

Isabelle/UTP: Mechanised Theory Engineering for the UTP

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

January 21, 2019

Abstract

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

Contents

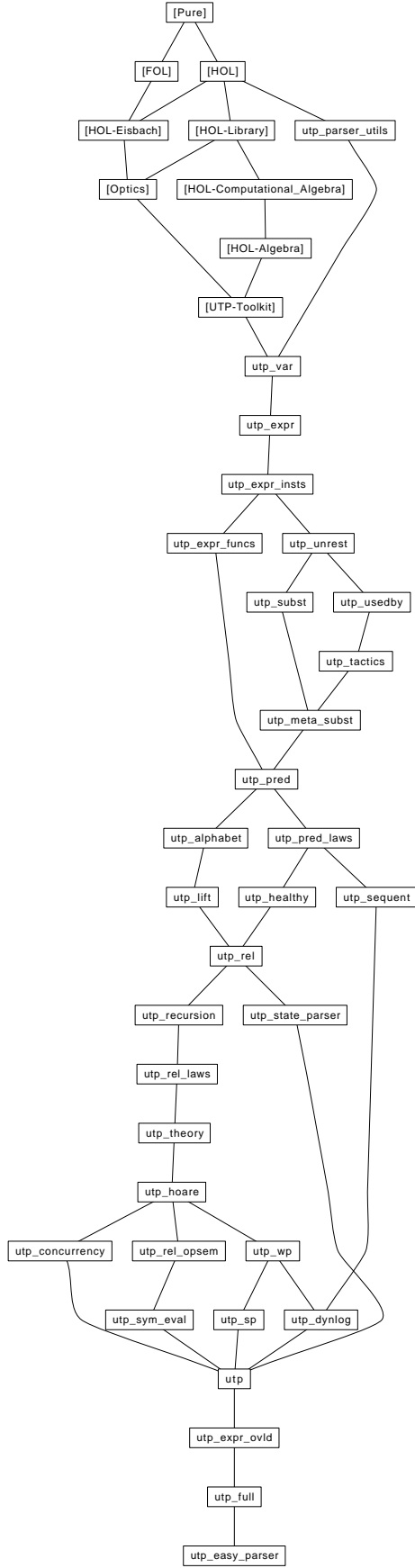
1	Introduction	7
2	UTP Variables	8
2.1	Initial syntax setup	8
2.2	Variable foundations	8
2.3	Variable lens properties	9
2.4	Lens simplifications	11
2.5	Syntax translations	11
3	UTP Expressions	14
3.1	Expression type	14
3.2	Core expression constructs	14
3.3	Type class instantiations	16
3.4	Syntax translations	17
3.5	Evaluation laws for expressions	18
3.6	Misc laws	18
3.7	Literalise tactics	19
4	Expression Type Class Instantiations	20
4.1	Expression construction from HOL terms	22
4.2	Lifting set collectors	25
4.3	Lifting limits	25
5	Unrestriction	26
5.1	Definitions and Core Syntax	26
5.2	Unrestriction laws	27

6	Used-by	30
7	Substitution	32
7.1	Substitution definitions	32
7.2	Syntax translations	34
7.3	Substitution Application Laws	35
7.4	Substitution laws	38
7.5	Ordering substitutions	39
7.6	Unrestriction laws	40
7.7	Conditional Substitution Laws	40
7.8	Parallel Substitution Laws	40
8	UTP Tactics	42
8.1	Theorem Attributes	42
8.2	Generic Methods	42
8.3	Transfer Tactics	43
8.3.1	Robust Transfer	43
8.3.2	Faster Transfer	43
8.4	Interpretation	44
8.5	User Tactics	44
9	Meta-level Substitution	46
10	Alphabetised Predicates	47
10.1	Predicate type and syntax	47
10.2	Predicate operators	48
10.3	Unrestriction Laws	53
10.4	Used-by laws	55
10.5	Substitution Laws	55
10.6	Sandbox for conjectures	57
11	Alphabet Manipulation	58
11.1	Preliminaries	58
11.2	Alphabet Extrusion	58
11.3	Expression Alphabet Restriction	60
11.4	Predicate Alphabet Restriction	62
11.5	Alphabet Lens Laws	62
11.6	Substitution Alphabet Extension	63
11.7	Substitution Alphabet Restriction	63
12	Lifting Expressions	64
12.1	Lifting definitions	64
12.2	Lifting Laws	64
12.3	Substitution Laws	65
12.4	Unrestriction laws	65
13	Predicate Calculus Laws	65
13.1	Propositional Logic	65
13.2	Lattice laws	69
13.3	Equality laws	74

13.4	HOL Variable Quantifiers	75
13.5	Case Splitting	76
13.6	UTP Quantifiers	77
13.7	Variable Restriction	79
13.8	Conditional laws	79
13.9	Additional Expression Laws	80
13.10	Refinement By Observation	81
13.11	Cylindric Algebra	82
14	Healthiness Conditions	82
14.1	Main Definitions	82
14.2	Properties of Healthiness Conditions	84
15	Alphabetised Relations	88
15.1	Relational Alphabets	88
15.2	Relational Types and Operators	89
15.3	Syntax Translations	93
15.4	Relation Properties	94
15.5	Introduction laws	94
15.6	Unrestriction Laws	95
15.7	Substitution laws	96
15.8	Alphabet laws	98
15.9	Relational unrestricted	99
16	Fixed-points and Recursion	102
16.1	Fixed-point Laws	102
16.2	Obtaining Unique Fixed-points	102
16.3	Noetherian Induction Instantiation	104
17	Sequent Calculus	106
18	Relational Calculus Laws	106
18.1	Conditional Laws	106
18.2	Precondition and Postcondition Laws	107
18.3	Sequential Composition Laws	107
18.4	Iterated Sequential Composition Laws	111
18.5	Quantale Laws	111
18.6	Skip Laws	111
18.7	Assignment Laws	112
18.8	Non-deterministic Assignment Laws	114
18.9	Converse Laws	114
18.10	Assertion and Assumption Laws	115
18.11	Frame and Antiframe Laws	115
18.12	While Loop Laws	117
18.13	Algebraic Properties	118
18.13.1	Kleene Star	120
18.14	Kleene Plus	120
18.15	Omega	120
18.16	Relation Algebra Laws	121
18.17	Kleene Algebra Laws	121

18.18	Omega Algebra Laws	122
18.19	Refinement Laws	123
18.20	Domain and Range Laws	123
19	UTP Theories	124
19.1	Complete lattice of predicates	124
19.2	UTP theories hierarchy	125
19.3	UTP theory hierarchy	127
19.4	Theory of relations	135
19.5	Theory links	136
20	Relational Hoare calculus	137
20.1	Hoare Triple Definitions and Tactics	137
20.2	Basic Laws	137
20.3	Assignment Laws	138
20.4	Sequence Laws	138
20.5	Conditional Laws	139
20.6	Recursion Laws	139
20.7	Iteration Rules	139
20.8	Frame Rules	141
21	Weakest (Liberal) Precondition Calculus	142
22	Dynamic Logic	143
22.1	Definitions	143
22.2	Box Laws	143
22.3	Diamond Laws	144
22.4	Sequent Laws	144
23	State Variable Declaration Parser	145
23.1	Examples	146
24	Relational Operational Semantics	146
25	Symbolic Evaluation of Relational Programs	148
26	Strong Postcondition Calculus	149
27	Concurrent Programming	150
27.1	Variable Renamings	150
27.2	Merge Predicates	152
27.3	Separating Simulations	152
27.4	Associative Merges	154
27.5	Parallel Operators	154
27.6	Unrestriction Laws	155
27.7	Substitution laws	155
27.8	Parallel-by-merge laws	156
27.9	Example: Simple State-Space Division	158
28	Meta-theory for the Standard Core	159

29 Overloaded Expression Constructs	160
29.1 Overloadable Constants	160
29.2 Syntax Translations	161
29.3 Simplifications	162
29.4 Indexed Assignment	162
30 Meta-theory for the Standard Core with Overloaded Constructs	162
31 UTP Easy Expression Parser	163
31.1 Replacing the Expression Grammar	163
31.2 Expression Operators	163
31.3 Predicate Operators	163
31.4 Arithmetic Operators	164
31.5 Sets	164
31.6 Imperative Program Syntax	165



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
    UTP-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-inv-bij [simp]
declare id-bij-lens [simp]
declare lens-indep-left-ext [simp]
declare lens-indep-right-ext [simp]
declare lens-comp-quotient [simp]
declare plus-lens-distr [THEN sym, simp]

```

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

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

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

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

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

definition $pr-var :: ('a \Longrightarrow '\beta) \Rightarrow ('a \Longrightarrow '\beta)$ **where**
 $[lens-defs]: pr-var\ x = x$

2.3 Variable lens properties

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

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

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

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

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

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

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

lemma $out-var-weak-lens\ [simp]:$
 $weak-lens\ x \Longrightarrow weak-lens\ (out-var\ x)$

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

lemma *out-var-semi-uvar* [*simp*]:
 $mwb\text{-}lens\ x \implies mwb\text{-}lens\ (out\text{-}var\ x)$
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]: $in\text{-}var\ (x\ +_L\ y) = in\text{-}var\ x\ +_L\ in\text{-}var\ y$
 by (simp add: in-var-def)

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

Similar properties follow for sublens

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

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

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

2.4 Lens simplifications

We also define some lookup abstraction simplifications.

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

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

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

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

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

2.5 Syntax translations

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

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

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

syntax — Identifiers

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

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

syntax — Decorations

-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 :: $salpha$ (Σ)
-salpha-none :: $salpha$ (\emptyset)
-svar-nil :: $svar \Rightarrow svars$ (-)
-svar-cons :: $svar \Rightarrow svars \Rightarrow svars$ (-, / -)
-salphaset :: $svars \Rightarrow salpha$ ({-})
-salphamk :: $logic \Rightarrow salpha$

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

syntax — Quotations

-ualpha-set :: $svars \Rightarrow logic$ ({-} _{α})
-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 $x \ y \rightarrow y ;_L x$
 -mk-svid-list $(\text{-svid-unit } x) \rightarrow x$
 -mk-svid-list $(\text{-svid-list } x \ xs) \rightarrow x +_L \text{-mk-svid-list } xs$

— Decorations

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

-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 $A \rightarrow A$
 $(\text{-svar-cons } x \ (\text{-salphamk } y)) \leftarrow \text{-salphamk } (x +_L y)$
 $x \leftarrow \text{-salphamk } x$
 -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 :: *type* \Rightarrow *type* \Rightarrow *type*

parse-translation \langle

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

\rangle

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 ' α ' to the expression's type ' a '. This general model will allow us to unify all constructions under one type. The majority definitions in the file are given using the *lifting* package [15], which allows us to reuse much of the existing library of HOL functions.

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

setup-lifting *type-definition-uexpr*

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

notation *Abs-uexpr* (*mk_e*)

lemma *uexpr-eq-iff*:

e = *f* \longleftrightarrow (\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 *uexpr-defs* **and** *ueval* **and** *lit-simps* **and** *lit-norm*

3.2 Core expression constructs

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

lift-definition *var* :: (*'t* \Rightarrow ' α) \Rightarrow (*'t*, ' α) *uexpr* **is** *lens-get* .

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

lift-definition *lit* :: $'t \Rightarrow ('t, 'α) \text{ ueexpr } (\ll\text{-}\gg) \text{ is } \lambda v b. v$.

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

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

lift-definition *bop* ::
 $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'α) \text{ ueexpr } \Rightarrow ('b, 'α) \text{ ueexpr } \Rightarrow ('c, 'α) \text{ ueexpr}$
is $\lambda f u v b. f (u b) (v b)$.

lift-definition *trop* ::
 $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow ('a, 'α) \text{ ueexpr } \Rightarrow ('b, 'α) \text{ ueexpr } \Rightarrow ('c, 'α) \text{ ueexpr } \Rightarrow ('d, 'α) \text{ ueexpr}$
is $\lambda f u v w b. f (u b) (v b) (w b)$.

lift-definition *qtop* ::
 $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e) \Rightarrow$
 $('a, 'α) \text{ ueexpr } \Rightarrow ('b, 'α) \text{ ueexpr } \Rightarrow ('c, 'α) \text{ ueexpr } \Rightarrow ('d, 'α) \text{ ueexpr } \Rightarrow$
 $('e, 'α) \text{ ueexpr}$
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* :: $('a \Rightarrow ('b, 'α) \text{ ueexpr}) \Rightarrow ('a \Rightarrow 'b, 'α) \text{ ueexpr}$
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* :: $bool \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**
 $[ueexpr\text{-}defs]: uIf = If$

abbreviation *cond* ::
 $('a, 'α) \text{ ueexpr } \Rightarrow (bool, 'α) \text{ ueexpr } \Rightarrow ('a, 'α) \text{ ueexpr } \Rightarrow ('a, 'α) \text{ ueexpr}$
 $((3\text{-}\triangleleft\text{-}\triangleright\text{-}) [52, 0, 53] 52)$
where $P \triangleleft b \triangleright Q \equiv trop\ uIf\ b\ P\ Q$

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

definition *eq-upred* :: $('a, 'α) \text{ ueexpr } \Rightarrow ('a, 'α) \text{ ueexpr } \Rightarrow (bool, 'α) \text{ ueexpr}$ (**infixl** $=_u$ 50)
where $[ueexpr\text{-}defs]: eq\text{-}upred\ x\ y = bop\ HOL.eq\ x\ y$

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

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

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

3.3 Type class instantiations

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

```
instantiation uepr :: (zero, type) zero
begin
  definition zero-uepr-def [uepr-defs]:  $0 = \text{lit } 0$ 
instance ..
end
```

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

```
end
```

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

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

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

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

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

```
instantiation uepr :: (ord, type) ord
begin
  lift-definition less-eq-uepr :: ('a, 'b) uepr  $\Rightarrow$  ('a, 'b) uepr  $\Rightarrow$  bool
  is  $\lambda P \ Q. (\forall A. P \ A \leq Q \ A)$  .
  definition less-uepr :: ('a, 'b) uepr  $\Rightarrow$  ('a, 'b) uepr  $\Rightarrow$  bool
  where [uepr-defs]: less-uepr 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 uepr :: (order, type) order
proof
  fix x y z :: ('a, 'b) uepr
```



```

show  $(x < y) = (x \leq y \wedge \neg y \leq x)$  by (simp add: less-uepr-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

```

3.4 Syntax translations

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

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

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

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

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

syntax — Core expression constructs

```

-ucorce   :: logic  $\Rightarrow$  type  $\Rightarrow$  logic (infix  $:_u$  50)
-ulambda   :: pttrn  $\Rightarrow$  logic  $\Rightarrow$  logic  $(\lambda \cdot - \cdot [0, 10] \ 10)$ 
-ulens-ovrd :: logic  $\Rightarrow$  logic  $\Rightarrow$  salpha  $\Rightarrow$  logic  $(- \oplus - \text{on } [85, 0, 86] \ 86)$ 
-ulens-get  :: logic  $\Rightarrow$  svar  $\Rightarrow$  logic  $(:- [900, 901] \ 901)$ 
-umem      ::  $('a, 'a) \text{ uepr} \Rightarrow ('a \text{ set}, 'a) \text{ uepr} \Rightarrow (\text{bool}, 'a) \text{ uepr}$  (infix  $\in_u$  50)

```

translations

```

 $\lambda x \cdot p == \text{CONST ulambda } (\lambda x. p)$ 
 $x :_u 'a == x :: ('a, -) \text{ uepr}$ 
-ulens-ovrd  $f \ g \ a \Rightarrow \text{CONST bop } (\text{CONST ulens-override } a) \ f \ g$ 
-ulens-ovrd  $f \ g \ a \leq \text{CONST bop } (\lambda x \ y. \text{CONST lens-override } x1 \ y1 \ a) \ f \ g$ 
-ulens-get  $x \ y == \text{CONST uop } (\text{CONST lens-get } y) \ x$ 
 $x \in_u A == \text{CONST bop } (\in) \ x \ A$ 

```

syntax — Tuples

```

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

```

translations

```

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

```

syntax — Orders

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

translations

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

3.5 Evaluation laws for expressions

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

lemma *lit-ueval* [ueval]: $\llbracket \langle x \rangle \rrbracket_e b = x$
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*)

3.6 Misc laws

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

lemma *uop-const* [simp]: $uop \text{ id } u = u$
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 *ueprfst* [simp]: $\pi_1((e, f)_u) = e$
by (*transfer*, *simp*)

lemma *ueprsnd* [simp]: $\pi_2((e, f)_u) = f$
by (*transfer*, *simp*)

3.7 Literalise tactics

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

lemma *lit-fun-simps* [lit_simps]:

```

  <<i x y z u>> = qtop i <<x>> <<y>> <<z>> <<u>>
  <<h x y z>> = trop h <<x>> <<y>> <<z>>
  <<g x y>> = bop g <<x>> <<y>>
  <<f x>> = uop f <<x>>
  by (transfer, simp)+

```

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

lemma *numeral-uepr-rep-eq* [ueval]: $\llbracket \text{numeral } x \rrbracket_e b = \text{numeral } x$

```

  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*: $\text{numeral } x = \llbracket \text{numeral } x \rrbracket$

```

  by (simp add: uepr-eq-iff numeral-uepr-rep-eq lit.rep-eq)

```

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

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

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

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

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

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

$0 = \llbracket 0 \rrbracket$

$1 = \llbracket 1 \rrbracket$

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

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

set-of ?t = UNIV in reverse to correctly interpret these. Moreover, numerals must be handled separately by first simplifying them and then converting them into UTP expression numerals; hence the following two simplification rules.

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

```

  by (simp add: uop-def)

```

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

```

  by (metis lit.abs-eq lit-numeral)

```

method *literalise* = (unfold lit_simps[THEN sym])

```

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

```

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

final unliteralises at the end.

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

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

end

4 Expression Type Class Instantiations

theory *utp-expr-insts*
imports *utp-expr*
begin

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

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

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

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

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

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

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

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

```

begin
  definition mod-uepr-def [uepr-defs]: u mod v = bop (mod) u v
instance ..
end

```

```

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

```

```

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

```

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

We also lift the properties from certain ordered groups.

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

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

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

The next theorem lifts powers.

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

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

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

4.1 Expression construction from HOL terms

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

named-theorems *mkueexpr*

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

lemma *mkueexpr-zero* [mkueexpr]: $\text{mk}_e (\lambda s. 0) = 0$
by (simp add: zero-uexpr-def, transfer, simp)

lemma *mkueexpr-one* [mkueexpr]: $\text{mk}_e (\lambda s. 1) = 1$
by (simp add: one-uexpr-def, transfer, simp)

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

lemma *mkueexpr-lit* [mkueexpr]: $\text{mk}_e (\lambda s. k) = \llbracket k \rrbracket$
by (transfer, simp)

lemma *mkueexpr-pair* [mkueexpr]: $\text{mk}_e (\lambda s. (f s, g s)) = (\text{mk}_e f, \text{mk}_e g)_u$
by (transfer, simp)

lemma *mkueexpr-plus* [mkueexpr]: $\text{mk}_e (\lambda s. f s + g s) = \text{mk}_e f + \text{mk}_e g$
by (simp add: plus-uexpr-def, transfer, simp)

lemma *mkueexpr-uminus* [mkueexpr]: $\text{mk}_e (\lambda s. - f s) = - \text{mk}_e f$
by (simp add: uminus-uexpr-def, transfer, simp)

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

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

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

end

theory *utp-expr-funcs*

imports *utp-expr-insts*

begin

syntax — Polymorphic constructs

-*uceil* :: $logic \Rightarrow logic \Rightarrow logic \ (\lceil _ \rceil_u)$
-*ufloor* :: $logic \Rightarrow logic \Rightarrow logic \ (\lfloor _ \rfloor_u)$
-*umin* :: $logic \Rightarrow logic \Rightarrow logic \ (min_u'(-, -'))$
-*umax* :: $logic \Rightarrow logic \Rightarrow logic \ (max_u'(-, -'))$
-*ugcd* :: $logic \Rightarrow logic \Rightarrow logic \ (gcd_u'(-, -'))$

translations

— Type-class polymorphic constructs

$min_u(x, y) == CONST \ bop \ (CONST \ min) \ x \ y$
 $max_u(x, y) == CONST \ bop \ (CONST \ max) \ x \ y$
 $gcd_u(x, y) == CONST \ bop \ (CONST \ gcd) \ x \ y$
 $\lceil x \rceil_u == CONST \ uop \ CONST \ ceiling \ x$
 $\lfloor x \rfloor_u == CONST \ uop \ CONST \ floor \ x$

syntax — Lists / Sequences

-*ucons* :: $logic \Rightarrow logic \Rightarrow logic \ (\mathbf{infixr} \ \#_u \ 65)$
-*unil* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \ (\langle \rangle)$
-*ulist* :: $args \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \ (\langle \langle - \rangle \rangle)$
-*uappend* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \ (\mathbf{infixr} \ \hat{_}_u \ 80)$
-*udconcat* :: $logic \Rightarrow logic \Rightarrow logic \ (\mathbf{infixr} \ \frown_u \ 90)$
-*ulast* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a}, \text{'}\alpha) \ uexpr \ (last_u'(-))$
-*ufront* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \ (front_u'(-))$
-*uhead* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a}, \text{'}\alpha) \ uexpr \ (head_u'(-))$
-*utail* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \ (tail_u'(-))$
-*utake* :: $(nat, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \ (take_u'(-, -'))$
-*udrop* :: $(nat, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \ (drop_u'(-, -'))$
-*ufilter* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a set}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \ (\mathbf{infixl} \ \downarrow_u \ 75)$
-*uextract* :: $(\text{'a set}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a list}, \text{'}\alpha) \ uexpr \ (\mathbf{infixl} \ \downarrow_u \ 75)$
-*uelems* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (\text{'a set}, \text{'}\alpha) \ uexpr \ (elems_u'(-))$
-*usorted* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (bool, \text{'}\alpha) \ uexpr \ (sorted_u'(-))$
-*udistinct* :: $(\text{'a list}, \text{'}\alpha) \ uexpr \Rightarrow (bool, \text{'}\alpha) \ uexpr \ (distinct_u'(-))$
-*uupto* :: $logic \Rightarrow logic \Rightarrow logic \ (\langle \dots \rangle)$
-*uupt* :: $logic \Rightarrow logic \Rightarrow logic \ (\langle \dots < \rangle)$
-*umap* :: $logic \Rightarrow logic \Rightarrow logic \ (map_u)$
-*uzip* :: $logic \Rightarrow logic \Rightarrow logic \ (zip_u)$

translations

$x \ \#_u \ ys == CONST \ bop \ (\#) \ x \ ys$
 $\langle \rangle == \llbracket \rrbracket$

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

syntax — Sets

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

translations

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

$A ;_u B == \text{CONST bop } \text{CONST relcomp } A B$

syntax — Partial functions

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

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

$\text{-uinl} :: \text{logic} \Rightarrow \text{logic} \text{ (inl}_u \text{ '(-))}$
 $\text{-uinr} :: \text{logic} \Rightarrow \text{logic} \text{ (inr}_u \text{ '(-))}$

translations

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

4.2 Lifting set collectors

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

syntax

$\text{-uset-atLeastAtMost} :: ('a, 'α) \text{ uexpr} \Rightarrow ('a, 'α) \text{ uexpr} \Rightarrow ('a \text{ set}, 'α) \text{ uexpr} \text{ ((1\{-..\}-\}_u))$
 $\text{-uset-atLeastLessThan} :: ('a, 'α) \text{ uexpr} \Rightarrow ('a, 'α) \text{ uexpr} \Rightarrow ('a \text{ set}, 'α) \text{ uexpr} \text{ ((1\{--<-\}_u))}$
 $\text{-uset-compr} :: \text{pttrn} \Rightarrow ('a \text{ set}, 'α) \text{ uexpr} \Rightarrow (\text{bool}, 'α) \text{ uexpr} \Rightarrow ('b, 'α) \text{ uexpr} \Rightarrow ('b \text{ set}, 'α) \text{ uexpr} \text{ ((1\{-:/ - \}/ - \cdot / -\}_u))}$
 $\text{-uset-compr-nset} :: \text{pttrn} \Rightarrow (\text{bool}, 'α) \text{ uexpr} \Rightarrow ('b, 'α) \text{ uexpr} \Rightarrow ('b \text{ set}, 'α) \text{ uexpr} \text{ ((1\{- \}/ - \cdot / -\}_u))}$

lift-definition $\text{ZedSetCompr} ::$

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

translations

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

4.3 Lifting limits

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

definition $\text{ulim-left} :: 'a::\text{order-topology} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b::t2\text{-space} \text{ where}$

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

definition $\text{ulim-right} :: 'a::\text{order-topology} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b::t2\text{-space} \text{ where}$

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

definition $\text{ucont-on} :: ('a::\text{topological-space} \Rightarrow 'b::\text{topological-space}) \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \text{ where}$

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

syntax

$\text{-ulim-left} :: \text{id} \Rightarrow \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \text{ (lim}_u \text{ '(-} \rightarrow \text{-} \text{')'(-))}$

$\text{-ulim-right} :: \text{id} \Rightarrow \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\lim_u'(- \rightarrow -^+)'(-'))$
 $\text{-ucont-on} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \ (\mathbf{infix} \ \text{cont-on}_u \ 90)$

translations

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

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

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

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

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

end

5 Unrestriction

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

5.1 Definitions and Core Syntax

Unrestriction is an encoding of semantic freshness that allows us to reason about the presence of variables in predicates without being concerned with abstract syntax trees. An expression p is unrestricted by lens x , written $x \sharp p$, 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

$-unrest :: salpha \Rightarrow logic \Rightarrow logic \Rightarrow logic$ (**infix** $\#$ 20)

translations

$-unrest\ x\ p == CONST\ unrest\ x\ p$

$-unrest\ (-salphaset\ (-salphamk\ (x\ +_L\ y)))\ P\ <= -unrest\ (x\ +_L\ y)\ P$

Our syntax translations support both variables and variable sets such that we can write down predicates like $\&x \# P$ and also $\{\&x, \&y, \&z\} \# P$.

