

# A Shallow Model of the UTP in Isabelle/HOL

Abderrahmane Feliachi      Simon Foster      Marie-Claude Gaudel  
Burkhart Wolff              Frank Zeyda

March 16, 2016

## Contents

<b>1</b>	<b>UTP variables</b>	<b>2</b>
1.1	Deep UTP variables . . . . .	5
1.2	Cardinalities . . . . .	5
1.3	Injection functions . . . . .	6
1.4	Deep variables . . . . .	7
<b>2</b>	<b>UTP expressions</b>	<b>10</b>
2.1	Evaluation laws for expressions . . . . .	16
<b>3</b>	<b>Unrestriction</b>	<b>17</b>
<b>4</b>	<b>Substitution</b>	<b>19</b>
4.1	Substitution definitions . . . . .	19
4.2	Substitution laws . . . . .	20
<b>5</b>	<b>Lifting expressions</b>	<b>22</b>
5.1	Lifting definitions . . . . .	23
5.2	Lifting laws . . . . .	23
<b>6</b>	<b>Alphabetised Predicates</b>	<b>24</b>
6.1	Predicate syntax . . . . .	24
6.2	Predicate operators . . . . .	25
6.3	Proof support . . . . .	28
6.4	Unrestriction Laws . . . . .	28
6.5	Substitution Laws . . . . .	29
6.6	Predicate Laws . . . . .	30
6.7	Quantifier lifting . . . . .	34
<b>7</b>	<b>Alphabetised relations</b>	<b>34</b>
7.1	Unrestriction Laws . . . . .	36
7.2	Substitution laws . . . . .	37
7.3	Lifting laws . . . . .	38
7.4	Relation laws . . . . .	38
7.5	Converse laws . . . . .	41
7.6	Weakest precondition calculus . . . . .	43
<b>8</b>	<b>UTP Theories</b>	<b>44</b>

<b>9 Example UTP theory: Boyle's laws</b>	<b>46</b>
<b>10 Designs</b>	<b>47</b>
10.1 Definitions . . . . .	48
10.2 Design laws . . . . .	50
10.3 H1: No observation is allowed before initiation . . . . .	53
10.4 H2: A specification cannot require non-termination . . . . .	54
10.5 H3: The design assumption is a precondition . . . . .	57
10.6 H4: Feasibility . . . . .	59
<b>11 Concurrent programming</b>	<b>59</b>
11.1 Design parallel composition . . . . .	59
11.2 Parallel by merge . . . . .	60
<b>12 Reactive processes</b>	<b>61</b>
12.1 Preliminaries . . . . .	61
12.2 R1: Events cannot be undone . . . . .	63
12.3 R2 . . . . .	64
12.4 R3 . . . . .	66

## 1 UTP variables

**theory** *utp-var*

**imports**

../contrib/Kleene-Algebras/Quantales  
 ../utils/cardinals  
 ../utils/Continuum  
 ../utils/finite-bijection  
 ../utils/Lenses  
 ../utils/Library-extra/Pfun  
 ../utils/Library-extra/Derivative-extra  
 ~/src/HOL/Library/Prefix-Order  
 ~/src/HOL/Library/Adhoc-Overloading  
 ~/src/HOL/Library/Monad-Syntax  
 ~/src/HOL/Library/Countable  
 ~/src/HOL/Eisbach/Eisbach  
 utp-parser-utils

**begin**

**no-notation** *inner* (**infix**  $\cdot$  70)

This theory describes the foundational structure of UTP variables, upon which the rest of our model rests. We start by defining alphabets, which in this shallow model are simply represented as types, though by convention usually a record type where each field corresponds to a variable.

**type-synonym**  $'\alpha$  *alphabet* =  $'\alpha$

UTP variables carry two type parameters,  $'a$  that corresponds to the variable's type and  $'\alpha$  that corresponds to alphabet of which the variable is a type. There is thus a strong link between alphabets and variables in this model. Variables are characterized by two functions, *var-lookup* and *var-update*, that respectively lookup and update the variable's value in some alphabetised state space. These functions can readily be extracted from an Isabelle record type.

**type-synonym**  $('a, '\alpha)$  *uvar* =  $('a, '\alpha)$  *lens*

The *VAR* function is a syntactic translations that allows to retrieve a variable given its name, assuming the variable is a field in a record.

**abbreviation** *rec-put*  $f \equiv (\lambda \sigma u. f (\lambda -. u) \sigma)$

**syntax** *-VAR*  $:: id \Rightarrow ('a, 'r) \text{ uvar } (VAR -)$

**translations** *VAR*  $x \Rightarrow \langle \mid \text{ lens-get } = x, \text{ lens-put } = \text{CONST rec-put } (-\text{update-name } x) \mid \rangle$

**abbreviation** *var-lookup*  $:: ('a, 'a) \text{ uvar } \Rightarrow 'a \Rightarrow 'a$  **where**  
*var-lookup*  $\equiv \text{lens-get}$

**abbreviation** *var-assign*  $:: ('a, 'a) \text{ uvar } \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a)$  **where**  
*var-assign*  $x \ v \ \sigma \equiv \text{lens-put } x \ \sigma \ v$

**abbreviation** *var-update*  $:: ('a, 'a) \text{ uvar } \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$  **where**  
*var-update*  $\equiv \text{weak-lens.update}$

**abbreviation** *semi-uvar*  $\equiv \text{mwb-lens}$

**abbreviation** *uvar*  $\equiv \text{vwb-lens}$

We also 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 to a tuple alphabet.

**definition** *in-var*  $:: ('a, 'a) \text{ uvar } \Rightarrow ('a, 'a \times 'b) \text{ uvar}$  **where**  
*in-var*  $x = x ;_l \text{fst}_l$

**definition** *out-var*  $:: ('a, 'b) \text{ uvar } \Rightarrow ('a, 'a \times 'b) \text{ uvar}$  **where**  
*out-var*  $x = x ;_l \text{snd}_l$

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

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

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

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

**lemma** *in-out-indep* [simp]:  
*in-var*  $x \bowtie \text{out-var } y$   
**by** (simp add: lens-indep-def in-var-def out-var-def fst-lens-def snd-lens-def lens-comp-def)

**lemma** *out-in-indep* [simp]:  
*out-var*  $x \bowtie \text{in-var } y$   
**by** (simp add: lens-indep-def in-var-def out-var-def fst-lens-def snd-lens-def lens-comp-def)

**lemma** *in-var-indep* [simp]:  
 $x \bowtie y \implies \text{in-var } x \bowtie \text{in-var } y$

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

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

$x \bowtie y \implies \text{out-var } x \bowtie \text{out-var } y$

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

We also define some lookup abstraction simplifications.

**lemma** *var-lookup-in* [*simp*]: *lens-get* (*in-var* *x*) (*A*, *A'*) = *lens-get* *x* *A*

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

**lemma** *var-lookup-out* [*simp*]: *lens-get* (*out-var* *x*) (*A*, *A'*) = *lens-get* *x* *A'*

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

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

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

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

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

Variables can also be used to effectively define sets of variables. Here we define the the universal alphabet ( $\Sigma$ ) to be a variable with identity for both the lookup and update functions. Effectively this is just a function directly on the alphabet type.

**definition** *univ-alpha* :: ( $'\alpha$ ,  $'\alpha$ ) *uvar* ( $\Sigma$ ) **where**

*univ-alpha* = *id-lens*

The following operator attempts to combine two variables to produce a unified projection update pair. I hoped this could be used to define alphabet subsets by allowing a finite composition of variables. However, I don't think it works as the update function can't really be split into it's constituent parts if, e.g. the update of the first component depends on the second etc. You really want to update the two fields in parallel, but I don't think this is possible.

