

Isabelle/UTP: Mechanised Theory Engineering for the UTP

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

April 27, 2018

Abstract

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

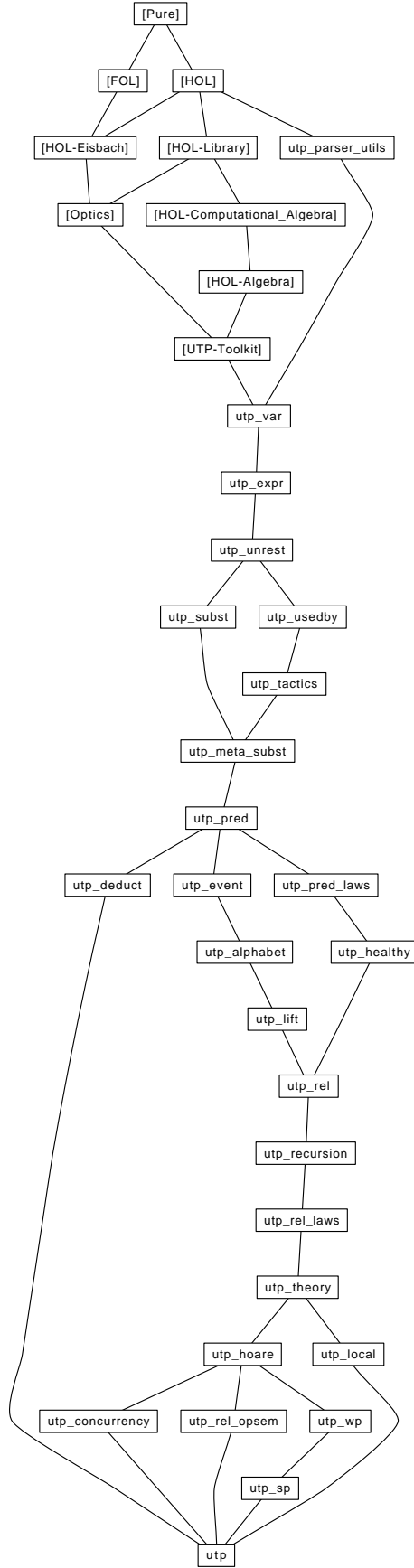
Contents

1	Introduction	6
2	UTP Variables	7
2.1	Initial syntax setup	7
2.2	Variable foundations	7
2.3	Variable lens properties	8
2.4	Lens simplifications	10
2.5	Syntax translations	10
3	UTP Expressions	13
3.1	Expression type	13
3.2	Core expression constructs	13
3.3	Type class instantiations	15
3.4	Overloaded expression constructors	18
3.5	Syntax translations	19
3.6	Lifting set collectors	23
3.7	Lifting limits	23
3.8	Evaluation laws for expressions	24
3.9	Misc laws	25
3.10	Literalise tactics	25
4	Unrestriction	27
4.1	Definitions and Core Syntax	27
4.2	Unrestriction laws	28
5	Used-by	30

6	Substitution	33
6.1	Substitution definitions	33
6.2	Syntax translations	34
6.3	Substitution Application Laws	35
6.4	Substitution laws	38
6.5	Ordering substitutions	40
6.6	Unrestriction laws	40
6.7	Conditional Substitution Laws	41
6.8	Parallel Substitution Laws	41
7	UTP Tactics	43
7.1	Theorem Attributes	43
7.2	Generic Methods	43
7.3	Transfer Tactics	44
7.3.1	Robust Transfer	44
7.3.2	Faster Transfer	44
7.4	Interpretation	45
7.5	User Tactics	45
8	Meta-level Substitution	47
9	Alphabetised Predicates	48
9.1	Predicate type and syntax	48
9.2	Predicate operators	49
9.3	Unrestriction Laws	54
9.4	Used-by laws	56
9.5	Substitution Laws	56
10	UTP Events	59
10.1	Events	59
10.2	Channels	59
10.2.1	Operators	59
11	Alphabet Manipulation	60
11.1	Preliminaries	60
11.2	Alphabet Extrusion	60
11.3	Expression Alphabet Restriction	63
11.4	Predicate Alphabet Restriction	64
11.5	Alphabet Lens Laws	64
11.6	Substitution Alphabet Extension	65
11.7	Substitution Alphabet Restriction	65
12	Lifting Expressions	66
12.1	Lifting definitions	66
12.2	Lifting Laws	66
12.3	Substitution Laws	67
12.4	Unrestriction laws	67

13 Predicate Calculus Laws	67
13.1 Propositional Logic	67
13.2 Lattice laws	71
13.3 Equality laws	76
13.4 HOL Variable Quantifiers	77
13.5 Case Splitting	78
13.6 UTP Quantifiers	79
13.7 Variable Restriction	80
13.8 Conditional laws	81
13.9 Additional Expression Laws	82
13.10Refinement By Observation	83
13.11Cylindric Algebra	84
14 Healthiness Conditions	84
14.1 Main Definitions	84
14.2 Properties of Healthiness Conditions	86
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	95
15.5 Introduction laws	96
15.6 Unrestriction Laws	96
15.7 Substitution laws	97
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	103
16.3 Noetherian Induction Instantiation	105
17 UTP Deduction Tactic	107
18 Relational Calculus Laws	109
18.1 Conditional Laws	109
18.2 Precondition and Postcondition Laws	109
18.3 Sequential Composition Laws	110
18.4 Iterated Sequential Composition Laws	113
18.5 Quantale Laws	113
18.6 Skip Laws	114
18.7 Assignment Laws	114
18.8 Converse Laws	116
18.9 Assertion and Assumption Laws	117
18.10Frame and Antiframe Laws	117
18.11While Loop Laws	119
18.12Algebraic Properties	120
18.12.1 Kleene Star	122
18.13Kleene Plus	122

18.14	Omega	123
18.15	Relation Algebra Laws	123
18.16	Kleene Algebra Laws	123
18.17	Omega Algebra Laws	125
18.18	Refinement Laws	125
18.19	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	129
19.4	Theory of relations	137
19.5	Theory links	138
20	Relational Hoare calculus	139
20.1	Hoare Triple Definitions and Tactics	139
20.2	Basic Laws	139
20.3	Assignment Laws	140
20.4	Sequence Laws	140
20.5	Conditional Laws	140
20.6	Recursion Laws	141
20.7	Iteration Rules	141
20.8	Frame Rules	143
21	Weakest Precondition Calculus	144
22	Strong Postcondition Calculus	145
23	Concurrent Programming	146
23.1	Variable Renamings	147
23.2	Merge Predicates	148
23.3	Separating Simulations	149
23.4	Associative Merges	151
23.5	Parallel Operators	151
23.6	Unrestriction Laws	151
23.7	Substitution laws	152
23.8	Parallel-by-merge laws	152
23.9	Example: Simple State-Space Division	154
24	Relational Operational Semantics	156
25	Local Variables	157
25.1	Preliminaries	158
25.2	State Primitives	158
25.3	Relational State Spaces	160
26	Meta-theory for the Standard Core	161



1 Introduction

This document contains the description of our mechanisation of Hoare and He’s *Unifying Theories of Programming* [14, 7] (UTP) in Isabelle/HOL. UTP uses the “programs-as-predicate” approach to encode denotational semantics and facilitate reasoning about programs. It uses the alphabetised relational calculus, which combines predicate calculus and relation algebra, to denote programs as relations between initial variables (x) and their subsequent values (x'). Isabelle/UTP¹ [13, 20, 12] semantically embeds this relational calculus into Isabelle/HOL, which enables application of the latter’s proof facilities to program verification. For an introduction to UTP, we recommend two tutorials [6, 7], and also the UTP book itself [14].

The Isabelle/UTP core mechanises most of definitions and theorems from chapters 1, 2, 4, and 7, and some material contained in chapters 5 and 10. This essentially amounts to alphabetised predicate calculus, its core laws, the UTP theory infrastructure, and also parallel-by-merge [14, chapter 5], which adds concurrency primitives. The Isabelle/UTP core does not contain the theory of designs [6] and CSP [7], which are both represented in their own theory developments.

A large part of the mechanisation, however, is foundations that enable these core UTP theories. In particular, Isabelle/UTP builds on our implementation of lenses [13, 11], which gives a formal semantics to state spaces and variables. This, in turn, builds on a previous version of Isabelle/UTP [8, 9], which provided a shallow embedding of UTP by using Isabelle record types to represent alphabets. We follow this approach and, additionally, use the lens laws [10, 13] to characterise well-behaved variables. We also add meta-logical infrastructure for dealing with free variables and substitution. All this, we believe, adds an additional layer rigour to the UTP.

The alphabets-as-types approach does impose a number of limitations on Isabelle/UTP. For example, alphabets can only be extended when an injection into a larger state-space type can be exhibited. It is therefore not possible to arbitrarily augment an alphabet with additional variables, but new types must be created to do this. The pay-off is that the Isabelle/HOL type checker can be directly applied to relational constructions, which makes proof much more automated and efficient. Moreover, our use of lenses mitigates the limitations by providing meta-logical style operators, such as equality on variables, and alphabet membership [13]. For a detailed discussion of semantic embedding approaches, please see [20].

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

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

1. Formalisation of variables and state-spaces using lenses [13];
2. an expression model, together with lifted operators from HOL;
3. the meta-logical operators of unrestriction, used-by, substitution, alphabet extrusion, and alphabet restriction;
4. the alphabetised predicate calculus and associated algebraic laws;
5. the alphabetised relational calculus and associated algebraic laws;

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

6. an implementation of local variables using stacks;
7. proof tactics for the above based on interpretation [15];
8. a formalisation of UTP theories using locales [4] and building on HOL-Algebra [5];
9. Hoare logic;
10. weakest precondition and strongest postcondition calculi;
11. concurrent programming with parallel-by-merge;
12. relational operational semantics.

2 UTP Variables

```
theory utp-var
  imports
    ../toolkit/utp-toolkit
    utp-parser-utils
begin
```

In this first UTP theory we set up variables, which are built on lenses [10, 13]. A large part of this theory is setting up the parser for UTP variable syntax.

2.1 Initial syntax setup

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

```
purge-notation
  Order.le (infixl  $\sqsubseteq_1$  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-indep-left-ext [simp]
declare lens-indep-right-ext [simp]
```

2.2 Variable foundations

This theory describes the foundational structure of UTP variables, upon which the rest of our model rests. We start by defining alphabets, which following [8, 9] in this shallow model are simply represented as types $'\alpha$, 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 $in-var :: ('a \Rightarrow 'α) \Rightarrow ('a \Rightarrow 'α \times 'β) \text{ where}$
 $[lens-defs]: in-var\ x = x ;_L fst_L$

definition $out-var :: ('a \Rightarrow 'β) \Rightarrow ('a \Rightarrow 'α \times 'β) \text{ where}$
 $[lens-defs]: out-var\ x = x ;_L snd_L$

Variables can also be used to effectively define sets of variables. Here we define the 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 $(input)\ univ-alpha :: ('α \Rightarrow 'α)\ (\Sigma) \text{ where}$
 $univ-alpha \equiv 1_L$

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

definition $pr-var :: ('a \Rightarrow 'β) \Rightarrow ('a \Rightarrow 'β) \text{ where}$
 $[lens-defs]: pr-var\ x = x$

2.3 Variable lens properties

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

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

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

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

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

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

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

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

lemma $out-var-semi-uvar\ [simp]:$
 $mwb-lens\ x \Rightarrow mwb-lens\ (out-var\ x)$

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

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

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

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

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

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

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

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

lemma *pr-var-lens-comp-1 [simp]:*
 $pr\text{-}var\ x ;_L y = pr\text{-}var\ (x ;_L y)$

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

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

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

Similar properties follow for sublens

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

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

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

2.4 Lens simplifications

We also define some lookup abstraction simplifications.

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

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

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

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

2.5 Syntax translations

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

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

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

syntax — Identifiers

-svid :: *id* \Rightarrow *svid* ($- [999] 999$)
-svid-unit :: *svid* \Rightarrow *svids* ($-$)
-svid-list :: *svid* \Rightarrow *svids* \Rightarrow *svids* ($-, / -$)
-svid-alpha :: *svid* (\mathbf{v})
-svid-dot :: *svid* \Rightarrow *svid* \Rightarrow *svid* ($:- [998, 999] 998$)

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

syntax — Decorations

```
-spvar      :: svid  $\Rightarrow$  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 ( $\Sigma$ )
-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

```
-ualpha-set :: svars  $\Rightarrow$  logic ({-} $\alpha$ )
-svar      :: svar  $\Rightarrow$  logic ('(-') $v$ )
```

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

Next we need to construct the syntax translations rules. First we need a few polymorphic constants.

consts

```
svar :: 'v  $\Rightarrow$  'e
ivar :: 'v  $\Rightarrow$  'e
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

-svid $x \rightarrow x$

-svid-alpha $\Rightarrow \Sigma$

-svid-dot $x\ y \rightarrow y ;_L x$

— Decorations

-spvar $\Sigma \leftarrow \text{CONST svar } \text{CONST id-lens}$

-sinvar $\Sigma \leftarrow \text{CONST ivar } 1_L$

-soutvar $\Sigma \leftarrow \text{CONST ovar } 1_L$

-spvar (*-svid-dot* $x\ y$) $\leftarrow \text{CONST svar } (\text{CONST lens-comp } y\ x)$

-sinvar (*-svid-dot* $x\ y$) $\leftarrow \text{CONST ivar } (\text{CONST lens-comp } y\ x)$

-soutvar (*-svid-dot* $x\ y$) $\leftarrow \text{CONST ovar } (\text{CONST lens-comp } y\ x)$

-svid-dot (*-svid-dot* $x\ y$) $z \leftarrow \text{-svid-dot } (\text{CONST lens-comp } y\ x)\ z$

-spvar $x \Rightarrow \text{CONST svar } x$

-sinvar $x \Rightarrow \text{CONST ivar } x$

-soutvar $x \Rightarrow \text{CONST ovar } x$

— Alphabets

-salphaparen $a \rightarrow a$

-salphaid $x \rightarrow x$

-salphacomp $x\ y \rightarrow x +_L y$

-salphaprod $a\ b \Rightarrow a \times_L b$

-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)) $\leftarrow \text{-salphamk } (x +_L y)$

$x \leftarrow \text{-salphamk } x$

-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 $:: \text{type} \Rightarrow \text{type} \Rightarrow \text{type}$

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 [15], which allows us to reuse much of the existing library of HOL functions.

typedef $('t, 'a) \text{ uexpr} = \text{UNIV} :: ('a \Rightarrow 't) \text{ set} ..$

setup-lifting *type-definition-uexpr*

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

lemma *uexpr-eq-iff*:

$e = f \iff (\forall b. \llbracket e \rrbracket_e b = \llbracket f \rrbracket_e b)$
using *Rep-uexpr-inject*[of $e f$, *THEN sym*] **by** (*auto*)

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

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

3.2 Core expression constructs

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

lift-definition $\text{var} :: ('t \Rightarrow 'a) \Rightarrow ('t, 'a) \text{ uexpr}$ **is** *lens-get* .

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

lift-definition $\text{lit} :: 't \Rightarrow ('t, 'a) \text{ uexpr}$ **is** $\lambda v b. v$.

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

lift-definition $\text{uop} :: ('a \Rightarrow 'b) \Rightarrow ('a, 'a) \text{ uexpr} \Rightarrow ('b, 'a) \text{ uexpr}$
is $\lambda f e b. f (e b)$.

lift-definition $\text{bop} :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'a) \text{ uexpr} \Rightarrow ('b, 'a) \text{ uexpr} \Rightarrow ('c, 'a) \text{ uexpr}$

is $\lambda f u v b. f (u b) (v b) .$

lift-definition *trop* ::

$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow ('a, ' \alpha) \text{ uepr} \Rightarrow ('b, ' \alpha) \text{ uepr} \Rightarrow ('c, ' \alpha) \text{ uepr} \Rightarrow ('d, ' \alpha) \text{ uepr}$

is $\lambda f u v w b. f (u b) (v b) (w b) .$

lift-definition *qtop* ::

$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e) \Rightarrow$

$('a, ' \alpha) \text{ uepr} \Rightarrow ('b, ' \alpha) \text{ uepr} \Rightarrow ('c, ' \alpha) \text{ uepr} \Rightarrow ('d, ' \alpha) \text{ uepr} \Rightarrow$

$('e, ' \alpha) \text{ uepr}$

is $\lambda f u v w x b. f (u b) (v b) (w b) (x b) .$

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

lift-definition *ulambda* :: $('a \Rightarrow ('b, ' \alpha) \text{ uepr}) \Rightarrow ('a \Rightarrow 'b, ' \alpha) \text{ uepr}$

is $\lambda f A x. f x A .$

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

definition *uIf* :: $\text{bool} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**

uIf = *If*

abbreviation *cond* ::

$('a, ' \alpha) \text{ uepr} \Rightarrow (\text{bool}, ' \alpha) \text{ uepr} \Rightarrow ('a, ' \alpha) \text{ uepr} \Rightarrow ('a, ' \alpha) \text{ uepr}$

$((\beta - \triangleleft - \triangleright / -) [52, 0, 53] 52)$

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

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

definition *eq-upred* :: $('a, ' \alpha) \text{ uepr} \Rightarrow ('a, ' \alpha) \text{ uepr} \Rightarrow (\text{bool}, ' \alpha) \text{ uepr}$

where *eq-upred* $x y = \text{bop } HOL.eq x y$

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

consts

ulit :: $'t \Rightarrow 'e (\ll - \gg)$

ueq :: $'a \Rightarrow 'a \Rightarrow 'b (\text{infixl } =_u 50)$

adhoc-overloading

ulit lit and

ueq eq-upred

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

We also set up syntax for UTP variable expressions.

syntax

-uuvar :: $svar \Rightarrow \text{logic } (-)$

translations

-uuvar $x == \text{CONST } var x$

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:  $0 = \text{lit } 0$ 
instance ..
end
```

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

```
end
```

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

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:  $- u = \text{uop } \text{uminus } u$ 
instance ..
end
```

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

```
instantiation uexpr :: (times, type) times
begin
  definition times-uexpr-def:  $u * v = \text{bop } (op *) u v$ 
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
    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 inverse-uexpr u = uop inverse u
instance ..
end

```

```

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

```

```

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

```

```

instantiation uexpr :: (abs, type) abs
begin
  definition abs-uexpr-def: 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 :: (semigroup-add, type) semigroup-add
  by (intro-classes) (simp add: plus-uexpr-def zero-uexpr-def, transfer, simp add: add.assoc)+

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```


instance *uexpr* :: (*ab-group-add*, *type*) *ab-group-add*

by (*intro-classes*)

(*simp add: plus-uexpr-def uminus-uexpr-def minus-uexpr-def zero-uexpr-def*, *transfer*, *simp*)+

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

by (*intro-classes*) (*simp add: plus-uexpr-def times-uexpr-def*, *transfer*, *simp add: fun-eq-iff add.commute*

semiring-class.distrib-right semiring-class.distrib-left)+

instance *uexpr* :: (*ring-1*, *type*) *ring-1*

by (*intro-classes*) (*simp add: plus-uexpr-def uminus-uexpr-def minus-uexpr-def times-uexpr-def zero-uexpr-def*

one-uexpr-def, *transfer*, *simp add: fun-eq-iff*)+

We can also define the order relation on expressions. Now, unlike the previous group and ring constructs, the order relations $op \leq$ and $op \leq$ 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 *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* \implies *y* \leq *z* \implies *x* \leq *z*

by (*transfer*, *blast intro:order.trans*)

show *x* \leq *y* \implies *y* \leq *x* \implies *x* = *y*

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

qed

We also lift the properties from certain ordered groups.

instance *uexpr* :: (*ordered-ab-group-add*, *type*) *ordered-ab-group-add*

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

instance *uexpr* :: (*ordered-ab-group-add-abs*, *type*) *ordered-ab-group-add-abs*

apply (*intro-classes*)

apply (*simp add: abs-uexpr-def zero-uexpr-def plus-uexpr-def 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 following instantiation sets up numerals. This will allow us to have Isabelle number representations (i.e. 3,7,42,198 etc.) to UTP expressions directly.

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

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

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

```
lemma numeral-uepr-rep-eq:  $\llbracket \text{numeral } x \rrbracket_e b = \text{numeral } x$ 
apply (induct x)
apply (simp add: lit.rep-eq one-uepr-def)
apply (simp add: bop.rep-eq numeral-Bit0 plus-uepr-def)
apply (simp add: bop.rep-eq lit.rep-eq numeral-code(3) one-uepr-def plus-uepr-def)
done
```

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

The next theorem lifts powers.

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

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

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

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

3.4 Overloaded expression constructors

For convenience, we often want to utilise the same expression syntax for multiple constructs. This can be achieved using ad-hoc overloading. We create a number of polymorphic constants and then overload their definitions using appropriate implementations. In order for this to work, each collection must have its own unique type. Thus we do not use the HOL map type directly, but rather our own partial function type, for example.

```
consts
— Empty elements, for example empty set, nil list, 0...
uepr      :: 'f
— Function application, map application, list application...
uapply    :: 'f  $\Rightarrow$  'k  $\Rightarrow$  'v
— Function update, map update, list update...
uupd      :: 'f  $\Rightarrow$  'k  $\Rightarrow$  'v  $\Rightarrow$  'f
— Domain of maps, lists...
uom       :: 'f  $\Rightarrow$  'a set
— Range of maps, lists...
uran      :: 'f  $\Rightarrow$  'b set
— Domain restriction
uomres    :: '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 :: $'k \text{ set} \Rightarrow ('k \Rightarrow 'v) \Rightarrow 'f$

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* :: $'k \text{ set} \Rightarrow ('k \Rightarrow 'v) \Rightarrow ('k, 'v) \text{ ffun}$ **where**
ffun-entries *d* *f* = *graph-ffun* $\{(k, f\ k) \mid k. k \in d\}$

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

ad hoc-overloading

uempty 0 **and**
uapply *fun-apply* **and** *uapply* *nth* **and** *uapply* *pfun-app* **and**
uapply *ffun-app* **and**
upd *pfun-upd* **and** *upd* *ffun-upd* **and** *upd* *list-augment* **and**
udom *Domain* **and** *udom* *pdom* **and** *udom* *fdom* **and** *udom* *seq-dom* **and**
udom *Range* **and** *uran* *pran* **and** *uran* *fran* **and** *uran* *set* **and**
udomres *pdom-res* **and** *udomres* *fdom-res* **and**
uranres *pran-res* **and** *udomres* *fran-res* **and**
ucard *card* **and** *ucard* *pcard* **and** *ucard* *length* **and**
usums *list-sum* **and** *usums* *Sum* **and** *usums* *pfun-sum* **and**
uentries *pfun-entries* **and** *uentries* *ffun-entries*

3.5 Syntax translations

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

abbreviation (*input*) *ulens-override* *x* *f* *g* \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* :: $'a \text{ itself} \Rightarrow 'a \text{ set}$ **where**
set-of *t* = *UNIV*

translations

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

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

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

syntax — Core expression constructs

-*ucoerce* :: $\text{logic} \Rightarrow \text{type} \Rightarrow \text{logic}$ (**infix** :_u 50)
-*ulambda* :: $\text{pttrn} \Rightarrow \text{logic} \Rightarrow \text{logic}$ (λ - · - [0, 10] 10)
-*ulens-ovrd* :: $\text{logic} \Rightarrow \text{logic} \Rightarrow \text{salph} \Rightarrow \text{logic}$ (- \oplus - on - [85, 0, 86] 86)
-*ulens-get* :: $\text{logic} \Rightarrow \text{svar} \Rightarrow \text{logic}$ (-:- [900,901] 901)

translations

λ *x* · *p* == *CONST* *ulambda* (λ *x*. *p*)
x :_u *'a* == *x* :: (*'a*, -) *uexpr*

$-ulens-ovrd\ f\ g\ a \Rightarrow CONST\ bop\ (CONST\ ulens-override\ a)\ f\ g$
 $-ulens-ovrd\ f\ g\ a \Leftarrow CONST\ bop\ (\lambda x\ y.\ CONST\ lens-override\ x1\ y1\ a)\ f\ g$
 $-ulens-get\ x\ y == CONST\ uop\ (CONST\ lens-get\ y)\ x$

syntax — Tuples

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

translations

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

syntax — Polymorphic constructs

$-uundef \quad :: logic\ (\perp_u)$
 $-umap-empty \quad :: logic\ ([\]_u)$
 $-uapply \quad :: ('a \Rightarrow 'b, 'α)\ uexpr \Rightarrow utuple-args \Rightarrow ('b, 'α)\ uexpr\ (-'(-)_a\ [999,0]\ 999)$
 $-umaplet \quad :: [logic, logic] \Rightarrow umaplet\ (-\ /\mapsto\ -)$
 $\quad :: umaplet \Rightarrow umaplets\ \quad (-)$
 $-UMaplets \quad :: [umaplet, umaplets] \Rightarrow umaplets\ (-, / -)$
 $-UMapUpd \quad :: [logic, umaplets] \Rightarrow logic\ (-/'(-)_u\ [900,0]\ 900)$
 $-UMap \quad :: umaplets \Rightarrow logic\ ((1[-]_u))$
 $-ucard \quad :: logic \Rightarrow logic\ (\#_u'(-))$
 $-uless \quad :: logic \Rightarrow logic \Rightarrow logic\ (\mathbf{infix}\ <_u\ 50)$
 $-uleq \quad :: logic \Rightarrow logic \Rightarrow logic\ (\mathbf{infix}\ \leq_u\ 50)$
 $-ugreat \quad :: logic \Rightarrow logic \Rightarrow logic\ (\mathbf{infix}\ >_u\ 50)$
 $-ugeq \quad :: logic \Rightarrow logic \Rightarrow logic\ (\mathbf{infix}\ \geq_u\ 50)$
 $-uceil \quad :: logic \Rightarrow logic\ (\lceil - \rceil_u)$
 $-ufloor \quad :: logic \Rightarrow logic\ (\lfloor - \rfloor_u)$
 $-udom \quad :: logic \Rightarrow logic\ (dom_u'(-))$
 $-uran \quad :: logic \Rightarrow logic\ (ran_u'(-))$
 $-usum \quad :: logic \Rightarrow logic\ (sum_u'(-))$
 $-udom-res \quad :: logic \Rightarrow logic \Rightarrow logic\ (\mathbf{infixl}\ \triangleleft_u\ 85)$
 $-uran-res \quad :: logic \Rightarrow logic \Rightarrow logic\ (\mathbf{infixl}\ \triangleright_u\ 85)$
 $-umin \quad :: logic \Rightarrow logic \Rightarrow logic\ (min_u'(-, -))$
 $-umax \quad :: logic \Rightarrow logic \Rightarrow logic\ (max_u'(-, -))$
 $-ugcd \quad :: logic \Rightarrow logic \Rightarrow logic\ (gcd_u'(-, -))$
 $-uentries \quad :: logic \Rightarrow logic \Rightarrow logic\ (entr_u'(-, -))$

translations

— Pretty printing for adhoc-overloaded constructs

$f(x)_a \quad \Leftarrow CONST\ uapply\ f\ x$
 $dom_u(f) \Leftarrow CONST\ udom\ f$
 $ran_u(f) \Leftarrow CONST\ uran\ f$
 $A \triangleleft_u f \Leftarrow CONST\ udomres\ A\ f$
 $f \triangleright_u A \Leftarrow CONST\ uranres\ f\ A$
 $\#_u(f) \Leftarrow CONST\ ucard\ f$
 $f(k \mapsto v)_u \Leftarrow CONST\ wupd\ f\ k\ v$

— Overloaded construct translations

$f(x,y,z,u)_a == \text{CONST bop } \text{CONST uapply } f \ (x,y,z,u)_u$
 $f(x,y,z)_a == \text{CONST bop } \text{CONST uapply } f \ (x,y,z)_u$
 $f(x,y)_a == \text{CONST bop } \text{CONST uapply } f \ (x,y)_u$
 $f(x)_a == \text{CONST bop } \text{CONST uapply } f \ x$
 $\#_u(xs) == \text{CONST uop } \text{CONST ucard } xs$
 $\text{sum}_u(A) == \text{CONST uop } \text{CONST usums } A$
 $\text{dom}_u(f) == \text{CONST uop } \text{CONST udom } f$
 $\text{ran}_u(f) == \text{CONST uop } \text{CONST uran } f$
 $\Box_u == \ll \text{CONST uempty} \gg$
 $\perp_u == \ll \text{CONST undefined} \gg$
 $A \triangleleft_u f == \text{CONST bop } (\text{CONST udomres}) \ A \ f$
 $f \triangleright_u A == \text{CONST bop } (\text{CONST uranres}) \ f \ A$
 $\text{entr}_u(d,f) == \text{CONST bop } \text{CONST uentries } d \ \ll f \gg$
 $\text{-UMapUpd } m \ (\text{-UMaplets } xy \ ms) == \text{-UMapUpd } (\text{-UMapUpd } m \ xy) \ ms$
 $\text{-UMapUpd } m \ (\text{-umaplet } x \ y) == \text{CONST trop } \text{CONST uupd } m \ x \ y$
 $\text{-UMap } ms == \text{-UMapUpd } \Box_u \ ms$
 $\text{-UMap } (\text{-UMaplets } ms1 \ ms2) \leq \text{-UMapUpd } (\text{-UMap } ms1) \ ms2$
 $\text{-UMaplets } ms1 \ (\text{-UMaplets } ms2 \ ms3) \leq \text{-UMaplets } (\text{-UMaplets } ms1 \ ms2) \ ms3$