We set up a simple tactic for discharging unrestriction 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-uevr* :: $('a \Longrightarrow 'a) \Rightarrow ('b, 'a)\ uevr \Rightarrow bool$

is $\lambda x\ e.\ \forall\ b\ v.\ e\ (put_x\ b\ v) = e\ b$.

adhoc-overloading

unrest unrest-uevr

lemma *unrest-expr-alt-def*:

$weak-lens\ x \Longrightarrow (x \# P) = (\forall\ b\ b'. \llbracket P \rrbracket_e (b \oplus_L b' \text{ on } x) = \llbracket P \rrbracket_e b)$

by (*transfer, metis lens-override-def weak-lens.put-get*)

5.2 Unrestriction laws

We now prove unrestriction 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*]:

$\llbracket x \# P; y \# P \rrbracket \Longrightarrow x;y \# P$

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

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

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

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

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

by (*transfer, simp*)

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

lemma *unrest-sublens*:
fixes $P :: ('a, 'α) \text{ uexpr}$
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 unrestrictions over them are equivalent.

lemma *unrest-equiv*:
fixes $P :: ('a, 'α) \text{ uexpr}$
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-uexpr.rep-eq*)

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

lemma *bij-lens-unrest-all*:
fixes $P :: ('a, 'α) \text{ uexpr}$
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 $\text{bij-lens } X$
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 unrestriction 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*]: $\llbracket \text{mwb-lens } x; x \bowtie y \rrbracket \Longrightarrow \$y' \# \$x'$
by (*simp add: unrest-var*)

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

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

lemma *unrest-ouvar-iuvar* [*unrest*]:
fixes $x :: ('a \Longrightarrow 'a)$
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 unrestrictions for the majority of the expression language.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

6 Used-by

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

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

consts
usedBy :: $'a \Rightarrow 'b \Rightarrow \text{bool}$

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

translations
 $\text{-usedBy } x p == \text{CONST usedBy } x p$
 $\text{-usedBy } (\text{-salphaset } (\text{-salphamk } (x +_L y))) P <= \text{-usedBy } (x +_L y) P$

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

adhoc-overloading *usedBy usedBy-uepr*

lemma *usedBy-lit* [*unrest*]: $x \# \langle v \rangle$
by (*transfer, simp*)

lemma *usedBy-sublens*:
fixes $P :: ('a, 'a) \text{ uepr}$
assumes $x \# P x \subseteq_L y \text{ vwb-lens } y$
shows $y \# P$
using *assms*

by (*transfer*, *auto*, *metis* *Lens-Order.lens-override-idem* *lens-override-def* *sublens-obs-get* *vwb-lens-mwb*)

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

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

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

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

lemma *usedBy-indep-uses*:
fixes $P :: ('a, 'a) \text{ uexpr}$
assumes $x \Vdash P \ x \bowtie y$
shows $y \nVdash P$
using *assms* **by** (*transfer*, *auto*, *metis* *lens-indep-get* *lens-override-def*)

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

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

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

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

lemma *usedBy-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-uexpr-def*)

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

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

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

```

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

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

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

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

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

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

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

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

end

```

7 Substitution

```

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

```

7.1 Substitution definitions

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

```

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

```

```

named-theorems usubst

```

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

```

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

```


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

7.2 Syntax translations

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

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

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

syntax

```

-smaplet :: [salpha, 'a] => smaplet      (- / $\mapsto_s$  / -)
          :: smaplet => smaplets          (-)
-SMaplets :: [smaplet, smaplets] => smaplets (-, / -)
-SubstUpd :: ['m usubst, smaplets] => 'm usubst (-/'(-) [900,0] 900)
-Subst    :: smaplets => 'a  $\rightarrow$  'b      ((1[-]))
-psubst   :: [logic, svars, uexprs]  $\Rightarrow$  logic
-subst    :: logic  $\Rightarrow$  uexprs  $\Rightarrow$  salphas  $\Rightarrow$  logic (([-]'(-)] [990,0,0] 991)
-uexp-l   :: logic  $\Rightarrow$  uexp (- [64] 64)
-uexprs   :: [uexp, uexprs] => uexprs (-, / -)
          :: uexp => uexprs (-)
-salphas  :: [salpha, salphas] => salphas (-, / -)
          :: salpha => salphas (-)
-par-subst :: logic  $\Rightarrow$  salpha  $\Rightarrow$  salpha  $\Rightarrow$  logic  $\Rightarrow$  logic (- [-]'(-)s - [100,0,0,101] 101)

```

translations

```

-SubstUpd m (-SMaplets xy ms)      == -SubstUpd (-SubstUpd m xy) ms
-SubstUpd m (-smaplet x y)          == CONST subst-upd m x y
-Subst ms                            == -SubstUpd (CONST id) ms
-Subst (-SMaplets ms1 ms2)          <= -SubstUpd (-Subst ms1) ms2
-SMaplets ms1 (-SMaplets ms2 ms3) <= -SMaplets (-SMaplets ms1 ms2) ms3
-subst P es vs => CONST subst (-psubst (CONST id) vs es) P
-psubst m (-salphas x xs) (-uexprs v vs) => -psubst (-psubst m x v) xs vs
-psubst m x v => CONST subst-upd m x v
-subst P v x <= CONST usubst (CONST subst-upd (CONST id) x v) P
-subst P v x <= -subst P (-spvar x) v
-par-subst σ1 A B σ2 == CONST par-subst σ1 A B σ2
-uexp-l e => e

```

Thus we can write things like $\sigma(x \mapsto_s v)$ to update a variable x in σ with expression v , $[x \mapsto_s e, y \mapsto_s f]$ to construct a substitution with two variables, and finally $P[v/x]$, the traditional syntax.

We can now express deletion of a substitution maplet.

definition $\text{subst-del} :: 'a \text{ usubst} \Rightarrow ('a \Longrightarrow 'a) \Rightarrow 'a \text{ usubst}$ (**infix** $-_s$ 85) **where**
 $\text{subst-del } \sigma \ x = \sigma(x \mapsto_s \&x)$

7.3 Substitution Application Laws

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

method *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*]: $\langle id \rangle_s x = var\ x$
by (*transfer, simp*)

lemma *subst-upd-id-lam* [*usubst*]: $subst\text{-}upd\ (\lambda\ x.\ x)\ x\ v = subst\text{-}upd\ id\ x\ v$
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 $\langle \sigma(x \mapsto_s v) \rangle_s (pr\text{-}var\ x) = v$
using *assms*
by (*simp add: subst-upd-uvar-def pr-var-def, transfer*) (*simp*)

Substitution update is idempotent.

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

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

Substitution updates commute when the lenses are independent.

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

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

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

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

lemma *swap-usubst-inj*:
fixes $x\ y :: ('a \implies 'α)$
assumes $vwb\text{-}lens\ x\ vwb\text{-}lens\ y\ x \bowtie y$
shows $inj\ [x \mapsto_s \&y, y \mapsto_s \&x]$
proof (*rule injI*)
fix $b_1 :: 'α$ **and** $b_2 :: 'α$
assume $[x \mapsto_s \&y, y \mapsto_s \&x]\ b_1 = [x \mapsto_s \&y, y \mapsto_s \&x]\ b_2$
hence $a: put_y (put_x b_1 (\llbracket \&y \rrbracket_e b_1)) (\llbracket \&x \rrbracket_e b_1) = put_y (put_x b_2 (\llbracket \&y \rrbracket_e b_2)) (\llbracket \&x \rrbracket_e b_2)$
by (*auto simp add: subst-upd-uvar-def*)
then have $(\forall a\ b\ c. put_x (put_y a\ b)\ c = put_y (put_x a\ c)\ b) \wedge$
 $(\forall a\ b. get_x (put_y a\ b) = get_x a) \wedge (\forall a\ b. get_y (put_x a\ b) = get_y a)$
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*]:
 $vwb\text{-}lens\ x \implies [x \mapsto_s var\ x] = id$
apply (*simp add: subst-upd-uvar-def*)
apply (*transfer*)
apply (*rule ext*)
apply (*auto*)
done

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

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

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

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

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

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

$x \bowtie y \implies \text{subst-upd } s \ (x +_L y) \ e = s(x \mapsto_s \pi_1(e), y \mapsto_s \pi_2(e))$

by (*simp add: subst-upd-uvar-def lens-defs, transfer, auto simp add: fun-eq-iff prod.case-eq-if lens-indep-comm*)

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

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

$\llbracket \text{vwb-lens } x; x \# \sigma \rrbracket \implies \langle \sigma \rangle_s x = \text{var } x$

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*]:

$\text{vwb-lens } x \implies \text{id } -_s x = \text{id}$

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

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

$\text{mwb-lens } x \implies \sigma(x \mapsto_s v) -_s x = \sigma -_s x$

by (*simp add: subst-del-def subst-upd-uvar-def*)

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

$x \bowtie y \implies \sigma(y \mapsto_s v) -_s x = (\sigma -_s x)(y \mapsto_s v)$

by (*simp add: subst-del-def subst-upd-uvar-def lens-indep-comm*)

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

lemma *subst-unrest* [*usubst*]: $x \# P \implies \sigma(x \mapsto_s v) \dagger P = \sigma \dagger P$

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

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

fixes $P :: ('a, 'α) \text{ uexpr}$

assumes $x \# P \ x \bowtie y$

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 $P :: ('a, 'α) \text{ uexpr}$

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 $P :: ('a, 'α) \text{ uexpr}$

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 $P :: ('a, 'α) \text{ uexpr}$

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-uepr.rep-eq mono-def subst.rep-eq)

7.4 Substitution laws

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

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

lemma *subst-lit* [usubst]: $\sigma \dagger \langle v \rangle = \langle v \rangle$
 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-uepr-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 $P :: 'i \Rightarrow 'j \Rightarrow ('a, 'a) \text{ uepr}$
 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-uepr-def subst-bop)

lemma *subst-times* [usubst]: $\sigma \dagger (x * y) = \sigma \dagger x * \sigma \dagger y$

by (simp add: times-ueexpr-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-ueexpr-def usubst)

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

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

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

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

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

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

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

lemma subst-eq-upred [usubst]: $\sigma \dagger (x =_u y) = (\sigma \dagger x =_u \sigma \dagger y)$
 by (simp add: eq-upred-def usubst)

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

lemma subst-subst [usubst]: $\sigma \dagger \varrho \dagger e = (\varrho \circ \sigma) \dagger e$
 by (transfer, simp)

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

lemma subst-upd-comp [usubst]:
 fixes $x :: ('a \Rightarrow 'a)$
 shows $\varrho(x \mapsto_s v) \circ \sigma = (\varrho \circ \sigma)(x \mapsto_s \sigma \dagger v)$
 by (rule ext, simp add: ueexpr-defs subst-upd-uvar-def, transfer, simp)

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

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

7.5 Ordering substitutions

A simplification procedure to reorder substitutions maplets lexicographically by variable syntax

simproc-setup subst-order (subst-upd-uvar (subst-upd-uvar σ x u) y v) =

```

    << (fn - => fn ctx => fn ct =>
      case (Thm.term-of ct) of
        Const (utp-subst.subst-upd-uvar, -) $ (Const (utp-subst.subst-upd-uvar, -) $ s $ x $ u) $ y $ v
      => if (YXML.content-of (Syntax.string-of-term ctx x) > YXML.content-of (Syntax.string-of-term
ctx y))
        then SOME (mk-meta-eq @{thm usubst-upd-comm})
        else NONE |
      - => NONE)
    >>

```

7.6 Unrestriction laws

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

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

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

7.7 Conditional Substitution Laws

lemma *usubst-cond-upd-1* [*usubst*]:
 $\sigma(x \mapsto_s u) \triangleleft b \triangleright_s \varrho(x \mapsto_s v) = (\sigma \triangleleft b \triangleright_s \varrho)(x \mapsto_s u \triangleleft b \triangleright v)$
by (*simp add: cond-subst-def subst-upd-uvar-def uexpr-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 uexpr-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 uexpr-defs, transfer*)
(metis (full-types, hide-lams) id-apply pr-var-def subst-upd-uvar-def usubst-upd-pr-var-id var.rep-eq)

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

7.8 Parallel Substitution Laws

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

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

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

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

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

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

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

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

end

8 UTP Tactics

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

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

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

8.1 Theorem Attributes

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

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

8.2 Generic Methods

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

Generic Predicate Tactics

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

Generic Relational Tactics

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

```

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

```

8.3 Transfer Tactics

Next, we define the component tactics used for transfer.

8.3.1 Robust Transfer

Robust transfer uses the transfer method of the lifting package.

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

8.3.2 Faster Transfer

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

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

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

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

```

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

```

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

```

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

```

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

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

named-theorems *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*)

8.4 Interpretation

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

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

8.5 User Tactics

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

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

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

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

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

ML-file *utp-tactics.ML*

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

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

```
method-setup rel-simp = ⟨
```

```

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

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

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

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

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

```

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

```

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

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

```

end

9 Meta-level Substitution

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

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

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

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

syntax

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

translations

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

end

```

10 Alphabetised Predicates

```

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

```

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

10.1 Predicate type and syntax

An alphabetised predicate is simply a boolean valued expression.

```

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

```

```

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

```

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

```

purge-notation
  conj (infixr  $\wedge$  35) and
  disj (infixr  $\vee$  30) and
  Not ( $\neg$  - [40] 40)

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

```

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

ad hoc-overloading

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

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

nonterminal *idt-list*

syntax

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

translations

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

10.2 Predicate operators

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

class *refine* = *order*

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

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

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

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

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

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

purge-syntax

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

syntax

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

We trivially instantiate our refinement class

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

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

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

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

apply (*transfer*)

apply (*clarsimp*)

done

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

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

begin

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

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

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

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

instance

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

end

```

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

```

```

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

```

```

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

```

```

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

```

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

```

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

```

```

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

```

```

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

```

```

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

```

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

```

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

```

```

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

```

```

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

```

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

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

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

definition *conj-upred* = (*Lattices.inf* :: ' α upred \Rightarrow ' α upred \Rightarrow ' α upred)

definition *disj-upred* = (*Lattices.sup* :: ' α upred \Rightarrow ' α upred \Rightarrow ' α upred)

definition *not-upred* = (*uminus* :: ' α upred \Rightarrow ' α upred)

definition *diff-upred* = (*minus* :: ' α upred \Rightarrow ' α upred \Rightarrow ' α upred)

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

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

notation

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

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

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

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

syntax

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

translations

$\bigcap x \mid P \cdot F \Rightarrow \text{CONST UINF } (\lambda x. P) (\lambda x. F)$
 $\bigcap x \cdot F == \bigcap x \mid \text{true} \cdot F$
 $\bigcap x \cdot F == \bigcap x \mid \text{true} \cdot F$
 $\bigcap x \in A \cdot F \Rightarrow \bigcap x \mid \llbracket x \rrbracket \in_u \llbracket A \rrbracket \cdot F$
 $\bigcap x \in A \cdot F \Leftarrow \bigcap x \mid \llbracket y \rrbracket \in_u \llbracket A \rrbracket \cdot F$
 $\bigcap x \mid P \cdot F \Leftarrow \text{CONST UINF } (\lambda y. P) (\lambda x. F)$
 $\bigcap x \mid P \cdot F(x) \Leftarrow \text{CONST UINF } (\lambda x. P) F$
 $\bigcap x \mid P \cdot F \Rightarrow \text{CONST USUP } (\lambda x. P) (\lambda x. F)$
 $\bigcap x \cdot F == \bigcap x \mid \text{true} \cdot F$
 $\bigcap x \in A \cdot F \Rightarrow \bigcap x \mid \llbracket x \rrbracket \in_u \llbracket A \rrbracket \cdot F$
 $\bigcap x \in A \cdot F \Leftarrow \bigcap x \mid \llbracket y \rrbracket \in_u \llbracket A \rrbracket \cdot F$
 $\bigcap x \mid P \cdot F \Leftarrow \text{CONST USUP } (\lambda y. P) (\lambda x. F)$

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

We also define the other predicate operators

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

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

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

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

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

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

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

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

translations

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

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

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

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

Configuration for UTP tactics

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

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

adhoc-overloading

uttrue *true-upred* **and**
ufalse *false-upred* **and**
unot *not-upred* **and**
uconj *conj-upred* **and**
udisj *disj-upred* **and**
uimpl *impl* **and**
uiff *iff-upred* **and**
uex *ex* **and**
uall *all* **and**
ushEx *shEx* **and**
ushAll *shAll*

syntax

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

translations

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

declare *true-upred-def* [*upred-defs*]
declare *false-upred-def* [*upred-defs*]
declare *conj-upred-def* [*upred-defs*]
declare *disj-upred-def* [*upred-defs*]
declare *not-upred-def* [*upred-defs*]
declare *diff-upred-def* [*upred-defs*]
declare *subst-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,simp*]
declare *false-alt-def* [*THEN sym,simp*]

10.3 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*]:
 $\ll (\bigwedge i. x \# P(i)); (\bigwedge i. x \# Q(i)) \gg \Longrightarrow x \# (\bigcap i \mid P(i) \cdot Q(i))$
by (*pred-auto*)

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

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

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

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

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

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

The sublens proviso can be thought of as membership below.

lemma *unrest-ex-in* [*unrest*]:
 $\llbracket \text{mwb-lens } y; x \subseteq_L y \rrbracket \implies x \# (\exists y. y \cdot P)$
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 $y \# (\exists x. x \cdot P)$
using *assms lens-indep-comm*
by (*rel-simp'*, *fastforce*)

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

lemma *unrest-all-diff* [*unrest*]:
assumes $x \bowtie y$ $y \# P$
shows $y \# (\forall x. x \cdot P)$
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 $\text{mwb-lens } x y \subseteq_L x y \# P$
shows $y \# (P \upharpoonright_v x)$
using *assms*

```

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

```

```

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

```

```

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

```

```

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

```

10.4 Used-by laws

```

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

```

10.5 Substitution Laws

Substitution is monotone

```

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

```

```

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

```

```

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

```

```

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

```

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

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

mw-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 $x \bowtie y \cdot y \nmid v$
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]: $\text{true} \llbracket x \rightarrow v \rrbracket = \text{true}$

by (*pred-auto*)

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

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]:

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

by (*pred-auto*)

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

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

by (*pred-auto*)⁺

lemma *msubst-shAll* [usubst]:

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

by (*pred-auto*)

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

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

by (*pred-auto*)⁺

10.6 Sandbox for conjectures

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

$TRY(P) = (P = \text{undefined})$

translations

$P \leq \text{CONST utp-sandbox } P$

end

11 Alphabet Manipulation

```
theory utp-alphabet
  imports
    utp-pred utp-usedby
begin
```

11.1 Preliminaries

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

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

```
named-theorems alpha
```

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

11.2 Alphabet Extrusion

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

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

lift-definition $aext :: ('a, 'b) uexpr \Rightarrow ('b, 'a) lens \Rightarrow ('a, 'a) uexpr$ (**infixr** \oplus_p 95)
is $\lambda P x b. P (get_x b)$.