**definition** *uvar-comp* :: ( $'a$ ,  $'\alpha$ ) *uvar*  $\Rightarrow$  ( $'b$ ,  $'\alpha$ ) *uvar*  $\Rightarrow$  ( $'a \times 'b$ ,  $'\alpha$ ) *uvar* (**infix**  $\circ_v$  65) **where**

*uvar-comp* *x y* = *prod-lens* *x y*

**nonterminal** *svar*

**syntax**

*-svar* :: *id*  $\Rightarrow$  *svar* (- [999] 999)

*-spvar* :: *id*  $\Rightarrow$  *svar* (&- [999] 999)

*-sinvar* :: *id*  $\Rightarrow$  *svar* (\$- [999] 999)

*-soutvar* :: *id*  $\Rightarrow$  *svar* (\$-' [999] 999)

**consts**

*svar* ::  $'v \Rightarrow 'e$

*ivar* ::  $'v \Rightarrow 'e$

*ovar* ::  $'v \Rightarrow 'e$

**ad hoc-overloading**

*ivar in-var* **and** *ovar out-var*

**translations**

*-svar* *x*  $=>$  *x*

*-spvar* *x*  $=>$  *x*

*-sinvar* *x* == *CONST* *ivar* *x*

*-soutvar* *x* == *CONST* *ovar* *x*

end

## 1.1 Deep UTP variables

```
theory utp-dvar
  imports utp-var
begin
```

UTP variables represented by record fields are shallow, nameless entities. They are fundamentally static in nature, since a new record field can only be introduced definitionally and cannot be otherwise arbitrarily created. They are nevertheless very useful as proof automation is excellent, and they can fully make use of the Isabelle type system. However, for constructs like alphabet extension that can introduce new variables they are inadequate. As a result we also introduce a notion of deep variables to complement them. A deep variable is not a record field, but rather a key within a store map that records the values of all deep variables. As such the Isabelle type system is agnostic of them, and the creation of a new deep variable does not change the portion of the alphabet specified by the type system.

In order to create a type of stores (or bindings) for variables, we must fix a universe for the variable valuations. This is the major downside of deep variables – they cannot have any type, but only a type whose cardinality is up to  $\mathfrak{c}$ , the cardinality of the continuum. This is why we need both deep and shallow variables, as the latter are unrestricted in this respect. Each deep variable will therefore specify the cardinality of the type it possesses.

## 1.2 Cardinalities

We first fix a datatype representing all possible cardinalities for a deep variable. These include finite cardinalities,  $\aleph_0$  (countable), and  $\mathfrak{c}$  (uncountable up to the continuum).

```
datatype ucard = fin nat | aleph0 ( $\aleph_0$ ) | cont ( $\mathfrak{c}$ )
```

Our universe is simply the set of natural numbers; this is sufficient for all types up to cardinality  $\mathfrak{c}$ .

```
type-synonym uuniv = nat set
```

We introduce a function that gives the set of values within our universe of the given cardinality. Since a cardinality of 0 is no proper type, we use finite cardinality 0 to mean cardinality 1, 1 to mean 2 etc.

```
fun uuniv :: ucard  $\Rightarrow$  uuniv set ( $\mathcal{U}'(-)$ ) where
 $\mathcal{U}'(\text{fin } n) = \{\{x\} \mid x. x \leq n\} \mid$ 
 $\mathcal{U}'(\aleph_0) = \{\{x\} \mid x. \text{True}\} \mid$ 
 $\mathcal{U}'(\mathfrak{c}) = \text{UNIV}$ 
```

We also define the following function that gives the cardinality of a type within the *continuum* type class.

```
definition ucard-of :: 'a::continuum itself  $\Rightarrow$  ucard where
ucard-of x = (if (finite (UNIV :: 'a set))
  then fin(card(UNIV :: 'a set) - 1)
  else if (countable (UNIV :: 'a set))
    then  $\aleph_0$ 
  else  $\mathfrak{c}$ )
```

**syntax**

$-ucard :: type \Rightarrow ucard \ (UCARD('a))$

**translations**

$UCARD('a) == CONST \ ucard-of \ (TYPE('a))$

**lemma** *ucard-non-empty*:

$\mathcal{U}(x) \neq \{\}$   
**by** (*induct*  $x$ , *auto*)

**lemma** *ucard-of-finite* [*simp*]:

$finite \ (UNIV :: 'a::continuum \ set) \Longrightarrow UCARD('a) = fin(card(UNIV :: 'a \ set) - 1)$   
**by** (*simp* *add*: *ucard-of-def*)

**lemma** *ucard-of-countably-infinite* [*simp*]:

$\llbracket countable(UNIV :: 'a::continuum \ set); infinite(UNIV :: 'a \ set) \rrbracket \Longrightarrow UCARD('a) = \aleph_0$   
**by** (*simp* *add*: *ucard-of-def*)

**lemma** *ucard-of-uncountably-infinite* [*simp*]:

$uncountable \ (UNIV :: 'a \ set) \Longrightarrow UCARD('a :: continuum) = c$   
**apply** (*simp* *add*: *ucard-of-def*)  
**using** *countable-finite* **apply** *blast*

**done**

### 1.3 Injection functions

**definition** *uinject-finite* ::  $'a::finite \Rightarrow uuniv$  **where**

*uinject-finite*  $x = \{to\_nat\_fin \ x\}$

**definition** *uinject-aleph0* ::  $'a::\{countable, infinite\} \Rightarrow uuniv$  **where**

*uinject-aleph0*  $x = \{to\_nat\_bij \ x\}$

**definition** *uinject-continuum* ::  $'a::\{continuum, infinite\} \Rightarrow uuniv$  **where**

*uinject-continuum*  $x = to\_nat\_set\_bij \ x$

**definition** *uinject* ::  $'a::continuum \Rightarrow uuniv$  **where**

*uinject*  $x = (if \ (finite \ (UNIV :: 'a \ set))$   
      $then \ \{to\_nat\_fin \ x\}$   
      $else \ if \ (countable \ (UNIV :: 'a \ set))$   
      $then \ \{to\_nat\_on \ (UNIV :: 'a \ set) \ x\}$   
      $else \ to\_nat\_set \ x)$

**definition** *uproject* ::  $uuniv \Rightarrow 'a::continuum$  **where**

*uproject* = *inv uinject*

**lemma** *uinject-finite*:

$finite \ (UNIV :: 'a::continuum \ set) \Longrightarrow uinject = (\lambda \ x :: 'a. \ \{to\_nat\_fin \ x\})$   
**by** (*rule* *ext*, *auto* *simp* *add*: *uinject-def*)

**lemma** *uinject-uncountable*:

$uncountable \ (UNIV :: 'a::continuum \ set) \Longrightarrow (uinject :: 'a \Rightarrow uuniv) = to\_nat\_set$   
**by** (*rule* *ext*, *auto* *simp* *add*: *uinject-def* *countable-finite*)

**lemma** *card-finite-lemma*:

**assumes** *finite* ( $UNIV :: 'a \ set$ )  
**shows**  $x < card \ (UNIV :: 'a \ set) \longleftrightarrow x \leq card \ (UNIV :: 'a \ set) - Suc \ 0$

```

proof –
  have card (UNIV :: 'a set) > 0
    by (simp add: assms finite-UNIV-card-ge-0)
  thus ?thesis
    by linarith
qed

```

This is a key theorem that shows that the injection function provides a bijection between any continuum type and the subuniverse of types with a matching cardinality.

```

lemma uinject-bij:
  bij-betw (uinject :: 'a::continuum  $\Rightarrow$  uuniv) UNIV  $\mathcal{U}$ (UCARD('a))
proof (cases finite (UNIV :: 'a set))
  case True thus ?thesis
    apply (auto simp add: uinject-def bij-betw-def inj-on-def image-def card-finite-lemma[THEN sym])
    apply (auto simp add: inj-eq to-nat-fin-inj to-nat-fin-bounded)
    using to-nat-fin-ex apply blast
  done
  next
  case False note infinite = this thus ?thesis
proof (cases countable (UNIV :: 'a set))
  case True thus ?thesis
    apply (auto simp add: uinject-def bij-betw-def inj-on-def infinite image-def card-finite-lemma[THEN sym])
    apply (meson image-to-nat-on infinite surj-def)
  done
  next
  case False note uncount = this thus ?thesis
    apply (simp add: uinject-uncountable)
    using to-nat-set-bij apply blast
  done
qed
qed