— Type-class polymorphic constructs

$x <_u y == \text{CONST bop } (op <) \ x \ y$
 $x \leq_u y == \text{CONST bop } (op \leq) \ x \ y$
 $x >_u y \Rightarrow y <_u x$
 $x \geq_u y \Rightarrow y \leq_u x$
 $\text{min}_u(x, y) == \text{CONST bop } (\text{CONST min}) \ x \ y$
 $\text{max}_u(x, y) == \text{CONST bop } (\text{CONST max}) \ x \ y$
 $\text{gcd}_u(x, y) == \text{CONST bop } (\text{CONST gcd}) \ x \ y$
 $\lceil x \rceil_u == \text{CONST uop } \text{CONST ceiling } x$
 $\lfloor x \rfloor_u == \text{CONST uop } \text{CONST floor } x$

syntax — Lists / Sequences

$\text{-unil} :: ('a \text{ list}, 'a) \text{ uexpr } (\langle \rangle)$
 $\text{-ulist} :: \text{args} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr } (\langle \langle - \rangle \rangle)$
 $\text{-uappend} :: ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr } (\text{infixr } \hat{_}_u \ 80)$
 $\text{-ulast} :: ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a, 'a) \text{ uexpr } (\text{last}_u'(-))$
 $\text{-ufront} :: ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr } (\text{front}_u'(-))$
 $\text{-uhead} :: ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a, 'a) \text{ uexpr } (\text{head}_u'(-))$
 $\text{-utail} :: ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr } (\text{tail}_u'(-))$
 $\text{-utake} :: (\text{nat}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr } (\text{take}_u'(-, -))$
 $\text{-udrop} :: (\text{nat}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr } (\text{drop}_u'(-, -))$
 $\text{-ufilter} :: ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a \text{ set}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr } (\text{infixl } \lfloor_u \ 75)$
 $\text{-uextract} :: ('a \text{ set}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a \text{ list}, 'a) \text{ uexpr } (\text{infixl } \lfloor_u \ 75)$
 $\text{-uelems} :: ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow ('a \text{ set}, 'a) \text{ uexpr } (\text{elems}_u'(-))$
 $\text{-usorted} :: ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow (\text{bool}, 'a) \text{ uexpr } (\text{sorted}_u'(-))$
 $\text{-udistinct} :: ('a \text{ list}, 'a) \text{ uexpr} \Rightarrow (\text{bool}, 'a) \text{ uexpr } (\text{distinct}_u'(-))$
 $\text{-uupto} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\langle \dots \rangle)$
 $\text{-uupt} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\langle \dots < \rangle)$
 $\text{-umap} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{map}_u)$
 $\text{-uzip} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{zip}_u)$
 $\text{-utr-iter} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{iter}[-]'(-))$

translations

$\langle \rangle == \ll \Box \gg$
 $\langle x, xs \rangle == \text{CONST bop } (op \#) \ x \ \langle xs \rangle$

$\langle x \rangle == \text{CONST bop } (op \ \#) \ x \ \ll[]\gg$
 $x \ \hat{ }_u \ y == \text{CONST bop } (op \ @) \ x \ y$
 $\text{last}_u(xs) == \text{CONST uop } \text{CONST last } xs$
 $\text{front}_u(xs) == \text{CONST uop } \text{CONST butlast } xs$
 $\text{head}_u(xs) == \text{CONST uop } \text{CONST hd } xs$
 $\text{tail}_u(xs) == \text{CONST uop } \text{CONST tl } xs$
 $\text{drop}_u(n, xs) == \text{CONST bop } \text{CONST drop } n \ xs$
 $\text{take}_u(n, xs) == \text{CONST bop } \text{CONST take } n \ xs$
 $\text{elems}_u(xs) == \text{CONST uop } \text{CONST set } xs$
 $\text{sorted}_u(xs) == \text{CONST uop } \text{CONST sorted } xs$
 $\text{distinct}_u(xs) == \text{CONST uop } \text{CONST distinct } xs$
 $xs \ \downarrow_u \ A == \text{CONST bop } \text{CONST seq-filter } xs \ A$
 $A \ \uparrow_u \ xs == \text{CONST bop } (op \ \downarrow_l) \ A \ xs$
 $\langle n..k \rangle == \text{CONST bop } \text{CONST upto } n \ k$
 $\langle n..<k \rangle == \text{CONST bop } \text{CONST upt } n \ k$
 $\text{map}_u f \ xs == \text{CONST bop } \text{CONST map } f \ xs$
 $\text{zip}_u \ xs \ ys == \text{CONST bop } \text{CONST zip } xs \ ys$
 $\text{iter}[n](P) == \text{CONST uop } (\text{CONST tr-iter } n) \ P$

syntax — Sets

$\text{-ufinite} \quad :: \text{logic} \Rightarrow \text{logic } (\text{finite}_u '(-))$
 $\text{-uempset} \quad :: ('a \ \text{set}, ' \alpha) \ \text{uexpr } (\{\}_u)$
 $\text{-uset} \quad :: \text{args} \Rightarrow ('a \ \text{set}, ' \alpha) \ \text{uexpr } (\{(-)\}_u)$
 $\text{-uunion} \quad :: ('a \ \text{set}, ' \alpha) \ \text{uexpr} \Rightarrow ('a \ \text{set}, ' \alpha) \ \text{uexpr} \Rightarrow ('a \ \text{set}, ' \alpha) \ \text{uexpr } (\text{infixl } \cup_u \ 65)$
 $\text{-uinter} \quad :: ('a \ \text{set}, ' \alpha) \ \text{uexpr} \Rightarrow ('a \ \text{set}, ' \alpha) \ \text{uexpr} \Rightarrow ('a \ \text{set}, ' \alpha) \ \text{uexpr } (\text{infixl } \cap_u \ 70)$
 $\text{-umem} \quad :: ('a, ' \alpha) \ \text{uexpr} \Rightarrow ('a \ \text{set}, ' \alpha) \ \text{uexpr} \Rightarrow (\text{bool}, ' \alpha) \ \text{uexpr } (\text{infix } \in_u \ 50)$
 $\text{-usubset} \quad :: ('a \ \text{set}, ' \alpha) \ \text{uexpr} \Rightarrow ('a \ \text{set}, ' \alpha) \ \text{uexpr} \Rightarrow (\text{bool}, ' \alpha) \ \text{uexpr } (\text{infix } \subset_u \ 50)$
 $\text{-usubseteq} \quad :: ('a \ \text{set}, ' \alpha) \ \text{uexpr} \Rightarrow ('a \ \text{set}, ' \alpha) \ \text{uexpr} \Rightarrow (\text{bool}, ' \alpha) \ \text{uexpr } (\text{infix } \subseteq_u \ 50)$
 $\text{-uconverse} \quad :: \text{logic} \Rightarrow \text{logic } ((\sim) \ [1000] \ 999)$
 $\text{-ucarrier} \quad :: \text{type} \Rightarrow \text{logic } ([_]_T)$
 $\text{-uid} \quad :: \text{type} \Rightarrow \text{logic } (\text{id}[_])$
 $\text{-uproduct} \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{infixr } \times_u \ 80)$
 $\text{-urelcomp} \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{infixr } ;_u \ 75)$

translations

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

syntax — Partial functions

-umap-plus :: *logic* \Rightarrow *logic* \Rightarrow *logic* (**infixl** \oplus_u 85)
-umap-minus :: *logic* \Rightarrow *logic* \Rightarrow *logic* (**infixl** \ominus_u 85)

translations

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

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*

3.6 Lifting set collectors

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

syntax

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

lift-definition ZedSetCompr ::

('*a set*, '*α*') *uexpr* \Rightarrow ('*a* \Rightarrow (*bool*, '*α*') *uexpr* \times ('*b*, '*α*') *uexpr*) \Rightarrow ('*b set*, '*α*') *uexpr*
is $\lambda A \text{ PF } b. \{ \text{snd } (PF \ x) \ b \mid x. x \in A \ b \wedge \text{fst } (PF \ x) \ b \} .$

translations

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

3.7 Lifting limits

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

definition *ulim-left* :: '*a*::*order-topology* \Rightarrow ('*a* \Rightarrow '*b*) \Rightarrow '*b*::*t2-space* **where**

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

definition *ulim-right* :: '*a*::*order-topology* \Rightarrow ('*a* \Rightarrow '*b*) \Rightarrow '*b*::*t2-space* **where**

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

definition *ucont-on* :: ('*a*::*topological-space* \Rightarrow '*b*::*topological-space*) \Rightarrow '*a set* \Rightarrow *bool* **where**

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

syntax

-ulim-left :: *id* \Rightarrow *logic* \Rightarrow *logic* \Rightarrow *logic* (*lim_u*'(- \rightarrow -⁻)'(-'))
-ulim-right :: *id* \Rightarrow *logic* \Rightarrow *logic* \Rightarrow *logic* (*lim_u*'(- \rightarrow -⁺)'(-'))
-ucont-on :: *logic* \Rightarrow *logic* \Rightarrow *logic* (**infix** *cont-on_u* 90)

translations

$$\begin{aligned}
\lim_u(x \rightarrow p^-)(e) &== \text{CONST } bop \text{ CONST } ulim\text{-left } p (\lambda x \cdot e) \\
\lim_u(x \rightarrow p^+)(e) &== \text{CONST } bop \text{ CONST } ulim\text{-right } p (\lambda x \cdot e) \\
f \text{ cont-on}_u A &== \text{CONST } bop \text{ CONST } continuous\text{-on } A f
\end{aligned}$$

3.8 Evaluation laws for expressions

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

lemmas *ueexpr-defs* =

zero-ueexpr-def
one-ueexpr-def
plus-ueexpr-def
uminus-ueexpr-def
minus-ueexpr-def
times-ueexpr-def
inverse-ueexpr-def
divide-ueexpr-def
sgn-ueexpr-def
abs-ueexpr-def
mod-ueexpr-def
eq-upred-def
numeral-ueexpr-simp
ulim-left-def
ulim-right-def
ucont-on-def
uIf-def

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

lemma *lit-ueval* [*ueval*]: $\llbracket \langle x \rangle \rrbracket_e b = x$
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*]: $\llbracket uop f x \rrbracket_e b = f (\llbracket x \rrbracket_e b)$
by (*transfer*, *simp*)

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

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

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

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

declare *ueexpr-defs* [*ueval*]

3.9 Misc laws

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

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

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

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

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

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

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

lemma *ulist-filter-empty* [simp]: $x \downarrow_u \{\}_u = \langle \rangle$
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*)⁺

lemma *iter-0* [simp]: $iter[0](t) = \langle \rangle$
by (*transfer*, *simp add: zero-list-def*)

lemma *ufun-apply-lit* [simp]:
 $\langle\!\langle f \rangle\!\rangle(\langle\!\langle x \rangle\!\rangle)_a = \langle\!\langle f(x) \rangle\!\rangle$
by (*transfer*, *simp*)

3.10 Literalise tactics

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

lemma *lit-zero* [lit_simps]: $\langle\!\langle 0 \rangle\!\rangle = 0$ **by** (*simp add: ueval*)

lemma *lit-one* [lit_simps]: $\langle\!\langle 1 \rangle\!\rangle = 1$ **by** (*simp add: ueval*)

lemma *lit-numeral* [lit_simps]: $\langle\!\langle numeral\ n \rangle\!\rangle = numeral\ n$ **by** (*simp add: ueval*)

lemma *lit-uminus* [lit_simps]: $\langle\!\langle -\ x \rangle\!\rangle = -\ \langle\!\langle x \rangle\!\rangle$ **by** (*simp add: ueval, transfer, simp*)

lemma *lit-plus* [lit_simps]: $\langle\!\langle x + y \rangle\!\rangle = \langle\!\langle x \rangle\!\rangle + \langle\!\langle y \rangle\!\rangle$ **by** (*simp add: ueval, transfer, simp*)

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

lemma *lit-plus-appl* [*lit-norm*]: $\langle op + \rangle (x)_a (y)_a = x + y$ **by** (*simp add: ueval, transfer, simp*)
lemma *lit-minus-appl* [*lit-norm*]: $\langle op - \rangle (x)_a (y)_a = x - y$ **by** (*simp add: ueval, transfer, simp*)
lemma *lit-mult-appl* [*lit-norm*]: $\langle op * \rangle (x)_a (y)_a = x * y$ **by** (*simp add: ueval, transfer, simp*)
lemma *lit-divide-apply* [*lit-norm*]: $\langle op / \rangle (x)_a (y)_a = x / y$ **by** (*simp add: ueval, transfer, simp*)

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

In general unliteralising converts function applications to corresponding expression liftings. Since some operators, like $+$ and $*$, have specific operators we also have to use $0 = \llbracket u \rrbracket$

$1 = \langle 1 :: ?'a \rangle$
 $?u + ?v = bop \ op + \ ?u \ ?v$
 $- \ ?u = uop \ uminus \ ?u$
 $?u - ?v = bop \ op - \ ?u \ ?v$
 $?u * ?v = bop \ op * \ ?u \ ?v$
 $inverse \ ?u = uop \ inverse \ ?u$
 $?u \text{ div } ?v = bop \ op \text{ div } \ ?u \ ?v$
 $sgn \ ?u = uop \ sgn \ ?u$
 $|?u| = uop \ abs \ ?u$
 $?u \bmod ?v = bop \ op \bmod \ ?u \ ?v$
 $(?x =_u ?y) = bop \ op = \ ?x \ ?y$
 $numeral \ ?x = \langle numeral \ ?x \rangle$
 $ulim-left = (\lambda p. \ Lim \ (at-left \ p))$
 $ulim-right = (\lambda p. \ Lim \ (at-right \ p))$
 $ucont-on = (\lambda f \ A. \ continuous-on \ A \ f)$

$uIf = If$ 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*: $uop \ numeral \ x = Abs-uexpr \ (\lambda b. \ numeral \ (\llbracket x \rrbracket_e \ b))$
by (*simp add: uop-def*)

lemma *lit-numeral-2*: $Abs-uexpr \ (\lambda b. \ numeral \ v) = numeral \ v$
by (*metis lit.abs-eq lit-numeral*)

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

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

final unliteralises at the end.

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

lemma $(1::(int, 'α) \text{ uexpr}) + \ll 2 \gg = 4 \longleftrightarrow \ll 3 \gg = 4$
apply (*uexpr-simp*) **oops**

end

4 Unrestriction

theory *utp-unrest*
imports *utp-expr*
begin

4.1 Definitions and Core Syntax

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

Unrestriction was first defined in the work of Marcel Oliveira [19, 18] in his UTP mechanisation in *ProofPowerZ*. Our definition modifies his in that our variables are semantically characterised as lenses, and supported by the lens laws, rather than named syntactic entities. We effectively fuse the ideas from both Feliachi [8] and Oliveira's [18] mechanisations of the UTP, the former being also purely semantic in nature.

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

consts
unrest :: ' $a \Rightarrow 'b \Rightarrow bool$

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

translations
 $-unrest \ x \ p == CONST \ unrest \ x \ p$
 $-unrest \ (-salphaset \ (-salphamk \ (x +_L \ y))) \ P \leq -unrest \ (x +_L \ y) \ P$

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-uexpr* :: ' $a \implies 'α \Rightarrow ('b, 'α) \text{ uexpr} \Rightarrow bool$

is $\lambda x e. \forall b v. e \text{ (put}_x b v) = e b$.

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*)

4.2 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*]: $x \# \llbracket v \rrbracket$

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 $P :: ('a, 'a) \text{ uepr}$
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 $P :: ('a, 'a) \text{ uepr}$
assumes $\text{mwb-lens } y x \approx_L y x \# P$
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 $P :: ('a, 'a) \text{ uepr}$
assumes $\text{bij-lens } X X \# P$
shows $\Sigma \# P$
using *assms bij-lens-equiv-id lens-equiv-def unrest-sublens* **by** *blast*

lemma *bij-lens-unrest-all-eq*:

fixes $P :: ('a, 'α) \text{ uexpr}$

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 $e :: ('a, 'α) \text{ uexpr}$

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 $P :: ('a, 'α) \text{ uexpr}$

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

shows $\text{unrest } (x +_L y) P \longleftrightarrow (x \# P) \wedge (y \# P)$

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*]: $\llbracket \text{mwb-lens } x; x \bowtie y \rrbracket \implies y \# \text{var } x$

by (*transfer, auto*)

lemma *unrest-iuvar* [*unrest*]: $\llbracket \text{mwb-lens } x; x \bowtie y \rrbracket \implies \$y \# \$x$

by (*simp add: unrest-var*)

lemma *unrest-ouvar* [*unrest*]: $\llbracket \text{mwb-lens } x; x \bowtie y \rrbracket \implies \$y' \# \$x'$

by (*simp add: unrest-var*)

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

lemma *unrest-iuvar-ouvar* [*unrest*]:

fixes $x :: ('a \implies 'α)$

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 $x :: ('a \implies 'α)$

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*]: $x \# e \implies x \# \text{uop } f e$

by (*transfer, simp*)

lemma *unrest-bop* [*unrest*]: $\llbracket x \# u; x \# v \rrbracket \implies x \# \text{bop } f u v$

by (*transfer, simp*)

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

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

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

lemma *unrest-eq* [*unrest*]: $\llbracket x \# u; x \# v \rrbracket \Longrightarrow x \# u =_u v$
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*]: $x \# (\text{numeral } n)$
by (*simp add: numeral-uepr-simp unrest-lit*)

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

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

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

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

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

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

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

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*]:
 $\llbracket \bigwedge x. v \# F \ x \rrbracket \Longrightarrow v \# (\lambda x. F \ x)$
by (*transfer*, *simp*)

end

5 Used-by

theory *utp-usedby*
imports *utp-unrest*

begin

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

consts

usedBy :: 'a \Rightarrow 'b \Rightarrow bool

syntax

-usedBy :: *salpha* \Rightarrow *logic* \Rightarrow *logic* \Rightarrow *logic* (**infix** \bowtie 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 λ *x e*. (\forall *b b'*. *e* (*b' ⊕_L b on x*) = *e b*) .

adhoc-overloading *usedBy usedBy-uepr*

lemma *usedBy-lit* [*unrest*]: *x* \bowtie $\ll v \gg$

by (*transfer*, *simp*)

lemma *usedBy-sublens*:

fixes *P* :: ('a, 'α) *uepr*

assumes *x* \bowtie *P x* \subseteq_L *y vwb-lens y*

shows *y* \bowtie *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* \bowtie *P* \Longrightarrow $\&x$ \bowtie *P*

by (*transfer*, *simp* *add: lens-defs*)

lemma *usedBy-lens-plus-1* [*unrest*]: *x* \bowtie *P* \Longrightarrow *x;y* \bowtie *P*

by (*transfer*, *simp* *add: lens-defs*)

lemma *usedBy-lens-plus-2* [*unrest*]: $\ll x \bowtie y; y \bowtie P \gg \Longrightarrow x;y \bowtie P$

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* \bowtie *P x* \bowtie *y*

shows *y* \nbowtie *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* \bowtie *var y*

using *assms*

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

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

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

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

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

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

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

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

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

lemma *usedBy-divide* [*unrest*]: $\llbracket x \Vdash u; x \Vdash v \rrbracket \implies x \Vdash u / v$
by (*simp add: divide-uepr-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*]:
 $\text{vwb-lens } x \implies x \Vdash \&x:y$
by (*transfer*, *simp add: lens-defs*)

end

6 Substitution

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

6.1 Substitution definitions

Variable substitution, like unrestriction, will be characterised semantically using lenses and state-spaces. Effectively a substitution 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 subst-upd-uvar :: (' $\alpha$ , ' $\beta$ ) psubst  $\Rightarrow$  ('a  $\Rightarrow$  ' $\beta$ )  $\Rightarrow$  ('a, ' $\alpha$ ) uexpr  $\Rightarrow$  (' $\alpha$ , ' $\beta$ ) psubst where
subst-upd-uvar  $\sigma x v = (\lambda b. put_x (\sigma b) (\llbracket v \rrbracket_e b))$ 
```

ad hoc-overloading

subst-upd 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 *usubst-lookup* :: ($'\alpha, '\beta$) *psubst* \Rightarrow ($'a \Rightarrow '\beta$) \Rightarrow ($'a, '\alpha$) *uepr* ($\langle - \rangle_s$)
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 *unrest-usubst* :: ($'a \Rightarrow '\alpha$) \Rightarrow $'\alpha$ *usubst* \Rightarrow *bool*
where *unrest-usubst* $x \sigma = (\forall \rho v. \sigma (\text{put}_x \rho v) = \text{put}_x (\sigma \rho) v)$

ad hoc-overloading

unrest unrest-usubst

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

definition *cond-subst* :: $'\alpha$ *usubst* \Rightarrow (*bool*, $'\alpha$) *uepr* \Rightarrow $'\alpha$ *usubst* \Rightarrow $'\alpha$ *usubst* ($(\exists - \triangleleft - \triangleright_s / -)$ [52,0,53]
52) **where**
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 *par-subst* :: $'\alpha$ *usubst* \Rightarrow ($'a \Rightarrow '\alpha$) \Rightarrow ($'b \Rightarrow '\alpha$) \Rightarrow $'\alpha$ *usubst* \Rightarrow $'\alpha$ *usubst* **where**
par-subst $\sigma_1 A B \sigma_2 = (\lambda s. (s \oplus_L (\sigma_1 s) \text{ on } A) \oplus_L (\sigma_2 s) \text{ on } B)$

6.2 Syntax translations

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

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

nonterminal *smaplet* **and** *smaplets* **and** *ueprs* **and** *salphas*

syntax

-smaplet :: [*salpha*, $'a$] \Rightarrow *smaplet* ($- \mapsto_s / -$)
:: *smaplet* \Rightarrow *smaplets* ($-$)
-SMaplets :: [*smaplet*, *smaplets*] \Rightarrow *smaplets* ($- / -$)
-SubstUpd :: [$'m$ *usubst*, *smaplets*] \Rightarrow $'m$ *usubst* ($- / (-)$) [900,0] 900)
-Subst :: *smaplets* \Rightarrow $'a \mapsto 'b$ ($(1[-])$)
-psubst :: [*logic*, *svars*, *ueprs*] \Rightarrow *logic*
-subst :: *logic* \Rightarrow *ueprs* \Rightarrow *salphas* \Rightarrow *logic* ($(-[- / -])$) [990,0,0] 991)

```

-uexprs :: [logic, uexprs] => uexprs (-,/ -)
        :: logic => 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

```

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 *subst-del* :: 'α usubst ⇒ ('a ⇒ 'α) ⇒ 'α usubst (**infix** $-_s$ 85) **where**
subst-del σ $x = \sigma(x \mapsto_s \&x)$

6.3 Substitution Application Laws

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

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

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

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

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

A substitution update naturally yields the given expression.

lemma *usubst-lookup-upd* [*usubst*]:
assumes *weak-lens* x
shows $\langle \sigma(x \mapsto_s v) \rangle_s x = v$
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 *vwb-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*)

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 $x\ y :: ('a \Longrightarrow 'a)$
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 $a: put_y (put_x b_1 ([\&y]_e b_1)) ([\&x]_e b_1) = put_y (put_x b_2 ([\&y]_e b_2)) ([\&x]_e b_2)$
by (*auto simp add: subst-upd-uvar-def*)
then have $(\forall a\ b\ c. put_x (put_y a\ b)\ c = put_y (put_x a\ c)\ b) \wedge$
 $(\forall a\ b. get_x (put_y a\ b) = get_x a) \wedge (\forall a\ b. get_y (put_x a\ b) = get_y a)$
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*]:
 $vwb-lens\ x \Longrightarrow [x \mapsto_s var\ x] = id$
apply (*simp add: subst-upd-uvar-def*)
apply (*transfer*)
apply (*rule ext*)
apply (*auto*)
done

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

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

fixes $x :: ('a \Rightarrow 'α)$
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-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-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-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-lens* $x \bowtie y$
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*)

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

lemma *usubst-apply-unrest* [*usubst*]:

$\ll vwb\text{-}lens\ x; x \nmid \sigma \rrbracket \Rightarrow \langle \sigma \rangle_s x = var\ x$
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-lens $x \Rightarrow id -_s x = id$
by (*simp add: subst-del-def subst-upd-uvar-def pr-var-def, transfer, auto*)

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

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

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

$x \bowtie y \Rightarrow \sigma(y \mapsto_s v) -_s x = (\sigma -_s x)(y \mapsto_s v)$
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*]: $x \nmid P \Rightarrow \sigma(x \mapsto_s v) \dagger P = \sigma \dagger P$

by (*simp add: subst-upd-uvar-def, transfer, auto*)

lemma *subst-unrest-2* [*usubst*]:

fixes $P :: ('a, 'α) uexpr$
assumes $x \nmid P \bowtie y$
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 $P :: ('a, 'α) uexpr$
assumes $x \# P \ x \bowtie y \ x \bowtie z$
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 $P :: ('a, 'α) uexpr$
assumes $x \# P \ x \bowtie y \ x \bowtie z \ x \bowtie u$
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 $P :: ('a, 'α) uexpr$
assumes $x \# P \ x \bowtie y \ x \bowtie z \ x \bowtie u \ x \bowtie v$
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*]: $x \# \sigma \implies \sigma \circ \varrho(x \mapsto_s v) = (\sigma \circ \varrho)(x \mapsto_s v)$
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*)

6.4 Substitution laws

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *subst-one* [*usubst*]: $\sigma \dagger 1 = 1$
by (*simp add: one-uexpr-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 $x :: ('a \Rightarrow 'α)$
shows $\varrho(x \mapsto_s v) \circ \sigma = (\varrho \circ \sigma)(x \mapsto_s \sigma \uparrow v)$
by (*rule ext, simp add: uexpr-defs subst-upd-uvar-def, transfer, simp*)

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

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

6.5 Ordering substitutions

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

definition *var-name-ord* :: $('a \Rightarrow 'α) \Rightarrow ('b \Rightarrow 'α) \Rightarrow \text{bool}$ **where**
 $[no-atp]: \text{var-name-ord } x \ y = \text{True}$

syntax

-var-name-ord :: $\text{salpha} \Rightarrow \text{salpha} \Rightarrow \text{bool}$ (**infix** \prec_v 65)

translations

-var-name-ord $x \ y == \text{CONST } \text{var-name-ord } x \ y$

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

lemma *usubst-upd-comm-ord* [*usubst*]:
assumes $x \bowtie y \ y \prec_v x$
shows $\sigma(x \mapsto_s u, y \mapsto_s v) = \sigma(y \mapsto_s v, x \mapsto_s u)$
by (*simp add: assms(1) usubst-upd-comm*)

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

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

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

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

6.6 Unrestriction laws

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

lemma *unrest-usubst-single* [*unrest*]:

$\llbracket \text{mwb-lens } x; x \# v \rrbracket \implies x \# P[v/x]$
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*)

6.7 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*]:
 $\text{id} \triangleleft b \triangleright_s \text{id} = \text{id}$
by (*auto simp add: cond-subst-def*)

6.8 Parallel Substitution Laws

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

7 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.

7.1 Theorem Attributes

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

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

7.2 Generic Methods

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

Generic Predicate Tactics

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

Generic Relational Tactics

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

```

(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)

```

7.3 Transfer Tactics

Next, we define the component tactics used for transfer.

7.3.1 Robust Transfer

Robust transfer uses the transfer method of the lifting package.

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

7.3.2 Faster Transfer

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

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

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

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

```

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

```

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

```

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

```

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 *uexpr-transfer-laws* *uexpr transfer laws*

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

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

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

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

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

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

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

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

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

method *fast-uexpr-transfer* =

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

7.4 Interpretation

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

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

7.5 User Tactics

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

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

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

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

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

ML-file *utp-tactics.ML*

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

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

```
method-setup rel-simp = <<
```

```

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

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

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

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

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

```

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

```

method rel-simp'
  uses simp
  = (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

8 Meta-level Substitution

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

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

lift-definition $msubst :: ('b \Rightarrow ('a, 'a) uexpr) \Rightarrow ('b, 'a) uexpr \Rightarrow ('a, 'a) uexpr$
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

```

9 Alphabetised Predicates

```

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

```

In this theory we begin to create an Isabelle version of the alphabetised predicate calculus that is described in Chapter 1 of the UTP book [14].

9.1 Predicate type and syntax

An alphabetised predicate is 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

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

9.2 Predicate operators

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

class *refine* = *order*

abbreviation *refineBy* :: $'a :: refine \Rightarrow 'a \Rightarrow bool$ (**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
  apply (intro-classes)
  apply (transfer, rule ext, auto)
  using sup-INF apply fastforce
  apply (transfer, rule ext, auto)
  using inf-SUP apply fastforce
  done

```

```

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

```

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

```

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

```

```

notation gfp ( $\mu$ )

```

notation $lfp (\nu)$

translations

$$\begin{aligned}\nu X \cdot P &== \text{CONST } lfp (\lambda X. P) \\ \mu X \cdot P &== \text{CONST } gfp (\lambda X. P)\end{aligned}$$

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

definition $true\text{-upred} = (\text{Orderings.top} :: 'a \text{ upred})$

definition $false\text{-upred} = (\text{Orderings.bot} :: 'a \text{ upred})$

definition $conj\text{-upred} = (\text{Lattices.inf} :: 'a \text{ upred} \Rightarrow 'a \text{ upred} \Rightarrow 'a \text{ upred})$

definition $disj\text{-upred} = (\text{Lattices.sup} :: 'a \text{ upred} \Rightarrow 'a \text{ upred} \Rightarrow 'a \text{ upred})$

definition $not\text{-upred} = (\text{uminus} :: 'a \text{ upred} \Rightarrow 'a \text{ upred})$

definition $diff\text{-upred} = (\text{minus} :: 'a \text{ upred} \Rightarrow 'a \text{ upred} \Rightarrow 'a \text{ upred})$

abbreviation $Conj\text{-upred} :: 'a \text{ upred set} \Rightarrow 'a \text{ upred } (\bigwedge - [900] 900) \text{ where}$
 $\bigwedge A \equiv \bigcap A$

abbreviation $Disj\text{-upred} :: 'a \text{ upred set} \Rightarrow 'a \text{ upred } (\bigvee - [900] 900) \text{ where}$
 $\bigvee A \equiv \bigcup A$

notation

$conj\text{-upred}$ (**infixr** \wedge_p 35) **and**
 $disj\text{-upred}$ (**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 $UINF :: ('a \Rightarrow 'a \text{ upred}) \Rightarrow ('a \Rightarrow ('b :: \text{complete-lattice}, 'a) \text{ uexpr}) \Rightarrow ('b, 'a) \text{ uexpr}$
is $\lambda P F b. \text{Sup } \{\llbracket F x \rrbracket_e b \mid x. \llbracket P x \rrbracket_e b\}.$