```
update-uexpr-rep-eq-thms
```

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

lemma *aext-twice*: $(P \oplus_p a) \oplus_p b = P \oplus_p (a ;_L b)$
by (*pred-auto*)

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

lemma *aext-id* [*simp*]: $P \oplus_p 1_L = P$
by (*pred-auto*)

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

lemma *aext-lit* [*simp*]: $\ll v \gg \oplus_p a = \ll v \gg$

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

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

Alphabet extension distributes through the function liftings.

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

lemma aext-plus [alpha]:

$(x + y) \oplus_p a = (x \oplus_p a) + (y \oplus_p a)$
by (*pred-auto*)

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

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

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

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

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

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

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

Alphabet extension is monotonic and continuous.

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

lemma *aext-cont* [*alpha*]: $\text{vwb-lens } a \implies (\bigsqcap A) \oplus_p a = (\bigsqcap P \in A. P \oplus_p a)$
by (*pred-simp*)

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

lemma *unrest-aext* [*unrest*]:
 $\llbracket \text{mwb-lens } a; x \# p \rrbracket \implies \text{unrest } (x ;_L a) (p \oplus_p a)$
by (*transfer, simp add: lens-comp-def*)

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

lemma *unrest-aext-indep* [*unrest*]:
 $a \bowtie b \implies b \# (p \oplus_p a)$
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* :: $('a, ' \alpha) \text{ uexpr} \Rightarrow (' \beta, ' \alpha) \text{ lens} \Rightarrow ('a, ' \beta) \text{ uexpr}$ (**infixr** \vdash_e 90)

is $\lambda P x b. P \text{ (create}_x b) \text{ .}$

update-uexpr-rep-eq-thms

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

11.7 Substitution Alphabet Restriction

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

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

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

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

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

lemma *unrest-subst-alpha-ext* [*unrest*]:
 $x \bowtie y \implies x \# (P \oplus_s y)$
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* :: $('a, '\alpha)\ uexpr \Rightarrow ('a, '\alpha \times '\beta)\ uexpr\ (\lceil \cdot \rceil_<)$
where $\lceil P \rceil_< \equiv P \oplus_p fst_L$

abbreviation *drop-pre* :: $('a, '\alpha \times '\beta)\ uexpr \Rightarrow ('a, '\alpha)\ uexpr\ (\lfloor \cdot \rfloor_<)$
where $\lfloor P \rfloor_< \equiv P \downarrow_e fst_L$

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* :: $('a, '\beta)\ uexpr \Rightarrow ('a, '\alpha \times '\beta)\ uexpr\ (\lceil \cdot \rceil_>)$
where $\lceil P \rceil_> \equiv P \oplus_p snd_L$

abbreviation *drop-post* :: $('a, '\alpha \times '\beta)\ uexpr \Rightarrow ('a, '\beta)\ uexpr\ (\lfloor \cdot \rfloor_>)$
where $\lfloor P \rfloor_> \equiv P \downarrow_e snd_L$

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 var\ x \rceil_< = \x
by (*alpha-tac*)

lemma *lift-post-var* [*simp*]:
 $\lceil var\ x \rceil_> = \x'

by (*alpha-tac*)

12.3 Substitution Laws

lemma *pre-var-subst* [*usubst*]:
 $\sigma(\$x \mapsto_s \ll v \gg) \dagger [P]_< = \sigma \dagger [P[\ll v \gg / \&x]]_<$
 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 $x :: ('a \Rightarrow 'a)$
 shows $\$x' \# [p]_<$
 by (*pred-auto*)

end

13 Predicate Calculus Laws

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

13.1 Propositional Logic

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

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

lemma *taut-true* [*simp*]: *'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!*]:
 $\ll 'A \wedge B'; \ll 'A'; 'B' \gg \Rightarrow P \gg \Rightarrow P$
 by (*rel-auto*)

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

lemma *taut-shEx-elim*:
 $\ll '(\exists x \cdot P x)'; \bigwedge x. \Sigma \# P x; \bigwedge x. 'P x' \Rightarrow Q \gg \Rightarrow Q$
 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*:

$\forall x y. \text{ 'P x y' } \Longrightarrow \forall (x, y). x \cdot P x y$
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*:

$\text{ 'P } \llbracket \llbracket b \rrbracket / \&\mathbf{v} \rrbracket \text{ ' } = \llbracket P \rrbracket_e b$
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*:

$\llbracket \bigwedge x. P_1(x) = P_2(x); \bigwedge x. 'P_1(x) \Rightarrow Q_1(x) =_u Q_2(x)' \rrbracket \Rightarrow$
 $(\bigcap x \mid P_1(x) \cdot Q_1(x)) = (\bigcap x \mid P_2(x) \cdot Q_2(x))$
 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]: $\sigma \uparrow (\bigsqcap A) = (\bigsqcap \{\sigma \uparrow P \mid P. P \in A\})$
 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 uexpr-eq-iff, auto simp add: conj-upred-def USUP.rep-eq inf-uexpr.rep-eq bop.rep-eq lit.rep-eq*)

lemma *USUP-conj-USUP [simp]*: $((\bigsqcup P \in A \cdot F(P)) \wedge (\bigsqcup P \in A \cdot G(P))) = (\bigsqcup P \in A \cdot F(P) \wedge G(P))$

by (*simp add: upred-defs bop.rep-eq lit.rep-eq, pred-auto*)

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

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

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

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

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

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

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

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

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

lemma *UINF-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 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*:
 $(\bigcap i \in \{0..<\text{length}(xs)\} \cdot f (xs ! i)) = \text{foldr } (\vee) (map f xs) \text{ false}$
apply (*induct xs*)
apply (*rel-auto*)
apply (*simp add: UINF-upto-expand-first UINF-Suc-shift*)
done

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

lemma *UINF-refines*:
 $\llbracket \bigwedge i. i \in I \implies P \sqsubseteq Q i \rrbracket \implies P \sqsubseteq (\bigcap i \in I \cdot Q i)$
by (*simp add: UINF-as-Sup-collect, metis SUP-least*)

lemma *UINF-refines'*:
assumes $\bigwedge i. P \sqsubseteq Q(i)$
shows $P \sqsubseteq (\bigcap i \cdot Q(i))$
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]*: $(x =_u x) = \text{true}$
by (*pred-auto*)

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

lemma *eq-cong-left*:
assumes $vwb\text{-lens } x \ \$x \ \sharp \ Q \ \$x' \ \sharp \ Q \ \$x \ \sharp \ R \ \$x' \ \sharp \ R$
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 $x :: ('a \implies 'a)$
assumes $vwb\text{-lens } x$
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 $x :: ('a \implies 'a)$
assumes $vwb\text{-lens } x$

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 $(\$x \wedge Q) = (\$x \wedge Q\llbracket true/\$x \rrbracket)$
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 $(\neg \$x \wedge Q) = (\neg \$x \wedge Q\llbracket false/\$x \rrbracket)$
using *assms*
by $(pred-auto, metis\ (full-types)\ vwb-lens-wb\ wb-lens.get-put, metis\ (full-types)\ vwb-lens-wb\ wb-lens.get-put)$

lemma *upred-eq-true [simp]*: $(p =_u true) = p$

by $(pred-auto)$

lemma *upred-eq-false [simp]*: $(p =_u false) = (\neg p)$

by $(pred-auto)$

lemma *upred-true-eq [simp]*: $(true =_u p) = p$

by $(pred-auto)$

lemma *upred-false-eq [simp]*: $(false =_u p) = (\neg p)$

by $(pred-auto)$

lemma *conj-var-subst*:

assumes *vwb-lens x*
shows $(P \wedge var\ x =_u v) = (P\llbracket v/x \rrbracket \wedge var\ x =_u v)$
using *assms*
by $(pred-simp, (metis\ (full-types)\ vwb-lens-def\ wb-lens.get-put)+)$

13.4 HOL Variable Quantifiers

lemma *shEx-unbound [simp]*: $(\exists\ x \cdot P) = P$

by $(pred-auto)$

lemma *shEx-bool [simp]*: $shEx\ P = (P\ True \vee P\ False)$

by $(pred-simp, metis\ (full-types))$

lemma *shEx-commute*: $(\exists\ x \cdot \exists\ y \cdot P\ x\ y) = (\exists\ y \cdot \exists\ x \cdot P\ x\ y)$

by $(pred-auto)$

lemma *shEx-cong*: $\llbracket \bigwedge\ x. P\ x = Q\ x \rrbracket \implies shEx\ P = shEx\ Q$

by $(pred-auto)$

lemma *shEx-insert*: $(\exists\ x \in insert_u\ y\ A \cdot P(x)) = (P(x)\llbracket x \rightarrow y \rrbracket \vee (\exists\ x \in A \cdot P(x)))$

by $(pred-auto)$

lemma *shEx-one-point*: $(\exists\ x \cdot \langle x \rangle =_u v \wedge P(x)) = P(x)\llbracket x \rightarrow v \rrbracket$

by $(rel-auto)$

lemma *shAll-unbound [simp]*: $(\forall\ x \cdot P) = P$

by $(pred-auto)$

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

lemma *shAll-cong*: $\llbracket \bigwedge x. P\ x = Q\ x \rrbracket \implies shAll\ P = shAll\ Q$
by (*pred-auto*)

Quantifier lifting

named-theorems *uquant-lift*

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

lemma *shEx-lift-conj-2* [*uquant-lift*]:
 $(P \wedge (\exists x \cdot Q(x))) = (\exists x \cdot P \wedge Q(x))$
by (*pred-auto*)

13.5 Case Splitting

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

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

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

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

lemma *bool-eq-splitI*:
assumes *vwb-lens* *x* $P\llbracket true/x \rrbracket = Q\llbracket true/x \rrbracket$ $P\llbracket false/x \rrbracket = Q\llbracket false/x \rrbracket$
shows $P = Q$
by (*metis* (*full-types*) *assms* *eq-split-subst* *false-alt-def* *true-alt-def*)

lemma *subst-bool-split*:
assumes *vwb-lens* *x*
shows $\langle P \rangle = \langle (P\llbracket false/x \rrbracket \wedge P\llbracket true/x \rrbracket) \rangle$
proof –
from *assms* **have** $\langle P \rangle = (\forall v. \langle P\llbracket \llbracket v \rrbracket / x \rrbracket \rangle)$
by (*subst* *taut-split-subst*[*of* *x*], *auto*)
also have $\dots = \langle P\llbracket \llbracket True \rrbracket / x \rrbracket \rangle \wedge \langle P\llbracket \llbracket False \rrbracket / x \rrbracket \rangle$
by (*metis* (*mono-tags*, *lifting*))

also have ... = ‘ $(P\llbracket false/x \rrbracket \wedge P\llbracket true/x \rrbracket)$ ’,
 by (pred-auto)
 finally show *?thesis* .
 qed

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

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

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

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

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

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

lemma *cond-assoc*: $((P \triangleleft b \triangleright Q) \triangleleft c \triangleright R) = (P \triangleleft b \wedge c \triangleright (Q \triangleleft c \triangleright R))$ **by** (*pred-auto*)

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

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

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

lemma *cond-conj-not*: $((P \triangleleft b \triangleright Q) \wedge (\neg b)) = (Q \wedge (\neg b))$
by (*rel-auto*)

lemma *cond-and-T-integrate*:

$((P \wedge b) \vee (Q \triangleleft b \triangleright R)) = ((P \vee Q) \triangleleft b \triangleright R)$
by (*pred-auto*)

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

lemma *cond-L7*: $(P \triangleleft b \triangleright (P \triangleleft c \triangleright Q)) = (P \triangleleft b \vee c \triangleright Q)$ **by** (*pred-auto*)

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

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

lemma *cond-imp-distr*:

$((P \implies Q) \triangleleft b \triangleright (R \implies S)) = ((P \triangleleft b \triangleright R) \implies (Q \triangleleft b \triangleright S))$ **by** (*pred-auto*)

lemma *cond-eq-distr*:

$((P \Leftrightarrow Q) \triangleleft b \triangleright (R \Leftrightarrow S)) = ((P \triangleleft b \triangleright R) \Leftrightarrow (Q \triangleleft b \triangleright S))$ **by** (*pred-auto*)

lemma *cond-conj-distr*: $(P \wedge (Q \triangleleft b \triangleright S)) = ((P \wedge Q) \triangleleft b \triangleright (P \wedge S))$ **by** (*pred-auto*)

lemma *cond-disj-distr*: $(P \vee (Q \triangleleft b \triangleright S)) = ((P \vee Q) \triangleleft b \triangleright (P \vee S))$ **by** (*pred-auto*)

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

lemma *cond-conj*: $P \triangleleft b \wedge c \triangleright Q = (P \triangleleft c \triangleright Q) \triangleleft b \triangleright Q$
by (*pred-auto*)

lemma *spec-cond-dist*: $(P \Rightarrow (Q \triangleleft b \triangleright R)) = ((P \Rightarrow Q) \triangleleft b \triangleright (P \Rightarrow R))$
by (*pred-auto*)

lemma *cond-USUP-dist*: $(\bigsqcup P \in S \cdot F(P)) \triangleleft b \triangleright (\bigsqcup P \in S \cdot G(P)) = (\bigsqcup P \in S \cdot F(P) \triangleleft b \triangleright G(P))$
by (*pred-auto*)

lemma *cond-UINF-dist*: $(\prod P \in S \cdot F(P)) \triangleleft b \triangleright (\prod P \in S \cdot G(P)) = (\prod P \in S \cdot F(P) \triangleleft b \triangleright G(P))$
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*:
 $(P1 \triangleleft b \triangleright Q1 \wedge P2 \triangleleft b \triangleright Q2) = (P1 \wedge P2) \triangleleft b \triangleright (Q1 \wedge Q2)$
by *pred-auto*

lemma *disj-conds*:
 $(P1 \triangleleft b \triangleright Q1 \vee P2 \triangleleft b \triangleright Q2) = (P1 \vee P2) \triangleleft b \triangleright (Q1 \vee Q2)$
by *pred-auto*

lemma *cond-mono*:
 $\llbracket P1 \sqsubseteq P2; Q1 \sqsubseteq Q2 \rrbracket \implies (P1 \triangleleft b \triangleright Q1) \sqsubseteq (P2 \triangleleft b \triangleright Q2)$
by (*rel-auto*)

lemma *cond-monotonic*:
 $\llbracket \text{mono } P; \text{mono } Q \rrbracket \implies \text{mono } (\lambda X. P X \triangleleft b \triangleright Q X)$
by (*simp add: mono-def, rel-blast*)

13.9 Additional Expression Laws

lemma *le-pred-refl [simp]*:
fixes $x :: ('a::\text{preorder}, 'a) \text{ uexpr}$
shows $(x \leq_u x) = \text{true}$
by (*pred-auto*)

lemma *uzero-le-laws [simp]*:
 $(0 :: ('a::\{\text{linordered-semidom}\}, 'a) \text{ uexpr}) \leq_u \text{numeral } x = \text{true}$

$(1 :: ('a::\{\text{linordered-semidom}\}, 'a) \text{ uexpr}) \leq_u \text{numeral } x = \text{true}$
 $(0 :: ('a::\{\text{linordered-semidom}\}, 'a) \text{ uexpr}) \leq_u 1 = \text{true}$
by (*pred-simp*)+

lemma *unumeral-le-1* [*simp*]:
assumes ($\text{numeral } i :: 'a::\{\text{numeral,ord}\} \leq \text{numeral } j$)
shows ($\text{numeral } i :: ('a, 'a) \text{ uexpr}) \leq_u \text{numeral } j = \text{true}$
using *assms* **by** (*pred-auto*)

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

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

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

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

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

13.10 Refinement By Observation

Function to obtain the set of observations of a predicate

definition *obs-upred* :: $'a \text{ upred} \Rightarrow 'a \text{ set } ([_])_o$
where [*upred-defs*]: $[\![P]\!]_o = \{b. [\![P]\!]_e b\}$

lemma *obs-upred-refine-iff*:
 $P \sqsubseteq Q \iff [\![Q]\!]_o \subseteq [\![P]\!]_o$
by (*pred-auto*)

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

lemma *refine-by-obs*:
assumes $x \bowtie y \text{ bij-lens } (x +_L y) \ y \# P \ y \# Q \ \{v. 'P[\![\llbracket v \rrbracket/x]\!]\} \subseteq \{v. 'Q[\![\llbracket v \rrbracket/x]\!]\}$
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=get_xb 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 $'\alpha \text{ health} = '\alpha \text{ upred} \Rightarrow '\alpha \text{ upred}$

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 is $H \equiv (H\ P = P)$

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

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

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

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

abbreviation *Healthy-carrier* :: ' α health \Rightarrow ' α 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 is H
shows $F(P)$ 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-case-prod [closure]*:
 $\llbracket \bigwedge x\ y. P\ x\ y \text{ is } H \rrbracket \implies \text{case-prod } P\ v \text{ is } H$
by (*simp add: prod.case-eq-if*)

lemma *Healthy-SUPREMUM*:
 $A \subseteq \llbracket H \rrbracket_H \implies \text{SUPREMUM } A\ H = \bigcap A$
by (*drule Healthy-carrier-image, presburger*)

lemma *Healthy-INFIMUM*:
 $A \subseteq \llbracket H \rrbracket_H \implies \text{INFIMUM } A\ H = \bigcup A$
by (*drule Healthy-carrier-image, presburger*)

lemma *Healthy-nu [closure]*:
assumes *mono* $F\ F \in \llbracket id \rrbracket_H \rightarrow \llbracket H \rrbracket_H$
shows $\nu\ F$ is H

by (metis (mono-tags) Healthy-def Healthy-func assms eq-id-iff lfp-unfold)

lemma *Healthy-mu* [closure]:

assumes *mono* $F \ F \in \llbracket id \rrbracket_H \rightarrow \llbracket H \rrbracket_H$

shows $\mu \ F$ 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*: $\text{Idempotent } f \implies f \circ f \circ A = f \circ A$
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*:
 $\text{Conjunctive}(H) \implies \text{Idempotent}(H)$
by (*auto* *simp* *add*: *Conjunctive-def Idempotent-def*)

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

lemma *Conjunctive-conj*:
assumes *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]:

$\llbracket \text{Continuous } H; \bigwedge i. i \in A \implies P(i) \text{ is } H; A \neq \{\} \rrbracket \implies (\bigcap_{i \in A} \cdot P(i)) \text{ is } H$
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 $(\bigcap_{(i,j) \in A} \cdot P \ i \ j) \text{ is } H$

proof –

have $(\bigcap_{(i,j) \in A} \cdot P \ i \ j) = (\bigcap_{x \in A} \cdot P \ (\text{fst } x) \ (\text{snd } x))$
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 $(\bigcap_{(i,j,k) \in A} \cdot P \ i \ j \ k) \text{ is } H$

proof –

have $(\bigcap_{(i,j,k) \in A} \cdot P \ i \ j \ k) = (\bigcap_{x \in A} \cdot P \ (\text{fst } x) \ (\text{fst } (\text{snd } x)) \ (\text{snd } (\text{snd } x)))$
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-mem-Continuous-closed-quad* [closure]:

assumes *Continuous* $H \bigwedge i j k l. (i, j, k, l) \in A \implies P \ i \ j \ k \ l \text{ is } H \ A \neq \{\}$
shows $(\bigcap_{(i,j,k,l) \in A} \cdot P \ i \ j \ k \ l) \text{ is } H$

proof –

have $(\bigcap_{(i,j,k,l) \in A} \cdot P \ i \ j \ k \ l) = (\bigcap_{x \in A} \cdot P \ (\text{fst } x) \ (\text{fst } (\text{snd } x)) \ (\text{fst } (\text{snd } (\text{snd } x))) \ (\text{snd } (\text{snd } (\text{snd } x))))$
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-mem-Continuous-closed-quint* [closure]:

assumes *Continuous* $H \bigwedge i j k l m. (i, j, k, l, m) \in A \implies P \ i \ j \ k \ l \ m \text{ is } H \ A \neq \{\}$
shows $(\bigcap_{(i,j,k,l,m) \in A} \cdot P \ i \ j \ k \ l \ m) \text{ is } H$

proof –

have $(\bigcap_{(i,j,k,l,m) \in A} \cdot P \ i \ j \ k \ l \ m)$
 $= (\bigcap_{x \in A} \cdot P \ (\text{fst } x) \ (\text{fst } (\text{snd } x)) \ (\text{fst } (\text{snd } (\text{snd } x))) \ (\text{fst } (\text{snd } (\text{snd } (\text{snd } x)))) \ (\text{snd } (\text{snd } (\text{snd } (\text{snd } (\text{snd } x)))))$
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-ind-closed* [closure]:
 assumes $\text{Continuous } H \wedge i. P\ i = \text{true} \wedge i. Q\ i \text{ is } H$
 shows $\text{UINF } P\ Q \text{ is } H$
proof –
 from *assms*(2) have $\text{UINF } P\ Q = (\bigsqcap i. Q\ i)$
 by (*rel-auto*)
 also have ... is H
 using *UINF-mem-Continuous-closed*[of $H\ UNIV\ P$]
 by (*simp add: Sup-Continuous-closed UINF-as-Sup-collect' assms*)
 finally show ?thesis .
 qed

All continuous functions are also Scott-continuous

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

lemma *USUP-healthy*: $A \subseteq \llbracket H \rrbracket_H \implies (\bigsqcup P \in A. F(P)) = (\bigsqcup P \in A. F(H(P)))$
 by (*rule USUP-cong, simp add: Healthy-subset-member*)

lemma *UINF-healthy*: $A \subseteq \llbracket H \rrbracket_H \implies (\bigsqcap P \in A. F(P)) = (\bigsqcap P \in A. F(H(P)))$
 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 $\text{in}\alpha :: ('a \implies 'a \times 'b)$ **where**
 [*lens-defs*]: $\text{in}\alpha = \text{fst}_L$

definition $\text{out}\alpha :: ('a \implies 'a \times 'b)$ **where**
 [*lens-defs*]: $\text{out}\alpha = \text{snd}_L$

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

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

lemma *var-in-alpha* [simp]: *x ;_L in α = ivar x*
 by (*simp add: fst-lens-def in α -def in-var-def*)

lemma *var-out-alpha* [simp]: *x ;_L out α = ovar x*
 by (*simp add: out α -def out-var-def snd-lens-def*)

lemma *drop-pre-inv* [simp]: $\llbracket \text{out}\alpha \# p \rrbracket \implies \llbracket p \rrbracket_{<} = p$
 by (*pred-simp*)

lemma *usubst-lookup-ivar-unrest* [usubst]:
in α # $\sigma \implies \langle \sigma \rangle_s (\text{ivar } x) = \x
 by (*rel-simp*, *metis fstI*)

lemma *usubst-lookup-ovar-unrest* [usubst]:
out α # $\sigma \implies \langle \sigma \rangle_s (\text{ovar } x) = \x'
 by (*rel-simp*, *metis sndI*)

lemma *out-alpha-in-indep* [simp]:
out α \bowtie in-var x in-var x \bowtie out α
 by (*simp-all add: in-var-def out α -def lens-indep-def fst-lens-def snd-lens-def lens-comp-def*)

lemma *in-alpha-out-indep* [simp]:
in α \bowtie out-var x out-var x \bowtie in α
 by (*simp-all add: in-var-def in α -def lens-indep-def fst-lens-def lens-comp-def*)

The following two functions lift a predicate substitution to a relational one.

abbreviation *usubst-rel-lift* :: *' α usubst \Rightarrow (' α \times ' β) usubst ($\llbracket - \rrbracket_s$)* **where**
 $\llbracket \sigma \rrbracket_s \equiv \sigma \oplus_s \text{in}\alpha$

abbreviation *usubst-rel-drop* :: *(' α \times ' α) usubst \Rightarrow ' α usubst ($\llbracket - \rrbracket_s$)* **where**
 $\llbracket \sigma \rrbracket_s \equiv \sigma \upharpoonright_s \text{in}\alpha$

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 *' α cond* = *' α upred*
type-synonym *(' α , ' β) urel* = *(' α \times ' β) upred*
type-synonym *' α hrel* = *(' α \times ' α) upred*
type-synonym *('a, ' α) hexpr* = *('a, ' α \times ' α) uexpr*

translations
(type) (' α , ' β) urel <= *(type) (' α \times ' β) upred*

We set up some overloaded constants for sequential composition and the identity in case we want to overload their definitions later.

consts

$useq \quad :: 'a \Rightarrow 'b \Rightarrow 'c \text{ (infixr ;; 61)}$
 $uassigns \quad :: 'a \text{ usubst} \Rightarrow 'b \text{ (}\langle - \rangle_a\text{)}$
 $uskip \quad :: 'a \text{ (II)}$

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

definition $lift-rcond \text{ (}\lceil - \rceil_{<}\text{) where}$

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

abbreviation

$rcond \quad :: ('a, 'b) \text{ urel} \Rightarrow 'a \text{ cond} \Rightarrow ('a, 'b) \text{ urel} \Rightarrow ('a, 'b) \text{ urel}$
 $((\beta \triangleleft - \triangleright_r / -) [52, 0, 53] 52)$
where $(P \triangleleft b \triangleright_r Q) \equiv (P \triangleleft \lceil b \rceil_{<} \triangleright Q)$

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

lift-definition $seqr :: ('a, 'b) \text{ urel} \Rightarrow ('b, 'c) \text{ urel} \Rightarrow ('a \times 'c) \text{ upred}$
is $\lambda P Q b. b \in (\{p. P p\} O \{q. Q q\})$.

ad hoc-overloading

$useq \text{ seqr}$

We also set up a homogeneous sequential composition operator, and versions of *true* and *false* that are explicitly typed by a homogeneous alphabet.

abbreviation $seqh \quad :: 'a \text{ hrel} \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel (infixr ;;}_h\text{ 61) where}$

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

abbreviation $truer \quad :: 'a \text{ hrel (true}_h\text{) where}$

$truer \equiv true$

abbreviation $falserr \quad :: 'a \text{ hrel (false}_h\text{) where}$

$falserr \equiv false$

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 $conv-r \quad :: ('a, 'a \times 'b) \text{ uexpr} \Rightarrow ('a, 'b \times 'a) \text{ uexpr (- [999] 999)}$

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 $assigns-r \quad :: 'a \text{ usubst} \Rightarrow 'a \text{ hrel}$

is $\lambda \sigma (b, b'). b' = \sigma(b)$.

ad hoc-overloading

$uassigns \text{ assigns-r}$

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

definition $skip-r :: 'α \ hrel \ \mathbf{where}$

$[urel-defs]: skip-r = assigns-r \ id$

ad hoc-overloading

$uskip \ skip-r$

Non-deterministic assignment, also known as “choose”, assigns an arbitrarily chosen value to the given variable

definition $nd-assign :: ('a \implies 'α) \Rightarrow 'α \ hrel \ \mathbf{where}$

$[urel-defs]: nd-assign \ x = (\bigsqcap \ v \cdot assigns-r \ [x \mapsto_s \ll v \gg])$

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

definition $seqr-iter :: 'a \ list \Rightarrow ('a \Rightarrow 'b \ hrel) \Rightarrow 'b \ hrel \ \mathbf{where}$

$[urel-defs]: seqr-iter \ xs \ P = foldr \ (\lambda \ i \ Q. P(i) ;; Q) \ xs \ II$

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

abbreviation $assign-r :: ('t \implies 'α) \Rightarrow ('t, 'α) \ uexpr \Rightarrow 'α \ hrel$

where $assign-r \ x \ v \equiv \langle [x \mapsto_s \ v] \rangle_a$

abbreviation $assign-2-r ::$

$('t1 \implies 'α) \Rightarrow ('t2 \implies 'α) \Rightarrow ('t1, 'α) \ uexpr \Rightarrow ('t2, 'α) \ uexpr \Rightarrow 'α \ hrel$

where $assign-2-r \ x \ y \ u \ v \equiv assigns-r \ [x \mapsto_s \ u, y \mapsto_s \ v]$

We also define the alphabetised skip operator that identifies all input and output variables in the given alphabet lens. All other variables are unrestricted. We also set up syntax for it.

definition $skip-ra :: ('β, 'α) \ lens \Rightarrow 'α \ hrel \ \mathbf{where}$

$[urel-defs]: skip-ra \ v = (\$v' =_u \$v)$

Similarly, we define the alphabetised assignment operator.

definition $assigns-ra :: 'α \ usubst \Rightarrow ('β, 'α) \ lens \Rightarrow 'α \ hrel \ (\langle - \rangle_-) \ \mathbf{where}$

$\langle \sigma \rangle_a = (\lceil \sigma \rceil_s \uparrow skip-ra \ a)$

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. They are the same as tests, as in Kleene Algebra with Tests [16, 1] (KAT), which embeds a Boolean algebra into a Kleene algebra to represent conditions.

definition $rassume :: 'α \ upred \Rightarrow 'α \ hrel \ \mathbf{where}$

$[urel-defs]: rassume \ c = II \triangleleft c \triangleright_r \ false$

definition $raassert :: 'α \ upred \Rightarrow 'α \ hrel \ \mathbf{where}$

$[urel-defs]: raassert \ c = II \triangleleft c \triangleright_r \ true$

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

definition $while-top :: 'α \ cond \Rightarrow 'α \ hrel \Rightarrow 'α \ hrel \ \mathbf{where}$

$[urel-defs]: while-top \ b \ P = (\nu \ X \cdot (P ;; X) \triangleleft b \triangleright_r \ II)$

definition $while-bot :: 'α \ cond \Rightarrow 'α \ hrel \Rightarrow 'α \ hrel \ \mathbf{where}$

$[urel-defs]: while-bot \ b \ P = (\mu \ X \cdot (P ;; X) \triangleleft b \triangleright_r \ II)$

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

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

While loops with invariant decoration – total correctness.

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

While loops with invariant and variant decorations – total correctness.

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

syntax

$\text{-uassume} \quad :: \text{uexp} \Rightarrow \text{logic } ([\cdot]^\top)$
 $\text{-uassume} \quad :: \text{uexp} \Rightarrow \text{logic } (?[\cdot])$
 $\text{-uassert} \quad :: \text{uexp} \Rightarrow \text{logic } (\{\cdot\}_\perp)$
 $\text{-uwhile} \quad :: \text{uexp} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{while}^\top - \text{do} - \text{od})$
 $\text{-uwhile-top} \quad :: \text{uexp} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{while} - \text{do} - \text{od})$
 $\text{-uwhile-bot} \quad :: \text{uexp} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{while}_\perp - \text{do} - \text{od})$
 $\text{-uwhile-inv} \quad :: \text{uexp} \Rightarrow \text{uexp} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{while} - \text{invr} - \text{do} - \text{od})$
 $\text{-uwhile-inv-bot} \quad :: \text{uexp} \Rightarrow \text{uexp} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{while}_\perp - \text{invr} - \text{do} - \text{od } 71)$
 $\text{-uwhile-vrt} \quad :: \text{uexp} \Rightarrow \text{uexp} \Rightarrow \text{uexp} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{while} - \text{invr} - \text{vrt} - \text{do} - \text{od})$

translations

$\text{-uassume } b == \text{CONST } \text{rassume } b$
 $\text{-uassert } b == \text{CONST } \text{rassert } b$
 $\text{-uwhile } b \text{ } P == \text{CONST } \text{while-top } b \text{ } P$
 $\text{-uwhile-top } b \text{ } P == \text{CONST } \text{while-top } b \text{ } P$
 $\text{-uwhile-bot } b \text{ } P == \text{CONST } \text{while-bot } b \text{ } P$
 $\text{-uwhile-inv } b \text{ } p \text{ } S == \text{CONST } \text{while-inv } b \text{ } p \text{ } S$
 $\text{-uwhile-inv-bot } b \text{ } p \text{ } S == \text{CONST } \text{while-inv-bot } b \text{ } p \text{ } S$
 $\text{-uwhile-vrt } b \text{ } p \text{ } v \text{ } S == \text{CONST } \text{while-vrt } b \text{ } p \text{ } v \text{ } S$

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

definition $\text{rel-var-res} :: 'a \text{ hrel} \Rightarrow ('a \Longrightarrow 'a) \Rightarrow 'a \text{ hrel}$ (**infix** \downarrow_α 80) **where**
 $[\text{urel-defs}]: P \downarrow_\alpha x = (\exists \$x \cdot \exists \$x' \cdot P)$

Alphabet extension and restriction add additional variables by the given lens in both their primed and unprimed versions.

definition $\text{rel-aext} :: 'b \text{ hrel} \Rightarrow ('b \Longrightarrow 'a) \Rightarrow 'a \text{ hrel}$
where $[\text{upred-defs}]: \text{rel-aext } P \text{ } a = P \oplus_p (a \times_L a)$

definition $\text{rel-ares} :: 'a \text{ hrel} \Rightarrow ('b \Longrightarrow 'a) \Rightarrow 'b \text{ hrel}$
where $[\text{upred-defs}]: \text{rel-ares } P \text{ } a = (P \downarrow_p (a \times a))$

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 $\text{frame} :: ('a \Longrightarrow 'a) \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel}$ **where**
 $[\text{urel-defs}]: \text{frame } a \text{ } P = (P \wedge \$\mathbf{v}' =_u \$\mathbf{v} \oplus \$\mathbf{v}' \text{ on } \&a)$

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

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

definition *rel-frex* :: ($'β \Rightarrow 'α \Rightarrow 'β \text{ hrel} \Rightarrow 'α \text{ hrel}$ **where**
 $[upred-defs]: \text{rel-frex } a \ P = \text{frame } a \ (\text{rel-aext } P \ a)$

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* :: ($'a \Rightarrow 'α \Rightarrow 'α \text{ hrel} \Rightarrow 'α \text{ hrel}$ **where**
 $[urel-defs]: \text{nameset } a \ P = (P \upharpoonright_v \{\$ \mathbf{v}, \$a'\})$

15.3 Syntax Translations

syntax

- Alternative traditional conditional syntax
- utp-if :: $uexp \Rightarrow logic \Rightarrow logic \Rightarrow logic \ ((\text{if}_u \ (-) / \text{then } (-) / \text{else } (-)) \ [0, 0, 71] \ 71)$
- Iterated sequential composition
- seqr-iter :: $pttrn \Rightarrow 'a \text{ list} \Rightarrow 'σ \text{ hrel} \Rightarrow 'σ \text{ hrel} \ ((3;; - : - \cdot / -) \ [0, 0, 10] \ 10)$
- Single and multiple assignement
- assignment :: $svids \Rightarrow uexprs \Rightarrow 'α \text{ hrel} \ ((-') := '(-))$
- assignment :: $svids \Rightarrow uexprs \Rightarrow 'α \text{ hrel} \ (\text{infixr} := 62)$
- Non-deterministic assignment
- nd-assign :: $svids \Rightarrow logic \ (- := * \ [62] \ 62)$
- Substitution constructor
- mk-usubst :: $svids \Rightarrow uexprs \Rightarrow 'α \text{ usubst}$
- 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 :: $logic \Rightarrow salpha \Rightarrow logic \ (\text{infixl } \oplus_r \ 90)$
- Relational Alphabet Restriction
- rel-ares :: $logic \Rightarrow salpha \Rightarrow logic \ (\text{infixl } \upharpoonright_r \ 90)$
- Frame Extension
- rel-frex :: $salpha \Rightarrow logic \Rightarrow logic \ (-:[-]^+ \ [99,0] \ 100)$
- Nameset
- nameset :: $salpha \Rightarrow logic \Rightarrow logic \ (ns \ - \cdot - \ [0,999] \ 999)$

translations

- utp-if $b \ P \ Q \Rightarrow P \triangleleft b \triangleright_r Q$
- ;; $x : l \cdot P \equiv (CONST \text{ seqr-iter}) \ l \ (\lambda x. P)$
- mk-usubst $\sigma \ (-\text{svid-unit } x) \ v \equiv \sigma(\&x \mapsto_s v)$
- mk-usubst $\sigma \ (-\text{svid-list } x \ xs) \ (-uexprs \ v \ vs) \equiv (-\text{mk-usubst } (\sigma(\&x \mapsto_s v)) \ xs \ vs)$
- assignment $xs \ vs \Rightarrow CONST \ uassigns \ (-\text{mk-usubst } (CONST \text{ id}) \ xs \ vs)$
- assignment $x \ v \leq CONST \ uassigns \ (CONST \text{ subst-upd } (CONST \text{ id}) \ x \ v)$
- assignment $x \ v \leq -\text{assignment } (-\text{spvar } x) \ v$
- nd-assign $x \Rightarrow CONST \ nd\text{-assign } (-\text{mk-svid-list } x)$
- nd-assign $x \leq CONST \ nd\text{-assign } x$
- $x, y := u, v \leq CONST \ uassigns \ (CONST \text{ subst-upd } (CONST \text{ subst-upd } (CONST \text{ id}) \ (CONST \text{ svar } u) \ (CONST \text{ svar } y) \ v)$

