies \sigma \dagger (\exists x \cdot P) = (\exists x \cdot \sigma \dagger P)$
by (*pred-auto*)

lemma *subst-all-same* [*usubst*]:

$mwb\text{-}lens\ x \implies \sigma(x \mapsto_s v) \dagger (\forall x \cdot P) = \sigma \dagger (\forall x \cdot P)$
by (*simp add: id-subst subst-unrest unrest-all-in*)

lemma *subst-all-indep* [*usubst*]:

assumes $x \bowtie y \ y \# v$
shows $(\forall y \cdot P) \llbracket v/x \rrbracket = (\forall y \cdot P \llbracket v/x \rrbracket)$
using *assms*
by (*pred-simp, simp-all add: lens-indep-comm*)

lemma *msubst-true* [*usubst*]: $true \llbracket x \rightarrow v \rrbracket = true$

by (*pred-auto*)

lemma *msubst-false* [*usubst*]: $false \llbracket x \rightarrow v \rrbracket = false$

by (*pred-auto*)

lemma *msubst-not* [*usubst*]: $(\neg P(x)) \llbracket x \rightarrow v \rrbracket = (\neg ((P\ x) \llbracket x \rightarrow v \rrbracket))$

by (*pred-auto*)

lemma *msubst-not-2* [*usubst*]: $(\neg P\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket = (\neg ((P\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket))$

by (*pred-auto*)⁺

lemma *msubst-disj* [*usubst*]: $(P(x) \vee Q(x)) \llbracket x \rightarrow v \rrbracket = ((P(x)) \llbracket x \rightarrow v \rrbracket \vee (Q(x)) \llbracket x \rightarrow v \rrbracket)$

by (*pred-auto*)

lemma *msubst-disj-2* [*usubst*]: $(P\ x\ y \vee Q\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket = ((P\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket \vee (Q\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket)$

by (*pred-auto*)⁺

lemma *msubst-conj* [*usubst*]: $(P(x) \wedge Q(x)) \llbracket x \rightarrow v \rrbracket = ((P(x)) \llbracket x \rightarrow v \rrbracket \wedge (Q(x)) \llbracket x \rightarrow v \rrbracket)$

by (*pred-auto*)

lemma *msubst-conj-2* [*usubst*]: $(P\ x\ y \wedge Q\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket = ((P\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket \wedge (Q\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket)$

by (*pred-auto*)⁺

lemma *msubst-implies* [*usubst*]:

$(P\ x \implies Q\ x) \llbracket x \rightarrow v \rrbracket = ((P\ x) \llbracket x \rightarrow v \rrbracket \implies (Q\ x) \llbracket x \rightarrow v \rrbracket)$

by (*pred-auto*)

lemma *msubst-implies-2* [*usubst*]:

$(P\ x\ y \implies Q\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket = ((P\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket \implies (Q\ x\ y) \llbracket (x,y) \rightarrow v \rrbracket)$

by (*pred-auto*)⁺

lemma *msubst-shAll* [*usubst*]:

$(\forall x \cdot P\ x\ y) \llbracket y \rightarrow v \rrbracket = (\forall x \cdot (P\ x\ y) \llbracket y \rightarrow v \rrbracket)$

by (*pred-auto*)

lemma *msubst-shAll-2* [*usubst*]:

$(\forall x \cdot P\ x\ y\ z) \llbracket (y,z) \rightarrow v \rrbracket = (\forall x \cdot (P\ x\ y\ z) \llbracket (y,z) \rightarrow v \rrbracket)$

by (*pred-auto*)⁺

10.6 Sandbox for conjectures

definition *utp-sandbox* :: $'\alpha \text{ upred} \Rightarrow \text{bool}$ (*TRY*'(-')) **where**

TRY(*P*) = (*P* = *undefined*)

translations

P <= *CONST utp-sandbox P*

end

11 Alphabet Manipulation

theory *utp-alphabet*

imports

utp-pred utp-usedby

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 an 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 (β) injects into the larger alphabet (α). 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* :: ($'a, 'b$) *uexpr* \Rightarrow ($'b, 'a$) *lens* \Rightarrow ($'a, 'a$) *uexpr* (**infixr** \oplus_p 95)

is $\lambda P x b. P (get_x b)$.

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*: ($P \oplus_p a$) $\oplus_p b$ = $P \oplus_p (a ;_L b)$

by (*pred-auto*)

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

lemma *aext-id* [*simp*]: $P \oplus_p 1_L$ = 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*]: $\llbracket v \rrbracket \oplus_p a = \llbracket v \rrbracket$
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*]: $\text{numeral } n \oplus_p a = \text{numeral } n$
by (*pred-auto*)

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

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

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

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

lemma *aext-or* [*alpha*]: $(P \vee Q) \oplus_p x = (P \oplus_p x \vee Q \oplus_p x)$
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-shEx* [*alpha*]: $(\exists x \cdot P x) \oplus_p a = (\exists x \cdot P x \oplus_p a)$
by (*rel-auto*)

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

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

lemma *aext-UINF-ind-2* [*alpha*]: $(\bigcap (i, j) \cdot P i j) \oplus_p a = (\bigcap (i, j) \cdot P i j \oplus_p a)$
by (*rel-auto*)

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

Alphabet extension distributes through the function liftings.

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

lemma *aext-bop* [*alpha*]: $\text{bop } f u v \oplus_p a = \text{bop } f (u \oplus_p a) (v \oplus_p a)$

by (*pred-auto*)

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

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

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

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

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

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

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

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

lemma *aext-var* [*alpha*]:
 $\text{var } x \oplus_p \ a = \text{var } (x \ ;_L \ a)$
by (*pred-auto*)

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

Alphabet extension is monotonic and continuous.

lemma *aext-mono*: $P \sqsubseteq Q \implies P \oplus_p \ a \sqsubseteq Q \oplus_p \ a$
by (*pred-auto*)

lemma *aext-cont* [*alpha*]: $\text{vwb-lens } a \implies (\bigsqcap \ A) \oplus_p \ a = (\bigsqcap \ P \in A. \ P \oplus_p \ a)$
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*]:
 $\llbracket \text{mwb-lens } a; \ x \ \sharp \ p \rrbracket \implies \text{unrest } (x \ ;_L \ a) \ (p \oplus_p \ a)$
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*]:
 $a \bowtie b \implies b \ \sharp \ (p \oplus_p \ a)$
by *pred-auto*

11.3 Expression Alphabet Restriction

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

lift-definition $arestr :: ('a, 'α) uexpr \Rightarrow ('β, 'α) lens \Rightarrow ('a, 'β) uexpr \text{ (infixr } \vdash_e 90)$
is $\lambda P x b. P \text{ (create}_x b) \text{ .}$

update-uexpr-rep-eq-thms

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

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

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

lemma $aext-arestr \text{ [alpha]:}$
assumes $mwb-lens a \text{ bij-lens } (a +_L b) \text{ } a \bowtie b \text{ } b \# P$
shows $(P \vdash_e a) \oplus_p a = P$
proof –
from $assms(2)$ **have** $1_L \subseteq_L a +_L b$
by $(simp \text{ add: } bij-lens-equiv-id \text{ lens-equiv-def})$
with $assms(1,3,4)$ **show** $?thesis$
apply $(auto \text{ simp add: id-lens-def lens-plus-def sublens-def lens-comp-def prod.case-eq-if})$
apply $(pred-simp)$
apply $(metis \text{ lens-indep-comm mwb-lens-weak weak-lens.put-get})$
done
qed

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

lemma $aext-arestr' \text{ [alpha]:}$
assumes $a \nmid P$
shows $(P \vdash_e a) \oplus_p a = P$
by $(rel-simp, \text{metis } assms \text{ lens-override-def usedBy-uexpr.rep-eq})$

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

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

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

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

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

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

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

lemma *arestr-not* [*alpha*]: $(\neg P) \upharpoonright_e a = (\neg (P \upharpoonright_e a))$
by (*pred-auto*)

lemma *arestr-and* [*alpha*]: $(P \wedge Q) \upharpoonright_e x = (P \upharpoonright_e x \wedge Q \upharpoonright_e x)$
by (*pred-auto*)

lemma *arestr-or* [*alpha*]: $(P \vee Q) \upharpoonright_e x = (P \upharpoonright_e x \vee Q \upharpoonright_e x)$
by (*pred-auto*)

lemma *arestr-imp* [*alpha*]: $(P \Rightarrow Q) \upharpoonright_e x = (P \upharpoonright_e x \Rightarrow Q \upharpoonright_e x)$
by (*pred-auto*)

lemma *ares-UNIF-ind* [*alpha*]: $(\prod i \cdot P i) \upharpoonright_e a = (\prod i \cdot P i \upharpoonright_e a)$
by (*rel-auto*)

lemma *ares-UNIF-ind-2* [*alpha*]: $(\prod (i, j) \cdot P i j) \upharpoonright_e a = (\prod (i, j) \cdot P i j \upharpoonright_e a)$
by (*rel-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* :: $'\alpha \text{ upred} \Rightarrow (' \beta \Longrightarrow ' \alpha) \Rightarrow ' \beta \text{ upred}$
where [*upred-defs*]: $\text{upred-ares } P a = (P \upharpoonright_v a) \upharpoonright_e a$

syntax

-upred-ares :: $\text{logic} \Rightarrow \text{salpha} \Rightarrow \text{logic}$ (**infixl** \upharpoonright_p 90)

translations

-upred-ares $P a == \text{CONST upred-ares } P a$

lemma *upred-aext-ares* [*alpha*]:
 $\text{vwb-lens } a \Longrightarrow P \oplus_p a \upharpoonright_p a = P$
by (*pred-auto*)

lemma *upred-ares-aext* [*alpha*]:
 $a \nmid P \Longrightarrow (P \upharpoonright_p a) \oplus_p a = P$
by (*pred-auto*)

lemma *upred-arestr-lit* [*simp*]: $\ll v \gg \upharpoonright_p a = \ll v \gg$
by (*pred-auto*)

lemma *upred-arestr-true* [*simp*]: $\text{true} \upharpoonright_p a = \text{true}$
by (*pred-auto*)

lemma *upred-arestr-false* [*simp*]: $\text{false} \upharpoonright_p a = \text{false}$
by (*pred-auto*)

lemma *upred-arestr-or* [*alpha*]: $(P \vee Q) \upharpoonright_p x = (P \upharpoonright_p x \vee Q \upharpoonright_p x)$
by (*pred-auto*)

11.5 Alphabet Lens Laws

lemma *alpha-in-var* [*alpha*]: $x ;_L \text{fst}_L = \text{in-var } x$
by (*simp add: in-var-def*)

lemma *alpha-out-var* [*alpha*]: $x ;_L \text{snd}_L = \text{out-var } x$
by (*simp add: out-var-def*)

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

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

11.6 Substitution Alphabet Extension

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

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

lemma *id-subst-ext* [*usubst*]:
 $\text{wb-lens } x \implies \text{id} \oplus_s x = \text{id}$
by *pred-auto*

lemma *upd-subst-ext* [*alpha*]:
 $\text{vwb-lens } x \implies \sigma(y \mapsto_s v) \oplus_s x = (\sigma \oplus_s x)(\&x:y \mapsto_s v \oplus_p x)$
by *pred-auto*

lemma *apply-subst-ext* [*alpha*]:
 $\text{vwb-lens } x \implies (\sigma \dagger e) \oplus_p x = (\sigma \oplus_s x) \dagger (e \oplus_p x)$
by (*pred-auto*)

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

lemma *subst-aext-comp* [*usubst*]:
 $\text{vwb-lens } a \implies (\sigma \oplus_s a) \circ (\varrho \oplus_s a) = (\sigma \circ \varrho) \oplus_s a$
by *pred-auto*

lemma *subst-arestr* [*usubst*]: $\text{vwb-lens } a \implies \sigma \dagger (P \upharpoonright_e a) = (((\sigma \oplus_s a) \dagger P) \upharpoonright_e a)$
by (*pred-auto*)

11.7 Substitution Alphabet Restriction

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

definition $\text{subst-res} :: 'a \text{ usubst} \Rightarrow ('b \Longrightarrow 'a) \Rightarrow 'b \text{ usubst}$ (**infix** \downarrow_s 65) **where**
 $[\text{upred-defs}]: \sigma \downarrow_s x = (\lambda s. \text{get}_x (\sigma (\text{create}_x s)))$

lemma id-subst-res $[\text{usubst}]$:
 $\text{mwb-lens } x \Longrightarrow \text{id} \downarrow_s x = \text{id}$
by pred-auto

lemma upd-subst-res $[\text{alpha}]$:
 $\text{mwb-lens } x \Longrightarrow \sigma(\&x:y \mapsto_s v) \downarrow_s x = (\sigma \downarrow_s x)(\&y \mapsto_s v \downarrow_e x)$
by (pred-auto)

lemma subst-ext-res $[\text{usubst}]$:
 $\text{mwb-lens } x \Longrightarrow (\sigma \oplus_s x) \downarrow_s x = \sigma$
by (pred-auto)

lemma $\text{unrest-subst-alpha-ext}$ $[\text{unrest}]$:
 $x \boxtimes y \Longrightarrow x \# (P \oplus_s y)$
by $(\text{pred-simp robust, metis lens-indep-def})$
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 $\lceil P \rceil$ with some subscript to denote lifting an expression into a larger alphabet, and $\lfloor P \rfloor$ for dropping into a smaller alphabet.

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

abbreviation $\text{lift-pre} :: ('a, 'a) \text{ uexpr} \Rightarrow ('a, 'a \times 'b) \text{ uexpr}$ ($\lceil \cdot \rceil_{<}$)
where $\lceil P \rceil_{<} \equiv P \oplus_p \text{fst}_L$

abbreviation $\text{drop-pre} :: ('a, 'a \times 'b) \text{ uexpr} \Rightarrow ('a, 'a) \text{ uexpr}$ ($\lfloor \cdot \rfloor_{<}$)
where $\lfloor P \rfloor_{<} \equiv P \downarrow_e \text{fst}_L$

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

abbreviation $\text{lift-post} :: ('a, 'b) \text{ uexpr} \Rightarrow ('a, 'a \times 'b) \text{ uexpr}$ ($\lceil \cdot \rceil_{>}$)
where $\lceil P \rceil_{>} \equiv P \oplus_p \text{snd}_L$

abbreviation $\text{drop-post} :: ('a, 'a \times 'b) \text{ uexpr} \Rightarrow ('a, 'b) \text{ uexpr}$ ($\lfloor \cdot \rfloor_{>}$)
where $\lfloor P \rfloor_{>} \equiv P \downarrow_e \text{snd}_L$

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

```
lemma lift-post-var [simp]:  
   $\llbracket \text{var } x \rrbracket_{>} = \$x'$   
  by (alpha-tac)
```

12.3 Substitution Laws

```
lemma pre-var-subst [usubst]:  
   $\sigma(\$x \mapsto_s \llbracket v \rrbracket) \dagger \llbracket P \rrbracket_{<} = \sigma \dagger \llbracket P[\llbracket v \rrbracket / \&x] \rrbracket_{<}$   
  by (pred-simp)
```

12.4 Unrestriction laws

Crucially, the lifting operators allow us to demonstrate unrestricted 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  $x :: ('a \Longrightarrow 'a)$   
  shows  $\$x' \# \llbracket p \rrbracket_{<}$   
  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 ( $\leq$ ) ( $<$ )  
  disj-upred false-upred true-upred  
  by (unfold-locales; pred-auto)
```

```
lemma taut-true [simp]:  $\text{'true'}$   
  by (pred-auto)
```

```
lemma taut-false [simp]:  $\text{'false'}$  = False  
  by (pred-auto)
```

```
lemma taut-conj:  $\text{'A } \wedge B' = (\text{'A'} \wedge \text{'B'})$   
  by (rel-auto)
```

lemma *taut-conj-elim* [elim!]:
 $\llbracket 'A \wedge B'; \llbracket 'A'; 'B' \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$
by (*rel-auto*)

lemma *taut-refine-impl*: $\llbracket Q \sqsubseteq P; 'P' \rrbracket \Longrightarrow 'Q'$
by (*rel-auto*)

lemma *taut-shEx-elim*:
 $\llbracket '(\exists x. P x)'; \bigwedge x. \Sigma \# P x; \bigwedge x. 'P x' \Longrightarrow Q \rrbracket \Longrightarrow Q$
by (*rel-blast*)

Linking refinement and HOL implication

lemma *refine-prop-intro*:
assumes $\Sigma \# P \Sigma \# Q 'Q' \Longrightarrow 'P'$
shows $P \sqsubseteq Q$
using *assms*
by (*pred-auto*)

lemma *taut-not*: $\Sigma \# P \Longrightarrow (\neg 'P') = ' \neg P '$
by (*rel-auto*)

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

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

lemma *taut-impl-intro*:
 $\llbracket \Sigma \# P; 'P' \Longrightarrow 'Q' \rrbracket \Longrightarrow 'P \Rightarrow Q'$
by (*rel-auto*)

lemma *upred-eval-taut*:
 $'P[\llbracket b \rrbracket_e / \&\mathbf{v}]' = \llbracket P \rrbracket_e b$
by (*pred-auto*)

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

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

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

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

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

lemma *conj-subst*: $P = R \Longrightarrow ((P::'\alpha \text{ upred}) \wedge Q) = (R \wedge Q)$
by (*pred-auto*)

lemma *disj-subst*: $P = R \implies ((P::'\alpha \text{ upred}) \vee Q) = (R \vee Q)$
by (*pred-auto*)

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

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

lemma *conj-disj-abs*: $((P::'\alpha \text{ upred}) \wedge (P \vee Q)) = P$
by (*pred-auto*)

lemma *disj-conj-abs*: $((P::'\alpha \text{ upred}) \vee (P \wedge Q)) = P$
by (*pred-auto*)

lemma *conj-disj-distr*: $((P::'\alpha \text{ upred}) \wedge (Q \vee R)) = ((P \wedge Q) \vee (P \wedge R))$
by (*pred-auto*)

lemma *disj-conj-distr*: $((P::'\alpha \text{ upred}) \vee (Q \wedge R)) = ((P \vee Q) \wedge (P \vee R))$
by (*pred-auto*)

lemma *true-disj-zero* [*simp*]:
 $(P \vee \text{true}) = \text{true} \quad (\text{true} \vee P) = \text{true}$
by (*pred-auto*) +

lemma *true-conj-zero* [*simp*]:
 $(P \wedge \text{false}) = \text{false} \quad (\text{false} \wedge P) = \text{false}$
by (*pred-auto*) +

lemma *false-sup* [*simp*]: $\text{false} \sqcap P = P \quad P \sqcap \text{false} = P$
by (*pred-auto*) +

lemma *true-inf* [*simp*]: $\text{true} \sqcup P = P \quad P \sqcup \text{true} = P$
by (*pred-auto*) +

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

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

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

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

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

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

lemma *impl-refine-intro*:

$\llbracket Q_1 \sqsubseteq P_1; P_2 \sqsubseteq (P_1 \wedge Q_2) \rrbracket \Longrightarrow (P_1 \Rightarrow P_2) \sqsubseteq (Q_1 \Rightarrow Q_2)$
by (*pred-auto*)

lemma *spec-refine*:

$Q \sqsubseteq (P \wedge R) \Longrightarrow (P \Rightarrow Q) \sqsubseteq R$
by (*rel-auto*)

lemma *impl-disjI*: $\llbracket 'P \Rightarrow R'; 'Q \Rightarrow R' \rrbracket \Longrightarrow '(P \vee Q) \Rightarrow R'$
by (*rel-auto*)

lemma *conditional-iff*:

$(P \Rightarrow Q) = (P \Rightarrow R) \longleftrightarrow 'P \Rightarrow (Q \Leftrightarrow R)'$
by (*pred-auto*)

lemma *p-and-not-p* [*simp*]: $(P \wedge \neg P) = \text{false}$
by (*pred-auto*)

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

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

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

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

lemma *not-conj-deMorgans* [*simp*]: $(\neg ((P::'\alpha \text{ upred}) \wedge Q)) = ((\neg P) \vee (\neg Q))$
by (*pred-auto*)

lemma *not-disj-deMorgans* [*simp*]: $(\neg ((P::'\alpha \text{ upred}) \vee Q)) = ((\neg P) \wedge (\neg Q))$
by (*pred-auto*)

lemma *conj-disj-not-abs* [*simp*]: $((P::'\alpha \text{ upred}) \wedge ((\neg P) \vee Q)) = (P \wedge Q)$
by (*pred-auto*)

lemma *subsumption1*:

$'P \Rightarrow Q' \Longrightarrow (P \vee Q) = Q$
by (*pred-auto*)

lemma *subsumption2*:

$'Q \Rightarrow P' \Longrightarrow (P \vee Q) = P$
by (*pred-auto*)

lemma *neg-conj-cancel1*: $(\neg P \wedge (P \vee Q)) = (\neg P \wedge Q :: '\alpha \text{ upred})$
by (*pred-auto*)

lemma *neg-conj-cancel2*: $(\neg Q \wedge (P \vee Q)) = (\neg Q \wedge P :: '\alpha \text{ upred})$
by (*pred-auto*)

lemma *double-negation* [*simp*]: $(\neg \neg (P::'\alpha \text{ upred})) = P$
by (*pred-auto*)

lemma *true-not-false* [simp]: $true \neq false$ $false \neq true$
 by (pred-auto)+

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

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

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

lemma *taut-iff-eq*:
 $'P \Leftrightarrow Q' \longleftrightarrow (P = Q)$
 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 $(P \sqcap Q) = (P \vee Q)$
 by (pred-auto)

lemma *usup-and*:
 fixes $P Q :: 'a$ upred
 shows $(P \sqcup Q) = (P \wedge Q)$
 by (pred-auto)

lemma *UINF-alt-def*:
 $(\bigsqcap i \mid A(i) \cdot P(i)) = (\bigsqcap i \cdot A(i) \wedge P(i))$
 by (rel-auto)

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

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

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

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

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

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

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

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

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

lemma *UINF-false* [simp]: $(\bigcap i \mid P(i) \cdot \text{false}) = \text{false}$
by (*pred-auto*)

lemma *UINF-where-false* [simp]: $(\bigcap i \mid \text{false} \cdot P(i)) = \text{false}$
by (*rel-auto*)

lemma *UINF-cong-eq*:
 $\llbracket \bigwedge x. P_1(x) = P_2(x); \bigwedge x. 'P_1(x) \Rightarrow Q_1(x) =_u Q_2(x)' \rrbracket \implies$
 $(\bigcap x \mid P_1(x) \cdot Q_1(x)) = (\bigcap x \mid P_2(x) \cdot Q_2(x))$
by (*unfold UINF-def, pred-simp, metis*)

lemma *UINF-as-Sup*: $(\bigcap P \in \mathcal{P} \cdot P) = \bigcap \mathcal{P}$
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*: $(\bigcap P \in A \cdot f(P)) = (\bigcap P \in A. f(P))$
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'*: $(\bigcap P \cdot f(P)) = (\bigcap P. f(P))$
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*: $(\bigcap P \mid \ll P \gg \in_u \ll A \gg \cdot f(P)) = \bigcap (f \text{ ' } A)$
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*: $(\bigcup P \in \mathcal{P} \cdot P) = \bigcup \mathcal{P}$
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*: $(\bigcup P \in A \cdot f(P)) = (\bigcup P \in A. f(P))$
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*: $(\bigsqcup P \in \mathcal{P} \cdot f(P)) = \bigsqcup (f \text{ ' } \mathcal{P})$
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-image-eq [simp]*: $USUP (\lambda i. \llbracket i \rrbracket \in_u \llbracket f \text{ ' } A \rrbracket) g = (\bigsqcup i \in A \cdot g(f(i)))$
by (*pred-simp, rule-tac cong[of Inf Inf], auto*)

lemma *UINF-image-eq [simp]*: $UINF (\lambda i. \llbracket i \rrbracket \in_u \llbracket f \text{ ' } A \rrbracket) g = (\prod i \in A \cdot g(f(i)))$
by (*pred-simp, rule-tac cong[of Sup Sup], auto*)

lemma *subst-continuous [usubst]*: $\sigma \dagger (\prod A) = (\prod \{\sigma \dagger P \mid P. P \in A\})$
by (*simp add: UINF-as-Sup[THEN sym] usubst setcompr-eq-image*)

lemma *not-UINF*: $(\neg (\prod i \in A \cdot P(i))) = (\bigsqcup i \in A \cdot \neg P(i))$
by (*pred-auto*)

lemma *not-USUP*: $(\neg (\bigsqcup i \in A \cdot P(i))) = (\prod i \in A \cdot \neg P(i))$
by (*pred-auto*)

lemma *not-UINF-ind*: $(\neg (\prod i \cdot P(i))) = (\bigsqcup i \cdot \neg P(i))$
by (*pred-auto*)

lemma *not-USUP-ind*: $(\neg (\bigsqcup i \cdot P(i))) = (\prod i \cdot \neg P(i))$
by (*pred-auto*)

lemma *UINF-empty [simp]*: $(\prod i \in \{\} \cdot P(i)) = \text{false}$
by (*pred-auto*)

lemma *UINF-insert [simp]*: $(\prod i \in \text{insert } x \text{ } xs \cdot P(i)) = (P(x) \sqcap (\prod i \in xs \cdot P(i)))$
apply (*pred-simp*)
apply (*subst Sup-insert[THEN sym]*)
apply (*rule-tac cong[of Sup Sup]*)
apply (*auto*)
done

lemma *UINF-atLeast-first*:
 $P(n) \sqcap (\prod i \in \{\text{Suc } n..\} \cdot P(i)) = (\prod i \in \{n..\} \cdot P(i))$
proof –
have *insert* $n \{\text{Suc } n..\} = \{n..\}$
by (*auto*)
thus *?thesis*
by (*metis UINF-insert*)
qed

lemma *UINF-atLeast-Suc*:

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

lemma *USUP-empty* [*simp*]: $(\bigsqcup i \in \{\} \cdot P(i)) = true$
by (*pred-auto*)

lemma *USUP-insert* [*simp*]: $(\bigsqcup i \in insert\ x\ xs \cdot P(i)) = (P(x) \sqcup (\bigsqcup i \in xs \cdot P(i)))$
apply (*pred-simp*)
apply (*subst Inf-insert* [*THEN sym*])
apply (*rule-tac cong* [*of Inf Inf*])
apply (*auto*)
done

lemma *USUP-atLeast-first*:
 $(P(n) \wedge (\bigsqcup i \in \{Suc\ n..\} \cdot P(i))) = (\bigsqcup i \in \{n..\} \cdot P(i))$
proof –
have *insert n {Suc n..} = {n..}*
by (*auto*)
thus *?thesis*
by (*metis USUP-insert conj-upred-def*)
qed

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

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

lemma *conj-UINF-ind-dist*:
 $(P \wedge (\bigsqcup Q \cdot F(Q))) = (\bigsqcup Q \cdot P \wedge F(Q))$
by *pred-auto*

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

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

lemma *conj-USUP-dist*:
 $S \neq \{\} \implies (P \wedge (\bigsqcup Q \in S \cdot F(Q))) = (\bigsqcup Q \in S \cdot P \wedge F(Q))$
by (*subst uexpr-eq-iff, auto simp add: conj-upred-def USUP.rep-eq inf-uexpr.rep-eq bop.rep-eq lit.rep-eq*)

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

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

lemma *UINF-cong*:

assumes $\bigwedge P. P \in A \implies F(P) = G(P)$
shows $(\bigcap P \in A \cdot F(P)) = (\bigcap P \in A \cdot G(P))$
by (*simp add: UINF-as-Sup-collect assms*)

lemma *USUP-all-cong*:

assumes $\bigwedge P. F(P) = G(P)$
shows $(\bigsqcup P \cdot F(P)) = (\bigsqcup P \cdot G(P))$
by (*simp add: assms*)

lemma *USUP-cong*:

assumes $\bigwedge P. P \in A \implies F(P) = G(P)$
shows $(\bigsqcup P \in A \cdot F(P)) = (\bigsqcup P \in A \cdot G(P))$
by (*simp add: USUP-as-Inf-collect assms*)

lemma *UINF-subset-mono*: $A \subseteq B \implies (\bigcap P \in B \cdot F(P)) \sqsubseteq (\bigcap P \in A \cdot F(P))$
by (*simp add: SUP-subset-mono UINF-as-Sup-collect*)

lemma *USUP-subset-mono*: $A \subseteq B \implies (\bigsqcup P \in A \cdot F(P)) \sqsubseteq (\bigsqcup P \in B \cdot F(P))$
by (*simp add: INF-superset-mono USUP-as-Inf-collect*)

lemma *UINF-impl*: $(\bigcap P \in A \cdot F(P) \Rightarrow G(P)) = ((\bigsqcup P \in A \cdot F(P)) \Rightarrow (\bigcap P \in A \cdot G(P)))$
by (*pred-auto*)

lemma *USUP-is-forall*: $(\bigsqcup x \cdot P(x)) = (\forall x \cdot P(x))$
by (*pred-simp*)

lemma *USUP-ind-is-forall*: $(\bigsqcup x \in A \cdot P(x)) = (\forall x \in \ll A \gg \cdot P(x))$
by (*pred-auto*)

lemma *UINF-is-exists*: $(\bigcap x \cdot P(x)) = (\exists x \cdot P(x))$
by (*pred-simp*)

lemma *UINF-all-nats* [*simp*]:

fixes $P :: \text{nat} \Rightarrow 'a \text{ upred}$
shows $(\bigcap n \cdot \bigcap i \in \{0..n\} \cdot P(i)) = (\bigcap n \cdot P(n))$
by (*pred-auto*)

lemma *USUP-all-nats* [*simp*]:

fixes $P :: \text{nat} \Rightarrow 'a \text{ upred}$
shows $(\bigsqcup n \cdot \bigsqcup i \in \{0..n\} \cdot P(i)) = (\bigsqcup n \cdot P(n))$
by (*pred-auto*)

lemma *UINF-upto-expand-first*:

$m < n \implies (\bigcap i \in \{m..<n\} \cdot P(i)) = ((P(m) :: 'a \text{ upred}) \vee (\bigcap i \in \{\text{Suc } m..<n\} \cdot P(i)))$
apply (*rel-auto*) **using** *Suc-leI le-eq-less-or-eq* **by** *auto*

lemma *UINF-upto-expand-last*:

$(\bigcap i \in \{0..<\text{Suc}(n)\} \cdot P(i)) = ((\bigcap i \in \{0..<n\} \cdot P(i)) \vee P(n))$
apply (*rel-auto*)
using *less-SucE* **by** *blast*

lemma *UINF-Suc-shift*: $(\bigcap i \in \{\text{Suc } 0..<\text{Suc } n\} \cdot P(i)) = (\bigcap i \in \{0..<n\} \cdot P(\text{Suc } i))$

apply (*rel-simp*)
apply (*rule cong[of Sup], auto*)

using *less-Suc-eq-0-disj* **by** *auto*

lemma *USUP-upto-expand-first*:

$(\bigsqcup i \in \{0..< \text{Suc}(n)\} \cdot P(i)) = (P(0) \wedge (\bigsqcup i \in \{1..< \text{Suc}(n)\} \cdot P(i)))$
apply (*rel-auto*)
using *not-less* **by** *auto*

lemma *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 *UINF-list-conv*:

$(\prod i \in \{0..< \text{length}(xs)\} \cdot f (xs ! i)) = \text{foldr } (\vee) (\text{map } f xs) \text{ false}$
apply (*induct xs*)
apply (*rel-auto*)
apply (*simp add: UINF-upto-expand-first UINF-Suc-shift*)
done

lemma *USUP-list-conv*:

$(\bigsqcup i \in \{0..< \text{length}(xs)\} \cdot f (xs ! i)) = \text{foldr } (\wedge) (\text{map } f xs) \text{ true}$
apply (*induct xs*)
apply (*rel-auto*)
apply (*simp-all add: USUP-upto-expand-first USUP-Suc-shift*)
done

lemma *UINF-refines*:

$\llbracket \bigwedge i. i \in I \implies P \sqsubseteq Q i \rrbracket \implies P \sqsubseteq (\prod i \in I \cdot Q i)$
by (*simp add: UINF-as-Sup-collect,metis SUP-least*)

lemma *UINF-refines'*:

assumes $\bigwedge i. P \sqsubseteq Q(i)$
shows $P \sqsubseteq (\prod i \cdot Q(i))$
using *assms*
apply (*rel-auto*) **using** *Sup-le-iff* **by** *fastforce*

lemma *UINF-pred-ueq [simp]*:

$(\prod x \mid \ll x \gg =_u v \cdot P(x)) = (P x) \llbracket x \rightarrow v \rrbracket$
by (*pred-auto*)

lemma *UINF-pred-lit-eq [simp]*:

$(\prod x \mid \ll x = v \gg \cdot P(x)) = (P v)$
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* $x \ \$x \ \# \ Q \ \$x' \ \# \ Q \ \$x \ \# \ R \ \$x' \ \# \ R$
shows $((\$x' =_u \$x \wedge Q) = (\$x' =_u \$x \wedge R)) \longleftrightarrow (Q = R)$
using *assms*

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 $(P \wedge \$x =_u v) = (P[v/\$x] \wedge \$x =_u v)$
 using assms
 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 $(P \wedge \$x' =_u v) = (P[v/\$x'] \wedge \$x' =_u v)$
 using assms
 by (pred-simp, (metis vwb-lens-wb wb-lens.get-put)+)

lemma conj-pos-var-subst:

assumes vwb-lens x
 shows $(\$x \wedge Q) = (\$x \wedge Q[true/\$x])$
 using assms
 by (pred-auto, metis (full-types) vwb-lens-wb wb-lens.get-put, metis (full-types) vwb-lens-wb wb-lens.get-put)

lemma conj-neg-var-subst:

assumes vwb-lens x
 shows $(\neg \$x \wedge Q) = (\neg \$x \wedge Q[false/\$x])$
 using assms
 by (pred-auto, metis (full-types) vwb-lens-wb wb-lens.get-put, metis (full-types) vwb-lens-wb wb-lens.get-put)

lemma upred-eq-true [simp]: $(p =_u true) = p$

by (pred-auto)

lemma upred-eq-false [simp]: $(p =_u false) = (\neg p)$

by (pred-auto)

lemma upred-true-eq [simp]: $(true =_u p) = p$

by (pred-auto)

lemma upred-false-eq [simp]: $(false =_u p) = (\neg p)$

by (pred-auto)

lemma conj-var-subst:

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

13.4 HOL Variable Quantifiers

lemma shEx-unbound [simp]: $(\exists x \cdot P) = P$

by (pred-auto)

lemma shEx-bool [simp]: $\text{shEx } P = (P \text{ True} \vee P \text{ False})$

by (pred-simp, metis (full-types))

lemma shEx-commute: $(\exists x \cdot \exists y \cdot P x y) = (\exists y \cdot \exists x \cdot P x y)$

by (pred-auto)

lemma *shEx-cong*: $\llbracket \bigwedge x. P\ x = Q\ x \rrbracket \implies \text{shEx}\ P = \text{shEx}\ Q$
by (*pred-auto*)

lemma *shEx-insert*: $(\exists\ x \in \text{insert}_u\ y\ A \cdot P(x)) = (P(x) \llbracket x \rightarrow y \rrbracket \vee (\exists\ x \in A \cdot P(x)))$
by (*pred-auto*)

lemma *shEx-one-point*: $(\exists\ x \cdot \llbracket x \rrbracket =_u v \wedge P(x)) = P(x) \llbracket x \rightarrow v \rrbracket$
by (*rel-auto*)

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

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

lemma *shAll-cong*: $\llbracket \bigwedge x. P\ x = Q\ x \rrbracket \implies \text{shAll}\ P = \text{shAll}\ Q$
by (*pred-auto*)

Quantifier lifting

named-theorems *uquant-lift*

lemma *shEx-lift-conj-1* [*uquant-lift*]:
 $((\exists\ x \cdot P(x)) \wedge Q) = (\exists\ x \cdot P(x) \wedge Q)$
by (*pred-auto*)

lemma *shEx-lift-conj-2* [*uquant-lift*]:
 $(P \wedge (\exists\ x \cdot Q(x))) = (\exists\ x \cdot P \wedge Q(x))$
by (*pred-auto*)

13.5 Case Splitting

lemma *eq-split-subst*:
assumes *vwb-lens* *x*
shows $(P = Q) \longleftrightarrow (\forall\ v. P \llbracket \llbracket v \rrbracket / x \rrbracket = Q \llbracket \llbracket v \rrbracket / x \rrbracket)$
using *assms*
by (*pred-auto*, *metis* *vwb-lens-wb* *wb-lens.source-stability*)

lemma *eq-split-substI*:
assumes *vwb-lens* *x* $\bigwedge v. P \llbracket \llbracket v \rrbracket / x \rrbracket = Q \llbracket \llbracket v \rrbracket / x \rrbracket$
shows $P = Q$
using *assms*(1) *assms*(2) *eq-split-subst* **by** *blast*

lemma *taut-split-subst*:
assumes *vwb-lens* *x*
shows $\llbracket P \rrbracket \longleftrightarrow (\forall\ v. \llbracket P \llbracket \llbracket v \rrbracket / x \rrbracket \rrbracket)$
using *assms*
by (*pred-auto*, *metis* *vwb-lens-wb* *wb-lens.source-stability*)

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

lemma *bool-eq-splitI*:

assumes *vwb-lens* x $P\llbracket true/x \rrbracket = Q\llbracket true/x \rrbracket$ $P\llbracket false/x \rrbracket = Q\llbracket false/x \rrbracket$
shows $P = Q$
by (*metis* (*full-types*) *assms* *eq-split-subst* *false-alt-def* *true-alt-def*)

lemma *subst-bool-split*:

assumes *vwb-lens* x
shows $\langle P \rangle = \langle (P\llbracket false/x \rrbracket \wedge P\llbracket true/x \rrbracket) \rangle$

proof –

from *assms* **have** $\langle P \rangle = (\forall v. \langle P\llbracket \langle v \rangle / x \rrbracket \rangle)$
by (*subst* *taut-split-subst*[*of* x], *auto*)
also have $\dots = (\langle P\llbracket \langle True \rangle / x \rrbracket \rangle \wedge \langle P\llbracket \langle False \rangle / x \rrbracket \rangle)$
by (*metis* (*mono-tags*, *lifting*))
also have $\dots = \langle (P\llbracket false/x \rrbracket \wedge P\llbracket true/x \rrbracket) \rangle$
by (*pred-auto*)
finally show *?thesis* .

qed

lemma *subst-eq-replace*:

fixes $x :: ('a \implies 'b)$
shows $(p\llbracket u/x \rrbracket \wedge u =_u v) = (p\llbracket v/x \rrbracket \wedge u =_u v)$
by (*pred-auto*)

13.6 UTP Quantifiers

lemma *one-point*:

assumes *mwb-lens* x $x \# v$
shows $(\exists x. P \wedge \text{var } x =_u v) = P\llbracket v/x \rrbracket$
using *assms*
by (*pred-auto*)

lemma *exists-twice*: *mwb-lens* $x \implies (\exists x. \exists x. P) = (\exists x. P)$
by (*pred-auto*)

lemma *all-twice*: *mwb-lens* $x \implies (\forall x. \forall x. P) = (\forall x. P)$
by (*pred-auto*)

lemma *exists-sub*: $\llbracket \text{mwb-lens } y; x \subseteq_L y \rrbracket \implies (\exists x. \exists y. P) = (\exists y. P)$
by (*pred-auto*)

lemma *all-sub*: $\llbracket \text{mwb-lens } y; x \subseteq_L y \rrbracket \implies (\forall x. \forall y. P) = (\forall y. P)$
by (*pred-auto*)

lemma *ex-commute*:

assumes $x \bowtie y$
shows $(\exists x. \exists y. P) = (\exists y. \exists x. P)$
using *assms*
apply (*pred-auto*)
using *lens-indep-comm* **apply** *fastforce* +
done

lemma *all-commute*:

assumes $x \bowtie y$
shows $(\forall x. \forall y. P) = (\forall y. \forall x. P)$
using *assms*
apply (*pred-auto*)
using *lens-indep-comm* **apply** *fastforce* +

done

lemma *ex-equiv*:

assumes $x \approx_L y$

shows $(\exists x \cdot P) = (\exists y \cdot P)$

using *assms*

by (*pred-simp*, *metis* (*no-types*, *lifting*) *lens.select-convs*(2))

lemma *all-equiv*:

assumes $x \approx_L y$

shows $(\forall x \cdot P) = (\forall y \cdot P)$

using *assms*

by (*pred-simp*, *metis* (*no-types*, *lifting*) *lens.select-convs*(2))

lemma *ex-zero*:

$(\exists \emptyset \cdot P) = P$

by (*pred-auto*)

lemma *all-zero*:

$(\forall \emptyset \cdot P) = P$

by (*pred-auto*)

lemma *ex-plus*:

$(\exists y; x \cdot P) = (\exists x \cdot \exists y \cdot P)$

by (*pred-auto*)

lemma *all-plus*:

$(\forall y; x \cdot P) = (\forall x \cdot \forall y \cdot P)$

by (*pred-auto*)

lemma *closure-all*:

$[P]_u = (\forall \Sigma \cdot P)$

by (*pred-auto*)

lemma *unrest-as-exists*:

$vwb\text{-}lens\ x \implies (x \sharp P) \longleftrightarrow ((\exists x \cdot P) = P)$

by (*pred-simp*, *metis* *vwb-lens.put-eq*)

lemma *ex-mono*: $P \sqsubseteq Q \implies (\exists x \cdot P) \sqsubseteq (\exists x \cdot Q)$

by (*pred-auto*)

lemma *ex-weakens*: $wb\text{-}lens\ x \implies (\exists x \cdot P) \sqsubseteq P$

by (*pred-simp*, *metis* *wb-lens.get-put*)

lemma *all-mono*: $P \sqsubseteq Q \implies (\forall x \cdot P) \sqsubseteq (\forall x \cdot Q)$

by (*pred-auto*)

lemma *all-strengthens*: $wb\text{-}lens\ x \implies P \sqsubseteq (\forall x \cdot P)$

by (*pred-simp*, *metis* *wb-lens.get-put*)

lemma *ex-unrest*: $x \sharp P \implies (\exists x \cdot P) = P$

by (*pred-auto*)

lemma *all-unrest*: $x \sharp P \implies (\forall x \cdot P) = P$

by (*pred-auto*)

lemma *not-ex-not*: $\neg (\exists x \cdot \neg P) = (\forall x \cdot P)$
by (*pred-auto*)

lemma *not-all-not*: $\neg (\forall x \cdot \neg P) = (\exists x \cdot P)$
by (*pred-auto*)

lemma *ex-conj-contr-left*: $x \# P \implies (\exists x \cdot P \wedge Q) = (P \wedge (\exists x \cdot Q))$
by (*pred-auto*)

lemma *ex-conj-contr-right*: $x \# Q \implies (\exists x \cdot P \wedge Q) = ((\exists x \cdot P) \wedge Q)$
by (*pred-auto*)

13.7 Variable Restriction

lemma *var-res-all*:
 $P \upharpoonright_v \Sigma = P$
by (*rel-auto*)

lemma *var-res-twice*:
 $mwb\text{-}lens\ x \implies P \upharpoonright_v x \upharpoonright_v x = P \upharpoonright_v x$
by (*pred-auto*)

13.8 Conditional laws

lemma *cond-def*:
 $(P \triangleleft b \triangleright Q) = ((b \wedge P) \vee ((\neg b) \wedge Q))$
by (*pred-auto*)

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

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

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

lemma *cond-assoc*: $((P \triangleleft b \triangleright Q) \triangleleft c \triangleright R) = (P \triangleleft b \wedge c \triangleright (Q \triangleleft c \triangleright R))$ **by** (*pred-auto*)

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

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

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

lemma *cond-conj-not*: $((P \triangleleft b \triangleright Q) \wedge (\neg b)) = (Q \wedge (\neg b))$
by (*rel-auto*)

lemma *cond-and-T-integrate*:
 $((P \wedge b) \vee (Q \triangleleft b \triangleright R)) = ((P \vee Q) \triangleleft b \triangleright R)$
by (*pred-auto*)

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

lemma *cond-L7*: $(P \triangleleft b \triangleright (P \triangleleft c \triangleright Q)) = (P \triangleleft b \vee c \triangleright Q)$ **by** (*pred-auto*)

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

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

lemma *cond-imp-distr*:

$((P \Rightarrow Q) \triangleleft b \triangleright (R \Rightarrow S)) = ((P \triangleleft b \triangleright R) \Rightarrow (Q \triangleleft b \triangleright S))$ **by** (*pred-auto*)

lemma *cond-eq-distr*:

$((P \Leftrightarrow Q) \triangleleft b \triangleright (R \Leftrightarrow S)) = ((P \triangleleft b \triangleright R) \Leftrightarrow (Q \triangleleft b \triangleright S))$ **by** (*pred-auto*)

lemma *cond-conj-distr*: $(P \wedge (Q \triangleleft b \triangleright S)) = ((P \wedge Q) \triangleleft b \triangleright (P \wedge S))$ **by** (*pred-auto*)

lemma *cond-disj-distr*: $(P \vee (Q \triangleleft b \triangleright S)) = ((P \vee Q) \triangleleft b \triangleright (P \vee S))$ **by** (*pred-auto*)

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

lemma *cond-conj*: $P \triangleleft b \wedge c \triangleright Q = (P \triangleleft c \triangleright Q) \triangleleft b \triangleright Q$

by (*pred-auto*)

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

by (*pred-auto*)

lemma *cond-USUP-dist*: $(\bigsqcup P \in S \cdot F(P)) \triangleleft b \triangleright (\bigsqcup P \in S \cdot G(P)) = (\bigsqcup P \in S \cdot F(P) \triangleleft b \triangleright G(P))$

by (*pred-auto*)

lemma *cond-UINF-dist*: $(\bigsqcap P \in S \cdot F(P)) \triangleleft b \triangleright (\bigsqcap P \in S \cdot G(P)) = (\bigsqcap P \in S \cdot F(P) \triangleleft b \triangleright G(P))$

by (*pred-auto*)

lemma *cond-var-subst-left*:

assumes *vwb-lens* x

shows $(P \llbracket \text{true}/x \rrbracket \triangleleft \text{var } x \triangleright Q) = (P \triangleleft \text{var } x \triangleright Q)$

using *assms* **by** (*pred-auto*, *metis* (*full-types*) *vwb-lens-wb* *wb-lens.get-put*)

lemma *cond-var-subst-right*:

assumes *vwb-lens* x

shows $(P \triangleleft \text{var } x \triangleright Q \llbracket \text{false}/x \rrbracket) = (P \triangleleft \text{var } x \triangleright Q)$

using *assms* **by** (*pred-auto*, *metis* (*full-types*) *vwb-lens.put-eq*)

lemma *cond-var-split*:

vwb-lens $x \implies (P \llbracket \text{true}/x \rrbracket \triangleleft \text{var } x \triangleright P \llbracket \text{false}/x \rrbracket) = P$

by (*rel-simp*, (*metis* (*full-types*) *vwb-lens.put-eq*)+)

lemma *cond-assign-subst*:

vwb-lens $x \implies (P \triangleleft \text{utp-expr.var } x =_u v \triangleright Q) = (P \llbracket v/x \rrbracket \triangleleft \text{utp-expr.var } x =_u v \triangleright Q)$

apply (*rel-simp*) **using** *vwb-lens.put-eq* **by** *force*

lemma *conj-conds*:

$(P1 \triangleleft b \triangleright Q1 \wedge P2 \triangleleft b \triangleright Q2) = (P1 \wedge P2) \triangleleft b \triangleright (Q1 \wedge Q2)$

by *pred-auto*

lemma *disj-conds*:

$(P1 \triangleleft b \triangleright Q1 \vee P2 \triangleleft b \triangleright Q2) = (P1 \vee P2) \triangleleft b \triangleright (Q1 \vee Q2)$

by *pred-auto*

lemma *cond-mono*:

$\llbracket P1 \sqsubseteq P2; Q1 \sqsubseteq Q2 \rrbracket \implies (P1 \triangleleft b \triangleright Q1) \sqsubseteq (P2 \triangleleft b \triangleright Q2)$

by (*rel-auto*)

lemma *cond-monotonic*:
 $\llbracket \text{mono } P; \text{mono } Q \rrbracket \implies \text{mono } (\lambda X. P X \triangleleft b \triangleright Q X)$
by (*simp add: mono-def, rel-blast*)

13.9 Additional Expression Laws

lemma *le-pred-refl* [*simp*]:
fixes $x :: ('a::\text{preorder}, 'a) \text{ uexpr}$
shows $(x \leq_u x) = \text{true}$
by (*pred-auto*)

lemma *uzero-le-laws* [*simp*]:
 $(0 :: ('a::\{\text{linordered-semidom}\}, 'a) \text{ uexpr}) \leq_u \text{numeral } x = \text{true}$
 $(1 :: ('a::\{\text{linordered-semidom}\}, 'a) \text{ uexpr}) \leq_u \text{numeral } x = \text{true}$
 $(0 :: ('a::\{\text{linordered-semidom}\}, 'a) \text{ uexpr}) \leq_u 1 = \text{true}$
by (*pred-simp*)+

lemma *unumeral-le-1* [*simp*]:
assumes $(\text{numeral } i :: ('a::\{\text{numeral,ord}\}, 'a) \text{ uexpr}) \leq \text{numeral } j$
shows $(\text{numeral } i :: ('a, 'a) \text{ uexpr}) \leq_u \text{numeral } j = \text{true}$
using *assms* **by** (*pred-auto*)

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

lemma *uset-laws* [*simp*]:
 $x \in_u \{\} = \text{false}$
 $x \in_u \{m..n\} = (m \leq_u x \wedge x \leq_u n)$
by (*pred-auto*)+

lemma *ulit-eq* [*simp*]: $x = y \implies (\llbracket x \rrbracket =_u \llbracket y \rrbracket) = \text{true}$
by (*rel-auto*)

lemma *ulit-neq* [*simp*]: $x \neq y \implies (\llbracket x \rrbracket =_u \llbracket y \rrbracket) = \text{false}$
by (*rel-auto*)

lemma *uset-mems* [*simp*]:
 $x \in_u \{y\} = (x =_u y)$
 $x \in_u A \cup_u B = (x \in_u A \vee x \in_u B)$
 $x \in_u A \cap_u B = (x \in_u A \wedge x \in_u B)$
by (*rel-auto*)+

13.10 Refinement By Observation

Function to obtain the set of observations of a predicate

definition *obs-upred* :: $'a \text{ upred} \Rightarrow 'a \text{ set } (\llbracket \cdot \rrbracket_o)$
where [*upred-defs*]: $\llbracket P \rrbracket_o = \{b. \llbracket P \rrbracket_e b\}$

lemma *obs-upred-refine-iff*:
 $P \sqsubseteq Q \longleftrightarrow \llbracket Q \rrbracket_o \subseteq \llbracket P \rrbracket_o$
by (*pred-auto*)

A refinement can be demonstrated by considering only the observations of the predicates which

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

lemma *refine-by-obs*:

```

assumes  $x \bowtie y$  bij-lens  $(x +_L y) \not\# P \not\# Q$   $\{v. 'P \llbracket \llbracket v \rrbracket / x \rrbracket '\} \subseteq \{v. 'Q \llbracket \llbracket v \rrbracket / x \rrbracket '\}$ 
shows  $Q \sqsubseteq P$ 
using assms(3-5)
apply (simp add: obs-upred-refine-iff subset-eq)
apply (pred-simp)
apply (rename-tac b)
apply (drule-tac x=getxb in spec)
apply (auto simp add: assms)
apply (metis assms(1) assms(2) bij-lens.axioms(2) bij-lens.axioms-def lens-override-def lens-override-plus)+
done

```

13.11 Cylindric Algebra

lemma *C1*: $(\exists x \cdot \text{false}) = \text{false}$
by (*pred-auto*)

lemma *C2*: $\text{wb-lens } x \implies 'P \Rightarrow (\exists x \cdot P)'$
by (*pred-simp, metis wb-lens.get-put*)

lemma *C3*: $\text{mwb-lens } x \implies (\exists x \cdot (P \wedge (\exists x \cdot Q))) = ((\exists x \cdot P) \wedge (\exists x \cdot Q))$
by (*pred-auto*)

lemma *C4a*: $x \approx_L y \implies (\exists x \cdot \exists y \cdot P) = (\exists y \cdot \exists x \cdot P)$
by (*pred-simp, metis (no-types, lifting) lens.select-convs(2)*)+

lemma *C4b*: $x \bowtie y \implies (\exists x \cdot \exists y \cdot P) = (\exists y \cdot \exists x \cdot P)$
using *ex-commute* **by** *blast*

lemma *C5*:
fixes $x :: ('a \implies 'a)$
shows $(\&x =_u \&x) = \text{true}$
by (*pred-auto*)

lemma *C6*:
assumes $\text{wb-lens } x \bowtie y \bowtie z$
shows $(\&y =_u \&z) = (\exists x \cdot \&y =_u \&x \wedge \&x =_u \&z)$
using *assms*
by (*pred-simp, (metis lens-indep-def)*)+

lemma *C7*:
assumes $\text{weak-lens } x \bowtie y$
shows $((\exists x \cdot \&x =_u \&y \wedge P) \wedge (\exists x \cdot \&x =_u \&y \wedge \neg P)) = \text{false}$
using *assms*
by (*pred-simp, simp add: lens-indep-sym*)

end

14 Healthiness Conditions

theory *utp-healthy*

```

imports utp-pred-laws
begin

```

14.1 Main Definitions

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

```

named-theorems closure

```

```

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

```

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

```

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

```

```

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

```

```

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

```

```

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

```

```

declare Healthy-def' [upred-defs]

```

```

abbreviation Healthy-carrier :: 'α health ⇒ 'α upred set (⟦·⟧H)
where ⟦H⟧H ≡ {P. P is H}

```

```

lemma Healthy-carrier-image:
  A ⊆ ⟦H⟧H ⟹ H ' A = A
by (auto simp add: image-def, (metis Healthy-if mem-Collect-eq subsetCE)+)

```

```

lemma Healthy-carrier-Collect: A ⊆ ⟦H⟧H ⟹ A = {H(P) | P. P ∈ A}
by (simp add: Healthy-carrier-image Setcompr-eq-image)

```

```

lemma Healthy-func:
  ⟦ F ∈ ⟦H1⟧H → ⟦H2⟧H; P is H1 ⟧ ⟹ H2(F(P)) = F(P)
using Healthy-if by blast

```

```

lemma Healthy-comp:
  ⟦ P is H1; P is H2 ⟧ ⟹ P is H1 ∘ H2
by (simp add: Healthy-def)

```

```

lemma Healthy-apply-closed:
  assumes F ∈ ⟦H⟧H → ⟦H⟧H P is H
  shows F(P) is H
using assms(1) assms(2) by auto

```

```

lemma Healthy-set-image-member:
  ⟦ P ∈ F ' A; ∧ x. F x is H ⟧ ⟹ P is H
by blast

```

```

lemma Healthy-case-prod [closure]:
  ⟦ ∧ x y. P x y is H ⟧ ⟹ case-prod P v is H
by (simp add: prod.case-eq-if)

```

lemma *Healthy-SUPREMUM*:

$A \subseteq \llbracket H \rrbracket_H \implies \text{SUPREMUM } A \ H = \bigcap A$
by (*drule Healthy-carrier-image, presburger*)

lemma *Healthy-INFIMUM*:

$A \subseteq \llbracket H \rrbracket_H \implies \text{INFIMUM } A \ H = \bigcup A$
by (*drule Healthy-carrier-image, presburger*)

lemma *Healthy-nu [closure]*:

assumes *mono* $F \ F \in \llbracket id \rrbracket_H \rightarrow \llbracket H \rrbracket_H$
shows $\nu \ F \text{ is } H$
by (*metis (mono-tags) Healthy-def Healthy-func assms eq-id-iff lfp-unfold*)

lemma *Healthy-mu [closure]*:

assumes *mono* $F \ F \in \llbracket id \rrbracket_H \rightarrow \llbracket H \rrbracket_H$
shows $\mu \ F \text{ is } H$
by (*metis (mono-tags) Healthy-def Healthy-func assms eq-id-iff gfp-unfold*)

lemma *Healthy-subset-member*: $\llbracket A \subseteq \llbracket H \rrbracket_H; P \in A \rrbracket \implies H(P) = P$

by (*meson Ball-Collect Healthy-if*)

lemma *is-Healthy-subset-member*: $\llbracket A \subseteq \llbracket H \rrbracket_H; P \in A \rrbracket \implies P \text{ is } H$

by *blast*

14.2 Properties of Healthiness Conditions

definition *Idempotent* :: $'\alpha \text{ health} \Rightarrow \text{bool}$ **where**

$\text{Idempotent}(H) \longleftrightarrow (\forall P. H(H(P)) = H(P))$

abbreviation *Monotonic* :: $'\alpha \text{ health} \Rightarrow \text{bool}$ **where**

$\text{Monotonic}(H) \equiv \text{mono } H$

definition *IMH* :: $'\alpha \text{ health} \Rightarrow \text{bool}$ **where**

$\text{IMH}(H) \longleftrightarrow \text{Idempotent}(H) \wedge \text{Monotonic}(H)$

definition *Antitone* :: $'\alpha \text{ health} \Rightarrow \text{bool}$ **where**

$\text{Antitone}(H) \longleftrightarrow (\forall P \ Q. Q \sqsubseteq P \longrightarrow (H(P) \sqsubseteq H(Q)))$

definition *Conjunctive* :: $'\alpha \text{ health} \Rightarrow \text{bool}$ **where**

$\text{Conjunctive}(H) \longleftrightarrow (\exists Q. \forall P. H(P) = (P \wedge Q))$

definition *FunctionalConjunctive* :: $'\alpha \text{ health} \Rightarrow \text{bool}$ **where**

$\text{FunctionalConjunctive}(H) \longleftrightarrow (\exists F. \forall P. H(P) = (P \wedge F(P)) \wedge \text{Monotonic}(F))$

definition *WeakConjunctive* :: $'\alpha \text{ health} \Rightarrow \text{bool}$ **where**

$\text{WeakConjunctive}(H) \longleftrightarrow (\forall P. \exists Q. H(P) = (P \wedge Q))$

definition *Disjunctuous* :: $'\alpha \text{ health} \Rightarrow \text{bool}$ **where**

[*upred-defs*]: $\text{Disjunctuous } H = (\forall P \ Q. H(P \sqcap Q) = (H(P) \sqcap H(Q)))$

definition *Continuous* :: $'\alpha \text{ health} \Rightarrow \text{bool}$ **where**

[*upred-defs*]: $\text{Continuous } H = (\forall A. A \neq \{\} \longrightarrow H(\bigcap A) = \bigcap (H \restriction A))$

lemma *Healthy-Idempotent [closure]*:

$\text{Idempotent } H \implies H(P) \text{ is } H$

by (simp add: Healthy-def Idempotent-def)

lemma Healthy-range: $\text{Idempotent } H \implies \text{range } H = \llbracket H \rrbracket_H$
 by (auto simp add: image-def Healthy-if Healthy-Idempotent, metis Healthy-if)

lemma Idempotent-id [simp]: $\text{Idempotent } id$
 by (simp add: Idempotent-def)

lemma Idempotent-comp [intro]:
 $\llbracket \text{Idempotent } f; \text{Idempotent } g; f \circ g = g \circ f \rrbracket \implies \text{Idempotent } (f \circ g)$
 by (auto simp add: Idempotent-def comp-def, metis)

lemma Idempotent-image: $\text{Idempotent } f \implies f ' f ' A = f ' A$
 by (metis (mono-tags, lifting) Idempotent-def image-cong image-image)

lemma Monotonic-id [simp]: $\text{Monotonic } id$
 by (simp add: monoI)

lemma Monotonic-id' [closure]:
 $\text{mono } (\lambda X. X)$
 by (simp add: monoI)

lemma Monotonic-const [closure]:
 $\text{Monotonic } (\lambda x. c)$
 by (simp add: mono-def)

lemma Monotonic-comp [intro]:
 $\llbracket \text{Monotonic } f; \text{Monotonic } g \rrbracket \implies \text{Monotonic } (f \circ g)$
 by (simp add: mono-def)

lemma Monotonic-inf [closure]:
 assumes $\text{Monotonic } P \text{ Monotonic } Q$
 shows $\text{Monotonic } (\lambda X. P(X) \sqcap Q(X))$
 using assms by (simp add: mono-def, rel-auto)

lemma Monotonic-cond [closure]:
 assumes $\text{Monotonic } P \text{ Monotonic } Q$
 shows $\text{Monotonic } (\lambda X. P(X) \triangleleft b \triangleright Q(X))$
 by (simp add: assms cond-monotonic)

lemma Conjunctive-Idempotent:
 $\text{Conjunctive}(H) \implies \text{Idempotent}(H)$
 by (auto simp add: Conjunctive-def Idempotent-def)

lemma Conjunctive-Monotonic:
 $\text{Conjunctive}(H) \implies \text{Monotonic}(H)$
 unfolding Conjunctive-def mono-def
 using dual-order.trans by fastforce

lemma Conjunctive-conj:
 assumes $\text{Conjunctive}(HC)$
 shows $HC(P \wedge Q) = (HC(P) \wedge Q)$
 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 $HC(P \wedge Q) = (HC(P) \wedge HC(Q))$
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 $HC(P \vee Q) = (HC(P) \vee HC(Q))$
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) \implies *Monotonic*(H)
unfolding *FunctionalConjunctive-def* **by** (*metis mono-def utp-pred-laws.inf-mono*)

lemma *WeakConjunctive-Refinement*:
assumes *WeakConjunctive*(HC)
shows $P \sqsubseteq HC(P)$
using *assms* **unfolding** *WeakConjunctive-def* **by** (*metis utp-pred-laws.inf.cobounded1*)

lemma *WeakCojunctive-Healthy-Refinement*:
assumes *WeakConjunctive*(HC) **and** P is HC
shows $HC(P) \sqsubseteq P$
using *assms* **unfolding** *WeakConjunctive-def Healthy-def* **by** *simp*

lemma *WeakConjunctive-implies-WeakConjunctive*:
Conjunctive(H) \implies *WeakConjunctive*(H)
unfolding *WeakConjunctive-def Conjunctive-def* **by** *pred-auto*

declare *Conjunctive-def* [*upred-defs*]
declare *mono-def* [*upred-defs*]

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

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

lemma *Continuous-Disjunctous*: *Continuous* $H \implies$ *Disjunctuous* H
apply (*auto simp add: Continuous-def Disjunctuous-def*)
apply (*rename-tac P Q*)
apply (*drule-tac x={P,Q} in spec*)
apply (*simp*)
done

lemma *Continuous-Monotonic* [*closure*]: *Continuous* $H \implies$ *Monotonic* H
by (*simp add: Continuous-Disjunctous Disjunctuous-Monotonic*)

lemma *Continuous-comp* [intro]:
 $\llbracket \text{Continuous } f; \text{Continuous } g \rrbracket \implies \text{Continuous } (f \circ g)$
 by (simp add: Continuous-def)

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

lemma *Continuous-cond* [closure]:
 assumes *Continuous* F *Continuous* G
 shows *Continuous* $(\lambda X. F(X) \triangleleft b \triangleright G(X))$
 using assms by (pred-auto)

Closure laws derived from continuity

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

lemma *UINF-mem-Continuous-closed* [closure]:
 $\llbracket \text{Continuous } H; \bigwedge i. i \in A \implies P(i) \text{ is } H; A \neq \{\} \rrbracket \implies (\bigcap_{i \in A} P(i)) \text{ is } H$
 by (simp add: Sup-Continuous-closed UINF-as-Sup-collect)

lemma *UINF-mem-Continuous-closed-pair* [closure]:
 assumes *Continuous* $H \bigwedge i j. (i, j) \in A \implies P \ i \ j \text{ is } H \ A \neq \{\}$
 shows $(\bigcap_{(i,j) \in A} P \ i \ j) \text{ is } H$

proof –

have $(\bigcap_{(i,j) \in A} P \ i \ j) = (\bigcap_{x \in A} P \ (fst \ x) \ (snd \ x))$
 by (rel-auto)

also have ... is H

by (metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse)
 finally show ?thesis .

qed

lemma *UINF-mem-Continuous-closed-triple* [closure]:
 assumes *Continuous* $H \bigwedge i j k. (i, j, k) \in A \implies P \ i \ j \ k \text{ is } H \ A \neq \{\}$
 shows $(\bigcap_{(i,j,k) \in A} P \ i \ j \ k) \text{ is } H$

proof –

have $(\bigcap_{(i,j,k) \in A} P \ i \ j \ k) = (\bigcap_{x \in A} P \ (fst \ x) \ (fst \ (snd \ x)) \ (snd \ (snd \ x)))$
 by (rel-auto)

also have ... is H

by (metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse)
 finally show ?thesis .

qed

lemma *UINF-mem-Continuous-closed-quad* [closure]:
 assumes *Continuous* $H \bigwedge i j k l. (i, j, k, l) \in A \implies P \ i \ j \ k \ l \text{ is } H \ A \neq \{\}$
 shows $(\bigcap_{(i,j,k,l) \in A} P \ i \ j \ k \ l) \text{ is } H$

proof –

have $(\bigcap_{(i,j,k,l) \in A} P \ i \ j \ k \ l) = (\bigcap_{x \in A} P \ (fst \ x) \ (fst \ (snd \ x)) \ (fst \ (snd \ (snd \ x))) \ (snd \ (snd \ (snd \ x))))$
 by (rel-auto)

also have ... is H

by (metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse)
 finally show ?thesis .

qed

lemma *UINF-mem-Continuous-closed-quint* [closure]:

assumes *Continuous* $H \wedge i\ j\ k\ l\ m. (i, j, k, l, m) \in A \implies P\ i\ j\ k\ l\ m\ is\ H\ A \neq \{\}$

shows $(\bigcap (i,j,k,l,m) \in A \cdot P\ i\ j\ k\ l\ m)\ is\ H$

proof –

have $(\bigcap (i,j,k,l,m) \in A \cdot P\ i\ j\ k\ l\ m)$

$= (\bigcap x \in A \cdot P\ (fst\ x)\ (fst\ (snd\ x))\ (fst\ (snd\ (snd\ x)))\ (fst\ (snd\ (snd\ (snd\ x))))\ (snd\ (snd\ (snd\ (snd\ x))))\ (snd\ (snd\ (snd\ (snd\ (snd\ x)))))$

by (*rel-auto*)

also have ... *is H*

by (*metis* (*mono-tags*) *UINF-mem-Continuous-closed* *assms*(1) *assms*(2) *assms*(3) *prod.collapse*)

finally show *?thesis* .

qed

lemma *UINF-ind-closed* [closure]:

assumes *Continuous* $H \wedge i. P\ i = true \wedge i. Q\ i\ is\ H$

shows *UINF* $P\ Q\ is\ H$

proof –

from *assms*(2) **have** *UINF* $P\ Q = (\bigcap i \cdot Q\ i)$

by (*rel-auto*)

also have ... *is H*

using *UINF-mem-Continuous-closed*[*of H UNIV P*]

by (*simp add: Sup-Continuous-closed UINF-as-Sup-collect' assms*)

finally show *?thesis* .

qed

All continuous functions are also Scott-continuous

lemma *sup-continuous-Continuous* [closure]: *Continuous* $F \implies sup\text{-}continuous\ F$

by (*simp add: Continuous-def sup-continuous-def*)

lemma *USUP-healthy*: $A \subseteq \llbracket H \rrbracket_H \implies (\bigsqcup P \in A \cdot F(P)) = (\bigsqcup P \in A \cdot F(H(P)))$

by (*rule USUP-cong, simp add: Healthy-subset-member*)

lemma *UINF-healthy*: $A \subseteq \llbracket H \rrbracket_H \implies (\bigcap P \in A \cdot F(P)) = (\bigcap P \in A \cdot F(H(P)))$

by (*rule UINF-cong, simp add: Healthy-subset-member*)

end

15 Alphabetised Relations

theory *utp-rel*

imports

utp-pred-laws

utp-healthy

utp-lift

utp-tactics

begin

An alphabetised relation is simply a predicate whose state-space is a product type. In this theory we construct the core operators of the relational calculus, and prove a library of associated theorems, based on Chapters 2 and 5 of the UTP book [22].

15.1 Relational Alphabets

We set up convenient syntax to refer to the input and output parts of the alphabet, as is common in UTP. Since we are in a product space, these are simply the lenses fst_L and snd_L .

definition $in\alpha :: ('\alpha \Rightarrow '\alpha \times '\beta)$ **where**
 $[lens-defs]: in\alpha = fst_L$

definition $out\alpha :: ('\beta \Rightarrow '\alpha \times '\beta)$ **where**
 $[lens-defs]: out\alpha = snd_L$

lemma $in\alpha\text{-uvar}$ $[simp]: vwb\text{-lens } in\alpha$
by $(unfold\text{-locales}, auto simp add: in\alpha\text{-def})$

lemma $out\alpha\text{-uvar}$ $[simp]: vwb\text{-lens } out\alpha$
by $(unfold\text{-locales}, auto simp add: out\alpha\text{-def})$

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

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

lemma $drop\text{-pre}\text{-inv}$ $[simp]: \llbracket out\alpha \# p \rrbracket \Longrightarrow \lceil [p]_{<} \rceil_{<} = p$
by $(pred\text{-simp})$

lemma $usubst\text{-lookup}\text{-ivar}\text{-unrest}$ $[usubst]:$
 $in\alpha \# \sigma \Longrightarrow \langle \sigma \rangle_s (ivar\ x) = \x
by $(rel\text{-simp}, metis\ fstI)$

lemma $usubst\text{-lookup}\text{-ovar}\text{-unrest}$ $[usubst]:$
 $out\alpha \# \sigma \Longrightarrow \langle \sigma \rangle_s (ovar\ x) = \x'
by $(rel\text{-simp}, metis\ sndI)$

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

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

The following two functions lift a predicate substitution to a relational one.

abbreviation $usubst\text{-rel}\text{-lift} :: '\alpha\ usubst \Rightarrow (''\alpha \times '\beta)\ usubst\ (\lceil - \rceil_s)$ **where**
 $\lceil \sigma \rceil_s \equiv \sigma \oplus_s in\alpha$

abbreviation $usubst\text{-rel}\text{-drop} :: (''\alpha \times '\alpha)\ usubst \Rightarrow '\alpha\ usubst\ (\lfloor - \rfloor_s)$ **where**
 $\lfloor \sigma \rfloor_s \equiv \sigma \upharpoonright_s in\alpha$

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

lemma $alpha\text{-in}\text{-out}$:
 $\Sigma \approx_L in\alpha +_L out\alpha$
by $(simp add: fst\text{-snd}\text{-id}\text{-lens } in\alpha\text{-def } lens\text{-equiv}\text{-refl } out\alpha\text{-def})$

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 $'\alpha \text{ cond} = '\alpha \text{ upred}$
type-synonym $(''\alpha, '\beta) \text{ urel} = (''\alpha \times '\beta) \text{ upred}$
type-synonym $'\alpha \text{ hrel} = (''\alpha \times '\alpha) \text{ upred}$
type-synonym $('a, '\alpha) \text{ hexpr} = ('a, '\alpha \times '\alpha) \text{ uexpr}$

translations

$(\text{type}) (''\alpha, '\beta) \text{ urel} \leq (\text{type}) (''\alpha \times '\beta) \text{ upred}$

We set up some overloaded constants for sequential composition and the identity in case we want to overload their definitions later.

consts

$\text{useq} :: 'a \Rightarrow 'b \Rightarrow 'c \text{ (infixr ;; 61)}$
 $\text{uassigns} :: 'a \text{ usubst} \Rightarrow 'b \text{ (}\langle \cdot \rangle_a\text{)}$
 $\text{uskip} :: 'a \text{ (II)}$

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 $\lceil b \rceil_{<}$.

definition $\text{lift-rcond} (\lceil \cdot \rceil_{<})$ **where**

$[\text{upred-defs}]: \lceil b \rceil_{<} = \lceil b \rceil_{<}$

abbreviation

$\text{rcond} :: (''\alpha, '\beta) \text{ urel} \Rightarrow '\alpha \text{ cond} \Rightarrow (''\alpha, '\beta) \text{ urel} \Rightarrow (''\alpha, '\beta) \text{ urel}$
 $((\beta \cdot \triangleleft \cdot \triangleright_r / -) [52, 0, 53] 52)$
where $(P \triangleleft b \triangleright_r Q) \equiv (P \triangleleft \lceil b \rceil_{<} \triangleright Q)$

Sequential composition is heterogeneous, and simply requires that the output alphabet of the first matches then input alphabet of the second. We define it by lifting HOL's built-in relational composition operator $((O))$. Since this returns a set, the definition states that the state binding b is an element of this set.

lift-definition $\text{seqr} :: (''\alpha, '\beta) \text{ urel} \Rightarrow (''\beta, '\gamma) \text{ urel} \Rightarrow (''\alpha \times '\gamma) \text{ upred}$
is $\lambda P Q b. b \in (\{p. P p\} O \{q. Q q\})$.

ad hoc-overloading

useq seqr

We also set up a homogeneous sequential composition operator, and versions of *true* and *false* that are explicitly typed by a homogeneous alphabet.

abbreviation $\text{seqh} :: '\alpha \text{ hrel} \Rightarrow '\alpha \text{ hrel} \Rightarrow '\alpha \text{ hrel} \text{ (infixr ;;}_h\text{ 61)}$ **where**
 $\text{seqh } P Q \equiv (P ;; Q)$

abbreviation $\text{truer} :: '\alpha \text{ hrel} \text{ (true}_h\text{)}$ **where**

$\text{truer} \equiv \text{true}$

abbreviation $\text{falser} :: '\alpha \text{ hrel} \text{ (false}_h\text{)}$ **where**

$\text{falser} \equiv \text{false}$

We define the relational converse operator as an alphabet extrusion on the bijective lens swap_L that swaps the elements of the product state-space.

abbreviation $conv-r :: ('a, 'α \times 'β) \text{ uexpr} \Rightarrow ('a, 'β \times 'α) \text{ uexpr} \text{ (- [999] 999)}$
where $conv-r \ e \equiv e \oplus_p \text{ swap}_L$

Assignment is defined using substitutions, where latter defines what each variable should map to. This approach, which is originally due to Back [3], permits more general assignment expressions. The definition of the operator identifies the after state binding, b' , with the substitution function applied to the before state binding b .

lift-definition $assigns-r :: 'α \text{ usubst} \Rightarrow 'α \text{ hrel}$
is $\lambda \sigma \ (b, b'). \ b' = \sigma(b) \ .$

ad hoc-overloading
 $uassigns \ assigns-r$

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

definition $skip-r :: 'α \text{ hrel}$ **where**
 $[urel-defs]: \ skip-r = \ assigns-r \ id$

ad hoc-overloading
 $uskip \ skip-r$

Non-deterministic assignment, also known as “choose”, assigns an arbitrarily chosen value to the given variable

definition $nd-assign :: ('a \Rightarrow 'α) \Rightarrow 'α \text{ hrel}$ **where**
 $[urel-defs]: \ nd-assign \ x = (\bigcap \ v \cdot \ assigns-r \ [x \mapsto_s \ll v \gg])$

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

definition $seqr-iter :: 'a \text{ list} \Rightarrow ('a \Rightarrow 'b \text{ hrel}) \Rightarrow 'b \text{ hrel}$ **where**
 $[urel-defs]: \ seqr-iter \ xs \ P = \ foldr \ (\lambda \ i \ Q. \ P(i) ;; Q) \ xs \ II$

A singleton assignment simply applies a singleton substitution function, and similarly for a double assignment.

abbreviation $assign-r :: ('t \Rightarrow 'α) \Rightarrow ('t, 'α) \text{ uexpr} \Rightarrow 'α \text{ hrel}$
where $assign-r \ x \ v \equiv \langle [x \mapsto_s \ v] \rangle_a$

abbreviation $assign-2-r ::$
 $('t1 \Rightarrow 'α) \Rightarrow ('t2 \Rightarrow 'α) \Rightarrow ('t1, 'α) \text{ uexpr} \Rightarrow ('t2, 'α) \text{ uexpr} \Rightarrow 'α \text{ hrel}$
where $assign-2-r \ x \ y \ u \ v \equiv \ assigns-r \ [x \mapsto_s \ u, \ y \mapsto_s \ v]$

We also define the alphabetised skip operator that identifies all input and output variables in the given alphabet lens. All other variables are unrestricted. We also set up syntax for it.

definition $skip-ra :: ('β, 'α) \text{ lens} \Rightarrow 'α \text{ hrel}$ **where**
 $[urel-defs]: \ skip-ra \ v = (\$v' =_u \$v)$

Similarly, we define the alphabetised assignment operator.

definition $assigns-ra :: 'α \text{ usubst} \Rightarrow ('β, 'α) \text{ lens} \Rightarrow 'α \text{ hrel} \ (\langle \cdot \rangle_-)$ **where**
 $\langle \sigma \rangle_a = (\lceil \sigma \rceil_s \dagger \ skip-ra \ a)$

Assumptions (c^\top) and assertions (c_\perp) are encoded as conditionals. An assumption behaves like skip if the condition is true, and otherwise behaves like *false* (miracle). An assertion is the same, but yields *true*, which is an abort. They are the same as tests, as in Kleene Algebra

with Tests [24, 1] (KAT), which embeds a Boolean algebra into a Kleene algebra to represent conditions.

definition $rassume :: 'a \text{ upred} \Rightarrow 'a \text{ hrel}$ **where**
 $[urel-defs]: rassume\ c = II \triangleleft c \triangleright_r false$

definition $rassert :: 'a \text{ upred} \Rightarrow 'a \text{ hrel}$ **where**
 $[urel-defs]: rassert\ c = II \triangleleft c \triangleright_r true$

We also encode “naked” guarded commands [8, ?] by composing an assumption with a relation.

definition $rgcmd :: 'a \text{ upred} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel}$ **where**
 $[urel-defs]: rgcmd\ b\ P = (rassume\ b ;; P)$

We define two variants of while loops based on strongest and weakest fixed points. The former is *false* for an infinite loop, and the latter is *true*.

definition $while-top :: 'a \text{ cond} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel}$ **where**
 $[urel-defs]: while-top\ b\ P = (\nu\ X \cdot (P ;; X) \triangleleft b \triangleright_r II)$

definition $while-bot :: 'a \text{ cond} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel}$ **where**
 $[urel-defs]: while-bot\ b\ P = (\mu\ X \cdot (P ;; X) \triangleleft b \triangleright_r II)$

While loops with invariant decoration (cf. [1]) – partial correctness.

definition $while-inv :: 'a \text{ cond} \Rightarrow 'a \text{ cond} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel}$ **where**
 $[urel-defs]: while-inv\ b\ p\ S = while-top\ b\ S$

While loops with invariant decoration – total correctness.

definition $while-inv-bot :: 'a \text{ cond} \Rightarrow 'a \text{ cond} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel}$ **where**
 $[urel-defs]: while-inv-bot\ b\ p\ S = while-bot\ b\ S$

While loops with invariant and variant decorations – total correctness.

definition $while-vrt ::$
 $'a \text{ cond} \Rightarrow 'a \text{ cond} \Rightarrow (nat, 'a) \text{ uexpr} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel}$ **where**
 $[urel-defs]: while-vrt\ b\ p\ v\ S = while-bot\ b\ S$

syntax

$-uassume :: uexp \Rightarrow logic\ ([\cdot]^\top)$
 $-uassume :: uexp \Rightarrow logic\ (\cdot)$
 $-uassert :: uexp \Rightarrow logic\ (\{\cdot\}_\perp)$
 $-ugcmd :: uexp \Rightarrow logic \Rightarrow logic\ (- \longrightarrow_r - [55, 56] 55)$
 $-uwhile :: uexp \Rightarrow logic \Rightarrow logic\ (while^\top - do - od)$
 $-uwhile-top :: uexp \Rightarrow logic \Rightarrow logic\ (while - do - od)$
 $-uwhile-bot :: uexp \Rightarrow logic \Rightarrow logic\ (while_\perp - do - od)$
 $-uwhile-inv :: uexp \Rightarrow uexp \Rightarrow logic \Rightarrow logic\ (while - invr - do - od)$
 $-uwhile-inv-bot :: uexp \Rightarrow uexp \Rightarrow logic \Rightarrow logic\ (while_\perp - invr - do - od 71)$
 $-uwhile-vrt :: uexp \Rightarrow uexp \Rightarrow uexp \Rightarrow logic \Rightarrow logic\ (while - invr - vrt - do - od)$

translations

$-uassume\ b == CONST\ rassume\ b$
 $-uassert\ b == CONST\ rassert\ b$
 $-ugcmd\ b\ P == CONST\ rgcmd\ b\ P$
 $-uwhile\ b\ P == CONST\ while-top\ b\ P$
 $-uwhile-top\ b\ P == CONST\ while-top\ b\ P$
 $-uwhile-bot\ b\ P == CONST\ while-bot\ b\ P$
 $-uwhile-inv\ b\ p\ S == CONST\ while-inv\ b\ p\ S$
 $-uwhile-inv-bot\ b\ p\ S == CONST\ while-inv-bot\ b\ p\ S$

$-uwhile-vrt\ b\ p\ v\ S == CONST\ while-vrt\ b\ p\ v\ S$

We implement a poor man's version of alphabet restriction that hides a variable within a relation.

definition $rel-var-res :: 'a\ hrel \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a\ hrel$ (**infix** $\downarrow_\alpha\ 80$) **where**
 $[urel-defs]: P \downarrow_\alpha x = (\exists \$x \cdot \exists \$x' \cdot P)$

Alphabet extension and restriction add additional variables by the given lens in both their primed and unprimed versions.

definition $rel-aext :: 'b\ hrel \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a\ hrel$
where $[upred-defs]: rel-aext\ P\ a = P \oplus_p (a \times_L a)$

definition $rel-ares :: 'a\ hrel \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b\ hrel$
where $[upred-defs]: rel-ares\ P\ a = (P \downarrow_p (a \times a))$

We next describe frames and antiframes with the help of lenses. A frame states that P defines how variables in a changed, and all those outside of a remain the same. An antiframe describes the converse: all variables outside a are specified by P , and all those in remain the same. For more information please see [25].

definition $frame :: ('a \Rightarrow 'a) \Rightarrow 'a\ hrel \Rightarrow 'a\ hrel$ **where**
 $[urel-defs]: frame\ a\ P = (P \wedge \$v' =_u \$v \oplus \$v' \text{ on } \&a)$

definition $antiframe :: ('a \Rightarrow 'a) \Rightarrow 'a\ hrel \Rightarrow 'a\ hrel$ **where**
 $[urel-defs]: antiframe\ a\ P = (P \wedge \$v' =_u \$v' \oplus \$v \text{ on } \&a)$

Frame extension combines alphabet extension with the frame operator to both add additional variables and then frame those.

definition $rel-frext :: ('b \Rightarrow 'a) \Rightarrow 'b\ hrel \Rightarrow 'a\ hrel$ **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 $nameset :: ('a \Rightarrow 'a) \Rightarrow 'a\ hrel \Rightarrow 'a\ hrel$ **where**
 $[urel-defs]: nameset\ a\ P = (P \downarrow_v \{\$v, \$a'\})$

15.3 Syntax Translations

syntax

- Alternative traditional conditional syntax
- $-utp-if :: uexp \Rightarrow logic \Rightarrow logic \Rightarrow logic\ ((if_u\ (-)/\ then\ (-)/\ else\ (-))\ [0, 0, 71]\ 71)$
- Iterated sequential composition
- $-seqr-iter :: pttm \Rightarrow 'a\ list \Rightarrow 'a\ hrel \Rightarrow 'a\ hrel\ ((3;;\ -\ :\ -\ /\ -)\ [0, 0, 10]\ 10)$
- Single and multiple assignment
- $-assignment :: svids \Rightarrow uexprs \Rightarrow 'a\ hrel\ ('(-') := '(-'))$
- $-assignment :: svids \Rightarrow uexprs \Rightarrow 'a\ hrel\ (\mathbf{infixr}\ :=\ 62)$
- Non-deterministic assignment
- $-nd-assign :: svids \Rightarrow logic\ (-\ :=\ * [62]\ 62)$
- Substitution constructor
- $-mk-usubst :: svids \Rightarrow uexprs \Rightarrow 'a\ usubst$
- Alphabetised skip
- $-skip-ra :: salpha \Rightarrow logic\ (II.)$
- Frame
- $-frame :: salpha \Rightarrow logic \Rightarrow logic\ (-:[-]\ [99,0]\ 100)$