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

syntax

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

translations

$$\begin{aligned}\bigcap x \mid P \cdot F &=> \text{CONST } UINF (\lambda x. P) (\lambda x. F) \\ \bigcap x \cdot F &== \bigcap x \mid \text{true} \cdot F \\ \bigcap x \cdot F &== \bigcap x \mid \text{true} \cdot F \\ \bigcap x \in A \cdot F &=> \bigcap x \mid \llbracket x \rrbracket \in_u \llbracket A \rrbracket \cdot F\end{aligned}$$

$$\begin{aligned}
\Box x \in A \cdot F &\leq \Box x \mid \llbracket y \rrbracket \in_u \llbracket A \rrbracket \cdot F \\
\Box x \mid P \cdot F &\leq \text{CONST UINF } (\lambda y. P) (\lambda x. F) \\
\Box x \mid P \cdot F(x) &\leq \text{CONST UINF } (\lambda x. P) F \\
\Box x \mid P \cdot F &\Rightarrow \text{CONST USUP } (\lambda x. P) (\lambda x. F) \\
\Box x \cdot F &= \Box x \mid \text{true} \cdot F \\
\Box x \in A \cdot F &\Rightarrow \Box x \mid \llbracket x \rrbracket \in_u \llbracket A \rrbracket \cdot F \\
\Box x \in A \cdot F &\leq \Box x \mid \llbracket y \rrbracket \in_u \llbracket A \rrbracket \cdot F \\
\Box x \mid P \cdot F &\leq \text{CONST USUP } (\lambda y. P) (\lambda x. F) \\
\Box x \mid P \cdot F(x) &\leq \text{CONST USUP } (\lambda x. P) F
\end{aligned}$$

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

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

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

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

ad hoc overloading

uttrue true-upred **and**
ufalse false-upred **and**
unot not-upred **and**

uconj conj-upred and
udisj disj-upred and
wimpl impl and
wiff iff-upred and
uex ex and
uall all and
ushEx shEx and
ushAll shAll

syntax

-uneq :: *logic* \Rightarrow *logic* \Rightarrow *logic* (**infixl** \neq_u 50)
-unmem :: (*'a*, *'α*) *uexpr* \Rightarrow (*'a set*, *'α*) *uexpr* \Rightarrow (*bool*, *'α*) *uexpr* (**infix** \notin_u 50)

translations

$x \neq_u y == \text{CONST } \text{unot } (x =_u y)$
 $x \notin_u A == \text{CONST } \text{unot } (\text{CONST } \text{bop } (op \in) x A)$

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

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

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

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

9.3 Unrestriction Laws

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

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

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

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

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

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

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

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

lemma *unrest-USUP-mem* [*unrest*]:
 $\llbracket (\bigwedge i. i \in A \Longrightarrow x \# P(i)) \rrbracket \Longrightarrow x \# (\bigcup i \in A \cdot P(i))$
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 $mwb\text{-}lens\ x\ y \subseteq_L x\ y \# P$
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*)

9.4 Used-by laws

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

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

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

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

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

9.5 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*]:
 $mwb\text{-}lens\ x \Longrightarrow \sigma(x \mapsto_s v) \dagger (\exists x \cdot P) = \sigma \dagger (\exists x \cdot P)$
by (*pred-auto*)

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

lemma *subst-ex-indep* [*usubst*]:
assumes $x \bowtie y \ y \nmid v$
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)+
end

```

10 UTP Events

```

theory utp-event
imports utp-pred
begin

```

10.1 Events

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

```

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

```

10.2 Channels

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

```

type-synonym ('a, ' $\vartheta$ ) chan = 'a  $\Rightarrow$  ' $\vartheta$  event

```

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

10.2.1 Operators

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

definition *chan-type* :: ('a, ' ϑ) chan \Rightarrow 'a set (δ_u) **where**
 $[upred-defs]: \delta_u c = UNIV$

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

definition *chan-apply* ::
('a, ' ϑ) chan \Rightarrow ('a, ' α) uexpr \Rightarrow (' ϑ event, ' α) uexpr ('(-/-')_u) **where**
 $[upred-defs]: (c \cdot e)_u = \ll c \gg (e)_a$

lemma *unrest-chan-apply* [unrest]: $x \# e \Longrightarrow x \# (c \cdot e)_u$
by (rel-auto)

lemma *usubst-chan-apply* [usubst]: $\sigma \dagger (c \cdot v)_u = (c \cdot \sigma \dagger v)_u$
by (rel-auto)

```

lemma msubst-event [usubst]:
   $(c \cdot v \ x)_u \llbracket x \rightarrow u \rrbracket = (c \cdot (v \ x) \llbracket x \rightarrow u \rrbracket)_u$ 
  by (pred-simp)

lemma msubst-event-2 [usubst]:
   $(c \cdot v \ x \ y)_u \llbracket (x, y) \rightarrow u \rrbracket = (c \cdot (v \ x \ y) \llbracket (x, y) \rightarrow u \rrbracket)_u$ 
  by (pred-simp)+

end

```

11 Alphabet Manipulation

```

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

```

11.1 Preliminaries

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

In this theory we will set up operators for extending and contracting 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* :: $('a, 'b) \text{ uexpr} \Rightarrow ('b, 'a) \text{ lens} \Rightarrow ('a, 'a) \text{ uexpr}$ (**infixr** \oplus_p 95)
is $\lambda P \ x \ b. P \ (\text{get}_x \ b)$.

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

lemma *aext-event* [alpha]: $(c \cdot v)_u \oplus_p a = (c \cdot v \oplus_p a)_u$
by (*pred-auto*)

Alphabet extension distributes through the function liftings.

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

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

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

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

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

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

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

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

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

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

lemma *aext-var* [*alpha*]:
 $var\ x\ \oplus_p\ a = var\ (x\ ;_L\ a)$
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*]: $vwb\text{-}lens\ a \implies (\bigsqcap A) \oplus_p a = (\bigsqcap_{P \in A} P \oplus_p a)$
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*]:
 $\llbracket mwb\text{-}lens\ a; x \# p \rrbracket \implies unrest\ (x\ ;_L\ a)\ (p \oplus_p a)$
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*]:
 $a \bowtie b \implies b \# (p \oplus_p a)$
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 $arestr :: ('a, 'α) uexpr \Rightarrow ('β, 'α) lens \Rightarrow ('a, 'β) uexpr$ (**infixr** \vdash_e 90)
is $\lambda P x b. P (create_x b)$.

update-uexpr-rep-eq-thms

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

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

lemma $arestr-lit$ $[simp]$: $\langle\!\langle v \rangle\!\rangle \vdash_e a = \langle\!\langle v \rangle\!\rangle$
by ($pred-auto$)

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

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

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

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

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

lemma $arestr-false$ $[simp]$: $false \vdash_e a = false$
by ($pred-auto$)

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

lemma *arestr-and* [*alpha*]: $(P \wedge Q) \upharpoonright_{ex} = (P \upharpoonright_{ex} \wedge Q \upharpoonright_{ex})$
by (*pred-auto*)

lemma *arestr-or* [*alpha*]: $(P \vee Q) \upharpoonright_{ex} = (P \upharpoonright_{ex} \vee Q \upharpoonright_{ex})$
by (*pred-auto*)

lemma *arestr-imp* [*alpha*]: $(P \Rightarrow Q) \upharpoonright_{ex} = (P \upharpoonright_{ex} \Rightarrow Q \upharpoonright_{ex})$
by (*pred-auto*)

11.4 Predicate Alphabet Restriction

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

definition *upred-ares* :: $'\alpha \text{ upred} \Rightarrow (' \beta \Longrightarrow ' \alpha) \Rightarrow ' \beta \text{ upred}$
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*]:
 $a \Vdash P \Longrightarrow (P \upharpoonright_p a) \oplus_p a = P$
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*]: $(P \vee Q) \upharpoonright_{px} = (P \upharpoonright_{px} \vee Q \upharpoonright_{px})$
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*]:

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

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

lemma *upd-subst-ext* [*alpha*]:
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*]:
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*]:
vwb-lens $a \implies (\sigma \oplus_s a) \circ (\varrho \oplus_s a) = (\sigma \circ \varrho) \oplus_s a$
by *pred-auto*

11.7 Substitution Alphabet Restriction

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

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

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

lemma *upd-subst-res* [*alpha*]:
mwb-lens $x \implies \sigma(\&x:y \mapsto_s v) \upharpoonright_s x = (\sigma \upharpoonright_s x)(\&y \mapsto_s v \upharpoonright_e x)$
by (*pred-auto*)

lemma *subst-ext-res* [*usubst*]:
mwb-lens $x \implies (\sigma \oplus_s x) \upharpoonright_s x = \sigma$

by (*pred-auto*)

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

12 Lifting Expressions

theory *utp-lift*
imports
utp-alphabet
begin

12.1 Lifting definitions

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

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

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

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

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

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

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

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

lemma *lift-post-var* [*simp*]:
 $\lceil \text{var } x \rceil_> = \x'
 by (*alpha-tac*)

12.3 Substitution Laws

lemma *pre-var-subst* [*usubst*]:
 $\sigma(\$x \mapsto_s \ll v \gg) \uparrow \lceil P \rceil_< = \sigma \uparrow \lceil P[\ll v \gg / \&x] \rceil_<$
 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 $\$x' \# \lceil p \rceil_<$
 by (*pred-auto*)

end

13 Predicate Calculus Laws

theory *utp-pred-laws*
 imports *utp-pred*
 begin

13.1 Propositional Logic

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

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

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

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

lemma *taut-conj*: $'A \wedge B' = ('A' \wedge 'B')$
 by (*rel-auto*)

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

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

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

Linking refinement and HOL implication

lemma *refine-prop-intro*:

assumes $\Sigma \# P \Sigma \# Q \text{ 'Q' } \Longrightarrow \text{ 'P' }$
shows $P \sqsubseteq Q$
using *assms*
by (*pred-auto*)

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

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

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

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

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

lemma *refBy-order*: $P \sqsubseteq Q = \text{ 'Q } \Rightarrow P\text{'}$
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*: $P = R \Longrightarrow ((P::'\alpha \text{ upred}) \vee Q) = (R \vee Q)$
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]: $\text{false} \sqcap P = P \sqcap \text{false} = P$
 by (pred-auto)+

lemma true-inf [simp]: $\text{true} \sqcup P = P \sqcup \text{true} = P$
 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]: $\text{true} \neq \text{false} \text{ false} \neq \text{true}$
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]: $(P \Leftrightarrow \text{true}) = P$
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 $P Q :: 'a \text{ upred}$
shows $(P \sqcap Q) = (P \vee Q)$
by (pred-auto)

lemma *usup-and*:
fixes $P Q :: 'a \text{ upred}$
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]: $(\bigsqcup P \mid F(P) \cdot \text{true}) = \text{true}$
by (pred-auto)

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

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

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

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

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

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

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

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

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

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

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

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

lemma *not-UINF*: $(\neg (\bigsqcap_{i \in A} \cdot P(i))) = (\bigsqcup_{i \in A} \cdot \neg P(i))$
 by (pred-auto)

lemma *not-USUP*: $(\neg (\bigsqcup_{i \in A} \cdot P(i))) = (\bigsqcap_{i \in A} \cdot \neg P(i))$
 by (pred-auto)

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

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

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

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

lemma *UINF-atLeast-first*:
 $P(n) \sqcap (\bigsqcap_{i \in \{Suc\ n..\}} \cdot P(i)) = (\bigsqcap_{i \in \{n..\}} \cdot P(i))$

proof –
 have $insert\ n\ \{Suc\ n..\} = \{n..\}$
 by (auto)
 thus ?thesis
 by (metis UINF-insert)
qed

lemma *UINF-atLeast-Suc*:
 $(\bigsqcap_{i \in \{Suc\ m..\}} \cdot P(i)) = (\bigsqcap_{i \in \{m..\}} \cdot P(Suc\ i))$
 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-UNIF-dist*:

$(P \wedge (\prod Q \in S \cdot F(Q))) = (\prod Q \in S \cdot P \wedge F(Q))$
by $(simp\ add: upred-defs\ bop.rep-eq\ lit.rep-eq, pred-auto)$

lemma *conj-UNIF-ind-dist*:

$(P \wedge (\prod Q \cdot F(Q))) = (\prod Q \cdot P \wedge F(Q))$
by $pred-auto$

lemma *disj-UNIF-dist*:

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

lemma *UNIF-conj-UNIF [simp]*:

$((\prod i \in I \cdot P(i)) \vee (\prod i \in I \cdot Q(i))) = (\prod i \in I \cdot P(i) \vee Q(i))$
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 $(subst\ ueqpr-eq-iff, auto\ simp\ add: conj-upred-def\ USUP.rep-eq\ inf-ueqpr.rep-eq\ bop.rep-eq\ lit.rep-eq)$

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 *UNIF-all-cong [cong]*:

assumes $\bigwedge P. F(P) = G(P)$
shows $(\prod P \cdot F(P)) = (\prod P \cdot G(P))$
by $(simp\ add: UNIF-as-Sup-collect\ assms)$

lemma *UNIF-cong*:

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

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*: $A \subseteq B \implies (\prod P \in B \cdot F(P)) \sqsubseteq (\prod P \in A \cdot F(P))$
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*: $(\prod P \in A \cdot F(P) \Rightarrow G(P)) = ((\bigsqcup P \in A \cdot F(P)) \Rightarrow (\prod P \in A \cdot G(P)))$
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*: $(\prod x \cdot P(x)) = (\exists x \cdot P(x))$
by (*pred-simp*)

lemma *UINF-all-nats* [*simp*]:
fixes $P :: \text{nat} \Rightarrow 'a \text{ upred}$
shows $(\prod n \cdot \prod i \in \{0..n\} \cdot P(i)) = (\prod n \cdot P(n))$
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*:
 $m < n \implies (\prod i \in \{m..<n\} \cdot P(i)) = ((P(m) :: 'a \text{ upred}) \vee (\prod i \in \{\text{Suc } m..<n\} \cdot P(i)))$
apply (*rel-auto*) **using** *Suc-leI le-eq-less-or-eq* **by** *auto*

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

lemma *UINF-Suc-shift*: $(\prod i \in \{\text{Suc } 0..<\text{Suc } n\} \cdot P(i)) = (\prod i \in \{0..<n\} \cdot P(\text{Suc } i))$
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*:
 $(\prod i \in \{0..<\text{length}(xs)\} \cdot f(xs ! i)) = \text{foldr } op \vee (\text{map } f \text{ } xs) \text{ } false$

```

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

```

```

lemma USUP-list-conv:
  ( $\bigcup i \in \{0..<\text{length}(xs)\} \cdot f (xs \ ! \ i)$ ) = foldr op  $\wedge$  (map f xs) true
apply (induct xs)
apply (rel-auto)
apply (simp-all add: USUP-upto-expand-first USUP-Suc-shift)
done

```

```

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

```

```

lemma UINF-pred-ueq [simp]:
  ( $\bigcap x \mid \ll x \gg =_u v \cdot P(x)$ ) = (P x) $\llbracket x \rightarrow v \rrbracket$ 
  by (pred-auto)

```

```

lemma UINF-pred-lit-eq [simp]:
  ( $\bigcap x \mid \ll x = v \gg \cdot P(x)$ ) = (P v)
  by (pred-auto)

```

13.3 Equality laws

```

lemma eq-upred-refl [simp]: ( $x =_u x$ ) = true
  by (pred-auto)

```

```

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

```

```

lemma eq-cong-left:
  assumes  $\text{vwb-lens } x \ \$x \ \# \ Q \ \$x' \ \# \ Q \ \$x \ \# \ R \ \$x' \ \# \ R$ 
  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  $x :: ('a \Longrightarrow 'a)$ 
  assumes vwb-lens x
  shows ( $P \wedge \$x =_u v$ ) = ( $P \llbracket v / \$x \rrbracket \wedge \$x =_u v$ )
  using assms
  by (pred-simp, (metis vwb-lens-wb wb-lens.get-put)+)

```

```

lemma conj-eq-out-var-subst:
  fixes  $x :: ('a \Longrightarrow 'a)$ 
  assumes vwb-lens x
  shows ( $P \wedge \$x' =_u v$ ) = ( $P \llbracket v / \$x' \rrbracket \wedge \$x' =_u v$ )
  using assms
  by (pred-simp, (metis vwb-lens-wb wb-lens.get-put)+)

```

```

lemma conj-pos-var-subst:
  assumes vwb-lens x

```

shows $(\$x \wedge Q) = (\$x \wedge Q[\text{true}/\$x])$
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 $(\neg \$x \wedge Q) = (\neg \$x \wedge Q[\text{false}/\$x])$
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*]: $(p =_u \text{true}) = p$
by (*pred-auto*)

lemma *upred-eq-false* [*simp*]: $(p =_u \text{false}) = (\neg p)$
by (*pred-auto*)

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

lemma *upred-false-eq* [*simp*]: $(\text{false} =_u p) = (\neg p)$
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*: $[\bigwedge x. P x = Q x] \implies \text{shEx } P = \text{shEx } Q$
by (*pred-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*: $[\bigwedge x. P x = Q x] \implies \text{shAll } P = \text{shAll } Q$
by (*pred-auto*)

Quantifier lifting

named-theorems *uquant-lift*

lemma *shEx-lift-conj-1* [*uquant-lift*]:
 $((\exists x \cdot P(x)) \wedge Q) = (\exists x \cdot P(x) \wedge Q)$

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 $(P = Q) \longleftrightarrow (\forall v. P[\llbracket v \gg / x \rrbracket] = Q[\llbracket v \gg / x \rrbracket])$
 using *assms*
 by (*pred-auto*, *metis vwb-lens-wb wb-lens.source-stability*)

lemma *eq-split-substI*:
 assumes *vwb-lens* *x* $\wedge v. P[\llbracket v \gg / x \rrbracket] = Q[\llbracket v \gg / x \rrbracket]$
 shows $P = Q$
 using *assms*(1) *assms*(2) *eq-split-subst* by *blast*

lemma *taut-split-subst*:
 assumes *vwb-lens* *x*
 shows $'P' \longleftrightarrow (\forall v. 'P[\llbracket v \gg / x \rrbracket]')$
 using *assms*
 by (*pred-auto*, *metis vwb-lens-wb wb-lens.source-stability*)

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

lemma *bool-eq-splitI*:
 assumes *vwb-lens* *x* $P[\llbracket true / x \rrbracket] = Q[\llbracket true / x \rrbracket]$ $P[\llbracket false / x \rrbracket] = Q[\llbracket false / x \rrbracket]$
 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 $'P' = '(P[\llbracket false / x \rrbracket] \wedge P[\llbracket true / x \rrbracket])'$
proof –
 from *assms* have $'P' = (\forall v. 'P[\llbracket v \gg / x \rrbracket]')$
 by (*subst taut-split-subst*[*of x*], *auto*)
 also have $\dots = ('P[\llbracket \llbracket True \rrbracket \gg / x \rrbracket]' \wedge 'P[\llbracket \llbracket False \rrbracket \gg / x \rrbracket]')$
 by (*metis* (*mono-tags*, *lifting*))
 also have $\dots = '(P[\llbracket false / x \rrbracket] \wedge P[\llbracket true / x \rrbracket])'$
 by (*pred-auto*)
 finally show *?thesis* .
qed

lemma *subst-eq-replace*:
 fixes $x :: ('a \Longrightarrow 'a)$
 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* $x \# v$
shows $(\exists x \cdot P \wedge \text{var } x =_u v) = P[v/x]$
using *assms*
by (*pred-auto*)

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

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

lemma *exists-sub*: $\llbracket \text{mwb-lens } y; x \subseteq_L y \rrbracket \implies (\exists x \cdot \exists y \cdot P) = (\exists y \cdot P)$
by (*pred-auto*)

lemma *all-sub*: $\llbracket \text{mwb-lens } y; x \subseteq_L y \rrbracket \implies (\forall x \cdot \forall y \cdot P) = (\forall y \cdot P)$
by (*pred-auto*)

lemma *ex-commute*:

assumes $x \bowtie y$
shows $(\exists x \cdot \exists y \cdot P) = (\exists y \cdot \exists x \cdot P)$
using *assms*
apply (*pred-auto*)
using *lens-indep-comm* **apply** *fastforce* +
done

lemma *all-commute*:

assumes $x \bowtie y$
shows $(\forall x \cdot \forall y \cdot P) = (\forall y \cdot \forall x \cdot P)$
using *assms*
apply (*pred-auto*)
using *lens-indep-comm* **apply** *fastforce* +
done

lemma *ex-equiv*:

assumes $x \approx_L y$
shows $(\exists x \cdot P) = (\exists y \cdot P)$
using *assms*
by (*pred-simp*, *metis* (*no-types*, *lifting*) *lens.select-convs*(2))

lemma *all-equiv*:

assumes $x \approx_L y$
shows $(\forall x \cdot P) = (\forall y \cdot P)$
using *assms*
by (*pred-simp*, *metis* (*no-types*, *lifting*) *lens.select-convs*(2))

lemma *ex-zero*:

$(\exists \emptyset \cdot P) = P$
by (*pred-auto*)

lemma *all-zero*:

$(\forall \emptyset \cdot P) = P$
by (*pred-auto*)

lemma *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*:

$vwb\text{-}lens\ x \implies (x \# P) \longleftrightarrow ((\exists x \cdot P) = P)$
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*: $x \# P \implies (\exists x \cdot P) = P$

by (*pred-auto*)

lemma *all-unrest*: $x \# P \implies (\forall x \cdot P) = P$

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*]: $\text{true} \triangleleft b \triangleright \text{false} = b$ **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*]: $(P \triangleleft \text{true} \triangleright Q) = P$ **by** (*pred-auto*)

lemma *cond-unit-F* [*simp*]: $(P \triangleleft \text{false} \triangleright Q) = Q$ **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-UNIF-dist*: $(\prod P \in S \cdot F(P)) \triangleleft b \triangleright (\prod P \in S \cdot G(P)) = (\prod P \in S \cdot F(P) \triangleleft b \triangleright G(P))$
by (*pred-auto*)

lemma *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*:

$(P_1 \triangleleft b \triangleright Q_1 \wedge P_2 \triangleleft b \triangleright Q_2) = (P_1 \wedge P_2) \triangleleft b \triangleright (Q_1 \wedge Q_2)$

by *pred-auto*

lemma *disj-conds*:

$(P_1 \triangleleft b \triangleright Q_1 \vee P_2 \triangleleft b \triangleright Q_2) = (P_1 \vee P_2) \triangleleft b \triangleright (Q_1 \vee Q_2)$

by *pred-auto*

lemma *cond-mono*:

$\llbracket P_1 \sqsubseteq P_2; Q_1 \sqsubseteq Q_2 \rrbracket \implies (P_1 \triangleleft b \triangleright Q_1) \sqsubseteq (P_2 \triangleleft b \triangleright Q_2)$

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::preorder, 'α) *uexpr*

shows $(x \leq_u x) = \text{true}$

by (*pred-auto*)

lemma *uzero-le-laws* [*simp*]:

$(0 :: ('a::\{\text{linordered-semidom}\}, 'α) \text{uexpr}) \leq_u \text{numeral } x = \text{true}$

$(1 :: ('a::\{\text{linordered-semidom}\}, 'α) \text{uexpr}) \leq_u \text{numeral } x = \text{true}$

$(0 :: ('a::\{\text{linordered-semidom}\}, 'α) \text{uexpr}) \leq_u 1 = \text{true}$

by (*pred-simp*)+

lemma *unumeral-le-1* [*simp*]:

assumes $(\text{numeral } i :: 'a::\{\text{numeral,ord}\}) \leq \text{numeral } j$

shows $(\text{numeral } i :: ('a, 'α) \text{uexpr}) \leq_u \text{numeral } j = \text{true}$

using *assms* **by** (*pred-auto*)

lemma *unumeral-le-2* [*simp*]:

assumes (*numeral* *i* :: '*a*::{*numeral*,*linorder*} > *numeral* *j*)
shows (*numeral* *i* :: ('*a*, '*α*) *uepr*) ≤_{*u*} *numeral* *j* = *false*
using *assms* **by** (*pred-auto*)

lemma *uset-laws* [*simp*]:
 $x \in_u \{\} = \text{false}$
 $x \in_u \{m..n\}_u = (m \leq_u x \wedge x \leq_u n)$
by (*pred-auto*)⁺

lemma *pfun-entries-apply* [*simp*]:
 $(\text{entr}_u(d, f) :: (('k, 'v) \text{pfun}, 'α) \text{uepr})(i)_a = ((\ll f \gg(i)_a) \triangleleft i \in_u d \triangleright \perp_u)$
by (*pred-auto*)

lemma *uupdate-pfun* [*simp*]:
fixes *m* :: (('k, 'v) *pfun*, 'α) *uepr*
shows $\text{dom}_u(m(k \mapsto v)_u) = \{k\}_u \cup_u \text{dom}_u(m)$
by (*rel-auto*)

lemma *uapply-uupdate-pfun* [*simp*]:
fixes *m* :: (('k, 'v) *pfun*, 'α) *uepr*
shows $(m(k \mapsto v)_u)(i)_a = v \triangleleft i =_u k \triangleright m(i)_a$
by (*rel-auto*)

lemma *ulit-eq* [*simp*]: $x = y \implies (\ll x \gg =_u \ll y \gg) = \text{true}$
by (*rel-auto*)

lemma *ulit-neq* [*simp*]: $x \neq y \implies (\ll x \gg =_u \ll y \gg) = \text{false}$
by (*rel-auto*)

lemma *uset-mems* [*simp*]:
 $x \in_u \{y\}_u = (x =_u y)$
 $x \in_u A \cup_u B = (x \in_u A \vee x \in_u B)$
 $x \in_u A \cap_u B = (x \in_u A \wedge x \in_u B)$
by (*rel-auto*)⁺

13.10 Refinement By Observation

Function to obtain the set of observations of a predicate

definition *obs-upred* :: '*α* *upred* \Rightarrow '*α* *set* ($\ll \cdot \gg_o$)
where [*upred-defs*]: $\ll P \gg_o = \{b. \ll P \gg_e b\}$

lemma *obs-upred-refine-iff*:
 $P \sqsubseteq Q \iff \ll P \gg_o \subseteq \ll Q \gg_o$
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 $x \bowtie y \text{ bij-lens } (x +_L y) \ y \# P \ y \# Q \ \{v. 'P[\ll v \gg/x]\} \subseteq \{v. 'Q[\ll v \gg/x]\}$
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  $\Rightarrow$  'α upred

```

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

definition *Healthy* :: $'\alpha \text{ upred} \Rightarrow '\alpha \text{ health} \Rightarrow \text{bool}$ (**infix** is 30)
where $P \text{ is } H \equiv (H P = P)$

lemma *Healthy-def'*: $P \text{ is } H \longleftrightarrow (H P = P)$
unfolding *Healthy-def* **by** *auto*

lemma *Healthy-if*: $P \text{ is } H \implies (H P = P)$
unfolding *Healthy-def* **by** *auto*

lemma *Healthy-intro*: $H(P) = P \implies P \text{ is } H$
by (*simp add: Healthy-def*)

declare *Healthy-def'* [*upred-defs*]

abbreviation *Healthy-carrier* :: $'\alpha \text{ health} \Rightarrow '\alpha \text{ upred set}$ ($\llbracket - \rrbracket_H$)
where $\llbracket H \rrbracket_H \equiv \{P. P \text{ is } H\}$

lemma *Healthy-carrier-image*:
 $A \subseteq \llbracket \mathcal{H} \rrbracket_H \implies \mathcal{H} \text{ ' } A = A$
by (*auto simp add: image-def, (metis Healthy-if mem-Collect-eq subsetCE)+*)

lemma *Healthy-carrier-Collect*: $A \subseteq \llbracket H \rrbracket_H \implies A = \{H(P) \mid P. P \in A\}$
by (*simp add: Healthy-carrier-image Setcompr-eq-image*)

lemma *Healthy-func*:
 $\llbracket F \in \llbracket \mathcal{H}_1 \rrbracket_H \rightarrow \llbracket \mathcal{H}_2 \rrbracket_H; P \text{ is } \mathcal{H}_1 \rrbracket \implies \mathcal{H}_2(F(P)) = F(P)$
using *Healthy-if* **by** *blast*

lemma *Healthy-comp*:
 $\llbracket P \text{ is } \mathcal{H}_1; P \text{ is } \mathcal{H}_2 \rrbracket \implies P \text{ is } \mathcal{H}_1 \circ \mathcal{H}_2$
by (*simp add: Healthy-def*)

lemma *Healthy-apply-closed*:
assumes $F \in \llbracket H \rrbracket_H \rightarrow \llbracket H \rrbracket_H P \text{ is } H$
shows $F(P) \text{ is } H$
using *assms(1) assms(2)* **by** *auto*

lemma *Healthy-set-image-member*:
 $\llbracket P \in F \text{ ' } A; \bigwedge x. F x \text{ is } H \rrbracket \implies P \text{ is } H$
by *blast*

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* $F \in \llbracket id \rrbracket_H \rightarrow \llbracket H \rrbracket_H$
shows μF 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*: $\llbracket A \subseteq \llbracket H \rrbracket_H; P \in A \rrbracket \implies P$ is H
by *blast*