```

-skip-ra v  $\Rightarrow$  CONST skip-ra v
-frame x P  $\Rightarrow$  CONST frame x P
-frame (-salphaset (-salphamk x)) P  $\leq$  CONST frame x P
-antiframe x P  $\Rightarrow$  CONST antiframe x P
-antiframe (-salphaset (-salphamk x)) P  $\leq$  CONST antiframe x P
-nameset x P  $\equiv$  CONST nameset x P
-rel-aext P a  $\equiv$  CONST rel-aext P a
-rel-ares P a  $\equiv$  CONST rel-ares P a
-rel-frext a P  $\equiv$  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  $\leq$  (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* :: (α, β) *urel* \Rightarrow bool
where [*urel-defs*]: *ufunctional* *R* \longleftrightarrow $II \sqsubseteq R^-$;; *R*

definition *uinj* :: (α, β) *urel* \Rightarrow bool
where [*urel-defs*]: *uinj* *R* \longleftrightarrow $II \sqsubseteq R$;; R^-

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

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

— Configuration for UTP tactics.

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

15.5 Introduction laws

lemma *urel-refine-ext*:

$$\llbracket \bigwedge s s'. P[\llbracket s \rrbracket, \llbracket s' \rrbracket / \mathbf{v}, \mathbf{v}'] \rrbracket \sqsubseteq Q[\llbracket s \rrbracket, \llbracket s' \rrbracket / \mathbf{v}, \mathbf{v}'] \rrbracket \implies P \sqsubseteq Q$$

by (rel-auto)

lemma *urel-eq-ext*:

$\llbracket \bigwedge s s'. P[\llbracket s \gg, \ll s' \gg / \$v, \$v' \rrbracket = Q[\llbracket s \gg, \ll s' \gg / \$v, \$v' \rrbracket] \rrbracket \implies P = Q$
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]:

$x \# p1 \implies \$x' \# [p1]_>$

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*]:
 $out\alpha \# [\sigma]_s$
by (*rel-auto*)

lemma *unrest-in-rel-aext* [*unrest*]: $x \bowtie y \implies \$y \# P \oplus_r x$
by (*simp add: rel-aext-def unrest-aext-indep*)

lemma *unrest-out-rel-aext* [*unrest*]: $x \bowtie y \implies \$y' \# P \oplus_r x$
by (*simp add: rel-aext-def unrest-aext-indep*)

lemma *rel-aext-false* [*alpha*]:
 $false \oplus_r a = false$
by (*pred-auto*)

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

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 mwb-lens.put-put vwb-lens-mwb)
 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) mwb-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 ... = ((($\exists x \cdot \exists x' \cdot P$) $\wedge x' =_u x$) ;; (($\exists x \cdot \exists x' \cdot Q$) $\wedge x' =_u x$)) $\wedge x' =_u x$)
 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 Sequent Calculus

```

theory utp-sequent
  imports utp-pred-laws
begin

definition sequent :: ' $\alpha$  upred  $\Rightarrow$  ' $\alpha$  upred  $\Rightarrow$  bool (infixr  $\Vdash$  15) where
[upred-defs]: sequent  $P$   $Q = (Q \sqsubseteq P)$ 

abbreviation sequent-triv ( $\Vdash$  - [15] 15) where  $\Vdash P \equiv (true \Vdash P)$ 

translations
 $\Vdash P <= true \Vdash P$ 

lemma sTrue:  $P \Vdash true$ 
  by pred-auto

lemma sAx:  $P \Vdash P$ 
  by pred-auto

lemma sNotI:  $\Gamma \wedge P \Vdash false \implies \Gamma \Vdash \neg P$ 
  by pred-auto

lemma sConjI:  $\llbracket \Gamma \Vdash P; \Gamma \Vdash Q \rrbracket \implies \Gamma \Vdash P \wedge Q$ 
  by pred-auto

lemma sImplI:  $\llbracket (\Gamma \wedge P) \Vdash Q \rrbracket \implies \Gamma \Vdash (P \Rightarrow Q)$ 
  by pred-auto

end

```

18 Relational Calculus Laws

```

theory utp-rel-laws
  imports
    utp-rel
    utp-recursion
begin

```

18.1 Conditional Laws

```

lemma comp-cond-left-distr:
 $((P \triangleleft b \triangleright_r Q) ;; R) = ((P ;; R) \triangleleft b \triangleright_r (Q ;; R))$ 
  by (rel-auto)

lemma cond-seq-left-distr:
 $out\alpha \# b \implies ((P \triangleleft b \triangleright Q) ;; R) = ((P ;; R) \triangleleft b \triangleright (Q ;; R))$ 
  by (rel-auto)

lemma cond-seq-right-distr:
 $in\alpha \# b \implies (P ;; (Q \triangleleft b \triangleright R)) = ((P ;; Q) \triangleleft b \triangleright (P ;; R))$ 
  by (rel-auto)

```

Alternative expression of conditional using assumptions and choice

```

lemma rcond-rassume-expand:  $P \triangleleft b \triangleright_r Q = ([b]^\top ;; P) \sqcap ([(\neg b)]^\top ;; Q)$ 

```

by (rel-auto)

18.2 Precondition and Postcondition Laws

theorem *precond-equiv*:

$$P = (P ;; \text{true}) \longleftrightarrow (\text{out}\alpha \nmid P)$$

by (rel-auto)

theorem *postcond-equiv*:

$$P = (\text{true} ;; P) \longleftrightarrow (\text{in}\alpha \nmid P)$$

by (rel-auto)

lemma *precond-right-unit*: $\text{out}\alpha \nmid p \implies (p ;; \text{true}) = p$

by (metis *precond-equiv*)

lemma *postcond-left-unit*: $\text{in}\alpha \nmid p \implies (\text{true} ;; p) = p$

by (metis *postcond-equiv*)

theorem *precond-left-zero*:

assumes $\text{out}\alpha \nmid p \implies p \neq \text{false}$

shows $(\text{true} ;; p) = \text{true}$

using *assms* by (rel-auto)

theorem *feasible-iff-true-right-zero*:

$$P ;; \text{true} = \text{true} \longleftrightarrow \exists \text{out}\alpha \cdot P$$

by (rel-auto)

18.3 Sequential Composition Laws

lemma *seqr-assoc*: $(P ;; Q) ;; R = P ;; (Q ;; R)$

by (rel-auto)

lemma *seqr-left-unit* [*simp*]:

$$II ;; P = P$$

by (rel-auto)

lemma *seqr-right-unit* [*simp*]:

$$P ;; II = P$$

by (rel-auto)

lemma *seqr-left-zero* [*simp*]:

$$\text{false} ;; P = \text{false}$$

by *pred-auto*

lemma *seqr-right-zero* [*simp*]:

$$P ;; \text{false} = \text{false}$$

by *pred-auto*

lemma *impl-seqr-mono*: $\llbracket P \Rightarrow Q; R \Rightarrow S \rrbracket \implies (P ;; R) \Rightarrow (Q ;; S)$

by (*pred-blast*)

lemma *seqr-mono*:

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

by (*rel-blast*)

lemma *seqr-monotonic*:

$\llbracket \text{mono } P; \text{mono } Q \rrbracket \implies \text{mono } (\lambda X. P X ;; Q X)$
by (*simp add: mono-def, rel-blast*)

lemma *Monotonic-seqr-tail* [*closure*]:
assumes *Monotonic F*
shows *Monotonic* $(\lambda X. P ;; F(X))$
by (*simp add: assms monoD monoI seqr-mono*)

lemma *seqr-exists-left*:
 $((\exists \$x \cdot P) ;; Q) = (\exists \$x \cdot (P ;; Q))$
by (*rel-auto*)

lemma *seqr-exists-right*:
 $(P ;; (\exists \$x' \cdot Q)) = (\exists \$x' \cdot (P ;; Q))$
by (*rel-auto*)

lemma *seqr-or-distl*:
 $((P \vee Q) ;; R) = ((P ;; R) \vee (Q ;; R))$
by (*rel-auto*)

lemma *seqr-or-distr*:
 $(P ;; (Q \vee R)) = ((P ;; Q) \vee (P ;; R))$
by (*rel-auto*)

lemma *seqr-inf-distl*:
 $((P \sqcap Q) ;; R) = ((P ;; R) \sqcap (Q ;; R))$
by (*rel-auto*)

lemma *seqr-inf-distr*:
 $(P ;; (Q \sqcap R)) = ((P ;; Q) \sqcap (P ;; R))$
by (*rel-auto*)

lemma *seqr-and-distr-ufunc*:
 $\text{ufunctional } P \implies (P ;; (Q \wedge R)) = ((P ;; Q) \wedge (P ;; R))$
by (*rel-auto*)

lemma *seqr-and-distl-ujnj*:
 $\text{ujnj } R \implies ((P \wedge Q) ;; R) = ((P ;; R) \wedge (Q ;; R))$
by (*rel-auto*)

lemma *seqr-unfold*:
 $(P ;; Q) = (\exists v \cdot P \llbracket \llbracket v \rrbracket / \$v' \rrbracket \wedge Q \llbracket \llbracket v \rrbracket / \$v \rrbracket)$
by (*rel-auto*)

lemma *seqr-middle*:
assumes *vwb-lens x*
shows $(P ;; Q) = (\exists v \cdot P \llbracket \llbracket v \rrbracket / \$x' \rrbracket ;; Q \llbracket \llbracket v \rrbracket / \$x \rrbracket)$
using *assms*
by (*rel-auto',metis vwb-lens-wb wb-lens.source-stability*)

lemma *seqr-left-one-point*:
assumes *vwb-lens x*
shows $((P \wedge \$x' =_u \llbracket v \rrbracket) ;; Q) = (P \llbracket \llbracket v \rrbracket / \$x' \rrbracket ;; Q \llbracket \llbracket v \rrbracket / \$x \rrbracket)$
using *assms*
by (*rel-auto,metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-right-one-point*:

assumes *vwb-lens* *x*
shows $(P ;; (\$x =_u \ll v \gg \wedge Q)) = (P[\ll v \gg / \$x'] ;; Q[\ll v \gg / \$x])$
using *assms*
by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-left-one-point-true*:

assumes *vwb-lens* *x*
shows $((P \wedge \$x') ;; Q) = (P[\text{true} / \$x'] ;; Q[\text{true} / \$x])$
by (*metis assms seqr-left-one-point true-alt-def upred-eq-true*)

lemma *seqr-left-one-point-false*:

assumes *vwb-lens* *x*
shows $((P \wedge \neg \$x') ;; Q) = (P[\text{false} / \$x'] ;; Q[\text{false} / \$x])$
by (*metis assms false-alt-def seqr-left-one-point upred-eq-false*)

lemma *seqr-right-one-point-true*:

assumes *vwb-lens* *x*
shows $(P ;; (\$x \wedge Q)) = (P[\text{true} / \$x'] ;; Q[\text{true} / \$x])$
by (*metis assms seqr-right-one-point true-alt-def upred-eq-true*)

lemma *seqr-right-one-point-false*:

assumes *vwb-lens* *x*
shows $(P ;; (\neg \$x \wedge Q)) = (P[\text{false} / \$x'] ;; Q[\text{false} / \$x])$
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 *seq-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 $P ;; Q = (P[\text{true} / \$x'] ;; Q[\text{true} / \$x] \vee P[\text{false} / \$x'] ;; Q[\text{false} / \$x])$
using *assms*
by (*subst seqr-middle[of x]*, *simp-all*)

lemma *cond-inter-var-split*:

assumes *vwb-lens* *x*
shows $(P \triangleleft \$x' \triangleright Q) ;; R = (P[\text{true} / \$x'] ;; R[\text{true} / \$x] \vee Q[\text{false} / \$x'] ;; R[\text{false} / \$x])$

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 ... = $((P \wedge \$x') ;; R \vee (Q \wedge \neg \$x') ;; R)$
 by (rel-auto)
 also have ... = $(P \llbracket \text{true}/\$x' \rrbracket ;; R \llbracket \text{true}/\$x \rrbracket \vee Q \llbracket \text{false}/\$x' \rrbracket ;; R \llbracket \text{false}/\$x \rrbracket)$
 by (simp add: seqr-left-one-point-true seqr-left-one-point-false assms)
 finally show ?thesis .
 qed

theorem seqr-pre-transfer: $\text{in}\alpha \# q \implies ((P \wedge q) ;; R) = (P ;; (q^- \wedge R))$
 by (rel-auto)

theorem seqr-pre-transfer':
 $((P \wedge \lceil q \rceil_{>}) ;; R) = (P ;; (\lceil q \rceil_{<} \wedge R))$
 by (rel-auto)

theorem seqr-post-out: $\text{in}\alpha \# r \implies (P ;; (Q \wedge r)) = ((P ;; Q) \wedge r)$
 by (rel-blast)

lemma seqr-post-var-out:
 fixes $x :: (\text{bool} \implies '\alpha)$
 shows $(P ;; (Q \wedge \$x')) = ((P ;; Q) \wedge \$x')$
 by (rel-auto)

theorem seqr-post-transfer: $\text{out}\alpha \# q \implies (P ;; (q \wedge R)) = ((P \wedge q^-) ;; R)$
 by (rel-auto)

lemma seqr-pre-out: $\text{out}\alpha \# p \implies ((p \wedge Q) ;; R) = (p \wedge (Q ;; R))$
 by (rel-blast)

lemma seqr-pre-var-out:
 fixes $x :: (\text{bool} \implies '\alpha)$
 shows $((\$x \wedge P) ;; Q) = (\$x \wedge (P ;; Q))$
 by (rel-auto)

lemma seqr-true-lemma:
 $(P = (\neg ((\neg P) ;; \text{true}))) = (P = (P ;; \text{true}))$
 by (rel-auto)

lemma seqr-to-conj: $\llbracket \text{out}\alpha \# P; \text{in}\alpha \# Q \rrbracket \implies (P ;; Q) = (P \wedge Q)$
 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 rel-auto

lemma shEx-mem-lift-seq-1 [uquant-lift]:
 assumes $\text{out}\alpha \# A$
 shows $((\exists x \in A \cdot P x) ;; Q) = (\exists x \in A \cdot (P x ;; Q))$
 using assms by rel-blast

lemma shEx-lift-seq-2 [uquant-lift]:
 $(P ;; (\exists x \cdot Q x)) = (\exists x \cdot (P ;; Q x))$
 by rel-auto

lemma shEx-mem-lift-seq-2 [uquant-lift]:

assumes $in\alpha \# A$
shows $(P ;; (\exists x \in A \cdot Q\ x)) = (\exists x \in A \cdot (P ;; Q\ x))$
using *assms* **by** *rel-blast*

18.4 Iterated Sequential Composition Laws

lemma *iter-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*: $P ;; (\bigcap_{Q \in A} F(Q)) = (\bigcap_{Q \in A} P ;; F(Q))$
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*: $out\alpha \# b \implies (b \wedge II) = (II \wedge b^-)$
by (*rel-auto*)

lemma *pre-skip-post*: $([b]_< \wedge II) = (II \wedge [b]_>)$
by (*rel-auto*)

lemma *skip-var*:
fixes $x :: (bool \implies 'a)$
shows $(\$x \wedge II) = (II \wedge \$x')$
by (*rel-auto*)

lemma *skip-r-unfold*:
 $vwb\text{-}lens\ x \implies II = (\$x' =_u \$x \wedge II \upharpoonright_\alpha x)$
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*:
 $\llbracket vwb\text{-}lens\ y; x +_L y \approx_L 1_L; x \bowtie y \rrbracket \implies II|_{\alpha} x = II_y$
apply (*rel-auto*)
apply (*metis* (*no-types*, *lifting*) *lens-indep-def*)
apply (*metis* *vwb-lens.put-eq*)
done

18.7 Assignment Laws

lemma *assigns-subst* [*usubst*]:
 $[\sigma]_s \dagger \langle \varrho \rangle_a = \langle \varrho \circ \sigma \rangle_a$
by (*rel-auto*)

lemma *assigns-r-comp*: $(\langle \sigma \rangle_a ;; P) = ([\sigma]_s \dagger P)$
by (*rel-auto*)

lemma *assigns-r-feasible*:
 $(\langle \sigma \rangle_a ;; true) = true$
by (*rel-auto*)

lemma *assign-subst* [*usubst*]:
 $\llbracket mwb\text{-}lens\ x; mwb\text{-}lens\ y \rrbracket \implies [\$x \mapsto_s [u]_{<}] \dagger (y := v) = (x, y) := (u, [x \mapsto_s u] \dagger v)$
by (*rel-auto*)

lemma *assign-vacuous-skip*:
assumes *vwb-lens* *x*
shows $(x := \&x) = II$
using *assms* **by** *rel-auto*

The following law shows the case for the above law when *x* is only mainly-well behaved. We require that the state is one of those in which *x* is well defined using and assumption.

lemma *assign-vacuous-assume*:
assumes *mwb-lens* *x*
shows $[(\&\mathbf{v} \in_u \llbracket \mathcal{S}_x \rrbracket)]^\top ;; (x := \&x) = [(\&\mathbf{v} \in_u \llbracket \mathcal{S}_x \rrbracket)]^\top$
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*:

$\text{bij } f \implies \langle f \rangle_a^- = \langle \text{inv } f \rangle_a$

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*: $\text{mwb-lens } x \implies (x := \llbracket u \rrbracket ;; x := \llbracket v \rrbracket) = (x := \llbracket v \rrbracket)$

by (simp add: assigns-comp usubst)

lemma *assign-twice*: $\llbracket \text{mwb-lens } x; x \# f \rrbracket \implies (x := e ;; x := f) = (x := f)$

by (simp add: assigns-comp usubst unrest)

lemma *assign-commute*:

assumes $x \bowtie y \text{ } x \# f \text{ } y \# e$

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 $\text{out } \alpha \# b$

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

assumes $\text{out } \alpha \# b$

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*: $\text{inj } f \implies \text{uinj } \langle f \rangle_a$

by (rel-simp, simp add: inj-eq)

lemma *assigns-r-swap-uinj*:

$\llbracket \text{vwb-lens } x; \text{vwb-lens } y; x \bowtie y \rrbracket \implies \text{uinj } ((x, y) := (\&y, \&x))$

by (metis assigns-r-uinj pr-var-def swap-usubst-inj)

lemma *assign-unfold*:

$\text{vwb-lens } x \implies (x := v) = (\$x' =_u [v]_{<} \wedge II \downarrow_{\alpha} x)$

apply (*rel-auto*, *auto simp add: comp-def*)
using *vwb-lens.put-eq* **by** *fastforce*

18.8 Non-deterministic Assignment Laws

lemma *nd-assign-comp*:
 $x \bowtie y \implies x := * ;; y := * = x, y := *$
apply (*rel-auto*) **using** *lens-indep-comm* **by** *fastforce*+

lemma *nd-assign-assign*:
 $\llbracket \text{vwb-lens } x; x \# e \rrbracket \implies x := * ;; x := e = x := e$
by (*rel-auto*)