```

```

lemma uinject-card [simp]: uinject (x :: 'a::continuum)  $\in \mathcal{U}$ (UCARD('a))
  by (metis bij-betw-def rangeI uinject-bij)

```

```

lemma uinject-inv [simp]:
  uproject (uinject x) = x
  by (metis UNIV-I bij-betw-def inv-into-f-f uinject-bij uproject-def)

```

```

lemma uproject-inv [simp]:
   $x \in \mathcal{U}(\text{UCARD}('a::\text{continuum})) \implies \text{uinject } ((\text{uproject} :: \text{nat set} \Rightarrow 'a) \ x) = x$ 
  by (metis bij-betw-inv-into-right uinject-bij uproject-def)

```

## 1.4 Deep variables

A deep variable name stores both a name and the cardinality of the type it points to

```

record dname =
  dname-name :: string
  dname-card :: ucard

```

A *vstore* is a function mapping deep variable names to corresponding values in the universe, such that the deep variables specified cardinality is matched by the value it points to.

```

typedef vstore = {f :: dname  $\Rightarrow$  uuniv.  $\forall$  x. f(x)  $\in \mathcal{U}(\text{dname-card } x)$ }

```

```

  apply (rule-tac x = λ x. {0} in exI)
  apply (auto)
  apply (rename-tac x)
  apply (case-tac dname-card x)
  apply (simp-all)
done

```

setup-lifting *type-definition-vstore*

```

typedef ('a::continuum) dvar = {x :: dname. dname-card x = UCARD('a)}
  by (auto, meson dname.select-convs(2))

```

setup-lifting *type-definition-dvar*

```

lift-definition mk-dvar :: string ⇒ ('a::continuum) dvar ([-]d)
is λ n. (| dname-name = n, dname-card = UCARD('a) |)
  by auto

```

lift-definition *dvar-name* :: 'a::continuum dvar ⇒ string is *dname-name* .

lift-definition *dvar-card* :: 'a::continuum dvar ⇒ ucard is *dname-card* .

```

lemma dvar-name [simp]: dvar-name [x]d = x
  by (transfer, simp)

```

```

lift-definition vstore-lookup :: ('a::continuum) dvar ⇒ vstore ⇒ 'a
is λ x s. (uproject :: uuniv ⇒ 'a) (s(x)) .

```

```

lift-definition vstore-put :: ('a::continuum) dvar ⇒ 'a ⇒ vstore ⇒ vstore
is λ (x :: dname) (v :: 'a) f . f(x := uinject v)
  by (auto)

```

```

definition vstore-upd :: ('a::continuum) dvar ⇒ ('a ⇒ 'a) ⇒ vstore ⇒ vstore
where vstore-upd x f s = vstore-put x (f (vstore-lookup x s)) s

```

```

lemma vstore-upd-comp [simp]:
  vstore-upd x f (vstore-upd x g s) = vstore-upd x (f ∘ g) s
  by (simp add: vstore-upd-def, transfer, simp)

```

```

lemma vstore-lookup-put [simp]: vstore-lookup x (vstore-put x v s) = v
  by (transfer, simp)

```

```

lemma vstore-lookup-upd [simp]: vstore-lookup x (vstore-upd x f s) = f (vstore-lookup x s)
  by (simp add: vstore-upd-def)

```

```

lemma vstore-upd-eta [simp]: vstore-upd x (λ -. vstore-lookup x s) s = s
  apply (simp add: vstore-upd-def, transfer, auto)
  apply (metis Domainp-iff dvar.domain fun-upd-idem-iff uproject-inv)
done

```

```

lemma vstore-lookup-put-diff-var [simp]:
  assumes dvar-name x ≠ dvar-name y
  shows vstore-lookup x (vstore-put y v s) = vstore-lookup x s
  using assms by (transfer, auto)

```

lemma *vstore-put-commute*:



```

assumes dvar-name  $x \neq \textit{dvar-name } y$ 
shows  $\textit{vstore-put } x \ u \ (\textit{vstore-put } y \ v \ s) = \textit{vstore-put } y \ v \ (\textit{vstore-put } x \ u \ s)$ 
using assms
by (transfer, fastforce)

```

```

lemma vstore-put-put [simp]:
   $\textit{vstore-put } x \ u \ (\textit{vstore-put } x \ v \ s) = \textit{vstore-put } x \ u \ s$ 
by (transfer, simp)

```

The *vst* class provides an interface for extracting a variable store from a state space. For now, the state-space is limited to countably infinite types, though we will in the future build a more expressive universe.

```

class vst =
  fixes get-vstore :: 'a  $\Rightarrow$  vstore
  and put-vstore :: 'a  $\Rightarrow$  vstore  $\Rightarrow$  'a
  assumes put-get-vstore [simp]:  $\textit{get-vstore } (\textit{put-vstore } s \ x) = x$ 
  and get-put-vstore [simp]:  $\textit{put-vstore } s \ (\textit{get-vstore } s) = s$ 
  and put-put-vstore [simp]:  $\textit{put-vstore } (\textit{put-vstore } s \ x) \ y = \textit{put-vstore } s \ y$ 

```

```

definition dvar-lift :: 'a::continuum dvar  $\Rightarrow$  ('a, 'α::vst) uvar (-↑ [999] 999)
where dvar-lift  $x = \langle \mid \textit{lens-get} = (\lambda \ v. \ \textit{vstore-lookup } x \ (\textit{get-vstore } v))$ 
  ,  $\textit{lens-put} = (\lambda \ s \ v. \ \textit{put-vstore } s \ (\textit{vstore-put } x \ v \ (\textit{get-vstore } s)))$ 
   $\mid \rangle$ 

```

```

definition [simp]:  $\textit{in-dvar } x = \textit{in-var } (x\uparrow)$ 
definition [simp]:  $\textit{out-dvar } x = \textit{out-var } (x\uparrow)$ 

```

#### ad hoc-overloading

```

ivar in-dvar and ovar out-dvar

```

```

lemma uvar-dvar:  $\textit{uvar } (x\uparrow)$ 
  apply (unfold-locales)
  apply (simp-all add: dvar-lift-def)
  apply (metis get-put-vstore vstore-upd-def vstore-upd-eta)
done

```

Deep variables with different names are independent

```

lemma dvar-indep-diff-name:
  assumes dvar-name  $x \neq \textit{dvar-name } y$ 
  shows  $x\uparrow \bowtie y\uparrow$ 
  by (simp add: assms dvar-lift-def lens-indep-def vstore-put-commute)

```

```

lemma dvar-indep-diff-name' [simp]:
   $x \neq y \Longrightarrow [x]_{d\uparrow} \bowtie [y]_{d\uparrow}$ 
  by (auto intro: dvar-indep-diff-name)

```

A basic record structure for vstores

```

record vstore-d =
  vstore :: vstore

```

```

instantiation vstore-d-ext :: (type) vst
begin
  definition [simp]:  $\textit{get-vstore-vstore-d-ext} = \textit{vstore}$ 
  definition [simp]:  $\textit{put-vstore-vstore-d-ext} = (\lambda \ x \ s. \ \textit{vstore-update } (\lambda \cdot. \ s) \ x)$ 
end

```

```

instance
  by (intro-classes, simp-all)
end

end

```

## 2 UTP expressions

```

theory utp-expr
imports
  utp-var
  utp-dvar
begin

```

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 to the expression's type. This general model will allow us to unify all constructions under one type. All definitions in the file are given using the *lifting* package.

Since we have two kinds of variable (deep and shallow) in the model, we will also need two versions of each construct that takes a variable. We make use of *ad hoc*-overloading to ensure the correct instance is automatically chosen, within the user noticing a difference.

```

typedef ('t, 'α) uexpr = UNIV :: ('α alphabet ⇒ 't) set ..

```

```

notation Rep-uexpr (⟦_⟧e)

```

```

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

```

```

named-theorems ueval

```

```

setup-lifting type-definition-uexpr

```

A variable expression corresponds to the lookup function of the variable.

```

lift-definition var :: ('t, 'α) uvar ⇒ ('t, 'α) uexpr is var-lookup .