14.2 Properties of Healthiness Conditions

definition *Idempotent* :: $'\alpha$ health \Rightarrow bool **where**
 $Idempotent(H) \longleftrightarrow (\forall P. H(H(P)) = H(P))$

abbreviation *Monotonic* :: $'\alpha$ health \Rightarrow bool **where**
 $Monotonic(H) \equiv mono\ H$

definition *IMH* :: $'\alpha$ health \Rightarrow bool **where**
 $IMH(H) \longleftrightarrow Idempotent(H) \wedge Monotonic(H)$

definition *Antitone* :: $'\alpha$ health \Rightarrow bool **where**
 $Antitone(H) \longleftrightarrow (\forall P\ Q. Q \sqsubseteq P \longrightarrow (H(P) \sqsubseteq H(Q)))$

definition *Conjunctive* :: $'\alpha$ health \Rightarrow bool **where**
 $Conjunctive(H) \longleftrightarrow (\exists Q. \forall P. H(P) = (P \wedge Q))$

definition *FunctionalConjunctive* :: $'\alpha$ health \Rightarrow bool **where**
 $FunctionalConjunctive(H) \longleftrightarrow (\exists F. \forall P. H(P) = (P \wedge F(P)) \wedge Monotonic(F))$

definition *WeakConjunctive* :: $'\alpha$ health \Rightarrow bool **where**
 $WeakConjunctive(H) \longleftrightarrow (\forall P. \exists Q. H(P) = (P \wedge Q))$

definition *Disjunctuous* :: $'\alpha$ health \Rightarrow bool **where**
[upred-defs]: $Disjunctuous\ H = (\forall P\ Q. H(P \sqcap Q) = (H(P) \sqcap H(Q)))$

definition *Continuous* :: $'\alpha$ health \Rightarrow bool **where**
[upred-defs]: $Continuous\ H = (\forall A. A \neq \{\} \longrightarrow H(\bigsqcap A) = \bigsqcap (H \restriction A))$

lemma *Healthy-Idempotent* [closure]:
 $Idempotent\ H \implies H(P)$ is H
by (*simp* *add*: *Healthy-def* *Idempotent-def*)

lemma *Healthy-range*: $Idempotent\ H \implies range\ H = \llbracket H \rrbracket_H$
by (*auto* *simp* *add*: *image-def* *Healthy-if* *Healthy-Idempotent*, *metis* *Healthy-if*)

lemma *Idempotent-id* [simp]: $Idempotent\ id$
by (*simp* *add*: *Idempotent-def*)

lemma *Idempotent-comp* [intro]:
 $\llbracket Idempotent\ f; Idempotent\ g; f \circ g = g \circ f \rrbracket \implies Idempotent\ (f \circ g)$
by (*auto* *simp* *add*: *Idempotent-def* *comp-def*, *metis*)

lemma *Idempotent-image*: $Idempotent\ f \implies f \restriction f \restriction A = f \restriction A$
by (*metis* (*mono-tags*, *lifting*) *Idempotent-def* *image-cong* *image-image*)

lemma *Monotonic-id* [simp]: *Monotonic id*
by (simp add: monoI)

lemma *Monotonic-id'* [closure]:
mono ($\lambda X. X$)
by (simp add: monoI)

lemma *Monotonic-const* [closure]:
Monotonic ($\lambda x. c$)
by (simp add: mono-def)

lemma *Monotonic-comp* [intro]:
 $\llbracket \text{Monotonic } f; \text{Monotonic } g \rrbracket \implies \text{Monotonic } (f \circ g)$
by (simp add: mono-def)

lemma *Monotonic-inf* [closure]:
assumes *Monotonic P Monotonic Q*
shows *Monotonic* ($\lambda X. P(X) \sqcap Q(X)$)
using *assms* **by** (simp add: mono-def, rel-auto)

lemma *Monotonic-cond* [closure]:
assumes *Monotonic P Monotonic Q*
shows *Monotonic* ($\lambda X. P(X) \triangleleft b \triangleright Q(X)$)
by (simp add: assms cond-monotonic)

lemma *Conjunctive-Idempotent*:
Conjunctive(*H*) \implies *Idempotent*(*H*)
by (auto simp add: Conjunctive-def Idempotent-def)

lemma *Conjunctive-Monotonic*:
Conjunctive(*H*) \implies *Monotonic*(*H*)
unfolding *Conjunctive-def* *mono-def*
using *dual-order.trans* **by** *fastforce*

lemma *Conjunctive-conj*:
assumes *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 *WeakConjunctive-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*]: $\llbracket Continuous H; A \neq \{\} \rrbracket \implies H (\bigcap A) = (\bigcap P \in A. H(P))$
by (*simp add: Continuous-def*)

lemma *Continuous-Disjunctuous*: $Continuous H \implies Disjunctuous H$
apply (*auto simp add: Continuous-def Disjunctuous-def*)
apply (*rename-tac P Q*)
apply (*drule-tac x={P,Q} in spec*)
apply (*simp*)
done

lemma *Continuous-Monotonic* [*closure*]: $Continuous H \implies Monotonic H$
by (*simp add: Continuous-Disjunctuous Disjunctuous-Monotonic*)

lemma *Continuous-comp* [*intro*]:
 $\llbracket Continuous f; Continuous g \rrbracket \implies Continuous (f \circ g)$
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 $(\bigcap_{(i,j) \in A} . P i j) = (\bigcap_{x \in A} . P (\text{fst } x) (\text{snd } x))$
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 $(\bigcap_{(i,j,k) \in A} . P i j k) = (\bigcap_{x \in A} . P (\text{fst } x) (\text{fst } (\text{snd } x)) (\text{snd } (\text{snd } x)))$
by (*rel-auto*)

also have ... *is H*

by (*metis (mono-tags) UINF-mem-Continuous-closed assms(1) assms(2) assms(3) prod.collapse*)
finally show ?thesis .

qed

lemma *UINF-Continuous-closed* [closure]:

$\llbracket \text{Continuous } H; \bigwedge i. P(i) \text{ is } H \rrbracket \implies (\bigcap i . P(i)) \text{ is } H$
using *UINF-mem-Continuous-closed[of H UNIV P]*
by (*simp add: UINF-mem-UNIV*)

All continuous functions are also Scott-continuous

lemma *sup-continuous-Continuous* [closure]: *Continuous F* $\implies \text{sup-continuous } F$

by (*simp add: Continuous-def sup-continuous-def*)

lemma *USUP-healthy*: $A \subseteq \llbracket H \rrbracket_H \implies (\bigcup_{P \in A} . F(P)) = (\bigcup_{P \in A} . F(H(P)))$

by (*rule USUP-cong, simp add: Healthy-subset-member*)

lemma *UINF-healthy*: $A \subseteq \llbracket H \rrbracket_H \implies (\bigcap_{P \in A} . F(P)) = (\bigcap_{P \in A} . F(H(P)))$

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 [14].

15.1 Relational Alphabets

We set up convenient syntax to refer to the input and output parts of the alphabet, as is common in UTP. Since we are in a product space, these are simply the lenses fst_L and snd_L .

definition $in\alpha :: ('\alpha \Longrightarrow '\alpha \times '\beta)$ **where**
 $[lens-defs]: in\alpha = fst_L$

definition $out\alpha :: ('\beta \Longrightarrow '\alpha \times '\beta)$ **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-in-out*:

$\Sigma \approx_L \text{in}\alpha +_L \text{out}\alpha$

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

15.2 Relational Types and Operators

We create type synonyms for conditions (which are simply predicates) – i.e. relations without dashed variables –, alphabetised relations where the input and output alphabet can be different, and finally homogeneous relations.

type-synonym $'\alpha \text{ cond}$ = $'\alpha \text{ upred}$

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

useq :: $'a \Rightarrow 'b \Rightarrow 'c$ (**infixr** ;; 71)

uassigns :: $'a \text{ usubst} \Rightarrow 'b ((_)_a)$

uskip :: $'a$ (*II*)

We define a specialised version of the conditional where the condition can refer only to undashed variables, as is usually the case in programs, but not universally in UTP models. We implement this by lifting the condition predicate into the relational state-space with construction $\lceil b \rceil_{\leftarrow}$.

definition *lift-rcond* ($\lceil _ \rceil_{\leftarrow}$) **where**

$\lceil \text{upred-defs} \rceil$: $\lceil b \rceil_{\leftarrow} = \lceil b \rceil_{<}$

abbreviation

rcond :: $(' \alpha, ' \beta) \text{ urel} \Rightarrow ' \alpha \text{ cond} \Rightarrow (' \alpha, ' \beta) \text{ urel} \Rightarrow (' \alpha, ' \beta) \text{ urel}$

$((\beta \triangleleft - \triangleright_r / -) [52, 0, 53] 52)$

where $(P \triangleleft b \triangleright_r Q) \equiv (P \triangleleft \lceil b \rceil_{\leftarrow} \triangleright Q)$

Sequential composition is heterogeneous, and simply requires that the output alphabet of the first matches then input alphabet of the second. We define it by lifting HOL's built-in relational composition operator (*op O*). Since this returns a set, the definition states that the state binding *b* is an element of this set.

lift-definition *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 *seqh* :: $' \alpha \text{ hrel} \Rightarrow ' \alpha \text{ hrel} \Rightarrow ' \alpha \text{ hrel}$ (**infixr** ;;_h 71) **where**

seqh $P Q \equiv (P ;; Q)$

abbreviation *truer* :: $' \alpha \text{ hrel}$ (*true_h*) **where**

$truer \equiv true$

abbreviation $false_r :: 'a \text{ hrel } (false_h) \text{ where }$
 $false_r \equiv false$

We define the relational converse operator as an alphabet extrusion on the bijective lens $swap_L$ that swaps the elements of the product state-space.

abbreviation $conv-r :: ('a, 'a \times 'b) \text{ uexpr} \Rightarrow ('a, 'b \times 'a) \text{ uexpr } (- [999] 999)$
where $conv-r \ e \equiv e \oplus_p swap_L$

Assignment is defined using substitutions, where latter defines what each variable should map to. The definition of the operator identifies the after state binding, b' , with the substitution function applied to the before state binding b .

lift-definition $assigns-r :: 'a \text{ usubst} \Rightarrow 'a \text{ hrel}$
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 $skip-r :: 'a \text{ hrel} \text{ where }$
 $[urel-defs]: skip-r = assigns-r \ id$

ad hoc-overloading
 $uskip \ skip-r$

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

definition $segr-iter :: 'a \text{ list} \Rightarrow ('a \Rightarrow 'b \text{ hrel}) \Rightarrow 'b \text{ hrel} \text{ where }$
 $[urel-defs]: segr-iter \ xs \ P = foldr \ (\lambda \ i \ Q. \ P(i) ;; Q) \ xs \ II$

A singleton assignment simply applies a singleton substitution function, and similarly for a double assignment.

abbreviation $assign-r :: ('t \Rightarrow 'a) \Rightarrow ('t, 'a) \text{ uexpr} \Rightarrow 'a \text{ hrel}$
where $assign-r \ x \ v \equiv \langle [x \mapsto_s v] \rangle_a$

abbreviation $assign-2-r ::$
 $(t1 \Rightarrow 'a) \Rightarrow (t2 \Rightarrow 'a) \Rightarrow (t1, 'a) \text{ uexpr} \Rightarrow (t2, 'a) \text{ uexpr} \Rightarrow 'a \text{ hrel}$
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 $skip-ra :: ('b, 'a) \text{ lens} \Rightarrow 'a \text{ hrel} \text{ where }$
 $[urel-defs]: skip-ra \ v = (\$v' =_u \$v)$

Similarly, we define the alphabetised assignment operator.

definition $assigns-ra :: 'a \text{ usubst} \Rightarrow ('b, 'a) \text{ lens} \Rightarrow 'a \text{ hrel } (\langle \cdot \rangle_-) \text{ where }$
 $\langle \sigma \rangle_a = (\lceil \sigma \rceil_s \uparrow skip-ra \ a)$

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.

definition $rassume :: 'a \text{ upred} \Rightarrow 'a \text{ hrel } ([\cdot]^\top)$ **where**
 $[urel-defs]: rassume c = II \triangleleft c \triangleright_r false$

definition $rasassert :: 'a \text{ upred} \Rightarrow 'a \text{ hrel } (\{\cdot\}_\perp)$ **where**
 $[urel-defs]: rasassert c = II \triangleleft c \triangleright_r true$

A test is like a precondition, except that it identifies to the postcondition, and is thus a refinement of II . It forms the basis for Kleene Algebra with Tests [16, 1] (KAT), which embeds a Boolean algebra into a Kleene algebra to represent conditions.

definition $lift-test :: 'a \text{ cond} \Rightarrow 'a \text{ hrel } ([\cdot]_t)$
where $[urel-defs]: [b]_t = ([b]_{<} \wedge II)$

We define two variants of while loops based on strongest and weakest fixed points. The former is *false* for an infinite loop, and the latter is *true*.

definition $while :: 'a \text{ cond} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel } (while^\top - do - od)$ **where**
 $[urel-defs]: while^\top b \text{ do } P \text{ od} = (\nu X \cdot (P ;; X) \triangleleft b \triangleright_r II)$

abbreviation $while-top :: 'a \text{ cond} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel } (while - do - od)$ **where**
 $while b \text{ do } P \text{ od} \equiv while^\top b \text{ do } P \text{ od}$

definition $while-bot :: 'a \text{ cond} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel } (while_\perp - do - od)$ **where**
 $[urel-defs]: while_\perp b \text{ do } P \text{ od} = (\mu X \cdot (P ;; X) \triangleleft b \triangleright_r II)$

While loops with invariant decoration (cf. [1]) – partial correctness.

definition $while-inv :: 'a \text{ cond} \Rightarrow 'a \text{ cond} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel } (while - invr - do - od)$ **where**
 $[urel-defs]: while b \text{ invr } p \text{ do } S \text{ od} = while b \text{ do } S \text{ od}$

While loops with invariant decoration – total correctness.

definition $while-inv-bot :: 'a \text{ cond} \Rightarrow 'a \text{ cond} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel } (while_\perp - invr - do - od)$ **where**
 $[urel-defs]: while_\perp b \text{ invr } p \text{ do } S \text{ od} = while_\perp b \text{ do } S \text{ od}$

While loops with invariant and variant decorations – total correctness.

definition $while-vrt :: 'a \text{ cond} \Rightarrow 'a \text{ cond} \Rightarrow (nat, 'a) \text{ uexpr} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel } (while - invr - vrt - do - od)$ **where**
 $[urel-defs]: while b \text{ invr } p \text{ vrt } v \text{ do } S \text{ od} = while_\perp b \text{ do } S \text{ od}$

We implement a poor man's version of alphabet restriction that hides a variable within a relation.

definition $rel-var-res :: 'a \text{ hrel} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \text{ hrel } (\text{infix } \downarrow_\alpha 80)$ **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 $rel-aext :: 'b \text{ hrel} \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a \text{ hrel}$
where $[upred-defs]: rel-aext P a = P \oplus_p (a \times_L a)$

definition $rel-ares :: 'a \text{ hrel} \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b \text{ hrel}$
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 [17].

definition *frame* :: ('a \Rightarrow 'α) \Rightarrow 'α hrel \Rightarrow 'α hrel **where**
[urel-defs]: *frame* a P = (P \wedge \$v' =_u \$v \oplus \$v' on &a)

definition *antiframe* :: ('a \Rightarrow 'α) \Rightarrow 'α hrel \Rightarrow 'α hrel **where**
[urel-defs]: *antiframe* a P = (P \wedge \$v' =_u \$v' \oplus \$v on &a)

Frame extension combines alphabet extension with the frame operator to both add additional variables and then frame those.

definition *rel-frex* :: ('β \Rightarrow 'α) \Rightarrow 'β hrel \Rightarrow 'α hrel **where**
[upred-defs]: *rel-frex* 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 'α) \Rightarrow 'α hrel \Rightarrow 'α hrel **where**
[urel-defs]: *nameset* a P = (P \upharpoonright_v {\$v, \$a'})

15.3 Syntax Translations

syntax

- Alternative traditional conditional syntax
- utp-if :: logic \Rightarrow logic \Rightarrow logic \Rightarrow logic ((if_u (-)/ then (-)/ else (-)) [0, 0, 71] 71)
- Iterated sequential composition
- seqr-iter :: ptrn \Rightarrow 'a list \Rightarrow 'σ hrel \Rightarrow 'σ hrel ((3;; - : - · / -) [0, 0, 10] 10)
- Single and multiple assignement
- assignment :: svids \Rightarrow uexprs \Rightarrow 'α hrel ('(-') := '(-'))
- assignment :: svids \Rightarrow uexprs \Rightarrow 'α hrel (**infixr** := 72)
- Indexed assignment
- assignment-upd :: svid \Rightarrow logic \Rightarrow logic \Rightarrow logic (([-] := / -) [73, 0, 0] 72)
- Substitution constructor
- mk-usubst :: svids \Rightarrow uexprs \Rightarrow 'α usubst
- Alphabetised skip
- skip-ra :: salpha \Rightarrow logic (II.)
- Frame
- frame :: salpha \Rightarrow logic \Rightarrow logic (:-[-] [99,0] 100)
- Antiframe
- antiframe :: salpha \Rightarrow logic \Rightarrow logic (:-[-] [79,0] 80)
- Relational Alphabet Extension
- rel-aext :: logic \Rightarrow salpha \Rightarrow logic (**infixl** \oplus_r 90)
- Relational Alphabet Restriction
- rel-ares :: logic \Rightarrow salpha \Rightarrow logic (**infixl** \upharpoonright_r 90)
- Frame Extension
- rel-frex :: salpha \Rightarrow logic \Rightarrow logic (:-[-]⁺ [99,0] 100)
- Nameset
- nameset :: salpha \Rightarrow logic \Rightarrow logic (ns - · - [0,999] 999)

translations

- utp-if b P Q \Rightarrow P \triangleleft b \triangleright_r Q
- ;; x : l · P \Rightarrow (CONST *seqr-iter*) l (λx. P)
- mk-usubst σ (-svid-unit x) v \Rightarrow σ(&x \mapsto_s v)
- mk-usubst σ (-svid-list x xs) (-uexprs v vs) \Rightarrow (-mk-usubst (σ(&x \mapsto_s v)) xs vs)
- assignment xs vs \Rightarrow CONST *uassigns* (-mk-usubst (CONST id) xs vs)
- assignment x v \Leftarrow CONST *uassigns* (CONST *subst-upd* (CONST id) x v)
- assignment x v \Leftarrow -assignment (-spvar x) v
- x,y := u,v \Leftarrow CONST *uassigns* (CONST *subst-upd* (CONST *subst-upd* (CONST id) (CONST svar x) u) (CONST svar y) v)

— Indexed assignment uses the overloaded collection update function *uupd*.

```

 $x[k] := v \Rightarrow x := \&x(k \mapsto v)_u$ 
 $\text{-skip-ra } v \Rightarrow \text{CONST skip-ra } v$ 
 $\text{-frame } x P \Rightarrow \text{CONST frame } x P$ 
 $\text{-frame } (-\text{salphaset } (-\text{salphamk } x)) P \Leftarrow \text{CONST frame } x P$ 
 $\text{-antiframe } x P \Rightarrow \text{CONST antiframe } x P$ 
 $\text{-antiframe } (-\text{salphaset } (-\text{salphamk } x)) P \Leftarrow \text{CONST antiframe } x P$ 
 $\text{-nameset } x P == \text{CONST nameset } x P$ 
 $\text{-rel-aext } P a == \text{CONST rel-aext } P a$ 
 $\text{-rel-ares } P a == \text{CONST rel-ares } P a$ 
 $\text{-rel-frext } a P == \text{CONST rel-frext } a P$ 

```

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') *uexpr* type, determine that it is relational (product alphabet), and then checks if the types *alpha* and *beta* are the same. If they are, the type is printed as a *hexpr*. Otherwise, we have no match. We then set up a regular translation for the *hrel* type that uses this.

```

print-translation <<
let
fun tr' ctx [ a
  , Const (@{type-syntax prod},-) $ alpha $ beta ] =
  if (alpha = beta)
  then Syntax.const @{type-syntax hexpr} $ a $ alpha
  else raise Match;
in [(@{type-syntax uexpr},tr')]
end
>>

```

translations

```

(type) ' $\alpha$  hrel <= (type) (bool, ' $\alpha$ ) 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* :: $(\alpha, \alpha') \text{ urel} \Rightarrow \text{bool}$
where [*urel-defs*]: *ufunctional* *R* $\longleftrightarrow II \sqsubseteq R^- \;; R$

definition *uinj* :: $(\alpha, \alpha') \text{ urel} \Rightarrow \text{bool}$
where [*urel-defs*]: *uinj* *R* $\longleftrightarrow II \sqsubseteq R \;; R^-$

definition *Dom* :: $\alpha' \text{ hrel} \Rightarrow \alpha \text{ upred}$
where [*upred-defs*]: *Dom* *P* = $[\exists \mathbf{v}' \cdot P]_<$

definition *Ran* :: $\alpha \text{ hrel} \Rightarrow \alpha' \text{ upred}$
where [*upred-defs*]: *Ran* *P* = $[\exists \mathbf{v} \cdot P]_>$

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

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

15.5 Introduction laws

lemma *urel-refine-ext*:

$\llbracket \bigwedge s s'. P[\llbracket \langle s \rangle, \langle s' \rangle / \$v, \$v' \rrbracket] \sqsubseteq Q[\llbracket \langle s \rangle, \langle s' \rangle / \$v, \$v' \rrbracket] \rrbracket \implies P \sqsubseteq Q$
by (*rel-auto*)

lemma *urel-eq-ext*:

$\llbracket \bigwedge s s'. P[\llbracket \langle s \rangle, \langle s' \rangle / \$v, \$v' \rrbracket] = Q[\llbracket \langle s \rangle, \langle s' \rangle / \$v, \$v' \rrbracket] \rrbracket \implies P = Q$
by (*rel-auto*)

15.6 Unrestriction Laws

lemma *unrest-iuvar* [*unrest*]: $out\alpha \# \$x$

by (*metis fst-snd-lens-indep lift-pre-var out α -def unrest-aext-indep*)

lemma *unrest-ouvar* [*unrest*]: $in\alpha \# \$x'$

by (*metis in α -def lift-post-var snd-fst-lens-indep unrest-aext-indep*)

lemma *unrest-semir-undash* [*unrest*]:

fixes $x :: ('a \implies 'a)$
assumes $\$x \# P$
shows $\$x \# P ;; Q$
using *assms* **by** (*rel-auto*)

lemma *unrest-semir-dash* [*unrest*]:

fixes $x :: ('a \implies 'a)$
assumes $\$x' \# Q$
shows $\$x' \# P ;; Q$
using *assms* **by** (*rel-auto*)

lemma *unrest-cond* [*unrest*]:

$\llbracket x \# P; x \# b; x \# Q \rrbracket \implies x \# P \triangleleft b \triangleright Q$
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*]:

$\llbracket mwb\text{-}lens\ x; in\alpha \# (P :: ('a, ('a \times 'b))\ uexpr) \rrbracket \implies \$x \# P$
by (*rel-auto*)

lemma *unrest-out α -var* [*unrest*]:

$\llbracket mwb\text{-}lens\ x; out\alpha \# (P :: ('a, ('a \times 'b))\ uexpr) \rrbracket \implies \$x' \# P$
by (*rel-auto*)

lemma *unrest-pre-out α* [*unrest*]: $out\alpha \# [b]_<$

by (*transfer, auto simp add: out α -def*)

lemma *unrest-post-in α* [*unrest*]: $in\alpha \# [b]_>$

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]:

$x \# p1 \implies \$x' \# \lceil p1 \rceil_>$
by (*transfer*, *simp*)

lemma *unrest-convr-out α* [unrest]:

$in\alpha \# p \implies out\alpha \# p^-$
by (*transfer*, *auto simp add: lens-defs*)

lemma *unrest-convr-in α* [unrest]:

$out\alpha \# p \implies in\alpha \# p^-$
by (*transfer*, *auto simp add: lens-defs*)

lemma *unrest-in-rel-var-res* [unrest]:

$vwb\text{-}lens\ x \implies \$x \# (P \upharpoonright_\alpha x)$
by (*simp add: rel-var-res-def unrest*)

lemma *unrest-out-rel-var-res* [unrest]:

$vwb\text{-}lens\ x \implies \$x' \# (P \upharpoonright_\alpha x)$
by (*simp add: rel-var-res-def unrest*)

lemma *unrest-out-alpha-usubst-rel-lift* [unrest]:

$out\alpha \# \lceil \sigma \rceil_s$
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-seq* [alpha]:

$weak\text{-}lens\ a \implies (P ;; Q) \oplus_r a = (P \oplus_r a ;; Q \oplus_r a)$
apply (*rel-auto*)
apply (*rename-tac aa b y*)
apply (*rule-tac x=create_a y in exI*)
apply (*simp*)
done

lemma *rel-aext-cond* [alpha]:

$(P \triangleleft b \triangleright_r Q) \oplus_r a = (P \oplus_r a \triangleleft b \oplus_p a \triangleright_r Q \oplus_r a)$
by (*rel-auto*)

15.7 Substitution laws

lemma *subst-seq-left* [usubst]:

$out\alpha \# \sigma \implies \sigma \dagger (P ;; Q) = (\sigma \dagger P) ;; Q$
by (*rel-simp*, (*metis (no-types, lifting) Pair-inject surjective-pairing*)+)

lemma *subst-seq-right* [usubst]:

$in\alpha \# \sigma \implies \sigma \dagger (P ;; Q) = P ;; (\sigma \dagger Q)$
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 $x :: (bool \implies 'a)$

shows

$\bigwedge P Q \sigma. \sigma(\$x \mapsto_s true) \dagger (P ;; Q) = \sigma \dagger (P \llbracket true/\$x \rrbracket ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x \mapsto_s false) \dagger (P ;; Q) = \sigma \dagger (P \llbracket false/\$x \rrbracket ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s true) \dagger (P ;; Q) = \sigma \dagger (P ;; Q \llbracket true/\$x' \rrbracket)$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s false) \dagger (P ;; Q) = \sigma \dagger (P ;; Q \llbracket false/\$x' \rrbracket)$
by (rel-auto)+

lemma *zero-one-seqr-laws* [usubst]:

fixes $x :: (- \implies 'a)$

shows

$\bigwedge P Q \sigma. \sigma(\$x \mapsto_s 0) \dagger (P ;; Q) = \sigma \dagger (P \llbracket 0/\$x \rrbracket ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x \mapsto_s 1) \dagger (P ;; Q) = \sigma \dagger (P \llbracket 1/\$x \rrbracket ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s 0) \dagger (P ;; Q) = \sigma \dagger (P ;; Q \llbracket 0/\$x' \rrbracket)$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s 1) \dagger (P ;; Q) = \sigma \dagger (P ;; Q \llbracket 1/\$x' \rrbracket)$
by (rel-auto)+

lemma *numeral-seqr-laws* [usubst]:

fixes $x :: (- \implies 'a)$

shows

$\bigwedge P Q \sigma. \sigma(\$x \mapsto_s numeral\ n) \dagger (P ;; Q) = \sigma \dagger (P \llbracket numeral\ n/\$x \rrbracket ;; Q)$
 $\bigwedge P Q \sigma. \sigma(\$x' \mapsto_s numeral\ n) \dagger (P ;; Q) = \sigma \dagger (P ;; Q \llbracket numeral\ n/\$x' \rrbracket)$
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]:

$out\alpha \# \sigma \implies \sigma \dagger II = \langle \llbracket \sigma \rrbracket_s \rangle_a$

by (rel-simp, (metis (mono-tags, lifting) prod.sel(1) sndI surjective-pairing)+)

lemma *subst-pre-skip* [usubst]: $\llbracket \sigma \rrbracket_s \dagger II = \langle \sigma \rangle_a$

by (rel-auto)

lemma *subst-rel-lift-seq* [usubst]:

$\llbracket \sigma \rrbracket_s \dagger (P ;; Q) = (\llbracket \sigma \rrbracket_s \dagger P) ;; Q$

by (rel-auto)

lemma *subst-rel-lift-comp* [usubst]:

$\llbracket \sigma \rrbracket_s \circ \llbracket \varrho \rrbracket_s = \llbracket \sigma \circ \varrho \rrbracket_s$

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 $x :: ('a \implies 'b)$

shows $\llbracket \sigma(x \mapsto_s v) \rrbracket_s = \llbracket \sigma \rrbracket_s(\$x \mapsto_s \llbracket v \rrbracket_<)$

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 $x :: ('a \Rightarrow 'a)$

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]: $\lfloor \sigma \rfloor_s \dagger \lfloor b \rfloor_< = \lfloor \sigma \dagger b \rfloor_<$

by (metis apply-subst-ext fst-vwb-lens in α -def)

lemma unrest-usubst-lift-in [unrest]:

$x \# P \Rightarrow \$x \# \lfloor P \rfloor_s$

by pred-simp

lemma unrest-usubst-lift-out [unrest]:

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

shows $\$x' \# \lfloor P \rfloor_s$

by pred-simp

lemma subst-lift-cond [usubst]: $\lfloor \sigma \rfloor_s \dagger \lfloor s \rfloor_{\leftarrow} = \lfloor \sigma \dagger s \rfloor_{\leftarrow}$

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]:

$wb\text{-}lens\ a \Rightarrow ((P ;; Q) \oplus_p (a \times_L a)) = ((P \oplus_p (a \times_L a)) ;; (Q \oplus_p (a \times_L a)))$

by (rel-simp, metis wb-lens-weak weak-lens.put-get)

lemma rcond-lift-true [simp]:

$\lfloor true \rfloor_{\leftarrow} = true$

by rel-auto

lemma rcond-lift-false [simp]:

$\lfloor false \rfloor_{\leftarrow} = false$

by rel-auto

lemma rel-ares-aext [alpha]:

$vwb\text{-}lens\ a \Rightarrow (P \oplus_r a) \upharpoonright_r a = P$

by (rel-auto)

lemma rel-aext-ares [alpha]:

$\{\$a, \$a'\} \Vdash P \Rightarrow P \upharpoonright_r a \oplus_r a = P$

by (rel-auto)

lemma rel-aext-uses [unrest]:

$vwb\text{-}lens\ a \Rightarrow \{\$a, \$a'\} \Vdash (P \oplus_r a)$

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 $RID :: ('a \implies 'α) \Rightarrow 'α \text{ hrel} \Rightarrow 'α \text{ hrel}$
where $RID\ x\ P = ((\exists\ \$x \cdot \exists\ \$x' \cdot P) \wedge \$x' =_u \$x)$

declare $RID\text{-}def\ [\text{urel-defs}]$

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 $(rel\text{-}auto)$
apply $(metis\ vwb\text{-}lens.put\text{-}eq)$
apply $(metis\ vwb\text{-}lens\text{-}wb\ wb\text{-}lens.get\text{-}put\ wb\text{-}lens\text{-}weak\ weak\text{-}lens.put\text{-}get)$
done

lemma $RID2$: $vwb\text{-}lens\ x \implies x := \ll v \gg ;; RID(x)(P) = RID(x)(P) ;; x := \ll v \gg$
apply $(rel\text{-}auto)$
apply $(metis\ mwb\text{-}lens.put\text{-}put\ vwb\text{-}lens\text{-}mwb\ vwb\text{-}lens\text{-}wb\ wb\text{-}lens.get\text{-}put\ wb\text{-}lens\text{-}def\ weak\text{-}lens.put\text{-}get)$
apply $blast$
done

lemma $RID\text{-}assign\text{-}commute$:
 $vwb\text{-}lens\ x \implies P = RID(x)(P) \longleftrightarrow (\forall\ v. x := \ll v \gg ;; P = P ;; x := \ll v \gg)$
by $(metis\ RID1\ RID2)$

lemma $RID\text{-}idem$:
 $mwb\text{-}lens\ x \implies RID(x)(RID(x)(P)) = RID(x)(P)$
by $(rel\text{-}auto)$

lemma $RID\text{-}mono$:
 $P \sqsubseteq Q \implies RID(x)(P) \sqsubseteq RID(x)(Q)$
by $(rel\text{-}auto)$

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

lemma $RID\text{-}skip\text{-}r$:
 $vwb\text{-}lens\ x \implies RID(x)(II) = II$
apply $(rel\text{-}auto)$ **using** $vwb\text{-}lens.put\text{-}eq$ **by** $fastforce$

lemma $skip\text{-}r\text{-}RID\ [closure]$: $vwb\text{-}lens\ x \implies II\ is\ RID(x)$
by $(simp\ add:\ Healthy\text{-}def\ RID\text{-}skip\text{-}r)$

lemma $RID\text{-}disj$:
 $RID(x)(P \vee Q) = (RID(x)(P) \vee RID(x)(Q))$
by $(rel\text{-}auto)$

lemma $disj\text{-}RID\ [closure]$: $\ll P\ is\ RID(x); Q\ is\ RID(x) \gg \implies (P \vee Q)\ is\ RID(x)$
by $(simp\ add:\ Healthy\text{-}def\ RID\text{-}disj)$

lemma $RID\text{-}conj$:
 $vwb\text{-}lens\ x \implies RID(x)(RID(x)(P) \wedge RID(x)(Q)) = (RID(x)(P) \wedge RID(x)(Q))$
by $(rel\text{-}auto)$

lemma *conj-RID* [closure]: $\llbracket \text{vwb-lens } x; P \text{ is RID}(x); Q \text{ is RID}(x) \rrbracket \implies (P \wedge Q) \text{ is RID}(x)$
 by (metis Healthy-if Healthy-intro RID-conj)

lemma *RID-assigns-r-diff*:
 $\llbracket \text{vwb-lens } x; x \# \sigma \rrbracket \implies \text{RID}(x)(\langle \sigma \rangle_a) = \langle \sigma \rangle_a$
 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]: $\llbracket \text{vwb-lens } x; x \# \sigma \rrbracket \implies \langle \sigma \rangle_a \text{ is RID}(x)$
 by (simp add: Healthy-def RID-assigns-r-diff)

lemma *RID-assign-r-same*:
 $\text{vwb-lens } x \implies \text{RID}(x)(x := v) = \text{ID}$
 apply (rel-auto)
 using vwb-lens.put-eq apply fastforce
 done

lemma *RID-seq-left*:
 assumes vwb-lens x
 shows $\text{RID}(x)(\text{RID}(x)(P) ;; Q) = (\text{RID}(x)(P) ;; \text{RID}(x)(Q))$
proof –
 have $\text{RID}(x)(\text{RID}(x)(P) ;; Q) = ((\exists \$x \cdot \exists \$x' \cdot ((\exists \$x \cdot \exists \$x' \cdot P) \wedge \$x' =_u \$x) ;; Q) \wedge \$x' =_u \$x)$
 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 mw-lens.put-put vwb-lens-mw)
 done
 also from assms have $\dots = (((\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) mw-lens.put-put vwb-lens-def wb-lens-weak weak-lens.put-get)
 also have $\dots = (((\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 $\dots = (((\exists \$x \cdot \exists \$x' \cdot P) \wedge \$x' =_u \$x) ;; ((\exists \$x \cdot \exists \$x' \cdot Q) \wedge \$x' =_u \$x))$
 by (rel-auto)
 also have $\dots = (\text{RID}(x)(P) ;; \text{RID}(x)(Q))$
 by (rel-auto)
 finally show ?thesis .
qed

lemma *RID-seq-right*:
 assumes vwb-lens x
 shows $\text{RID}(x)(P ;; \text{RID}(x)(Q)) = (\text{RID}(x)(P) ;; \text{RID}(x)(Q))$
proof –
 have $\text{RID}(x)(P ;; \text{RID}(x)(Q)) = ((\exists \$x \cdot \exists \$x' \cdot P ;; ((\exists \$x \cdot \exists \$x' \cdot Q) \wedge \$x' =_u \$x)) \wedge \$x' =_u \$x)$
 by (simp add: RID-def usubst)
 also from assms have $\dots = (((\exists \$x \cdot P) ;; (\exists \$x \cdot \exists \$x' \cdot Q) \wedge (\exists \$x' \cdot \$x' =_u \$x)) \wedge \$x' =_u \$x)$
 by (rel-auto)

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 ... = ((($\exists x \cdot \exists x' \cdot P$) $\wedge x' =_u x$) ;; (($\exists x \cdot \exists x' \cdot Q$) $\wedge x' =_u x$)) $\wedge x' =_u x$
 \$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 :: ($'a \implies 'a$) \Rightarrow $'a \ hrel \Rightarrow bool$ (infix $\#\#$ 20)
 where $(x \#\# P) \longleftrightarrow (P\ is\ RID(x))$

declare unrest-relation-def [urel-defs]

lemma runrest-assign-commute:
 $\llbracket vwb\text{-}lens\ x; x \#\# P \rrbracket \implies x := \langle v \rangle ;; P = P ;; x := \langle v \rangle$
 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]:
 $vwb\text{-}lens\ x \implies x \#\# I$
 by (simp add: unrest-relation-def closure)

lemma assigns-r-runrest:
 $\llbracket vwb\text{-}lens\ x; x \# \sigma \rrbracket \implies x \#\# \langle \sigma \rangle_a$
 by (simp add: unrest-relation-def closure)

lemma seq-r-runrest [unrest]:
 assumes $vwb\text{-}lens\ x\ x \#\# P\ x \#\# Q$
 shows $x \#\# (P ;; Q)$
 using *assms* by (simp add: unrest-relation-def closure)