18.9 Converse Laws

lemma *convr-invol* [*simp*]: $p^{- -} = p$
by *pred-auto*

lemma *lit-convr* [*simp*]: $\llbracket v \rrbracket^{-} = \llbracket v \rrbracket$
by *pred-auto*

lemma *uivar-convr* [*simp*]:
fixes $x :: ('a \implies 'a)$
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*]: $(p ;; q)^{-} = (q^{-} ;; p^{-})$
by (*rel-auto*)

lemma *pre-convr* [*simp*]: $[p]_{<}^{-} = [p]_{>}$
by (*rel-auto*)

lemma *post-convr* [*simp*]: $[p]_{>}^{-} = [p]_{<}$

by (rel-auto)

18.10 Assertion and Assumption Laws

declare *sublens-def* [*lens-defs del*]

lemma *assume-false*: $[false]^\top = false$
by (rel-auto)

lemma *assume-true*: $[true]^\top = II$
by (rel-auto)

lemma *assume-seq*: $[b]^\top ;; [c]^\top = [(b \wedge c)]^\top$
by (rel-auto)

lemma *assert-false*: $\{false\}_\perp = true$
by (rel-auto)

lemma *assert-true*: $\{true\}_\perp = II$
by (rel-auto)

lemma *assert-seq*: $\{b\}_\perp ;; \{c\}_\perp = \{(b \wedge c)\}_\perp$
by (rel-auto)

18.11 Frame and Antiframe Laws

named-theorems *frame*

lemma *frame-all* [*frame*]: $\Sigma:[P] = P$
by (rel-auto)

lemma *frame-none* [*frame*]:
 $\emptyset:[P] = (P \wedge II)$
by (rel-auto)

lemma *frame-commute*:
assumes $\$y \# P \$y' \# P \$x \# Q \$x' \# Q x \bowtie y$
shows $x:[P] ;; y:[Q] = y:[Q] ;; x:[P]$
apply (insert *assms*)
apply (rel-auto)
apply (rename-tac *s s' 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* *s₀* (*get_x* *s'*))
 apply (metis *lens-indep-get lens-indep-sym*)
 apply (metis *lens-indep.lens-put-comm*)
done

lemma *frame-contract-RID*:
assumes *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]:
   $\llbracket \text{vwb-lens } a; x \subseteq_L a \rrbracket \implies a:[x := v] = x := v$ 
  by (rel-auto, simp-all add: lens-get-put-quasi-commute lens-put-of-quotient)

lemma frame-conj-true [frame]:
   $\llbracket \{ \$x, \$x' \} \Vdash P; \text{vwb-lens } x \rrbracket \implies (P \wedge x:[\text{true}]) = x:[P]$ 
  by (rel-auto)

lemma frame-is-assign [frame]:
  vwb-lens x  $\implies$  x:[ $\$x' =_u [v]_{<}$ ] = x := v
  by (rel-auto)

lemma frame-seq [frame]:
   $\llbracket \text{vwb-lens } x; \{ \$x, \$x' \} \Vdash P; \{ \$x, \$x' \} \Vdash Q \rrbracket \implies x:[P ;; Q] = x:[P] ;; x:[Q]$ 
  apply (rel-auto)
  apply (metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens-def weak-lens.put-get)
  apply (metis mwb-lens.put-put vwb-lens-mwb)
  done

lemma frame-to-antiframe [frame]:
   $\llbracket x \bowtie y; x +_L y = 1_L \rrbracket \implies x:[P] = y:[P]$ 
  by (rel-auto, metis lens-indep-def, metis lens-indep-def surj-pair)

lemma rel-frex-miracle [frame]:
  a:[false]+ = false
  by (rel-auto)

lemma rel-frex-skip [frame]:
  vwb-lens a  $\implies$  a:[II]+ = II
  by (rel-auto)

lemma rel-frex-seq [frame]:
  vwb-lens a  $\implies$  a:[P ;; Q]+ = (a:[P]+ ;; a:[Q]+)
  apply (rel-auto)
  apply (rename-tac s s' s0)
  apply (rule-tac x=puta s s0 in exI)
  apply (auto)
  apply (metis mwb-lens.put-put vwb-lens-mwb)
  done

```

lemma *rel-frex-assigms* [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*]

18.12 While Loop Laws

theorem *while-unfold*:
 $while\ b\ do\ P\ od = ((P ;; while\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$
proof –
have $m:mono\ (\lambda X.\ (P ;; X) \triangleleft b \triangleright_r II)$
by (*auto intro: monoI segr-mono cond-mono*)
have $(while\ b\ do\ P\ od) = (\nu\ X \cdot (P ;; X) \triangleleft b \triangleright_r II)$
by (*simp add: while-top-def*)
also have $\dots = ((P ;; (\nu\ X \cdot (P ;; X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
by (*subst lfp-unfold, simp-all add: m*)
also have $\dots = ((P ;; while\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$
by (*simp add: while-top-def*)
finally show *?thesis* .
qed

theorem *while-false*: $\text{while false do } P \text{ od} = II$
 by (subst while-unfold, rel-auto)

theorem *while-true*: $\text{while true do } P \text{ od} = \text{false}$
 apply (simp add: while-top-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 $\dots = ((P ;; (\mu X \cdot (P ;; X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
 by (subst gfp-unfold, simp-all add: m)
 also have $\dots = ((P ;; \text{while}_\perp b \text{ do } P \text{ od}) \triangleleft b \triangleright_r II)$
 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

18.13 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* :: ' $\alpha \text{ hrel} \Rightarrow \text{nat} \Rightarrow ' \alpha \text{ hrel}$ (infixr ^ 80) **where**
 $\text{upower } P \ n \equiv \text{upred-semiring.power } P \ n$

translations

$P \wedge i \leq \text{CONST power.power } II \text{ op} ;; P \ i$
 $P \wedge i \leq (\text{CONST power.power } II \text{ op} ;; P) \ i$

Set up transfer tactic for powers

```

lemma upower-rep-eq:
   $\llbracket P \wedge i \rrbracket_e = (\lambda b. b \in (\{p. \llbracket P \rrbracket_e p\} \wedge i))$ 
proof (induct i arbitrary: P)
  case 0
  then show ?case
    by (auto, rel-auto)
next
  case (Suc i)
  show ?case
    by (simp add: Suc seqr.rep-eq relpow-commute upred-semiring.power-Suc)
qed

```

```

lemma upower-rep-eq-alt:
   $\llbracket power.power \langle id \rangle_a (;;) P i \rrbracket_e = (\lambda b. b \in (\{p. \llbracket P \rrbracket_e p\} \wedge i))$ 
  by (metis skip-r-def upower-rep-eq)

```

update-uexpr-rep-eq-thms

```

lemma Sup-power-expand:
  fixes P :: nat  $\Rightarrow$  'a::complete-lattice
  shows  $P(0) \sqcap (\bigsqcap i. P(i+1)) = (\bigsqcap i. P(i))$ 
proof -
  have UNIV = insert (0::nat) {1..}
    by auto
  moreover have  $(\bigsqcap i. P(i)) = \bigsqcap (P \restriction UNIV)$ 
    by (blast)
  moreover have  $\bigsqcap (P \restriction insert\ 0\ \{1..\}) = P(0) \sqcap SUPREMUM\ \{1..\}\ P$ 
    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  $(\bigsqcap i \in \{0..Suc\ n\}. P \wedge i) = (\bigsqcap i \in insert\ (Suc\ n)\ \{0..n\}. P \wedge i)$ 
    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
      seqr-assoc seqr-mono)

```

qed

```

lemma upower-inductr:
  assumes  $Q \sqsubseteq R \sqcap (Q ;; P)$ 
  shows  $Q \sqsubseteq R ;; (P \wedge n)$ 
using assms proof (induct n)
  case 0
  then show ?case by auto
next
  case (Suc n)
  have  $R ;; P \wedge \text{Suc } n = (R ;; P \wedge n) ;; P$ 
  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  $(\bigsqcap_{i \in \{m..n\}}. P(i)) = P(m) \sqcap (\bigsqcap_{i \in \{\text{Suc } m..n\}}. P(i))$ 
  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+)

```

18.13.1 Kleene Star

```

definition ustar :: ' $\alpha$  hrel  $\Rightarrow$  ' $\alpha$  hrel  $(-^* [999] 999)$  where
 $P^* = (\bigsqcap_{i \in \{0..\}} \cdot P^i)$ 

```

```

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

18.14 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:  $P^+ = (\bigsqcap i \cdot P \wedge (\text{Suc } i))$ 
  by (simp add: uplus-def ustar-def seq-UNF-distl' UNF-atLeast-Suc upred-semiring.power-Suc)

```

18.15 Omega

```

definition uomega :: ' $\alpha$  hrel  $\Rightarrow$  ' $\alpha$  hrel  $(-^\omega [999] 999)$  where
 $P^\omega = (\mu X \cdot P ;; X)$ 

```


18.16 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*)

18.17 Kleene Algebra Laws

lemma *ustar-alt-def*: $P^* = (\bigcap i. P \cdot P^* \cdot i)$
 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 $P^* ;; R = (\bigcap i. P \cdot P^* \cdot i ;; R)$
 by (*simp add: ustar-def UINF-as-Sup-collect' seq-SUP-distr*)
 also have $Q \sqsubseteq \dots$
 by (*simp add: SUP-least assms upower-inductl*)
 finally show *?thesis* .
qed

theorem *ustar-inductr*:
 assumes $Q \sqsubseteq R$ $Q \sqsubseteq Q ;; P$
 shows $Q \sqsubseteq R ;; P^*$

proof –
 have $R ;; P^* = (\bigcap i. R ;; P \cdot P^* \cdot i)$
 by (*simp add: ustar-def UINF-as-Sup-collect' seq-SUP-distl*)
 also have $Q \sqsubseteq \dots$
 by (*simp add: SUP-least assms upower-inductr*)
 finally show *?thesis* .
qed

lemma *ustar-refines-nu*: $(\nu X. P \cdot (P ;; X) \sqcap II) \sqsubseteq P^*$
 by (*metis (no-types, lifting) lfp-greatest semilattice-sup-class.le-sup-iff semilattice-sup-class.sup-idem upred-semiring.mult-2-right*)

upred-semiring.one-add-one ustar-inductl)

lemma *ustar-as-nu*: $P^* = (\nu X \cdot (P ;; X) \sqcap II)$
proof (*rule antisym*)
 show $(\nu X \cdot (P ;; X) \sqcap II) \sqsubseteq P^*$
 by (*simp add: ustar-refines-nu*)
 show $P^* \sqsubseteq (\nu X \cdot (P ;; X) \sqcap II)$
 by (*metis lfp-lowerbound upred-semiring.add-commute ustar-sub-unfoldl*)
qed

lemma *ustar-unfoldl*: $P^* = II \sqcap (P ;; P^*)$
apply (*simp add: ustar-as-nu*)
apply (*subst lfp-unfold*)
apply (*rule monoI*)
apply (*rel-auto*)
done

While loop can be expressed using Kleene star

lemma *while-star-form*:
 $\text{while } b \text{ do } P \text{ od} = (P \triangleleft b \triangleright_r II)^* ;; [(\neg b)]^\top$
proof –
 have 1: *Continuous* $(\lambda X. P ;; X \triangleleft b \triangleright_r II)$
 by (*rel-auto*)
 have *while* $b \text{ do } P \text{ od} = (\bigsqcap i. ((\lambda X. P ;; X \triangleleft b \triangleright_r II) \hat{\ } i) \text{ false})$
 by (*simp add: 1 false-upred-def sup-continuous-Continuous sup-continuous-lfp while-top-def*)
 also have ... = $((\lambda X. P ;; X \triangleleft b \triangleright_r II) \hat{\ } 0) \text{ false} \sqcap (\bigsqcap i. ((\lambda X. P ;; X \triangleleft b \triangleright_r II) \hat{\ } (i+1)) \text{ false})$
 by (*subst Sup-power-expand, simp*)
 also have ... = $(\bigsqcap i. ((\lambda X. P ;; X \triangleleft b \triangleright_r II) \hat{\ } (i+1)) \text{ false})$
 by (*simp*)
 also have ... = $(\bigsqcap i. (P \triangleleft b \triangleright_r II) \hat{\ } i ;; (\text{false} \triangleleft b \triangleright_r II))$
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 ... = $(\bigsqcap i \in \{0..\} \cdot (P \triangleleft b \triangleright_r II) \hat{\ } i ;; [(\neg b)]^\top)$
 by (*rel-auto*)
 also have ... = $(P \triangleleft b \triangleright_r II)^* ;; [(\neg b)]^\top$
 by (*metis seq-UNF-distr ustar-def*)
 finally show ?thesis .
qed

18.18 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*)

18.19 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 $S \sqsubseteq ([b]_< \wedge P)$ $S \sqsubseteq ([\neg b]_< \wedge Q)$
shows $S \sqsubseteq P \triangleleft b \triangleright_r Q$
by (*metis aext-not assms(1) assms(2) cond-def lift-rcond-def utp-pred-laws.le-sup-iff*)

lemma *seq-refine-pred*:

assumes $([b]_< \Rightarrow [s]_>) \sqsubseteq P$ **and** $([s]_< \Rightarrow [c]_>) \sqsubseteq Q$
shows $([b]_< \Rightarrow [c]_>) \sqsubseteq (P ;; Q)$
using *assms* **by** *rel-auto*

lemma *seq-refine-unrest*:

assumes $out\alpha \nmid b$ $in\alpha \nmid c$
assumes $(b \Rightarrow [s]_>) \sqsubseteq P$ **and** $([s]_< \Rightarrow c) \sqsubseteq Q$
shows $(b \Rightarrow c) \sqsubseteq (P ;; Q)$
using *assms* **by** *rel-blast*

18.20 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)]⊤)
by (rel-auto)

lemma Dom-disj:
  Dom(P ∨ Q) = (Dom(P) ∨ Dom(Q))
by (rel-auto)

lemma Dom-inf:
  Dom(P ⊓ Q) = (Dom(P) ∨ Dom(Q))
by (rel-auto)

lemma assume-Dom:
  [Dom(P)]⊤ ;; P = P
by (rel-auto)

end

```

19 UTP Theories

```

theory utp-theory
imports utp-rel-laws
begin

```

Here, we mechanise a representation of UTP theories using locales [4]. We also link them to the HOL-Algebra library [5], which allows us to import properties from complete lattices and Galois connections.

19.1 Complete lattice of predicates

definition *upred-lattice* :: (α upred) gorder (\mathcal{P}) **where**
upred-lattice = (λ carrier = UNIV, eq = (=), le = (\sqsubseteq))

\mathcal{P} is the complete lattice of alphabetised predicates. All other theories will be defined relative to it.

```

interpretation upred-lattice: complete-lattice  $\mathcal{P}$ 
proof (unfold-locales, simp-all add: upred-lattice-def)
  fix A ::  $\alpha$  upred set
  show  $\exists s$ . is-lub ( $\lambda$  carrier = UNIV, eq = (=), le = ( $\sqsubseteq$ )) 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  $\exists i$ . is-glb ( $\lambda$  carrier = UNIV, eq = (=), le = ( $\sqsubseteq$ )) i A
    apply (rule-tac x= $\sqcap$  A in exI)
    apply (rule greatest-LowerI)
    apply (auto intro: Sup-least simp add: Sup-upper Lower-def)
  done
qed

```

```

lemma upred-weak-complete-lattice [simp]: weak-complete-lattice  $\mathcal{P}$ 
  by (simp add: upred-lattice.weak.weak-complete-lattice-axioms)

```

lemma *upred-lattice-eq* [*simp*]:
 $(\cdot =_{\mathcal{P}}) = (=)$
by (*simp add: upred-lattice-def*)

lemma *upred-lattice-le* [*simp*]:
 $le_{\mathcal{P}} P Q = (P \sqsubseteq Q)$
by (*simp add: upred-lattice-def*)

lemma *upred-lattice-carrier* [*simp*]:
 $carrier \mathcal{P} = UNIV$
by (*simp add: upred-lattice-def*)

lemma *Healthy-fixed-points* [*simp*]: $fps \mathcal{P} H = \llbracket H \rrbracket_H$
by (*simp add: fps-def upred-lattice-def Healthy-def*)

lemma *upred-lattice-Idempotent* [*simp*]: $Idem_{\mathcal{P}} H = Idempotent H$
using *upred-lattice.weak-partial-order-axioms* **by** (*auto simp add: idempotent-def Idempotent-def*)

lemma *upred-lattice-Monotonic* [*simp*]: $Mono_{\mathcal{P}} H = Monotonic H$
using *upred-lattice.weak-partial-order-axioms* **by** (*auto simp add: isotone-def mono-def*)

19.2 UTP theories hierarchy

typedef (\mathcal{T} , α) *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* :: (α , β) *uthy* **where**
 $uthy = Abs-uthy ()$

lemma *uthy-eq* [*intro*]:
fixes $x y :: (\alpha, \beta) uthy$
shows $x = y$
by (*cases x, cases y, simp*)

syntax
 $-UTHY :: type \Rightarrow type \Rightarrow logic (UTHY'(-, -'))$

translations
 $UTHY(\mathcal{T}, \alpha) == CONST uthy :: (\mathcal{T}, \alpha) uthy$

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
 $utp-hcond :: (\mathcal{T}, \alpha) uthy \Rightarrow (\alpha \times \alpha) health (\mathcal{H}_1)$

definition *utp-order* :: ($'\alpha \times '\alpha$) *health* \Rightarrow $'\alpha$ *hrel* *gorder* **where**
utp-order $H = \langle \text{carrier} = \{P. P \text{ is } H\}, \text{eq} = (=), \text{le} = (\sqsubseteq) \rangle$

abbreviation *uthy-order* $T \equiv \text{utp-order } \mathcal{H}_T$

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*]:
 $\text{carrier } (\text{utp-order } H) = \llbracket H \rrbracket_H$
by (*simp add: utp-order-def*)

lemma *utp-order-eq* [*simp*]:
 $\text{eq } (\text{utp-order } T) = (=)$
by (*simp add: utp-order-def*)

lemma *utp-order-le* [*simp*]:
 $\text{le } (\text{utp-order } T) = (\sqsubseteq)$
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*:
 $\text{mono } f \Longrightarrow \text{Monotone } (\text{utp-order } T) f$
apply (*auto simp add: isotone-def*)
apply (*metis partial-order-def utp-partial-order*)
apply (*metis monoD*)
done

lemma *isotone-utp-orderI*: $\text{Monotonic } H \Longrightarrow \text{isotone } (\text{utp-order } X) (\text{utp-order } Y) H$
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*: $\text{utp-order } H = \text{fpl } \mathcal{P} H$
by (*auto simp add: utp-order-def upred-lattice-def fps-def Healthy-def*)

definition *uth-eq* :: ($'T_1, '\alpha$) *uthy* \Rightarrow ($'T_2, '\alpha$) *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*: $\llbracket T_1 \approx_T T_2; T_2 \approx_T T_3 \rrbracket \Longrightarrow T_1 \approx_T T_3$
by (*auto simp add: uth-eq-def*)

definition $uthy-plus :: ('T_1, 'α) uthy \Rightarrow ('T_2, 'α) uthy \Rightarrow ('T_1 \times 'T_2, 'α) uthy$ (**infixl** $+_T$ 65) **where**
 $uthy-plus\ T_1\ T_2 = uthy$

overloading

$prod-hcond == utp-hcond :: ('T_1 \times 'T_2, 'α) uthy \Rightarrow ('α \times 'α) health$

begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition $prod-hcond :: ('T_1 \times 'T_2, 'α) uthy \Rightarrow ('α \times 'α) upred \Rightarrow ('α \times 'α) upred$ **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 $\mathcal{T} :: ('T, 'α) uthy$ (**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 $T_1 = uthy\ T_2 = uthy$

by $blast+$

thus $?thesis$

using $assms$ **by** ($simp\ add: uth-eq-def\ prod-hcond-def$)

qed

lemma $uthy-plus-assoc: T_1 +_T (T_2 +_T T_3) \approx_T (T_1 +_T T_2) +_T T_3$

by ($simp\ add: uth-eq-def\ prod-hcond-def\ comp-def$)

lemma $uthy-plus-idem: utp-theory\ T \Longrightarrow T +_T T \approx_T T$

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** $\mathcal{T} :: ('T, 'α) uthy$ (**structure**)

The healthiness conditions of a UTP theory lattice form a complete lattice, and allows us to make use of complete lattice results from HOL-Algebra, such as the Knaster-Tarski theorem. We can also retrieve lattice operators as below.

abbreviation *utp-top* (\top_1)

where *utp-top* $\mathcal{T} \equiv \text{top } (\text{uthy-order } \mathcal{T})$

abbreviation *utp-bottom* (\perp_1)

where *utp-bottom* $\mathcal{T} \equiv \text{bottom } (\text{uthy-order } \mathcal{T})$

abbreviation *utp-join* (**infixl** \sqcup_1 65) **where**

utp-join $\mathcal{T} \equiv \text{join } (\text{uthy-order } \mathcal{T})$

abbreviation *utp-meet* (**infixl** \sqcap_1 70) **where**

utp-meet $\mathcal{T} \equiv \text{meet } (\text{uthy-order } \mathcal{T})$

abbreviation *utp-sup* (\bigsqcup_1 - [90] 90) **where**

utp-sup $\mathcal{T} \equiv \text{Lattice.sup } (\text{uthy-order } \mathcal{T})$

abbreviation *utp-inf* (\bigsqcap_1 - [90] 90) **where**

utp-inf $\mathcal{T} \equiv \text{Lattice.inf } (\text{uthy-order } \mathcal{T})$

abbreviation *utp-gfp* (ν_1) **where**

utp-gfp $\mathcal{T} \equiv \text{GREATEST-FP } (\text{uthy-order } \mathcal{T})$

abbreviation *utp-lfp* (μ_1) **where**

utp-lfp $\mathcal{T} \equiv \text{LEAST-FP } (\text{uthy-order } \mathcal{T})$

syntax

-tmu :: *logic* \Rightarrow *pttrn* \Rightarrow *logic* \Rightarrow *logic* (μ_1 - \cdot - [0, 10] 10)

-tnu :: *logic* \Rightarrow *pttrn* \Rightarrow *logic* \Rightarrow *logic* (ν_1 - \cdot - [0, 10] 10)

notation *gfp* (μ)

notation *lfp* (ν)

translations

$\mu_T X \cdot P == \text{CONST } \text{utp-lfp } T (\lambda X. P)$

$\nu_T X \cdot P == \text{CONST } \text{utp-gfp } T (\lambda X. P)$

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: *Mono_{uthy-order \mathcal{T}}* \mathcal{H}
 by (*simp add: HCond-Mono isotone-utp-orderI*)
 have 3: *Idem_{uthy-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 upred-lattice-inf utp-order-def*)

qed

end

locale *utp-theory-continuous* = *utp-theory* +

assumes *HCond-Cont [closure,intro]: Continuous \mathcal{H}*

sublocale *utp-theory-continuous* \subseteq *utp-theory-mono*

proof

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-Disjunctous 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* $F F \in \llbracket \mathcal{H} \rrbracket_H \rightarrow \llbracket \mathcal{H} \rrbracket_H$
shows $\mu F = (\mu X \cdot F(\mathcal{H}(X)))$
proof (*rule antisym*)
 have *ne*: $\{P. (P \text{ is } \mathcal{H}) \wedge F P \sqsubseteq P\} \neq \{\}$
proof –
 have $F \top \sqsubseteq \top$
 using *assms(2) utp-top weak.top-closed* **by** *force*
 thus *?thesis*
 by (*auto, rule-tac x=\top in exI, auto simp add: top-healthy*)
qed
 show $\mu F \sqsubseteq (\mu X \cdot F(\mathcal{H}(X)))$
proof –
 have $\sqcap \{P. (P \text{ is } \mathcal{H}) \wedge F(P) \sqsubseteq P\} \sqsubseteq \sqcap \{P. F(\mathcal{H}(P)) \sqsubseteq P\}$
proof –
 have $1: \bigwedge P. F(\mathcal{H}(P)) = \mathcal{H}(F(\mathcal{H}(P)))$

```

    by (metis HCond-Idem Healthy-def assms(2) funcset-mem mem-Collect-eq)
  show ?thesis
  proof (rule Sup-least, auto)
    fix P
    assume a:  $F (\mathcal{H} P) \sqsubseteq P$ 
    hence F:  $(F (\mathcal{H} P)) \sqsubseteq (\mathcal{H} P)$ 
    by (metis 1 HCond-Mono mono-def)
    show  $\sqcap \{P. (P \text{ is } \mathcal{H}) \wedge F P \sqsubseteq P\} \sqsubseteq P$ 
    proof (rule Sup-upper2[of F ( $\mathcal{H} P$ )])
      show  $F (\mathcal{H} P) \in \{P. (P \text{ is } \mathcal{H}) \wedge F P \sqsubseteq P\}$ 
      proof (auto)
        show  $F (\mathcal{H} P) \text{ is } \mathcal{H}$ 
        by (metis 1 Healthy-def)
        show  $F (F (\mathcal{H} P)) \sqsubseteq F (\mathcal{H} P)$ 
        using F mono-def assms(1) by blast
      qed
      show  $F (\mathcal{H} P) \sqsubseteq P$ 
      by (simp add: a)
    qed
  qed
  qed
  qed

  with ne show ?thesis
  by (simp add: LEAST-FP-def gfp-def, subst healthy-inf-cont, auto simp add: lfp-def)
  qed
  from ne show  $(\mu X \cdot F (\mathcal{H} X)) \sqsubseteq \mu F$ 
  apply (simp add: LEAST-FP-def gfp-def, subst healthy-inf-cont, auto simp add: lfp-def)
  apply (rule Sup-least)
  apply (auto simp add: Healthy-def Sup-upper)
  done
  qed

lemma UINF-ind-Healthy [closure]:
  assumes  $\bigwedge i. P(i) \text{ is } \mathcal{H}$ 
  shows  $(\sqcap i \cdot P(i)) \text{ is } \mathcal{H}$ 
  by (simp add: closure assms)

end

In another direction, we can also characterise UTP theories that are relational. Minimally this
requires that the healthiness condition is closed under sequential composition.

locale utp-theory-rel =
  utp-theory +
  assumes Healthy-Sequence [closure]:  $\llbracket P \text{ is } \mathcal{H}; Q \text{ is } \mathcal{H} \rrbracket \implies (P ;; Q) \text{ is } \mathcal{H}$ 
begin

lemma upower-Suc-Healthy [closure]:
  assumes  $P \text{ is } \mathcal{H}$ 
  shows  $P \wedge \text{Suc } n \text{ is } \mathcal{H}$ 
  by (induct n, simp-all add: closure assms upred-semiring.power-Suc)

end

locale utp-theory-cont-rel = utp-theory-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  $P ;; (\bigcap A) = \bigcap \{P ;; Q \mid Q. Q \in A\}$ 
proof –
  have  $\{P ;; Q \mid Q. Q \in A\} \subseteq \llbracket \mathcal{H} \rrbracket_H$ 
    using Healthy-Sequence assms(1) assms(2) by (auto)
  thus ?thesis
    by (simp add: healthy-inf-cont seq-Sup-distl setcompr-eq-image assms)
qed