```

— Antiframe
-antiframe      :: salpha ⇒ logic ⇒ logic (·[-] [79,0] 80)
— Relational Alphabet Extension
-rel-aext :: logic ⇒ salpha ⇒ logic (infixl ⊕r 90)
— Relational Alphabet Restriction
-rel-ares :: logic ⇒ salpha ⇒ logic (infixl ⊢r 90)
— Frame Extension
-rel-frext :: salpha ⇒ logic ⇒ logic (·[-]+ [99,0] 100)
— Nameset
-nameset        :: salpha ⇒ logic ⇒ logic (ns - · - [0,999] 999)

```

translations

```

-utp-if b P Q => P ◁ b ▷r Q
;; x : l · P ⇒ (CONST segr-iter) l (λx. P)
-mk-usubst σ (-svid-unit x) v ⇒ σ(&x ↦s v)
-mk-usubst σ (-svid-list x xs) (-uexprs v vs) ⇒ (-mk-usubst (σ(&x ↦s v)) xs vs)
-assignment xs vs => CONST uassigns (-mk-usubst (CONST id) xs vs)
-assignment x v <= CONST uassigns (CONST subst-upd (CONST id) x v)
-assignment x v <= -assignment (-spvar x) v
-nd-assign x => CONST nd-assign (-mk-svid-list x)
-nd-assign x <= CONST nd-assign x
x,y := u,v <= CONST uassigns (CONST subst-upd (CONST subst-upd (CONST id) (CONST svar
x) u) (CONST svar y) v)
-skip-ra v ⇒ CONST skip-ra v
-frame x P => CONST frame x P
-frame (-salphaset (-salphamk x)) P <= CONST frame x P
-antiframe x P => CONST antiframe x P
-antiframe (-salphaset (-salphamk x)) P <= CONST antiframe x P
-nameset x P == CONST nameset x P
-rel-aext P a == CONST rel-aext P a
-rel-ares P a == CONST rel-ares P a
-rel-frext a P == CONST rel-frext a P

```

The following code sets up pretty-printing for homogeneous relational expressions. We cannot do this via the “translations” command as we only want the rule to apply when the input and output alphabet types are the same. The code has to deconstruct a (α, α) *ueexpr* type, determine that it is relational (product alphabet), and then checks if the types *alpha* and *beta* are the same. If they are, the type is printed as a *hexpr*. Otherwise, we have no match. We then set up a regular translation for the *hrel* type that uses this.

```

print-translation (
let
fun tr' ctx [ a
, Const (@{type-syntax prod},-) $ alpha $ beta ] =
if (alpha = beta)
then Syntax.const @{type-syntax hexpr} $ a $ alpha
else raise Match;
in [(@{type-syntax ueexpr},tr')]
end
)

```

translations

```

(type) 'α hrel <= (type) (bool, 'α) 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 \Rightarrow bool
where [*urel-defs*]: *ufunctional* $R \longleftrightarrow II \sqsubseteq R^- ;; R$

definition *winj* :: ('a, 'b) urel \Rightarrow bool
where [*urel-defs*]: *winj* $R \longleftrightarrow II \sqsubseteq R ;; R^-$

definition *Dom* :: 'α hrel \Rightarrow 'α upred
where [*upred-defs*]: *Dom* $P = [\exists \$\mathbf{v}' \cdot P]_<$

definition *Ran* :: 'α hrel \Rightarrow 'α upred
where [*upred-defs*]: *Ran* $P = [\exists \$\mathbf{v} \cdot P]_>$

— Configuration for UTP tactics.

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

15.5 Introduction laws

lemma *urel-refine-ext*:

$$[\bigwedge s s'. P[\llbracket \llbracket s \rrbracket, \llbracket s' \rrbracket \rrbracket / \$\mathbf{v}, \$\mathbf{v}']] \sqsubseteq Q[\llbracket \llbracket s \rrbracket, \llbracket s' \rrbracket \rrbracket / \$\mathbf{v}, \$\mathbf{v}']] \Longrightarrow P \sqsubseteq Q$$
by (*rel-auto*)

lemma *urel-eq-ext*:

$$[\bigwedge s s'. P[\llbracket \llbracket s \rrbracket, \llbracket s' \rrbracket \rrbracket / \$\mathbf{v}, \$\mathbf{v}']] = Q[\llbracket \llbracket s \rrbracket, \llbracket s' \rrbracket \rrbracket / \$\mathbf{v}, \$\mathbf{v}']] \Longrightarrow P = Q$$
by (*rel-auto*)

15.6 Unrestriction Laws

lemma *unrest-iuvar* [*unrest*]: $out\alpha \# \$x$
by (*metis fst-snd-lens-indep lift-pre-var outα-def unrest-aext-indep*)

lemma *unrest-ouvar* [*unrest*]: $in\alpha \# \$x'$
by (*metis inα-def lift-post-var snd-fst-lens-indep unrest-aext-indep*)

lemma *unrest-semir-undash* [*unrest*]:
fixes $x :: ('a \Longrightarrow 'α)$
assumes $\$x \# P$
shows $\$x \# P ;; Q$
using *assms* **by** (*rel-auto*)

lemma *unrest-semir-dash* [*unrest*]:
fixes $x :: ('a \Longrightarrow 'α)$
assumes $\$x' \# Q$
shows $\$x' \# P ;; Q$
using *assms* **by** (*rel-auto*)

lemma *unrest-cond* [*unrest*]:

$$[\$x \# P; \$x \# b; \$x \# Q] \Longrightarrow \$x \# P \triangleleft b \triangleright Q$$
by (*rel-auto*)

lemma *unrest-lift-rcond* [*unrest*]:

$x \# [b]_< \implies x \# [b]_{\leftarrow}$
by (*simp add: lift-rcond-def*)

lemma *unrest-in α -var* [*unrest*]:
 $\llbracket \text{mwb-lens } x; \text{in}\alpha \# (P :: ('a, ('\alpha \times '\beta)) \text{ uexpr}) \rrbracket \implies \$x \# P$
by (*rel-auto*)

lemma *unrest-out α -var* [*unrest*]:
 $\llbracket \text{mwb-lens } x; \text{out}\alpha \# (P :: ('a, ('\alpha \times '\beta)) \text{ uexpr}) \rrbracket \implies \$x' \# P$
by (*rel-auto*)

lemma *unrest-pre-out α* [*unrest*]: $\text{out}\alpha \# [b]_<$
by (*transfer, auto simp add: out α -def*)

lemma *unrest-post-in α* [*unrest*]: $\text{in}\alpha \# [b]_>$
by (*transfer, auto simp add: in α -def*)

lemma *unrest-pre-in-var* [*unrest*]:
 $x \# p1 \implies \$x \# [p1]_<$
by (*transfer, simp*)

lemma *unrest-post-out-var* [*unrest*]:
 $x \# p1 \implies \$x' \# [p1]_>$
by (*transfer, simp*)

lemma *unrest-convr-out α* [*unrest*]:
 $\text{in}\alpha \# p \implies \text{out}\alpha \# p^-$
by (*transfer, auto simp add: lens-defs*)

lemma *unrest-convr-in α* [*unrest*]:
 $\text{out}\alpha \# p \implies \text{in}\alpha \# p^-$
by (*transfer, auto simp add: lens-defs*)

lemma *unrest-in-rel-var-res* [*unrest*]:
 $\text{vwb-lens } x \implies \$x \# (P \upharpoonright_{\alpha} x)$
by (*simp add: rel-var-res-def unrest*)

lemma *unrest-out-rel-var-res* [*unrest*]:
 $\text{vwb-lens } x \implies \$x' \# (P \upharpoonright_{\alpha} x)$
by (*simp add: rel-var-res-def unrest*)

lemma *unrest-out-alpha-usubst-rel-lift* [*unrest*]:
 $\text{out}\alpha \# [\sigma]_s$
by (*rel-auto*)

lemma *unrest-in-rel-aext* [*unrest*]: $x \bowtie y \implies \$y \# P \oplus_r x$
by (*simp add: rel-aext-def unrest-aext-indep*)

lemma *unrest-out-rel-aext* [*unrest*]: $x \bowtie y \implies \$y' \# P \oplus_r x$
by (*simp add: rel-aext-def unrest-aext-indep*)

lemma *rel-aext-false* [*alpha*]:
 $\text{false} \oplus_r a = \text{false}$
by (*pred-auto*)

lemma *rel-aext-seq* [*alpha*]:
weak-lens $a \implies (P ;; Q) \oplus_r a = (P \oplus_r a ;; Q \oplus_r a)$
apply (*rel-auto*)
apply (*rename-tac* *aa* *b* *y*)
apply (*rule-tac* $x = \text{create}_a y$ **in** *exI*)
apply (*simp*)
done

lemma *rel-aext-cond* [*alpha*]:
 $(P \triangleleft b \triangleright_r Q) \oplus_r a = (P \oplus_r a \triangleleft b \oplus_r a \triangleright_r Q \oplus_r a)$
by (*rel-auto*)

15.7 Substitution laws

lemma *subst-seq-left* [*usubst*]:
 $\text{out}\alpha \# \sigma \implies \sigma \dagger (P ;; Q) = (\sigma \dagger P) ;; Q$
by (*rel-simp*, (*metis* (*no-types*, *lifting*) *Pair-inject surjective-pairing*)+)

lemma *subst-seq-right* [*usubst*]:
 $\text{in}\alpha \# \sigma \implies \sigma \dagger (P ;; Q) = P ;; (\sigma \dagger Q)$
by (*rel-simp*, (*metis* (*no-types*, *lifting*) *Pair-inject surjective-pairing*)+)

The following laws support substitution in heterogeneous relations for polymorphically typed literal expressions. These cannot be supported more generically due to limitations in HOL's type system. The laws are presented in a slightly strange way so as to be as general as possible.

lemma *bool-seqr-laws* [*usubst*]:
fixes $x :: (\text{bool} \implies 'a)$
shows
 $\bigwedge P Q \sigma. \sigma(\$x \mapsto_s \text{true}) \dagger (P ;; Q) = \sigma \dagger (P[\text{true}/\$x] ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x \mapsto_s \text{false}) \dagger (P ;; Q) = \sigma \dagger (P[\text{false}/\$x] ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s \text{true}) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[\text{true}/\$x'])$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s \text{false}) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[\text{false}/\$x'])$
by (*rel-auto*)+

lemma *zero-one-seqr-laws* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q \sigma. \sigma(\$x \mapsto_s 0) \dagger (P ;; Q) = \sigma \dagger (P[0/\$x] ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x \mapsto_s 1) \dagger (P ;; Q) = \sigma \dagger (P[1/\$x] ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s 0) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[0/\$x'])$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s 1) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[1/\$x'])$
by (*rel-auto*)+

lemma *numeral-seqr-laws* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q \sigma. \sigma(\$x \mapsto_s \text{numeral } n) \dagger (P ;; Q) = \sigma \dagger (P[\text{numeral } n/\$x] ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s \text{numeral } n) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[\text{numeral } n/\$x'])$
by (*rel-auto*)+

lemma *usubst-condr* [*usubst*]:
 $\sigma \dagger (P \triangleleft b \triangleright Q) = (\sigma \dagger P \triangleleft \sigma \dagger b \triangleright \sigma \dagger Q)$
by (*rel-auto*)

lemma *subst-skip-r* [*usubst*]:

$out\alpha \# \sigma \Longrightarrow \sigma \uparrow II = \langle \lfloor \sigma \rfloor_s \rangle_a$
by (*rel-simp*, (*metis* (*mono-tags*, *lifting*) *prod.sel*(1) *sndI* *surjective-pairing*)+)

lemma *subst-pre-skip* [*usubst*]: $\lfloor \sigma \rfloor_s \uparrow II = \langle \sigma \rangle_a$
by (*rel-auto*)

lemma *subst-rel-lift-seq* [*usubst*]:
 $\lfloor \sigma \rfloor_s \uparrow (P ;; Q) = (\lfloor \sigma \rfloor_s \uparrow P) ;; Q$
by (*rel-auto*)

lemma *subst-rel-lift-comp* [*usubst*]:
 $\lfloor \sigma \rfloor_s \circ \lfloor \varrho \rfloor_s = \lfloor \sigma \circ \varrho \rfloor_s$
by (*rel-auto*)

lemma *usubst-upd-in-comp* [*usubst*]:
 $\sigma(\&in\alpha:x \mapsto_s v) = \sigma(\$x \mapsto_s v)$
by (*simp* *add*: *pr-var-def* *fst-lens-def* *in α -def* *in-var-def*)

lemma *usubst-upd-out-comp* [*usubst*]:
 $\sigma(\&out\alpha:x \mapsto_s v) = \sigma(\$x' \mapsto_s v)$
by (*simp* *add*: *pr-var-def* *out α -def* *out-var-def* *snd-lens-def*)

lemma *subst-lift-upd* [*alpha*]:
fixes $x :: ('a \Longrightarrow 'a)$
shows $\lfloor \sigma(x \mapsto_s v) \rfloor_s = \lfloor \sigma \rfloor_s(\$x \mapsto_s \lfloor v \rfloor_<)$
by (*simp* *add*: *alpha* *usubst*, *simp* *add*: *pr-var-def* *fst-lens-def* *in α -def* *in-var-def*)

lemma *subst-drop-upd* [*alpha*]:
fixes $x :: ('a \Longrightarrow 'a)$
shows $\lfloor \sigma(\$x \mapsto_s v) \rfloor_s = \lfloor \sigma \rfloor_s(x \mapsto_s \lfloor v \rfloor_<)$
by *pred-simp*

lemma *subst-lift-pre* [*usubst*]: $\lfloor \sigma \rfloor_s \uparrow \lfloor b \rfloor_< = \lfloor \sigma \uparrow b \rfloor_<$
by (*metis* *apply-subst-ext* *fst-vwb-lens* *in α -def*)

lemma *unrest-usubst-lift-in* [*unrest*]:
 $x \# P \Longrightarrow \$x \# \lfloor P \rfloor_s$
by *pred-simp*

lemma *unrest-usubst-lift-out* [*unrest*]:
fixes $x :: ('a \Longrightarrow 'a)$
shows $\$x' \# \lfloor P \rfloor_s$
by *pred-simp*

lemma *subst-lift-cond* [*usubst*]: $\lfloor \sigma \rfloor_s \uparrow \lfloor s \rfloor_{\leftarrow} = \lfloor \sigma \uparrow s \rfloor_{\leftarrow}$
by (*rel-auto*)

lemma *msubst-seq* [*usubst*]: $(P(x) ;; Q(x))\llbracket x \rightarrow \ll v \gg \rrbracket = ((P(x))\llbracket x \rightarrow \ll v \gg \rrbracket ;; (Q(x))\llbracket x \rightarrow \ll v \gg \rrbracket)$
by (*rel-auto*)

15.8 Alphabet laws

lemma *aext-cond* [*alpha*]:
 $(P \triangleleft b \triangleright Q) \oplus_p a = ((P \oplus_p a) \triangleleft (b \oplus_p a) \triangleright (Q \oplus_p a))$
by (*rel-auto*)

lemma *aext-seq* [*alpha*]:
 $wb\text{-}lens\ a \implies ((P ;; Q) \oplus_p (a \times_L a)) = ((P \oplus_p (a \times_L a)) ;; (Q \oplus_p (a \times_L a)))$
by (*rel-simp*, *metis wb-lens-weak weak-lens.put-get*)

lemma *rcond-lift-true* [*simp*]:
 $[true]_{\leftarrow} = true$
by *rel-auto*

lemma *rcond-lift-false* [*simp*]:
 $[false]_{\leftarrow} = false$
by *rel-auto*

lemma *rel-ares-aext* [*alpha*]:
 $vwb\text{-}lens\ a \implies (P \oplus_r a) \upharpoonright_r a = P$
by (*rel-auto*)

lemma *rel-aext-ares* [*alpha*]:
 $\{\$a, \$a'\} \Vdash P \implies P \upharpoonright_r a \oplus_r a = P$
by (*rel-auto*)

lemma *rel-aext-uses* [*unrest*]:
 $vwb\text{-}lens\ a \implies \{\$a, \$a'\} \Vdash (P \oplus_r a)$
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* :: $('a \implies 'a) \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel}$
where *RID* $x\ P = ((\exists \$x \cdot \exists \$x' \cdot P) \wedge \$x' =_u \$x)$

declare *RID-def* [*urel-defs*]

lemma *RID1*: $vwb\text{-}lens\ x \implies (\forall v. x := \ll v \gg ;; P = P ;; x := \ll v \gg) \implies RID(x)(P) = P$
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*: $vwb\text{-}lens\ x \implies x := \ll v \gg ;; RID(x)(P) = RID(x)(P) ;; x := \ll v \gg$
apply (*rel-auto*)
apply (*metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens.get-put wb-lens-def weak-lens.put-get*)
apply *blast*
done

lemma *RID-assign-commute*:
 $vwb\text{-}lens\ x \implies P = RID(x)(P) \longleftrightarrow (\forall v. x := \ll v \gg ;; P = P ;; x := \ll v \gg)$
by (*metis RID1 RID2*)

lemma *RID-idem*:
 $mwb\text{-}lens\ x \implies RID(x)(RID(x)(P)) = RID(x)(P)$
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]:
 $RID (pr\text{-}var\ x) = RID\ x$
 by (simp add: pr-var-def)

lemma *RID-skip-r*:
 $vwb\text{-}lens\ x \implies RID(x)(II) = II$
 apply (rel-auto) using vwb-lens.put-eq by fastforce

lemma *skip-r-RID* [closure]: $vwb\text{-}lens\ x \implies II\ is\ RID(x)$
 by (simp add: Healthy-def RID-skip-r)

lemma *RID-disj*:
 $RID(x)(P \vee Q) = (RID(x)(P) \vee RID(x)(Q))$
 by (rel-auto)

lemma *disj-RID* [closure]: $\llbracket P\ is\ RID(x); Q\ is\ RID(x) \rrbracket \implies (P \vee Q)\ is\ RID(x)$
 by (simp add: Healthy-def RID-disj)

lemma *RID-conj*:
 $vwb\text{-}lens\ x \implies RID(x)(RID(x)(P) \wedge RID(x)(Q)) = (RID(x)(P) \wedge RID(x)(Q))$
 by (rel-auto)

lemma *conj-RID* [closure]: $\llbracket vwb\text{-}lens\ x; P\ is\ RID(x); Q\ is\ RID(x) \rrbracket \implies (P \wedge Q)\ is\ RID(x)$
 by (metis Healthy-if Healthy-intro RID-conj)

lemma *RID-assigns-r-diff*:
 $\llbracket vwb\text{-}lens\ x; x \# \sigma \rrbracket \implies RID(x)(\langle \sigma \rangle_a) = \langle \sigma \rangle_a$
 apply (rel-auto)
 apply (metis vwb-lens.put-eq)
 apply (metis vwb-lens-wb wb-lens.get-put wb-lens-weak weak-lens.put-get)
 done

lemma *assigns-r-RID* [closure]: $\llbracket vwb\text{-}lens\ x; x \# \sigma \rrbracket \implies \langle \sigma \rangle_a\ is\ RID(x)$
 by (simp add: Healthy-def RID-assigns-r-diff)

lemma *RID-assign-r-same*:
 $vwb\text{-}lens\ x \implies RID(x)(x := v) = II$
 apply (rel-auto)
 using vwb-lens.put-eq apply fastforce
 done

lemma *RID-seq-left*:
 assumes $vwb\text{-}lens\ x$
 shows $RID(x)(RID(x)(P) ;; Q) = (RID(x)(P) ;; RID(x)(Q))$
 proof -
 have $RID(x)(RID(x)(P) ;; Q) = ((\exists \$x \cdot \exists \$x' \cdot ((\exists \$x \cdot \exists \$x' \cdot P) \wedge \$x' =_u \$x) ;; Q) \wedge \$x' =_u \$x)$
 by (simp add: RID-def usubst)
 also from assms have $\dots = (((\exists \$x \cdot \exists \$x' \cdot P) \wedge (\exists \$x \cdot \$x' =_u \$x)) ;; (\exists \$x' \cdot Q)) \wedge \$x' =_u \$x$
 by (rel-auto)
 also from assms have $\dots = (((\exists \$x \cdot \exists \$x' \cdot P) ;; (\exists \$x \cdot \exists \$x' \cdot Q)) \wedge \$x' =_u \$x)$
 apply (rel-auto)

```

    apply (metis vwb-lens.put-eq)
  apply (metis mwb-lens.put-put vwb-lens-mwb)
done
also from assms have ... = ((( $\exists x \cdot \exists x' \cdot P$ )  $\wedge$   $x' =_u x$ ) ;; ( $\exists x \cdot \exists x' \cdot Q$ ))  $\wedge$   $x' =_u x$ )
  by (rel-simp, metis (full-types) mwb-lens.put-put vwb-lens-def wb-lens-weak weak-lens.put-get)
also have ... = ((( $\exists x \cdot \exists x' \cdot P$ )  $\wedge$   $x' =_u x$ ) ;; (( $\exists x \cdot \exists x' \cdot Q$ )  $\wedge$   $x' =_u x$ ))  $\wedge$   $x' =_u x$ )
  by (rel-simp, fastforce)
also have ... = ((( $\exists x \cdot \exists x' \cdot P$ )  $\wedge$   $x' =_u x$ ) ;; (( $\exists x \cdot \exists x' \cdot Q$ )  $\wedge$   $x' =_u x$ )))
  by (rel-auto)
also have ... = ( $RID(x)(P)$  ;;  $RID(x)(Q)$ )
  by (rel-auto)
finally show ?thesis .
qed

```

lemma *RID-seq-right*:

```

  assumes vwb-lens x
  shows  $RID(x)(P$  ;;  $RID(x)(Q)) = (RID(x)(P)$  ;;  $RID(x)(Q))$ 
proof -
  have  $RID(x)(P$  ;;  $RID(x)(Q)) = ((\exists x \cdot \exists x' \cdot P$  ;; (( $\exists x \cdot \exists x' \cdot Q$ )  $\wedge$   $x' =_u x$ ))  $\wedge$   $x' =_u x$ )
    by (simp add: RID-def usubst)
  also from assms have ... = ((( $\exists x \cdot P$ ) ;; ( $\exists x \cdot \exists x' \cdot Q$ )  $\wedge$  ( $\exists x' \cdot x' =_u x$ ))  $\wedge$   $x' =_u x$ )
    by (rel-auto)
  also from assms have ... = ((( $\exists x \cdot \exists x' \cdot P$ ) ;; ( $\exists x \cdot \exists x' \cdot Q$ ))  $\wedge$   $x' =_u x$ )
    apply (rel-auto)
    apply (metis vwb-lens.put-eq)
    apply (metis mwb-lens.put-put vwb-lens-mwb)
    done
  also from assms have ... = (((( $\exists x \cdot \exists x' \cdot P$ )  $\wedge$   $x' =_u x$ ) ;; ( $\exists x \cdot \exists x' \cdot Q$ ))  $\wedge$   $x' =_u x$ )
    by (rel-simp robust, metis (full-types) mwb-lens.put-put vwb-lens-def wb-lens-weak weak-lens.put-get)
  also have ... = (((( $\exists x \cdot \exists x' \cdot P$ )  $\wedge$   $x' =_u x$ ) ;; (( $\exists x \cdot \exists x' \cdot Q$ )  $\wedge$   $x' =_u x$ ))  $\wedge$   $x' =_u x$ )
    by (rel-simp, fastforce)
  also have ... = (((( $\exists x \cdot \exists x' \cdot P$ )  $\wedge$   $x' =_u x$ ) ;; (( $\exists x \cdot \exists x' \cdot Q$ )  $\wedge$   $x' =_u x$ )))
    by (rel-auto)
  also have ... = ( $RID(x)(P)$  ;;  $RID(x)(Q)$ )
    by (rel-auto)
  finally show ?thesis .
qed

```

lemma *seqr-RID-closed* [closure]: $\llbracket vwb\text{-}lens\ x; P\ is\ RID(x); Q\ is\ RID(x) \rrbracket \implies P\ ;;\ Q\ is\ RID(x)$
 by (metis Healthy-def RID-seq-right)

definition *unrest-relation* :: $(a \implies \alpha) \Rightarrow \alpha\ hrel \Rightarrow bool$ (**infix** $\#\#$ 20)
 where $(x\ \#\# P) \longleftrightarrow (P\ is\ RID(x))$

declare *unrest-relation-def* [urel-defs]

lemma *runrest-assign-commute*:

```

 $\llbracket vwb\text{-}lens\ x; x\ \#\# P \rrbracket \implies x := \llbracket v \rrbracket ;; P = P ;; x := \llbracket v \rrbracket$ 
  by (metis RID2 Healthy-def unrest-relation-def)

```

lemma *runrest-ident-var*:

```

assumes  $x \# \# P$ 
shows  $(\$x \wedge P) = (P \wedge \$x')$ 
proof –
  have  $P = (\$x' =_u \$x \wedge P)$ 
  by (metis RID-def assms Healthy-def unrest-relation-def utp-pred-laws.inf.cobounded2 utp-pred-laws.inf-absorb2)
moreover have  $(\$x' =_u \$x \wedge (\$x \wedge P)) = (\$x' =_u \$x \wedge (P \wedge \$x'))$ 
  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  $x \implies x \# \# II$ 
by (simp add: unrest-relation-def closure)

```

```

lemma assigns-r-runrest:
   $\llbracket \text{vwb-lens } x; x \# \sigma \rrbracket \implies x \# \# \langle \sigma \rangle_a$ 
by (simp add: unrest-relation-def closure)

```

```

lemma seq-r-runrest [unrest]:
  assumes vwb-lens  $x \# \# P \# \# Q$ 
shows  $x \# \# (P ;; Q)$ 
using assms by (simp add: unrest-relation-def closure)

```

```

lemma false-runrest [unrest]:  $x \# \# \text{false}$ 
by (rel-auto)

```

```

lemma and-runrest [unrest]:  $\llbracket \text{vwb-lens } x; x \# \# P; x \# \# Q \rrbracket \implies x \# \# (P \wedge Q)$ 
by (metis RID-conj Healthy-def unrest-relation-def)