```

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

lemma and-runrest [unrest]:  $\llbracket x \# \# P; x \# \# Q \rrbracket \implies x \# \# (P \wedge Q)$ 
  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 [14].

```

type-synonym 'a chain = nat  $\Rightarrow$  'a upred

```

```

definition chain :: 'a chain  $\Rightarrow$  bool where
  chain Y =  $((Y\ 0 = \text{false}) \wedge (\forall i. Y\ (Suc\ i) \sqsubseteq Y\ i))$ 

```

```

lemma chain0 [simp]: chain Y  $\implies Y\ 0 = \text{false}$ 
  by (simp add: chain-def)

```

```

lemma chainI:
  assumes  $Y\ 0 = \text{false} \wedge i. Y\ (\text{Suc}\ i) \sqsubseteq Y\ i$ 
  shows chain  $Y$ 
  using assms by (auto simp add: chain-def)

lemma chainE:
  assumes chain  $Y \wedge i. \llbracket Y\ 0 = \text{false}; Y\ (\text{Suc}\ i) \sqsubseteq Y\ i \rrbracket \implies P$ 
  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  $F\ E \longleftrightarrow \text{chain}\ E \wedge (\forall X\ n. ((F(X) \wedge E(n+1)) = (F(X \wedge E(n)) \wedge E(n+1))))$ 

```

```

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

```



```

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
}
thus ?*thesis*
by *pred-simp*
qed

The next lemma shows that using substitution also work. However it is not that generic nor practical for proof automation ...

lemma *refine-ustbst-to-ueq*:
 $vwb\text{-}lens\ E \implies (Pre \Rightarrow Post) \ll [st']_{>=E} \sqsubseteq f \ll [st']_{>=E} = (((Pre \wedge \$E =_u \ll st' \gg) \Rightarrow Post) \sqsubseteq f)$
by (*rel-auto, metis vwb-lens-wb wb-lens.get-put*)

By instantiation of $\ll [fp\ ?B = ?B\ (?fp\ ?B); wf\ ?R; \bigwedge f\ st.\ [\bigwedge st'. (st', st) \in ?R \implies (?Pre \wedge [?e]_{<=u} \ll st' \gg \Rightarrow ?Post) \sqsubseteq f; ?fp\ ?B = f] \implies (?Pre \wedge [?e]_{<=u} \ll st \gg \Rightarrow ?Post) \sqsubseteq ?B\ f] \implies (?Pre \Rightarrow ?Post) \sqsubseteq ?fp\ ?B$ with μ and lifting of the well-founded relation we have ...

lemma *mu-rec-total-pure-rule*:
assumes WF : *wf R*
and M : *mono B*
and *induct-step*:
 $\bigwedge f\ st.\ [((Pre \wedge ([e]_{<=u} \ll st \gg)_u \in_u \ll R \gg \Rightarrow Post) \sqsubseteq f)]$
 $\implies \mu\ B = f \implies (Pre \wedge [e]_{<=u} \ll st \gg \Rightarrow Post) \sqsubseteq (B\ f)$
shows $(Pre \Rightarrow Post) \sqsubseteq \mu\ B$
proof (*rule wf-fixp-uinduct-pure-ueq-gen* [**where** $fp=\mu$ **and** $Pre=Pre$ **and** $B=B$ **and** $R=R$ **and** $e=e$])
show $\mu\ B = B\ (\mu\ B)$

by (simp add: M def-gfp-unfold)
 show wf R
 by (fact WF)
 show $\bigwedge f st. (\bigwedge st'. (st', st) \in R \implies (Pre \wedge [e]_{<} =_u \ll st' \gg \Rightarrow Post) \sqsubseteq f) \implies$
 $\mu B = f \implies$
 $(Pre \wedge [e]_{<} =_u \ll st \gg \Rightarrow Post) \sqsubseteq B f$
 by (rule induct-step, rel-simp, simp)
 qed

lemma nu-rec-total-pure-rule:

assumes WF: wf R
 and M: mono B
 and induct-step:
 $\bigwedge f st. \llbracket (Pre \wedge ([e]_{<}, \ll st \gg)_u \in_u \ll R \gg \Rightarrow Post) \sqsubseteq f \rrbracket$
 $\implies \nu B = f \implies (Pre \wedge [e]_{<} =_u \ll st \gg \Rightarrow Post) \sqsubseteq (B f)$
 shows $(Pre \Rightarrow Post) \sqsubseteq \nu B$
proof (rule wf-fixp-uinduct-pure-ueq-gen[where fp= ν and Pre=Pre and B=B and R=R and e=e])
 show $\nu B = B (\nu B)$
 by (simp add: M def-lfp-unfold)
 show wf R
 by (fact WF)
 show $\bigwedge f st. (\bigwedge st'. (st', st) \in R \implies (Pre \wedge [e]_{<} =_u \ll st' \gg \Rightarrow Post) \sqsubseteq f) \implies$
 $\nu B = f \implies$
 $(Pre \wedge [e]_{<} =_u \ll st \gg \Rightarrow Post) \sqsubseteq B f$
 by (rule induct-step, rel-simp, simp)
 qed

Since $B (Pre \wedge ([E]_{<}, \ll st \gg)_u \in_u \ll R \gg \Rightarrow Post) \sqsubseteq B (\mu B)$ and *mono* B, thus, $\llbracket wf ?R; Monotonic ?B; \bigwedge f st. \llbracket (?Pre \wedge ([?e]_{<}, \ll st \gg)_u \in_u \ll ?R \gg \Rightarrow ?Post) \sqsubseteq f; \mu ?B = f \rrbracket \implies (?Pre \wedge [?e]_{<} =_u \ll st \gg \Rightarrow ?Post) \sqsubseteq ?B f \rrbracket \implies (?Pre \Rightarrow ?Post) \sqsubseteq \mu ?B$ can be expressed as follows

lemma mu-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 $(Pre \Rightarrow Post) \sqsubseteq \mu B$
proof (rule mu-rec-total-pure-rule[where R=R and e=e], simp-all add: assms)
 show $\bigwedge f st. (Pre \wedge ([e]_{<}, \ll st \gg)_u \in_u \ll R \gg \Rightarrow Post) \sqsubseteq f \implies \mu B = f \implies (Pre \wedge [e]_{<} =_u \ll st \gg \Rightarrow$
 $Post) \sqsubseteq B f$
 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 $\bigwedge f st. (Pre \wedge ([e]_{<}, \ll st \gg)_u \in_u \ll R \gg \Rightarrow Post) \sqsubseteq f \implies \nu B = f \implies (Pre \wedge [e]_{<} =_u \ll st \gg \Rightarrow$
 $Post) \sqsubseteq B f$
 by (simp add: M induct-step monoD order-subst2)
 qed

end

17 UTP Deduction Tactic

theory *utp-deduct*
imports *utp-pred*
begin

named-theorems *uintro*
named-theorems *uelim*
named-theorems *udest*

lemma *uttrueI* [*uintro*]: $\llbracket \text{true} \rrbracket_e b$
by (*pred-auto*)

lemma *uopI* [*uintro*]: $f (\llbracket x \rrbracket_e b) \Longrightarrow \llbracket uop\ f\ x \rrbracket_e b$
by (*pred-auto*)

lemma *bopI* [*uintro*]: $f (\llbracket x \rrbracket_e b) (\llbracket y \rrbracket_e b) \Longrightarrow \llbracket bop\ f\ x\ y \rrbracket_e b$
by (*pred-auto*)

lemma *tropI* [*uintro*]: $f (\llbracket x \rrbracket_e b) (\llbracket y \rrbracket_e b) (\llbracket z \rrbracket_e b) \Longrightarrow \llbracket trop\ f\ x\ y\ z \rrbracket_e b$
by (*pred-auto*)

lemma *uconjI* [*uintro*]: $\llbracket \llbracket p \rrbracket_e b; \llbracket q \rrbracket_e b \rrbracket \Longrightarrow \llbracket p \wedge q \rrbracket_e b$
by (*pred-auto*)

lemma *uconjE* [*uelim*]: $\llbracket \llbracket p \wedge q \rrbracket_e b; \llbracket \llbracket p \rrbracket_e b ; \llbracket q \rrbracket_e b \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$
by (*pred-auto*)

lemma *uimpI* [*uintro*]: $\llbracket \llbracket p \rrbracket_e b \Longrightarrow \llbracket q \rrbracket_e b \rrbracket \Longrightarrow \llbracket p \Rightarrow q \rrbracket_e b$
by (*pred-auto*)

lemma *uimpE* [*elim*]: $\llbracket \llbracket p \Rightarrow q \rrbracket_e b; (\llbracket p \rrbracket_e b \Longrightarrow \llbracket q \rrbracket_e b) \Longrightarrow P \rrbracket \Longrightarrow P$
by (*pred-auto*)

lemma *ushAllI* [*uintro*]: $\llbracket \bigwedge x. \llbracket p(x) \rrbracket_e b \rrbracket \Longrightarrow \llbracket \forall x. p(x) \rrbracket_e b$
by *pred-auto*

lemma *ushExI* [*uintro*]: $\llbracket \llbracket p(x) \rrbracket_e b \rrbracket \Longrightarrow \llbracket \exists x. p(x) \rrbracket_e b$
by *pred-auto*

lemma *udeduct-trueI* [*uintro*]: $\llbracket \bigwedge b. \llbracket p \rrbracket_e b \rrbracket \Longrightarrow 'p'$
using *taut.rep-eq* **by** *blast*

lemma *udeduct-refineI* [*uintro*]: $\llbracket \bigwedge b. \llbracket q \rrbracket_e b \Longrightarrow \llbracket p \rrbracket_e b \rrbracket \Longrightarrow p \sqsubseteq q$
by *pred-auto*

lemma *udeduct-eqI* [*uintro*]: $\llbracket \bigwedge b. \llbracket p \rrbracket_e b \Longrightarrow \llbracket q \rrbracket_e b; \bigwedge b. \llbracket q \rrbracket_e b \Longrightarrow \llbracket p \rrbracket_e b \rrbracket \Longrightarrow p = q$
by (*pred-auto*)

Some of the following lemmas help backward reasoning with bindings

lemma *conj-implies*: $\llbracket \llbracket P \wedge Q \rrbracket_e b \rrbracket \Longrightarrow \llbracket P \rrbracket_e b \wedge \llbracket Q \rrbracket_e b$
by *pred-auto*

lemma *conj-implies2*: $\llbracket \llbracket P \rrbracket_e b \wedge \llbracket Q \rrbracket_e b \rrbracket \Longrightarrow \llbracket P \wedge Q \rrbracket_e b$
by *pred-auto*

lemma *disj-eq*: $\llbracket [P]_e b \vee [Q]_e b \rrbracket \Longrightarrow \llbracket P \vee Q \rrbracket_e b$
by *pred-auto*

lemma *disj-eq2*: $\llbracket [P \vee Q]_e b \rrbracket \Longrightarrow \llbracket P \rrbracket_e b \vee \llbracket Q \rrbracket_e b$
by *pred-auto*

lemma *conj-eq-subst*: $(\llbracket P \wedge Q \rrbracket_e b \wedge \llbracket P \rrbracket_e b = \llbracket R \rrbracket_e b) = (\llbracket R \wedge Q \rrbracket_e b \wedge \llbracket P \rrbracket_e b = \llbracket R \rrbracket_e b)$
by *pred-auto*

lemma *conj-imp-subst*: $(\llbracket P \wedge Q \rrbracket_e b \wedge (\llbracket Q \rrbracket_e b \longrightarrow (\llbracket P \rrbracket_e b = \llbracket R \rrbracket_e b))) = (\llbracket R \wedge Q \rrbracket_e b \wedge (\llbracket Q \rrbracket_e b \longrightarrow (\llbracket P \rrbracket_e b = \llbracket R \rrbracket_e b)))$
by *pred-auto*

lemma *disj-imp-subst*: $(\llbracket Q \wedge (P \vee S) \rrbracket_e b \wedge (\llbracket Q \rrbracket_e b \longrightarrow (\llbracket P \rrbracket_e b = \llbracket R \rrbracket_e b))) = (\llbracket Q \wedge (R \vee S) \rrbracket_e b \wedge (\llbracket Q \rrbracket_e b \longrightarrow (\llbracket P \rrbracket_e b = \llbracket R \rrbracket_e b)))$
by *pred-auto*

Simplifications on value equality

lemma *uexpr-eq*: $(\llbracket e_0 \rrbracket_e b = \llbracket e_1 \rrbracket_e b) = \llbracket e_0 =_u e_1 \rrbracket_e b$
by *pred-auto*

lemma *uexpr-trans*: $(\llbracket P \wedge e_0 =_u e_1 \rrbracket_e b \wedge (\llbracket P \rrbracket_e b \longrightarrow \llbracket e_1 =_u e_2 \rrbracket_e b)) = (\llbracket P \wedge e_0 =_u e_2 \rrbracket_e b \wedge (\llbracket P \rrbracket_e b \longrightarrow \llbracket e_1 =_u e_2 \rrbracket_e b))$
by (*pred-auto*)

lemma *uexpr-trans2*: $(\llbracket P \wedge Q \wedge e_0 =_u e_1 \rrbracket_e b \wedge (\llbracket Q \rrbracket_e b \longrightarrow \llbracket e_1 =_u e_2 \rrbracket_e b)) = (\llbracket P \wedge Q \wedge e_0 =_u e_2 \rrbracket_e b \wedge (\llbracket Q \rrbracket_e b \longrightarrow \llbracket e_1 =_u e_2 \rrbracket_e b))$
by (*pred-auto*)

lemma *uequality*: $\llbracket (\llbracket R \rrbracket_e b = \llbracket Q \rrbracket_e b) \rrbracket \Longrightarrow \llbracket P \wedge R \rrbracket_e b = \llbracket P \wedge Q \rrbracket_e b$
by *pred-auto*

lemma *ueqe1*: $\llbracket [P]_e b \Longrightarrow (\llbracket R \rrbracket_e b = \llbracket Q \rrbracket_e b) \rrbracket \Longrightarrow (\llbracket P \wedge R \rrbracket_e b \Longrightarrow \llbracket P \wedge Q \rrbracket_e b)$
by *pred-auto*

lemma *ueqe2*: $(\llbracket [P]_e b \Longrightarrow (\llbracket Q \rrbracket_e b = \llbracket R \rrbracket_e b) \wedge \llbracket Q \wedge P \rrbracket_e b = \llbracket R \wedge P \rrbracket_e b) \Longrightarrow (\llbracket [P]_e b \Longrightarrow (\llbracket Q \rrbracket_e b = \llbracket R \rrbracket_e b))$
by *pred-auto*

lemma *ueqe3*: $\llbracket [P]_e b \Longrightarrow (\llbracket Q \rrbracket_e b = \llbracket R \rrbracket_e b) \rrbracket \Longrightarrow (\llbracket R \wedge P \rrbracket_e b = \llbracket Q \wedge P \rrbracket_e b)$
by *pred-auto*

The following allows simplifying the equality if $P \Rightarrow Q = R$

lemma *ueqe3-imp*: $(\bigwedge b. \llbracket P \rrbracket_e b \Longrightarrow (\llbracket Q \rrbracket_e b = \llbracket R \rrbracket_e b)) \Longrightarrow ((R \wedge P) = (Q \wedge P))$
by *pred-auto*

lemma *ueqe3-imp3*: $(\bigwedge b. \llbracket P \rrbracket_e b \Longrightarrow (\llbracket Q \rrbracket_e b = \llbracket R \rrbracket_e b)) \Longrightarrow ((P \wedge Q) = (P \wedge R))$
by *pred-auto*

lemma *ueqe3-imp2*: $\llbracket (\bigwedge b. \llbracket P0 \wedge P1 \rrbracket_e b \Longrightarrow \llbracket Q \rrbracket_e b \Longrightarrow \llbracket R \rrbracket_e b = \llbracket S \rrbracket_e b) \rrbracket \Longrightarrow ((P0 \wedge P1 \wedge (Q \Rightarrow R)) = (P0 \wedge P1 \wedge (Q \Rightarrow S)))$
by *pred-auto*

The following can introduce the binding notation into predicates

lemma *conj-bind-dist*: $\llbracket P \wedge Q \rrbracket_e b = (\llbracket P \rrbracket_e b \wedge \llbracket Q \rrbracket_e b)$
 by *pred-auto*

lemma *disj-bind-dist*: $\llbracket P \vee Q \rrbracket_e b = (\llbracket P \rrbracket_e b \vee \llbracket Q \rrbracket_e b)$
 by *pred-auto*

lemma *imp-bind-dist*: $\llbracket P \Rightarrow Q \rrbracket_e b = (\llbracket P \rrbracket_e b \longrightarrow \llbracket Q \rrbracket_e b)$
 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*:
 $out\alpha \# b \Longrightarrow ((P \triangleleft b \triangleright Q) ;; R) = ((P ;; R) \triangleleft b \triangleright (Q ;; R))$
 by (*rel-auto*)

lemma *cond-seq-right-distr*:
 $in\alpha \# b \Longrightarrow (P ;; (Q \triangleleft b \triangleright R)) = ((P ;; Q) \triangleleft b \triangleright (P ;; R))$
 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*: $out\alpha \# p \Longrightarrow (p ;; true) = p$
 by (*metis precond-equiv*)

lemma *postcond-left-unit*: $in\alpha \# p \Longrightarrow (true ;; p) = p$
 by (*metis postcond-equiv*)

theorem *precond-left-zero*:
 assumes $out\alpha \# p \neq false$
 shows $(true ;; p) = true$

using *assms* **by** (*rel-auto*)

theorem *feasibile-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-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 add: true-alt-def false-alt-def*)

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*: $\text{in}\alpha \# q \Longrightarrow ((P \wedge q) ;; R) = (P ;; (q^- \wedge R))$
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*: $\text{in}\alpha \# r \Longrightarrow (P ;; (Q \wedge r)) = ((P ;; Q) \wedge r)$
by (*rel-blast*)

lemma *seqr-post-var-out*:
fixes $x :: (\text{bool} \Longrightarrow 'a)$
shows $(P ;; (Q \wedge \$x')) = ((P ;; Q) \wedge \$x')$
by (*rel-auto*)

theorem *seqr-post-transfer*: $\text{out}\alpha \# q \Longrightarrow (P ;; (q \wedge R)) = ((P \wedge q^-) ;; R)$
by (*rel-auto*)

lemma *seqr-pre-out*: $\text{out}\alpha \# p \Longrightarrow ((p \wedge Q) ;; R) = (p \wedge (Q ;; R))$
by (*rel-blast*)

lemma *seqr-pre-var-out*:
fixes $x :: (bool \implies 'a)$
shows $((\$x \wedge P) ;; Q) = (\$x \wedge (P ;; Q))$
by (*rel-auto*)

lemma *seqr-true-lemma*:
 $(P = (\neg ((\neg P) ;; true))) = (P = (P ;; true))$
by (*rel-auto*)

lemma *seqr-to-conj*: $\llbracket out\alpha \# P; in\alpha \# Q \rrbracket \implies (P ;; Q) = (P \wedge Q)$
by (*metis postcond-left-unit seqr-pre-out utp-pred-laws.inf-top.right-neutral*)

lemma *shEx-lift-seq-1* [*uquant-lift*]:
 $((\exists x \cdot P\ x) ;; Q) = (\exists x \cdot (P\ x ;; Q))$
by *pred-auto*

lemma *shEx-lift-seq-2* [*uquant-lift*]:
 $(P ;; (\exists x \cdot Q\ x)) = (\exists x \cdot (P ;; Q\ x))$
by *pred-auto*

18.4 Iterated Sequential Composition Laws

lemma *iter-seqr-nil* [*simp*]: $(;; i : [] \cdot P(i)) = II$
by (*simp add: seqr-iter-def*)

lemma *iter-seqr-cons* [*simp*]: $(;; i : (x \# xs) \cdot P(i)) = P(x) ;; (;; i : xs \cdot P(i))$
by (*simp add: seqr-iter-def*)

18.5 Quantale Laws

lemma *seq-Sup-distl*: $P ;; (\bigcap A) = (\bigcap_{Q \in A} P ;; Q)$
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*: $P ;; (\bigcap_{i \in A} Q(i)) = (\bigcap_{i \in A} P ;; Q(i))$
by (*metis image-image seq-Sup-distl*)

lemma *seq-SUP-distr*: $(\bigcap_{i \in A} P(i)) ;; Q = (\bigcap_{i \in A} P(i) ;; Q)$
by (*simp add: seq-Sup-distr*)

18.6 Skip Laws

lemma *cond-skip*: $\text{out}\alpha \# b \implies (b \wedge II) = (II \wedge b^-)$
by (*rel-auto*)

lemma *pre-skip-post*: $(\lceil b \rceil_{<} \wedge II) = (II \wedge \lceil b \rceil_{>})$
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*]:
 $\lceil \sigma \rceil_s \dagger \langle \varrho \rangle_a = \langle \varrho \circ \sigma \rangle_a$
by (*rel-auto*)

lemma *assigns-r-comp*: $(\langle \sigma \rangle_a ;; P) = (\lceil \sigma \rceil_s \dagger P)$
by (*rel-auto*)

lemma *assigns-r-feasible*:
 $(\langle \sigma \rangle_a ;; \text{true}) = \text{true}$
by (*rel-auto*)

lemma *assign-subst* [*usubst*]:
 $\llbracket \text{mwb-lens } x; \text{mwb-lens } y \rrbracket \implies [\$x \mapsto_s \lceil u \rceil_{<}] \dagger (y := v) = (x, y) := (u, [x \mapsto_s u] \dagger v)$
by (*rel-auto*)

lemma *assign-vacuous-skip*:
assumes $\text{vwb-lens } x$
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 $x :: ('a \implies 'a)$
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 $x :: ('a \implies 'a)$
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* $f \implies \text{uinj } \langle f \rangle_a$
by (*rel-simp*, *simp add: inj-eq*)

lemma *assigns-r-swap-uinj*:
 $[[vwb\text{-}lens\ x; vwb\text{-}lens\ y; x \bowtie y]] \implies \text{uinj } ((x,y) := (\&y, \&x))$
by (*metis assigns-r-uinj pr-var-def swap-usubst-inj*)

lemma *assign-unfold*:
 $vwb\text{-}lens\ x \implies (x := v) = (\$x' =_u [v]_{<} \wedge II|_{\alpha} x)$
apply (*rel-auto*, *auto simp add: comp-def*)
using *vwb-lens.put-eq* **by** *fastforce*

18.8 Converse Laws

lemma *convr-invol* [*simp*]: $p^{--} = p$
by *pred-auto*

lemma *lit-convr* [*simp*]: $\ll v \gg^- = \ll v \gg$
by *pred-auto*

lemma *uivar-convr* [*simp*]:
fixes $x :: ('a \implies 'a)$
shows $(\$x)^- = \x'
by *pred-auto*

lemma *uovar-convr* [*simp*]:
fixes $x :: ('a \implies 'a)$
shows $(\$x')^- = \x
by *pred-auto*

lemma *uop-convr* [*simp*]: $(uop\ f\ u)^- = uop\ f\ (u^-)$
by (*pred-auto*)

lemma *bop-convr* [*simp*]: $(bop\ f\ u\ v)^- = bop\ f\ (u^-)\ (v^-)$
by (*pred-auto*)

lemma *eq-convr* [*simp*]: $(p =_u q)^- = (p^- =_u q^-)$
by (*pred-auto*)

lemma *not-convr* [*simp*]: $(\neg p)^- = (\neg p^-)$
by (*pred-auto*)

lemma *disj-convr* [*simp*]: $(p \vee q)^- = (q^- \vee p^-)$
by (*pred-auto*)

lemma *conj-convr* [*simp*]: $(p \wedge q)^- = (q^- \wedge p^-)$
by (*pred-auto*)

lemma *seqr-convr* [*simp*]: $(p \mathrel{;;} q)^- = (q^- \mathrel{;;} p^-)$
by (*rel-auto*)

lemma *pre-convr* [*simp*]: $\lceil p \rceil_{<}^- = \lceil p \rceil_{>}$
by (*rel-auto*)

lemma *post-convr* [*simp*]: $\lceil p \rceil_{>}^- = \lceil p \rceil_{<}$
by (*rel-auto*)

18.9 Assertion and Assumption Laws

declare *sublens-def* [*lens-defs del*]

lemma *assume-false*: $\lceil false \rceil^\top = false$
by (*rel-auto*)

lemma *assume-true*: $\lceil true \rceil^\top = II$
by (*rel-auto*)

lemma *assume-seq*: $\lceil b \rceil^\top \mathrel{;;} \lceil c \rceil^\top = \lceil b \wedge c \rceil^\top$
by (*rel-auto*)

lemma *assert-false*: $\{false\}_\perp = true$
by (*rel-auto*)

lemma *assert-true*: $\{true\}_\perp = II$
by (*rel-auto*)

lemma *assert-seq*: $\{b\}_\perp \mathrel{;;} \{c\}_\perp = \{b \wedge c\}_\perp$
by (*rel-auto*)

18.10 Frame and Antiframe Laws

named-theorems *frame*

lemma *frame-all* [*frame*]: $\Sigma:[P] = P$
by (*rel-auto*)

lemma *frame-none* [*frame*]:
 $\emptyset:[P] = (P \wedge II)$
by (*rel-auto*)

lemma *frame-commute*:
assumes $\$y \# P \$y' \# P \$x \# Q \$x' \# Q x \bowtie y$
shows $x:[P] \mathrel{;;} y:[Q] = y:[Q] \mathrel{;;} x:[P]$
apply (*insert assms*)
apply (*rel-auto*)
apply (*rename-tac s s' s₀*)
apply (*subgoal-tac (s \oplus_L s' on y) \oplus_L s₀ on x = s₀ \oplus_L s' on y)*)
apply (*metis lens-indep-get lens-indep-sym lens-override-def*)
apply (*simp add: lens-indep.lens-put-comm lens-override-def*)
apply (*rename-tac s s' s₀*)
apply (*subgoal-tac put_y (put_x s (get_x (put_x s₀ (get_x s')))) (get_y (put_y s (get_y s₀)))*)
 $= put_x s_0 (get_x s')$
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*]:

$x:[false] = false$

by (*rel-auto*)

lemma *frame-skip* [*simp*]:

vwb-lens $x \implies x:[II] = II$

by (*rel-auto*)

lemma *frame-assign-in* [*frame*]:

$\llbracket vwb-lens\ a;\ x \subseteq_L\ a \rrbracket \implies a:[x := v] = x := v$

by (*rel-auto, simp-all add: lens-get-put-quasi-commute lens-put-of-quotient*)

lemma *frame-conj-true* [*frame*]:

$\llbracket \{\$x, \$x'\} \Vdash P;\ vwb-lens\ x \rrbracket \implies (P \wedge x:[true]) = x:[P]$

by (*rel-auto*)

lemma *frame-is-assign* [*frame*]:

vwb-lens $x \implies x:[\$x' =_u [v]_{<}] = x := v$

by (*rel-auto*)

lemma *frame-seq* [*frame*]:

$\llbracket vwb-lens\ x;\ \{\$x, \$x'\} \Vdash P;\ \{\$x, \$x'\} \Vdash Q \rrbracket \implies x:[P ;; Q] = x:[P] ;; x:[Q]$

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*]:

$vwb\text{-}lens\ a \implies a:[P ;; Q]^+ = (a:[P]^+ ;; a:[Q]^+)$
apply (*rel-auto*)
apply (*rename-tac s s' s₀*)
apply (*rule-tac x=put_a s s₀ in exI*)
apply (*auto*)
apply (*metis mwb-lens.put-put vwb-lens-mwb*)
done

lemma *rel-frex-assigns* [*frame*]:
 $vwb\text{-}lens\ a \implies a:[\langle\sigma\rangle_a]^+ = \langle\sigma \oplus_s a\rangle_a$
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*]:
 $\llbracket vwb\text{-}lens\ x; \$x' \# P; \$x \# Q \rrbracket \implies (x:[P] ;; x:[Q]) = x:[P ;; Q]$
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*]

18.11 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: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-def*)

also have ... = $((P ;; (\nu X \cdot (P ;; X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
 by (subst lfp-unfold, simp-all add: m)
 also have ... = $((P ;; \text{while } b \text{ do } P \text{ od}) \triangleleft b \triangleright_r II)$
 by (simp add: while-def)
 finally show ?thesis .
 qed

theorem while-false: $\text{while false do } P \text{ od} = II$
 by (subst while-unfold, rel-auto)

theorem while-true: $\text{while true do } P \text{ od} = \text{false}$
 apply (simp add: while-def alpha)
 apply (rule antisym)
 apply (simp-all)
 apply (rule lfp-lowerbound)
 apply (rel-auto)
 done

theorem while-bot-unfold:
 $\text{while}_\perp b \text{ do } P \text{ od} = ((P ;; \text{while}_\perp b \text{ do } P \text{ od}) \triangleleft b \triangleright_r II)$

proof –
 have $m:\text{mono } (\lambda X. (P ;; X) \triangleleft b \triangleright_r II)$
 by (auto intro: monoI segr-mono cond-mono)
 have $(\text{while}_\perp b \text{ do } P \text{ od}) = (\mu X \cdot (P ;; X) \triangleleft b \triangleright_r II)$
 by (simp add: while-bot-def)
 also have ... = $((P ;; (\mu X \cdot (P ;; X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
 by (subst gfp-unfold, simp-all add: m)
 also have ... = $((P ;; \text{while}_\perp b \text{ do } P \text{ od}) \triangleleft b \triangleright_r II)$
 by (simp add: while-bot-def)
 finally show ?thesis .
 qed

theorem while-bot-false: $\text{while}_\perp \text{false do } P \text{ od} = II$
 by (simp add: while-bot-def mu-const alpha)

theorem while-bot-true: $\text{while}_\perp \text{true do } P \text{ od} = (\mu X \cdot P ;; X)$
 by (simp add: while-bot-def alpha)

An infinite loop with a feasible body corresponds to a program error (non-termination).

theorem while-infinite: $P ;; \text{true}_h = \text{true} \implies \text{while}_\perp \text{true do } P \text{ od} = \text{true}$
 apply (simp add: while-bot-true)
 apply (rule antisym)
 apply (simp)
 apply (rule gfp-upperbound)
 apply (simp)
 done

18.12 Algebraic Properties

interpretation upred-semiring: semiring-1

where times = segr and one = skip-r and zero = false_h and plus = Lattices.sup
 by (unfold-locales, (rel-auto)+)

declare upred-semiring.power-Suc [simp del]

We introduce the power syntax derived from semirings

abbreviation $upower :: 'a \text{ hrel} \Rightarrow \text{nat} \Rightarrow 'a \text{ hrel}$ (**infixr** \wedge 80) **where**
 $upower\ P\ n \equiv upred\text{-semiring}.power\ P\ n$

translations

$P \wedge i \leq CONST\ power.power\ II\ op \;; P\ i$
 $P \wedge i \leq (CONST\ power.power\ II\ op \;; P)\ i$

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 segr.rep-eq relpow-commute upred-semiring.power-Suc*)

qed

lemma *upower-rep-eq-alt*:

$\llbracket power.power\ \langle id \rangle_a\ op \;; P\ i \rrbracket_e = (\lambda b. b \in (\{p. \llbracket P \rrbracket_e p\} \wedge i))$

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 $UNIV = \text{insert}\ (0::\text{nat})\ \{1..\}$

by *auto*

moreover have $(\bigsqcap i. P(i)) = \bigsqcap (P \restriction UNIV)$

by (*blast*)

moreover have $\bigsqcap (P \restriction \text{insert}\ 0\ \{1..\}) = P(0) \sqcap SUPREMUM\ \{1..\}\ P$

by (*simp*)

moreover have $SUPREMUM\ \{1..\}\ P = (\bigsqcap i. P(i+1))$

by (*simp add: atLeast-Suc-greaterThan greaterThan-0*)

ultimately show ?thesis

by (*simp only:*)

qed

lemma *Sup-upto-Suc*: $(\bigsqcap i \in \{0..Suc\ n\}. P \wedge i) = (\bigsqcap i \in \{0..n\}. P \wedge i) \sqcap P \wedge Suc\ n$

proof –

have $(\bigsqcap i \in \{0..Suc\ n\}. P \wedge i) = (\bigsqcap i \in \text{insert}\ (Suc\ n)\ \{0..n\}. P \wedge i)$

by (*simp add: atLeast0-atMost-Suc*)

also have $\dots = P \wedge Suc\ n \sqcap (\bigsqcap i \in \{0..n\}. P \wedge i)$

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*: $Q \sqsubseteq (P \;; Q \sqcap R) \Longrightarrow Q \sqsubseteq P \wedge n \;; R$

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
segr-assoc segr-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  $R ;; P \wedge \text{Suc } n = (R ;; P \wedge n) ;; P$ 
  by (metis segr-assoc upred-semiring.power-Suc2)
also have  $Q ;; P \sqsubseteq \dots$ 
  by (meson Suc.hyps assms eq-iff segr-mono)
also have  $Q \sqsubseteq \dots$ 
  using assms by auto
finally show ?case .
qed