```

```

declare [[coercion-enabled]]
declare [[coercion var]]

```

```

definition dvar-exp :: 't::continuum dvar ⇒ ('t, 'α::vst) uexpr
where dvar-exp x = var (dvar-lift x)

```

We can then define specific cases for input and output variables, that simply perform tuple lifting. We also have variants for deep variables.

```

definition iuvar :: ('t, 'α) uvar ⇒ ('t, 'α × 'β) uexpr
where iuvar x = var (in-var x)

```

```

definition ouvar :: ('t, 'β) uvar ⇒ ('t, 'α × 'β) uexpr
where ouvar x = var (out-var x)

```

```

definition idvar :: 't::continuum dvar ⇒ ('t, 'α::vst × 'β) uexpr
where idvar x = var (in-var (dvar-lift x))

```

**definition**  $odvar :: 't::continuum\ dvar \Rightarrow ('t, 'α \times 'β::vst)\ uexpr$   
**where**  $odvar\ x = var\ (out-var\ (dvar-lift\ x))$

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

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

We define lifting for unary, binary, and ternary functions, that simply apply the function to all possible results of the expressions.

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

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

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

We also define a UTP expression version of function abstract

**lift-definition**  $ulambda :: ('a \Rightarrow ('b, 'α)\ uexpr) \Rightarrow ('a \Rightarrow 'b, 'α)\ uexpr$   
**is**  $\lambda\ f\ A\ x.\ f\ x\ A\ .$

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

**consts**  
 $ulit :: 't \Rightarrow 'e\ (\ll\!-\!\gg)$   
 $ueq :: 'a \Rightarrow 'a \Rightarrow 'b\ (\text{infixl}\ =_u\ 50)$   
 $ueuvar :: 'v \Rightarrow 'p$   
 $uiuvar :: 'v \Rightarrow 'p$   
 $uouvar :: 'v \Rightarrow 'p$

**adhoc-overloading**

$ulit\ lit\ \text{and}$   
 $ueuvar\ var\ \text{and}$   
 $ueuvar\ dvar-exp\ \text{and}$   
 $uiuvar\ iuvar\ \text{and}$   
 $uiuvar\ idvar\ \text{and}$   
 $uouvar\ ouvar\ \text{and}$   
 $uouvar\ odvar$

**syntax**

$-uuvar :: ('t, 'α)\ uvar \Rightarrow logic\ (\&- [999]\ 999)$   
 $-uiuvar :: ('t, 'α)\ uvar \Rightarrow logic\ (\$- [999]\ 999)$   
 $-uouvar :: ('t, 'α)\ uvar \Rightarrow logic\ (\$-'\ [999]\ 999)$

**translations**

$\&x == CONST\ ueuvar\ x$   
 $\$x == CONST\ uiuvar\ x$   
 $\$x' == CONST\ uouvar\ x$

We also set up some useful standard arithmetic operators for Isabelle by lifting the functions to binary operators.

**instantiation**  $uexpr :: (plus, type)\ plus$   
**begin**

```

definition plus-uepr-def:  $u + v = \text{bop } (op \ +) \ u \ v$ 
instance ..
end

```

Instantiating uminus also provides negation for predicates later

```

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

```

```

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

```

```

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

```

```

instantiation uepr :: (inverse, type) inverse
begin
  definition inverse-uepr-def:  $\text{inverse } u = \text{uop } \text{inverse } u$ 
  definition divide-uepr-def:  $u / v = \text{bop } (op \ /) \ u \ v$ 
instance ..
end

```

```

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

```

```

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

```

```

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

```

```

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

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

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

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

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

Set up automation for numerals

lemma numeral-uexpr-rep-eq:  $\llbracket \text{numeral } x \rrbracket_e b = \text{numeral } x$ 
  by (induct x, simp-all add: plus-uexpr-def one-uexpr-def numeral.simps lit.rep-eq bop.rep-eq)

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

definition eq-upred :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr
where eq-upred x y = bop HOL.eq x y

adhoc-overloading
  ueq eq-upred

definition fun-apply f x = f x
declare fun-apply-def [simp]

consts
  uapply :: 'f ⇒ 'k ⇒ 'v
  udom    :: 'f ⇒ 'a set
  uran    :: 'f ⇒ 'b set
  ucard   :: 'f ⇒ nat

adhoc-overloading
  uapply fun-apply and uapply nth and uapply pfun-app and
  udom Domain and udom pdom and udom seq-dom and
  udom Range and uran pran and uran set and
  ucard card and ucard pcard and ucard length

nonterminal utuple-args and umaplet and umaplets

syntax
  -ucoerce    :: ('a, 'α) uexpr ⇒ type ⇒ ('a, 'α) uexpr (infix :u 50)
  -unil       :: ('a list, 'α) uexpr (⟨⟩)
  -ulist      :: args => ('a list, 'α) uexpr  ((⟨-⟩))
  -uappend    :: ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (infixr ^u 80)
  -ulast      :: ('a list, 'α) uexpr ⇒ ('a, 'α) uexpr (lastu'(-))
  -ufront     :: ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (frontu'(-))

```

`-uhead` :: ('a list, 'α) uexpr ⇒ ('a, 'α) uexpr (head<sub>u</sub>'(-))  
`-utail` :: ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (tail<sub>u</sub>'(-))  
`-ucard` :: ('a list, 'α) uexpr ⇒ (nat, 'α) uexpr (#<sub>u</sub>'(-))  
`-ufilter` :: ('a list, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a list, 'α) uexpr (**infixl** |<sub>u</sub> 75)  
`-uextract` :: ('a set, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (**infixl** |<sub>u</sub> 75)  
`-uelems` :: ('a list, 'α) uexpr ⇒ ('a set, 'α) uexpr (elems<sub>u</sub>'(-))  
`-usorted` :: ('a list, 'α) uexpr ⇒ (bool, 'α) uexpr (sorted<sub>u</sub>'(-))  
`-udistinct` :: ('a list, 'α) uexpr ⇒ (bool, 'α) uexpr (distinct<sub>u</sub>'(-))  
`-uless` :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** <<sub>u</sub> 50)  
`-uleq` :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** ≤<sub>u</sub> 50)  
`-ugreat` :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** ><sub>u</sub> 50)  
`-ugeq` :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** ≥<sub>u</sub> 50)  
`-uempset` :: ('a set, 'α) uexpr ({ }<sub>u</sub>)  
`-uset` :: args => ('a set, 'α) uexpr ({(-)}<sub>u</sub>)  
`-uunion` :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr (**infixl** ∪<sub>u</sub> 65)  
`-uinter` :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr (**infixl** ∩<sub>u</sub> 70)  
`-umem` :: ('a, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** ∈<sub>u</sub> 50)  
`-unmem` :: ('a, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** ∉<sub>u</sub> 50)  
`-usubset` :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** ⊆<sub>u</sub> 50)  
`-usubseteq` :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** ⊆<sub>u</sub> 50)  
`-utuple` :: ('a, 'α) uexpr ⇒ utuple-args ⇒ ('a \* 'b, 'α) uexpr ((1'(-, / -)<sub>u</sub>)  
`-utuple-arg` :: ('a, 'α) uexpr ⇒ utuple-args (-)  
`-utuple-args` :: ('a, 'α) uexpr => utuple-args ⇒ utuple-args (-, / -)  
`-uunit` :: ('a, 'α) uexpr ('()<sub>u</sub>)  
`-ufst` :: ('a × 'b, 'α) uexpr ⇒ ('a, 'α) uexpr (π<sub>1</sub>'(-))  
`-usnd` :: ('a × 'b, 'α) uexpr ⇒ ('b, 'α) uexpr (π<sub>2</sub>'(-))  
`-uapply` :: ('a ⇒ 'b, 'α) uexpr ⇒ utuple-args ⇒ ('b, 'α) uexpr (-|<sub>u</sub> [999,0] 999)  
`-ulambda` :: ptttrn ⇒ logic ⇒ logic (λ - · - [0, 10] 10)  
`-udom` :: logic ⇒ logic (dom<sub>u</sub>'(-))  
`-uran` :: logic ⇒ logic (ran<sub>u</sub>'(-))  
`-uinl` :: logic ⇒ logic (inl<sub>u</sub>'(-))  
`-uinr` :: logic ⇒ logic (inr<sub>u</sub>'(-))  
`-umap-empty` :: logic (|<sub>u</sub>)  
`-umap-plus` :: logic ⇒ logic ⇒ logic (**infixl** ⊕<sub>u</sub> 85)  
`-umap-minus` :: logic ⇒ logic ⇒ logic (**infixl** ⊖<sub>u</sub> 85)  
`-udom-res` :: logic ⇒ logic ⇒ logic (**infixl** ≲<sub>u</sub> 85)  
`-uran-res` :: logic ⇒ logic ⇒ logic (**infixl** ≳<sub>u</sub> 85)  
`-umaplet` :: [logic, logic] => umaplet (- /|> / -)  
:: umaplet => umaplets (-)  
`-UMaplets` :: [umaplet, umaplets] => umaplets (-, / -)  
`-UMapUpd` :: [logic, umaplets] => logic (-/'(-) [900,0] 900)  
`-UMap` :: umaplets => logic ((1[-]<sub>u</sub>))