```

```

lemma or-runrest [unrest]:  $\llbracket x \# \# P; x \# \# Q \rrbracket \implies x \# \# (P \vee Q)$ 
by (simp add: RID-disj Healthy-def unrest-relation-def)

```

end

16 Fixed-points and Recursion

```

theory utp-recursion
imports
  utp-pred-laws
  utp-rel
begin

```

16.1 Fixed-point Laws

```

lemma mu-id:  $(\mu X \cdot X) = \text{true}$ 
by (simp add: antisym gfp-upperbound)

```

```

lemma mu-const:  $(\mu X \cdot P) = P$ 
by (simp add: gfp-const)

```

```

lemma nu-id:  $(\nu X \cdot X) = \text{false}$ 
by (meson lfp-lowerbound utp-pred-laws.bot.extremum-unique)

```

```

lemma nu-const:  $(\nu X \cdot P) = P$ 

```

by (simp add: lfp-const)

lemma *mu-refine-intro*:
 assumes $(C \Rightarrow S) \sqsubseteq F(C \Rightarrow S) \ (C \wedge \mu F) = (C \wedge \nu F)$
 shows $(C \Rightarrow S) \sqsubseteq \mu F$
proof –
 from *assms* have $(C \Rightarrow S) \sqsubseteq \nu F$
 by (simp add: lfp-lowerbound)
 with *assms* show ?thesis
 by (pred-auto)
qed

16.2 Obtaining Unique Fixed-points

Obtaining termination proofs via approximation chains. Theorems and proofs adapted from Chapter 2, page 63 of the UTP book [22].

type-synonym 'a chain = nat \Rightarrow 'a upred

definition *chain* :: 'a chain \Rightarrow bool **where**
chain Y = ((Y 0 = false) \wedge (\forall i. Y (Suc i) \sqsubseteq Y i))

lemma *chain0* [simp]: *chain* Y \implies Y 0 = false
 by (simp add: chain-def)

lemma *chainI*:
 assumes Y 0 = false \wedge i. Y (Suc i) \sqsubseteq Y i
 shows *chain* Y
 using *assms* by (auto simp add: chain-def)

lemma *chainE*:
 assumes *chain* Y \wedge i. \llbracket Y 0 = false; Y (Suc i) \sqsubseteq Y i $\rrbracket \implies P$
 shows P
 using *assms* by (simp add: chain-def)

lemma *L274*:
 assumes $\forall n. ((E\ n \wedge_p X) = (E\ n \wedge Y))$
 shows $(\bigcap (\text{range } E) \wedge X) = (\bigcap (\text{range } E) \wedge Y)$
 using *assms* by (pred-auto)

Constructive chains

definition *constr* ::
 ('a upred \Rightarrow 'a upred) \Rightarrow 'a chain \Rightarrow bool **where**
constr F E \longleftrightarrow *chain* E \wedge (\forall X n. ((F(X) \wedge E(n + 1)) = (F(X \wedge E(n)) \wedge E (n + 1))))

lemma *constrI*:
 assumes *chain* E \wedge X n. ((F(X) \wedge E(n + 1)) = (F(X \wedge E(n)) \wedge E (n + 1)))
 shows *constr* F E
 using *assms* by (auto simp add: constr-def)

This lemma gives a way of showing that there is a unique fixed-point when the predicate function can be built using a constructive function F over an approximation chain E

lemma *chain-pred-terminates*:
 assumes *constr* F E mono F
 shows $(\bigcap (\text{range } E) \wedge \mu F) = (\bigcap (\text{range } E) \wedge \nu F)$


```

proof –
  from assms have  $\forall n. (E\ n \wedge \mu\ F) = (E\ n \wedge \nu\ F)$ 
  proof (rule-tac allI)
    fix n
    from assms show  $(E\ n \wedge \mu\ F) = (E\ n \wedge \nu\ F)$ 
    proof (induct n)
      case 0 thus ?case by (simp add: constr-def)
    next
      case (Suc n)
      note hyp = this
      thus ?case
      proof –
        have  $(E\ (n + 1) \wedge \mu\ F) = (E\ (n + 1) \wedge F\ (\mu\ F))$ 
        using gfp-unfold[OF hyp(3), THEN sym] by (simp add: constr-def)
        also from hyp have  $\dots = (E\ (n + 1) \wedge F\ (E\ n \wedge \mu\ F))$ 
        by (metis conj-comm constr-def)
        also from hyp have  $\dots = (E\ (n + 1) \wedge F\ (E\ n \wedge \nu\ F))$ 
        by simp
        also from hyp have  $\dots = (E\ (n + 1) \wedge \nu\ F)$ 
        by (metis (no-types, lifting) conj-comm constr-def lfp-unfold)
        ultimately show ?thesis
        by simp
      qed
    qed
  qed
  thus ?thesis
  by (auto intro: L274)
qed

```

```

theorem constr-fp-uniq:
  assumes constr F E mono F  $\sqcap$  (range E) = C
  shows  $(C \wedge \mu\ F) = (C \wedge \nu\ F)$ 
  using 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-uinduct-pure-ueq-gen:
  assumes fixp-unfold: fp B = B (fp B)
  and WF: wf R
  and induct-step:
     $\bigwedge f\ st. [\bigwedge st'. (st', st) \in R \implies (((Pre \wedge [e]_{<=u} \ll st' \gg) \Rightarrow Post) \sqsubseteq f)]$ 
     $\implies fp\ B = f \implies ((Pre \wedge [e]_{<=u} \ll st \gg) \Rightarrow Post) \sqsubseteq (B\ f)$ 
  shows  $((Pre \Rightarrow Post) \sqsubseteq fp\ B)$ 
proof –
  { fix st
    have  $((Pre \wedge [e]_{<=u} \ll st \gg) \Rightarrow Post) \sqsubseteq (fp\ B)$ 
    using WF proof (induction rule: wf-induct-rule)
      case (less x)
      hence  $(Pre \wedge [e]_{<=u} \ll x \gg \Rightarrow Post) \sqsubseteq B\ (fp\ B)$ 
      by (rule induct-step, rel-blast, simp)
      then show ?case
      using fixp-unfold by auto
    }
  qed

```



```

assumes WF: wf R
and      M: mono B
and      induct-step:
   $\bigwedge st. (Pre \wedge [e]_{< =_u \ll st \gg} \Rightarrow Post) \sqsubseteq (B ((Pre \wedge ([e]_{<, \ll st \gg})_u \in_u \ll R \gg) \Rightarrow Post)))$ 
shows  $(Pre \Rightarrow Post) \sqsubseteq_{\mu} B$ 
proof (rule mu-rec-total-pure-rule[where R=R and e=e], simp-all add: assms)
  show  $\bigwedge f st. (Pre \wedge ([e]_{<, \ll st \gg})_u \in_u \ll R \gg \Rightarrow Post) \sqsubseteq f \Longrightarrow \mu B = f \Longrightarrow (Pre \wedge [e]_{< =_u \ll st \gg} \Rightarrow Post) \sqsubseteq B f$ 
  by (simp add: M induct-step monoD order-subst2)
qed

```

lemma *nu-rec-total-utp-rule*:

```

assumes WF: wf R
and      M: mono B
and      induct-step:
   $\bigwedge st. (Pre \wedge [e]_{< =_u \ll st \gg} \Rightarrow Post) \sqsubseteq (B ((Pre \wedge ([e]_{<, \ll st \gg})_u \in_u \ll R \gg) \Rightarrow Post)))$ 
shows  $(Pre \Rightarrow Post) \sqsubseteq_{\nu} B$ 
proof (rule nu-rec-total-pure-rule[where R=R and e=e], simp-all add: assms)
  show  $\bigwedge f st. (Pre \wedge ([e]_{<, \ll st \gg})_u \in_u \ll R \gg \Rightarrow Post) \sqsubseteq f \Longrightarrow \nu B = f \Longrightarrow (Pre \wedge [e]_{< =_u \ll st \gg} \Rightarrow Post) \sqsubseteq B f$ 
  by (simp add: M induct-step monoD order-subst2)
qed

```

end

17 Sequent Calculus

theory *utp-sequent*

imports *utp-pred-laws*

begin

definition *sequent* :: ' α *upred* \Rightarrow ' α *upred* \Rightarrow bool (**infixr** \Vdash 15) **where**
[upred-defs]: *sequent P Q* = (*Q* \sqsubseteq *P*)

abbreviation *sequent-triv* (\Vdash - [15] 15) **where** $\Vdash P \equiv (true \Vdash P)$

translations

$\Vdash P <= true \Vdash P$

lemma *sTrue*: $P \Vdash true$

by *pred-auto*

lemma *sAx*: $P \Vdash P$

by *pred-auto*

lemma *sNotI*: $\Gamma \wedge P \Vdash false \Longrightarrow \Gamma \Vdash \neg P$

by *pred-auto*

lemma *sConjI*: $\llbracket \Gamma \Vdash P; \Gamma \Vdash Q \rrbracket \Longrightarrow \Gamma \Vdash P \wedge Q$

by *pred-auto*

lemma *sImplI*: $\llbracket (\Gamma \wedge P) \Vdash Q \rrbracket \Longrightarrow \Gamma \Vdash (P \Rightarrow Q)$

by *pred-auto*

end

18 Relational Calculus Laws

```

theory utp-rel-laws
imports
  utp-rel
  utp-recursion
begin

```

18.1 Conditional Laws

```

lemma comp-cond-left-distr:
   $((P \triangleleft b \triangleright_r Q) ;; R) = ((P ;; R) \triangleleft b \triangleright_r (Q ;; R))$ 
by (rel-auto)

```

```

lemma cond-seq-left-distr:
   $out\alpha \# b \implies ((P \triangleleft b \triangleright Q) ;; R) = ((P ;; R) \triangleleft b \triangleright (Q ;; R))$ 
by (rel-auto)

```

```

lemma cond-seq-right-distr:
   $in\alpha \# b \implies (P ;; (Q \triangleleft b \triangleright R)) = ((P ;; Q) \triangleleft b \triangleright (P ;; R))$ 
by (rel-auto)

```

Alternative expression of conditional using assumptions and choice

```

lemma rcond-rassume-expand:  $P \triangleleft b \triangleright_r Q = ([b]^\top ;; P) \sqcap [(\neg b)]^\top ;; Q$ 
by (rel-auto)

```

18.2 Precondition and Postcondition Laws

```

theorem precond-equiv:
   $P = (P ;; true) \longleftrightarrow (out\alpha \# P)$ 
by (rel-auto)

```

```

theorem postcond-equiv:
   $P = (true ;; P) \longleftrightarrow (in\alpha \# P)$ 
by (rel-auto)

```

```

lemma precond-right-unit:  $out\alpha \# p \implies (p ;; true) = p$ 
by (metis precond-equiv)

```

```

lemma postcond-left-unit:  $in\alpha \# p \implies (true ;; p) = p$ 
by (metis postcond-equiv)

```

```

theorem precond-left-zero:
  assumes  $out\alpha \# p \neq false$ 
  shows  $(true ;; p) = true$ 
  using assms by (rel-auto)

```

```

theorem feasible-iff-true-right-zero:
   $P ;; true = true \longleftrightarrow \exists out\alpha \cdot P$ 
by (rel-auto)

```

18.3 Sequential Composition Laws

```

lemma seqr-assoc:  $(P ;; Q) ;; R = P ;; (Q ;; R)$ 
by (rel-auto)

```

lemma *seqr-left-unit* [*simp*]:
 $II ;; P = P$
by (*rel-auto*)

lemma *seqr-right-unit* [*simp*]:
 $P ;; II = P$
by (*rel-auto*)

lemma *seqr-left-zero* [*simp*]:
 $false ;; P = false$
by *pred-auto*

lemma *seqr-right-zero* [*simp*]:
 $P ;; false = false$
by *pred-auto*

lemma *impl-seqr-mono*: $\llbracket 'P \Rightarrow Q'; 'R \Rightarrow S' \rrbracket \Longrightarrow '(P ;; R) \Rightarrow (Q ;; S)'$
by (*pred-blast*)

lemma *seqr-mono*:
 $\llbracket P_1 \sqsubseteq P_2; Q_1 \sqsubseteq Q_2 \rrbracket \Longrightarrow (P_1 ;; Q_1) \sqsubseteq (P_2 ;; Q_2)$
by (*rel-blast*)

lemma *seqr-monotonic*:
 $\llbracket mono\ P; mono\ Q \rrbracket \Longrightarrow mono\ (\lambda X. P\ X ;; Q\ X)$
by (*simp add: mono-def, rel-blast*)

lemma *Monotonic-seqr-tail* [*closure*]:
assumes *Monotonic F*
shows *Monotonic* $(\lambda X. P ;; F(X))$
by (*simp add: assms monoD monoI seqr-mono*)

lemma *seqr-exists-left*:
 $((\exists \$x \cdot P) ;; Q) = (\exists \$x \cdot (P ;; Q))$
by (*rel-auto*)

lemma *seqr-exists-right*:
 $(P ;; (\exists \$x' \cdot Q)) = (\exists \$x' \cdot (P ;; Q))$
by (*rel-auto*)

lemma *seqr-or-distl*:
 $((P \vee Q) ;; R) = ((P ;; R) \vee (Q ;; R))$
by (*rel-auto*)

lemma *seqr-or-distr*:
 $(P ;; (Q \vee R)) = ((P ;; Q) \vee (P ;; R))$
by (*rel-auto*)

lemma *seqr-inf-distl*:
 $((P \sqcap Q) ;; R) = ((P ;; R) \sqcap (Q ;; R))$
by (*rel-auto*)

lemma *seqr-inf-distr*:
 $(P ;; (Q \sqcap R)) = ((P ;; Q) \sqcap (P ;; R))$
by (*rel-auto*)

lemma *seqr-and-distr-ufunc*:

ufunctional $P \implies (P ;; (Q \wedge R)) = ((P ;; Q) \wedge (P ;; R))$
by (*rel-auto*)

lemma *seqr-and-distl-winj*:

winj $R \implies ((P \wedge Q) ;; R) = ((P ;; R) \wedge (Q ;; R))$
by (*rel-auto*)

lemma *seqr-unfold*:

$(P ;; Q) = (\exists v \cdot P[\llbracket \langle v \rangle / \$v' \rrbracket] \wedge Q[\llbracket \langle v \rangle / \$v \rrbracket])$
by (*rel-auto*)

lemma *seqr-middle*:

assumes *vwb-lens* x
shows $(P ;; Q) = (\exists v \cdot P[\llbracket \langle v \rangle / \$x' \rrbracket] ;; Q[\llbracket \langle v \rangle / \$x \rrbracket])$
using *assms*
by (*rel-auto'*, *metis vwb-lens-wb wb-lens.source-stability*)

lemma *seqr-left-one-point*:

assumes *vwb-lens* x
shows $((P \wedge \$x' =_u \langle v \rangle) ;; Q) = (P[\llbracket \langle v \rangle / \$x' \rrbracket] ;; Q[\llbracket \langle v \rangle / \$x \rrbracket])$
using *assms*
by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-right-one-point*:

assumes *vwb-lens* x
shows $(P ;; (\$x =_u \langle v \rangle \wedge Q)) = (P[\llbracket \langle v \rangle / \$x' \rrbracket] ;; Q[\llbracket \langle v \rangle / \$x \rrbracket])$
using *assms*
by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-left-one-point-true*:

assumes *vwb-lens* x
shows $((P \wedge \$x') ;; Q) = (P[\llbracket \text{true} / \$x' \rrbracket] ;; Q[\llbracket \text{true} / \$x \rrbracket])$
by (*metis assms seqr-left-one-point true-alt-def upred-eq-true*)

lemma *seqr-left-one-point-false*:

assumes *vwb-lens* x
shows $((P \wedge \neg \$x') ;; Q) = (P[\llbracket \text{false} / \$x' \rrbracket] ;; Q[\llbracket \text{false} / \$x \rrbracket])$
by (*metis assms false-alt-def seqr-left-one-point upred-eq-false*)

lemma *seqr-right-one-point-true*:

assumes *vwb-lens* x
shows $(P ;; (\$x \wedge Q)) = (P[\llbracket \text{true} / \$x' \rrbracket] ;; Q[\llbracket \text{true} / \$x \rrbracket])$
by (*metis assms seqr-right-one-point true-alt-def upred-eq-true*)

lemma *seqr-right-one-point-false*:

assumes *vwb-lens* x
shows $(P ;; (\neg \$x \wedge Q)) = (P[\llbracket \text{false} / \$x' \rrbracket] ;; Q[\llbracket \text{false} / \$x \rrbracket])$
by (*metis assms false-alt-def seqr-right-one-point upred-eq-false*)

lemma *seqr-insert-ident-left*:

assumes *vwb-lens* x $\$x' \# P$ $\$x \# Q$
shows $((\$x' =_u \$x \wedge P) ;; Q) = (P ;; Q)$
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 $\$x'$ $\#$ P $\$x$ $\#$ Q

shows $(P ;; (\$x' =_u \$x \wedge Q)) = (P ;; Q)$

using *assms*

by (rel-simp, metis (no-types, hide-lams) vwb-lens-def wb-lens-def weak-lens.put-get)

lemma *seqr-var-ident-lift*:

assumes *vwb-lens* x $\$x'$ $\#$ P $\$x$ $\#$ Q

shows $((\$x' =_u \$x \wedge P) ;; (\$x' =_u \$x \wedge Q)) = (\$x' =_u \$x \wedge (P ;; Q))$

using *assms* **by** (rel-auto', metis (no-types, lifting) vwb-lens-wb wb-lens-weak weak-lens.put-get)

lemma *seqr-bool-split*:

assumes *vwb-lens* x

shows $P ;; Q = (P \llbracket \text{true}/\$x' \rrbracket ;; Q \llbracket \text{true}/\$x \rrbracket \vee P \llbracket \text{false}/\$x' \rrbracket ;; Q \llbracket \text{false}/\$x \rrbracket)$

using *assms*

by (subst seqr-middle[of x], simp-all)

lemma *cond-inter-var-split*:

assumes *vwb-lens* x

shows $(P \triangleleft \$x' \triangleright Q) ;; R = (P \llbracket \text{true}/\$x' \rrbracket ;; R \llbracket \text{true}/\$x \rrbracket \vee Q \llbracket \text{false}/\$x' \rrbracket ;; R \llbracket \text{false}/\$x \rrbracket)$

proof –

have $(P \triangleleft \$x' \triangleright Q) ;; R = ((\$x' \wedge P) ;; R \vee (\neg \$x' \wedge Q) ;; R)$

by (simp add: cond-def seqr-or-distl)

also have $\dots = ((P \wedge \$x') ;; R \vee (Q \wedge \neg \$x') ;; R)$

by (rel-auto)

also have $\dots = (P \llbracket \text{true}/\$x' \rrbracket ;; R \llbracket \text{true}/\$x \rrbracket \vee Q \llbracket \text{false}/\$x' \rrbracket ;; R \llbracket \text{false}/\$x \rrbracket)$

by (simp add: seqr-left-one-point-true seqr-left-one-point-false *assms*)

finally show ?thesis .

qed

theorem *seqr-pre-transfer*: $\text{in}\alpha \# q \implies ((P \wedge q) ;; R) = (P ;; (q^- \wedge R))$

by (rel-auto)

theorem *seqr-pre-transfer'*:

$((P \wedge \lceil q \rceil_{>}) ;; R) = (P ;; (\lceil q \rceil_{<} \wedge R))$

by (rel-auto)

theorem *seqr-post-out*: $\text{in}\alpha \# r \implies (P ;; (Q \wedge r)) = ((P ;; Q) \wedge r)$

by (rel-blast)

lemma *seqr-post-var-out*:

fixes $x :: (\text{bool} \implies 'a)$

shows $(P ;; (Q \wedge \$x')) = ((P ;; Q) \wedge \$x')$

by (rel-auto)

theorem *seqr-post-transfer*: $\text{out}\alpha \# q \implies (P ;; (q \wedge R)) = ((P \wedge q^-) ;; R)$

by (rel-auto)

lemma *seqr-pre-out*: $\text{out}\alpha \# p \implies ((p \wedge Q) ;; R) = (p \wedge (Q ;; R))$

by (rel-blast)

lemma *seqr-pre-var-out*:

fixes $x :: (\text{bool} \implies 'a)$

shows $((\$x \wedge P) ;; Q) = (\$x \wedge (P ;; Q))$
by *(rel-auto)*

lemma *segr-true-lemma*:
 $(P = (\neg ((\neg P) ;; \text{true}))) = (P = (P ;; \text{true}))$
by *(rel-auto)*

lemma *segr-to-conj*: $\llbracket \text{out}\alpha \# P; \text{in}\alpha \# Q \rrbracket \implies (P ;; Q) = (P \wedge Q)$
by *(metis postcond-left-unit segr-pre-out utp-pred-laws.inf-top.right-neutral)*

lemma *shEx-lift-seq-1* [*uquant-lift*]:
 $((\exists x \cdot P x) ;; Q) = (\exists x \cdot (P x ;; Q))$
by *rel-auto*

lemma *shEx-mem-lift-seq-1* [*uquant-lift*]:
assumes $\text{out}\alpha \# A$
shows $((\exists x \in A \cdot P x) ;; Q) = (\exists x \in A \cdot (P x ;; Q))$
using *assms* **by** *rel-blast*

lemma *shEx-lift-seq-2* [*uquant-lift*]:
 $(P ;; (\exists x \cdot Q x)) = (\exists x \cdot (P ;; Q x))$
by *rel-auto*

lemma *shEx-mem-lift-seq-2* [*uquant-lift*]:
assumes $\text{in}\alpha \# A$
shows $(P ;; (\exists x \in A \cdot Q x)) = (\exists x \in A \cdot (P ;; Q x))$
using *assms* **by** *rel-blast*

18.4 Iterated Sequential Composition Laws

lemma *iter-segr-nil* [*simp*]: $(;; i : [] \cdot P(i)) = II$
by *(simp add: segr-iter-def)*

lemma *iter-segr-cons* [*simp*]: $(;; i : (x \# xs) \cdot P(i)) = P(x) ;; (;; i : xs \cdot P(i))$
by *(simp add: segr-iter-def)*

18.5 Quantale Laws

lemma *seq-Sup-distl*: $P ;; (\bigcap A) = (\bigcap_{Q \in A} P ;; Q)$
by *(transfer, auto)*

lemma *seq-Sup-distr*: $(\bigcap A) ;; Q = (\bigcap_{P \in A} P ;; Q)$
by *(transfer, auto)*

lemma *seq-UNIF-distl*: $P ;; (\bigcap_{Q \in A} F(Q)) = (\bigcap_{Q \in A} P \cdot F(Q))$
by *(simp add: UNIF-as-Sup-collect seq-Sup-distl)*

lemma *seq-UNIF-distl'*: $P ;; (\bigcap Q \cdot F(Q)) = (\bigcap Q \cdot P ;; F(Q))$
by *(metis UNIF-mem-UNIV seq-UNIF-distl)*

lemma *seq-UNIF-distr*: $(\bigcap_{P \in A} F(P)) ;; Q = (\bigcap_{P \in A} P \cdot F(P) ;; Q)$
by *(simp add: UNIF-as-Sup-collect seq-Sup-distr)*

lemma *seq-UNIF-distr'*: $(\bigcap P \cdot F(P)) ;; Q = (\bigcap P \cdot F(P) ;; Q)$
by *(metis UNIF-mem-UNIV seq-UNIF-distr)*

lemma *seq-SUP-distl*: $P ;; (\prod_{i \in A}. Q(i)) = (\prod_{i \in A}. P ;; Q(i))$
by (*metis image-image seq-Sup-distl*)

lemma *seq-SUP-distr*: $(\prod_{i \in A}. P(i)) ;; Q = (\prod_{i \in A}. P(i) ;; Q)$
by (*simp add: seq-Sup-distr*)

18.6 Skip Laws

lemma *cond-skip*: $\text{out}\alpha \# b \implies (b \wedge II) = (II \wedge b^-)$
by (*rel-auto*)

lemma *pre-skip-post*: $([b]_< \wedge II) = (II \wedge [b]_>)$
by (*rel-auto*)

lemma *skip-var*:
fixes $x :: (\text{bool} \implies 'a)$
shows $(\$x \wedge II) = (II \wedge \$x')$
by (*rel-auto*)

lemma *skip-r-unfold*:
 $\text{vwb-lens } x \implies II = (\$x' =_u \$x \wedge II \upharpoonright_\alpha x)$
by (*rel-simp, metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens.get-put*)

lemma *skip-r-alpha-eq*:
 $II = (\$v' =_u \$v)$
by (*rel-auto*)

lemma *skip-ra-unfold*:
 $II_{x;y} = (\$x' =_u \$x \wedge II_y)$
by (*rel-auto*)

lemma *skip-res-as-ra*:
 $\llbracket \text{vwb-lens } y; x +_L y \approx_L 1_L; x \bowtie y \rrbracket \implies II \upharpoonright_\alpha x = II_y$
apply (*rel-auto*)
apply (*metis (no-types, lifting) lens-indep-def*)
apply (*metis vwb-lens.put-eq*)
done

18.7 Assignment Laws

lemma *assigns-subst* [*usubst*]:
 $[\sigma]_s \upharpoonright \langle \varrho \rangle_a = \langle \varrho \circ \sigma \rangle_a$
by (*rel-auto*)

lemma *assigns-r-comp*: $(\langle \sigma \rangle_a ;; P) = ([\sigma]_s \upharpoonright P)$
by (*rel-auto*)

lemma *assigns-r-feasible*:
 $(\langle \sigma \rangle_a ;; \text{true}) = \text{true}$
by (*rel-auto*)

lemma *assign-subst* [*usubst*]:
 $\llbracket \text{mwb-lens } x; \text{mwb-lens } y \rrbracket \implies [\$x \mapsto_s [u]_<] \upharpoonright (y := v) = (x, y) := (u, [x \mapsto_s u] \upharpoonright v)$
by (*rel-auto*)

lemma *assign-vacuous-skip*:

assumes *vwb-lens* x
shows $(x := \&x) = II$
using *assms* **by** *rel-auto*

The following law shows the case for the above law when x is only mainly-well behaved. We require that the state is one of those in which x is well defined using and assumption.

lemma *assign-vacuous-assume*:

assumes *mwb-lens* x
shows $[(\&\mathbf{v} \in_u \ll \mathcal{S}_x \gg)]^\top \;; (x := \&x) = [(\&\mathbf{v} \in_u \ll \mathcal{S}_x \gg)]^\top$
using *assms* **by** *rel-auto*

lemma *assign-simultaneous*:

assumes *vwb-lens* y $x \bowtie y$
shows $(x, y) := (e, \&y) = (x := e)$
by (*simp* *add*: *assms* *usubst-upd-comm* *usubst-upd-var-id*)

lemma *assigns-idem*: *mwb-lens* $x \implies (x, x) := (u, v) = (x := v)$
by (*simp* *add*: *usubst*)

lemma *assigns-comp*: $(\langle f \rangle_a \;; \langle g \rangle_a) = \langle g \circ f \rangle_a$
by (*simp* *add*: *assigns-r-comp* *usubst*)

lemma *assigns-cond*: $(\langle f \rangle_a \triangleleft b \triangleright_r \langle g \rangle_a) = \langle f \triangleleft b \triangleright_s g \rangle_a$
by (*rel-auto*)

lemma *assigns-r-conv*:

bij $f \implies \langle f \rangle_a^- = \langle \text{inv } f \rangle_a$
by (*rel-auto*, *simp-all* *add*: *bij-is-inj* *bij-is-surj* *surj-f-inv-f*)

lemma *assign-pred-transfer*:

fixes $x :: ('a \implies 'a)$
assumes $\$x \# b \text{ out } \alpha \# b$
shows $(b \wedge x := v) = (x := v \wedge b^-)$
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*: *mwb-lens* $x \implies (x := \ll u \gg \;; x := \ll v \gg) = (x := \ll v \gg)$
by (*simp* *add*: *assigns-comp* *usubst*)

lemma *assign-twice*: $\ll \text{mwb-lens } x; x \# f \gg \implies (x := e \;; x := f) = (x := f)$
by (*simp* *add*: *assigns-comp* *usubst* *unrest*)

lemma *assign-commute*:

assumes $x \bowtie y$ $x \# f$ $y \# e$
shows $(x := e \;; y := f) = (y := f \;; x := e)$
using *assms*
by (*rel-simp*, *simp-all* *add*: *lens-indep-comm*)

lemma *assign-cond*:

fixes $x :: ('a \implies 'a)$
assumes *out* $\alpha \# b$
shows $(x := e \;; (P \triangleleft b \triangleright Q)) = ((x := e \;; P) \triangleleft (b[[e]_{<}/\$x]) \triangleright (x := e \;; Q))$
by (*rel-auto*)

lemma *assign-rcond*:
fixes $x :: ('a \Rightarrow 'α)$
shows $(x := e ;; (P \triangleleft b \triangleright_r Q)) = ((x := e ;; P) \triangleleft (b[e/x])) \triangleright_r (x := e ;; Q)$
by (*rel-auto*)

lemma *assign-r-alt-def*:
fixes $x :: ('a \Rightarrow 'α)$
shows $x := v = II[[v]_</\$x]$
by (*rel-auto*)

lemma *assigns-r-ufunc*: *ufunctional* $\langle f \rangle_a$
by (*rel-auto*)

lemma *assigns-r-uinj*: *inj* $f \Rightarrow \text{uinj } \langle f \rangle_a$
by (*rel-simp*, *simp add: inj-eq*)

lemma *assigns-r-swap-uinj*:
 $\llbracket \text{vwb-lens } x; \text{vwb-lens } y; x \bowtie y \rrbracket \Rightarrow \text{uinj } ((x,y) := (\&y, \&x))$
by (*metis assigns-r-uinj pr-var-def swap-usubst-inj*)

lemma *assign-unfold*:
 $\text{vwb-lens } x \Rightarrow (x := v) = (\$x' =_u [v]_< \wedge II \downarrow_\alpha x)$
apply (*rel-auto*, *auto simp add: comp-def*)
using *vwb-lens.put-eq* **by** *fastforce*

18.8 Non-deterministic Assignment Laws

lemma *nd-assign-comp*:
 $x \bowtie y \Rightarrow x := * ;; y := * = x, y := *$
apply (*rel-auto*) **using** *lens-indep-comm* **by** *fastforce+*

lemma *nd-assign-assign*:
 $\llbracket \text{vwb-lens } x; x \# e \rrbracket \Rightarrow x := * ;; x := e = x := e$
by (*rel-auto*)

18.9 Converse Laws

lemma *convr-invol* [*simp*]: $p^{--} = p$
by *pred-auto*

lemma *lit-convr* [*simp*]: $\llbracket v \rrbracket^- = \llbracket v \rrbracket$
by *pred-auto*

lemma *uivar-convr* [*simp*]:
fixes $x :: ('a \Rightarrow 'α)$
shows $(\$x)^- = \x'
by *pred-auto*

lemma *uovar-convr* [*simp*]:
fixes $x :: ('a \Rightarrow 'α)$
shows $(\$x')^- = \x
by *pred-auto*

lemma *uop-convr* [*simp*]: $(\text{uop } f \text{ } u)^- = \text{uop } f \text{ } (u^-)$
by (*pred-auto*)

lemma *bop-convr* [simp]: $(bop\ f\ u\ v)^- = bop\ f\ (u^-)\ (v^-)$
 by (pred-auto)

lemma *eq-convr* [simp]: $(p =_u q)^- = (p^- =_u q^-)$
 by (pred-auto)

lemma *not-convr* [simp]: $(\neg p)^- = (\neg p^-)$
 by (pred-auto)

lemma *disj-convr* [simp]: $(p \vee q)^- = (q^- \vee p^-)$
 by (pred-auto)

lemma *conj-convr* [simp]: $(p \wedge q)^- = (q^- \wedge p^-)$
 by (pred-auto)

lemma *seqr-convr* [simp]: $(p ;; q)^- = (q^- ;; p^-)$
 by (rel-auto)

lemma *pre-convr* [simp]: $[p]_{<}^- = [p]_{>}$
 by (rel-auto)

lemma *post-convr* [simp]: $[p]_{>}^- = [p]_{<}$
 by (rel-auto)

18.10 Assertion and Assumption Laws

declare *sublens-def* [lens-defs del]

lemma *assume-false*: $[false]^\top = false$
 by (rel-auto)

lemma *assume-true*: $[true]^\top = II$
 by (rel-auto)

lemma *assume-seq*: $[b]^\top ;; [c]^\top = [(b \wedge c)]^\top$
 by (rel-auto)

lemma *assert-false*: $\{false\}_\perp = true$
 by (rel-auto)

lemma *assert-true*: $\{true\}_\perp = II$
 by (rel-auto)

lemma *assert-seq*: $\{b\}_\perp ;; \{c\}_\perp = \{(b \wedge c)\}_\perp$
 by (rel-auto)

18.11 Frame and Antiframe Laws

named-theorems *frame*

lemma *frame-all* [frame]: $\Sigma:[P] = P$
 by (rel-auto)

lemma *frame-none* [frame]:
 $\emptyset:[P] = (P \wedge II)$

by (rel-auto)

lemma *frame-commute*:

assumes $\$y \# P \ \$y' \# P \ \$x \# Q \ \$x' \# Q \ x \bowtie y$
 shows $x:[P] \;; \; y:[Q] = y:[Q] \;; \; x:[P]$
 apply (insert assms)
 apply (rel-auto)
 apply (rename-tac $s \ s' \ s_0$)
 apply (subgoal-tac ($s \oplus_L s' \text{ on } y \oplus_L s_0 \text{ on } x = s_0 \oplus_L s' \text{ on } y$)
 apply (metis lens-indep-get lens-indep-sym lens-override-def)
 apply (simp add: lens-indep.lens-put-comm lens-override-def)
 apply (rename-tac $s \ s' \ s_0$)
 apply (subgoal-tac $\text{put}_y (\text{put}_x s (\text{get}_x (\text{put}_x s_0 (\text{get}_x s')))) (\text{get}_y (\text{put}_y s (\text{get}_y s_0)))$
 $= \text{put}_x s_0 (\text{get}_x s')$)
 apply (metis lens-indep-get lens-indep-sym)
 apply (metis lens-indep.lens-put-comm)
 done

lemma *frame-contract-RID*:

assumes *vwb-lens* $x \ P$ is $RID(x) \ x \bowtie y$
 shows $(x;y):[P] = y:[P]$
 proof –
 from assms(1,3) have $(x;y):[RID(x)(P)] = y:[RID(x)(P)]$
 apply (rel-auto)
 apply (simp add: lens-indep.lens-put-comm)
 apply (metis (no-types) vwb-lens-wb wb-lens.get-put)
 done
 thus ?thesis
 by (simp add: Healthy-if assms)
 qed

lemma *frame-miracle* [simp]:

$x:[\text{false}] = \text{false}$
 by (rel-auto)

lemma *frame-skip* [simp]:

vwb-lens $x \implies x:[II] = II$
 by (rel-auto)

lemma *frame-assign-in* [frame]:

$\llbracket \text{vwb-lens } a; x \subseteq_L a \rrbracket \implies a:[x := v] = x := v$
 by (rel-auto, simp-all add: lens-get-put-quasi-commute lens-put-of-quotient)

lemma *frame-conj-true* [frame]:

$\llbracket \{ \$x, \$x' \} \vdash P; \text{vwb-lens } x \rrbracket \implies (P \wedge x:[\text{true}]) = x:[P]$
 by (rel-auto)

lemma *frame-is-assign* [frame]:

vwb-lens $x \implies x:[\$x' =_u \lceil v \rceil_<] = x := v$
 by (rel-auto)

lemma *frame-seq* [frame]:

$\llbracket \text{vwb-lens } x; \{ \$x, \$x' \} \vdash P; \{ \$x, \$x' \} \vdash Q \rrbracket \implies x:[P \;; \; Q] = x:[P] \;; \; x:[Q]$
 apply (rel-auto)
 apply (metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens-def weak-lens.put-get)

```

apply (metis mwb-lens.put-put vwb-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 lens-indep-def, metis lens-indep-def surj-pair)

lemma rel-frex-miracle [frame]:
   $a:[false]^+ = false$ 
  by (rel-auto)

lemma rel-frex-skip [frame]:
   $vwb-lens\ a \implies a:[II]^+ = II$ 
  by (rel-auto)

lemma rel-frex-seq [frame]:
   $vwb-lens\ a \implies a:[P ;; Q]^+ = (a:[P]^+ ;; a:[Q]^+)$ 
  apply (rel-auto)
  apply (rename-tac s s' s0)
  apply (rule-tac x=puta s s0 in exI)
  apply (auto)
  apply (metis mwb-lens.put-put vwb-lens-mwb)
done

lemma rel-frex-assigns [frame]:
   $vwb-lens\ a \implies a:[\langle \sigma \rangle_a]^+ = \langle \sigma \oplus_s a \rangle_a$ 
  by (rel-auto)

lemma rel-frex-rcond [frame]:
   $a:[P \triangleleft b \triangleright_r Q]^+ = (a:[P]^+ \triangleleft b \oplus_p a \triangleright_r a:[Q]^+)$ 
  by (rel-auto)

lemma rel-frex-commute:
   $x \bowtie y \implies x:[P]^+ ;; y:[Q]^+ = y:[Q]^+ ;; x:[P]^+$ 
  apply (rel-auto)
  apply (rename-tac a c b)
  apply (subgoal-tac  $\bigwedge b\ a.\ get_y\ (put_x\ b\ a) = get_y\ b$ )
  apply (metis (no-types, hide-lams) lens-indep-comm lens-indep-get)
  apply (simp add: lens-indep.lens-put-irr2)
  apply (subgoal-tac  $\bigwedge b\ c.\ get_x\ (put_y\ b\ c) = get_x\ b$ )
  apply (subgoal-tac  $\bigwedge b\ a.\ get_y\ (put_x\ b\ a) = get_y\ b$ )
  apply (metis (mono-tags, lifting) lens-indep-comm)
  apply (simp-all add: lens-indep.lens-put-irr2)
done

lemma antiframe-disj [frame]:  $(x:[P] \vee x:[Q]) = x:[P \vee Q]$ 
  by (rel-auto)

lemma antiframe-seq [frame]:
   $\llbracket vwb-lens\ x; \$x' \# P; \$x \# Q \rrbracket \implies (x:[P] ;; x:[Q]) = x:[P ;; Q]$ 
  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 *nameset-skip*: $vwb\text{-}lens\ x \implies (ns\ x \cdot II) = II_x$
 by (*rel-auto*, *meson vwb-lens-wb wb-lens.get-put*)

lemma *nameset-skip-ra*: $vwb\text{-}lens\ x \implies (ns\ x \cdot II_x) = II_x$
 by (*rel-auto*)

declare *sublens-def* [*lens-defs*]

18.12 While Loop Laws

theorem *while-unfold*:

$while\ b\ do\ P\ od = ((P\ ;;\ while\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$

proof –

have $m\text{:}mono\ (\lambda X. (P\ ;;\ X) \triangleleft b \triangleright_r II)$
 by (*auto intro: monoI segr-mono cond-mono*)
have $(while\ b\ do\ P\ od) = (\nu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)$
 by (*simp add: while-top-def*)
also have $\dots = ((P\ ;;\ (\nu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
 by (*subst lfp-unfold, simp-all add: m*)
also have $\dots = ((P\ ;;\ while\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$
 by (*simp add: while-top-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-top-def alpha*)
apply (*rule antisym*)
apply (*simp-all*)
apply (*rule lfp-lowerbound*)
apply (*rel-auto*)
done

theorem *while-bot-unfold*:

$while_{\perp}\ b\ do\ P\ od = ((P\ ;;\ while_{\perp}\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$

proof –

have $m\text{:}mono\ (\lambda X. (P\ ;;\ X) \triangleleft b \triangleright_r II)$
 by (*auto intro: monoI segr-mono cond-mono*)
have $(while_{\perp}\ b\ do\ P\ od) = (\mu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)$
 by (*simp add: while-bot-def*)
also have $\dots = ((P\ ;;\ (\mu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
 by (*subst gfp-unfold, simp-all add: m*)
also have $\dots = ((P\ ;;\ while_{\perp}\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$
 by (*simp add: while-bot-def*)
finally show *?thesis* .

qed

theorem *while-bot-false*: $while_{\perp}\ false\ do\ P\ od = II$

by (*simp add: while-bot-def mu-const alpha*)

theorem *while-bot-true*: $while_{\perp}\ true\ do\ P\ od = (\mu\ X \cdot P\ ;;\ X)$

by (*simp add: while-bot-def alpha*)

An infinite loop with a feasible body corresponds to a program error (non-termination).