```

```

lemma SUP-atLeastAtMost-first:
  fixes  $P :: \text{nat} \Rightarrow 'a::\text{complete-lattice}$ 
  assumes  $m \leq n$ 
  shows  $(\bigcap_{i \in \{m..n\}}. P(i)) = P(m) \sqcap (\bigcap_{i \in \{\text{Suc } m..n\}}. P(i))$ 
  by (metis SUP-insert assms atLeastAtMost-insertL)

```

```

lemma upower-segr-iter:  $P \wedge n = (;; Q : \text{replicate } n P \cdot Q)$ 
  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+)

```

18.12.1 Kleene Star

```

definition ustar :: ' $\alpha$  hrel  $\Rightarrow$  ' $\alpha$  hrel  $(-^* [999] 999)$  where
 $P^* = (\bigcap_{i \in \{0..\}} \cdot P^i)$ 

```

```

lemma ustar-rep-eq:
 $\llbracket P^* \rrbracket_e = (\lambda b. b \in (\{p. \llbracket P \rrbracket_e p\}^*))$ 
  by (simp add: ustar-def, rel-auto, simp-all add: relpow-imp-rtrancl rtrancl-imp-relpow)

```

update-uexpr-rep-eq-thms

18.13 Kleene Plus

```

purge-notation trancl  $((-^+) [1000] 999)$ 

```

```

definition uplus :: ' $\alpha$  hrel  $\Rightarrow$  ' $\alpha$  hrel  $(-^+ [999] 999)$  where
 $[upred-defs]: P^+ = P ;; P^*$ 

```

lemma *uplus-power-def*: $P^+ = (\bigsqcap i \cdot P \hat{\ } (Suc\ i))$
by (*simp add: uplus-def ustar-def seq-UNF-distl' UNF-atLeast-Suc upred-semiring.power-Suc*)

18.14 Omega

definition *uomega* :: $'\alpha \text{ hrel} \Rightarrow '\alpha \text{ hrel} (-^\omega [999] 999)$ **where**
 $P^\omega = (\mu\ X \cdot P \ ;\ ;\ X)$

18.15 Relation Algebra Laws

theorem *RA1*: $(P \ ;\ ;\ (Q \ ;\ ;\ R)) = ((P \ ;\ ;\ Q) \ ;\ ;\ R)$
by (*simp add: seqr-assoc*)

theorem *RA2*: $(P \ ;\ ;\ II) = P \ (II \ ;\ ;\ P) = P$
by *simp-all*

theorem *RA3*: $P^{--} = P$
by *simp*

theorem *RA4*: $(P \ ;\ ;\ Q)^- = (Q^- \ ;\ ;\ P^-)$
by *simp*

theorem *RA5*: $(P \vee Q)^- = (P^- \vee Q^-)$
by (*rel-auto*)

theorem *RA6*: $((P \vee Q) \ ;\ ;\ R) = (P \ ;\ ;\ R \vee Q \ ;\ ;\ R)$
using *seqr-or-distl* **by** *blast*

theorem *RA7*: $((P^- \ ;\ ;\ (\neg(P \ ;\ ;\ Q))) \vee (\neg Q)) = (\neg Q)$
by (*rel-auto*)

18.16 Kleene Algebra Laws

lemma *ustar-alt-def*: $P^* = (\bigsqcap i \cdot P \hat{\ } i)$
by (*simp add: ustar-def*)

theorem *ustar-sub-unfoldl*: $P^* \sqsubseteq II \sqcap P \ ;\ ;\ P^*$
by (*rel-simp, simp add: rtrancl-into-trancl2 trancl-into-rtrancl*)

theorem *ustar-inductl*:
assumes $Q \sqsubseteq R \ Q \sqsubseteq P \ ;\ ;\ Q$
shows $Q \sqsubseteq P^* \ ;\ ;\ R$

proof –
have $P^* \ ;\ ;\ R = (\bigsqcap i. P \hat{\ } i \ ;\ ;\ R)$
by (*simp add: ustar-def UNF-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 $Q \sqsubseteq R \ Q \sqsubseteq Q \ ;\ ;\ P$
shows $Q \sqsubseteq R \ ;\ ;\ P^*$

proof –
have $R \ ;\ ;\ P^* = (\bigsqcap i. R \ ;\ ;\ P \hat{\ } i)$
by (*simp add: ustar-def UNF-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:
 $while\ b\ do\ P\ od = (P \triangleleft b \triangleright_r II)^* ;; [\neg b]^\top$
 proof –
 have 1: Continuous $(\lambda X. P ;; X \triangleleft b \triangleright_r II)$
 by (rel-auto)
 have while $while\ b\ do\ P\ od = (\bigcap i. ((\lambda X. P ;; X \triangleleft b \triangleright_r II) \hat{\ } i) \text{ false})$
 by (simp add: 1 false-upred-def sup-continuous-Continuous sup-continuous-lfp while-def)
 also have $\dots = ((\lambda X. P ;; X \triangleleft b \triangleright_r II) \hat{\ } 0) \text{ false} \sqcap (\bigcap i. ((\lambda X. P ;; X \triangleleft b \triangleright_r II) \hat{\ } (i+1)) \text{ false})$
 by (subst Sup-power-expand, simp)
 also have $\dots = (\bigcap i. ((\lambda X. P ;; X \triangleleft b \triangleright_r II) \hat{\ } (i+1)) \text{ false})$
 by (simp)
 also have $\dots = (\bigcap i. (P \triangleleft b \triangleright_r II) \hat{\ } i ;; (\text{false} \triangleleft b \triangleright_r II))$
 proof (rule SUP-cong, simp-all)
 fix i
 show $P ;; ((\lambda X. P ;; X \triangleleft b \triangleright_r II) \hat{\ } i) \text{ false} \triangleleft b \triangleright_r II = (P \triangleleft b \triangleright_r II) \hat{\ } i ;; (\text{false} \triangleleft b \triangleright_r II)$
 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 $\dots = (\bigcap i \in \{0..\} \cdot (P \triangleleft b \triangleright_r II) \hat{\ } i ;; [\neg b]^\top)$
 by (rel-auto)
 also have $\dots = (P \triangleleft b \triangleright_r II)^* ;; [\neg b]^\top$
 by (metis seq-UNF-distr ustar-def)

finally show *?thesis* .
qed

18.17 Omega Algebra Laws

lemma *uomega-induct*:
 $P \;; P^\omega \sqsubseteq P^\omega$
 by (simp add: uomega-def, metis eq-refl gfp-unfold monoI seqr-mono)

18.18 Refinement Laws

lemma *skip-r-refine*:
 $(p \Rightarrow p) \sqsubseteq II$
 by pred-blast

lemma *conj-refine-left*:
 $(Q \Rightarrow P) \sqsubseteq R \Longrightarrow P \sqsubseteq (Q \wedge R)$
 by (rel-auto)

lemma *pre-weak-rel*:
 assumes ' $Pre \Rightarrow I$ '
 and $(I \Rightarrow Post) \sqsubseteq P$
 shows $(Pre \Rightarrow Post) \sqsubseteq P$
 using assms by (rel-auto)

lemma *cond-refine-rel*:
 assumes $S \sqsubseteq (\lceil b \rceil_< \wedge P) \sqsubseteq (\lceil \neg b \rceil_< \wedge Q)$
 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 $(\lceil b \rceil_< \Rightarrow \lceil s \rceil_>) \sqsubseteq P$ and $(\lceil s \rceil_< \Rightarrow \lceil c \rceil_>) \sqsubseteq Q$
 shows $(\lceil b \rceil_< \Rightarrow \lceil c \rceil_>) \sqsubseteq (P \;; Q)$
 using assms by rel-auto

lemma *seq-refine-unrest*:
 assumes $out\alpha \nmid b \text{ in } \alpha \nmid c$
 assumes $(b \Rightarrow \lceil s \rceil_>) \sqsubseteq P$ and $(\lceil s \rceil_< \Rightarrow c) \sqsubseteq Q$
 shows $(b \Rightarrow c) \sqsubseteq (P \;; Q)$
 using assms by rel-blast

18.19 Domain and Range Laws

lemma *Dom-conv-Ran*:
 $Dom(P^-) = Ran(P)$
 by (rel-auto)

lemma *Ran-conv-Dom*:
 $Ran(P^-) = Dom(P)$
 by (rel-auto)

lemma *Dom-skip*:
 $Dom(II) = true$
 by (rel-auto)

lemma *Dom-assigns*:

```

Dom( $\langle \sigma \rangle_a$ ) = true
by (rel-auto)

```

```

lemma Dom-miracle:
  Dom(false) = false
  by (rel-auto)

```

```

lemma Dom-assume:
  Dom( $[b]^\top$ ) = b
  by (rel-auto)

```

```

lemma Dom-seq:
  Dom( $P ;; Q$ ) = Dom( $P ;; [Dom(Q)]^\top$ )
  by (rel-auto)

```

```

lemma Dom-disj:
  Dom( $P \vee Q$ ) = (Dom( $P$ )  $\vee$  Dom( $Q$ ))
  by (rel-auto)

```

```

lemma Dom-inf:
  Dom( $P \sqcap Q$ ) = (Dom( $P$ )  $\vee$  Dom( $Q$ ))
  by (rel-auto)

```

```

lemma assume-Dom:
   $[Dom(P)]^\top ;; P = P$ 
  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 :: ( $'\alpha$  upred) gorder ( $\mathcal{P}$ ) where
  upred-lattice =  $\langle \mid$  carrier = UNIV, eq = ( $op =$ ), le =  $op \sqsubseteq \mid$   $\rangle$ 

```

\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-locale, simp-all add: upred-lattice-def)

```

```

  fix A ::  $'\alpha$  upred set

```

```

  show  $\exists s. is\_lub \langle \mid$  carrier = UNIV, eq =  $op =$ , le =  $op \sqsubseteq \mid$   $\rangle s A$ 

```

```

    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  $\exists i. \text{is\_glb } (\text{carrier} = \text{UNIV}, \text{eq} = \text{op} =, \text{le} = \text{op} \sqsubseteq) i A$ 
  apply (rule-tac  $x = \sqcap A$  in  $\text{exI}$ )
  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]:
   $\text{op} \text{.}=\mathcal{P} = \text{op} =$ 
  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

```

typedef ( $'\mathcal{T}, ' \alpha$ ) uthy = UNIV :: unit set
  by auto

```

We create a unitary parametric type to represent UTP theories. These are merely tags and contain no data other than to help the type-system resolve polymorphic definitions. The two parameters denote the name of the UTP theory – as a unique type – and the minimal alphabet that the UTP theory requires. We will then use Isabelle’s ad-hoc overloading mechanism to associate theory constructs, like healthiness conditions and units, with each of these types. This will allow the type system to retrieve definitions based on a particular theory context.

```

definition uthy :: ( $'a, 'b$ ) uthy where
  uthy = Abs-uthy ()

```

```

lemma uthy-eq [intro]:
  fixes  $x y :: ('a, 'b) \text{uthy}$ 
  shows  $x = y$ 
  by (cases  $x$ , cases  $y$ , simp)

```

```

syntax
  -UTHY :: type  $\Rightarrow$  type  $\Rightarrow$  logic (UTHY'(-, -))

```

```

translations
  UTHY( $'T, ' \alpha$ ) == CONST uthy :: ( $'T, ' \alpha$ ) uthy

```

We set up polymorphic constants to denote the healthiness conditions associated with a UTP theory. Unfortunately we can currently only characterise UTP theories of homogeneous relations; this is due to restrictions in the instantiation of Isabelle's polymorphic constants which apparently cannot specialise types in this way.

consts

utp-hcond :: (*'T*, *'α*) *uthy* \Rightarrow (*'α* \times *'α*) *health* (\mathcal{H}_1)

definition *utp-order* :: (*'α* \times *'α*) *health* \Rightarrow *'α* *hrel gorder* **where**

utp-order *H* = (\lfloor *carrier* = {*P*. *P* is *H*}, *eq* = (*op* =), *le* = *op* \sqsubseteq \rfloor)

abbreviation *uthy-order* *T* \equiv *utp-order* \mathcal{H}_T

Constant *utp-order* obtains the order structure associated with a UTP theory. Its carrier is the set of healthy predicates, equality is HOL equality, and the order is refinement.

lemma *utp-order-carrier* [*simp*]:

carrier (*utp-order* *H*) = $\llbracket H \rrbracket_H$

by (*simp* *add*: *utp-order-def*)

lemma *utp-order-eq* [*simp*]:

eq (*utp-order* *T*) = *op* =

by (*simp* *add*: *utp-order-def*)

lemma *utp-order-le* [*simp*]:

le (*utp-order* *T*) = *op* \sqsubseteq

by (*simp* *add*: *utp-order-def*)

lemma *utp-partial-order*: *partial-order* (*utp-order* *T*)

by (*unfold-locales*, *simp-all* *add*: *utp-order-def*)

lemma *utp-weak-partial-order*: *weak-partial-order* (*utp-order* *T*)

by (*unfold-locales*, *simp-all* *add*: *utp-order-def*)

lemma *mono-Monotone-utp-order*:

mono *f* \Longrightarrow *Monotone* (*utp-order* *T*) *f*

apply (*auto* *simp* *add*: *isotone-def*)

apply (*metis* *partial-order-def* *utp-partial-order*)

apply (*metis* *monoD*)

done

lemma *isotone-utp-orderI*: *Monotonic* *H* \Longrightarrow *isotone* (*utp-order* *X*) (*utp-order* *Y*) *H*

by (*auto* *simp* *add*: *mono-def* *isotone-def* *utp-weak-partial-order*)

lemma *Mono-utp-orderI*:

$\llbracket \bigwedge P Q. \llbracket P \sqsubseteq Q; P \text{ is } H; Q \text{ is } H \rrbracket \Longrightarrow F(P) \sqsubseteq F(Q) \rrbracket \Longrightarrow \text{Mono}_{\text{utp-order } H} F$

by (*auto* *simp* *add*: *isotone-def* *utp-weak-partial-order*)

The UTP order can equivalently be characterised as the fixed point lattice, *fpl*.

lemma *utp-order-fpl*: *utp-order* *H* = *fpl* \mathcal{P} *H*

by (*auto* *simp* *add*: *utp-order-def* *upred-lattice-def* *fps-def* *Healthy-def*)

definition *uth-eq* :: (*'T*₁, *'α*) *uthy* \Rightarrow (*'T*₂, *'α*) *uthy* \Rightarrow *bool* (**infix** \approx_T 50) **where**

*T*₁ \approx_T *T*₂ $\longleftrightarrow \llbracket \mathcal{H}_{T_1} \rrbracket_H = \llbracket \mathcal{H}_{T_2} \rrbracket_H$

lemma *uth-eq-refl*: *T* \approx_T *T*

by (simp add: uth-eq-def)

lemma uth-eq-sym: $T_1 \approx_T T_2 \longleftrightarrow T_2 \approx_T T_1$
 by (auto simp add: uth-eq-def)

lemma uth-eq-trans: $\llbracket T_1 \approx_T T_2; T_2 \approx_T T_3 \rrbracket \implies T_1 \approx_T T_3$
 by (auto simp add: uth-eq-def)

definition uthy-plus :: $('T_1, 'α) \text{ uthy} \Rightarrow ('T_2, 'α) \text{ uthy} \Rightarrow ('T_1 \times 'T_2, 'α) \text{ uthy}$ (**infixl** $+_T$ 65) **where**
 uthy-plus $T_1 \ T_2 = \text{uthy}$

overloading

prod-hcond == utp-hcond :: $('T_1 \times 'T_2, 'α) \text{ uthy} \Rightarrow ('α \times 'α) \text{ health}$

begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition prod-hcond :: $('T_1 \times 'T_2, 'α) \text{ uthy} \Rightarrow ('α \times 'α) \text{ upred} \Rightarrow ('α \times 'α) \text{ upred}$ **where**
 prod-hcond $T = \mathcal{H}_{UTHY}('T_1, 'α) \circ \mathcal{H}_{UTHY}('T_2, 'α)$

end

19.3 UTP theory hierarchy

We next define a hierarchy of locales that characterise different classes of UTP theory. Minimally we require that a UTP theory's healthiness condition is idempotent.

locale utp-theory =
 fixes $\mathcal{T} :: ('T, 'α) \text{ uthy}$ (**structure**)
 assumes HCond-Idem: $\mathcal{H}(\mathcal{H}(P)) = \mathcal{H}(P)$
begin

lemma uthy-simp:
 uthy = \mathcal{T}
 by blast

A UTP theory fixes \mathcal{T} , the structural element denoting the UTP theory. All constants associated with UTP theories can then be resolved by the type system.

lemma HCond-Idempotent [closure,intro]: Idempotent \mathcal{H}
 by (simp add: Idempotent-def HCond-Idem)

sublocale partial-order uthy-order \mathcal{T}
 by (unfold-locales, simp-all add: utp-order-def)

end

Theory summation is commutative provided the healthiness conditions commute.

lemma uthy-plus-comm:
 assumes $\mathcal{H}_{T_1} \circ \mathcal{H}_{T_2} = \mathcal{H}_{T_2} \circ \mathcal{H}_{T_1}$
 shows $T_1 +_T T_2 \approx_T T_2 +_T T_1$
proof –
 have $T_1 = \text{uthy } T_2 = \text{uthy}$
 by blast+
 thus ?thesis
 using assms by (simp add: uth-eq-def prod-hcond-def)
qed

lemma *uthy-plus-assoc*: $T_1 +_T (T_2 +_T T_3) \approx_T (T_1 +_T T_2) +_T T_3$
by (*simp add: uth-eq-def prod-hcond-def comp-def*)

lemma *uthy-plus-idem*: $utp-theory\ T \implies T +_T T \approx_T T$
by (*simp add: uth-eq-def prod-hcond-def Healthy-def utp-theory.HCond-Idem utp-theory.uthy-simp*)

locale *utp-theory-lattice* = *utp-theory* \mathcal{T} + *complete-lattice* *uthy-order* \mathcal{T} **for** $\mathcal{T} :: ('T, 'a) \text{ uthy (structure)}$

The healthiness conditions of a UTP theory lattice form a complete lattice, and allows us to make use of complete lattice results from HOL-Algebra, such as the Knaster-Tarski theorem. We can also retrieve lattice operators as below.

abbreviation *utp-top* (\top_1)
where *utp-top* $\mathcal{T} \equiv top\ (uthy-order\ \mathcal{T})$

abbreviation *utp-bottom* (\perp_1)
where *utp-bottom* $\mathcal{T} \equiv bottom\ (uthy-order\ \mathcal{T})$

abbreviation *utp-join* (**infixl** \sqcup_1 65) **where**
utp-join $\mathcal{T} \equiv join\ (uthy-order\ \mathcal{T})$

abbreviation *utp-meet* (**infixl** \sqcap_1 70) **where**
utp-meet $\mathcal{T} \equiv meet\ (uthy-order\ \mathcal{T})$

abbreviation *utp-sup* (\bigsqcup_1 - [90] 90) **where**
utp-sup $\mathcal{T} \equiv Lattice.sup\ (uthy-order\ \mathcal{T})$

abbreviation *utp-inf* (\bigsqcap_1 - [90] 90) **where**
utp-inf $\mathcal{T} \equiv Lattice.inf\ (uthy-order\ \mathcal{T})$

abbreviation *utp-gfp* (ν_1) **where**
utp-gfp $\mathcal{T} \equiv GREATEST-FP\ (uthy-order\ \mathcal{T})$

abbreviation *utp-lfp* (μ_1) **where**
utp-lfp $\mathcal{T} \equiv LEAST-FP\ (uthy-order\ \mathcal{T})$

syntax
 $-tmu :: logic \Rightarrow ptnrn \Rightarrow logic \Rightarrow logic\ (\mu_1 - \cdot - [0, 10]\ 10)$
 $-tnu :: logic \Rightarrow ptnrn \Rightarrow logic \Rightarrow logic\ (\nu_1 - \cdot - [0, 10]\ 10)$

notation *gfp* (μ)
notation *lfp* (ν)

translations
 $\mu_T\ X \cdot P == CONST\ utp-lfp\ T\ (\lambda\ X.\ P)$
 $\nu_T\ X \cdot P == CONST\ utp-gfp\ T\ (\lambda\ X.\ P)$

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 -

```

We can then use the Knaster-Tarski theorem to obtain a complete lattice, and thus provide all the usual properties.

```

interpret weak-complete-lattice fpl  $\mathcal{P} \mathcal{H}$ 
  by (rule Knaster-Tarski, auto simp add: upred-lattice.weak.weak-complete-lattice-axioms)

```

```

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 (uthy-order  $\mathcal{T}$ )
  by (simp add: utp-order-def, simp add: upred-lattice-def)

```

```

thus utp-theory-lattice  $\mathcal{T}$ 
  by (simp add: utp-theory-axioms utp-theory-lattice-def)
qed

```

```

context utp-theory-mono
begin

```

In a monotone theory, the top and bottom can always be obtained by applying the healthiness condition to the predicate top and bottom, respectively.