## translations

$f(|v|)_u \leq \text{CONST } uapply \ f \ v$   
 $dom_u(f) \leq \text{CONST } udom \ f$   
 $ran_u(f) \leq \text{CONST } uran \ f$   
 $\#_u(f) \leq \text{CONST } ucard \ f$

## translations

$x :_u 'a == x :: ('a, -) uexpr$   
 $\langle \rangle == \llbracket \rrbracket$   
 $\langle x, xs \rangle == \text{CONST } bop \ (op \ \#) \ x \ \langle xs \rangle$   
 $\langle x \rangle == \text{CONST } bop \ (op \ \#) \ x \ \llbracket \rrbracket$

$x \hat{=}_u y == \text{CONST bop } (op \text{ @}) x y$   
 $\text{last}_u(xs) == \text{CONST uop CONST last } xs$   
 $\text{front}_u(xs) == \text{CONST uop CONST butlast } xs$   
 $\text{head}_u(xs) == \text{CONST uop CONST hd } xs$   
 $\text{tail}_u(xs) == \text{CONST uop CONST tl } xs$   
 $\#_u(xs) == \text{CONST uop CONST ucard } xs$   
 $\text{elems}_u(xs) == \text{CONST uop CONST set } xs$   
 $\text{sorted}_u(xs) == \text{CONST uop CONST sorted } xs$   
 $\text{distinct}_u(xs) == \text{CONST uop CONST distinct } xs$   
 $xs \downarrow_u A == \text{CONST bop CONST seq-filter } xs A$   
 $A \uparrow_u xs == \text{CONST bop } (op \downarrow_l) A xs$   
 $x <_u y == \text{CONST bop } (op <) x y$   
 $x \leq_u y == \text{CONST bop } (op \leq) x y$   
 $x >_u y == y <_u x$   
 $x \geq_u y == y \leq_u x$   
 $\{\}_u == \ll\{\}\gg$   
 $\{x, xs\}_u == \text{CONST bop } (\text{CONST insert}) x \{xs\}_u$   
 $\{x\}_u == \text{CONST bop } (\text{CONST insert}) x \ll\{\}\gg$   
 $A \cup_u B == \text{CONST bop } (op \cup) A B$   
 $A \cap_u B == \text{CONST bop } (op \cap) A B$   
 $f \oplus_u g => (f :: ((-, -) \text{ pfun}, -) \text{ uepr}) + g$   
 $f \ominus_u g => (f :: ((-, -) \text{ pfun}, -) \text{ uepr}) - g$   
 $x \in_u A == \text{CONST bop } (op \in) x A$   
 $x \notin_u A == \text{CONST bop } (op \notin) x A$   
 $A \subset_u B == \text{CONST bop } (op <) A B$   
 $A \subset_u B <= \text{CONST bop } (op \subset) A B$   
 $f \subset_u g <= \text{CONST bop } (op \subset_p) f g$   
 $A \subseteq_u B == \text{CONST bop } (op \leq) A B$   
 $A \subseteq_u B <= \text{CONST bop } (op \subseteq) A B$   
 $f \subseteq_u g <= \text{CONST bop } (op \subseteq_p) f g$   
 $()_u == \ll()\gg$   
 $(x, y)_u == \text{CONST bop } (\text{CONST Pair}) x y$   
 $\text{-utuple } x (\text{-utuple-args } y z) == \text{-utuple } x (\text{-utuple-arg } (\text{-utuple } y z))$   
 $\pi_1(x) == \text{CONST uop CONST fst } x$   
 $\pi_2(x) == \text{CONST uop CONST snd } x$   
 $f(\downarrow x)_u == \text{CONST bop CONST uapply } f x$   
 $\lambda x \cdot p == \text{CONST ulambda } (\lambda x. p)$   
 $\text{dom}_u(f) == \text{CONST uop CONST udom } f$   
 $\text{ran}_u(f) == \text{CONST uop CONST uran } f$   
 $\text{inl}_u(x) == \text{CONST uop CONST Inl } x$   
 $\text{inr}_u(x) == \text{CONST uop CONST Inr } x$   
 $\square_u == \ll\text{CONST pempty}\gg$   
 $A \triangleleft_u f == \text{CONST bop } (op \triangleleft_p) A f$   
 $f \triangleright_u A == \text{CONST bop } (op \triangleright_p) A f$   
 $\text{-UMapUpd } m (\text{-UMaplets } xy \text{ ms}) == \text{-UMapUpd } (\text{-UMapUpd } m xy) \text{ ms}$   
 $\text{-UMapUpd } m (\text{-umaplet } x y) == \text{CONST trop CONST pfun-upd } m x y$   
 $\text{-UMap } ms == \text{-UMapUpd } \square_u ms$   
 $\text{-UMap } (\text{-UMaplets } ms1 \text{ ms2}) <= \text{-UMapUpd } (\text{-UMap } ms1) \text{ ms2}$   
 $\text{-UMaplets } ms1 (\text{-UMaplets } ms2 \text{ ms3}) <= \text{-UMaplets } (\text{-UMaplets } ms1 \text{ ms2}) \text{ ms3}$   
 $f(\downarrow x, y)_u == \text{CONST bop CONST uapply } f (x, y)_u$

Lifting set intervals

**syntax**

$\text{-uset-atLeastAtMost} :: ('a, 'a) \text{ uepr} \Rightarrow ('a, 'a) \text{ uepr} \Rightarrow ('a \text{ set}, 'a) \text{ uepr } ((1\{-..\}_{u}))$   
 $\text{-uset-atLeastLessThan} :: ('a, 'a) \text{ uepr} \Rightarrow ('a, 'a) \text{ uepr} \Rightarrow ('a \text{ set}, 'a) \text{ uepr } ((1\{-..<\}_{u}))$

$-uset-compr :: id \Rightarrow ('a \text{ set}, 'α) \text{ ueexpr} \Rightarrow (bool, 'α) \text{ ueexpr} \Rightarrow ('b, 'α) \text{ ueexpr} \Rightarrow ('b \text{ set}, 'α) \text{ ueexpr} ((1\{- / - \mid - \cdot / -\}_u))$

**lift-definition** *ZedSetCompr* ::

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

**translations**

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

Lifting limits

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

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

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

**syntax**

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

**translations**

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

**lemmas** *ueexpr-defs* =

*iuvar-def*  
*ouvar-def*  
*zero-ueexpr-def*  
*one-ueexpr-def*  
*plus-ueexpr-def*  
*uminus-ueexpr-def*  
*minus-ueexpr-def*  
*times-ueexpr-def*  
*inverse-ueexpr-def*  
*divide-ueexpr-def*  
*div-ueexpr-def*  
*mod-ueexpr-def*  
*eq-upred-def*  
*numeral-ueexpr-simp*  
*ulim-left-def*  
*ulim-right-def*  
*ucont-on-def*

**lemma** *var-in-var*:  $\text{var } (\text{in-var } x) = \$x$

**by** (*simp add: iuvar-def*)

**lemma** *var-out-var*:  $\text{var } (\text{out-var } x) = \$x'$

**by** (*simp add: ouvar-def*)

## 2.1 Evaluation laws for expressions

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

**by** (*transfer, simp*)



```

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

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

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

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

declare uepr-defs [ueval]

end

```

### 3 Unrestriction

```

theory utp-unrest
  imports utp-expr
begin

```

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 variable  $x$ , written  $x \# p$ , if altering the value of  $x$  has no effect on the valuation of  $p$ . This is a sufficient notion to prove many laws that would ordinarily rely on an  $fv$  function.

```

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

syntax
  -unrest :: svar  $\Rightarrow$  logic  $\Rightarrow$  logic  $\Rightarrow$  logic (infix  $\#$  20)

translations
  -unrest x p == CONST unrest x p

```

```

named-theorems unrest

```

```

term var-update

```

```

lift-definition unrest-upred :: ('a, 'α) uvar  $\Rightarrow$  ('b, 'α) uepr  $\Rightarrow$  bool
is  $\lambda \ x \ e. \ \forall \ b \ v. \ e \ (\text{var-assign } x \ v \ b) = e \ b$  .