```

theorem while-infinite:  $P \;; \text{true}_h = \text{true} \implies \text{while}_\perp \text{true} \text{ do } P \text{ od} = \text{true}$ 
apply (simp add: while-bot-true)
apply (rule antisym)
apply (simp)
apply (rule gfp-upperbound)
apply (simp)
done

```

18.13 Algebraic Properties

interpretation *upred-semiring*: *semiring-1*

where *times* = *seqr* **and** *one* = *skip-r* **and** *zero* = *false_h* **and** *plus* = *Lattices.sup*
by (*unfold-locales, (rel-auto)+*)

declare *upred-semiring.power-Suc* [*simp del*]

We introduce the power syntax derived from semirings

abbreviation *upower* :: $'\alpha \text{ hrel} \Rightarrow \text{nat} \Rightarrow '\alpha \text{ hrel}$ (**infixr** $\wedge 80$) **where**
upower *P* *n* \equiv *upred-semiring.power* *P* *n*

translations

$P \wedge i \leq \text{CONST } \text{power.power } II \text{ op } ;; P \ i$
 $P \wedge i \leq (\text{CONST } \text{power.power } II \text{ op } ;; P) \ i$

Set up transfer tactic for powers

lemma *upower-rep-eq*:

$\llbracket P \wedge i \rrbracket_e = (\lambda b. b \in (\{p. \llbracket P \rrbracket_e p\} \wedge i))$

proof (*induct i arbitrary: P*)

case 0

then show ?*case*

by (*auto, rel-auto*)

next

case (*Suc i*)

show ?*case*

by (*simp add: Suc seqr.rep-eq relpow-commute upred-semiring.power-Suc*)

qed

lemma *upower-rep-eq-alt*:

$\llbracket \text{power.power } \langle \text{id} \rangle_a \ ;; P \ i \rrbracket_e = (\lambda b. b \in (\{p. \llbracket P \rrbracket_e p\} \wedge i))$

by (*metis skip-r-def upower-rep-eq*)

update-uexpr-rep-eq-thms

lemma *Sup-power-expand*:

fixes *P* :: $\text{nat} \Rightarrow 'a::\text{complete-lattice}$

shows $P(0) \sqcap (\bigsqcap i. P(i+1)) = (\bigsqcap i. P(i))$

proof –

have *UNIV* = *insert* (*0::nat*) {*1..*}

by *auto*

moreover have $(\bigsqcap i. P(i)) = \bigsqcap (P \restriction \text{UNIV})$

by (*blast*)

moreover have $\bigsqcap (P \restriction \text{insert } 0 \ \{1..\}) = P(0) \sqcap \text{SUPREMUM } \{1..\} P$

by (*simp*)

moreover have $\text{SUPREMUM } \{1..\} P = (\bigsqcap i. P(i+1))$

by (*simp add: atLeast-Suc-greaterThan greaterThan-0*)


```

ultimately show ?thesis
  by (simp only:)
qed

```

```

lemma Sup-upto-Suc:  $(\bigwedge i \in \{0..Suc\ n\}. P \wedge i) = (\bigwedge i \in \{0..n\}. P \wedge i) \sqcap P \wedge Suc\ n$ 
proof -
  have  $(\bigwedge i \in \{0..Suc\ n\}. P \wedge i) = (\bigwedge i \in insert\ (Suc\ n)\ \{0..n\}. P \wedge i)$ 
    by (simp add: atLeast0-atMost-Suc)
  also have  $\dots = P \wedge Suc\ n \sqcap (\bigwedge i \in \{0..n\}. P \wedge i)$ 
    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:  $Q \sqsubseteq ((P ;; Q) \sqcap R) \implies Q \sqsubseteq P \wedge n ;; R$ 
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  $Q \sqsubseteq R \sqcap (Q ;; P)$ 
  shows  $Q \sqsubseteq R ;; (P \wedge n)$ 
using assms proof (induct n)
  case 0
  then show ?case by auto
next
  case (Suc n)
  have  $R ;; P \wedge Suc\ n = (R ;; P \wedge n) ;; P$ 
    by (metis seqr-assoc upred-semiring.power-Suc2)
  also have  $Q ;; P \sqsubseteq \dots$ 
    by (meson Suc.hyps assms eq-iff seqr-mono)
  also have  $Q \sqsubseteq \dots$ 
    using assms by auto
  finally show ?case .
qed

```

```

lemma SUP-atLeastAtMost-first:
  fixes  $P :: nat \Rightarrow 'a::complete-lattice$ 
  assumes  $m \leq n$ 
  shows  $(\bigwedge i \in \{m..n\}. P(i)) = P(m) \sqcap (\bigwedge i \in \{Suc\ m..n\}. P(i))$ 
  by (metis SUP-insert assms atLeastAtMost-insertL)

```

```

lemma upower-seqr-iter:  $P \wedge n = (;; Q : replicate\ n\ P \cdot Q)$ 
  by (induct n, simp-all add: upred-semiring.power-Suc)

```

```

lemma assigns-power:  $\langle f \rangle_a \wedge n = \langle f \wedge n \rangle_a$ 
  by (induct n, rel-auto+)

```

18.14 Kleene Star

definition $ustar :: 'a \text{ hrel} \Rightarrow 'a \text{ hrel} (-^* [999] 999)$ **where**
 $P^* = (\bigsqcap_{i \in \{0..\}} \cdot P^i)$

lemma $ustar\text{-}rep\text{-}eq$:

$\llbracket P^* \rrbracket_e = (\lambda b. b \in (\{p. \llbracket P \rrbracket_e p\}^*))$

by ($simp$ add : $ustar\text{-}def$, $rel\text{-}auto$, $simp\text{-}all$ add : $relpow\text{-}imp\text{-}rtrancl$ $rtrancl\text{-}imp\text{-}relpow$)

update-uexpr-rep-eq-thms

18.15 Kleene Plus

purge-notation $trancl ((-^+) [1000] 999)$

definition $uplus :: 'a \text{ hrel} \Rightarrow 'a \text{ hrel} (-^+ [999] 999)$ **where**
 $[upred\text{-}defs]: P^+ = P ;; P^*$

lemma $uplus\text{-}power\text{-}def$: $P^+ = (\bigsqcap i \cdot P \hat{\ } (Suc\ i))$

by ($simp$ add : $uplus\text{-}def$ $ustar\text{-}def$ $seq\text{-}UINF\text{-}distl'$ $UINF\text{-}atLeast\text{-}Suc$ $upred\text{-}semiring.power\text{-}Suc$)

18.16 Omega

definition $uomega :: 'a \text{ hrel} \Rightarrow 'a \text{ hrel} (-^\omega [999] 999)$ **where**
 $P^\omega = (\mu X \cdot P ;; X)$

18.17 Relation Algebra Laws

theorem $RA1$: $(P ;; (Q ;; R)) = ((P ;; Q) ;; R)$
by ($simp$ add : $seqr\text{-}assoc$)

theorem $RA2$: $(P ;; II) = P$ $(II ;; P) = P$
by $simp\text{-}all$

theorem $RA3$: $P^{--} = P$
by $simp$

theorem $RA4$: $(P ;; Q)^- = (Q^- ;; P^-)$
by $simp$

theorem $RA5$: $(P \vee Q)^- = (P^- \vee Q^-)$
by ($rel\text{-}auto$)

theorem $RA6$: $((P \vee Q) ;; R) = (P ;; R \vee Q ;; R)$
using $seqr\text{-}or\text{-}distl$ **by** $blast$

theorem $RA7$: $((P^- ;; (\neg(P ;; Q))) \vee (\neg Q)) = (\neg Q)$
by ($rel\text{-}auto$)

18.18 Kleene Algebra Laws

lemma $ustar\text{-}alt\text{-}def$: $P^* = (\bigsqcap i \cdot P \hat{\ } i)$
by ($simp$ add : $ustar\text{-}def$)

theorem $ustar\text{-}sub\text{-}unfoldl$: $P^* \sqsubseteq II \sqcap (P ;; P^*)$
by ($rel\text{-}simp$, $simp$ add : $rtrancl\text{-}into\text{-}trancl2$ $trancl\text{-}into\text{-}rtrancl$)

theorem *ustar-inductl*:
 assumes $Q \sqsubseteq R \quad Q \sqsubseteq P \;; Q$
 shows $Q \sqsubseteq P^* \;; R$
proof –
 have $P^* \;; R = (\bigsqcap i. P \wedge i \;; R)$
 by (*simp add: ustar-def UINF-as-Sup-collect' seq-SUP-distr*)
 also have $Q \sqsubseteq \dots$
 by (*simp add: SUP-least assms upower-inductl*)
 finally show *?thesis* .
qed

theorem *ustar-inductr*:
 assumes $Q \sqsubseteq R \quad Q \sqsubseteq Q \;; P$
 shows $Q \sqsubseteq R \;; P^*$
proof –
 have $R \;; P^* = (\bigsqcap i. R \;; P \wedge i)$
 by (*simp add: ustar-def UINF-as-Sup-collect' seq-SUP-distl*)
 also have $Q \sqsubseteq \dots$
 by (*simp add: SUP-least assms upower-inductr*)
 finally show *?thesis* .
qed

lemma *ustar-refines-nu*: $(\nu X \cdot (P \;; X) \sqcap II) \sqsubseteq P^*$
 by (*metis (no-types, lifting) lfp-greatest semilattice-sup-class.le-sup-iff
 semilattice-sup-class.sup-idem upred-semiring.mult-2-right
 upred-semiring.one-add-one ustar-inductl*)

lemma *ustar-as-nu*: $P^* = (\nu X \cdot (P \;; X) \sqcap II)$
proof (*rule antisym*)
 show $(\nu X \cdot (P \;; X) \sqcap II) \sqsubseteq P^*$
 by (*simp add: ustar-refines-nu*)
 show $P^* \sqsubseteq (\nu X \cdot (P \;; X) \sqcap II)$
 by (*metis lfp-lowerbound upred-semiring.add-commute ustar-sub-unfoldl*)
qed

lemma *ustar-unfoldl*: $P^* = II \sqcap (P \;; P^*)$
 apply (*simp add: ustar-as-nu*)
 apply (*subst lfp-unfold*)
 apply (*rule monoI*)
 apply (*rel-auto*)
 done

While loop can be expressed using Kleene star

lemma *while-star-form*:
 $\text{while } b \text{ do } P \text{ od} = (P \triangleleft b \triangleright_r II)^* \;; [(\neg b)]^\top$
proof –
 have *1: Continuous* $(\lambda X. P \;; X \triangleleft b \triangleright_r II)$
 by (*rel-auto*)
 have $\text{while } b \text{ do } P \text{ od} = (\bigsqcap i. ((\lambda X. P \;; X \triangleleft b \triangleright_r II) \wedge i) \text{ false})$
 by (*simp add: 1 false-upred-def sup-continuous-Continuous sup-continuous-lfp while-top-def*)
 also have $\dots = ((\lambda X. P \;; X \triangleleft b \triangleright_r II) \wedge 0) \text{ false} \sqcap (\bigsqcap i. ((\lambda X. P \;; X \triangleleft b \triangleright_r II) \wedge (i+1)) \text{ false})$
 by (*subst Sup-power-expand, simp*)
 also have $\dots = (\bigsqcap i. ((\lambda X. P \;; X \triangleleft b \triangleright_r II) \wedge (i+1)) \text{ false})$
 by (*simp*)
 also have $\dots = (\bigsqcap i. (P \triangleleft b \triangleright_r II) \wedge i \;; (\text{false} \triangleleft b \triangleright_r II))$

```

proof (rule SUP-cong, simp-all)
  fix i
  show  $P \;; (\lambda X. P \;; X \triangleleft b \triangleright_r II) \wedge i \text{ false} \triangleleft b \triangleright_r II = (P \triangleleft b \triangleright_r II) \wedge i \;; (\text{false} \triangleleft b \triangleright_r II)$ 
  proof (induct i)
    case 0
    then show ?case by simp
  next
    case (Suc i)
    then show ?case
      by (simp add: upred-semiring.power-Suc)
        (metis (no-types, lifting) RA1 comp-cond-left-distr cond-L6 upred-semiring.mult.left-neutral)
  qed
qed
also have  $\dots = (\bigcap_{i \in \{0..\}} \cdot (P \triangleleft b \triangleright_r II) \wedge i \;; [(\neg b)]^\top)$ 
  by (rel-auto)
also have  $\dots = (P \triangleleft b \triangleright_r II)^* \;; [(\neg b)]^\top$ 
  by (metis seq-UINF-distr ustar-def)
finally show ?thesis .
qed

```

18.19 Omega Algebra Laws

```

lemma uomega-induct:
   $P \;; P^\omega \sqsubseteq P^\omega$ 
  by (simp add: uomega-def, metis eq-refl gfp-unfold monoI seqr-mono)

```

18.20 Refinement Laws

```

lemma skip-r-refine:
   $(p \Rightarrow p) \sqsubseteq II$ 
  by pred-blast

```

```

lemma conj-refine-left:
   $(Q \Rightarrow P) \sqsubseteq R \Longrightarrow P \sqsubseteq (Q \wedge R)$ 
  by (rel-auto)

```

```

lemma pre-weak-rel:
  assumes 'Pre  $\Rightarrow$  I'
  and  $(I \Rightarrow \text{Post}) \sqsubseteq P$ 
  shows  $(\text{Pre} \Rightarrow \text{Post}) \sqsubseteq P$ 
  using assms by (rel-auto)

```

```

lemma cond-refine-rel:
  assumes  $S \sqsubseteq ([b]_< \wedge P)$   $S \sqsubseteq ([\neg b]_< \wedge Q)$ 
  shows  $S \sqsubseteq P \triangleleft b \triangleright_r Q$ 
  by (metis aext-not assms(1) assms(2) cond-def lift-rcond-def utp-pred-laws.le-sup-iff)

```

```

lemma seq-refine-pred:
  assumes  $([b]_< \Rightarrow [s]_>) \sqsubseteq P$  and  $([s]_< \Rightarrow [c]_>) \sqsubseteq Q$ 
  shows  $([b]_< \Rightarrow [c]_>) \sqsubseteq (P \;; Q)$ 
  using assms by rel-auto

```

```

lemma seq-refine-unrest:
  assumes  $\text{out}\alpha \nmid b \text{ in}\alpha \nmid c$ 
  assumes  $(b \Rightarrow [s]_>) \sqsubseteq P$  and  $([s]_< \Rightarrow c) \sqsubseteq Q$ 
  shows  $(b \Rightarrow c) \sqsubseteq (P \;; Q)$ 

```

using *assms* by *rel-blast*

18.21 Domain and Range Laws

named-theorems *domran*

lemma *Dom-conv-Ran* [*domran*]:

$$\text{Dom}(P^-) = \text{Ran}(P)$$

by (*rel-auto*)

lemma *Ran-conv-Dom* [*domran*]:

$$\text{Ran}(P^-) = \text{Dom}(P)$$

by (*rel-auto*)

lemma *Dom-skip* [*domran*]:

$$\text{Dom}(II) = \text{true}$$

by (*rel-auto*)

lemma *Dom-assigns* [*domran*]:

$$\text{Dom}(\langle \sigma \rangle_a) = \text{true}$$

by (*rel-auto*)

lemma *Dom-miracle* [*domran*]:

$$\text{Dom}(\text{false}) = \text{false}$$

by (*rel-auto*)

lemma *Dom-assume* [*domran*]:

$$\text{Dom}([b]^\top) = b$$

by (*rel-auto*)

lemma *Dom-seq*:

$$\text{Dom}(P ;; Q) = \text{Dom}(P ;; [\text{Dom}(Q)]^\top)$$

by (*rel-auto*)

lemma *Dom-disj* [*domran*]:

$$\text{Dom}(P \vee Q) = (\text{Dom}(P) \vee \text{Dom}(Q))$$

by (*rel-auto*)

lemma *Dom-inf* [*domran*]:

$$\text{Dom}(P \sqcap Q) = (\text{Dom}(P) \vee \text{Dom}(Q))$$

by (*rel-auto*)

lemma *Dom-conj-rel-aext* [*domran*]:

$$\llbracket \text{vwb-lens } a; \text{vwb-lens } b; a \bowtie b \rrbracket \implies \text{Dom}(P \oplus_r a \wedge Q \oplus_r b) = (\text{Dom}(P \oplus_r a) \wedge \text{Dom}(Q \oplus_r b))$$

by (*rel-auto*, *metis* (*no-types*, *lifting*) *lens-indep-def* *mwb-lens-def* *vwb-lens-mwb* *weak-lens-def*)

If *P* uses on the variables in *a* and *Q* does not refer to the variables of *\$a'* then we can distribute.

lemma *Dom-conj-indep* [*domran*]: $\llbracket \{ \$a, \$a' \} \Vdash P; \$a' \nVdash Q; \text{vwb-lens } a \rrbracket \implies \text{Dom}(P \wedge Q) = (\text{Dom}(P) \wedge \text{Dom}(Q))$

by (*rel-auto*, *metis* *lens-override-def* *lens-override-idem*)

lemma *assume-Dom* [*domran*]:

$$[\text{Dom}(P)]^\top ;; P = P$$

by (*rel-auto*)

end

19 UTP Theories

```

theory utp-theory
imports utp-rel-laws
begin

```

Here, we mechanise a representation of UTP theories using locales [4]. We also link them to the HOL-Algebra library [5], which allows us to import properties from complete lattices and Galois connections.

19.1 Complete lattice of predicates

definition *upred-lattice* :: (α upred) gorder (\mathcal{P}) **where**
upred-lattice = $\langle \mid \text{carrier} = \text{UNIV}, \text{eq} = (=), \text{le} = (\sqsubseteq) \mid \rangle$

\mathcal{P} is the complete lattice of alphabetised predicates. All other theories will be defined relative to it.

```

interpretation upred-lattice: complete-lattice  $\mathcal{P}$ 
proof (unfold-locales, simp-all add: upred-lattice-def)
  fix A ::  $\alpha$  upred set
  show  $\exists s. \text{is-lub } \langle \mid \text{carrier} = \text{UNIV}, \text{eq} = (=), \text{le} = (\sqsubseteq) \mid \rangle s A$ 
    apply (rule-tac x= $\sqcup$  A in exI)
    apply (rule least-UpperI)
    apply (auto intro: Inf-greatest simp add: Inf-lower Upper-def)
  done
  show  $\exists i. \text{is-glb } \langle \mid \text{carrier} = \text{UNIV}, \text{eq} = (=), \text{le} = (\sqsubseteq) \mid \rangle i A$ 
    apply (rule-tac x= $\sqcap$  A in exI)
    apply (rule greatest-LowerI)
    apply (auto intro: Sup-least simp add: Sup-upper Lower-def)
  done
qed

```

lemma *upred-weak-complete-lattice* [simp]: *weak-complete-lattice* \mathcal{P}
by (simp add: upred-lattice.weak.weak-complete-lattice-axioms)

lemma *upred-lattice-eq* [simp]:
 $(\cdot =_{\mathcal{P}}) = (=)$
by (simp add: upred-lattice-def)

lemma *upred-lattice-le* [simp]:
 $\text{le } \mathcal{P} P Q = (P \sqsubseteq Q)$
by (simp add: upred-lattice-def)

lemma *upred-lattice-carrier* [simp]:
 $\text{carrier } \mathcal{P} = \text{UNIV}$
by (simp add: upred-lattice-def)

lemma *Healthy-fixed-points* [simp]: $\text{fps } \mathcal{P} H = \llbracket H \rrbracket_H$
by (simp add: fps-def upred-lattice-def Healthy-def)

lemma *upred-lattice-Idempotent* [simp]: $\text{Idem}_{\mathcal{P}} H = \text{Idempotent } H$
using upred-lattice.weak-partial-order-axioms **by** (auto simp add: idempotent-def Idempotent-def)

lemma *upred-lattice-Monotonic* [simp]: $\text{Mono}_{\mathcal{P}} H = \text{Monotonic } H$
using upred-lattice.weak-partial-order-axioms **by** (auto simp add: isotone-def mono-def)

19.2 UTP theories hierarchy

definition $utp\text{-}order :: ('α \times 'α) \text{ health} \Rightarrow 'α \text{ hrel } gorder$ **where**
 $utp\text{-}order\ H = (\mid carrier = \{P. P \text{ is } H\}, eq = (=), le = (\sqsubseteq) \mid)$

Constant $utp\text{-}order$ 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 $utp\text{-}order\text{-}carrier$ [simp]:
 $carrier\ (utp\text{-}order\ H) = \llbracket H \rrbracket_H$
by (simp add: utp-order-def)

lemma $utp\text{-}order\text{-}eq$ [simp]:
 $eq\ (utp\text{-}order\ T) = (=)$
by (simp add: utp-order-def)

lemma $utp\text{-}order\text{-}le$ [simp]:
 $le\ (utp\text{-}order\ T) = (\sqsubseteq)$
by (simp add: utp-order-def)

lemma $utp\text{-}partial\text{-}order$: $partial\text{-}order\ (utp\text{-}order\ T)$
by (unfold-locales, simp-all add: utp-order-def)

lemma $utp\text{-}weak\text{-}partial\text{-}order$: $weak\text{-}partial\text{-}order\ (utp\text{-}order\ T)$
by (unfold-locales, simp-all add: utp-order-def)

lemma $mono\text{-}Monotone\text{-}utp\text{-}order$:
 $mono\ f \Longrightarrow Monotone\ (utp\text{-}order\ T)\ f$
apply (auto simp add: isotone-def)
apply (metis partial-order-def utp-partial-order)
apply (metis monoD)
done

lemma $isotone\text{-}utp\text{-}orderI$: $Monotonic\ H \Longrightarrow isotone\ (utp\text{-}order\ X)\ (utp\text{-}order\ Y)\ H$
by (auto simp add: mono-def isotone-def utp-weak-partial-order)

lemma $Mono\text{-}utp\text{-}orderI$:
 $\llbracket \bigwedge P\ Q. \llbracket P \sqsubseteq Q; P \text{ is } H; Q \text{ is } H \rrbracket \Longrightarrow F(P) \sqsubseteq F(Q) \rrbracket \Longrightarrow Mono_{utp\text{-}order\ H}\ F$
by (auto simp add: isotone-def utp-weak-partial-order)

The UTP order can equivalently be characterised as the fixed point lattice, fpl .

lemma $utp\text{-}order\text{-}fpl$: $utp\text{-}order\ H = fpl\ \mathcal{P}\ H$
by (auto simp add: utp-order-def upred-lattice-def fps-def Healthy-def)

19.3 UTP theory hierarchy

We next define a hierarchy of locales that characterise different classes of UTP theory. Minimally we require that a UTP theory's healthiness condition is idempotent.

locale $utp\text{-}theory =$
fixes $hcond :: 'α \text{ hrel} \Rightarrow 'α \text{ hrel}\ (\mathcal{H})$
assumes $HCond\text{-}Idem$: $\mathcal{H}(\mathcal{H}(P)) = \mathcal{H}(P)$
begin

abbreviation $thy\text{-}order :: 'α \text{ hrel } gorder$ **where**
 $thy\text{-}order \equiv utp\text{-}order\ \mathcal{H}$

lemma *HCond-Idempotent* [*closure,intro*]: *Idempotent* \mathcal{H}
by (*simp add: Idempotent-def HCond-Idem*)

sublocale *utp-po: partial-order utp-order* \mathcal{H}
by (*unfold-locales, simp-all add: utp-order-def*)

We need to remove some transitivity rules to stop them being applied in calculations

declare *utp-po.trans* [*trans del*]

end

locale *utp-theory-lattice* = *utp-theory* +
assumes *uthy-lattice: complete-lattice* (*utp-order* \mathcal{H})
begin

sublocale *complete-lattice utp-order* \mathcal{H}
by (*simp add: uthy-lattice*)

declare *top-closed* [*simp del*]
declare *bottom-closed* [*simp del*]

The healthiness conditions of a UTP theory lattice form a complete lattice, and allows us to make use of complete lattice results from HOL-Algebra [5], such as the Knaster-Tarski theorem. We can also retrieve lattice operators as below.

abbreviation *utp-top* (\top)
where *utp-top* \equiv *top* (*utp-order* \mathcal{H})

abbreviation *utp-bottom* (\perp)
where *utp-bottom* \equiv *bottom* (*utp-order* \mathcal{H})

abbreviation *utp-join* (**infixl** \sqcup 65) **where**
utp-join \equiv *join* (*utp-order* \mathcal{H})

abbreviation *utp-meet* (**infixl** \sqcap 70) **where**
utp-meet \equiv *meet* (*utp-order* \mathcal{H})

abbreviation *utp-sup* (\bigsqcup - [90] 90) **where**
utp-sup \equiv *Lattice.sup* (*utp-order* \mathcal{H})

abbreviation *utp-inf* (\bigsqcap - [90] 90) **where**
utp-inf \equiv *Lattice.inf* (*utp-order* \mathcal{H})

abbreviation *utp-gfp* (ν) **where**
utp-gfp \equiv *GREATEST-FP* (*utp-order* \mathcal{H})

abbreviation *utp-lfp* (μ) **where**
utp-lfp \equiv *LEAST-FP* (*utp-order* \mathcal{H})

end

syntax
-tmu :: *logic* \Rightarrow *pttrn* \Rightarrow *logic* \Rightarrow *logic* (μ 1 - \cdot - [0, 10] 10)
-tnu :: *logic* \Rightarrow *pttrn* \Rightarrow *logic* \Rightarrow *logic* (ν 1 - \cdot - [0, 10] 10)

notation *gfp* (μ)

notation $\text{lf}p\ (\nu)$

translations

$\mu_H X \cdot P == \text{CONST LEAST-FP } (\text{CONST utp-order } H) (\lambda X. P)$
 $\nu_H X \cdot P == \text{CONST GREATEST-FP } (\text{CONST utp-order } H) (\lambda X. P)$

lemma *upred-lattice-inf*:

Lattice.inf $\mathcal{P} A = \bigsqcap A$

by (*metis Sup-least Sup-upper UNIV-I antisym-conv subsetI upred-lattice.weak.inf-greatest upred-lattice.weak.inf-lower upred-lattice-carrier upred-lattice-le*)

We can then derive a number of properties about these operators, as below.

context *utp-theory-lattice*

begin

lemma *LFP-healthy-comp*: $\mu F = \mu (F \circ \mathcal{H})$

proof –

have $\{P. (P \text{ is } \mathcal{H}) \wedge F P \sqsubseteq P\} = \{P. (P \text{ is } \mathcal{H}) \wedge F (\mathcal{H} P) \sqsubseteq P\}$

by (*auto simp add: Healthy-def*)

thus *?thesis*

by (*simp add: LEAST-FP-def*)

qed

lemma *GFP-healthy-comp*: $\nu F = \nu (F \circ \mathcal{H})$

proof –

have $\{P. (P \text{ is } \mathcal{H}) \wedge P \sqsubseteq F P\} = \{P. (P \text{ is } \mathcal{H}) \wedge P \sqsubseteq F (\mathcal{H} P)\}$

by (*auto simp add: Healthy-def*)

thus *?thesis*

by (*simp add: GREATEST-FP-def*)

qed

lemma *top-healthy [closure]*: $\top \text{ is } \mathcal{H}$

using *weak.top-closed* **by** *auto*

lemma *bottom-healthy [closure]*: $\perp \text{ is } \mathcal{H}$

using *weak.bottom-closed* **by** *auto*

lemma *utp-top*: $P \text{ is } \mathcal{H} \implies P \sqsubseteq \top$

using *weak.top-higher* **by** *auto*

lemma *utp-bottom*: $P \text{ is } \mathcal{H} \implies \perp \sqsubseteq P$

using *weak.bottom-lower* **by** *auto*

end

lemma *upred-top*: $\top_{\mathcal{P}} = \text{false}$

using *ball-UNIV greatest-def* **by** *fastforce*

lemma *upred-bottom*: $\perp_{\mathcal{P}} = \text{true}$

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* \mathcal{H}

```

sublocale utp-theory-mono  $\subseteq$  utp-theory-lattice
proof –
  interpret weak-complete-lattice fpl  $\mathcal{P}$   $\mathcal{H}$ 
    by (rule Knaster-Tarski, auto)

  have complete-lattice (fpl  $\mathcal{P}$   $\mathcal{H}$ )
    by (unfold-locales, simp add: fps-def sup-exists, (blast intro: sup-exists inf-exists)+)

  hence complete-lattice (utp-order  $\mathcal{H}$ )
    by (simp add: utp-order-def, simp add: upred-lattice-def)

  thus utp-theory-lattice  $\mathcal{H}$ 
    by (simp add: utp-theory-axioms utp-theory-lattice.intro utp-theory-lattice-axioms.intro)
qed

```

In a monotone theory, the top and bottom can always be obtained by applying the healthiness condition to the predicate top and bottom, respectively.