```

```

lemma seq-cont-Sup-distr:
  assumes  $Q$  is  $\mathcal{H}$   $A \subseteq \llbracket \mathcal{H} \rrbracket_H$   $A \neq \{\}$ 
  shows  $(\bigcap A) ;; Q = \bigcap \{P ;; Q \mid P. P \in A\}$ 
proof –
  have  $\{P ;; Q \mid P. P \in A\} \subseteq \llbracket \mathcal{H} \rrbracket_H$ 
    using Healthy-Sequence assms(1) assms(2) by (auto)
  thus ?thesis
    by (simp add: healthy-inf-cont seq-Sup-distr setcompr-eq-image assms)
qed

```

```

lemma uplus-healthy [closure]:
  assumes  $P$  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 ::  $(\mathcal{T}, 'a)$  uthy  $\Rightarrow$   $'a$  hrel (II1)

```

We can characterise the theory Kleene star by lifting the relational one.

```

definition utp-star ( $-\star_1$  [999] 999) where
  [upred-defs]: utp-star  $\mathcal{T}$   $P = (P^* ;; \text{II}_{\mathcal{T}})$ 

```

We can then characterise tests as refinements of units.

```

definition utest ::  $(\mathcal{T}, 'a)$  uthy  $\Rightarrow$   $'a$  hrel  $\Rightarrow$  bool where
  [upred-defs]: utest  $\mathcal{T}$   $b = (\text{II}_{\mathcal{T}} \sqsubseteq b)$ 

```

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]: II is  $\mathcal{H}$ 
  and Left-Unit:  $P$  is  $\mathcal{H} \implies (\text{II} ;; P) = P$ 

```

```

locale utp-theory-right-unital =
  utp-theory-rel +
  assumes Healthy-Right-Unit [closure]: II is  $\mathcal{H}$ 
  and Right-Unit:  $P$  is  $\mathcal{H} \implies (P ;; \text{II}) = P$ 

```

```

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  $\mathcal{H}$   $\text{utest } \mathcal{T} b$ 
  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.

```

typedecl 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  $\Rightarrow$  ('α × 'α) 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  $\Rightarrow$  ('α × 'α) upred  $\Rightarrow$  ('α × 'α) upred where
  [upred-defs]: rel-hcond T = id

```

The unit of the theory is simply the relational unit.

```

definition rel-unit :: (REL, 'α) uthy  $\Rightarrow$  '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 a Galois connections and retractions, using the following notation.

definition *mk-conn* $(- \Leftarrow \langle -, - \rangle \Rightarrow - [90, 0, 0, 91] \ 91)$ **where**
 $H1 \Leftarrow \langle \mathcal{H}_1, \mathcal{H}_2 \rangle \Rightarrow H2 \equiv () \text{ orderA} = \text{utp-order } H1, \text{ orderB} = \text{utp-order } H2, \text{ lower} = \mathcal{H}_2, \text{ upper} = \mathcal{H}_1 \ ()$

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*: $\text{mwb-lens } x \Longrightarrow \text{Idempotent } (ex \ x)$
by (simp add: Idempotent-def exists-twice)

lemma *Monotonic-ex*: $\text{mwb-lens } x \Longrightarrow \text{Monotonic } (ex \ x)$
by (simp add: mono-def ex-mono)

lemma *ex-closed-unrest*:
 $\text{vwb-lens } x \Longrightarrow \llbracket ex \ x \rrbracket_H = \{P. \ x \# P\}$
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 *Idempotent* H $ex \ x \circ H = H \circ ex \ x$
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* $\Leftarrow \langle \text{ex } x, \text{id} \rangle \Rightarrow \text{id}$)
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 ($\{\cdot\}$ / - / $\{\cdot\}_u$) **where**
 $\{p\} Q \{r\}_u = (([p]_< \Rightarrow [r]_>) \sqsubseteq Q)$

declare *hoare-r-def* [*upred-defs*]

named-theorems *hoare* and *hoare-safe*

method *hoare-split* **uses** *hr* =
 (*(simp add: assigns-comp)?*, — Combine Assignments where possible
 (*auto*
 intro: hoare intro!: hoare-safe hr
 simp add: conj-comm conj-assoc usubst unrest))[1] — Apply Hoare logic laws

method *hoare-auto* **uses** *hr* = (*hoare-split hr: hr; (rel-simp)?, auto?*)

20.2 Basic Laws

lemma *hoare-meaning*:
 $\{P\} S \{Q\}_u = (\forall s s'. \llbracket P \rrbracket_e s \wedge \llbracket S \rrbracket_e (s, s') \longrightarrow \llbracket Q \rrbracket_e s')$
by (*rel-auto*)

lemma *hoare-assume*: $\{P\} S \{Q\}_u \Longrightarrow ?[P] ;; S = ?[P] ;; S ;; ?[Q]$
by (*rel-auto*)

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 $\{p\} \text{ assign-}r \ x \ e \ \llbracket \exists v \cdot p[\llbracket v \rrbracket/x] \wedge \&x =_u e[\llbracket v \rrbracket/x] \rrbracket_u$
using *assms*
by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *assigns-init-hoare* [*hoare-safe*]:

$\llbracket \text{vwb-lens } x; x \# p; x \# v; \&x =_u v \wedge p\{q\}_u \rrbracket \Longrightarrow \{p\}x := v ;; S\{q\}_u$
by (*rel-auto*)

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*]: $\llbracket \{b \wedge p\} S \{q\}_u ; \{\neg b \wedge p\} T \{q\}_u \rrbracket \implies \{p\} S \triangleleft b \triangleright_r T \{q\}_u$
 by *rel-auto*

lemma *cond-hoare-r-wp*:
 assumes $\{p'\} S \{q\}_u$ and $\{p'\} T \{q\}_u$
 shows $\{(b \wedge p') \vee (\neg b \wedge p')\} S \triangleleft b \triangleright_r T \{q\}_u$
 using *assms* by *pred-simp*

lemma *cond-hoare-r-sp*:
 assumes $\langle \{b \wedge p\} S \{q\}_u \rangle$ and $\langle \{\neg b \wedge p\} T \{s\}_u \rangle$
 shows $\langle \{p\} S \triangleleft b \triangleright_r T \{q \vee s\}_u \rangle$
 using *assms* by *pred-simp*

20.6 Recursion Laws

lemma *nu-hoare-r-partial*:
 assumes *induct-step*:
 $\bigwedge st P. \{p\} P \{q\}_u \implies \{p\} F P \{q\}_u$
 shows $\{p\} \nu F \{q\}_u$
 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*:
 $\bigwedge st P. \{p \wedge (e, \ll st \gg)_u \in_u \ll R \gg\} P \{q\}_u \implies \{p \wedge e =_u \ll st \gg\} F P \{q\}_u$
 shows $\{p\} \mu F \{q\}_u$
 unfolding *hoare-r-def*
proof (*rule mu-rec-total-utp-rule*[*OF* *WF* *M* , *of* - *e*], *goal-cases*)
 case (*1 st*)
 then show ?*case*
 using *induct-step*[*unfolded hoare-r-def*, *of* ($\lceil p \rceil < \wedge (\lceil e \rceil <, \ll st \gg)_u \in_u \ll R \gg \Rightarrow \lceil q \rceil > st$)]
 by (*simp add: alpha*)
qed

lemma *mu-hoare-r'*:
 assumes *WF*: *wf* *R*
 assumes *M*:*mono* *F*
 assumes *induct-step*:
 $\bigwedge st P. \{p \wedge (e, \ll st \gg)_u \in_u \ll R \gg\} P \{q\}_u \implies \{p \wedge e =_u \ll st \gg\} F P \{q\}_u$
 assumes *I0*: '*p*' \Rightarrow '*p*'
 shows $\{p'\} \mu F \{q\}_u$
 by (*meson I0 M WF induct-step mu-hoare-r pre-str-hoare-r*)

20.7 Iteration Rules

lemma *iter-hoare-r*: $\{P\} S \{P\}_u \implies \{P\} S^* \{P\}_u$
 by (*rel-simp'*, *metis* (*mono-tags*, *lifting*) *mem-Collect-eq rtrancl-induct*)

lemma *while-hoare-r* [*hoare-safe*]:
 assumes $\{p \wedge b\} S \{p\}_u$
 shows $\{p\} \text{while } b \text{ do } S \text{ od } \{\neg b \wedge p\}_u$
 using *assms*
 by (*simp add: while-top-def hoare-r-def*, *rule-tac lfp-lowerbound*) (*rel-auto*)

lemma *while-invr-hoare-r [hoare-safe]*:
assumes $\llbracket p \wedge b \rrbracket S \llbracket p \rrbracket_u \text{ ‘pre} \Rightarrow p \text{ ‘} (\neg b \wedge p) \Rightarrow \text{post} \text{ ‘}$
shows $\llbracket \text{pre} \rrbracket \text{while } b \text{ invr } p \text{ do } S \text{ od} \llbracket \text{post} \rrbracket_u$
by (*metis assms hoare-r-conseq while-hoare-r while-inv-def*)

lemma *while-r-minimal-partial*:
assumes *seq-step*: $\text{‘}p \Rightarrow \text{invar}\text{ ‘}$
assumes *induct-step*: $\llbracket \text{invar} \wedge b \rrbracket C \llbracket \text{invar} \rrbracket_u$
shows $\llbracket p \rrbracket \text{while } b \text{ do } C \text{ od} \llbracket \neg b \wedge \text{invar} \rrbracket_u$
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 $\bigwedge z::\text{nat}. \llbracket p \wedge b \wedge v =_u \ll z \gg \rrbracket S \llbracket p \wedge v <_u \ll z \gg \rrbracket_u$
shows $\llbracket p \rrbracket \text{while}_\perp b \text{ do } S \text{ od} \llbracket \neg b \wedge p \rrbracket_u$
proof –
have $(\lceil p \rceil_< \Rightarrow \lceil \neg b \wedge p \rceil_>) \sqsubseteq (\mu X \cdot S ;; X \triangleleft b \triangleright_r II)$
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 $(\lceil p \rceil_< \wedge (\mu X \cdot S ;; X \triangleleft b \triangleright_r II)) = (\lceil p \rceil_< \wedge (\nu X \cdot S ;; X \triangleleft b \triangleright_r II))$
proof (*rule constr-fp-uniq[where E=?E]*)

show $(\bigcap n. ?E(n)) = \lceil p \rceil_<$
by (*rel-auto*)

show *mono* $(\lambda X. S ;; X \triangleleft b \triangleright_r II)$
by (*simp add: cond-mono monoI segr-mono*)

show *constr* $(\lambda X. S ;; X \triangleleft b \triangleright_r II) ?E$
proof (*rule constrI*)

show *chain* $?E$
proof (*rule chainI*)
show $\lceil p \wedge v <_u \ll 0 \gg \rceil_< = \text{false}$
by (*rel-auto*)
show $\bigwedge i. \lceil p \wedge v <_u \ll \text{Suc } i \gg \rceil_< \sqsubseteq \lceil p \wedge v <_u \ll i \gg \rceil_<$
by (*rel-auto*)
qed

from *assms*
show $\bigwedge X n. (S ;; X \triangleleft b \triangleright_r II \wedge \lceil p \wedge v <_u \ll n + 1 \gg \rceil_<) =$
 $(S ;; (X \wedge \lceil p \wedge v <_u \ll n \gg \rceil_<) \triangleleft b \triangleright_r II \wedge \lceil p \wedge v <_u \ll n + 1 \gg \rceil_<)$
apply (*rel-auto*)
using *less-antisym less-trans* **apply** *blast*
done

```

    qed
  qed
qed

thus ?thesis
  by (simp add: hoare-r-def while-bot-def)
qed

lemma while-vrt-hoare-r [hoare-safe]:
  assumes  $\bigwedge z::nat. \llbracket p \wedge b \wedge v =_u \llbracket z \rrbracket \rrbracket S \llbracket p \wedge v <_u \llbracket z \rrbracket \rrbracket_u$  'pre  $\Rightarrow$  p' ' $(\neg b \wedge p) \Rightarrow$  post'
  shows  $\llbracket pre \rrbracket while\ b\ invr\ p\ vrt\ v\ do\ S\ od\ \llbracket post \rrbracket_u$ 
  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: '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: ' $(\neg b \wedge p) \Rightarrow$  post'
  shows  $\llbracket pre \rrbracket while_{\perp}\ b\ invr\ p\ do\ Q\ od\ \llbracket post \rrbracket_u$ 
unfolding hoare-r-def while-inv-bot-def while-bot-def
proof (rule pre-weak-rel[of - [p]_< ])
  from I0 show '[pre]_<  $\Rightarrow$  [p]_<'
    by rel-auto
  show ([p]_<  $\Rightarrow$  [post]_>)  $\sqsubseteq$  ( $\mu X. X \cdot Q ;; X \triangleleft b \triangleright_r II$ )
  proof (rule mu-rec-total-utp-rule[where e=e, OF WF])
    show Monotonic ( $\lambda X. X \cdot Q ;; X \triangleleft b \triangleright_r II$ )
      by (simp add: closure)
    have induct-step':  $\bigwedge st. ([b \wedge p \wedge e =_u \llbracket st \rrbracket]_< \Rightarrow ([p \wedge (e, \llbracket st \rrbracket)_u \in_u \llbracket R \rrbracket]_>)) \sqsubseteq Q$ 
      using induct-step by rel-auto
    with PHI
    show  $\bigwedge st. ([p]_< \wedge [e]_< =_u \llbracket st \rrbracket \Rightarrow [post]_>) \sqsubseteq Q ;; ([p]_< \wedge ([e]_<, \llbracket st \rrbracket)_u \in_u \llbracket R \rrbracket \Rightarrow [post]_>)$ 
      < b  $\triangleright_r II$ 
      by (rel-auto)
  qed
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  $\sharp$  r a  $\sharp$  q  $\{r \wedge p\} S \{q\}_u$ 
  shows  $\{r \wedge p\} a : [S] \{r \wedge q\}_u$ 
  using assms
  by (simp add: frame-strong-hoare-r utp-pred-laws.inf commute)

```

```

lemma antiframe-hoare-r:
  assumes vwb-lens a a  $\sharp$  r a  $\sharp$  q  $\{p\} P \{q\}_u$ 
  shows  $\{p \wedge r\} a : [P] \{q \wedge r\}_u$ 
  using assms by (rel-auto, metis)

```

```

lemma antiframe-strong-hoare-r:
  assumes vwb-lens a a  $\sharp$  r a  $\sharp$  q  $\{p \wedge r\} P \{q\}_u$ 
  shows  $\{p \wedge r\} a : [P] \{q \wedge r\}_u$ 
  using assms by (rel-auto, metis)

```

end

21 Weakest (Liberal) Precondition Calculus

```

theory utp-wp
imports utp-hoare
begin

```

A very quick implementation of wlp – more laws still needed!

named-theorems *wp*

method *wp-tac* = (*simp add: wp*)

```

consts
  uwp :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'c

```

```

syntax
  -uwp :: logic  $\Rightarrow$  uexp  $\Rightarrow$  logic (infix wp 60)

```

```

translations
  -uwp P b == CONST uwp P b

```

definition *wp-upred* :: (' 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]:
   $\langle \sigma \rangle_a \text{ wp } r = \sigma \uparrow r$ 
  by rel-auto

```

theorem *wp-skip-r* [*wp*]:

II wp r = r
by *rel-auto*

theorem *wp-abort* [*wp*]:
 $r \neq \text{true} \implies \text{true wp } r = \text{false}$
by *rel-auto*

theorem *wp-conj* [*wp*]:
 $P \text{ wp } (q \wedge r) = (P \text{ wp } q \wedge P \text{ wp } r)$
by *rel-auto*

theorem *wp-seq-r* [*wp*]: $(P ;; Q) \text{ wp } r = P \text{ wp } (Q \text{ wp } r)$
by *rel-auto*

theorem *wp-choice* [*wp*]: $(P \sqcap Q) \text{ wp } R = (P \text{ wp } R \wedge Q \text{ wp } R)$
by (*rel-auto*)

theorem *wp-cond* [*wp*]: $(P \triangleleft b \triangleright_r Q) \text{ wp } r = ((b \Rightarrow P \text{ wp } r) \wedge ((\neg b) \Rightarrow Q \text{ wp } r))$
by *rel-auto*

lemma *wp-USUP-pre* [*wp*]: $P \text{ wp } (\bigsqcup_{i \in \{0..n\}} Q(i)) = (\bigsqcup_{i \in \{0..n\}} P \text{ wp } Q(i))$
by (*rel-auto*)

theorem *wp-hoare-link*:
 $\llbracket p \rrbracket Q \llbracket r \rrbracket_u \longleftrightarrow (Q \text{ wp } r \sqsubseteq p)$
by *rel-auto*

If two programs have the same weakest precondition for any postcondition then the programs are the same.

theorem *wp-eq-intro*: $\llbracket \bigwedge r. P \text{ wp } r = Q \text{ wp } r \rrbracket \implies P = Q$
by (*rel-auto robust, fastforce+*)
end

22 Dynamic Logic

theory *utp-dynlog*
imports *utp-sequent utp-wp*
begin

22.1 Definitions

named-theorems *dynlog-simp* and *dynlog-intro*

definition *dBox* :: $'s \text{ hrel} \Rightarrow 's \text{ upred} \Rightarrow 's \text{ upred} \rightarrow ([-] - [0,999] \ 999)$
where [*upred-defs*]: $dBox \ A \ \Phi = A \text{ wp } \Phi$

definition *dDia* :: $'s \text{ hrel} \Rightarrow 's \text{ upred} \Rightarrow 's \text{ upred} \rightarrow (<->- [0,999] \ 999)$
where [*upred-defs*]: $dDia \ A \ \Phi = (\neg [A] (\neg \Phi))$

22.2 Box Laws

lemma *dBox-false* [*dynlog-simp*]: $[false] \Phi = \text{true}$
by (*rel-auto*)

lemma *dBox-skip* [*dynlog-simp*]: $[I] \Phi = \Phi$

by (rel-auto)

lemma *dBox-assigns* [dynlog-simp]: $[\langle \sigma \rangle_a] \Phi = (\sigma \dagger \Phi)$
 by (simp add: dBox-def wp-assigns-r)

lemma *dBox-choice* [dynlog-simp]: $[P \sqcap Q] \Phi = ([P] \Phi \wedge [Q] \Phi)$
 by (rel-auto)

lemma *dBox-seq*: $[P ;; Q] \Phi = [P][Q] \Phi$
 by (simp add: dBox-def wp-seq-r)

lemma *dBox-star-unfold*: $[P^*] \Phi = (\Phi \wedge [P][P^*] \Phi)$
 by (metis dBox-choice dBox-seq dBox-skip ustar-unfoldl)

lemma *dBox-star-induct*: $'(\Phi \wedge [P^*](\Phi \Rightarrow [P] \Phi)) \Rightarrow [P^*] \Phi'$
 by (rel-simp, metis (mono-tags, lifting) mem-Collect-eq rtrancl-induct)

lemma *dBox-test*: $[? [p]] \Phi = (p \Rightarrow \Phi)$
 by (rel-auto)

22.3 Diamond Laws

lemma *dDia-false* [dynlog-simp]: $\langle \text{false} \rangle \Phi = \text{false}$
 by (simp add: dBox-false dDia-def)

lemma *dDia-skip* [dynlog-simp]: $\langle I \rangle \Phi = \Phi$
 by (simp add: dBox-skip dDia-def)

lemma *dDia-assigns* [dynlog-simp]: $\langle \langle \sigma \rangle_a \rangle \Phi = (\sigma \dagger \Phi)$
 by (simp add: dBox-assigns dDia-def subst-not)

lemma *dDia-choice*: $\langle P \sqcap Q \rangle \Phi = (\langle P \rangle \Phi \vee \langle Q \rangle \Phi)$
 by (simp add: dBox-def dDia-def wp-choice)

lemma *dDia-seq*: $\langle P ;; Q \rangle \Phi = \langle P \rangle \langle Q \rangle \Phi$
 by (simp add: dBox-def dDia-def wp-seq-r)

lemma *dDia-test*: $\langle ? [p] \rangle \Phi = (p \wedge \Phi)$
 by (rel-auto)

22.4 Sequent Laws

lemma *sBoxSeq* [dynlog-simp]: $\Gamma \Vdash [P ;; Q] \Phi \equiv \Gamma \Vdash [P][Q] \Phi$
 by (simp add: dBox-def wp-seq-r)

lemma *sBoxTest* [dynlog-intro]: $\Gamma \Vdash (b \Rightarrow \Psi) \Longrightarrow \Gamma \Vdash [? [b]] \Psi$
 by (rel-auto)

lemma *sBoxAssignFwd* [dynlog-simp]: $\llbracket \text{vwb-lens } x; x \# v; x \# \Gamma \rrbracket \Longrightarrow (\Gamma \Vdash [x := v] \Phi) = ((\&x =_u v \wedge \Gamma) \Vdash \Phi)$
 by (rel-auto, metis vwb-lens-wb wb-lens.get-put)

lemma *sBoxIndStar*: $\Vdash [\Phi \Rightarrow [P] \Phi]_u \Longrightarrow \Phi \Vdash [P^*] \Phi$
 by (rel-simp, metis (mono-tags, lifting) mem-Collect-eq rtrancl-induct)

lemma *hoare-as-dynlog*: $\llbracket p \rrbracket Q \llbracket r \rrbracket_u = (p \Vdash [Q] r)$

by (rel-auto)

end

23 State Variable Declaration Parser

```
theory utp-state-parser
  imports utp-rel
begin
```

This theory sets up a parser for state blocks, as an alternative way of providing lenses to a predicate. A program with local variables can be represented by a predicate indexed by a tuple of lenses, where each lens represents a variable. These lenses must then be supplied with respect to a suitable state space. Instead of creating a type to represent this alphabet, we can create a product type for the state space, with an entry for each variable. Then each variable becomes a composition of the fst_L and snd_L lenses to index the correct position in the variable vector. We first create a vacuous definition that will mark when an indexed predicate denotes a state block.

definition *state-block* :: $('v \Rightarrow 'p) \Rightarrow 'v \Rightarrow 'p$ **where**
[upred-defs]: *state-block* $f\ x = f\ x$

We declare a number of syntax translations to produce lens and product types, to obtain a type for the overall state space, to construct a tuple that denotes the lens vector parameter, to construct the vector itself, and finally to construct the state declaration.