```

lemma healthy-top:  $\top = \mathcal{H}(\text{false})$ 
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  $\bigcap A = \mathcal{H} (\bigcap A)$ 
proof –
  have 1: weak-complete-lattice (uthy-order  $\mathcal{T}$ )
    by (simp add: weak.weak-complete-lattice-axioms)
  have 2: Monouthy-order  $\mathcal{T}$   $\mathcal{H}$ 
    by (simp add: HCond-Mono isotone-utp-orderI)
  have 3: Idemuthy-order  $\mathcal{T}$   $\mathcal{H}$ 
    by (simp add: HCond-Idem idempotent-def)
  show ?thesis
    using Knaster-Tarski-idem-inf-eq[OF upred-weak-complete-lattice, of  $\mathcal{H}$ ]
    by (simp, metis HCond-Idempotent HCond-Mono assms partial-object.simps(3) upred-lattice-def)
qed

```

end

```

locale utp-theory-continuous = utp-theory +
  assumes HCond-Cont [closure,intro]: Continuous  $\mathcal{H}$ 

```

sublocale *utp-theory-continuous* \subseteq *utp-theory-mono*

proof

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 *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* $F \in \llbracket \mathcal{H} \rrbracket_H \rightarrow \llbracket \mathcal{H} \rrbracket_H$

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=⊤ 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 (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
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: HCond-Cont UINF-Continuous-closed assms)

```

end

In another direction, we can also characterise UTP theories that are relational. Minimally this requires that the healthiness condition is closed under sequential composition.

```

locale utp-theory-rel =
  utp-theory +
  assumes Healthy-Sequence [closure]:  $\llbracket P \text{ is } \mathcal{H}; Q \text{ is } \mathcal{H} \rrbracket \implies (P ;; Q) \text{ is } \mathcal{H}$ 
begin

```

```

lemma upower-Suc-Healthy [closure]:
  assumes  $P$  is  $\mathcal{H}$ 
  shows  $P \hat{\ } \text{Suc } n$  is  $\mathcal{H}$ 
  by (induct  $n$ , simp-all add: closure assms upred-semiring.power-Suc)

end

locale utp-theory-cont-rel = utp-theory-continuous + utp-theory-rel
begin

lemma seq-cont-Sup-distl:
  assumes  $P$  is  $\mathcal{H}$   $A \subseteq \llbracket \mathcal{H} \rrbracket_H$   $A \neq \{\}$ 
  shows  $P ;; (\bigsqcap A) = \bigsqcap \{P ;; Q \mid Q. Q \in A\}$ 
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  $Q$  is  $\mathcal{H}$   $A \subseteq \llbracket \mathcal{H} \rrbracket_H$   $A \neq \{\}$ 
  shows  $(\bigsqcap A) ;; Q = \bigsqcap \{P ;; Q \mid P. P \in A\}$ 
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  $P$  is  $\mathcal{H}$ 
  shows  $P^+$  is  $\mathcal{H}$ 
  by (simp add: uplus-power-def closure assms)

```

end

There also exist UTP theories with units, and the following operator is a theory specific operator for them.

consts

utp-unit :: $(\mathcal{T}, \alpha) \text{ uthy} \Rightarrow \alpha \text{ hrel } (\mathcal{IT}_1)$

We can characterise the theory Kleene star by lifting the relational one.

definition *utp-star* ($-\star_1$ [999] 999) **where**
[upred-defs]: *utp-star* $\mathcal{T} P = (P^* ;; \mathcal{IT}_{\mathcal{T}})$

We can then characterise tests as refinements of units.

definition *utest* :: $(\mathcal{T}, \alpha) \text{ uthy} \Rightarrow \alpha \text{ hrel} \Rightarrow \text{bool}$ **where**
[upred-defs]: *utest* $\mathcal{T} b = (\mathcal{IT}_{\mathcal{T}} \sqsubseteq b)$

Not all theories have both a left and a right unit (e.g. H1-H2 designs) and so we split up the locale into two cases.

locale *utp-theory-left-unital* =
utp-theory-rel +

```

assumes Healthy-Left-Unit [closure]:  $\mathcal{II}$  is  $\mathcal{H}$ 
and Left-Unit:  $P$  is  $\mathcal{H} \implies (\mathcal{II} ;; P) = P$ 

locale utp-theory-right-unital =
  utp-theory-rel +
  assumes Healthy-Right-Unit [closure]:  $\mathcal{II}$  is  $\mathcal{H}$ 
  and Right-Unit:  $P$  is  $\mathcal{H} \implies (P ;; \mathcal{II}) = P$ 

locale utp-theory-unital =
  utp-theory-rel +
  assumes Healthy-Unit [closure]:  $\mathcal{II}$  is  $\mathcal{H}$ 
  and Unit-Left:  $P$  is  $\mathcal{H} \implies (\mathcal{II} ;; P) = P$ 
  and Unit-Right:  $P$  is  $\mathcal{H} \implies (P ;; \mathcal{II}) = P$ 
begin

lemma Unit-self [simp]:
   $\mathcal{II} ;; \mathcal{II} = \mathcal{II}$ 
  by (simp add: Healthy-Unit Unit-Right)

lemma utest-intro:
   $\mathcal{II} \sqsubseteq P \implies \text{utest } \mathcal{T} P$ 
  by (simp add: utest-def)

lemma utest-Unit [closure]:
   $\text{utest } \mathcal{T} \mathcal{II}$ 
  by (simp add: utest-def)

end

sublocale utp-theory-unital  $\subseteq$  utp-theory-left-unital
  by (simp add: Healthy-Unit Unit-Left Healthy-Sequence utp-theory-rel-def utp-theory-axioms utp-theory-rel-axioms-def
utp-theory-left-unital-axioms-def utp-theory-left-unital-def)

sublocale utp-theory-unital  $\subseteq$  utp-theory-right-unital
  by (simp add: Healthy-Unit Unit-Right Healthy-Sequence utp-theory-rel-def utp-theory-axioms utp-theory-rel-axioms-def
utp-theory-right-unital-axioms-def utp-theory-right-unital-def)

locale utp-theory-mono-unital = utp-theory-mono + utp-theory-unital
begin

lemma utest-Top [closure]:
   $\text{utest } \mathcal{T} \top$ 
  by (simp add: Healthy-Unit utest-def utp-top)
end

locale utp-theory-cont-unital = utp-theory-cont-rel + utp-theory-unital

sublocale utp-theory-cont-unital  $\subseteq$  utp-theory-mono-unital
  by (simp add: utp-theory-mono-axioms utp-theory-mono-unital-def utp-theory-unital-axioms)

locale utp-theory-unital-zero =
  utp-theory-unital +
  assumes Top-Left-Zero:  $P$  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 is \mathcal{H} utest \mathcal{T} b*

shows *b ;; $\top = \top$*

proof –

have *b $\sqcap \mathcal{II} = \mathcal{II}$*

by (*meson assms(2) semilattice-sup-class.le-iff-sup utest-def*)

then show *?thesis*

by (*metis (no-types) Top-Left-Zero Unit-Left assms(1) meet-top top-healthy upred-semiring.distrib-right*)

qed

end

19.4 Theory of relations

We can exemplify the creation of a UTP theory with the theory of relations, a trivial theory.

typeddecl *REL*

abbreviation *REL* \equiv *UTHY*(*REL*, ' α)

We declare the type *REL* to be the tag for this theory. We need know nothing about this type (other than it's non-empty), since it is merely a name. We also create the corresponding constant to refer to the theory. Then we can use it to instantiate the relevant polymorphic constants.

overloading

rel-hcond == *utp-hcond* :: (*REL*, ' α) *uthy* \Rightarrow (' $\alpha \times \alpha$) *health*

rel-unit == *utp-unit* :: (*REL*, ' α) *uthy* \Rightarrow ' α *hrel*

begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition *rel-hcond* :: (*REL*, ' α) *uthy* \Rightarrow (' $\alpha \times \alpha$) *upred* \Rightarrow (' $\alpha \times \alpha$) *upred* **where**
[upred-defs]: rel-hcond T = id

The unit of the theory is simply the relational unit.

definition *rel-unit* :: (*REL*, ' α) *uthy* \Rightarrow ' α *hrel* **where**
[upred-defs]: rel-unit T = II

end

Finally we can show that relations are a monotone and unital theory using a locale interpretation, which requires that we prove all the relevant properties. It's convenient to rewrite some of the theorems so that the provisos are more UTP like; e.g. that the carrier is the set of healthy predicates.

interpretation *rel-theory*: *utp-theory-mono-unital REL*

rewrites *carrier* (*uthy-order REL*) = $\llbracket id \rrbracket_H$

by (*unfold-locales, simp-all add: rel-hcond-def rel-unit-def Healthy-def*)

We can then, for instance, determine what the top and bottom of our new theory is.

lemma *REL-top*: $\top_{REL} = false$

by (*simp add: rel-theory.healthy-top, simp add: rel-hcond-def*)

lemma *REL-bottom*: $\perp_{REL} = true$
by (*simp add: rel-theory.healthy-bottom, simp add: rel-hcond-def*)

A number of theorems have been exported, such as the fixed point unfolding laws.

thm *rel-theory.GFP-unfold*

19.5 Theory links

We can also describe links between theories, such as Galois connections and retractions, using the following notation.

definition *mk-conn* $(- \Leftarrow \langle -, - \rangle \Rightarrow - [90, 0, 0, 91] \ 91)$ **where**
 $H1 \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \Rightarrow H2 \equiv () \text{ orderA} = \text{utp-order } H1, \text{ orderB} = \text{utp-order } H2, \text{ lower} = \mathcal{H}_2, \text{ upper} = \mathcal{H}_1 \ ()$

abbreviation *mk-conn'* $(- \Leftarrow \langle -, - \rangle \rightarrow - [90, 0, 0, 91] \ 91)$ **where**
 $T1 \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \rightarrow T2 \equiv \mathcal{H}_{T1} \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \Rightarrow \mathcal{H}_{T2}$

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*: $\text{mwb-lens } x \implies \text{Idempotent } (ex \ x)$
by (*simp add: Idempotent-def exists-twice*)

lemma *Monotonic-ex*: $\text{mwb-lens } x \implies \text{Monotonic } (ex \ x)$
by (*simp add: mono-def ex-mono*)

lemma *ex-closed-unrest*:
 $\text{vwb-lens } x \implies \llbracket ex \ x \rrbracket_H = \{P. x \# P\}$
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 *vwb-lens* $x \text{ Idempotent } H \text{ ex } x \circ H = H \circ ex \ x$
shows *retract* $((ex \ x \circ H) \Leftarrow \langle ex \ x, H \rangle \Rightarrow H)$
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 vwb-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 vwb-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-r-comp usubst unrest)?*, — Eliminate assignments where possible
(auto
intro: hoare intro!: hoare-safe hr
simp add: assigns-r-comp conj-comm conj-assoc usubst unrest))[1] — Apply Hoare logic laws

method *hoare-auto* **uses** *hr* = (*hoare-split hr: hr; rel-auto?*)

20.2 Basic Laws

lemma *hoare-r-conj* [*hoare-safe*]: $\llbracket \{p\}Q\{r\}_u; \{p\}Q\{s\}_u \rrbracket \Longrightarrow \{p\}Q\{r \wedge s\}_u$
by *rel-auto*

lemma *hoare-r-weaken-pre* [*hoare*]:
 $\{p\}Q\{r\}_u \Longrightarrow \{p \wedge q\}Q\{r\}_u$
 $\{q\}Q\{r\}_u \Longrightarrow \{p \wedge q\}Q\{r\}_u$
by *rel-auto+*

lemma *pre-str-hoare-r*:
assumes ' $p_1 \Rightarrow p_2$ ' **and** $\{p_2\}C\{q\}_u$
shows $\{p_1\}C\{q\}_u$
using *assms* **by** *rel-auto*

lemma *post-weak-hoare-r*:
assumes $\{p\}C\{q_2\}_u$ **and** $'q_2 \Rightarrow q_1'$
shows $\{p\}C\{q_1\}_u$
using *assms* **by** *rel-auto*

lemma *hoare-r-conseq*: $\llbracket 'p_1 \Rightarrow p_2'; \{p_2\}S\{q_2\}_u; 'q_2 \Rightarrow q_1' \rrbracket \Longrightarrow \{p_1\}S\{q_1\}_u$
by *rel-auto*

20.3 Assignment Laws

lemma *assigns-hoare-r [hoare-safe]*: $'p \Rightarrow \sigma \dagger q' \Longrightarrow \{p\}\langle\sigma\rangle_a\{q\}_u$
by *rel-auto*

lemma *assigns-backward-hoare-r*:
 $\{\sigma \dagger p\}\langle\sigma\rangle_a\{p\}_u$
by *rel-auto*

lemma *assign-floyd-hoare-r*:
assumes *vwb-lens* x
shows $\{p\} \text{ assign-}r \ x \ e \ \{\exists v \cdot p[\llbracket v \rrbracket/x] \wedge \&x =_u e[\llbracket v \rrbracket/x]\}_u$
using *assms*
by (*rel-auto, metis vwb-lens-wb wb-lens.get-put*)

lemma *skip-hoare-r [hoare-safe]*: $\{p\}II\{p\}_u$
by *rel-auto*

lemma *skip-hoare-impl-r [hoare-safe]*: $'p \Rightarrow q' \Longrightarrow \{p\}II\{q\}_u$
by *rel-auto*

20.4 Sequence Laws

lemma *seq-hoare-r*: $\llbracket \{p\}Q_1\{s\}_u ; \{s\}Q_2\{r\}_u \rrbracket \Longrightarrow \{p\}Q_1 ;; Q_2\{r\}_u$
by *rel-auto*

lemma *seq-hoare-invariant [hoare-safe]*: $\llbracket \{p\}Q_1\{p\}_u ; \{p\}Q_2\{p\}_u \rrbracket \Longrightarrow \{p\}Q_1 ;; Q_2\{p\}_u$
by *rel-auto*

lemma *seq-hoare-stronger-pre-1 [hoare-safe]*:
 $\llbracket \{p \wedge q\}Q_1\{p \wedge q\}_u ; \{p \wedge q\}Q_2\{q\}_u \rrbracket \Longrightarrow \{p \wedge q\}Q_1 ;; Q_2\{q\}_u$
by *rel-auto*

lemma *seq-hoare-stronger-pre-2 [hoare-safe]*:
 $\llbracket \{p \wedge q\}Q_1\{p \wedge q\}_u ; \{p \wedge q\}Q_2\{p\}_u \rrbracket \Longrightarrow \{p \wedge q\}Q_1 ;; Q_2\{p\}_u$
by *rel-auto*

lemma *seq-hoare-inv-r-2 [hoare]*: $\llbracket \{p\}Q_1\{q\}_u ; \{q\}Q_2\{q\}_u \rrbracket \Longrightarrow \{p\}Q_1 ;; Q_2\{q\}_u$
by *rel-auto*

lemma *seq-hoare-inv-r-3 [hoare]*: $\llbracket \{p\}Q_1\{p\}_u ; \{p\}Q_2\{q\}_u \rrbracket \Longrightarrow \{p\}Q_1 ;; Q_2\{q\}_u$
by *rel-auto*

20.5 Conditional Laws

lemma *cond-hoare-r [hoare-safe]*: $\llbracket \{b \wedge p\}S\{q\}_u ; \{\neg b \wedge p\}T\{q\}_u \rrbracket \Longrightarrow \{p\}S \triangleleft b \triangleright_r T\{q\}_u$
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*:
 $\bigwedge st P. \llbracket p \rrbracket P \llbracket q \rrbracket_u \implies \llbracket p \rrbracket F P \llbracket q \rrbracket_u$
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* ($\lceil p \rceil < \wedge (\lceil e \rceil <, \llbracket st \rrbracket)_u \in_u \llbracket R \rrbracket \Rightarrow \lceil q \rceil > st$)]
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 *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-def hoare-r-def, rule-tac lfp-lowerbound*) (*rel-auto*)

lemma *while-invr-hoare-r* [*hoare-safe*]:
assumes $\llbracket p \wedge b \rrbracket S \llbracket p \rrbracket_u$ $\langle pre \Rightarrow p' \langle \neg b \wedge p \rangle \Rightarrow post \rangle$
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*: ' $p \Rightarrow \text{invar}$ '
assumes *induct-step*: $\{ \text{invar} \wedge b \} C \{ \text{invar} \}_u$
shows $\{ p \} \text{while } b \text{ do } C \text{ od } \{ \neg b \wedge \text{invar} \}_u$
using *induct-step pre-str-hoare-r seq-step while-hoare-r* **by** *blast*

lemma *approx-chain*:

$(\bigcap n::\text{nat}. \lceil p \wedge v <_u \ll n \gg \rceil <) = \lceil p \rceil <$
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 $\bigwedge z::\text{nat}. \{ p \wedge b \wedge v =_u \ll z \gg \} S \{ p \wedge v <_u \ll z \gg \}_u$
shows $\{ p \} \text{while}_\perp b \text{ do } S \text{ od } \{ \neg b \wedge p \}_u$

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 seqr-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

thus *?thesis*

by (*simp add: hoare-r-def while-bot-def*)

qed

```

lemma while-vrt-hoare-r [hoare-safe]:
  assumes  $\bigwedge z::nat. \llbracket p \wedge b \wedge v =_u \llbracket z \rrbracket \rrbracket S \llbracket p \wedge v <_u \llbracket z \rrbracket \rrbracket_u \text{ 'pre} \Rightarrow p' \text{ ' } (\neg b \wedge p) \Rightarrow \text{post}'$ 
  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:  $\text{'pre} \Rightarrow p'$ 
  assumes induct-step:  $\bigwedge st. \llbracket b \wedge p \wedge e =_u \llbracket st \rrbracket \rrbracket Q \llbracket p \wedge (e, \llbracket st \rrbracket)_u \in_u \llbracket R \rrbracket \rrbracket_u$ 
  assumes PHI:  $\text{' } (\neg b \wedge p) \Rightarrow \text{post}'$ 
  shows  $\llbracket \text{pre} \rrbracket \text{while}_\perp b \text{ invr } p \text{ do } Q \text{ od} \llbracket \text{post} \rrbracket_u$ 
unfolding hoare-r-def while-inv-bot-def while-bot-def
proof (rule pre-weak-rel[of -  $\llbracket p \rrbracket_{<}$  ])
  from I0 show  $\llbracket \text{pre} \rrbracket_{<} \Rightarrow \llbracket p \rrbracket_{<}'$ 
  by rel-auto
  show  $(\llbracket p \rrbracket_{<} \Rightarrow \llbracket \text{post} \rrbracket_{>}) \sqsubseteq (\mu X \cdot Q ;; X \triangleleft b \triangleright_r II)$ 
  proof (rule mu-rec-total-utp-rule[where  $e=e$ , OF WF])
  show Monotonic  $(\lambda X. Q ;; X \triangleleft b \triangleright_r II)$ 
  by (simp add: closure)
  have induct-step':  $\bigwedge st. (\llbracket b \wedge p \wedge e =_u \llbracket st \rrbracket \rrbracket_{<} \Rightarrow (\llbracket p \wedge (e, \llbracket st \rrbracket)_u \in_u \llbracket R \rrbracket \rrbracket_{>})) \sqsubseteq Q$ 
  using induct-step by rel-auto
  with PHI
  show  $\bigwedge st. (\llbracket p \rrbracket_{<} \wedge \llbracket e \rrbracket_{<} =_u \llbracket st \rrbracket \Rightarrow \llbracket \text{post} \rrbracket_{>}) \sqsubseteq Q ;; (\llbracket p \rrbracket_{<} \wedge (\llbracket e \rrbracket_{<}, \llbracket st \rrbracket)_u \in_u \llbracket R \rrbracket \Rightarrow \llbracket \text{post} \rrbracket_{>})$ 
 $\triangleleft b \triangleright_r II$ 
  by (rel-auto)
  qed
qed

```

20.8 Frame Rules

Frame rule: If starting S in a state satisfying *pestablishe* q in the final state, then we can insert an invariant predicate r when S is framed by a , provided that r does not refer to variables in the frame, and q does not refer to variables outside the frame.

```

lemma frame-hoare-r:
  assumes vwb-lens  $a\ a \# r\ a \triangleleft q \llbracket p \rrbracket P \llbracket q \rrbracket_u$ 
  shows  $\llbracket p \wedge r \rrbracket a: [P] \llbracket q \wedge r \rrbracket_u$ 
  using assms
  by (rel-auto, metis)

```

```

lemma frame-strong-hoare-r [hoare-safe]:
  assumes vwb-lens  $a\ a \# r\ a \triangleleft q \llbracket p \wedge r \rrbracket S \llbracket q \rrbracket_u$ 
  shows  $\llbracket p \wedge r \rrbracket a: [S] \llbracket q \wedge r \rrbracket_u$ 
  using assms by (rel-auto, metis)

```

```

lemma frame-hoare-r' [hoare-safe]:
  assumes vwb-lens  $a\ a \# r\ a \triangleleft q \llbracket r \wedge p \rrbracket S \llbracket q \rrbracket_u$ 
  shows  $\llbracket r \wedge p \rrbracket a: [S] \llbracket r \wedge q \rrbracket_u$ 
  using assms
  by (simp add: frame-strong-hoare-r utp-pred-laws.inf commute)

```

```

lemma antiframe-hoare-r:
  assumes vwb-lens a a  $\Vdash$  r a  $\sharp$  q  $\{p\}P\{q\}_u$ 
  shows  $\{p \wedge r\} a : \llbracket P \rrbracket \{q \wedge r\}_u$ 
  using assms by (rel-auto, metis)

lemma antiframe-strong-hoare-r:
  assumes vwb-lens a a  $\Vdash$  r a  $\sharp$  q  $\{p \wedge r\}P\{q\}_u$ 
  shows  $\{p \wedge r\} a : \llbracket P \rrbracket \{q \wedge r\}_u$ 
  using assms by (rel-auto, metis)

```

```

lemma antiframe-intro:
  assumes
    vwb-lens g vwb-lens g' vwb-lens l l  $\bowtie$  g g'  $\subseteq_L$  g
     $\{\&g', \&l\} : [C] = C \{p\} C \{q\}_u \text{ 'r} \Rightarrow p'$ 
  shows  $\{r\} l : \llbracket C \rrbracket \{(\exists l \cdot q) \wedge (\exists g' \cdot r)\}_u$ 
  using assms
  apply (rel-auto, simp-all add: lens-defs)
  apply metis
  apply (rename-tac Z a b)
  apply (rule-tac x=get g' a in exI)
oops

```

end

21 Weakest Precondition Calculus

```

theory utp-wp
imports utp-hoare
begin

```

A very quick implementation of wp – more laws still needed!

named-theorems *wp*

method *wp-tac* = (*simp add: wp*)

```

consts
  uwp :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'c (infix wp 60)

```

definition *wp-upred* :: (' α , ' β) *urel* \Rightarrow ' β *cond* \Rightarrow ' α *cond* **where**
wp-upred *Q* *r* = $\lfloor \neg (Q \;; (\neg \lceil r \rceil_{<})) \rfloor :: ('\alpha, '\beta) \text{ urel} \rfloor_{<}$

```

adhoc-overloading
  uwp wp-upred

```

declare *wp-upred-def* [*urel-defs*]