```

context utp-theory-mono
begin

```

```

lemma healthy-top:  $\top = \mathcal{H}(\text{false})$ 
proof –
  have  $\top = \top_{fpl} \mathcal{P} \mathcal{H}$ 
    by (simp add: utp-order-fpl)
  also have  $\dots = \mathcal{H} \top_{\mathcal{P}}$ 
    using Knaster-Tarski-idem-extremes(1) [of  $\mathcal{P} \mathcal{H}$ ]
    by (simp add: HCond-Idempotent HCond-Mono)
  also have  $\dots = \mathcal{H} \text{false}$ 
    by (simp add: upred-top)
  finally show ?thesis .
qed

```

```

lemma healthy-bottom:  $\perp = \mathcal{H}(\text{true})$ 
proof –
  have  $\perp = \perp_{fpl} \mathcal{P} \mathcal{H}$ 
    by (simp add: utp-order-fpl)
  also have  $\dots = \mathcal{H} \perp_{\mathcal{P}}$ 
    using Knaster-Tarski-idem-extremes(2) [of  $\mathcal{P} \mathcal{H}$ ]
    by (simp add: HCond-Idempotent HCond-Mono)
  also have  $\dots = \mathcal{H} \text{true}$ 
    by (simp add: upred-bottom)
  finally show ?thesis .
qed

```

```

lemma healthy-inf:
  assumes  $A \subseteq \llbracket \mathcal{H} \rrbracket_H$ 
  shows  $\bigcap A = \mathcal{H} (\bigcap A)$ 
  using Knaster-Tarski-idem-inf-eq [OF upred-weak-complete-lattice, of  $\mathcal{H}$ ]
  by (simp, metis HCond-Idempotent HCond-Mono asms partial-object.simps(3) upred-lattice-def upred-lattice-inf utp-order-def)

```

end

```

locale utp-theory-continuous = utp-theory +

```

assumes *HCond-Cont* [*closure,intro*]: *Continuous* \mathcal{H}

sublocale *utp-theory-continuous* \subseteq *utp-theory-mono*

proof

show *Monotonic* \mathcal{H}

by (*simp add: Continuous-Monotonic HCond-Cont*)

qed

context *utp-theory-continuous*

begin

lemma *healthy-inf-cont*:

assumes $A \subseteq \llbracket \mathcal{H} \rrbracket_H$ $A \neq \{\}$

shows $\sqcap A = \sqcap A$

proof –

have $\sqcap A = \sqcap (\mathcal{H}'A)$

using *Continuous-def HCond-Cont assms(1) assms(2) healthy-inf* **by** *auto*

also have $\dots = \sqcap A$

by (*unfold Healthy-carrier-image[OF assms(1)], simp*)

finally show *?thesis* .

qed

lemma *healthy-inf-def*:

assumes $A \subseteq \llbracket \mathcal{H} \rrbracket_H$

shows $\sqcap A = (\text{if } (A = \{\}) \text{ then } \top \text{ else } (\sqcap A))$

using *assms healthy-inf-cont weak.weak-inf-empty* **by** *auto*

lemma *healthy-meet-cont*:

assumes $P \text{ is } \mathcal{H}$ $Q \text{ is } \mathcal{H}$

shows $P \sqcap Q = P \sqcap Q$

using *healthy-inf-cont[of \{P, Q\} assms*

by (*simp add: Healthy-if meet-def*)

lemma *meet-is-healthy* [*closure*]:

assumes $P \text{ is } \mathcal{H}$ $Q \text{ is } \mathcal{H}$

shows $P \sqcap Q \text{ is } \mathcal{H}$

by (*metis Continuous-Disjunctuous Disjunctuous-def HCond-Cont Healthy-def' assms(1) assms(2)*)

lemma *disj-is-healthy* [*closure*]:

$\llbracket P \text{ is } \mathcal{H}; Q \text{ is } \mathcal{H} \rrbracket \implies (P \vee Q) \text{ is } \mathcal{H}$

by (*simp add: disj-upred-def meet-is-healthy*)

lemma *meet-bottom* [*simp*]:

assumes $P \text{ is } \mathcal{H}$

shows $P \sqcap \perp = \perp$

by (*simp add: assms semilattice-sup-class.sup-absorb2 utp-bottom*)

lemma *meet-top* [*simp*]:

assumes $P \text{ is } \mathcal{H}$

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  $F \in \llbracket \mathcal{H} \rrbracket_H \rightarrow \llbracket \mathcal{H} \rrbracket_H$ 
shows  $\mu F = (\mu X \cdot F(\mathcal{H}(X)))$ 
proof (rule antisym)
  have  $ne: \{P. (P \text{ is } \mathcal{H}) \wedge F P \sqsubseteq P\} \neq \{\}$ 
  proof –
    have  $F \top \sqsubseteq \top$ 
    using assms(2) utp-top weak.top-closed by force
    thus ?thesis
    by (auto, rule-tac x= $\top$  in exI, auto simp add: top-healthy)
  qed
show  $\mu F \sqsubseteq (\mu X \cdot F(\mathcal{H}(X)))$ 
proof –
  have  $\sqcap \{P. (P \text{ is } \mathcal{H}) \wedge F(P) \sqsubseteq P\} \sqsubseteq \sqcap \{P. F(\mathcal{H}(P)) \sqsubseteq P\}$ 
  proof –
    have  $1: \bigwedge P. F(\mathcal{H}(P)) = \mathcal{H}(F(\mathcal{H}(P)))$ 
    by (metis HCond-Idem Healthy-def assms(2) funcset-mem mem-Collect-eq)
  show ?thesis
  proof (rule Sup-least, auto)
    fix  $P$ 
    assume  $a: F(\mathcal{H} P) \sqsubseteq P$ 
    hence  $F: (F(\mathcal{H} P)) \sqsubseteq (\mathcal{H} P)$ 
    by (metis 1 HCond-Mono mono-def)
    show  $\sqcap \{P. (P \text{ is } \mathcal{H}) \wedge F P \sqsubseteq P\} \sqsubseteq P$ 
    proof (rule Sup-upper2[of F ( $\mathcal{H} P$ )])
      show  $F(\mathcal{H} P) \in \{P. (P \text{ is } \mathcal{H}) \wedge F P \sqsubseteq P\}$ 
      proof (auto)
        show  $F(\mathcal{H} P) \text{ is } \mathcal{H}$ 
        by (metis 1 Healthy-def)
        show  $F(F(\mathcal{H} P)) \sqsubseteq F(\mathcal{H} P)$ 
        using F mono-def assms(1) by blast
      qed
      show  $F(\mathcal{H} P) \sqsubseteq P$ 
      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  $(\mu X \cdot F(\mathcal{H}(X))) \sqsubseteq \mu F$ 
apply (simp add: LEAST-FP-def gfp-def, subst healthy-inf-cont, auto simp add: lfp-def)
apply (rule Sup-least)
apply (auto simp add: Healthy-def Sup-upper)
done
qed

lemma UINF-ind-Healthy [closure]:
assumes  $\bigwedge i. P(i) \text{ is } \mathcal{H}$ 
shows  $(\sqcap i \cdot P(i)) \text{ is } \mathcal{H}$ 
by (simp add: closure assms)

```

end

In another direction, we can also characterise UTP theories that are relational. Minimally this requires that the healthiness condition is closed under sequential composition.

```

locale utp-theory-rel =
  utp-theory +
  assumes Healthy-Sequence [closure]:  $\llbracket P \text{ is } \mathcal{H}; Q \text{ is } \mathcal{H} \rrbracket \implies (P ;; Q) \text{ is } \mathcal{H}$ 
begin

```

```

lemma upower-Suc-Healthy [closure]:
  assumes  $P \text{ is } \mathcal{H}$ 
  shows  $P \hat{\ } \text{Suc } n \text{ is } \mathcal{H}$ 
  by (induct n, simp-all add: closure assms upred-semiring.power-Suc)

```

end

```

locale utp-theory-cont-rel = utp-theory-rel + utp-theory-continuous
begin

```

```

lemma seq-cont-Sup-distl:
  assumes  $P \text{ is } \mathcal{H} \ A \subseteq \llbracket \mathcal{H} \rrbracket_H \ A \neq \{\}$ 
  shows  $P ;; (\bigcap A) = \bigcap \{P ;; Q \mid Q. Q \in A\}$ 
proof –
  have  $\{P ;; Q \mid Q. Q \in A\} \subseteq \llbracket \mathcal{H} \rrbracket_H$ 
    using Healthy-Sequence assms(1) assms(2) by (auto)
  thus ?thesis
    by (simp add: healthy-inf-cont seq-Sup-distl setcompr-eq-image assms)
qed

```

```

lemma seq-cont-Sup-distr:
  assumes  $Q \text{ is } \mathcal{H} \ A \subseteq \llbracket \mathcal{H} \rrbracket_H \ A \neq \{\}$ 
  shows  $(\bigcap A) ;; Q = \bigcap \{P ;; Q \mid P. P \in A\}$ 
proof –
  have  $\{P ;; Q \mid P. P \in A\} \subseteq \llbracket \mathcal{H} \rrbracket_H$ 
    using Healthy-Sequence assms(1) assms(2) by (auto)
  thus ?thesis
    by (simp add: healthy-inf-cont seq-Sup-distr setcompr-eq-image assms)
qed

```

```

lemma uplus-healthy [closure]:
  assumes  $P \text{ is } \mathcal{H}$ 
  shows  $P^+ \text{ is } \mathcal{H}$ 
  by (simp add: uplus-power-def closure assms)

```

end

There also exist UTP theories with units. Not all theories have both a left and a right unit (e.g. H1-H2 designs) and so we split up the locale into two cases.

```

locale utp-theory-units =
  utp-theory-rel +
  fixes utp-unit ( $\mathcal{I}\mathcal{I}$ )
  assumes Healthy-Unit [closure]:  $\mathcal{I}\mathcal{I} \text{ is } \mathcal{H}$ 
begin

```

We can characterise the theory Kleene star by lifting the relational one.

```

definition utp-star ( $-\star$  [999] 999) where
  [upred-defs]: utp-star  $P = (P^* ;; \mathcal{I}\mathcal{I})$ 

```

We can then characterise tests as refinements of units.

```

definition utp-test :: 'a hrel  $\Rightarrow$  bool where
[upred-defs]: utp-test b = ( $\mathcal{II} \sqsubseteq b$ )

end

locale utp-theory-left-unital =
  utp-theory-units +
  assumes Unit-Left:  $P \text{ is } \mathcal{H} \implies (\mathcal{II} ;; P) = P$ 

locale utp-theory-right-unital =
  utp-theory-units +
  assumes Unit-Right:  $P \text{ is } \mathcal{H} \implies (P ;; \mathcal{II}) = P$ 

locale utp-theory-unital =
  utp-theory-left-unital + utp-theory-right-unital
begin

lemma Unit-self [simp]:
   $\mathcal{II} ;; \mathcal{II} = \mathcal{II}$ 
  by (simp add: Healthy-Unit Unit-Right)

lemma utest-intro:
   $\mathcal{II} \sqsubseteq P \implies \text{utp-test } P$ 
  by (simp add: utp-test-def)

lemma utest-Unit [closure]:
  utp-test  $\mathcal{II}$ 
  by (simp add: utp-test-def)

end

locale utp-theory-mono-unital = utp-theory-unital + utp-theory-mono
begin

lemma utest-Top [closure]: utp-test  $\top$ 
  by (simp add: Healthy-Unit utp-test-def utp-top)

end

locale utp-theory-cont-unital = utp-theory-cont-rel + utp-theory-unital

sublocale utp-theory-cont-unital  $\subseteq$  utp-theory-mono-unital
  by (simp add: utp-theory-mono-axioms utp-theory-mono-unital-def utp-theory-unital-axioms)

locale utp-theory-unital-zero =
  utp-theory-unital +
  utp-theory-lattice +
  assumes Top-Left-Zero:  $P \text{ is } \mathcal{H} \implies \top ;; P = \top$ 

locale utp-theory-cont-unital-zero =
  utp-theory-cont-unital + utp-theory-unital-zero
begin

lemma Top-test-Right-Zero:
  assumes  $b \text{ is } \mathcal{H}$  utp-test b

```

```

shows b ;; T = T
proof -
  have b  $\sqcap$  II = II
    by (meson assms(2) semilattice-sup-class.le-iff-sup utp-test-def)
  then show ?thesis
    by (metis (no-types) Top-Left-Zero Unit-Left assms(1) meet-top top-healthy upred-semiring.distrib-right)
qed

end

```

19.4 Theory of relations

```

interpretation rel-theory: utp-theory-mono-unital id skip-r
  rewrites rel-theory.utp-top = false
  and rel-theory.utp-bottom = true
  and carrier (utp-order id) = UNIV
  and (P is id) = True
proof -
  show utp-theory-mono-unital id II
    by (unfold-locales, simp-all add: Healthy-def)
  then interpret utp-theory-mono-unital id skip-r
    by simp
  show utp-top = false utp-bottom = true
    by (simp-all add: healthy-top healthy-bottom)
  show carrier (utp-order id) = UNIV (P is id) = True
    by (auto simp add: utp-order-def Healthy-def)
qed

```

thm *rel-theory.GFP-unfold*

19.5 Theory links

We can also describe links between theories, such a Galois connections and retractions, using the following notation.

definition *mk-conn* $(- \Leftarrow \langle -, \rangle \Rightarrow - [90, 0, 0, 91] \ 91)$ **where**
 $H1 \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \Rightarrow H2 \equiv () \text{ orderA} = \text{utp-order } H1, \text{ orderB} = \text{utp-order } H2, \text{ lower} = \mathcal{H}_2, \text{ upper} = \mathcal{H}_1 \ ()$

lemma *mk-conn-orderA* [simp]: $\mathcal{X}_{H1} \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \Rightarrow H2 = \text{utp-order } H1$
 by (simp add: mk-conn-def)

lemma *mk-conn-orderB* [simp]: $\mathcal{Y}_{H1} \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \Rightarrow H2 = \text{utp-order } H2$
 by (simp add: mk-conn-def)

lemma *mk-conn-lower* [simp]: $\pi_* H1 \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \Rightarrow H2 = \mathcal{H}_1$
 by (simp add: mk-conn-def)

lemma *mk-conn-upper* [simp]: $\pi^* H1 \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \Rightarrow H2 = \mathcal{H}_2$
 by (simp add: mk-conn-def)

lemma *galois-comp*: $(H2 \Leftarrow \langle \mathcal{H}_3, \mathcal{H}_4 \rangle \Rightarrow H3) \circ_g (H1 \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \Rightarrow H2) = H1 \Leftarrow \langle \mathcal{H}_1 \circ \mathcal{H}_3, \mathcal{H}_4 \circ \mathcal{H}_2 \rangle \Rightarrow H3$
 by (simp add: comp-galcon-def mk-conn-def)

Example Galois connection / retract: Existential quantification

lemma *Idempotent-ex*: $\text{mwb-lens } x \implies \text{Idempotent } (\text{ex } x)$

by (simp add: Idempotent-def exists-twice)

lemma *Monotonic-ex: mwb-lens $x \implies \text{Monotonic } (ex\ x)$*
 by (simp add: mono-def ex-mono)

lemma *ex-closed-unrest:*
 $mwb\text{-}lens\ x \implies \llbracket ex\ x \rrbracket_H = \{P. x \# P\}$
 by (simp add: Healthy-def unrest-as-exists)

Any theory can be composed with an existential quantification to produce a Galois connection

theorem *ex-retract:*
assumes *$mwb\text{-}lens\ x$ Idempotent H $ex\ x \circ H = H \circ ex\ x$*
shows *$retract\ ((ex\ x \circ H) \Leftarrow (ex\ x, H) \Rightarrow H)$*
proof (unfold-locales, simp-all)
show $H \in \llbracket ex\ x \circ H \rrbracket_H \rightarrow \llbracket H \rrbracket_H$
using *Healthy-Idempotent assms* **by** blast
from *assms(1) assms(3)[THEN sym]* **show** $ex\ x \in \llbracket H \rrbracket_H \rightarrow \llbracket ex\ x \circ H \rrbracket_H$
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 $(H\ P \sqsubseteq Q) = (P \sqsubseteq (\exists\ x \cdot Q))$
by (metis (no-types, lifting) Healthy-Idempotent Healthy-if assms comp-apply dual-order.trans ex-weakens
 utp-pred-laws.ex-mono mwb-lens-wb)
next
fix P
assume *P is $(ex\ x \circ H)$*
thus $(\exists\ x \cdot H\ P) \sqsubseteq P$
by (simp add: Healthy-def)
qed

corollary *ex-retract-id:*
assumes *$mwb\text{-}lens\ x$*
shows *$retract\ (ex\ x \Leftarrow (ex\ x, id) \Rightarrow id)$*
using *assms ex-retract[where $H=id$]* **by** (auto)
end

20 Relational Hoare calculus

theory *utp-hoare*
imports
utp-rel-laws
utp-theory
begin

20.1 Hoare Triple Definitions and Tactics

definition *hoare-r :: ' α cond \Rightarrow ' α hrel \Rightarrow ' α cond \Rightarrow bool ($\{\cdot\}/\cdot/\{\cdot\}_u$) where*
 $\{p\}Q\{r\}_u = (([p]_{<} \Rightarrow [r]_{>}) \sqsubseteq Q)$

declare *hoare-r-def* [*upred-defs*]

named-theorems *hoare* **and** *hoare-safe*

method *hoare-split* **uses** *hr =*
((simp add: assigns-comp)?, — Combine Assignments where possible)

(*auto*
intro: hoare intro!: hoare-safe hr
simpl add: conj-comm conj-assoc usubst unrest))[1] — Apply Hoare logic laws

method *hoare-auto* **uses** *hr* = (*hoare-split hr: hr; (rel-simp)?, auto?*)

20.2 Basic Laws

lemma *hoare-meaning*:

$\llbracket P \rrbracket S \llbracket Q \rrbracket_u = (\forall s s'. \llbracket P \rrbracket_e s \wedge \llbracket S \rrbracket_e (s, s') \longrightarrow \llbracket Q \rrbracket_e s')$
by (*rel-auto*)

lemma *hoare-assume*: $\llbracket P \rrbracket S \llbracket Q \rrbracket_u \Longrightarrow ?[P] ;; S = ?[P] ;; S ;; ?[Q]$
by (*rel-auto*)

lemma *hoare-test* [*hoare-safe*]: $'p \wedge b \Rightarrow q' \Longrightarrow \llbracket p \rrbracket ?[b] \llbracket q \rrbracket_u$
by (*rel-simp*)

lemma *hoare-gcmd* [*hoare-safe*]: $\llbracket p \wedge b \rrbracket P \llbracket q \rrbracket_u \Longrightarrow \llbracket p \rrbracket b \longrightarrow_r P \llbracket q \rrbracket_u$
by (*rel-auto*)

lemma *hoare-r-conj* [*hoare-safe*]: $\llbracket \llbracket p \rrbracket Q \llbracket r \rrbracket_u; \llbracket p \rrbracket Q \llbracket s \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket Q \llbracket r \wedge s \rrbracket_u$
by *rel-auto*

lemma *hoare-r-weaken-pre* [*hoare*]:

$\llbracket p \rrbracket Q \llbracket r \rrbracket_u \Longrightarrow \llbracket p \wedge q \rrbracket Q \llbracket r \rrbracket_u$
 $\llbracket q \rrbracket Q \llbracket r \rrbracket_u \Longrightarrow \llbracket p \wedge q \rrbracket Q \llbracket r \rrbracket_u$
by *rel-auto+*

lemma *pre-str-hoare-r*:

assumes $'p_1 \Rightarrow p_2'$ **and** $\llbracket p_2 \rrbracket C \llbracket q \rrbracket_u$
shows $\llbracket p_1 \rrbracket C \llbracket q \rrbracket_u$
using *assms* **by** *rel-auto*

lemma *post-weak-hoare-r*:

assumes $\llbracket p \rrbracket C \llbracket q_2 \rrbracket_u$ **and** $'q_2 \Rightarrow q_1'$
shows $\llbracket p \rrbracket C \llbracket q_1 \rrbracket_u$
using *assms* **by** *rel-auto*

lemma *hoare-r-conseq*: $\llbracket 'p_1 \Rightarrow p_2'; \llbracket p_2 \rrbracket S \llbracket q_2 \rrbracket_u; 'q_2 \Rightarrow q_1' \rrbracket \Longrightarrow \llbracket p_1 \rrbracket S \llbracket q_1 \rrbracket_u$
by *rel-auto*

20.3 Assignment Laws

lemma *assigns-hoare-r* [*hoare-safe*]: $'p \Rightarrow \sigma \dagger q' \Longrightarrow \llbracket p \rrbracket \langle \sigma \rangle_a \llbracket q \rrbracket_u$
by *rel-auto*

lemma *assigns-backward-hoare-r*:

$\llbracket \sigma \dagger p \rrbracket \langle \sigma \rangle_a \llbracket p \rrbracket_u$
by *rel-auto*

lemma *assign-floyd-hoare-r*:

assumes *vwb-lens x*
shows $\llbracket p \rrbracket \text{assign-r } x \text{ e } \llbracket \exists v \cdot p[\langle v \rangle / x] \wedge \&x =_u e[\langle v \rangle / x] \rrbracket_u$
using *assms*
by (*rel-auto, metis vwb-lens-wb wb-lens.get-put*)

lemma *assigns-init-hoare* [*hoare-safe*]:
 $\llbracket \text{vwb-lens } x; x \# p; x \# v; \&x =_u v \wedge p \rrbracket S \llbracket q \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket x := v ;; S \llbracket q \rrbracket_u$
 by (*rel-auto*)

lemma *skip-hoare-r* [*hoare-safe*]: $\llbracket p \rrbracket II \llbracket p \rrbracket_u$
 by *rel-auto*

lemma *skip-hoare-impl-r* [*hoare-safe*]: $'p \Rightarrow q' \Longrightarrow \llbracket p \rrbracket II \llbracket q \rrbracket_u$
 by *rel-auto*

20.4 Sequence Laws

lemma *seq-hoare-r*: $\llbracket \llbracket p \rrbracket Q_1 \llbracket s \rrbracket_u ; \llbracket s \rrbracket Q_2 \llbracket r \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket Q_1 ;; Q_2 \llbracket r \rrbracket_u$
 by *rel-auto*

lemma *seq-hoare-invariant* [*hoare-safe*]: $\llbracket \llbracket p \rrbracket Q_1 \llbracket p \rrbracket_u ; \llbracket p \rrbracket Q_2 \llbracket p \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket Q_1 ;; Q_2 \llbracket p \rrbracket_u$
 by *rel-auto*

lemma *seq-hoare-stronger-pre-1* [*hoare-safe*]:
 $\llbracket \llbracket p \wedge q \rrbracket Q_1 \llbracket p \wedge q \rrbracket_u ; \llbracket p \wedge q \rrbracket Q_2 \llbracket q \rrbracket_u \rrbracket \Longrightarrow \llbracket p \wedge q \rrbracket Q_1 ;; Q_2 \llbracket q \rrbracket_u$
 by *rel-auto*

lemma *seq-hoare-stronger-pre-2* [*hoare-safe*]:
 $\llbracket \llbracket p \wedge q \rrbracket Q_1 \llbracket p \wedge q \rrbracket_u ; \llbracket p \wedge q \rrbracket Q_2 \llbracket p \rrbracket_u \rrbracket \Longrightarrow \llbracket p \wedge q \rrbracket Q_1 ;; Q_2 \llbracket p \rrbracket_u$
 by *rel-auto*

lemma *seq-hoare-inv-r-2* [*hoare*]: $\llbracket \llbracket p \rrbracket Q_1 \llbracket q \rrbracket_u ; \llbracket q \rrbracket Q_2 \llbracket q \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket Q_1 ;; Q_2 \llbracket q \rrbracket_u$
 by *rel-auto*

lemma *seq-hoare-inv-r-3* [*hoare*]: $\llbracket \llbracket p \rrbracket Q_1 \llbracket p \rrbracket_u ; \llbracket p \rrbracket Q_2 \llbracket q \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket Q_1 ;; Q_2 \llbracket q \rrbracket_u$
 by *rel-auto*

20.5 Conditional Laws

lemma *cond-hoare-r* [*hoare-safe*]: $\llbracket \llbracket b \wedge p \rrbracket S \llbracket q \rrbracket_u ; \llbracket \neg b \wedge p \rrbracket T \llbracket q \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket S \triangleleft b \triangleright_r T \llbracket q \rrbracket_u$
 by *rel-auto*

lemma *cond-hoare-r-wp*:
 assumes $\llbracket p' \rrbracket S \llbracket q \rrbracket_u$ and $\llbracket p' \rrbracket T \llbracket q \rrbracket_u$
 shows $\llbracket (b \wedge p') \vee (\neg b \wedge p'') \rrbracket S \triangleleft b \triangleright_r T \llbracket q \rrbracket_u$
 using *assms* by *pred-simp*

lemma *cond-hoare-r-sp*:
 assumes $\langle \llbracket b \wedge p \rrbracket S \llbracket q \rrbracket_u \rangle$ and $\langle \llbracket \neg b \wedge p \rrbracket T \llbracket s \rrbracket_u \rangle$
 shows $\langle \llbracket p \rrbracket S \triangleleft b \triangleright_r T \llbracket q \vee s \rrbracket_u \rangle$
 using *assms* by *pred-simp*

20.6 Recursion Laws

lemma *nu-hoare-r-partial*:
 assumes *induct-step*:
 $\bigwedge st P. \llbracket p \rrbracket P \llbracket q \rrbracket_u \Longrightarrow \llbracket p \rrbracket F P \llbracket q \rrbracket_u$
 shows $\llbracket p \rrbracket \nu F \llbracket q \rrbracket_u$
 by (*meson* *hoare-r-def* *induct-step* *lfp-lowerbound* *order-refl*)

lemma *mu-hoare-r*:
assumes *WF*: $wf\ R$
assumes *M*:*mono* *F*
assumes *induct-step*:
 $\bigwedge st\ P. \llbracket p \wedge (e, \llbracket st \rrbracket)_u \in_u \llbracket R \rrbracket \rrbracket P \llbracket q \rrbracket_u \implies \llbracket p \wedge e =_u \llbracket st \rrbracket \rrbracket F\ P \llbracket q \rrbracket_u$
shows $\llbracket p \rrbracket_\mu F \llbracket q \rrbracket_u$
unfolding *hoare-r-def*
proof (*rule mu-rec-total-utp-rule* [*OF* *WF* *M* , *of* - *e*], *goal-cases*)
case (*1 st*)
then show *?case*
using *induct-step* [*unfolded hoare-r-def*, *of* ($\llbracket p \rrbracket_< \wedge (\llbracket e \rrbracket_<, \llbracket st \rrbracket)_u \in_u \llbracket R \rrbracket \implies \llbracket q \rrbracket_> \rrbracket st$)]
by (*simp add: alpha*)
qed

lemma *mu-hoare-r'*:
assumes *WF*: $wf\ R$
assumes *M*:*mono* *F*
assumes *induct-step*:
 $\bigwedge st\ P. \llbracket p \wedge (e, \llbracket st \rrbracket)_u \in_u \llbracket R \rrbracket \rrbracket P \llbracket q \rrbracket_u \implies \llbracket p \wedge e =_u \llbracket st \rrbracket \rrbracket F\ P \llbracket q \rrbracket_u$
assumes *I0*: $'p' \Rightarrow p'$
shows $\llbracket p' \rrbracket_\mu F \llbracket q \rrbracket_u$
by (*meson I0 M WF induct-step mu-hoare-r pre-str-hoare-r*)

20.7 Iteration Rules

lemma *iter-hoare-r* [*hoare-safe*]: $\llbracket P \rrbracket S \llbracket P \rrbracket_u \implies \llbracket P \rrbracket S^* \llbracket P \rrbracket_u$
by (*rel-simp'*, *metis* (*mono-tags*, *lifting*) *mem-Collect-eq rtrancl-induct*)

lemma *while-hoare-r* [*hoare-safe*]:
assumes $\llbracket p \wedge b \rrbracket S \llbracket p \rrbracket_u$
shows $\llbracket p \rrbracket \text{while } b \text{ do } S \text{ od } \llbracket \neg b \wedge p \rrbracket_u$
using *assms*
by (*simp add: while-top-def hoare-r-def*, *rule-tac lfp-lowerbound*) (*rel-auto*)

lemma *while-invr-hoare-r* [*hoare-safe*]:
assumes $\llbracket p \wedge b \rrbracket S \llbracket p \rrbracket_u$ $'pre \Rightarrow p'$ $'(\neg b \wedge p) \Rightarrow post'$
shows $\llbracket pre \rrbracket \text{while } b \text{ invr } p \text{ do } S \text{ od } \llbracket post \rrbracket_u$
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*: $\llbracket invar \wedge b \rrbracket C \llbracket invar \rrbracket_u$
shows $\llbracket p \rrbracket \text{while } b \text{ do } C \text{ od } \llbracket \neg b \wedge invar \rrbracket_u$
using *induct-step pre-str-hoare-r seq-step while-hoare-r* **by** *blast*

lemma *approx-chain*:
 $(\bigcap n::nat. \llbracket p \wedge v <_u \llbracket n \rrbracket \rrbracket_<) = \llbracket p \rrbracket_<$
by (*rel-auto*)

Total correctness law for Hoare logic, based on constructive chains. This is limited to variants that have natural numbers as their range.

lemma *while-term-hoare-r*:
assumes $\bigwedge z::nat. \llbracket p \wedge b \wedge v =_u \llbracket z \rrbracket \rrbracket S \llbracket p \wedge v <_u \llbracket z \rrbracket \rrbracket_u$
shows $\llbracket p \rrbracket \text{while}_\perp b \text{ do } S \text{ od } \llbracket \neg b \wedge p \rrbracket_u$
proof –

```

have ( $\lceil p \rceil_{<} \Rightarrow \lceil \neg b \wedge p \rceil_{>}$ )  $\sqsubseteq$  ( $\mu X. S ;; X \triangleleft b \triangleright_r II$ )
proof (rule mu-refine-intro)

from assms show ( $\lceil p \rceil_{<} \Rightarrow \lceil \neg b \wedge p \rceil_{>}$ )  $\sqsubseteq S ;; (\lceil p \rceil_{<} \Rightarrow \lceil \neg b \wedge p \rceil_{>}) \triangleleft b \triangleright_r II$ 
  by (rel-auto)

let ?E =  $\lambda n. \lceil p \wedge v <_u \ll n \gg \rceil_{<}$ 
show ( $\lceil p \rceil_{<} \wedge (\mu X. S ;; X \triangleleft b \triangleright_r II)$ ) = ( $\lceil p \rceil_{<} \wedge (\nu X. S ;; X \triangleleft b \triangleright_r II)$ )
proof (rule constr-fp-uniq[where E=?E])

  show ( $\bigcap n. ?E(n)$ ) =  $\lceil p \rceil_{<}$ 
    by (rel-auto)

  show mono ( $\lambda X. S ;; X \triangleleft b \triangleright_r II$ )
    by (simp add: cond-mono monoI segr-mono)

  show constr ( $\lambda X. S ;; X \triangleleft b \triangleright_r II$ ) ?E
  proof (rule constrI)

    show chain ?E
    proof (rule chainI)
      show  $\lceil p \wedge v <_u \ll 0 \gg \rceil_{<} = \text{false}$ 
        by (rel-auto)
      show  $\bigwedge i. \lceil p \wedge v <_u \ll \text{Suc } i \gg \rceil_{<} \sqsubseteq \lceil p \wedge v <_u \ll i \gg \rceil_{<}$ 
        by (rel-auto)
    qed

    from assms
    show  $\bigwedge X n. (S ;; X \triangleleft b \triangleright_r II \wedge \lceil p \wedge v <_u \ll n + 1 \gg \rceil_{<}) =$ 
      ( $S ;; (X \wedge \lceil p \wedge v <_u \ll n \gg \rceil_{<}) \triangleleft b \triangleright_r II \wedge \lceil p \wedge v <_u \ll n + 1 \gg \rceil_{<}$ )
      apply (rel-auto)
      using less-antisym less-trans apply blast
    done
  qed
qed
qed
qed

thus ?thesis
  by (simp add: hoare-r-def while-bot-def)
qed

```

```

lemma while-vrt-hoare-r [hoare-safe]:
  assumes  $\bigwedge z::\text{nat}. \llbracket p \wedge b \wedge v =_u \ll z \gg \rrbracket S \llbracket p \wedge v <_u \ll z \gg \rrbracket_u \text{'pre} \Rightarrow p \text{' } (\neg b \wedge p) \Rightarrow \text{post'}$ 
  shows  $\llbracket \text{pre} \rrbracket \text{while } b \text{ invr } p \text{ vrt } v \text{ do } S \text{ od} \llbracket \text{post} \rrbracket_u$ 
  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:  $\text{'pre} \Rightarrow p \text{'}$ 
  assumes induct-step:  $\bigwedge st. \llbracket b \wedge p \wedge e =_u \ll st \gg \rrbracket Q \llbracket p \wedge (e, \ll st \gg)_u \in_u \ll R \gg \rrbracket_u$ 
  assumes PHI:  $\text{'pre} \wedge p \Rightarrow \text{post'}$ 

```

```

  shows  $\{pre\} \text{while}_\perp b \text{ invr } p \text{ do } Q \text{ od } \{post\}_u$ 
unfolding hoare-r-def while-inv-bot-def while-bot-def
proof (rule pre-weak-rel[of -  $\lceil p \rceil_<$  ])
  from I0 show ' $\lceil pre \rceil_< \Rightarrow \lceil p \rceil_<$ '
    by rel-auto
  show  $(\lceil p \rceil_< \Rightarrow \lceil post \rceil_>) \sqsubseteq (\mu X. X \cdot Q ;; X \triangleleft b \triangleright_r II)$ 
proof (rule mu-rec-total-utp-rule[where  $e=e$ , OF WF])
  show Monotonic  $(\lambda X. X \cdot Q ;; X \triangleleft b \triangleright_r II)$ 
    by (simp add: closure)
  have induct-step':  $\bigwedge st. (\lceil b \wedge p \wedge e =_u \ll st \gg \rceil_< \Rightarrow (\lceil p \wedge (e, \ll st \gg)_u \in_u \ll R \gg \rceil_>)) \sqsubseteq Q$ 
    using induct-step by rel-auto
  with PHI
  show  $\bigwedge st. (\lceil p \rceil_< \wedge \lceil e \rceil_< =_u \ll st \gg \Rightarrow \lceil post \rceil_>) \sqsubseteq Q ;; (\lceil p \rceil_< \wedge (\lceil e \rceil_<, \ll st \gg)_u \in_u \ll R \gg \Rightarrow \lceil post \rceil_>)$ 
 $\triangleleft b \triangleright_r II$ 
    by (rel-auto)
qed
qed

```

20.8 Frame Rules

Frame rule: If starting S in a state satisfying $pestablisheq$ 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  $vwb\text{-}lens\ a\ a \# r\ a \triangleleft q\ \{p\}P\{q\}_u$ 
  shows  $\{p \wedge r\} a : [P] \{q \wedge r\}_u$ 
  using assms
  by (rel-auto, metis)

```

lemma *frame-strong-hoare-r* [*hoare-safe*]:

```

  assumes  $vwb\text{-}lens\ a\ a \# r\ a \triangleleft q\ \{p \wedge r\}S\{q\}_u$ 
  shows  $\{p \wedge r\} a : [S] \{q \wedge r\}_u$ 
  using assms by (rel-auto, metis)

```

lemma *frame-hoare-r'* [*hoare-safe*]:

```

  assumes  $vwb\text{-}lens\ a\ a \# r\ a \triangleleft q\ \{r \wedge p\}S\{q\}_u$ 
  shows  $\{r \wedge p\} a : [S] \{r \wedge q\}_u$ 
  using assms
  by (simp add: frame-strong-hoare-r utp-pred-laws.inf commute)

```

lemma *antiframe-hoare-r*:

```

  assumes  $vwb\text{-}lens\ a\ a \triangleleft r\ a \# q\ \{p\}P\{q\}_u$ 
  shows  $\{p \wedge r\} a : [P] \{q \wedge r\}_u$ 
  using assms by (rel-auto, metis)

```

lemma *antiframe-strong-hoare-r*:

```

  assumes  $vwb\text{-}lens\ a\ a \triangleleft r\ a \# q\ \{p \wedge r\}P\{q\}_u$ 
  shows  $\{p \wedge r\} a : [P] \{q \wedge r\}_u$ 
  using assms by (rel-auto, metis)

```

end

21 Weakest Liberal Precondition Calculus

```
theory utp-wlp
imports utp-hoare
begin
```

The calculus we here define is termed “weakest precondition” in the UTP book, however it is in reality the liberal version that does not account for termination.