syntax

```
-lensT :: type  $\Rightarrow$  type  $\Rightarrow$  type (LENSTYPE'(-, -'))
-pairT :: type  $\Rightarrow$  type  $\Rightarrow$  type (PAIRTYPE'(-, -'))
-state-type :: pttrn  $\Rightarrow$  type
-state-tuple :: type  $\Rightarrow$  pttrn  $\Rightarrow$  logic
-state-lenses :: pttrn  $\Rightarrow$  logic
-state-decl :: pttrn  $\Rightarrow$  logic  $\Rightarrow$  logic (LOCAL - · - [0, 10] 10)
```

translations

```
(type) PAIRTYPE('a, 'b) => (type) 'a  $\times$  'b
(type) LENSTYPE('a, 'b) => (type) 'a  $\Longrightarrow$  'b

-state-type (-constrain x t) => t
-state-type (CONST Pair (-constrain x t) vs) => -pairT t (-state-type vs)

-state-tuple st (-constrain x t) => -constrain x (-lensT t st)
-state-tuple st (CONST Pair (-constrain x t) vs) =>
  CONST Product-Type.Pair (-constrain x (-lensT t st)) (-state-tuple st vs)

-state-decl vs P =>
  CONST state-block (-abs (-state-tuple (-state-type vs) vs) P) (-state-lenses vs)
-state-decl vs P <= CONST state-block (-abs vs P) k
```

parse-translation \ll

```
let
  open HOLogic;
  val lens-comp = Const (@{const-syntax lens-comp}, dummyT);
  val fst-lens = Const (@{const-syntax fst-lens}, dummyT);
  val snd-lens = Const (@{const-syntax snd-lens}, dummyT);
```

```

    val id-lens = Const (@{const-syntax id-lens}, dummyT);
    (* Construct a tuple of lenses for each of the possible locally declared variables *)
    fun
      state-lenses n st =
        if (n = 1)
          then st
          else pair-const dummyT dummyT $ (lens-comp $ fst-lens $ st) $ (state-lenses (n - 1) (lens-comp
$ snd-lens $ st));
    fun
      (* Add up the number of variable declarations in the tuple *)
      var-decl-num (Const (@{const-syntax Product-Type.Pair},-) $ - $ vs) = var-decl-num vs + 1 |
      var-decl-num - = 1;

    fun state-lens ctx [vs] = state-lenses (var-decl-num vs) id-lens ;
  in
    [(-state-lenses, state-lens)]
  end
end

```

23.1 Examples

```
term LOCAL (x::int, y::real, z::int) • x := (&x + &z)
```

```
lemma LOCAL p • II = II
  by (rel-auto)
```

end

24 Relational Operational Semantics

```

theory utp-rel-opsem
  imports
    utp-rel-laws
    utp-hoare
begin

```

This theory uses the laws of relational calculus to create a basic operational semantics. It is based on Chapter 10 of the UTP book [14].

```
fun trel :: 'α usubst × 'α hrel ⇒ 'α usubst × 'α hrel ⇒ bool (infix →u 85) where
  (σ, P) →u (ρ, Q) ⟷ (⟨σ⟩a ;; P) ⊆ (⟨ρ⟩a ;; Q)
```

```

lemma trans-trel:
  [ (σ, P) →u (ρ, Q); (ρ, Q) →u (φ, R) ] ⟹ (σ, P) →u (φ, R)
  by auto

```

```

lemma skip-trel: (σ, II) →u (σ, II)
  by simp

```

```

lemma assigns-trel: (σ, ⟨ρ⟩a) →u (ρ ◦ σ, II)
  by (simp add: assigns-comp)

```

```

lemma assign-trel:
  (σ, x := v) →u (σ(&x ↦s σ † v), II)

```

```

by (simp add: assigns-comp usubst)

lemma seq-trel:
  assumes  $(\sigma, P) \rightarrow_u (\varrho, Q)$ 
  shows  $(\sigma, P ;; R) \rightarrow_u (\varrho, Q ;; R)$ 
  by (metis (no-types, lifting) assms order-refl seqr-assoc seqr-mono trel.simps)

lemma seq-skip-trel:
   $(\sigma, II ;; P) \rightarrow_u (\sigma, P)$ 
  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  $\sigma_0$  satisfying
 $p$ , and  $Q$  reaches final state  $\sigma_1$  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  $x = \lambda \neg. a$  in spec, simp)
  apply (drule-tac  $x = \lambda \neg. b$  in spec, simp)
  done

declare trel.simps [simp del]

end

```

25 Symbolic Evaluation of Relational Programs

```
theory utp-sym-eval
  imports utp-rel-opsem
begin
```

The following operator applies a variable context Γ as an assignment, and composes it with a relation P for the purposes of evaluation.

definition *utp-sym-eval* :: '*s* usubst \Rightarrow '*s* hrel \Rightarrow '*s* hrel (infixr \models 55) **where**
[upred-defs]: utp-sym-eval Γ $P = ((\Gamma)_a \;; \; P)$

named-theorems *symeval*

lemma *seq-symeval* [*symeval*]: $\Gamma \models P \;; \; Q = (\Gamma \models P) \;; \; Q$
by (*rel-auto*)

lemma *assigns-symeval* [*symeval*]: $\Gamma \models \langle \sigma \rangle_a = (\sigma \circ \Gamma) \models II$
by (*rel-auto*)

lemma *term-symeval* [*symeval*]: $(\Gamma \models II) \;; \; P = \Gamma \models P$
by (*rel-auto*)

lemma *if-true-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{true} \rrbracket \Longrightarrow \Gamma \models (P \triangleleft b \triangleright_r Q) = \Gamma \models P$
by (*simp add: utp-sym-eval-def usubst assigns-r-comp*)

lemma *if-false-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{false} \rrbracket \Longrightarrow \Gamma \models (P \triangleleft b \triangleright_r Q) = \Gamma \models Q$
by (*simp add: utp-sym-eval-def usubst assigns-r-comp*)

lemma *while-true-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{true} \rrbracket \Longrightarrow \Gamma \models \text{while } b \text{ do } P \text{ od} = \Gamma \models (P \;; \; \text{while } b \text{ do } P \text{ od})$
by (*subst while-unfold, simp add: symeval*)

lemma *while-false-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{false} \rrbracket \Longrightarrow \Gamma \models \text{while } b \text{ do } P \text{ od} = \Gamma \models II$
by (*subst while-unfold, simp add: symeval*)

lemma *while-inv-true-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{true} \rrbracket \Longrightarrow \Gamma \models \text{while } b \text{ invr } S \text{ do } P \text{ od} = \Gamma \models (P \;; \; \text{while } b \text{ do } P \text{ od})$
by (*metis while-inv-def while-true-symeval*)

lemma *while-inv-false-symeval* [*symeval*]: $\llbracket \Gamma \uparrow b = \text{false} \rrbracket \Longrightarrow \Gamma \models \text{while } b \text{ invr } S \text{ do } P \text{ od} = \Gamma \models II$
by (*metis while-false-symeval while-inv-def*)

method *sym-eval* = (*simp add: symeval usubst lit-simps[THEN sym]*), (*simp del: One-nat-def add: One-nat-def[THEN sym]*)?

syntax

-terminated :: *logic* \Rightarrow *logic* (*terminated*: - [999] 999)

translations

terminated: $\Gamma == \Gamma \models II$

end

26 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 :: 'α cond  $\Rightarrow$  ('α, 'β) urel  $\Rightarrow$  'β cond where
  sp-upred p Q =  $\lfloor ([p]_{>} ;; Q) :: ('α, 'β) \text{urel} \rfloor_{>}$ 

adhoc-overloading
  usp sp-upred

declare sp-upred-def [upred-defs]

lemma sp-false [sp]: p sp false = false
  by (rel-simp)

lemma sp-true [sp]: q  $\neq$  false  $\implies$  q sp true = true
  by (rel-auto)

lemma sp-assigns-r [sp]:
  vwb-lens x  $\implies$  (p sp x := e) = ( $\exists v \cdot p \llbracket \llbracket v \rrbracket / x \rrbracket \wedge \&x =_u e \llbracket \llbracket v \rrbracket / x \rrbracket$ )
  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:
  ' $p \text{ sp } C \Rightarrow Q$ '  $\implies \{p\} C \{Q\}_u$ 
  by rel-blast

lemma sp-so:
  ' $p \text{ sp } C \Rightarrow Q$ ' =  $\{p\} C \{Q\}_u$ 
  by rel-blast

theorem sp-hoare-link:
   $\{p\} Q \{r\}_u \longleftrightarrow (r \sqsubseteq p \text{ sp } Q)$ 
  by rel-auto

lemma sp-while-r [sp]:
  assumes ' $\text{pre} \Rightarrow I$ ' and ' $\{I \wedge b\} C \{I\}_u$ ' and ' $I' \Rightarrow I$ '
  shows (pre sp invar I while⊥ b do C 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:
  ‘prog wp (true sp prog)’
  by rel-auto

lemma it-is-pre-condition:  $\{C \text{ wp } Q\} C \{Q\}_u$ 
  by rel-blast

lemma it-is-the-weakest-pre:  $P \Rightarrow C \text{ wp } Q' = \{P\} C \{Q\}_u$ 
  by rel-blast

lemma s-pre:  $P \Rightarrow C \text{ wp } Q' = \{P\} C \{Q\}_u$ 
  by rel-blast

end

```

27 Concurrent Programming

```

theory utp-concurrency
imports
  utp-hoare
  utp-rel
  utp-tactics
  utp-theory
begin

```

In this theory we describe the UTP scheme for concurrency, *parallel-by-merge*, which provides a general parallel operator parametrised by a “merge predicate” that explains how to merge the after states of the composed predicates. It can thus be applied to many languages and concurrency schemes, with this theory providing a number of generic laws. The operator is explained in more detail in Chapter 7 of the UTP book [14].

27.1 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 ( $'\alpha, '\beta_0, '\beta_1$ ) mrg =
  mrg-prior ::  $'\alpha$ 
  mrg-left  ::  $'\beta_0$ 
  mrg-right ::  $'\beta_1$ 

```

```

definition pre-uvar :: ( $'a \Longrightarrow '\alpha$ )  $\Rightarrow$  ( $'a \Longrightarrow (' \alpha, '\beta_0, '\beta_1) \text{ mrg}$ ) where
[upred-defs]: pre-uvar  $x = x ;_L \text{ mrg-prior}$ 

```

```

definition left-uvar :: ( $'a \Longrightarrow '\beta_0$ )  $\Rightarrow$  ( $'a \Longrightarrow (' \alpha, '\beta_0, '\beta_1) \text{ mrg}$ ) where
[upred-defs]: left-uvar  $x = x ;_L \text{ mrg-left}$ 

```

```

definition right-uvar :: ( $'a \Longrightarrow '\beta_1$ )  $\Rightarrow$  ( $'a \Longrightarrow (' \alpha, '\beta_0, '\beta_1) \text{ mrg}$ ) where
[upred-defs]: right-uvar  $x = x ;_L \text{ mrg-right}$ 

```

We set up syntax for the three variable classes using a subscript $<$, $0\text{-}x$, and $1\text{-}x$, respectively.

syntax

```
-svarpre  :: svid  $\Rightarrow$  svid ( $-<$  [995] 995)
-svarleft :: svid  $\Rightarrow$  svid ( $0--$  [995] 995)
-svarright :: svid  $\Rightarrow$  svid ( $1--$  [995] 995)
```

translations

```
-svarpre x == CONST pre-uvar x
-svarleft x == CONST left-uvar x
-svarright x == CONST right-uvar x
-svarpre  $\Sigma$  <= CONST pre-uvar  $1_L$ 
-svarleft  $\Sigma$  <= CONST left-uvar  $1_L$ 
-svarright  $\Sigma$  <= CONST right-uvar  $1_L$ 
```

We proved behavedness closure properties about the lenses.

lemma *left-uvar* [simp]: *vwb-lens* $x \Longrightarrow$ *vwb-lens* (*left-uvar* x)
by (*simp add: left-uvar-def*)

lemma *right-uvar* [simp]: *vwb-lens* $x \Longrightarrow$ *vwb-lens* (*right-uvar* x)
by (*simp add: right-uvar-def*)

lemma *pre-uvar* [simp]: *vwb-lens* $x \Longrightarrow$ *vwb-lens* (*pre-uvar* x)
by (*simp add: pre-uvar-def*)

lemma *left-uvar-mwb* [simp]: *mwb-lens* $x \Longrightarrow$ *mwb-lens* (*left-uvar* x)
by (*simp add: left-uvar-def*)

lemma *right-uvar-mwb* [simp]: *mwb-lens* $x \Longrightarrow$ *mwb-lens* (*right-uvar* x)
by (*simp add: right-uvar-def*)

lemma *pre-uvar-mwb* [simp]: *mwb-lens* $x \Longrightarrow$ *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]:
 $x \bowtie y \Longrightarrow$ *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]:

$x \bowtie y \implies \text{right-uvar } x \bowtie \text{right-uvar } y$
by (*simp add: right-uvar-def*)

lemma *pre-uvar-indep-left-uvar* [*simp*]:
 $\text{pre-uvar } x \bowtie \text{left-uvar } y$
by (*simp add: lens-indep-sym*)

lemma *pre-uvar-indep-right-uvar* [*simp*]:
 $\text{pre-uvar } x \bowtie \text{right-uvar } y$
by (*simp add: lens-indep-sym*)

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

27.2 Merge Predicates

A merge predicate is a relation whose input has three parts: the prior variables, the output variables of the left predicate, and the output of the right predicate.

type-synonym $'\alpha \text{ merge} = (('\alpha, '\alpha, '\alpha) \text{ mrg}, '\alpha) \text{ urel}$

skip is the merge predicate which ignores the output of both parallel predicates

definition $\text{skip}_m :: '\alpha \text{ merge}$ **where**
[upred-defs]: $\text{skip}_m = (\$ \mathbf{v}' =_u \$ \mathbf{v}_<)$

swap is a predicate that the swaps the left and right indices; it is used to specify commutativity of the parallel operator

definition $\text{swap}_m :: (('\alpha, '\beta, '\beta) \text{ mrg}) \text{ hrel}$ **where**
[upred-defs]: $\text{swap}_m = (0 - \mathbf{v}, 1 - \mathbf{v}) := (\& 1 - \mathbf{v}, \& 0 - \mathbf{v})$

A symmetric merge is one for which swapping the order of the merged concurrent predicates has no effect. We represent this by the following healthiness condition that states that swap_m is a left-unit.

abbreviation $\text{SymMerge} :: '\alpha \text{ merge} \Rightarrow '\alpha \text{ merge}$ **where**
 $\text{SymMerge}(M) \equiv (\text{swap}_m ;; M)$

27.3 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 $U0 :: (' \beta_0, (' \alpha, '\beta_0, '\beta_1) \text{ mrg}) \text{ urel}$ **where**
[upred-defs]: $U0 = (\$ 0 - \mathbf{v}' =_u \$ \mathbf{v})$

definition $U1 :: (' \beta_1, (' \alpha, '\beta_0, '\beta_1) \text{ mrg}) \text{ urel}$ **where**
[upred-defs]: $U1 = (\$ 1 - \mathbf{v}' =_u \$ \mathbf{v})$

lemma *U0-swap*: $(U0 ;; \text{swap}_m) = U1$
by (*rel-auto*)

lemma *U1-swap*: $(U1 ;; \text{swap}_m) = U0$
by (*rel-auto*)

As shown below, separating simulations can also be expressed using the following two alphabet extrusions

definition $U0\alpha$ **where** $[upred-defs]$: $U0\alpha = (1_L \times_L mrg-left)$

definition $U1\alpha$ **where** $[upred-defs]$: $U1\alpha = (1_L \times_L mrg-right)$

We then create the following intuitive syntax for separating simulations.

abbreviation $U0\text{-}\alpha\text{-lift}$ $(\lceil - \rceil_0)$ **where** $\lceil P \rceil_0 \equiv P \oplus_p U0\alpha$

abbreviation $U1\text{-}\alpha\text{-lift}$ $(\lceil - \rceil_1)$ **where** $\lceil P \rceil_1 \equiv P \oplus_p U1\alpha$

$\lceil P \rceil_0$ is predicate P where all variables are indexed by 0, and $\lceil P \rceil_1$ is where all variables are indexed by 1. We can thus equivalently express separating simulations using alphabet extrusion.

lemma $U0\text{-}as\text{-}\alpha$: $(P ;; U0) = \lceil P \rceil_0$
by $(rel\text{-}auto)$

lemma $U1\text{-}as\text{-}\alpha$: $(P ;; U1) = \lceil P \rceil_1$
by $(rel\text{-}auto)$

lemma $U0\alpha\text{-}vwb\text{-}lens$ $[simp]$: $vwb\text{-}lens\ U0\alpha$
by $(simp\ add:\ U0\alpha\text{-}def\ id\text{-}vwb\text{-}lens\ prod\text{-}vwb\text{-}lens)$

lemma $U1\alpha\text{-}vwb\text{-}lens$ $[simp]$: $vwb\text{-}lens\ U1\alpha$
by $(simp\ add:\ U1\alpha\text{-}def\ id\text{-}vwb\text{-}lens\ prod\text{-}vwb\text{-}lens)$

lemma $U0\alpha\text{-}indep\text{-}right\text{-}uvar$ $[simp]$: $vwb\text{-}lens\ x \implies U0\alpha \bowtie out\text{-}var\ (right\text{-}uvar\ x)$
by $(force\ intro:\ plus\text{-}pres\text{-}lens\text{-}indep\ fst\text{-}snd\text{-}lens\text{-}indep\ lens\text{-}indep\text{-}left\text{-}comp$
 $simp\ add:\ U0\alpha\text{-}def\ right\text{-}uvar\text{-}def\ out\text{-}var\text{-}def\ prod\text{-}as\text{-}plus\ lens\text{-}comp\text{-}assoc\ [THEN\ sym])$

lemma $U1\alpha\text{-}indep\text{-}left\text{-}uvar$ $[simp]$: $vwb\text{-}lens\ x \implies U1\alpha \bowtie out\text{-}var\ (left\text{-}uvar\ x)$
by $(force\ intro:\ plus\text{-}pres\text{-}lens\text{-}indep\ fst\text{-}snd\text{-}lens\text{-}indep\ lens\text{-}indep\text{-}left\text{-}comp$
 $simp\ add:\ U1\alpha\text{-}def\ left\text{-}uvar\text{-}def\ out\text{-}var\text{-}def\ prod\text{-}as\text{-}plus\ lens\text{-}comp\text{-}assoc\ [THEN\ sym])$

lemma $U0\text{-}\alpha\text{-lift}\text{-}bool\text{-}subst$ $[usubst]$:
 $\sigma(\$0 - x' \mapsto_s true) \uparrow \lceil P \rceil_0 = \sigma \uparrow \lceil P \llbracket true / \$x' \rrbracket \rceil_0$
 $\sigma(\$0 - x' \mapsto_s false) \uparrow \lceil P \rceil_0 = \sigma \uparrow \lceil P \llbracket false / \$x' \rrbracket \rceil_0$
by $(pred\text{-}auto+)$

lemma $U1\text{-}\alpha\text{-lift}\text{-}bool\text{-}subst$ $[usubst]$:
 $\sigma(\$1 - x' \mapsto_s true) \uparrow \lceil P \rceil_1 = \sigma \uparrow \lceil P \llbracket true / \$x' \rrbracket \rceil_1$
 $\sigma(\$1 - x' \mapsto_s false) \uparrow \lceil P \rceil_1 = \sigma \uparrow \lceil P \llbracket false / \$x' \rrbracket \rceil_1$
by $(pred\text{-}auto+)$

lemma $U0\text{-}\alpha\text{-out}\text{-}var$ $[alpha]$: $\lceil \$x' \rceil_0 = \$0 - x'$
by $(rel\text{-}auto)$

lemma $U1\text{-}\alpha\text{-out}\text{-}var$ $[alpha]$: $\lceil \$x' \rceil_1 = \$1 - x'$
by $(rel\text{-}auto)$

lemma $U0\text{-}skip$ $[alpha]$: $\lceil II \rceil_0 = (\$0 - \mathbf{v}' =_u \$\mathbf{v})$
by $(rel\text{-}auto)$

lemma $U1\text{-}skip$ $[alpha]$: $\lceil II \rceil_1 = (\$1 - \mathbf{v}' =_u \$\mathbf{v})$
by $(rel\text{-}auto)$

lemma $U0\text{-}segr$ $[alpha]$: $\lceil P ;; Q \rceil_0 = P ;; \lceil Q \rceil_0$
by $(rel\text{-}auto)$

lemma *U1-seqr* [*alpha*]: $\lceil P \rrceil_1 = P \rrceil; \lceil Q \rrceil_1$
by (*rel-auto*)

lemma *U0 α -comp-in-var* [*alpha*]: (*in-var* *x*) ;_L *U0 α* = *in-var* *x*
by (*simp add: U0 α -def alpha-in-var in-var-prod-lens pre-uvar-def*)

lemma *U0 α -comp-out-var* [*alpha*]: (*out-var* *x*) ;_L *U0 α* = *out-var* (*left-uvar* *x*)
by (*simp add: U0 α -def alpha-out-var id-wb-lens left-uvar-def out-var-prod-lens*)

lemma *U1 α -comp-in-var* [*alpha*]: (*in-var* *x*) ;_L *U1 α* = *in-var* *x*
by (*simp add: U1 α -def alpha-in-var in-var-prod-lens pre-uvar-def*)

lemma *U1 α -comp-out-var* [*alpha*]: (*out-var* *x*) ;_L *U1 α* = *out-var* (*right-uvar* *x*)
by (*simp add: U1 α -def alpha-out-var id-wb-lens right-uvar-def out-var-prod-lens*)

27.4 Associative Merges

Associativity of a merge means that if we construct a three way merge from a two way merge and then rotate the three inputs of the merge to the left, then we get exactly the same three way merge back.

We first construct the operator that constructs the three way merge by effectively wiring up the two way merge in an appropriate way.

definition *ThreeWayMerge* :: ' α merge \Rightarrow ((α , ' α , (α , ' α , ' α) mrg) mrg, ' α) urel ($\mathbf{M3}'(-)$) **where**
[upred-defs]: *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** *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* :: ' α merge \Rightarrow bool **where**
[upred-defs]: *AssocMerge* *M* = (*rotate_m* ;; $\mathbf{M3}(M) = \mathbf{M3}(M)$)

27.5 Parallel Operators

We implement the following useful abbreviation for separating of two parallel processes and copying of the before variables, all to act as input to the merge predicate.

abbreviation *par-sep* (*infixr* \parallel_s 85) **where**
 $P \parallel_s Q \equiv (P \rrceil; U0) \wedge (Q \rrceil; U1) \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}$

The following implementation of parallel by merge is less general than the book version, in that it does not properly partition the alphabet into two disjoint segments. We could actually achieve this specifying lenses into the larger alphabet, but this would complicate the definition of programs. May reconsider later.

definition
par-by-merge :: (α , ' β) urel \Rightarrow ((α , ' β , ' γ) mrg, ' δ) urel \Rightarrow (α , ' γ) urel \Rightarrow (α , ' δ) urel
 (- \parallel_s - [85,0,86] 85)
where *[upred-defs]*: $P \parallel_M Q = (P \parallel_s Q \rrceil; M)$

lemma *par-by-merge-alt-def*: $P \parallel_M Q = (\lceil P \rrceil_0 \wedge \lceil Q \rrceil_1 \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}) \rrceil; M$
by (*simp add: par-by-merge-def U0-as-alpha U1-as-alpha*)

lemma *shEx-pbm-left*: $((\exists x \cdot P x) \parallel_M Q) = (\exists x \cdot (P x \parallel_M Q))$
by (*rel-auto*)

lemma *shEx-pbm-right*: $(P \parallel_M (\exists x \cdot Q x)) = (\exists x \cdot (P \parallel_M Q x))$
by (*rel-auto*)

27.6 Unrestriction Laws

lemma *unrest-in-par-by-merge* [*unrest*]:
 $\llbracket \$x \# P; \$x_{<} \# M; \$x \# Q \rrbracket \implies \$x \# P \parallel_M Q$
by (*rel-auto*, *fastforce*+)

lemma *unrest-out-par-by-merge* [*unrest*]:
 $\llbracket \$x' \# M \rrbracket \implies \$x' \# P \parallel_M Q$
by (*rel-auto*)

27.7 Substitution laws

Substitution is a little tricky because when we push the expression through the composition operator the alphabet of the expression must also change. Consequently for now we only support literal substitution, though this could be generalised with suitable alphabet coercions. We need quite a number of variants to support this which are below.

lemma *U0-seq-subst*: $(P ;; U0) \llbracket \llbracket v \rrbracket / \$0 - x' \rrbracket = (P \llbracket \llbracket v \rrbracket / \$x' \rrbracket ;; U0)$
by (*rel-auto*)

lemma *U1-seq-subst*: $(P ;; U1) \llbracket \llbracket v \rrbracket / \$1 - x' \rrbracket = (P \llbracket \llbracket v \rrbracket / \$x' \rrbracket ;; U1)$
by (*rel-auto*)

lemma *lit-pbm-subst* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s \llbracket v \rrbracket) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket \llbracket v \rrbracket / \$x \rrbracket) \parallel_M \llbracket \llbracket v \rrbracket / \$x_{<} \rrbracket (Q \llbracket \llbracket v \rrbracket / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s \llbracket v \rrbracket) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_M \llbracket \llbracket v \rrbracket / \$x' \rrbracket Q)$
by (*rel-auto*)+

lemma *bool-pbm-subst* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s \text{false}) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket \text{false} / \$x \rrbracket) \parallel_M \llbracket \text{false} / \$x_{<} \rrbracket (Q \llbracket \text{false} / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s \text{true}) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket \text{true} / \$x \rrbracket) \parallel_M \llbracket \text{true} / \$x_{<} \rrbracket (Q \llbracket \text{true} / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s \text{false}) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_M \llbracket \text{false} / \$x' \rrbracket Q)$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s \text{true}) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_M \llbracket \text{true} / \$x' \rrbracket Q)$
by (*rel-auto*)+

lemma *zero-one-pbm-subst* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s 0) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket 0 / \$x \rrbracket) \parallel_M \llbracket 0 / \$x_{<} \rrbracket (Q \llbracket 0 / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s 1) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket 1 / \$x \rrbracket) \parallel_M \llbracket 1 / \$x_{<} \rrbracket (Q \llbracket 1 / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s 0) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_M \llbracket 0 / \$x' \rrbracket Q)$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s 1) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_M \llbracket 1 / \$x' \rrbracket Q)$
by (*rel-auto*)+

lemma *numeral-pbm-subst* [*usubst*]:
fixes $x :: (- \implies 'a)$
shows
 $\bigwedge P Q M \sigma. \sigma(\$x \mapsto_s \text{numeral } n) \dagger (P \parallel_M Q) = \sigma \dagger ((P \llbracket \text{numeral } n / \$x \rrbracket) \parallel_M \llbracket \text{numeral } n / \$x < \rrbracket (Q \llbracket \text{numeral } n / \$x \rrbracket))$
 $\bigwedge P Q M \sigma. \sigma(\$x' \mapsto_s \text{numeral } n) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_M \llbracket \text{numeral } n / \$x' \rrbracket Q)$
by (*rel-auto*)⁺