```

lemma wp-true [wp]: p wp true = true
  by (rel-simp)

```

```

theorem wp-assigns-r [wp]:
   $\langle \sigma \rangle_a \text{ wp } r = \sigma \dagger r$ 
  by rel-auto

```


theorem *wp-skip-r* [wp]:

$\text{II } wp \ r = r$

by *rel-auto*

theorem *wp-abort* [wp]:

$r \neq \text{true} \implies \text{true } wp \ r = \text{false}$

by *rel-auto*

theorem *wp-conj* [wp]:

$P \ wp \ (q \wedge r) = (P \ wp \ q \wedge P \ wp \ r)$

by *rel-auto*

theorem *wp-seq-r* [wp]: $(P \ ; \ Q) \ wp \ r = P \ wp \ (Q \ wp \ r)$

by *rel-auto*

theorem *wp-cond* [wp]: $(P \triangleleft b \triangleright_r Q) \ wp \ r = ((b \Rightarrow P \ wp \ r) \wedge ((\neg b) \Rightarrow Q \ wp \ r))$

by *rel-auto*

lemma *wp-USUP-pre* [wp]: $P \ wp \ (\bigsqcup_{i \in \{0..n\}} Q(i)) = (\bigsqcup_{i \in \{0..n\}} P \ wp \ Q(i))$

by (*rel-auto*)

theorem *wp-hoare-link*:

$\{p\} Q \{r\}_u \longleftrightarrow (Q \ wp \ r \sqsubseteq p)$

by *rel-auto*

If two programs have the same weakest precondition for any postcondition then the programs are the same.

theorem *wp-eq-intro*: $\llbracket \bigwedge r. P \ wp \ r = Q \ wp \ r \rrbracket \implies P = Q$

by (*rel-auto robust, fastforce+*)

end

22 Strong Postcondition Calculus

theory *utp-sp*

imports *utp-wp*

begin

named-theorems *sp*

method *sp-tac* = (*simp add: sp*)

consts

usp :: $'a \Rightarrow 'b \Rightarrow 'c$ (**infix** *sp* 60)

definition *sp-upred* :: $'\alpha \text{ cond} \Rightarrow (' \alpha, ' \beta) \text{ urel} \Rightarrow ' \beta \text{ cond}$ **where**

sp-upred *p* *Q* = $\llbracket ([p]_{>} \ ; \ Q) :: (' \alpha, ' \beta) \text{ urel} \rrbracket_{>}$

adhoc-overloading

usp sp-upred

declare *sp-upred-def* [*upred-defs*]

lemma *sp-false* [*sp*]: $p \ sp \ \text{false} = \text{false}$

by (*rel-simp*)

lemma *sp-true* [*sp*]: $q \neq \text{false} \implies q \text{ sp } \text{true} = \text{true}$
by (*rel-auto*)

lemma *sp-assigns-r* [*sp*]:
 $\text{vwb-lens } x \implies (p \text{ sp } x := e) = (\exists v \cdot p \llbracket \langle v \rangle / x \rrbracket \wedge \&x =_u e \llbracket \langle v \rangle / x \rrbracket)$
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*:
 $\langle p \text{ sp } C \Rightarrow Q \rangle \implies \{p\} C \{Q\}_u$
by *rel-blast*

lemma *sp-so*:
 $\langle p \text{ sp } C \Rightarrow Q \rangle = \{p\} C \{Q\}_u$
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*: $\llbracket \bigwedge r. r \text{ sp } P = r \text{ sp } Q \rrbracket \implies P = Q$
by (*rel-auto robust, fastforce+*)

lemma *wp-sp-sym*:
 $\langle \text{prog wp } (\text{true sp prog}) \rangle$
by *rel-auto*

lemma *it-is-pre-condition*: $\{C \text{ wp } Q\} C \{Q\}_u$
by *rel-blast*

lemma *it-is-the-weakest-pre*: $\langle P \Rightarrow C \text{ wp } Q \rangle = \{P\} C \{Q\}_u$
by *rel-blast*

lemma *s-pre*: $\langle P \Rightarrow C \text{ wp } Q \rangle = \{P\} C \{Q\}_u$
by *rel-blast*

end

23 Concurrent Programming

theory *utp-concurrency*
imports
utp-hoare
utp-rel
utp-tactics

begin

In this theory we describe the UTP scheme for concurrency, *parallel-by-merge*, which provides a general parallel operator parametrised by a “merge predicate” that explains how to merge the after states of the composed predicates. It can thus be applied to many languages and concurrency schemes, with this theory providing a number of generic laws. The operator is explained in more detail in Chapter 7 of the UTP book [14].

23.1 Variable Renamings

In parallel-by-merge constructions, a merge predicate defines the behaviour following execution of 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 $\text{pre-uvar} :: ('a \implies '\alpha) \Rightarrow ('a \implies (' \alpha, '\beta_0, '\beta_1) \text{ mrg})$ **where**
 $[\text{upred-defs}]: \text{pre-uvar } x = x ;_L \text{ mrg-prior}$

definition $\text{left-uvar} :: ('a \implies '\beta_0) \Rightarrow ('a \implies (' \alpha, '\beta_0, '\beta_1) \text{ mrg})$ **where**
 $[\text{upred-defs}]: \text{left-uvar } x = x ;_L \text{ mrg-left}$

definition $\text{right-uvar} :: ('a \implies '\beta_1) \Rightarrow ('a \implies (' \alpha, '\beta_0, '\beta_1) \text{ mrg})$ **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

$\text{-svarpre} :: \text{svid} \Rightarrow \text{svid} \text{ } (-_< [995] \text{ } 995)$
 $\text{-svarleft} :: \text{svid} \Rightarrow \text{svid} \text{ } (0_- [995] \text{ } 995)$
 $\text{-svarright} :: \text{svid} \Rightarrow \text{svid} \text{ } (1_- [995] \text{ } 995)$

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

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

lemma *pre-uvar [simp]: vwb-lens x \implies vwb-lens (pre-uvar x)*
by (*simp add: pre-uvar-def*)

lemma *left-uvar-mwb [simp]: mwb-lens x \implies mwb-lens (left-uvar x)*
by (*simp add: left-uvar-def*)

lemma *right-uvar-mwb [simp]: mwb-lens x \implies mwb-lens (right-uvar x)*
by (*simp add: right-uvar-def*)

lemma *pre-uvar-mwb [simp]: mwb-lens x \implies mwb-lens (pre-uvar x)*
by (*simp add: pre-uvar-def*)

We prove various independence laws about the variable classes.

lemma *left-uvar-indep-right-uvar [simp]:*
left-uvar x \bowtie right-uvar y
by (*simp add: left-uvar-def right-uvar-def lens-comp-assoc[THEN sym]*)

lemma *left-uvar-indep-pre-uvar [simp]:*
left-uvar x \bowtie pre-uvar y
by (*simp add: left-uvar-def pre-uvar-def*)

lemma *left-uvar-indep-left-uvar [simp]:*
x \bowtie y \implies left-uvar x \bowtie left-uvar y
by (*simp add: left-uvar-def*)

lemma *right-uvar-indep-left-uvar [simp]:*
right-uvar x \bowtie left-uvar y
by (*simp add: lens-indep-sym*)

lemma *right-uvar-indep-pre-uvar [simp]:*
right-uvar x \bowtie pre-uvar y
by (*simp add: right-uvar-def pre-uvar-def*)

lemma *right-uvar-indep-right-uvar [simp]:*
x \bowtie y \implies right-uvar x \bowtie right-uvar y
by (*simp add: right-uvar-def*)

lemma *pre-uvar-indep-left-uvar [simp]:*
pre-uvar x \bowtie left-uvar y
by (*simp add: lens-indep-sym*)

lemma *pre-uvar-indep-right-uvar [simp]:*
pre-uvar x \bowtie right-uvar y
by (*simp add: lens-indep-sym*)

lemma *pre-uvar-indep-pre-uvar [simp]:*
x \bowtie y \implies pre-uvar x \bowtie pre-uvar y
by (*simp add: pre-uvar-def*)

23.2 Merge Predicates

A merge predicate is a relation whose input has three parts: the prior variables, the output variables of the left predicate, and the output of the right predicate.

type-synonym *' α merge = ((' α , ' α , ' α) mrg, ' α) urel*

skip is the merge predicate which ignores the output of both parallel predicates

definition $skip_m :: 'a \text{ merge where}$
 $[upred-defs]: skip_m = (\$v' =_u \$v_<)$

swap is a predicate that the swaps the left and right indices; it is used to specify commutativity of the parallel operator

definition $swap_m :: (('a, 'b, 'b) \text{ mrg}) \text{ hrel where}$
 $[upred-defs]: swap_m = (0-v, 1-v) := (\&1-v, \&0-v)$

A symmetric merge is one for which swapping the order of the merged concurrent predicates has no effect. We represent this by the following healthiness condition that states that $swap_m$ is a left-unit.

abbreviation $SymMerge :: 'a \text{ merge} \Rightarrow 'a \text{ merge where}$
 $SymMerge(M) \equiv (swap_m ;; M)$

23.3 Separating Simulations

$U0$ and $U1$ are relations modify the variables of the input state-space such that they become indexed with 0 and 1, respectively.

definition $U0 :: ('b_0, ('a, 'b_0, 'b_1) \text{ mrg}) \text{ urel where}$
 $[upred-defs]: U0 = (\$0-v' =_u \$v)$

definition $U1 :: ('b_1, ('a, 'b_0, 'b_1) \text{ mrg}) \text{ urel where}$
 $[upred-defs]: U1 = (\$1-v' =_u \$v)$

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 $U0\alpha \text{ where } [upred-defs]: U0\alpha = (1_L \times_L \text{ mrg-left})$

definition $U1\alpha \text{ where } [upred-defs]: U1\alpha = (1_L \times_L \text{ mrg-right})$

We then create the following intuitive syntax for separating simulations.

abbreviation $U0\text{-alpha-lift } ([\cdot]_0) \text{ where } [P]_0 \equiv P \oplus_p U0\alpha$

abbreviation $U1\text{-alpha-lift } ([\cdot]_1) \text{ where } [P]_1 \equiv P \oplus_p U1\alpha$

$[P]_0$ is predicate P where all variables are indexed by 0, and $[P]_1$ is where all variables are indexed by 1. We can thus equivalently express separating simulations using alphabet extrusion.

lemma $U0\text{-as-alpha}: (P ;; U0) = [P]_0$
by $(rel\text{-auto})$

lemma $U1\text{-as-alpha}: (P ;; U1) = [P]_1$
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 $U1\alpha$ -vwb-lens [simp]: vwb-lens $U1\alpha$
 by (simp add: $U1\alpha$ -def id-vwb-lens prod-vwb-lens)

lemma $U0\alpha$ -indep-right-uvar [simp]: vwb-lens $x \implies U0\alpha \bowtie \text{out-var } (\text{right-uvar } x)$
 by (force intro: plus-pres-lens-indep fst-snd-lens-indep lens-indep-left-comp
 simp add: $U0\alpha$ -def right-uvar-def out-var-def prod-as-plus lens-comp-assoc[THEN sym])

lemma $U1\alpha$ -indep-left-uvar [simp]: vwb-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: $U1\alpha$ -def left-uvar-def out-var-def prod-as-plus lens-comp-assoc[THEN sym])

lemma $U0$ -alpha-lift-bool-subst [usubst]:
 $\sigma(\$0-x' \mapsto_s \text{true}) \uparrow \lceil P \rceil_0 = \sigma \uparrow \lceil P \llbracket \text{true}/\$x' \rrbracket \rceil_0$
 $\sigma(\$0-x' \mapsto_s \text{false}) \uparrow \lceil P \rceil_0 = \sigma \uparrow \lceil P \llbracket \text{false}/\$x' \rrbracket \rceil_0$
 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 $U0\alpha$ -comp-in-var [alpha]: $(\text{in-var } x) ;_L U0\alpha = \text{in-var } x$
 by (simp add: $U0\alpha$ -def alpha-in-var in-var-prod-lens pre-uvar-def)

lemma $U0\alpha$ -comp-out-var [alpha]: $(\text{out-var } x) ;_L U0\alpha = \text{out-var } (\text{left-uvar } x)$
 by (simp add: $U0\alpha$ -def alpha-out-var id-wb-lens left-uvar-def out-var-prod-lens)

lemma $U1\alpha$ -comp-in-var [alpha]: $(\text{in-var } x) ;_L U1\alpha = \text{in-var } x$
 by (simp add: $U1\alpha$ -def alpha-in-var in-var-prod-lens pre-uvar-def)

lemma $U1\alpha$ -comp-out-var [alpha]: $(\text{out-var } x) ;_L U1\alpha = \text{out-var } (\text{right-uvar } x)$
 by (simp add: $U1\alpha$ -def alpha-out-var id-wb-lens right-uvar-def out-var-prod-lens)

23.4 Associative Merges

Associativity of a merge means that if we construct a three way merge from a two way merge and then rotate the three inputs of the merge to the left, then we get exactly the same three way merge back.

We first construct the operator that constructs the three way merge by effectively wiring up the two way merge in an appropriate way.

definition *ThreeWayMerge* :: $'\alpha \text{ merge} \Rightarrow (('\alpha, '\alpha, ('\alpha, '\alpha, '\alpha) \text{ mrg}) \text{ mrg}, '\alpha) \text{ urel } (\mathbf{M3}'(-))$ **where**
 $[upred-defs]: \text{ThreeWayMerge } M = ((\$0-\mathbf{v}' =_u \$0-\mathbf{v} \wedge \$1-\mathbf{v}' =_u \$1-0-\mathbf{v} \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}_{<}) ;; M ;;$
 $U0 \wedge \$1-\mathbf{v}' =_u \$1-1-\mathbf{v} \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}_{<}) ;; M$

The next definition rotates the inputs to a three way merge to the left one place.

abbreviation *rotate_m* **where** $\text{rotate}_m \equiv (0-\mathbf{v}, 1-0-\mathbf{v}, 1-1-\mathbf{v}) := (\&1-0-\mathbf{v}, \&1-1-\mathbf{v}, \&0-\mathbf{v})$

Finally, a merge is associative if rotating the inputs does not effect the output.

definition *AssocMerge* :: $'\alpha \text{ merge} \Rightarrow \text{bool}$ **where**
 $[upred-defs]: \text{AssocMerge } M = (\text{rotate}_m ;; \mathbf{M3}(M) = \mathbf{M3}(M))$

23.5 Parallel Operators

We implement the following useful abbreviation for separating of two parallel processes and copying of the before variables, all to act as input to the merge predicate.

abbreviation *par-sep* (**infixr** \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
 $\text{par-by-merge} :: ('\alpha, '\beta) \text{ urel} \Rightarrow (('\alpha, '\beta, '\gamma) \text{ mrg}, '\delta) \text{ urel} \Rightarrow (''\alpha, '\gamma) \text{ urel} \Rightarrow (''\alpha, '\delta) \text{ urel}$
 $(- \parallel - \text{ } [85, 0, 86] \text{ } 85)$
where $[upred-defs]: P \parallel_M Q = (P \parallel_s Q ;; M)$

lemma *par-by-merge-alt-def*: $P \parallel_M Q = ([P]_0 \wedge [Q]_1 \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}) ;; M$
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*)

23.6 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*)

23.7 Substitution laws

Substitution is a little tricky because when we push the expression through the composition operator the alphabet of the expression must also change. Consequently for now we only support literal substitution, though this could be generalised with suitable alphabet coercsions. We need quite a number of variants to support this which are below.

lemma *U0-seq-subst*: $(P \;; \; U0) \llbracket \langle v \rangle / \$0 - x \rrbracket = (P \llbracket \langle v \rangle / \$x \rrbracket \;; \; U0)$
by (*rel-auto*)

lemma *U1-seq-subst*: $(P \;; \; U1) \llbracket \langle v \rangle / \$1 - x \rrbracket = (P \llbracket \langle v \rangle / \$x \rrbracket \;; \; U1)$
by (*rel-auto*)

lemma *lit-pbm-subst* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P \ Q \ M \ \sigma. \ \sigma(\$x \mapsto_s \langle v \rangle) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket \langle v \rangle / \$x \rrbracket) \parallel_{M \llbracket \langle v \rangle / \$x \rrbracket} (Q \llbracket \langle v \rangle / \$x \rrbracket))$
 $\bigwedge P \ Q \ M \ \sigma. \ \sigma(\$x' \mapsto_s \langle v \rangle) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M \llbracket \langle v \rangle / \$x' \rrbracket} Q)$
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*)+

23.8 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 $(P \parallel_M Q) \parallel_M R = ((P ;; U0) \wedge (Q ;; U0 ;; U1) \wedge (R ;; U1 ;; U1) \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}) ;; \mathbf{M}\beta(M)$
by (*rel-blast*)

also have $\dots = ((P ;; U0) \wedge (Q ;; U0 ;; U1) \wedge (R ;; U1 ;; U1) \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}) ;; rotate_m ;; \mathbf{M}\beta(M)$
using *AssocMerge-def assms(2)* **by** *force*

also have $\dots = ((Q ;; U0) \wedge (R ;; U0 ;; U1) \wedge (P ;; U1 ;; U1) \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}) ;; \mathbf{M}\beta(M)$
by (*rel-blast*)

also have $\dots = (Q \parallel_M R) \parallel_M P$
by (*rel-blast*)

also have $\dots = P \parallel_M (Q \parallel_M R)$
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*:

$(\bigcap_{i \in I} i \cdot P(i)) \parallel_M Q = (\bigcap_{i \in I} i \cdot P(i) \parallel_M Q)$
by (*rel-auto*)

theorem *par-by-merge-USUP-ind-left*:

$(\bigcap i \cdot P(i)) \parallel_M Q = (\bigcap i \cdot P(i) \parallel_M Q)$
by (*rel-auto*)

theorem *par-by-merge-USUP-mem-right*:

$P \parallel_M (\bigcap_{i \in I} i \cdot Q(i)) = (\bigcap_{i \in I} i \cdot P \parallel_M Q(i))$
by (*rel-auto*)

theorem *par-by-merge-USUP-ind-right*:

$P \parallel_M (\bigcap i \cdot Q(i)) = (\bigcap i \cdot P \parallel_M Q(i))$
by (*rel-auto*)

23.9 Example: Simple State-Space Division

The following merge predicate divides the state space using a pair of independent lenses.

definition *StateMerge* :: $('a \implies 'a) \Rightarrow ('b \implies 'a) \Rightarrow 'a \text{ merge } (M[-]_{\sigma})$ **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*: $a \bowtie b \implies (\text{swap}_m ;; M[a|b]_\sigma) = M[b|a]_\sigma$
by (*rel-auto*, *simp-all add: lens-indep-comm*)

abbreviation *StateParallel* :: $'\alpha \text{ hrel} \Rightarrow ('a \implies '\alpha) \Rightarrow ('b \implies '\alpha) \Rightarrow '\alpha \text{ hrel} \Rightarrow '\alpha \text{ hrel} (-|-|_\sigma -$
 $[85,0,0,86] \ 86)$

where $P \mid a|b|_\sigma Q \equiv P \parallel_{M[a|b]_\sigma} Q$

lemma *StateParallel-commute*: $a \bowtie b \implies P \mid a|b|_\sigma Q = Q \mid b|a|_\sigma P$
by (*metis par-by-merge-commute-swap swap-StateMerge*)

lemma *StateParallel-form*:

$P \mid a|b|_\sigma Q = (\exists (st_0, st_1) \cdot P[\ll st_0 \gg / \$\mathbf{v}'] \wedge Q[\ll st_1 \gg / \$\mathbf{v}'] \wedge \$\mathbf{v}' =_u (\$ \mathbf{v} \oplus \ll st_0 \gg \text{ on } \&a) \oplus$
 $\ll st_1 \gg \text{ on } \&b)$
by (*rel-auto*)

lemma *StateParallel-form'*:

assumes *vwb-lens a vwb-lens b a* $\bowtie b$
shows $P \mid a|b|_\sigma Q = \{\&a, \&b\} : [(P \mid_v \{\$ \mathbf{v}, \$a'\}) \wedge (Q \mid_v \{\$ \mathbf{v}, \$b'\})]$
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 $\{\&a, \&b\} : [P \mid a|b|_\sigma Q] = P \mid a|b|_\sigma Q$
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 $\llbracket c \rrbracket P \llbracket d_1 \rrbracket_u \llbracket c \rrbracket Q \llbracket d_2 \rrbracket_u$ $a \bowtie b$ $a \Vdash d_1$ $b \Vdash d_2$
shows $\llbracket c \rrbracket P \mid a|b|_\sigma Q \llbracket d_1 \wedge d_2 \rrbracket_u$

proof –

— Parallelise the specification

from *assms(4,5)*

have $1: (\llbracket c \rrbracket < \Rightarrow \llbracket d_1 \wedge d_2 \rrbracket >) \sqsubseteq (\llbracket c \rrbracket < \Rightarrow \llbracket d_1 \rrbracket >) \mid a|b|_\sigma (\llbracket c \rrbracket < \Rightarrow \llbracket d_2 \rrbracket >) \text{ (is ?lhs } \sqsubseteq \text{ ?rhs)}$
by (*simp add: StateParallel-form, rel-auto, metis assms(3) lens-indep-comm*)

— Prove Hoare rule by monotonicity of parallelism

have $2: \text{?rhs} \sqsubseteq P \mid a|b|_\sigma Q$

proof (*rule par-by-merge-mono*)

show $(\llbracket c \rrbracket < \Rightarrow \llbracket d_1 \rrbracket >) \sqsubseteq P$

using *assms(1) hoare-r-def* **by** *auto*

show $(\llbracket c \rrbracket < \Rightarrow \llbracket d_2 \rrbracket >) \sqsubseteq Q$

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  $\Downarrow$  d1 b  $\Downarrow$  d2 a  $\#$  c1 b  $\#$  c1  $\{c_1 \wedge c_2\}P\{d_1\}_u \{c_1 \wedge c_2\}Q\{d_2\}_u$ 
  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  $\{c_1 \wedge c_2\}\{\&a, \&b\}:[P \mid a \mid b \mid_\sigma Q]\{c_1 \wedge d_1 \wedge d_2\}_u$ 
    by (auto intro!: frame-hoare-r' StateParallel-hoare simp add: assms unrest plus-vwb-lens)
  thus ?thesis
    by (simp add: StateParallel-frame assms)
qed

end

```

24 Relational Operational Semantics

```

theory utp-rel-opsem
imports
  utp-rel-laws
  utp-hoare
begin

```

This theory uses the laws of relational calculus to create a basic operational semantics. It is based on Chapter 10 of the UTP book [14].

```

fun trel :: 'α usubst × 'α hrel ⇒ 'α usubst × 'α hrel ⇒ bool (infix →u 85) where
  (σ, P) →u (ρ, Q)  $\longleftrightarrow$  ( $\langle \sigma \rangle_a \;; P$ )  $\sqsubseteq$  ( $\langle \rho \rangle_a \;; Q$ )

```

```

lemma trans-trel:
   $\llbracket (\sigma, P) \rightarrow_u (\rho, Q); (\rho, Q) \rightarrow_u (\varphi, R) \rrbracket \implies (\sigma, P) \rightarrow_u (\varphi, R)$ 
by auto

```

```

lemma skip-trel: (σ, II) →u (σ, II)
by simp

```

```

lemma assigns-trel: (σ,  $\langle \rho \rangle_a$ ) →u (ρ ∘ σ, II)
by (simp add: assigns-comp)

```

```

lemma assign-trel:
  (σ, x := v) →u (σ( $\&x \mapsto_s \sigma \uparrow v$ ), 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*:
 $(\sigma, P \sqcap Q) \rightarrow_u (\sigma, P)$
by (*metis* (*no-types*, *hide-lams*) *disj-comm disj-upred-def semilattice-sup-class.sup.absorb-iff1 semilattice-sup-class.sup.l*
seqr-or-distr trel.simps)

lemma *nondet-right-trel*:
 $(\sigma, P \sqcap Q) \rightarrow_u (\sigma, Q)$
by (*simp add: seqr-mono*)

lemma *rcond-true-trel*:
assumes $\sigma \dagger b = \text{true}$
shows $(\sigma, P \triangleleft b \triangleright_r Q) \rightarrow_u (\sigma, P)$
using *assms*
by (*simp add: assigns-r-comp usubst alpha cond-unit-T*)

lemma *rcond-false-trel*:
assumes $\sigma \dagger b = \text{false}$
shows $(\sigma, P \triangleleft b \triangleright_r Q) \rightarrow_u (\sigma, Q)$
using *assms*
by (*simp add: assigns-r-comp usubst alpha cond-unit-F*)

lemma *while-true-trel*:
assumes $\sigma \dagger b = \text{true}$
shows $(\sigma, \text{while } b \text{ do } P \text{ od}) \rightarrow_u (\sigma, P ;; \text{while } b \text{ do } P \text{ od})$
by (*metis assms rcond-true-trel while-unfold*)

lemma *while-false-trel*:
assumes $\sigma \dagger b = \text{false}$
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*:
 $\llbracket p \rrbracket Q \llbracket r \rrbracket_u = (\forall \sigma_0 \sigma_1. \sigma_0 \dagger p' \wedge (\sigma_0, Q) \rightarrow_u (\sigma_1, II) \longrightarrow \sigma_1 \dagger r')$
apply (*rel-auto*)
apply (*rename-tac a b*)
apply (*drule-tac x=\lambda -. a in spec, simp*)
apply (*drule-tac x=\lambda -. b in spec, simp*)
done

declare *trel.simps* [*simp del*]

end

25 Local Variables

theory *utp-local*
imports
utp-rel-laws
utp-meta-subst
utp-theory
begin

25.1 Preliminaries

The following type is used to augment that state-space with a stack of local variables represented as a list in the special variable *store*. Local variables will be represented by pushing variables onto the stack, and popping them off after use. The element type of the stack is *'u* which corresponds to a suitable injection universe.

alphabet *'u local* =
store :: *'u list*

State-space with a countable universe for local variables.

type-synonym *'a clocal* = (*nat*, *'a*) *local-scheme*

The following predicate wraps the relation with assumptions that the stack has a particular size before and after execution.

definition *local-num* **where**
local-num *n P* = [*#_u*(&*store*) =_{*u*} <<*n*>>][⊤] ;; *P* ;; [*#_u*(&*store*) =_{*u*} <<*n*>>][⊤]

declare *inj-univ.from-univ-def* [*upred-defs*]
declare *inj-univ.to-univ-lens-def* [*upred-defs*]
declare *nat-inj-univ-def* [*upred-defs*]

25.2 State Primitives

The following record is used to characterise the UTP theory specific operators we require in order to create the local variable operators.

record (*'α*, *'s*) *state-prim* =

— The first field states where in the alphabet *'α* the user state-space type is *'s* is located with the form of a lens.

sstate :: *'s* ⇒ *'α* (*s1*)

— The second field is the theory's substitution operator. It takes a substitution over the state-space type and constructs a homogeneous assignment relation.

sassigns :: *'s usubst* ⇒ *'α hrel* (<->₁)

syntax

-sstate :: *logic* ⇒ *svid* (*s1*)

translations

-sstate *T* => *CONST sstate T*

The following record type adds an injection universe *'u* to the above operators. This is needed because the stack has a homogeneous type into which we must inject type variable bindings. The universe can be any Isabelle type, but must satisfy the axioms of the locale *inj-univ*, which broadly shows the injectable values permitted.

record (*'α*, *'s*, *'u*, *'a*) *local-prim* = (*'α*, (*'u*, *'s*) *local-scheme*) *state-prim* +
inj-local :: (*'a*, *'u*) *inj-univ*

The following locales give the assumptions required of the above signature types. The first gives the defining axioms for state-spaces. State-space lens *s* must be a very well-behaved lens, and

sequential composition of assignments corresponds to functional composition of the underlying substitutions. TODO: We might also need operators to properly handle framing in the future.

```

locale utp-state =
  fixes S (structure)
  assumes vwb-lens s
  and passigns-comp:  $\langle \sigma \rangle ;; \langle \varrho \rangle = \langle \varrho \circ \sigma \rangle$ 

```

The next locale combines the axioms of a state-space and an injection universe structure. It then uses the required constructs to create the local variable operators.

```

locale utp-local-state = utp-state S + inj-univ inj-local S for S :: ('α, 's, 'u::two, 'a) local-prim
(structure)
begin

```

The following two operators represent opening and closing a variable scope, which is implemented by pushing an arbitrary initial value onto the stack, and popping it off, respectively.

```

definition var-open :: 'α hrel (openv) where
var-open =  $(\prod v \cdot \langle [store \mapsto_s (\&store \hat{^}_u \langle \ll v \gg )] \rangle)$ 

```

```

definition var-close :: 'α hrel (closev) where
var-close =  $\langle [store \mapsto_s front_u(\&store) \triangleleft \#_u(\&store) >_u 0 \triangleright \&store] \rangle$ 

```

The next operator is an expression that returns a lens pointing to the top of the stack. This is effectively a dynamic lens, since where it points to depends on the initial number of variables on the stack.

```

definition top-var :: ('a  $\implies$  ('u, 'b) local-scheme, 'α) uexpr (topv) where
top-var =  $\ll \lambda l. to-univ-lens ;_L list-lens l ;_L store \gg (\#_u(\&s:store) - 1)_a$ 

```

Finally, we combine the above operators to represent variable scope. This is a kind of binder which takes a homogeneous relation, parametric over a lens, and returns a relation. It simply opens the variable scope, substitutes the top variable into the body, and then closes the scope afterwards.

```

definition var-scope :: (('a  $\implies$  ('u, 's) local-scheme)  $\Rightarrow$  'α hrel)  $\Rightarrow$  'α hrel where
var-scope f = openv ;; f(x)  $\llbracket x \mapsto [top_v]_{<} \rrbracket$  ;; closev

```

end

notation *utp-local-state.var-open* (*open*[*-*])

notation *utp-local-state.var-close* (*close*[*-*])

notation *utp-local-state.var-scope* (*V*[*-*, / *-*])

notation *utp-local-state.top-var* (*top*[*-*])

syntax

-var-scope :: *logic* \Rightarrow *id* \Rightarrow *logic* \Rightarrow *logic* (*var*[*-*] - · - [0, 0, 10] 10)

-var-scope-type :: *logic* \Rightarrow *id* \Rightarrow *type* \Rightarrow *logic* \Rightarrow *logic* (*var*[*-*] - :: - · - [0, 0, 0, 10] 10)

translations

-var-scope *T* *x* *P* == *CONST* *utp-local-state.var-scope* *T* ($\lambda x. P$)

-var-scope-type *T* *x* *t* *P* => *CONST* *utp-local-state.var-scope* *T* (*-abs* (*-constrain* *x* (*-uvar-ty* *t*)) *P*)

Next, we prove a collection of important generci laws about variable scopes using the axioms defined above.

context *utp-local-state*

begin

lemma *var-open-commute*:

$\llbracket x \bowtie \text{store}; \text{store} \# v \rrbracket \implies \langle [x \mapsto_s v] \rangle ;; \text{open}_v = \text{open}_v ;; \langle [x \mapsto_s v] \rangle$
by (*simp add: var-open-def passigns-comp seq-UINF-distl' seq-UINF-distr' usubst unrest lens-indep-sym, simp add: usubst-upd-comm*)

lemma *var-close-commute*:

$\llbracket x \bowtie \text{store}; \text{store} \# v \rrbracket \implies \langle [x \mapsto_s v] \rangle ;; \text{close}_v = \text{close}_v ;; \langle [x \mapsto_s v] \rangle$
by (*simp add: var-close-def passigns-comp seq-UINF-distl' seq-UINF-distr' usubst unrest lens-indep-sym, simp add: usubst-upd-comm*)

lemma *var-open-close-lemma*:

$[\text{store} \mapsto_s \text{front}_u(\&\text{store} \hat{\ }_u \langle \ll v \gg \rangle) \triangleleft 0 <_u \#_u(\&\text{store} \hat{\ }_u \langle \ll v \gg \rangle) \triangleright \&\text{store} \hat{\ }_u \langle \ll v \gg \rangle] = \text{id}$
by (*rel-auto*)

lemma *var-open-close*: $\text{open}_v ;; \text{close}_v = \langle \text{id} \rangle$

by (*simp add: var-open-def var-close-def seq-UINF-distr' passigns-comp usubst var-open-close-lemma*)

lemma *var-scope-skip*: $(\text{var}[S] \ x \cdot \langle \text{id} \rangle) = \langle \text{id} \rangle$

by (*simp add: var-scope-def var-open-def var-close-def seq-UINF-distr' passigns-comp var-open-close-lemma usubst*)

lemma *var-scope-nonlocal-left*:

$\llbracket x \bowtie \text{store} ; \text{store} \# v \rrbracket \implies \langle [x \mapsto_s v] \rangle ;; (\text{var}[S] \ y \cdot P(y)) = (\text{var}[S] \ y \cdot \langle [x \mapsto_s v] \rangle ;; P(y))$
oops

end

declare *utp-local-state.var-open-def* [*upred-defs*]

declare *utp-local-state.var-close-def* [*upred-defs*]

declare *utp-local-state.top-var-def* [*upred-defs*]

declare *utp-local-state.var-scope-def* [*upred-defs*]

25.3 Relational State Spaces

To illustrate the above technique, we instantiate it for relations with a *nat* as the universe type. The following definition defines the state-space location, assignment operator, and injection universe for this.

definition *rel-local-state* ::

'a::countable itself $\Rightarrow ((\text{nat}, 's) \text{ local-scheme}, 's, \text{nat}, 'a::\text{countable}) \text{ local-prim}$ **where**
rel-local-state *t* = $(\text{sstate} = 1_L, \text{sassigns} = \text{assigns-r}, \text{inj-local} = \text{nat-inj-univ})$

abbreviation *rel-local* (*R_l*) **where**

rel-local $\equiv \text{rel-local-state TYPE}('a::\text{countable})$

syntax

-rel-local-state-type :: *type* $\Rightarrow \text{logic } (R_l[-])$

translations

-rel-local-state-type *a* $\Rightarrow \text{CONST rel-local-state } (-\text{TYPE } a)$

lemma *get-rel-local* [*lens-defs*]:

gets_{R_l} = *id*

by (*simp add: rel-local-state-def lens-defs*)

lemma *rel-local-state* [simp]: *utp-local-state* R_l
 by (unfold-locales, simp-all add: upred-defs assigns-comp rel-local-state-def)

lemma *sassigns-rel-state* [simp]: $\langle \sigma \rangle_{R_l} = \langle \sigma \rangle_a$
 by (simp add: rel-local-state-def)

syntax

-rel-var-scope :: $id \Rightarrow logic \Rightarrow logic (var - \cdot - [0, 10] 10)$
 -rel-var-scope-type :: $id \Rightarrow type \Rightarrow logic \Rightarrow logic (var - :: - \cdot - [0, 0, 10] 10)$

translations

-rel-var-scope $x P \Rightarrow$ -var-scope $R_l x P$
 -rel-var-scope-type $x t P \Rightarrow$ -var-scope-type (-rel-local-state-type t) $x t P$

Next we prove some examples laws.

lemma *rel-var-ex-1*: $(var x :: string \cdot II) = II$
 by (rel-auto')

lemma *rel-var-ex-2*: $(var x \cdot x := 1) = II$
 by (rel-auto')

lemma *rel-var-ex-3*: $x \bowtie store \Longrightarrow x := 1 ;; open[R_l['a::countable]] = open[R_l['a]] ;; x := 1$
 by (metis pr-var-def rel-local-state sassigns-rel-state unrest-one utp-local-state.var-open-commute)

lemma *rel-var-ex-4*: $\llbracket x \bowtie store; store \sharp v \rrbracket \Longrightarrow x := v ;; open[R_l['a::countable]] = open[R_l['a]] ;; x := v$
 by (metis pr-var-def rel-local-state sassigns-rel-state utp-local-state.var-open-commute)

lemma *rel-var-ex-5*: $\llbracket x \bowtie store; store \sharp v \rrbracket \Longrightarrow x := v ;; (var y :: int \cdot P) = (var y :: int \cdot x := v ;; P)$
 by (simp add: utp-local-state.var-scope-def segr-assoc[THEN sym] rel-var-ex-4, rel-auto', force+)

end

26 Meta-theory for the Standard Core

theory *utp*

imports

utp-var
utp-expr
utp-unrest
utp-usedby
utp-subst
utp-meta-subst
utp-alphabet
utp-lift
utp-pred
utp-pred-laws
utp-recursion
utp-deduct
utp-rel
utp-rel-laws
utp-tactics
utp-hoare

```

    utp-wp
    utp-sp
    utp-theory
    utp-concurrency
    utp-rel-opsem
    utp-local
    utp-event
begin end

```

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] 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.
- [9] 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.
- [10] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.
- [11] S. Foster and F. Zeyda. Optics. *Archive of Formal Proofs*, May 2017. <http://isa-afp.org/entries/Optics.html>, Formal proof development.
- [12] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *UTP*, LNCS 8963, pages 21–41. Springer, 2014.
- [13] 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.

- [14] T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
- [15] 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.
- [16] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997.
- [17] C. Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.
- [18] M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In *UTP 2006*, volume 4010 of *LNCS*, pages 123–140. Springer, 2007.
- [19] 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.
- [20] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *UTP*, LNCS 10134. Springer, 2016.