```
named-theorems wp
```

```
method wp-tac = (simp add: wp)
```

```
consts
  uwlp :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'c
```

```
syntax
  -uwlp :: logic  $\Rightarrow$  uexp  $\Rightarrow$  logic (infix wlp 60)
```

```
translations
  -uwlp P b == CONST uwlp P b
```

```
definition wlp-upred :: (' $\alpha$ , ' $\beta$ ) urel  $\Rightarrow$  ' $\beta$  cond  $\Rightarrow$  ' $\alpha$  cond where
wlp-upred Q r =  $\lfloor \neg (Q ;; (\neg [r]_{<})) \rfloor :: (' $\alpha$ , ' $\beta$ ) urel]_{<}$ 
```

```
adhoc-overloading
  uwlp wlp-upred
```

```
declare wlp-upred-def [urel-defs]
```

```
lemma wlp-true [wp]: p wlp true = true
  by (rel-simp)
```

```
theorem wlp-assigns-r [wp]:
   $\langle \sigma \rangle_a$  wlp r =  $\sigma \uparrow r$ 
  by rel-auto
```

```
theorem wlp-skip-r [wp]:
   $\Pi$  wlp r = r
  by rel-auto
```

```
theorem wlp-abort [wp]:
   $r \neq \text{true} \implies \text{true wlp } r = \text{false}$ 
  by rel-auto
```

```
theorem wlp-conj [wp]:
  P wlp (q  $\wedge$  r) = (P wlp q  $\wedge$  P wlp r)
  by rel-auto
```

```
theorem wlp-seq-r [wp]: (P ;; Q) wlp r = P wlp (Q wlp r)
  by rel-auto
```

```
theorem wlp-choice [wp]: (P  $\sqcap$  Q) wlp R = (P wlp R  $\wedge$  Q wlp R)
  by (rel-auto)
```

```
theorem wlp-cond [wp]: (P  $\triangleleft$  b  $\triangleright_r$  Q) wlp r = ((b  $\Rightarrow$  P wlp r)  $\wedge$  (( $\neg$  b)  $\Rightarrow$  Q wlp r))
  by rel-auto
```

lemma *wlp-test* [wp]: $?[b] \text{ wlp } c = (b \Rightarrow c)$
by (*rel-auto*)

lemma *wlp-gcmd* [wp]: $(b \longrightarrow_r P) \text{ wlp } c = (b \Rightarrow P \text{ wlp } c)$
by (*simp add: rgcmd-def wp*)

lemma *wlp-USUP-pre* [wp]: $P \text{ wlp } (\bigsqcup_{i \in \{0..n\}} Q(i)) = (\bigsqcup_{i \in \{0..n\}} P \text{ wlp } Q(i))$
by (*rel-auto*)

theorem *wlp-hoare-link*:
 $\{p\} Q \{r\}_u \longleftrightarrow (Q \text{ wlp } r \sqsubseteq p)$
by *rel-auto*

If two programs have the same weakest precondition for any postcondition then the programs are the same.

theorem *wlp-eq-intro*: $\llbracket \bigwedge r. P \text{ wlp } r = Q \text{ wlp } r \rrbracket \Longrightarrow P = Q$
by (*rel-auto robust, fastforce+*)
end

22 Weakest Precondition Calculus

theory *utp-wp*
imports *utp-wlp*
begin

This calculus is like the liberal version, but also accounts for termination. It is equivalent to the relational preimage.

consts
 $uwp :: 'a \Rightarrow 'b \Rightarrow 'c$

syntax
 $-uwp :: \text{logic} \Rightarrow \text{uexp} \Rightarrow \text{logic} \text{ (infix wp 60)}$

translations
 $-uwp P b == \text{CONST } uwp P b$

definition *wp-upred* :: $'a \text{ hrel} \Rightarrow 'a \text{ upred} \Rightarrow 'a \text{ upred}$ **where**
 $[upred-defs]: \text{wp-upred } P b = \text{Dom}(P ;; ?[b])$

adhoc-overloading
 $uwp \text{ wp-upred}$

lemma *wp-true*: $P \text{ wp } \text{true} = \text{Dom}(P)$
by (*rel-auto*)

lemma *wp-false* [wp]: $P \text{ wp } \text{false} = \text{false}$
by (*rel-auto*)

lemma *wp-abort* [wp]: $\text{false} \text{ wp } b = \text{false}$
by (*rel-auto*)

lemma *wp-seq* [wp]: $(P ;; Q) \text{ wp } b = P \text{ wp } (Q \text{ wp } b)$
by (*simp add: wp-upred-def, metis Dom-seq RA1*)

lemma *wp-disj* [wp]: $(P \vee Q) \text{ wp } b = (P \text{ wp } b \vee Q \text{ wp } b)$
by (*rel-auto*)

lemma *wp-UINF-mem* [wp]: $(\prod_{i \in I} i \cdot P(i)) \text{ wp } b = (\prod_{i \in I} i \cdot P(i) \text{ wp } b)$
by (*rel-auto*)

lemma *wp-UINF-ind* [wp]: $(\prod i \cdot P(i)) \text{ wp } b = (\prod i \cdot P(i) \text{ wp } b)$
by (*rel-auto*)

lemma *wp-UINF-ind-2* [wp]: $(\prod (i, j) \cdot P \ i \ j) \text{ wp } b = (\bigvee (i, j) \cdot (P \ i \ j) \text{ wp } b)$
by (*rel-auto*)

lemma *wp-UINF-ind-3* [wp]: $(\prod (i, j, k) \cdot P \ i \ j \ k) \text{ wp } b = (\bigvee (i, j, k) \cdot (P \ i \ j \ k) \text{ wp } b)$
by (*rel-blast*)

lemma *wp-test* [wp]: $?[b] \text{ wp } c = (b \wedge c)$
by (*rel-auto*)

lemma *wp-gcmd* [wp]: $(b \longrightarrow_r P) \text{ wp } c = (b \wedge P \text{ wp } c)$
by (*rel-auto*)

lemma *wp-assigns* [wp]: $\langle \sigma \rangle_a \text{ wp } b = \sigma \dagger b$
by (*rel-auto*)

lemma *wp-nd-assign* [wp]: $(x := *) \text{ wp } b = (\bigvee v \cdot b[\ll v \gg / \&x])$
by (*simp add: nd-assign-def wp, rel-auto*)

lemma *wp-rel-fnext* [wp]:
assumes *vwb-lens* $a \ a \ \sharp \ q$
shows $a:[P]^+ \text{ wp } (p \oplus_p a \wedge q) = ((P \text{ wp } p) \oplus_p a \wedge q)$
using *assms*
by (*rel-auto, metis (full-types), metis mwb-lens-def vwb-lens-mwb weak-lens.put-get*)

lemma *wp-rel-aext-unrest* [wp]: $\ll vwb\text{-lens } a; a \ \sharp \ b \rrbracket \Longrightarrow a:[P]^+ \text{ wp } b = (b \wedge (P \text{ wp } true) \oplus_p a)$
by (*rel-auto, metis, metis mwb-lens-def vwb-lens-mwb weak-lens.put-get*)

lemma *wp-rel-aext-usedby* [wp]: $\ll vwb\text{-lens } a; a \ \sharp \ b \rrbracket \Longrightarrow a:[P]^+ \text{ wp } b = (P \text{ wp } (b \restriction_e a)) \oplus_p a$
by (*rel-auto, metis mwb-lens-def vwb-lens-mwb weak-lens.put-get*)

lemma *wp-wlp-conjugate*: $P \text{ wp } b = (\neg P \text{ wlp } (\neg b))$
by (*rel-auto*)

Weakest Precondition and Weakest Liberal Precondition are equivalent for terminating deterministic programs.

lemma *wlp-wp-equiv-total-det*: $\ll Dom(P) = true; ufunctional \ P \rrbracket \Longrightarrow P \text{ wp } b = P \text{ wlp } b$
by (*rel-blast*)

end

23 Dynamic Logic

theory *utp-dynlog*
imports *utp-sequent utp-wp*
begin

23.1 Definitions

named-theorems *dynlog-simp* and *dynlog-intro*

definition $dBox :: 's \text{ hrel} \Rightarrow 's \text{ upred} \Rightarrow 's \text{ upred} \ ([-] \text{ } [0,999] \ 999)$
where $[upred-defs]: dBox \ A \ \Phi = A \ \text{wlp} \ \Phi$

definition $dDia :: 's \text{ hrel} \Rightarrow 's \text{ upred} \Rightarrow 's \text{ upred} \ (\langle - \rangle \text{ } [0,999] \ 999)$
where $[upred-defs]: dDia \ A \ \Phi = A \ \text{wp} \ \Phi$

lemma $dDia\text{-}dBox\text{-}def: \langle A \rangle \Phi = (\neg [A](\neg \Phi))$
by (*simp add: dBox-def dDia-def wp-wlp-conjugate*)

23.2 Box Laws

lemma $dBox\text{-}false \ [dynlog\text{-}simp]: [false]\Phi = true$
by (*rel-auto*)

lemma $dBox\text{-}skip \ [dynlog\text{-}simp]: [I]\Phi = \Phi$
by (*rel-auto*)

lemma $dBox\text{-}assigns \ [dynlog\text{-}simp]: [\langle \sigma \rangle_a]\Phi = (\sigma \dagger \Phi)$
by (*simp add: dBox-def wlp-assigs-r*)

lemma $dBox\text{-}choice \ [dynlog\text{-}simp]: [P \sqcap Q]\Phi = ([P]\Phi \wedge [Q]\Phi)$
by (*rel-auto*)

lemma $dBox\text{-}seq: [P ;; Q]\Phi = [P][Q]\Phi$
by (*simp add: dBox-def wlp-seq-r*)

lemma $dBox\text{-}star\text{-}unfold: [P^*]\Phi = (\Phi \wedge [P][P^*]\Phi)$
by (*metis dBox-choice dBox-seq dBox-skip ustar-unfoldl*)

lemma $dBox\text{-}star\text{-}induct: '(\Phi \wedge [P^*](\Phi \Rightarrow [P]\Phi)) \Rightarrow [P^*]\Phi'$
by (*rel-simp, metis (mono-tags, lifting) mem-Collect-eq rtrancl-induct*)

lemma $dBox\text{-}test: [?[p]]\Phi = (p \Rightarrow \Phi)$
by (*rel-auto*)

23.3 Diamond Laws

lemma $dDia\text{-}false \ [dynlog\text{-}simp]: \langle false \rangle \Phi = false$
by (*simp add: dBox-false dDia-dBox-def*)

lemma $dDia\text{-}skip \ [dynlog\text{-}simp]: \langle I \rangle \Phi = \Phi$
by (*simp add: dBox-skip dDia-dBox-def*)

lemma $dDia\text{-}assigns \ [dynlog\text{-}simp]: \langle \langle \sigma \rangle_a \rangle \Phi = (\sigma \dagger \Phi)$
by (*simp add: dBox-assigs dDia-dBox-def subst-not*)

lemma $dDia\text{-}choice: \langle P \sqcap Q \rangle \Phi = (\langle P \rangle \Phi \vee \langle Q \rangle \Phi)$
by (*simp add: dBox-def dDia-dBox-def wlp-choice*)

lemma $dDia\text{-}seq: \langle P ;; Q \rangle \Phi = \langle P \rangle \langle Q \rangle \Phi$
by (*simp add: dBox-def dDia-dBox-def wlp-seq-r*)

lemma *dDia-test*: $\langle ?[p] \rangle \Phi = (p \wedge \Phi)$
by (*rel-auto*)

23.4 Sequent Laws

lemma *sBoxSeq* [*dynlog-simp*]: $\Gamma \Vdash [P ;; Q]\Phi \equiv \Gamma \Vdash [P][Q]\Phi$
by (*simp add: dBox-def wlp-seq-r*)

lemma *sBoxTest* [*dynlog-intro*]: $\Gamma \Vdash (b \Rightarrow \Psi) \Longrightarrow \Gamma \Vdash [?b]\Psi$
by (*rel-auto*)

lemma *sBoxAssignFwd* [*dynlog-simp*]: $\llbracket \text{vwb-lens } x; x \# v; x \# \Gamma \rrbracket \Longrightarrow (\Gamma \Vdash [x := v]\Phi) = ((\&x =_u v \wedge \Gamma) \Vdash \Phi)$
by (*rel-auto, metis vwb-lens-wb wb-lens.get-put*)

lemma *sBoxIndStar*: $\Vdash [\Phi \Rightarrow [P]\Phi]_u \Longrightarrow \Phi \Vdash [P^*]\Phi$
by (*rel-simp, metis (mono-tags, lifting) mem-Collect-eq rtrancl-induct*)

lemma *hoare-as-dynlog*: $\llbracket p \rrbracket Q \llbracket r \rrbracket_u = (p \Vdash [Q]r)$
by (*rel-auto*)

end

24 State Variable Declaration Parser

theory *utp-state-parser*
imports *utp-rel*
begin

This theory sets up a parser for state blocks, as an alternative way of providing lenses to a predicate. A program with local variables can be represented by a predicate indexed by a tuple of lenses, where each lens represents a variable. These lenses must then be supplied with respect to a suitable state space. Instead of creating a type to represent this alphabet, we can create a product type for the state space, with an entry for each variable. Then each variable becomes a composition of the *fst_L* and *snd_L* lenses to index the correct position in the variable vector. We first creation a vacuous definition that will mark when an indexed predicate denotes a state block.

definition *state-block* :: $('v \Rightarrow 'p) \Rightarrow 'v \Rightarrow 'p$ **where**
 $[upred-defs]: \text{state-block } f \ x = f \ x$

We declare a number of syntax translations to produce lens and product types, to obtain a type for the overall state space, to construct a tuple that denotes the lens vector parameter, to construct the vector itself, and finally to construct the state declaration.

syntax
 $\text{-lensT} :: \text{type} \Rightarrow \text{type} \Rightarrow \text{type} \ (\text{LENSTYPE}'(-, -'))$
 $\text{-pairT} :: \text{type} \Rightarrow \text{type} \Rightarrow \text{type} \ (\text{PAIRTYPE}'(-, -'))$
 $\text{-state-type} :: \text{pttrn} \Rightarrow \text{type}$
 $\text{-state-tuple} :: \text{type} \Rightarrow \text{pttrn} \Rightarrow \text{logic}$
 $\text{-state-lenses} :: \text{pttrn} \Rightarrow \text{logic}$
 $\text{-state-decl} :: \text{pttrn} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\text{LOCAL} \ - \ - \ [0, 10] \ 10)$

translations
 $(\text{type}) \ \text{PAIRTYPE}('a, 'b) \Rightarrow (\text{type}) \ 'a \times 'b$

$(type) \text{ LENSTYPE}('a, 'b) \Rightarrow (type) 'a \Rightarrow 'b$

$-state-type \ (-constrain \ x \ t) \Rightarrow t$

$-state-type \ (CONST \ Pair \ (-constrain \ x \ t) \ vs) \Rightarrow -pairT \ t \ (-state-type \ vs)$

$-state-tuple \ st \ (-constrain \ x \ t) \Rightarrow -constrain \ x \ (-lensT \ t \ st)$

$-state-tuple \ st \ (CONST \ Pair \ (-constrain \ x \ t) \ vs) \Rightarrow$

$CONST \ Product-Type.Pair \ (-constrain \ x \ (-lensT \ t \ st)) \ (-state-tuple \ st \ vs)$

$-state-decl \ vs \ P \Rightarrow$

$CONST \ state-block \ (-abs \ (-state-tuple \ (-state-type \ vs) \ vs) \ P) \ (-state-lenses \ vs)$

$-state-decl \ vs \ P \leq CONST \ state-block \ (-abs \ vs \ P) \ k$

parse-translation \ll

let

open *HOLogic*;

val *lens-comp* = *Const* (@{*const-syntax lens-comp*}, *dummyT*);

val *fst-lens* = *Const* (@{*const-syntax fst-lens*}, *dummyT*);

val *snd-lens* = *Const* (@{*const-syntax snd-lens*}, *dummyT*);

val *id-lens* = *Const* (@{*const-syntax id-lens*}, *dummyT*);

(* *Construct a tuple of lenses for each of the possible locally declared variables* *)

fun

state-lenses *n* *st* =

if (*n* = 1)

then *st*

else *pair-const* *dummyT* *dummyT* \$ (*lens-comp* \$ *fst-lens* \$ *st*) \$ (*state-lenses* (*n* - 1) (*lens-comp*

\$ *snd-lens* \$ *st*));

fun

(* *Add up the number of variable declarations in the tuple* *)

var-decl-num (*Const* (@{*const-syntax Product-Type.Pair*},-) \$ - \$ *vs*) = *var-decl-num* *vs* + 1 |

var-decl-num - = 1;

fun *state-lens* *ctx* [*vs*] = *state-lenses* (*var-decl-num* *vs*) *id-lens* ;

in

[(-*state-lenses*, *state-lens*)]

end

\gg

24.1 Examples

term *LOCAL* (*x::int*, *y::real*, *z::int*) • *x* := (&*x* + &*z*)

lemma *LOCAL* *p* • *II* = *II*

by (*rel-auto*)

end

25 Relational Operational Semantics

theory *utp-rel-opsem*

imports

utp-rel-laws

utp-hoare

begin

This theory uses the laws of relational calculus to create a basic operational semantics. It is based on Chapter 10 of the UTP book [22].

fun *trel* :: 'α *usubst* × 'α *hrel* ⇒ 'α *usubst* × 'α *hrel* ⇒ *bool* (**infix** →_u 85) **where**
 (σ, *P*) →_u (ρ, *Q*) ⇔ (⟨σ⟩_a ;; *P*) ⊆ (⟨ρ⟩_a ;; *Q*)

lemma *trans-trel*:

[[(σ, *P*) →_u (ρ, *Q*); (ρ, *Q*) →_u (φ, *R*)]] ⇒ (σ, *P*) →_u (φ, *R*)
by *auto*

lemma *skip-trel*: (σ, *II*) →_u (σ, *II*)

by *simp*

lemma *assigns-trel*: (σ, ⟨ρ⟩_a) →_u (ρ ∘ σ, *II*)

by (*simp add: assigns-comp*)

lemma *assign-trel*:

(σ, *x* := *v*) →_u (σ(&*x* ↦_s σ † *v*), *II*)
by (*simp add: assigns-comp usubst*)

lemma *seq-trel*:

assumes (σ, *P*) →_u (ρ, *Q*)
shows (σ, *P* ;; *R*) →_u (ρ, *Q* ;; *R*)
by (*metis (no-types, lifting) assms order-refl seqr-assoc seqr-mono trel.simps*)

lemma *seq-skip-trel*:

(σ, *II* ;; *P*) →_u (σ, *P*)
by *simp*

lemma *nondet-left-trel*:

(σ, *P* ⊓ *Q*) →_u (σ, *P*)
by (*metis (no-types, hide-lams) disj-comm disj-upred-def semilattice-sup-class.sup.absorb-iff1 semilattice-sup-class.sup.l*
seqr-or-distr trel.simps)

lemma *nondet-right-trel*:

(σ, *P* ⊓ *Q*) →_u (σ, *Q*)
by (*simp add: seqr-mono*)

lemma *rcond-true-trel*:

assumes σ † *b* = *true*
shows (σ, *P* ◁ *b* ▷_r *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* ▷_r *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 $\sigma \uparrow b = \text{false}$
shows $(\sigma, \text{while } b \text{ do } P \text{ od}) \rightarrow_u (\sigma, II)$
by (*metis assms rcond-false-trel while-unfold*)

Theorem linking Hoare calculus and operational semantics. If we start Q in a state σ_0 satisfying p , and Q reaches final state σ_1 then r holds in this final state.

theorem *hoare-opsem-link*:
 $\{p\} Q \{r\}_u = (\forall \sigma_0 \sigma_1. ' \sigma_0 \uparrow p' \wedge (\sigma_0, Q) \rightarrow_u (\sigma_1, II) \longrightarrow ' \sigma_1 \uparrow r')$
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

26 Symbolic Evaluation of Relational Programs

theory *utp-sym-eval*
imports *utp-rel-opsem*
begin

The following operator applies a variable context Γ as an assignment, and composes it with a relation P for the purposes of evaluation.

definition *utp-sym-eval* :: ' s *usubst* \Rightarrow ' s *hrel* \Rightarrow ' s *hrel* (**infixr** \models 55) **where**
[upred-defs]: utp-sym-eval Γ $P = (\langle \Gamma \rangle_a ;; P)$

named-theorems *symeval*

lemma *seq-symeval* [*symeval*]: $\Gamma \models P ;; Q = (\Gamma \models P) ;; Q$
by (*rel-auto*)

lemma *assigns-symeval* [*symeval*]: $\Gamma \models \langle \sigma \rangle_a = (\sigma \circ \Gamma) \models II$
by (*rel-auto*)

lemma *term-symeval* [*symeval*]: $(\Gamma \models II) ;; P = \Gamma \models P$
by (*rel-auto*)

lemma *if-true-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{true} \rrbracket \Longrightarrow \Gamma \models (P \triangleleft b \triangleright_r Q) = \Gamma \models P$
by (*simp add: utp-sym-eval-def usubst assigns-r-comp*)

lemma *if-false-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{false} \rrbracket \Longrightarrow \Gamma \models (P \triangleleft b \triangleright_r Q) = \Gamma \models Q$
by (*simp add: utp-sym-eval-def usubst assigns-r-comp*)

lemma *while-true-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{true} \rrbracket \Longrightarrow \Gamma \models \text{while } b \text{ do } P \text{ od} = \Gamma \models (P ;; \text{while } b \text{ do } P \text{ od})$
by (*subst while-unfold, simp add: symeval*)

lemma *while-false-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{false} \rrbracket \Longrightarrow \Gamma \models \text{while } b \text{ do } P \text{ od} = \Gamma \models II$
by (*subst while-unfold, simp add: symeval*)

lemma *while-inv-true-symeval* [*symeval*]: $\llbracket \Gamma \vdash b = \text{true} \rrbracket \implies \Gamma \models \text{while } b \text{ invr } S \text{ do } P \text{ od} = \Gamma \models (P \text{ ;; while } b \text{ do } P \text{ od})$

by (*metis while-inv-def while-true-symeval*)

lemma *while-inv-false-symeval* [*symeval*]: $\llbracket \Gamma \vdash b = \text{false} \rrbracket \implies \Gamma \models \text{while } b \text{ invr } S \text{ do } P \text{ od} = \Gamma \models II$

by (*metis while-false-symeval while-inv-def*)

method *sym-eval* = (*simp add: symeval usubst lit-simps[THEN sym]*), (*simp del: One-nat-def add: One-nat-def[THEN sym]*)?

syntax

-terminated :: *logic* \Rightarrow *logic* (*terminated*: - [999] 999)

translations

terminated: $\Gamma == \Gamma \models II$

Below are some theorems linking symbolic evaluation and Hoare logic.

lemma *hoare-symeval-link-1*: $\llbracket b \rrbracket P \llbracket c \rrbracket_u = (\forall s_1 s_2. 's_1 \vdash b' \wedge ((s_1 \models P) \sqsubseteq (s_2 \models II)) \longrightarrow 's_2 \vdash c')$

by (*simp add: utp-sym-eval-def usubst hoare-opsem-link trel.simps*)

lemma *hoare-symeval-link-2*: $\llbracket b \rrbracket P \llbracket c \rrbracket_u \implies 's_1 \vdash b' \wedge ((s_1 \models P) = (s_2 \models II)) \longrightarrow 's_2 \vdash c'$

by (*rel-blast*)

end

27 Strongest Postcondition Calculus

theory *utp-sp*

imports *utp-wlp*

begin

named-theorems *sp*

method *sp-tac* = (*simp add: sp*)

consts

usp :: $'a \Rightarrow 'b \Rightarrow 'c$ (**infix** *sp* 60)

definition *sp-upred* :: $'\alpha \text{ cond} \Rightarrow (''\alpha, ''\beta) \text{ urel} \Rightarrow '\beta \text{ cond}$ **where**

sp-upred *p* *Q* = $\llbracket ([p]_{>} \text{ ;; } Q) :: (''\alpha, ''\beta) \text{ urel} \rrbracket_{>}$

ad hoc-overloading

usp sp-upred

declare *sp-upred-def* [*upred-defs*]

lemma *sp-false* [*sp*]: $p \text{ sp false} = \text{false}$

by (*rel-simp*)

lemma *sp-true* [*sp*]: $q \neq \text{false} \implies q \text{ sp true} = \text{true}$

by (*rel-auto*)

lemma *sp-assigns-r* [*sp*]:

vwv-lens *x* $\implies (p \text{ sp } x := e) = (\exists v \cdot p \llbracket \langle v \rangle / x \rrbracket \wedge \&x =_u e \llbracket \langle v \rangle / x \rrbracket)$

by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*, *metis vwb-lens.put-eq*)

lemma *sp-it-is-post-condition*:
 $\{p\} C \{p \text{ sp } C\}_u$
 by *rel-blast*

lemma *sp-it-is-the-strongest-post*:
 $\langle p \text{ sp } C \Rightarrow Q' \Rightarrow \{p\} C \{Q\}_u$
 by *rel-blast*

lemma *sp-so*:
 $\langle p \text{ sp } C \Rightarrow Q' = \{p\} C \{Q\}_u$
 by *rel-blast*

theorem *sp-hoare-link*:
 $\{p\} Q \{r\}_u \longleftrightarrow (r \sqsubseteq p \text{ sp } Q)$
 by *rel-auto*

lemma *sp-while-r [sp]*:
 assumes $\langle \text{pre} \Rightarrow I' \rangle$ and $\langle \{I \wedge b\} C \{I\}_u \rangle$ and $\langle I' \Rightarrow I' \rangle$
 shows $(\text{pre sp invar } I \text{ while}_\perp b \text{ do } C \text{ od}) = (\neg b \wedge I)$
 unfolding *sp-upred-def*
 oops

theorem *sp-eq-intro*: $\llbracket \bigwedge r. r \text{ sp } P = r \text{ sp } Q \rrbracket \Rightarrow P = Q$
 by (*rel-auto robust*, *fastforce+*)

lemma *wlp-sp-sym*:
 $\langle \text{prog wlp } (\text{true sp prog}) \rangle$
 by *rel-auto*

lemma *it-is-pre-condition*: $\{C \text{ wlp } Q\} C \{Q\}_u$
 by *rel-blast*

lemma *it-is-the-weakest-pre*: $\langle P \Rightarrow C \text{ wlp } Q' = \{P\} C \{Q\}_u$
 by *rel-blast*

lemma *s-pre*: $\langle P \Rightarrow C \text{ wlp } Q' = \{P\} C \{Q\}_u$
 by *rel-blast*

end

28 Concurrent Programming

theory *utp-concurrency*
imports
utp-hoare
utp-rel
utp-tactics
utp-theory
begin

In this theory we describe the UTP scheme for concurrency, *parallel-by-merge*, which provides a general parallel operator parametrised by a “merge predicate” that explains how to merge the after states of the composed predicates. It can thus be applied to many languages and

concurrency schemes, with this theory providing a number of generic laws. The operator is explained in more detail in Chapter 7 of the UTP book [22].

28.1 Variable Renamings

In parallel-by-merge constructions, a merge predicate defines the behaviour following execution of of parallel processes, $P \parallel Q$, as a relation that merges the output of P and Q . In order to achieve this we need to separate the variable values output from P and Q , and in addition the variable values before execution. The following three constructs do these separations. The initial state-space before execution is $'\alpha$, the final state-space after the first parallel process is $'\beta_0$, and the final state-space for the second is $'\beta_1$. These three functions lift variables on these three state-spaces, respectively.

alphabet $('\alpha, '\beta_0, '\beta_1) \text{ mrg} =$
 $\text{mrg-prior} :: '\alpha$
 $\text{mrg-left} :: '\beta_0$
 $\text{mrg-right} :: '\beta_1$

definition $\text{pre-uvar} :: ('a \Rightarrow '\alpha) \Rightarrow ('a \Rightarrow (' \alpha, '\beta_0, '\beta_1) \text{ mrg})$ **where**
 $[\text{upred-defs}]: \text{pre-uvar } x = x ;_L \text{ mrg-prior}$

definition $\text{left-uvar} :: ('a \Rightarrow '\beta_0) \Rightarrow ('a \Rightarrow (' \alpha, '\beta_0, '\beta_1) \text{ mrg})$ **where**
 $[\text{upred-defs}]: \text{left-uvar } x = x ;_L \text{ mrg-left}$

definition $\text{right-uvar} :: ('a \Rightarrow '\beta_1) \Rightarrow ('a \Rightarrow (' \alpha, '\beta_0, '\beta_1) \text{ mrg})$ **where**
 $[\text{upred-defs}]: \text{right-uvar } x = x ;_L \text{ mrg-right}$

We set up syntax for the three variable classes using a subscript $<$, 0 - x , and 1 - x , respectively.

syntax
 $\text{-svarpre} :: \text{svld} \Rightarrow \text{svld} \text{ } (-_< [995] \text{ } 995)$
 $\text{-svarleft} :: \text{svld} \Rightarrow \text{svld} \text{ } (0_- [995] \text{ } 995)$
 $\text{-svarright} :: \text{svld} \Rightarrow \text{svld} \text{ } (1_- [995] \text{ } 995)$

translations
 $\text{-svarpre } x == \text{CONST pre-uvar } x$
 $\text{-svarleft } x == \text{CONST left-uvar } x$
 $\text{-svarright } x == \text{CONST right-uvar } x$
 $\text{-svarpre } \Sigma <= \text{CONST pre-uvar } 1_L$
 $\text{-svarleft } \Sigma <= \text{CONST left-uvar } 1_L$
 $\text{-svarright } \Sigma <= \text{CONST right-uvar } 1_L$

We proved behavedness closure properties about the lenses.

lemma $\text{left-uvar } [\text{simp}]: \text{vwb-lens } x \Longrightarrow \text{vwb-lens } (\text{left-uvar } x)$
by $(\text{simp add: left-uvar-def})$

lemma $\text{right-uvar } [\text{simp}]: \text{vwb-lens } x \Longrightarrow \text{vwb-lens } (\text{right-uvar } x)$
by $(\text{simp add: right-uvar-def})$

lemma $\text{pre-uvar } [\text{simp}]: \text{vwb-lens } x \Longrightarrow \text{vwb-lens } (\text{pre-uvar } x)$
by $(\text{simp add: pre-uvar-def})$

lemma $\text{left-uvar-mwb } [\text{simp}]: \text{mwb-lens } x \Longrightarrow \text{mwb-lens } (\text{left-uvar } x)$
by $(\text{simp add: left-uvar-def})$

lemma *right-ubar-mwb* [simp]: $\text{mwb-lens } x \implies \text{mwb-lens } (\text{right-ubar } x)$
by (simp add: right-ubar-def)

lemma *pre-ubar-mwb* [simp]: $\text{mwb-lens } x \implies \text{mwb-lens } (\text{pre-ubar } x)$
by (simp add: pre-ubar-def)

We prove various independence laws about the variable classes.

lemma *left-ubar-indep-right-ubar* [simp]:
 $\text{left-ubar } x \bowtie \text{right-ubar } y$
by (simp add: left-ubar-def right-ubar-def lens-comp-assoc[THEN sym])

lemma *left-ubar-indep-pre-ubar* [simp]:
 $\text{left-ubar } x \bowtie \text{pre-ubar } y$
by (simp add: left-ubar-def pre-ubar-def)

lemma *left-ubar-indep-left-ubar* [simp]:
 $x \bowtie y \implies \text{left-ubar } x \bowtie \text{left-ubar } y$
by (simp add: left-ubar-def)

lemma *right-ubar-indep-left-ubar* [simp]:
 $\text{right-ubar } x \bowtie \text{left-ubar } y$
by (simp add: lens-indep-sym)

lemma *right-ubar-indep-pre-ubar* [simp]:
 $\text{right-ubar } x \bowtie \text{pre-ubar } y$
by (simp add: right-ubar-def pre-ubar-def)

lemma *right-ubar-indep-right-ubar* [simp]:
 $x \bowtie y \implies \text{right-ubar } x \bowtie \text{right-ubar } y$
by (simp add: right-ubar-def)

lemma *pre-ubar-indep-left-ubar* [simp]:
 $\text{pre-ubar } x \bowtie \text{left-ubar } y$
by (simp add: lens-indep-sym)

lemma *pre-ubar-indep-right-ubar* [simp]:
 $\text{pre-ubar } x \bowtie \text{right-ubar } y$
by (simp add: lens-indep-sym)

lemma *pre-ubar-indep-pre-ubar* [simp]:
 $x \bowtie y \implies \text{pre-ubar } x \bowtie \text{pre-ubar } y$
by (simp add: pre-ubar-def)

28.2 Merge Predicates

A merge predicate is a relation whose input has three parts: the prior variables, the output variables of the left predicate, and the output of the right predicate.

type-synonym $'\alpha \text{ merge} = (('\alpha, '\alpha) \text{ mrg}, '\alpha) \text{ urel}$

skip is the merge predicate which ignores the output of both parallel predicates

definition $\text{skip}_m :: '\alpha \text{ merge}$ **where**
 $[\text{upred-defs}]: \text{skip}_m = (\$ \mathbf{v}' =_u \$ \mathbf{v}_<)$

swap is a predicate that the swaps the left and right indices; it is used to specify commutativity of the parallel operator

definition $swap_m :: (('α, 'β, 'β) \text{ mrg}) \text{ hrel}$ **where**
 $[upred-defs]: swap_m = (0 - \mathbf{v}, 1 - \mathbf{v}) := (\&1 - \mathbf{v}, \&0 - \mathbf{v})$

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 $SymMerge :: 'α \text{ merge} \Rightarrow 'α \text{ merge}$ **where**
 $SymMerge(M) \equiv (swap_m ;; M)$

28.3 Separating Simulations

$U0$ and $U1$ are relations modify the variables of the input state-space such that they become indexed with 0 and 1, respectively.

definition $U0 :: ('β_0, ('α, 'β_0, 'β_1) \text{ mrg}) \text{ urel}$ **where**
 $[upred-defs]: U0 = (\$0 - \mathbf{v}' =_u \$\mathbf{v})$

definition $U1 :: ('β_1, ('α, 'β_0, 'β_1) \text{ mrg}) \text{ urel}$ **where**
 $[upred-defs]: U1 = (\$1 - \mathbf{v}' =_u \$\mathbf{v})$

lemma $U0\text{-swap}: (U0 ;; swap_m) = U1$
by $(rel\text{-auto})$

lemma $U1\text{-swap}: (U1 ;; swap_m) = U0$
by $(rel\text{-auto})$

As shown below, separating simulations can also be expressed using the following two alphabet extrusions

definition $U0\alpha$ **where** $[upred-defs]: U0\alpha = (1_L \times_L \text{ mrg-left})$

definition $U1\alpha$ **where** $[upred-defs]: U1\alpha = (1_L \times_L \text{ mrg-right})$

We then create the following intuitive syntax for separating simulations.

abbreviation $U0\text{-alpha-lift } (\lceil - \rceil_0)$ **where** $\lceil P \rceil_0 \equiv P \oplus_p U0\alpha$

abbreviation $U1\text{-alpha-lift } (\lceil - \rceil_1)$ **where** $\lceil P \rceil_1 \equiv P \oplus_p U1\alpha$

$\lceil P \rceil_0$ is predicate P where all variables are indexed by 0, and $\lceil P \rceil_1$ is where all variables are indexed by 1. We can thus equivalently express separating simulations using alphabet extrusion.

lemma $U0\text{-as-alpha}: (P ;; U0) = \lceil P \rceil_0$
by $(rel\text{-auto})$

lemma $U1\text{-as-alpha}: (P ;; U1) = \lceil P \rceil_1$
by $(rel\text{-auto})$

lemma $U0\alpha\text{-vwb-lens } [simp]: \text{vwb-lens } U0\alpha$
by $(simp \text{ add: } U0\alpha\text{-def id-vwb-lens prod-vwb-lens})$

lemma $U1\alpha\text{-vwb-lens } [simp]: \text{vwb-lens } U1\alpha$
by $(simp \text{ add: } U1\alpha\text{-def id-vwb-lens prod-vwb-lens})$

lemma $U0\alpha\text{-indep-right-uvar } [simp]: \text{vwb-lens } x \Longrightarrow U0\alpha \bowtie \text{out-var } (right\text{-uvar } x)$
by $(force \text{ intro: plus-pres-lens-indep fst-snd-lens-indep lens-indep-left-comp}$
 $\text{ simp add: } U0\alpha\text{-def right-uvar-def out-var-def prod-as-plus})$

lemma *U1 α -indep-left-uvar* [simp]: *vwb-lens* $x \implies U1\alpha \bowtie \text{out-var } (\text{left-uvar } x)$
by (*force intro: plus-pres-lens-indep fst-snd-lens-indep lens-indep-left-comp*
simp add: U1 α -def left-uvar-def out-var-def prod-as-plus)

lemma *U0-alpha-lift-bool-subst* [usubst]:
 $\sigma(\$0 - x' \mapsto_s \text{true}) \uparrow \lceil P \rceil_0 = \sigma \uparrow \lceil P \llbracket \text{true} / \$x' \rrbracket \rceil_0$
 $\sigma(\$0 - x' \mapsto_s \text{false}) \uparrow \lceil P \rceil_0 = \sigma \uparrow \lceil P \llbracket \text{false} / \$x' \rrbracket \rceil_0$
by (*pred-auto+*)

lemma *U1-alpha-lift-bool-subst* [usubst]:
 $\sigma(\$1 - x' \mapsto_s \text{true}) \uparrow \lceil P \rceil_1 = \sigma \uparrow \lceil P \llbracket \text{true} / \$x' \rrbracket \rceil_1$
 $\sigma(\$1 - x' \mapsto_s \text{false}) \uparrow \lceil P \rceil_1 = \sigma \uparrow \lceil P \llbracket \text{false} / \$x' \rrbracket \rceil_1$
by (*pred-auto+*)

lemma *U0-alpha-out-var* [alpha]: $\lceil \$x' \rceil_0 = \$0 - x'$
by (*rel-auto*)

lemma *U1-alpha-out-var* [alpha]: $\lceil \$x' \rceil_1 = \$1 - x'$
by (*rel-auto*)

lemma *U0-skip* [alpha]: $\lceil II \rceil_0 = (\$0 - \mathbf{v}' =_u \$\mathbf{v})$
by (*rel-auto*)

lemma *U1-skip* [alpha]: $\lceil II \rceil_1 = (\$1 - \mathbf{v}' =_u \$\mathbf{v})$
by (*rel-auto*)

lemma *U0-seqr* [alpha]: $\lceil P ;; Q \rceil_0 = P ;; \lceil Q \rceil_0$
by (*rel-auto*)

lemma *U1-seqr* [alpha]: $\lceil P ;; Q \rceil_1 = P ;; \lceil Q \rceil_1$
by (*rel-auto*)

lemma *U0 α -comp-in-var* [alpha]: $(\text{in-var } x) ;_L U0\alpha = \text{in-var } x$
by (*simp add: U0 α -def alpha-in-var in-var-prod-lens pre-uvar-def*)

lemma *U0 α -comp-out-var* [alpha]: $(\text{out-var } x) ;_L U0\alpha = \text{out-var } (\text{left-uvar } x)$
by (*simp add: U0 α -def alpha-out-var id-wb-lens left-uvar-def out-var-prod-lens*)

lemma *U1 α -comp-in-var* [alpha]: $(\text{in-var } x) ;_L U1\alpha = \text{in-var } x$
by (*simp add: U1 α -def alpha-in-var in-var-prod-lens pre-uvar-def*)

lemma *U1 α -comp-out-var* [alpha]: $(\text{out-var } x) ;_L U1\alpha = \text{out-var } (\text{right-uvar } x)$
by (*simp add: U1 α -def alpha-out-var id-wb-lens right-uvar-def out-var-prod-lens*)

28.4 Associative Merges

Associativity of a merge means that if we construct a three way merge from a two way merge and then rotate the three inputs of the merge to the left, then we get exactly the same three way merge back.

We first construct the operator that constructs the three way merge by effectively wiring up the two way merge in an appropriate way.

definition *ThreeWayMerge* :: $'\alpha \text{ merge} \Rightarrow (('\alpha, '\alpha, (' \alpha, '\alpha, '\alpha) \text{ mrg}) \text{ mrg}, '\alpha) \text{ urel } (\mathbf{M3}'(-))$ **where**
[upred-defs]: ThreeWayMerge $M = ((\$0 - \mathbf{v}' =_u \$0 - \mathbf{v} \wedge \$1 - \mathbf{v}' =_u \$1 - \mathbf{v} \wedge \$\mathbf{v}_{<} =_u \$\mathbf{v}_{<}) ;; M ;;$

$U0 \wedge \$1-\mathbf{v}' =_u \$1-1-\mathbf{v} \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}_{<}) ;; M$

The next definition rotates the inputs to a three way merge to the left one place.

abbreviation $rotate_m$ **where** $rotate_m \equiv (0-\mathbf{v}, 1-0-\mathbf{v}, 1-1-\mathbf{v}) := (\&1-0-\mathbf{v}, \&1-1-\mathbf{v}, \&0-\mathbf{v})$

Finally, a merge is associative if rotating the inputs does not effect the output.

definition $AssocMerge :: 'a \text{ merge} \Rightarrow bool$ **where**
 $[upred-defs]: AssocMerge\ M = (rotate_m ;; \mathbf{M}\beta(M) = \mathbf{M}\beta(M))$

28.5 Parallel Operators

We implement the following useful abbreviation for separating of two parallel processes and copying of the before variables, all to act as input to the merge predicate.

abbreviation par_sep (**infixr** \parallel_s 85) **where**
 $P \parallel_s Q \equiv (P ;; U0) \wedge (Q ;; U1) \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}$

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
 $par_by_merge :: ('a, 'b) \text{ urel} \Rightarrow (('a, 'b, 'c) \text{ mrg}, 'd) \text{ urel} \Rightarrow ('a, 'c) \text{ urel} \Rightarrow ('a, 'd) \text{ urel}$
 $(- \parallel - [85, 0, 86] 85)$
where $[upred-defs]: P \parallel_M Q = (P \parallel_s Q ;; M)$

lemma $par_by_merge_alt_def: P \parallel_M Q = (\lceil P \rceil_0 \wedge \lceil Q \rceil_1 \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}) ;; M$
by ($simp\ add: par_by_merge_def\ U0-as-alpha\ U1-as-alpha$)

lemma $shEx-pbm-left: ((\exists x \cdot P\ x) \parallel_M Q) = (\exists x \cdot (P\ x \parallel_M Q))$
by ($rel-auto$)

lemma $shEx-pbm-right: (P \parallel_M (\exists x \cdot Q\ x)) = (\exists x \cdot (P \parallel_M Q\ x))$
by ($rel-auto$)

28.6 Unrestriction Laws

lemma $unrest-in-par-by-merge [unrest]:$
 $\llbracket \$x \# P; \$x_{<} \# M; \$x \# Q \rrbracket \Longrightarrow \$x \# P \parallel_M Q$
by ($rel-auto, fastforce+$)

lemma $unrest-out-par-by-merge [unrest]:$
 $\llbracket \$x' \# M \rrbracket \Longrightarrow \$x' \# P \parallel_M Q$
by ($rel-auto$)

28.7 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 coercsions. We need quite a number of variants to support this which are below.

lemma $U0-seq-subst: (P ;; U0) \llbracket \llbracket v \rrbracket / \$0-x' \rrbracket = (P \llbracket \llbracket v \rrbracket / \$x' \rrbracket ;; U0)$
by ($rel-auto$)

lemma *U1-seq-subst*: $(P ;; U1) \llbracket \llbracket v \rrbracket / \$1 - x' \rrbracket = (P \llbracket \llbracket v \rrbracket / \$x' \rrbracket ;; U1)$
by (*rel-auto*)

lemma *lit-pbm-subst* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s \llbracket v \rrbracket) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket \llbracket v \rrbracket / \$x \rrbracket) \parallel_{M \llbracket \llbracket v \rrbracket / \$x < \rrbracket} (Q \llbracket \llbracket v \rrbracket / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s \llbracket v \rrbracket) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M \llbracket \llbracket v \rrbracket / \$x' \rrbracket} Q)$
by (*rel-auto*)⁺

lemma *bool-pbm-subst* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s \text{false}) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket \text{false} / \$x \rrbracket) \parallel_{M \llbracket \text{false} / \$x < \rrbracket} (Q \llbracket \text{false} / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s \text{true}) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket \text{true} / \$x \rrbracket) \parallel_{M \llbracket \text{true} / \$x < \rrbracket} (Q \llbracket \text{true} / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s \text{false}) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M \llbracket \text{false} / \$x' \rrbracket} Q)$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s \text{true}) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M \llbracket \text{true} / \$x' \rrbracket} Q)$
by (*rel-auto*)⁺

lemma *zero-one-pbm-subst* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s 0) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket 0 / \$x \rrbracket) \parallel_{M \llbracket 0 / \$x < \rrbracket} (Q \llbracket 0 / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s 1) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket 1 / \$x \rrbracket) \parallel_{M \llbracket 1 / \$x < \rrbracket} (Q \llbracket 1 / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s 0) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M \llbracket 0 / \$x' \rrbracket} Q)$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s 1) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M \llbracket 1 / \$x' \rrbracket} Q)$
by (*rel-auto*)⁺

lemma *numeral-pbm-subst* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s \text{numeral } n) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket \text{numeral } n / \$x \rrbracket) \parallel_{M \llbracket \text{numeral } n / \$x < \rrbracket} (Q \llbracket \text{numeral } n / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s \text{numeral } n) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M \llbracket \text{numeral } n / \$x' \rrbracket} Q)$
by (*rel-auto*)⁺

28.8 Parallel-by-merge laws

lemma *par-by-merge-false* [*simp*]:

$$P \parallel_{\text{false}} Q = \text{false}$$

by (*rel-auto*)

lemma *par-by-merge-left-false* [*simp*]:

$$\text{false} \parallel_M Q = \text{false}$$

by (*rel-auto*)

lemma *par-by-merge-right-false* [*simp*]:

$$P \parallel_M \text{false} = \text{false}$$

by (*rel-auto*)

lemma *par-by-merge-seq-add*: $(P \parallel_M Q) ;; R = (P \parallel_M ;; R Q)$

by (*simp add: par-by-merge-def seqr-assoc*)

A skip parallel-by-merge yields a skip whenever the parallel predicates are both feasible.

lemma *par-by-merge-skip*:
assumes $P \;; \text{true} = \text{true } Q \;; \text{true} = \text{true}$
shows $P \parallel_{\text{skip}_m} Q = \text{II}$
using *assms* **by** (*rel-auto*)

lemma *skip-merge-swap*: $\text{swap}_m \;; \text{skip}_m = \text{skip}_m$
by (*rel-auto*)

lemma *par-sep-swap*: $P \parallel_s Q \;; \text{swap}_m = Q \parallel_s P$
by (*rel-auto*)

Parallel-by-merge commutes when the merge predicate is unchanged by swap

lemma *par-by-merge-commute-swap*:
shows $P \parallel_M Q = Q \parallel_{\text{swap}_m} \;; M P$
proof –
have $Q \parallel_{\text{swap}_m} \;; M P = (((Q \;; U0) \wedge (P \;; U1) \wedge \mathbf{v}_{<} =_u \mathbf{v}) \;; \text{swap}_m) \;; M$
by (*simp add: par-by-merge-def seqr-assoc*)
also have $\dots = (((Q \;; U0 \;; \text{swap}_m) \wedge (P \;; U1 \;; \text{swap}_m) \wedge \mathbf{v}_{<} =_u \mathbf{v}) \;; M)$
by (*rel-auto*)
also have $\dots = (((Q \;; U1) \wedge (P \;; U0) \wedge \mathbf{v}_{<} =_u \mathbf{v}) \;; M)$
by (*simp add: U0-swap U1-swap*)
also have $\dots = P \parallel_M Q$
by (*simp add: par-by-merge-def utp-pred-laws.inf.left-commute*)
finally show *?thesis* ..
qed

theorem *par-by-merge-commute*:
assumes M is *SymMerge*
shows $P \parallel_M Q = Q \parallel_M P$
by (*metis Healthy-if assms par-by-merge-commute-swap*)

lemma *par-by-merge-mono-1*:
assumes $P_1 \sqsubseteq P_2$
shows $P_1 \parallel_M Q \sqsubseteq P_2 \parallel_M Q$
using *assms* **by** (*rel-auto*)

lemma *par-by-merge-mono-2*:
assumes $Q_1 \sqsubseteq Q_2$
shows $(P \parallel_M Q_1) \sqsubseteq (P \parallel_M Q_2)$
using *assms* **by** (*rel-blast*)

lemma *par-by-merge-mono*:
assumes $P_1 \sqsubseteq P_2 \; Q_1 \sqsubseteq Q_2$
shows $P_1 \parallel_M Q_1 \sqsubseteq P_2 \parallel_M Q_2$
by (*meson assms dual-order.trans par-by-merge-mono-1 par-by-merge-mono-2*)

theorem *par-by-merge-assoc*:
assumes M is *SymMerge AssocMerge* M
shows $(P \parallel_M Q) \parallel_M R = P \parallel_M (Q \parallel_M R)$

proof –
have $(P \parallel_M Q) \parallel_M R = ((P \;; U0) \wedge (Q \;; U0 \;; U1) \wedge (R \;; U1 \;; U1) \wedge \mathbf{v}_{<} =_u \mathbf{v}) \;; \mathbf{M3}(M)$
by (*rel-blast*)
also have $\dots = ((P \;; U0) \wedge (Q \;; U0 \;; U1) \wedge (R \;; U1 \;; U1) \wedge \mathbf{v}_{<} =_u \mathbf{v}) \;; \text{rotate}_m \;; \mathbf{M3}(M)$
using *AssocMerge-def assms(2)* **by** *force*
also have $\dots = ((Q \;; U0) \wedge (R \;; U0 \;; U1) \wedge (P \;; U1 \;; U1) \wedge \mathbf{v}_{<} =_u \mathbf{v}) \;; \mathbf{M3}(M)$

by (*rel-blast*)
 also have ... = $(Q \parallel_M R) \parallel_M P$
 by (*rel-blast*)
 also have ... = $P \parallel_M (Q \parallel_M R)$
 by (*simp add: assms(1) par-by-merge-commute*)
 finally show ?thesis .
 qed

theorem *par-by-merge-choice-left*:
 $(P \sqcap Q) \parallel_M R = (P \parallel_M R) \sqcap (Q \parallel_M R)$
 by (*rel-auto*)

theorem *par-by-merge-choice-right*:
 $P \parallel_M (Q \sqcap R) = (P \parallel_M Q) \sqcap (P \parallel_M R)$
 by (*rel-auto*)

theorem *par-by-merge-or-left*:
 $(P \vee Q) \parallel_M R = (P \parallel_M R \vee Q \parallel_M R)$
 by (*rel-auto*)

theorem *par-by-merge-or-right*:
 $P \parallel_M (Q \vee R) = (P \parallel_M Q \vee P \parallel_M R)$
 by (*rel-auto*)

theorem *par-by-merge-USUP-mem-left*:
 $(\bigsqcap_{i \in I} i \cdot P(i)) \parallel_M Q = (\bigsqcap_{i \in I} i \cdot P(i) \parallel_M Q)$
 by (*rel-auto*)

theorem *par-by-merge-USUP-ind-left*:
 $(\bigsqcap i \cdot P(i)) \parallel_M Q = (\bigsqcap i \cdot P(i) \parallel_M Q)$
 by (*rel-auto*)

theorem *par-by-merge-USUP-mem-right*:
 $P \parallel_M (\bigsqcap_{i \in I} i \cdot Q(i)) = (\bigsqcap_{i \in I} i \cdot P \parallel_M Q(i))$
 by (*rel-auto*)

theorem *par-by-merge-USUP-ind-right*:
 $P \parallel_M (\bigsqcap i \cdot Q(i)) = (\bigsqcap i \cdot P \parallel_M Q(i))$
 by (*rel-auto*)

28.9 Example: Simple State-Space Division

The following merge predicate divides the state space using a pair of independent lenses.

definition *StateMerge* :: $('a \Longrightarrow 'a) \Rightarrow ('b \Longrightarrow 'a) \Rightarrow 'a \text{ merge } (M[-|-]_\sigma)$ **where**
[upred-defs]: $M[a|b]_\sigma = (\$ \mathbf{v}' =_u (\$ \mathbf{v}_< \oplus \$ 0 - \mathbf{v} \text{ on } \&a) \oplus \$ 1 - \mathbf{v} \text{ on } \&b)$

lemma *swap-StateMerge*: $a \bowtie b \Longrightarrow (\text{swap}_m ;; M[a|b]_\sigma) = M[b|a]_\sigma$
 by (*rel-auto, simp-all add: lens-indep-comm*)

abbreviation *StateParallel* :: $'a \text{ hrel} \Rightarrow ('a \Longrightarrow 'a) \Rightarrow ('b \Longrightarrow 'a) \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel } (-|-|_\sigma - [85,0,0,86] \ 86)$
where $P \mid a|b|_\sigma Q \equiv P \parallel_{M[a|b]_\sigma} Q$

lemma *StateParallel-commute*: $a \bowtie b \Longrightarrow P \mid a|b|_\sigma Q = Q \mid b|a|_\sigma P$
 by (*metis par-by-merge-commute-swap swap-StateMerge*)

lemma *StateParallel-form*:

$P \mid a \mid b \mid_\sigma Q = (\exists (st_0, st_1) \cdot P[\ll st_0 \gg / \$\mathbf{v}'] \wedge Q[\ll st_1 \gg / \$\mathbf{v}'] \wedge \$\mathbf{v}' =_u (\$ \mathbf{v} \oplus \ll st_0 \gg \text{ on } \&a) \oplus \ll st_1 \gg \text{ on } \&b)$
by (*rel-auto*)

lemma *StateParallel-form'*:

assumes *vwb-lens a vwb-lens b a \bowtie b*
shows $P \mid a \mid b \mid_\sigma Q = \{ \&a, \&b \} : [(P \mid_v \{ \$\mathbf{v}, \$a' \}) \wedge (Q \mid_v \{ \$\mathbf{v}, \$b' \})]$
using *assms*
apply (*simp add: StateParallel-form, rel-auto*)
apply (*metis vwb-lens-wb wb-lens-axioms-def wb-lens-def*)
apply (*metis vwb-lens-wb wb-lens.get-put*)
apply (*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 \bowtie b*
shows $\{ \&a, \&b \} : [P \mid a \mid b \mid_\sigma Q] = P \mid a \mid b \mid_\sigma Q$
using *assms*
apply (*simp add: StateParallel-form, rel-auto*)
using *lens-indep-comm* **apply** *fastforce+*
done

Parallel Hoare logic rule. This employs something similar to separating conjunction in the postcondition, but we explicitly require that the two conjuncts only refer to variables on the left and right of the parallel composition explicitly.

theorem *StateParallel-hoare* [*hoare*]:

assumes $\{ c \} P \{ d_1 \}_u \{ c \} Q \{ d_2 \}_u \ a \bowtie b \ a \nmid d_1 \ b \nmid d_2$
shows $\{ c \} P \mid a \mid b \mid_\sigma Q \{ d_1 \wedge d_2 \}_u$

proof –

— Parallelise the specification

from *assms(4,5)*

have $1: (\lceil c \rceil < \Rightarrow \lceil d_1 \wedge d_2 \rceil >) \sqsubseteq (\lceil c \rceil < \Rightarrow \lceil d_1 \rceil >) \mid a \mid b \mid_\sigma (\lceil c \rceil < \Rightarrow \lceil d_2 \rceil >) \text{ (is ?lhs } \sqsubseteq \text{ ?rhs)}$
by (*simp add: StateParallel-form, rel-auto, metis assms(3) lens-indep-comm*)

— Prove Hoare rule by monotonicity of parallelism

have $2: ?rhs \sqsubseteq P \mid a \mid b \mid_\sigma Q$

proof (*rule par-by-merge-mono*)

show $(\lceil c \rceil < \Rightarrow \lceil d_1 \rceil >) \sqsubseteq P$

using *assms(1) hoare-r-def* **by** *auto*

show $(\lceil c \rceil < \Rightarrow \lceil d_2 \rceil >) \sqsubseteq Q$

using *assms(2) hoare-r-def* **by** *auto*

qed

show *?thesis*

unfolding *hoare-r-def* **using** *1 2 order-trans* **by** *auto*

qed

Specialised version of the above law where an invariant expression referring to variables outside the frame is preserved.

theorem *StateParallel-frame-hoare* [*hoare*]:

assumes *vwb-lens a vwb-lens b a \bowtie b a \nmid d₁ b \nmid d₂ a $\#$ c₁ b $\#$ c₁ {c₁ \wedge c₂} P {d₁ }_u {c₁ \wedge c₂} Q {d₂ }_u*
shows $\{ c_1 \wedge c_2 \} P \mid a \mid b \mid_\sigma Q \{ c_1 \wedge d_1 \wedge d_2 \}_u$

proof –