```

```

definition unrest-dvar-upred :: 'a::continuum dvar  $\Rightarrow$  ('b, 'α::vst) uepr  $\Rightarrow$  bool where
unrest-dvar-upred x P = unrest-upred (x↑) P

```

```

adhoc-overloading
  unrest unrest-upred

```

```

lemma unrest-lit [unrest]:  $x \# \llbracket v \rrbracket$ 
  by (transfer, simp)

```

The following law demonstrates why we need variable independence: a variable expression is unrestricted by another variable only when the two variables are independent.

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

**lemma** *unrest-iuvar* [*unrest*]:  $\llbracket \text{uvar } x; x \bowtie y \rrbracket \Longrightarrow \$y \# \$x$   
 by (*metis in-var-indep in-var-uvar unrest-var var-in-var*)

**lemma** *unrest-ouvar* [*unrest*]:  $\llbracket \text{uvar } x; x \bowtie y \rrbracket \Longrightarrow \$y' \# \$x'$   
 by (*metis out-var-indep out-var-uvar unrest-var var-out-var*)

**lemma** *unrest-iuvar-ouvar* [*unrest*]:  
 fixes  $x :: ('a, 'α) \text{uvar}$   
 assumes *uvar y*  
 shows  $\$x \# \$y'$   
 by (*metis prod.collapse unrest-upred.rep-eq var.rep-eq var-lookup-out var-out-var var-update-in*)

**lemma** *unrest-ouvar-iuvar* [*unrest*]:  
 fixes  $x :: ('a, 'α) \text{uvar}$   
 assumes *uvar y*  
 shows  $\$x' \# \$y$   
 by (*metis prod.collapse unrest-upred.rep-eq var.rep-eq var-in-var var-lookup-in var-update-out*)

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

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

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

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

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

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

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

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

**lemma** *unrest-divide* [*unrest*]:  $\llbracket x \# u; x \# v \rrbracket \Longrightarrow x \# u / v$

```

by (simp add: divide-ueexpr-def unrest)

end

```

## 4 Substitution

```

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

```

### 4.1 Substitution definitions

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  $\Rightarrow$  'a (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.

```

type-synonym 'α usubst = 'α alphabet  $\Rightarrow$  'α alphabet

```

```

lift-definition subst :: 'α usubst  $\Rightarrow$  ('a, 'α) ueexpr  $\Rightarrow$  ('a, 'α) ueexpr is
λ σ e b. e (σ b) .

```

```

adhoc-overloading
  usubst subst

```

Update the value of a variable to an expression in a substitution

```

consts subst-upd :: 'α usubst  $\Rightarrow$  'v  $\Rightarrow$  ('a, 'α) ueexpr  $\Rightarrow$  'α usubst

```

```

definition subst-upd-uvar :: 'α usubst  $\Rightarrow$  ('a, 'α) uvar  $\Rightarrow$  ('a, 'α) ueexpr  $\Rightarrow$  'α usubst where
subst-upd-uvar σ x v = (λ b. var-assign x ( $\llbracket v \rrbracket_e b$ ) (σ b))

```

```

definition subst-upd-dvar :: 'α usubst  $\Rightarrow$  'a::continuum dvar  $\Rightarrow$  ('a, 'α::vst) ueexpr  $\Rightarrow$  'α usubst where
subst-upd-dvar σ x v = subst-upd-uvar σ (x $\uparrow$ ) v

```

```

adhoc-overloading
  subst-upd subst-upd-uvar and subst-upd subst-upd-dvar

```

Lookup the expression associated with a variable in a substitution

```

lift-definition usubst-lookup :: 'α usubst  $\Rightarrow$  ('a, 'α) uvar  $\Rightarrow$  ('a, 'α) ueexpr ( $\langle - \rangle_s$ )
is λ σ x b. var-lookup x (σ b) .

```

Relational lifting of a substitution to the first element of the state space

```

definition usubst-rel-lift :: 'α usubst  $\Rightarrow$  ('α  $\times$  'β) usubst ( $\lceil - \rceil_s$ ) where
 $\lceil \sigma \rceil_s = (\lambda (A, A'). (\sigma A, A'))$ 

```

```

definition usubst-rel-drop :: ('α  $\times$  'α) usubst  $\Rightarrow$  'α usubst ( $\lfloor - \rfloor_s$ ) where

```

$$[\sigma]_s = (\lambda A. \text{fst } (\sigma (A, A)))$$

**nonterminal** *smaplet* and *smaplets*

**syntax**

```
-smaplet  :: [svar, 'a] => smaplet      (- /↦s/ -)
           :: smaplet => smaplets      (-)
-SMaplets :: [smaplet, smaplets] => smaplets (-./ -)
-SubstUpd :: ['m usubst, smaplets] => 'm usubst (-/'(-) [900,0] 900)
-Subst    :: smaplets => 'a ~=> 'b      ((1[-]))
```