27.8 Parallel-by-merge laws

lemma *par-by-merge-false* [*simp*]:

$P \parallel_{\text{false}} Q = \text{false}$
by (*rel-auto*)

lemma *par-by-merge-left-false* [*simp*]:

$\text{false} \parallel_M Q = \text{false}$
by (*rel-auto*)

lemma *par-by-merge-right-false* [*simp*]:

$P \parallel_M \text{false} = \text{false}$
by (*rel-auto*)

lemma *par-by-merge-seq-add*: $(P \parallel_M Q) ;; R = (P \parallel_M ;; R Q)$

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 $P ;; \text{true} = \text{true} \quad Q ;; \text{true} = \text{true}$
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 $Q \parallel_{\text{swap}_m} ;; M P = (((Q ;; U0) \wedge (P ;; U1) \wedge \$v_{<}' =_u \$v) ;; \text{swap}_m) ;; M$
by (*simp add: par-by-merge-def seqr-assoc*)

also have $\dots = (((Q ;; U0 ;; \text{swap}_m) \wedge (P ;; U1 ;; \text{swap}_m) \wedge \$v_{<}' =_u \$v) ;; M)$
by (*rel-auto*)

also have $\dots = (((Q ;; U1) \wedge (P ;; U0) \wedge \$v_{<}' =_u \$v) ;; M)$
by (*simp add: U0-swap U1-swap*)

also have $\dots = P \parallel_M Q$

by (*simp add: par-by-merge-def utp-pred-laws.inf.left-commute*)

finally show *?thesis* ..

qed

theorem *par-by-merge-commute*:

assumes M is *SymMerge*

shows $P \parallel_M Q = Q \parallel_M P$
by (*metis Healthy-if assms par-by-merge-commute-swap*)

lemma *par-by-merge-mono-1*:

assumes $P_1 \sqsubseteq P_2$
shows $P_1 \parallel_M Q \sqsubseteq P_2 \parallel_M Q$
using *assms* **by** (*rel-auto*)

lemma *par-by-merge-mono-2*:

assumes $Q_1 \sqsubseteq Q_2$
shows $(P \parallel_M Q_1) \sqsubseteq (P \parallel_M Q_2)$
using *assms* **by** (*rel-blast*)

lemma *par-by-merge-mono*:

assumes $P_1 \sqsubseteq P_2$ $Q_1 \sqsubseteq Q_2$
shows $P_1 \parallel_M Q_1 \sqsubseteq P_2 \parallel_M Q_2$
by (*meson assms dual-order.trans par-by-merge-mono-1 par-by-merge-mono-2*)

theorem *par-by-merge-assoc*:

assumes *M is SymMerge AssocMerge M*
shows $(P \parallel_M Q) \parallel_M R = P \parallel_M (Q \parallel_M R)$

proof –

have $(P \parallel_M Q) \parallel_M R = ((P ;; U0) \wedge (Q ;; U0 ;; U1) \wedge (R ;; U1 ;; U1) \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}) ;; \mathbf{M3}(M)$
by (*rel-blast*)
also have $\dots = ((P ;; U0) \wedge (Q ;; U0 ;; U1) \wedge (R ;; U1 ;; U1) \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}) ;; rotate_m ;; \mathbf{M3}(M)$
using *AssocMerge-def assms(2)* **by** *force*
also have $\dots = ((Q ;; U0) \wedge (R ;; U0 ;; U1) \wedge (P ;; U1 ;; U1) \wedge \$\mathbf{v}_{<}' =_u \$\mathbf{v}) ;; \mathbf{M3}(M)$
by (*rel-blast*)
also have $\dots = (Q \parallel_M R) \parallel_M P$
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*:

$(\bigcap i \in I \cdot P(i)) \parallel_M Q = (\bigcap i \in I \cdot P(i) \parallel_M Q)$
by (*rel-auto*)

theorem *par-by-merge-USUP-ind-left*:
 $(\prod i \cdot P(i)) \parallel_M Q = (\prod i \cdot P(i) \parallel_M Q)$
by (*rel-auto*)

theorem *par-by-merge-USUP-mem-right*:
 $P \parallel_M (\prod_{i \in I} Q(i)) = (\prod_{i \in I} P \parallel_M Q(i))$
by (*rel-auto*)

theorem *par-by-merge-USUP-ind-right*:
 $P \parallel_M (\prod i \cdot Q(i)) = (\prod i \cdot P \parallel_M Q(i))$
by (*rel-auto*)

27.9 Example: Simple State-Space Division

The following merge predicate divides the state space using a pair of independent lenses.

definition *StateMerge* :: $('a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a \text{ merge } (M[-]_\sigma)$ **where**
 $[upred-defs]: M[a|b]_\sigma = (\$v' =_u (\$v_< \oplus \$0 - v \text{ on } \&a) \oplus \$1 - v \text{ on } \&b)$

lemma *swap-StateMerge*: $a \bowtie b \Rightarrow (swap_m ;; M[a|b]_\sigma) = M[b|a]_\sigma$
by (*rel-auto*, *simp-all add: lens-indep-comm*)

abbreviation *StateParallel* :: $'a \text{ hrel} \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a \text{ hrel} \Rightarrow 'a \text{ hrel } (-|-|_\sigma -$
 $[85,0,0,86] \ 86)$
where $P \mid a|b|_\sigma Q \equiv P \parallel_{M[a|b]_\sigma} Q$

lemma *StateParallel-commute*: $a \bowtie b \Rightarrow P \mid a|b|_\sigma Q = Q \mid b|a|_\sigma P$
by (*metis par-by-merge-commute-swap swap-StateMerge*)

lemma *StateParallel-form*:
 $P \mid a|b|_\sigma Q = (\exists (st_0, st_1) \cdot P[\ll st_0 \gg / \$v'] \wedge Q[\ll st_1 \gg / \$v'] \wedge \$v' =_u (\$v \oplus \ll st_0 \gg \text{ on } \&a) \oplus \ll st_1 \gg \text{ on } \&b)$
by (*rel-auto*)

lemma *StateParallel-form'*:
assumes *vwb-lens a vwb-lens b a* \bowtie *b*
shows $P \mid a|b|_\sigma Q = \{\&a, \&b\} : [(P \vdash_v \{\$v, \$a'\}) \wedge (Q \vdash_v \{\$v, \$b'\})]$
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  $\llbracket c \rrbracket P \mid a \mid b \mid_\sigma Q \llbracket d_1 \wedge d_2 \rrbracket_u$ 
proof –
  — Parallelise the specification
  from assms(4,5)
  have  $1: (\llbracket c \rrbracket_< \Rightarrow \llbracket d_1 \wedge d_2 \rrbracket_>) \sqsubseteq (\llbracket c \rrbracket_< \Rightarrow \llbracket d_1 \rrbracket_>) \mid a \mid b \mid_\sigma (\llbracket c \rrbracket_< \Rightarrow \llbracket d_2 \rrbracket_>)$  (is ?lhs  $\sqsubseteq$  ?rhs)
    by (simp add: StateParallel-form, rel-auto, metis assms(3) lens-indep-comm)
  — Prove Hoare rule by monotonicity of parallelism
  have  $2: ?rhs \sqsubseteq P \mid a \mid b \mid_\sigma Q$ 
  proof (rule par-by-merge-mono)
    show  $(\llbracket c \rrbracket_< \Rightarrow \llbracket d_1 \rrbracket_>) \sqsubseteq P$ 
    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  $b$   $a \bowtie b$   $a \Vdash d_1$   $b \Vdash d_2$   $a \# c_1$   $b \# c_1$   $\llbracket c_1 \wedge c_2 \rrbracket P \llbracket d_1 \rrbracket_u \llbracket c_1 \wedge c_2 \rrbracket Q \llbracket d_2 \rrbracket_u$ 
  shows  $\llbracket c_1 \wedge c_2 \rrbracket P \mid a \mid b \mid_\sigma Q \llbracket c_1 \wedge d_1 \wedge d_2 \rrbracket_u$ 
proof –
  have  $\llbracket c_1 \wedge c_2 \rrbracket \{ \&a, \&b \}: [P \mid a \mid b \mid_\sigma Q] \llbracket c_1 \wedge d_1 \wedge d_2 \rrbracket_u$ 
    by (auto intro!: frame-hoare-r' StateParallel-hoare simp add: assms unrest plus-vwb-lens)
  thus ?thesis
    by (simp add: StateParallel-frame assms)
qed
end

```

28 Meta-theory for the Standard Core

```

theory utp
imports
  utp-var
  utp-expr
  utp-expr-insts
  utp-expr-funcs
  utp-unrest
  utp-usedby
  utp-subst
  utp-meta-subst
  utp-alphabet
  utp-lift
  utp-pred
  utp-pred-laws
  utp-recursion
  utp-dynlog
  utp-rel

```

```

    utp-rel-laws
    utp-sequent
    utp-state-parser
    utp-sym-eval
    utp-tactics
    utp-hoare
    utp-wp
    utp-sp
    utp-theory
    utp-concurrency
    utp-rel-opsem
begin end

```

29 Overloaded Expression Constructs

```

theory utp-expr-ovld
  imports utp
begin

```

29.1 Overloadable Constants

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

```

consts
  — Empty elements, for example empty set, nil list, 0...
  uempty    :: 'f
  — Function application, map application, list application...
  uapply    :: 'f ⇒ 'k ⇒ 'v
  — Function update, map update, list update...
  uupd      :: 'f ⇒ 'k ⇒ 'v ⇒ 'f
  — Domain of maps, lists...
  udom      :: 'f ⇒ 'a set
  — Range of maps, lists...
  uran      :: 'f ⇒ 'b set
  — Domain restriction
  udomres   :: 'a set ⇒ 'f ⇒ 'f
  — Range restriction
  uranres   :: 'f ⇒ 'b set ⇒ 'f
  — Collection cardinality
  ucard     :: 'f ⇒ nat
  — Collection summation
  usums     :: 'f ⇒ 'a
  — Construct a collection from a list of entries
  uentries  :: 'k set ⇒ ('k ⇒ 'v) ⇒ 'f

```

We need a function corresponding to function application in order to overload.

```

definition fun-apply :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
where fun-apply f x = f x

```

```

declare fun-apply-def [simp]

```


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

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

adhoc-overloading

uempty 0 **and**
uapply *fun-apply* **and** *uapply* *nth* **and** *uapply* *pfun-app* **and**
uapply *ffun-app* **and**
uupd *pfun-upd* **and** *uupd* *ffun-upd* **and** *uupd* *list-augment* **and**
udom *Domain* **and** *udom* *pdom* **and** *udom* *fdom* **and** *udom* *seq-dom* **and**
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*

29.2 Syntax Translations

syntax

-*uundef* :: *logic* (\perp_u)
-*umap-empty* :: *logic* ($[]_u$)
-*uapply* :: ('a \Rightarrow 'b, 'α) *uexpr* \Rightarrow *utuple-args* \Rightarrow ('b, 'α) *uexpr* (-'(-')_a [999,0] 999)
-*umaplet* :: [*logic*, *logic*] \Rightarrow *umaplet* (- / \mapsto / -)
:: *umaplet* \Rightarrow *umaplets* (-)
-*UMaplets* :: [*umaplet*, *umaplets*] \Rightarrow *umaplets* (-, / -)
-*UMapUpd* :: [*logic*, *umaplets*] \Rightarrow *logic* (-/'(-')_u [900,0] 900)
-*UMap* :: *umaplets* \Rightarrow *logic* ((1[-]_u))
-*ucard* :: *logic* \Rightarrow *logic* ($\#_u$ '(-'))
-*udom* :: *logic* \Rightarrow *logic* (*dom*_u'(-'))
-*uran* :: *logic* \Rightarrow *logic* (*ran*_u'(-'))
-*usum* :: *logic* \Rightarrow *logic* (*sum*_u'(-'))
-*udom-res* :: *logic* \Rightarrow *logic* \Rightarrow *logic* (**infixl** \triangleleft_u 85)
-*uran-res* :: *logic* \Rightarrow *logic* \Rightarrow *logic* (**infixl** \triangleright_u 85)
-*uentries* :: *logic* \Rightarrow *logic* \Rightarrow *logic* (*entr*_u'(-,-'))

translations

— Pretty printing for adhoc-overloaded constructs

$f(x)_a <= \text{CONST } uapply\ f\ x$
 $dom_u(f) <= \text{CONST } udom\ f$
 $ran_u(f) <= \text{CONST } uran\ f$
 $A \triangleleft_u f <= \text{CONST } udomres\ A\ f$
 $f \triangleright_u A <= \text{CONST } uranres\ f\ A$
 $\#_u(f) <= \text{CONST } ucard\ f$
 $f(k \mapsto v)_u <= \text{CONST } uupd\ f\ k\ v$
 $0 <= \text{CONST } uempty$ — We have to do this so we don't see *uempty*. Is there a better way of printing?

— Overloaded construct translations

$f(x,y,z,u)_a == \text{CONST } bop\ \text{CONST } uapply\ f\ (x,y,z,u)_u$
 $f(x,y,z)_a == \text{CONST } bop\ \text{CONST } uapply\ f\ (x,y,z)_u$
 $f(x,y)_a == \text{CONST } bop\ \text{CONST } uapply\ f\ (x,y)_u$
 $f(x)_a == \text{CONST } bop\ \text{CONST } uapply\ f\ x$
 $\#_u(xs) == \text{CONST } uop\ \text{CONST } ucard\ xs$
 $sum_u(A) == \text{CONST } uop\ \text{CONST } usums\ A$
 $dom_u(f) == \text{CONST } uop\ \text{CONST } udom\ f$
 $ran_u(f) == \text{CONST } uop\ \text{CONST } uran\ f$

$\llbracket_u \quad \Rightarrow \langle\langle \text{CONST } uempty \rangle\rangle$
 $\perp_u \quad == \langle\langle \text{CONST } undefined \rangle\rangle$
 $A \triangleleft_u f == \text{CONST } bop (\text{CONST } udomres) A f$
 $f \triangleright_u A == \text{CONST } bop (\text{CONST } uranres) f A$
 $entr_u(d, f) == \text{CONST } bop \text{CONST } uentries d \langle\langle f \rangle\rangle$
 $-UMapUpd m (-UMaplets xy ms) == -UMapUpd (-UMapUpd m xy) ms$
 $-UMapUpd m (-umaplet x y) == \text{CONST } trop \text{CONST } uupd m x y$
 $-UMap ms == -UMapUpd \llbracket_u ms$
 $-UMap (-UMaplets ms1 ms2) \leq -UMapUpd (-UMap ms1) ms2$
 $-UMaplets ms1 (-UMaplets ms2 ms3) \leq -UMaplets (-UMaplets ms1 ms2) ms3$

29.3 Simplifications

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*)

lemma *lit-plus-appl [lit-norm]*: $\langle\langle (+) \rangle\rangle(x)_a(y)_a = x + y$ **by** (*simp add: uexpr-defs, transfer, simp*)

lemma *lit-minus-appl [lit-norm]*: $\langle\langle (-) \rangle\rangle(x)_a(y)_a = x - y$ **by** (*simp add: uexpr-defs, transfer, simp*)

lemma *lit-mult-appl [lit-norm]*: $\langle\langle times \rangle\rangle(x)_a(y)_a = x * y$ **by** (*simp add: uexpr-defs, transfer, simp*)

lemma *lit-divide-apply [lit-norm]*: $\langle\langle (/) \rangle\rangle(x)_a(y)_a = x / y$ **by** (*simp add: uexpr-defs, transfer, simp*)

lemma *pfun-entries-apply [simp]*:

$(entr_u(d, f) :: ((k, 'v) pfun, 'α) uexpr)(i)_a = ((\langle\langle f \rangle\rangle(i)_a) \triangleleft i \in_u d \triangleright \perp_u)$
by (*pred-auto*)

lemma *udom-uupdate-pfun [simp]*:

fixes $m :: ((k, 'v) pfun, 'α) uexpr$
shows $dom_u(m(k \mapsto v)_u) = \{k\}_u \cup_u dom_u(m)$
by (*rel-auto*)

lemma *uapply-uupdate-pfun [simp]*:

fixes $m :: ((k, 'v) pfun, 'α) uexpr$
shows $(m(k \mapsto v)_u)(i)_a = v \triangleleft i =_u k \triangleright m(i)_a$
by (*rel-auto*)

29.4 Indexed Assignment

syntax

— Indexed assignment
 $-assignment-upd :: svid \Rightarrow uexp \Rightarrow uexp \Rightarrow logic (([-] := / -) [63, 0, 0] 62)$

translations

— Indexed assignment uses the overloaded collection update function *uupd*.
 $-assignment-upd x k v \Rightarrow x := \&x(k \mapsto v)_u$

end

30 Meta-theory for the Standard Core with Overloaded Constructs

theory *utp-full*

imports *utp utp-expr-ovld*

begin end

31 UTP Easy Expression Parser

```
theory utp-easy-parser
  imports utp-full
begin
```

31.1 Replacing the Expression Grammar

The following theory provides an easy to use expression parser that is primarily targetted towards expressing programs. Unlike the built-in UTP expression syntax, this uses a closed grammar separate to the HOL *logic* nonterminal, that gives more freedom in what can be expressed. In particular, identifiers are interpreted as UTP variables rather than HOL variables and functions do not require subscripts and other strange decorations.

The first step is to remove the from the UTP parse the following grammar rule that uses arbitrary HOL logic to represent expressions. Instead, we will populate the *uexp* grammar manually.

```
purge-syntax
-uexp-l :: logic  $\Rightarrow$  uexp (- [64] 64)
```

31.2 Expression Operators

syntax

```
-ue-quote :: uexp  $\Rightarrow$  logic ('(-)')e
-ue-tuple :: uexprs  $\Rightarrow$  uexp ('(-)')
-ue-lit :: logic  $\Rightarrow$  uexp ( $\llcorner$ - $\gg$ )
-ue-var :: svid  $\Rightarrow$  uexp (-)
-ue-eq :: uexp  $\Rightarrow$  uexp  $\Rightarrow$  uexp (infix = 150)
-ue-uop :: id  $\Rightarrow$  uexp  $\Rightarrow$  uexp ('(-)') [999,0] 999)
-ue-bop :: id  $\Rightarrow$  uexp  $\Rightarrow$  uexp  $\Rightarrow$  uexp ('(-, -)') [999,0,0] 999)
-ue-trop :: id  $\Rightarrow$  uexp  $\Rightarrow$  uexp  $\Rightarrow$  uexp  $\Rightarrow$  uexp ('(-, -, -)') [999,0,0,0] 999)
-ue-apply :: uexp  $\Rightarrow$  uexp  $\Rightarrow$  uexp (-[-] [999] 999)
```

translations

```
-ue-quote e => e
-ue-tuple (-uexprs x (-uexprs y z)) => -ue-tuple (-uexprs x (-ue-tuple (-uexprs y z)))
-ue-tuple (-uexprs x y) => CONST bop CONST Pair x y
-ue-tuple x => x
-ue-lit x => CONST lit x
-ue-var x => CONST utp-expr.var (CONST pr-var x)
-ue-eq x y => x =u y
-ue-uop f x => CONST uop f x
-ue-bop f x y => CONST bop f x y
-ue-trop f x y => CONST trop f x y
-ue-apply f x => f(x)a
```

31.3 Predicate Operators

syntax

```
-ue-true :: uexp (true)
-ue-false :: uexp (false)
-ue-not :: uexp  $\Rightarrow$  uexp ( $\neg$  - [40] 40)
-ue-conj :: uexp  $\Rightarrow$  uexp  $\Rightarrow$  uexp (infixr  $\wedge$  135)
-ue-disj :: uexp  $\Rightarrow$  uexp  $\Rightarrow$  uexp (infixr  $\vee$  130)
-ue-impl :: uexp  $\Rightarrow$  uexp  $\Rightarrow$  uexp (infixr  $\Rightarrow$  125)
```

`-ue-iff` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infixr** \Rightarrow 125)
`-ue-mem` :: $uexp \Rightarrow uexp \Rightarrow uexp$ ((-/ \in -) [151, 151] 150)
`-ue-nmem` :: $uexp \Rightarrow uexp \Rightarrow uexp$ ((-/ \notin -) [151, 151] 150)

translations

`-ue-true` => *CONST true-upred*
`-ue-false` => *CONST false-upred*
`-ue-not` p => *CONST not-upred* p
`-ue-conj` p q => $p \wedge_p q$
`-ue-disj` p q => $p \vee_p q$
`-ue-impl` p q => $p \Rightarrow q$
`-ue-iff` p q => $p \Leftrightarrow q$
`-ue-mem` x A => $x \in_u A$
`-ue-nmem` x A => $x \notin_u A$

31.4 Arithmetic Operators

syntax

`-ue-num` :: $num-const \Rightarrow uexp$ (-)
`-ue-size` :: $uexp \Rightarrow uexp$ (#- [999] 999)
`-ue-eq` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infix** = 150)
`-ue-le` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infix** \leq 150)
`-ue-lt` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infix** < 150)
`-ue-ge` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infix** \geq 150)
`-ue-gt` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infix** > 150)
`-ue-zero` :: $uexp$ (0)
`-ue-one` :: $uexp$ (1)
`-ue-plus` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infixl** + 165)
`-ue-uminus` :: $uexp \Rightarrow uexp$ (- - [181] 180)
`-ue-minus` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infixl** - 165)
`-ue-times` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infixl** * 170)
`-ue-div` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infixl** div 170)

translations

`-ue-num` x => *-Numeral* x
`-ue-size` e => $\#_u(e)$
`-ue-le` x y => $x \leq_u y$
`-ue-lt` x y => $x <_u y$
`-ue-ge` x y => $x \geq_u y$
`-ue-gt` x y => $x >_u y$
`-ue-zero` => 0
`-ue-one` => 1
`-ue-plus` x y => $x + y$
`-ue-uminus` x => $-x$
`-ue-minus` x y => $x - y$
`-ue-times` x y => $x * y$
`-ue-div` x y => *CONST divide* x y

31.5 Sets

syntax

`-ue-empset` :: $uexp$ ({})
`-ue-setprod` :: $uexp \Rightarrow uexp \Rightarrow uexp$ (**infixr** \times 80)
`-ue-atLeastAtMost` :: $uexp \Rightarrow uexp \Rightarrow uexp$ ((1{..<})
`-ue-atLeastLessThan` :: $uexp \Rightarrow uexp \Rightarrow uexp$ ((1{.. $<$ -})

translations

$-ue-empset \Rightarrow \{\}_u$
 $-ue-setprod\ e\ f \Rightarrow CONST\ bop\ (CONST\ Product-Type.Times)\ e\ f$
 $-ue-atLeastAtMost\ m\ n \Rightarrow \{m..n\}_u$
 $-ue-atLeastLessThan\ m\ n \Rightarrow \{m..<n\}_u$

31.6 Imperative Program Syntax

syntax

$-ue-if-then \quad ::\ uexp \Rightarrow logic \Rightarrow logic \Rightarrow logic\ (if - then - else - fi)$
 $-ue-hoare \quad ::\ uexp \Rightarrow logic \Rightarrow uexp \Rightarrow logic\ (\{\{-\}\} / - / \{\{-\}\})$
 $-ue-wp \quad ::\ logic \Rightarrow uexp \Rightarrow uexp\ (\mathbf{infix}\ wp\ 60)$

translations

$-ue-if-then\ b\ P\ Q \Rightarrow P \triangleleft b \triangleright_r Q$
 $-ue-hoare\ b\ P\ c \Rightarrow \llbracket b \rrbracket P \llbracket c \rrbracket_u$
 $-ue-wp\ P\ b \Rightarrow P\ wp\ b$

end

References

- [1] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2015.
- [2] A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In *RAMiCS 2012*, volume 7560 of *LNCS*. Springer, September 2012.
- [3] R.-J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.
- [4] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In *Proc. 5th Intl. Conf. on Mathematical Knowledge Management (MKM)*, volume 4108 of *LNCS*, pages 31–43. Springer, 2006.
- [5] C. Ballarin et al. The Isabelle/HOL Algebra Library. *Isabelle/HOL*, October 2017. <https://isabelle.in.tum.de/library/HOL/HOL-Algebra/document.pdf>.
- [6] A. Cavalcanti and J. Woodcock. A tutorial introduction to designs in unifying theories of programming. In *Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*, pages 40–66. Springer, 2004.
- [7] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.
- [8] 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.