```

have  $\llbracket c_1 \wedge c_2 \rrbracket \{ \&a, \&b \} : [P \mid a \mid b]_{\sigma} Q \llbracket c_1 \wedge d_1 \wedge d_2 \rrbracket_u$ 
  by (auto intro! : frame-hoare-r' StateParallel-hoare simp add : assms unrest plus-vwb-lens)
thus ?thesis
  by (simp add : StateParallel-frame assms)
qed

end

```

29 Meta-theory for the Standard Core

```

theory utp
imports
  utp-var
  utp-expr
  utp-expr-insts
  utp-expr-funcs
  utp-unrest
  utp-usedby
  utp-subst
  utp-meta-subst
  utp-alphabet
  utp-lift
  utp-pred
  utp-pred-laws
  utp-recursion
  utp-dynlog
  utp-rel
  utp-rel-laws
  utp-sequent
  utp-state-parser
  utp-sym-eval
  utp-tactics
  utp-hoare
  utp-wlp
  utp-wp
  utp-sp
  utp-theory
  utp-concurrency
  utp-rel-opsem
begin end

```

30 Overloaded Expression Constructs

```

theory utp-expr-ovld
  imports utp
begin

```

30.1 Overloadable Constants

For convenience, we often want to utilise the same expression syntax for multiple constructs. This can be achieved using ad-hoc overloading. We create a number of polymorphic constants and then overload their definitions using appropriate implementations. In order for this to work, each collection must have its own unique type. Thus we do not use the HOL map type directly, but rather our own partial function type, for example.

consts

— Empty elements, for example empty set, nil list, 0...
uempty :: 'f
— Function application, map application, list application...
uapply :: 'f \Rightarrow 'k \Rightarrow 'v
— Overriding
uovrd :: 'f \Rightarrow 'f \Rightarrow 'f
— Function update, map update, list update...
uupd :: 'f \Rightarrow 'k \Rightarrow 'v \Rightarrow 'f
— Domain of maps, lists...
udom :: 'f \Rightarrow 'a set
— Range of maps, lists...
uran :: 'f \Rightarrow 'b set
— Domain restriction
udomres :: 'a set \Rightarrow 'f \Rightarrow 'f
— Range restriction
uranres :: 'f \Rightarrow 'b set \Rightarrow 'f
— Collection cardinality
ucard :: 'f \Rightarrow nat
— Collection summation
usums :: 'f \Rightarrow 'a
— Construct a collection from a list of entries
uentries :: 'k set \Rightarrow ('k \Rightarrow 'v) \Rightarrow 'f

We need a function corresponding to function application in order to overload.

definition *fun-apply* :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)

where *fun-apply* f x = f x

declare *fun-apply-def* [simp]

definition *ffun-entries* :: 'k set \Rightarrow ('k \Rightarrow 'v) \Rightarrow ('k, 'v) *ffun* **where**

ffun-entries d f = *graph-ffun* {(k, f k) | k. k \in d}

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

adhoc-overloading

uempty 0 **and**
uapply *rel-apply* **and** *uapply* *fun-apply* **and** *uapply* *nth* **and** *uapply* *pfun-app* **and** *uapply* *ffun-app* **and**
uovrd *rel-override* **and** *uovrd* *plus*
uupd *rel-update* **and** *uupd* *pfun-upd* **and** *uupd* *ffun-upd* **and** *uupd* *list-augment* **and**
udom *Domain* **and** *udom* *pdom* **and** *udom* *fdom* **and** *udom* *seq-dom* **and**
uran *Range* **and** *uran* *pran* **and** *uran* *fran* **and** *uran* *set* **and**
udomres *rel-domres* **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*

30.2 Syntax Translations

syntax

-*uundef* :: *logic* (\perp_u)
-*umap-empty* :: *logic* (\llbracket_u)
-*uapply* :: ('a \Rightarrow 'b, 'α) *uexpr* \Rightarrow *utuple-args* \Rightarrow ('b, 'α) *uexpr* (-'(-)'_a [999,0] 999)
-*uovrd* :: *logic* \Rightarrow *logic* \Rightarrow *logic* (**infixl** \oplus 65)
-*umaplet* :: [*logic*, *logic*] \Rightarrow *umaplet* (- / \mapsto / -)

```

      :: umaplet => umaplets      (-)
-UMaplets  :: [umaplet, umaplets] => umaplets (-,/ -)
-UMapUpd   :: [logic, umaplets] => logic (-/'(-')u [900,0] 900)
-UMap      :: umaplets => logic ((1[-]u))
-ucard     :: logic => logic (#u'(-'))
-udom      :: logic => logic (domu'(-'))
-uran      :: logic => logic (ranu'(-'))
-usum      :: logic => logic (sumu'(-'))
-udom-res  :: logic => logic => logic (infixl  $\triangleleft_u$  85)
-uran-res  :: logic => logic => logic (infixl  $\triangleright_u$  85)
-uentries  :: logic => logic => logic (entru'(-,-'))

```

translations

— Pretty printing for adhoc-overloaded constructs

```

f(x)a    <= CONST uapply f x
f ⊕ g    <= CONST uovrd f g
domu(f) <= CONST udom f
ranu(f) <= CONST uran f
A  $\triangleleft_u$  f <= CONST udomres A f
f  $\triangleright_u$  A <= CONST uranres f A
#u(f) <= CONST ucard f
f(k ↦ v)u <= CONST uupd f k v
0 <= CONST uempty — We have to do this so we don't see uempty. Is there a better way of printing?

```

— Overloaded construct translations

```

f(x,y,z,u)a == CONST bop CONST uapply f (x,y,z,u)u
f(x,y,z)a == CONST bop CONST uapply f (x,y,z)u
f(x,y)a == CONST bop CONST uapply f (x,y)u
f(x)a == CONST bop CONST uapply f x
f ⊕ g == CONST bop CONST uovrd f g
#u(xs) == CONST uop CONST ucard xs
sumu(A) == CONST uop CONST usums A
domu(f) == CONST uop CONST udom f
ranu(f) == CONST uop CONST uran f
[]u == <<CONST uempty>>
⊥u == <<CONST undefined>>
A  $\triangleleft_u$  f == CONST bop (CONST udomres) A f
f  $\triangleright_u$  A == CONST bop (CONST uranres) f A
entru(d,f) == CONST bop CONST uentries d <<f>>
-UMapUpd m (-UMaplets xy ms) == -UMapUpd (-UMapUpd m xy) ms
-UMapUpd m (-umaplet x y) == CONST trop CONST uupd m x y
-UMap ms == -UMapUpd []u ms
-UMap (-UMaplets ms1 ms2) <= -UMapUpd (-UMap ms1) ms2
-UMaplets ms1 (-UMaplets ms2 ms3) <= -UMaplets (-UMaplets ms1 ms2) ms3

```

30.3 Simplifications

lemma *ufun-apply-lit [simp]*:

```

<<f>>(x)a = <<f(x)>>
by (transfer, simp)

```

lemma *lit-plus-appl [lit-norm]*: <<(+)>>(x)_a(y)_a = x + y **by** (simp add: uexpr-defs, transfer, simp)

lemma *lit-minus-appl [lit-norm]*: <<(-)>>(x)_a(y)_a = x - y **by** (simp add: uexpr-defs, transfer, simp)

lemma *lit-mult-appl [lit-norm]*: <<times>>(x)_a(y)_a = x * y **by** (simp add: uexpr-defs, transfer, simp)

lemma *lit-divide-apply [lit-norm]*: <<(/)>>(x)_a(y)_a = x / y **by** (simp add: uexpr-defs, transfer, simp)

lemma *pfun-entries-apply* [simp]:
 $(entr_u(d, f) :: ((k, v) pfun, \alpha) uexp)(i)_a = ((\ll f \gg(i)_a) \triangleleft i \in_u d \triangleright \perp_u)$
by (*pred-auto*)

lemma *udom-uupdate-pfun* [simp]:
fixes $m :: ((k, v) pfun, \alpha) uexp$
shows $dom_u(m(k \mapsto v)_u) = \{k\}_u \cup_u dom_u(m)$
by (*rel-auto*)

lemma *uapply-uupdate-pfun* [simp]:
fixes $m :: ((k, v) pfun, \alpha) uexp$
shows $(m(k \mapsto v)_u)(i)_a = v \triangleleft i =_u k \triangleright m(i)_a$
by (*rel-auto*)

30.4 Indexed Assignment

syntax

— Indexed assignment
-assignment-upd $:: svid \Rightarrow uexp \Rightarrow uexp \Rightarrow logic (([-] := / -) [63, 0, 0] 62)$

translations

— Indexed assignment uses the overloaded collection update function *uupd*.
-assignment-upd $x\ k\ v \Rightarrow x := \&x(k \mapsto v)_u$

end

31 Meta-theory for the Standard Core with Overloaded Constructs

theory *utp-full*

imports *utp utp-expr-ovld*

begin end

32 UTP Easy Expression Parser

theory *utp-easy-parser*

imports *utp-full*

begin

32.1 Replacing the Expression Grammar

The following theory provides an easy to use expression parser that is primarily targetted towards expressing programs. Unlike the built-in UTP expression syntax, this uses a closed grammar separate to the HOL *logic* nonterminal, that gives more freedom in what can be expressed. In particular, identifiers are interpreted as UTP variables rather than HOL variables and functions do not require subscripts and other strange decorations.

The first step is to remove the from the UTP parse the following grammar rule that uses arbitrary HOL logic to represent expressions. Instead, we will populate the *uexp* grammar manually.

purge-syntax

-uexp-l $:: logic \Rightarrow uexp\ (- [64] 64)$

32.2 Expression Operators

syntax

```

-ue-quote :: uexp ⇒ logic ('('')e)
-ue-tuple :: uexprs ⇒ uexp ('('')
-ue-lit   :: logic ⇒ uexp («-» )
-ue-var   :: svid ⇒ uexp (-)
-ue-ivar  :: svid ⇒ uexp ($- [990] 990)
-ue-ovar  :: svid ⇒ uexp ($-' [990] 990)
-ue-nexp  :: id ⇒ uexp (@- [990] 990)
-ue-nexp-p:: logic ⇒ uexp (@'('') [990] 990)
-ue-eq    :: uexp ⇒ uexp ⇒ uexp (infix = 150)
-ue-neq   :: uexp ⇒ uexp ⇒ uexp (infix ≠ 150)
-ue-uop   :: id ⇒ uexp ⇒ uexp ('('') [999,0] 999)
-ue-bop   :: id ⇒ uexp ⇒ uexp ⇒ uexp ('(-, -') [999,0,0] 999)
-ue-trop  :: id ⇒ uexp ⇒ uexp ⇒ uexp ⇒ uexp ('(-, -, -') [999,0,0,0] 999)
-ue-apply :: uexp ⇒ uexp ⇒ uexp (-[-] [999] 999)
-ue-aext  :: uexp ⇒ salpha ⇒ uexp (infixr ⊕p 195)
-ue-ifthenelse :: uexp ⇒ uexp ⇒ uexp ⇒ uexp ((β- < - >/ -) [52,0,53] 52)
-ue-ovrd  :: uexp ⇒ uexp ⇒ uexp (infixl ⊕ 65)

```

translations

```

-ue-quote e => e
-ue-tuple (-uexprs x (-uexprs y z)) => -ue-tuple (-uexprs x (-ue-tuple (-uexprs y z)))
-ue-tuple (-uexprs x y) => CONST bop CONST Pair x y
-ue-tuple x => x
-ue-lit x   => CONST lit x
-ue-nexp P  => P
-ue-nexp-p P => P
-ue-var x   => CONST utp-expr.var (CONST pr-var x)
-ue-ivar x  => CONST utp-expr.var (CONST in-var x)
-ue-ovar x  => CONST utp-expr.var (CONST out-var x)
-ue-eq x y  => x =u y
-ue-neq x y => x ≠u y
-ue-uop f x => CONST uop f x
-ue-bop f x y => CONST bop f x y
-ue-trop f x y => CONST trop f x y
-ue-apply f x => f(x)a
-ue-ifthenelse P b Q => CONST cond P b Q
-ue-aext P a => CONST aext P a
-ue-ovrd x y => -uovrd x y

```

32.3 Predicate Operators

syntax

```

-ue-true :: uexp (true)
-ue-false :: uexp (false)
-ue-not  :: uexp ⇒ uexp (¬ - [40] 40)
-ue-conj :: uexp ⇒ uexp ⇒ uexp (infixr ∧ 135)
-ue-disj :: uexp ⇒ uexp ⇒ uexp (infixr ∨ 130)
-ue-impl :: uexp ⇒ uexp ⇒ uexp (infixr ⇒ 125)
-ue-iff  :: uexp ⇒ uexp ⇒ uexp (infixr ⇔ 125)
-ue-mem  :: uexp ⇒ uexp ⇒ uexp ((-/ ∈ -) [151, 151] 150)
-ue-nmem :: uexp ⇒ uexp ⇒ uexp ((-/ ∉ -) [151, 151] 150)
-ue-shEx :: pttrn ⇒ uexp ⇒ uexp (∃ - · - [0, 10] 10)
-ue-shAll:: pttrn ⇒ uexp ⇒ uexp (∀ - · - [0, 10] 10)

```

$-ue-shBEx :: pttrn \Rightarrow uexp \Rightarrow uexp \Rightarrow uexp \quad (\exists - \in - \cdot - [0, 0, 10] 10)$
 $-ue-shBAll :: pttrn \Rightarrow uexp \Rightarrow uexp \Rightarrow uexp \quad (\forall - \in - \cdot - [0, 0, 10] 10)$

translations

$-ue-true \Rightarrow CONST \text{ true-upred}$
 $-ue-false \Rightarrow CONST \text{ false-upred}$
 $-ue-not p \Rightarrow CONST \text{ not-upred } p$
 $-ue-conj p q \Rightarrow p \wedge_p q$
 $-ue-disj p q \Rightarrow p \vee_p q$
 $-ue-impl p q \Rightarrow p \Rightarrow q$
 $-ue-iff p q \Rightarrow p \Leftrightarrow q$
 $-ue-mem x A \Rightarrow x \in_u A$
 $-ue-nmem x A \Rightarrow x \notin_u A$
 $-ue-shEx x P \Rightarrow -ushEx x P$
 $-ue-shAll x P \Rightarrow -ushAll x P$
 $-ue-shBEx x A P \Rightarrow -ushBEx x A P$
 $-ue-shBAll x A P \Rightarrow -ushBAll x A P$

32.4 Arithmetic Operators

syntax

$-ue-num :: num-const \Rightarrow uexp \text{ (-)}$
 $-ue-size :: uexp \Rightarrow uexp \text{ (\#- [999] 999)}$
 $-ue-eq :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infix = 150)}$
 $-ue-le :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infix } \leq \text{ 150)}$
 $-ue-lt :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infix } < \text{ 150)}$
 $-ue-ge :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infix } \geq \text{ 150)}$
 $-ue-gt :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infix } > \text{ 150)}$
 $-ue-zero :: uexp (0)$
 $-ue-one :: uexp (1)$
 $-ue-plus :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infixl + 165)}$
 $-ue-uminus :: uexp \Rightarrow uexp \text{ (- - [181] 180)}$
 $-ue-minus :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infixl - 165)}$
 $-ue-times :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infixl * 170)}$
 $-ue-div :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infixl div 170)}$

translations

$-ue-num x \Rightarrow -Numeral x$
 $-ue-size e \Rightarrow \#_u(e)$
 $-ue-le x y \Rightarrow x \leq_u y$
 $-ue-lt x y \Rightarrow x <_u y$
 $-ue-ge x y \Rightarrow x \geq_u y$
 $-ue-gt x y \Rightarrow x >_u y$
 $-ue-zero \Rightarrow 0$
 $-ue-one \Rightarrow 1$
 $-ue-plus x y \Rightarrow x + y$
 $-ue-uminus x \Rightarrow - x$
 $-ue-minus x y \Rightarrow x - y$
 $-ue-times x y \Rightarrow x * y$
 $-ue-div x y \Rightarrow CONST \text{ divide } x y$

32.5 Sets

syntax

$-ue-empset :: uexp (\{\})$
 $-ue-setprod :: uexp \Rightarrow uexp \Rightarrow uexp \text{ (infixr } \times \text{ 180)}$

$-ue-setenum \quad :: uexprs \Rightarrow uexp \ (\{-\})$
 $-ue-subseteq \quad :: uexp \Rightarrow uexp \Rightarrow uexp \ (\mathbf{infix} \subseteq 150)$
 $-ue-subset \quad :: uexp \Rightarrow uexp \Rightarrow uexp \ (\mathbf{infix} \subset 150)$
 $-ue-atLeastAtMost \quad :: uexp \Rightarrow uexp \Rightarrow uexp \ ((1\{-..\}))$
 $-ue-atLeastLessThan \quad :: uexp \Rightarrow uexp \Rightarrow uexp \ ((1\{-..<-\}))$

translations

$-ue-empset \Rightarrow \{\}_u$
 $-ue-setprod \ e \ f \Rightarrow \text{CONST } bop \ (\text{CONST } Product\text{-}Type.Times) \ e \ f$
 $-ue-setenum \ (-uexprs \ x \ xs) \Rightarrow insert_u \ x \ (-ue-setenum \ xs)$
 $-ue-setenum \ x \Rightarrow insert_u \ x \ \{\}_u$
 $-ue-subseteq \ x \ y \Rightarrow x \subseteq_u y$
 $-ue-subset \ x \ y \Rightarrow x \subset_u y$
 $-ue-atLeastAtMost \ m \ n \Rightarrow \{m..n\}_u$
 $-ue-atLeastLessThan \ m \ n \Rightarrow \{m..<n\}_u$

32.6 Lists

syntax

$-ue-nil \quad :: uexp \ (\[])$
 $-ue-listenum \quad :: uexprs \Rightarrow uexp \ ([-])$
 $-ue-append \quad :: uexp \Rightarrow uexp \Rightarrow uexp \ (\mathbf{infixr} \ @ \ 65)$

translations

$-ue-nil \Rightarrow \langle \rangle$
 $-ue-listenum \ (-uexprs \ x \ xs) \Rightarrow x \ \#_u \ (-ue-listenum \ xs)$
 $-ue-listenum \ x \Rightarrow \langle x \rangle$
 $-ue-append \ xs \ ys \Rightarrow xs \ ^{}_u \ ys$

32.7 Imperative Program Syntax

syntax

$-ue-if-then \quad :: uexp \Rightarrow logic \Rightarrow logic \Rightarrow logic \ (if \ - \ then \ - \ else \ - \ fi)$
 $-ue-hoare \quad :: uexp \Rightarrow logic \Rightarrow uexp \Rightarrow logic \ (\{\{-\}\} \ / \ - \ / \ \{\{-\}\})$
 $-ue-wp \quad :: logic \Rightarrow uexp \Rightarrow uexp \ (\mathbf{infix} \ wp \ 60)$

translations

$-ue-if-then \ b \ P \ Q \Rightarrow P \triangleleft b \triangleright_r Q$
 $-ue-hoare \ b \ P \ c \Rightarrow \{b\}P\{c\}_u$
 $-ue-wp \ P \ b \Rightarrow P \ wp \ b$

end

References

- [1] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2015.
- [2] A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In *RAMiCS 2012*, volume 7560 of *LNCS*. Springer, September 2012.
- [3] R.-J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

- [4] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In *Proc. 5th Intl. Conf. on Mathematical Knowledge Management (MKM)*, volume 4108 of *LNCS*, pages 31–43. Springer, 2006.
- [5] C. Ballarin et al. The Isabelle/HOL Algebra Library. *Isabelle/HOL*, October 2017. <https://isabelle.in.tum.de/library/HOL/HOL-Algebra/document.pdf>.
- [6] A. Cavalcanti and J. Woodcock. A tutorial introduction to designs in unifying theories of programming. In *Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*, pages 40–66. Springer, 2004.
- [7] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.
- [8] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [9] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.
- [10] A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus: a process specification and verification environment. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, 2012.
- [11] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.
- [12] S. Foster, J. Baxter, A. Cavalcanti, A. Miyazawa, and J. Woodcock. Automating verification of state machines with reactive designs and Isabelle/UTP. In *Proc. 15th. Intl. Conf. on Formal Aspects of Component Software*, volume 11222 of *LNCS*. Springer, October 2018.
- [13] S. Foster, K. Ye, A. Cavalcanti, and J. Woodcock. Calculational verification of reactive programs with reactive relations and Kleene algebra. In *Proc. 17th Intl. Conf. on Relational and Algebraic Methods in Computer Science (RAMICS)*, volume 11194 of *LNCS*. Springer, October 2018.
- [14] S. Foster and F. Zeyda. Optics. *Archive of Formal Proofs*, May 2017. <http://isa-afp.org/entries/Optics.html>, Formal proof development.
- [15] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *UTP*, *LNCS* 8963, pages 21–41. Springer, 2014.
- [16] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC)*, volume 9965 of *LNCS*. Springer, 2016.
- [17] D. Harel. Dynamic logic. In *Handbook of Philosophical Logic*, volume 165 of *SYLI*, pages 497–604. Springer, 1984.
- [18] E. C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14:133–158, 1990.
- [19] E. C. R. Hehner. *A Practical Theory of Programming*. Springer, 1993.

- [20] E. C. R. Hehner and A. J. Malton. Termination conventions and comparative semantics. *Acta Informatica*, 25, 1988.
- [21] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
- [22] T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
- [23] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *CPP*, volume 8307 of *LNCs*, pages 131–146. Springer, 2013.
- [24] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997.
- [25] C. Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.
- [26] M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In *UTP 2006*, volume 4010 of *LNCs*, pages 123–140. Springer, 2007.
- [27] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2006. YCST-2006-02.
- [28] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *UTP*, LNCs 10134. Springer, 2016.