**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
```

## 4.2 Substitution laws

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

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

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

**lemma** *usubst-lookup-upd* [*usubst*]:  
**assumes** *semi-uvar 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-upd-idem* [*usubst*]:  
**assumes** *semi-uvar 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-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**  $z \bowtie y$  **and** *semi-uvar x*  
**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** *usubst-upd-comm-dash* [*usubst*]:  
**fixes**  $x :: ('a, 'α) \text{uvar}$   
**shows**  $\sigma(\$x' \mapsto_s v, \$x \mapsto_s u) = \sigma(\$x \mapsto_s u, \$x' \mapsto_s v)$   
**using** *in-out-indep usubst-upd-comm* **by** *force*

**lemma** *usubst-lookup-upd-indep* [*usubst*]:

**assumes**  $uvar\ x\ x \bowtie y$   
**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-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** *id-subst* [*usubst*]:  $id \dagger v = v$   
**by** (*transfer, simp*)

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

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

**lemma** *subst-ivar* [*usubst*]:  $\sigma \dagger \$x = \langle \sigma \rangle_s (in-var\ x)$   
**by** (*simp add: iuvar-def, transfer, simp*)

**lemma** *subst-ovar* [*usubst*]:  $\sigma \dagger \$x' = \langle \sigma \rangle_s (out-var\ x)$   
**by** (*simp add: ouvar-def, transfer, simp*)

We add the symmetric definition of input and output variables to substitution laws so that the variables are correctly normalised after substitution.

**declare** *iuvar-def*[*THEN sym, usubst*]  
**declare** *ouvar-def*[*THEN sym, usubst*]

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

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

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

**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-uepr-def subst-bop*)

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

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

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

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

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

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

**lemma** *subst-lift-id* [*usubst*]:  $\lceil id \rceil_s = id$   
**by** (*simp add: usubst-rel-lift-def*)

**lemma** *subst-drop-id* [*usubst*]:  $\lfloor id \rfloor_s = id$   
**by** (*auto simp add: usubst-rel-drop-def*)

**lemma** *subst-lift-drop* [*usubst*]:  $\lfloor \lceil \sigma \rceil_s \rfloor_s = \sigma$   
**by** (*simp add: usubst-rel-lift-def usubst-rel-drop-def*)

**lemma** *subst-lift-upd* [*usubst*]:  
**fixes**  $x :: ('a, 'α) \text{ uvar}$   
**shows**  $\lceil \sigma(x \mapsto_s v) \rceil_s = \lceil \sigma \rceil_s (\$x \mapsto_s \lceil v \rceil_<)$   
**by** (*simp add: usubst-rel-lift-def subst-upd-uvar-def*, *transfer*, *auto*)

**lemma** *subst-drop-upd* [*usubst*]:  
**fixes**  $x :: ('a, 'α) \text{ uvar}$   
**shows**  $\lfloor \sigma(\$x \mapsto_s v) \rfloor_s = \lfloor \sigma \rfloor_s (x \mapsto_s \lfloor v \rfloor_<)$   
**apply** (*simp add: usubst-rel-drop-def subst-upd-uvar-def*, *transfer*, *rule ext*, *auto simp add: in-var-def*)  
**apply** (*metis fst-conv in-var-def prod.collapse var-update-in*)  
**done**

**nonterminal** *uexprs* **and** *svars*

**syntax**

-*psubst* ::  $['α \text{ usubst}, \text{ svars}, \text{ uexprs}] \Rightarrow \text{ logic}$   
-*subst* ::  $('a, 'α) \text{ uexpr} \Rightarrow \text{ uexprs} \Rightarrow \text{ svars} \Rightarrow ('a, 'α) \text{ uexpr} ((-[-/-]) [999,999] 1000)$   
-*uexprs* ::  $[( 'a, 'α) \text{ uexpr}, \text{ uexprs}] \Rightarrow \text{ uexprs} (-, / -)$   
::  $( 'a, 'α) \text{ uexpr} \Rightarrow \text{ uexprs} (-)$   
-*svars* ::  $[\text{ svar}, \text{ svars}] \Rightarrow \text{ svars} (-, / -)$   
::  $\text{ svar} \Rightarrow \text{ svars} (-)$

**translations**

-*subst*  $P \text{ es } vs \Rightarrow \text{ CONST } \text{subst} (-\text{psubst} (\text{CONST } id) \text{ vs } \text{ es}) P$   
-*psubst*  $m (-\text{svar } x) v \Rightarrow \text{ CONST } \text{subst-upd } m x v$   
-*psubst*  $m (-\text{spvar } x) v \Rightarrow \text{ CONST } \text{subst-upd } m x v$   
-*psubst*  $m (-\text{sinvar } x) v \Rightarrow \text{ CONST } \text{subst-upd } m (\text{CONST } \text{ivar } x) v$   
-*psubst*  $m (-\text{soutvar } x) v \Rightarrow \text{ CONST } \text{subst-upd } m (\text{CONST } \text{ovar } x) v$   
-*psubst*  $m (-\text{svars } x \text{ xs}) (-\text{uexprs } v \text{ vs}) \Rightarrow -\text{psubst} (-\text{psubst } m x v) \text{ xs } \text{ vs}$   
-*subst*  $P \text{ e } x \Leftarrow \text{ CONST } \text{subst} (\text{CONST } \text{subst-upd} (\text{CONST } id) x \text{ e}) P$

**end**

## 5 Lifting expressions

**theory** *utp-lift*

**imports**

*utp-expr*

*utp-unrest*  
**begin**

## 5.1 Lifting definitions

We define operators for converting an expression to and from a relational state space

**lift-definition** *lift-pre* ::  $('a, 'α) \text{ uexpr} \Rightarrow ('a, 'α \times 'β) \text{ uexpr} ([\cdot]_{<})$   
**is**  $\lambda p (A, A'). p A$  .

**lift-definition** *drop-pre* ::  $('a, 'α \times 'α) \text{ uexpr} \Rightarrow ('a, 'α) \text{ uexpr} ([\cdot]_{<})$   
**is**  $\lambda p A. p (A, A)$  .

**lift-definition** *lift-post* ::  $('a, 'β) \text{ uexpr} \Rightarrow ('a, 'α \times 'β) \text{ uexpr} ([\cdot]_{>})$   
**is**  $\lambda p (A, A'). p A'$  .

**abbreviation** *drop-post* ::  $('a, 'α \times 'α) \text{ uexpr} \Rightarrow ('a, 'α) \text{ uexpr} ([\cdot]_{>})$   
**where**  $[b]_{>} \equiv [b]_{<}$

**named-theorems** *ulift*

**method** *ulift-tac* = (*simp add: ulift*)?

## 5.2 Lifting laws

**lemma** *lift-pre-var* [*simp*]:  
 $[\text{var } x]_{<} = \$x$   
**by** (*simp add: iuvar-def, transfer, auto*)

**lemma** *lift-post-var* [*simp*]:  
 $[\text{var } x]_{>} = \$x'$   
**by** (*simp add: ouvar-def, transfer, auto*)

**lemma** *lift-pre-lit* [*simp*]:  
 $[\ll v \gg]_{<} = \ll v \gg$   
**by** (*transfer, auto*)

**lemma** *lift-post-lit* [*simp*]:  
 $[\ll v \gg]_{>} = \ll v \gg$   
**by** (*transfer, auto*)

**lemma** *lift-pre-uop* [*simp*]:  
 $[\text{uop } f \ v]_{<} = \text{uop } f \ [\text{v}]_{<}$   
**by** (*transfer, auto*)

**lemma** *lift-post-uop* [*simp*]:  
 $[\text{uop } f \ v]_{>} = \text{uop } f \ [\text{v}]_{>}$   
**by** (*transfer, auto*)

**lemma** *lift-pre-bop* [*simp*]:  
 $[\text{bop } f \ u \ v]_{<} = \text{bop } f \ [\text{u}]_{<} \ [\text{v}]_{<}$   
**by** (*transfer, auto*)

**lemma** *lift-post-bop* [*simp*]:  
 $[\text{bop } f \ u \ v]_{>} = \text{bop } f \ [\text{u}]_{>} \ [\text{v}]_{>}$   
**by** (*transfer, auto*)

```

lemma lift-pre-trop [simp]:
   $\lceil \text{trop } f \ u \ v \ w \rceil_{<} = \text{trop } f \ \lceil u \rceil_{<} \ \lceil v \rceil_{<} \ \lceil w \rceil_{<}$ 
by (transfer, auto)

```

```

lemma lift-post-trop [simp]:
   $\lceil \text{trop } f \ u \ v \ w \rceil_{>} = \text{trop } f \ \lceil u \rceil_{>} \ \lceil v \rceil_{>} \ \lceil w \rceil_{>}$ 
by (transfer, auto)

```

```

end

```

## 6 Alphabetised Predicates

```

theory utp-pred

```

```

imports

```

```

  utp-expr

```

```

  utp-subst

```

```

begin

```

An alphabetised predicate is simply a boolean valued expression

```

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

```

```

translations

```

```

  (type) 'α upred <= (type) (bool, 'α) ueexpr

```

```

named-theorems upred-defs

```

### 6.1 Predicate syntax

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.

```

no-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, 'α) uvar  $\Rightarrow$  'p  $\Rightarrow$  'p

```

```

  uall :: ('a, 'α) uvar  $\Rightarrow$  'p  $\Rightarrow$  'p

```

```

  ushEx :: ['a  $\Rightarrow$  'p]  $\Rightarrow$  'p

```

```

  ushAll :: ['a  $\Rightarrow$  'p]  $\Rightarrow$  'p

```

```

adhoc-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. 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).

#### **syntax**

*-uex* :: *svar*  $\Rightarrow$  *logic*  $\Rightarrow$  *logic* ( $\exists$  - - -  $[0, 10]$  10)  
*-uall* :: *svar*  $\Rightarrow$  *logic*  $\Rightarrow$  *logic* ( $\forall$  - - -  $[0, 10]$  10)  
*-ushEx* :: *idt*  $\Rightarrow$  *logic*  $\Rightarrow$  *logic* ( $\exists$  - - -  $[0, 10]$  10)  
*-ushAll* :: *idt*  $\Rightarrow$  *logic*  $\Rightarrow$  *logic* ( $\forall$  - - -  $[0, 10]$  10)  
*-ushBEx* :: *idt*  $\Rightarrow$  *logic*  $\Rightarrow$  *logic*  $\Rightarrow$  *logic* ( $\exists$  -  $\in$  - - -  $[0, 0, 10]$  10)  
*-ushBAll* :: *idt*  $\Rightarrow$  *logic*  $\Rightarrow$  *logic*  $\Rightarrow$  *logic* ( $\forall$  -  $\in$  - - -  $[0, 0, 10]$  10)

#### **translations**

$\exists \&x \cdot P \Rightarrow \text{CONST } uex \ x \ P$   
 $\exists \$x \cdot P \Rightarrow \text{CONST } uex \ (\text{CONST } in\text{-var } x) \ P$   
 $\exists \$x' \cdot P \Rightarrow \text{CONST } uex \ (\text{CONST } out\text{-var } x) \ P$   
 $\exists x \cdot P \Rightarrow \text{CONST } uex \ x \ P$   
 $\forall \&x \cdot P \Rightarrow \text{CONST } uall \ x \ P$   
 $\forall \$x \cdot P \Rightarrow \text{CONST } uall \ (\text{CONST } in\text{-var } x) \ P$   
 $\forall \$x' \cdot P \Rightarrow \text{CONST } uall \ (\text{CONST } out\text{-var } x) \ P$   
 $\forall x \cdot P \Rightarrow \text{CONST } uall \ x \ P$   
 $\exists x \cdot P \Rightarrow \text{CONST } ushEx \ (\lambda x. P)$   
 $\exists x \in A \cdot P \Rightarrow \exists x \cdot \langle x \rangle \in_u A \wedge P$   
 $\forall x \cdot P \Rightarrow \text{CONST } ushAll \ (\lambda x. P)$   
 $\forall x \in A \cdot P \Rightarrow \forall x \cdot \langle x \rangle \in_u A \Rightarrow P$

## **6.2 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*  $\equiv$  *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.

**notation** *inf* (**infixl**  $\sqcap$  70)

**notation** *sup* (**infixl**  $\sqcup$  65)

**notation** *Inf* ( $\bigsqcap$  -  $[900]$  900)

**notation** *Sup* ( $\bigsqcup$  -  $[900]$  900)

**notation** *bot* ( $\top$ )

**notation** *top* ( $\perp$ )

We now introduce a partial order on expressions. Note this is more general than refinement since it lifts an order on any expression type (not just Boolean). However, the Boolean version does equate to refinement.

```

instantiation uexpr :: (order, type) order
begin
  lift-definition less-eq-uexpr :: ('a, 'b) uexpr  $\Rightarrow$  ('a, 'b) uexpr  $\Rightarrow$  bool
  is  $\lambda P Q. (\forall A. P A \leq Q A)$  .
  definition less-uexpr :: ('a, 'b) uexpr  $\Rightarrow$  ('a, 'b) uexpr  $\Rightarrow$  bool
  where less-uexpr P Q = (P  $\leq$  Q  $\wedge$   $\neg$  Q  $\leq$  P)
instance proof
  fix x y z :: ('a, 'b) uexpr
  show (x < y) = (x  $\leq$  y  $\wedge$   $\neg$  y  $\leq$  x) by (simp add: less-uexpr-def)
  show x  $\leq$  x by (transfer, auto)
  show x  $\leq$  y  $\Rightarrow$  y  $\leq$  z  $\Rightarrow$  x  $\leq$  z
    by (transfer, blast intro: order.trans)
  show x  $\leq$  y  $\Rightarrow$  y  $\leq$  x  $\Rightarrow$  x = y
    by (transfer, rule ext, simp add: eq-iff)
qed
end

```

We also trivially instantiate our refinement class

```

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

```

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  $\lambda P Q A. \text{sup } (P A) (Q A)$  .
  lift-definition inf-uexpr :: ('a, 'b) uexpr  $\Rightarrow$  ('a, 'b) uexpr  $\Rightarrow$  ('a, 'b) uexpr
  is  $\lambda P Q A. \text{inf } (P A) (Q A)$  .
instance
  by (intro-classes) (transfer, auto)+
end

```

```

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

```

Finally we show that predicates form a Boolean algebra (under the lattice operators).

```

instance uexpr :: (boolean-algebra, type) boolean-algebra
  by (intro-classes, simp-all add: uexpr-defs)
    (transfer, simp add: sup-inf-distrib1 inf-compl-bot sup-compl-top diff-eq)+

```

```

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

```

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

```

definition true-upred = (top :: 'α upred)
definition false-upred = (bot :: 'α upred)
definition conj-upred = (inf :: 'α upred ⇒ 'α upred ⇒ 'α upred)
definition disj-upred = (sup :: 'α upred ⇒ 'α upred ⇒ 'α upred)
definition not-upred = (uminus :: 'α upred ⇒ 'α upred)
definition diff-upred = (minus :: 'α upred ⇒ 'α upred ⇒ 'α upred)

```

We also define the other predicate operators

```

lift-definition impl::'α upred ⇒ 'α upred ⇒ 'α upred is
λ P Q A. P A ⟶ Q A .

```

```

lift-definition iff-upred ::'α upred ⇒ 'α upred ⇒ 'α upred is
λ P Q A. P A ⟷ Q A .

```

```

lift-definition ex :: ('a, 'α) uvar ⇒ 'α upred ⇒ 'α upred is
λ x P b. (∃ v. P(var-assign x v b)) .

```

```

lift-definition shEx ::['β ⇒ 'α upred] ⇒ 'α upred is
λ P A. ∃ x. (P x) A .

```

```

lift-definition all :: ('a, 'α) uvar ⇒ 'α upred ⇒ 'α upred is
λ x P b. (∀ v. P(var-assign x v b)) .

```

```

lift-definition shAll ::['β ⇒ 'α upred] ⇒ 'α upred is
λ P A. ∀ x. (P x) 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::'α upred ⇒ 'α upred ([·]u) is
λ P A. ∀ A'. P A' .

```

```

lift-definition taut :: 'α upred ⇒ bool ('-)
is λ P. ∀ A. P A .

```

**ad hoc-overloading**

```

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

```

### 6.3 Proof support

We set up a simple tactic with the help of *Eisbach* that applies predicate definitions, applies the transfer method to drop down to the core definitions, applies extensionality (to remove the resulting lambda term) and the applies auto. This simple tactic will suffice to prove most of the standard laws.

```
method pred-tac = ((simp only: upred-defs)? ; (transfer, (rule-tac ext)?, auto simp add: fun-eq-iff)?)
```

```
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 subst-upd-dvar-def [upred-defs]
declare uexpr-defs [upred-defs]
declare usubst-rel-lift-def [upred-defs]
declare usubst-rel-drop-def [upred-defs]
```

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

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

### 6.4 Unrestriction Laws

```
lemma unrest-true [unrest]: x # true
  by (pred-tac)
```

```
lemma unrest-false [unrest]: x # false
  by (pred-tac)
```

```
lemma unrest-conj [unrest]: [x # P; x # Q] ==> x # P ∧ Q
  by (pred-tac)
```

```
lemma unrest-disj [unrest]: [x # P; x # Q] ==> x # P ∨ Q
  by (pred-tac)
```

```
lemma unrest-impl [unrest]: [x # P; x # Q] ==> x # P ⇒ Q
  by (pred-tac)
```

```
lemma unrest-iff [unrest]: [x # P; x # Q] ==> x # P ⇔ Q
  by (pred-tac)
```

```
lemma unrest-not [unrest]: x # P ==> x # (¬ P)
  by (pred-tac)
```

```
lemma unrest-ex-same [unrest]:
  uvar x ==> x # (∃ x • P)
  by pred-tac
```

```
lemma unrest-ex-diff [unrest]:
  assumes x ⋈ y y # P
```

**shows**  $y \# (\exists x \cdot P)$   
**using** *assms*  
**by** (*pred-tac*, *auto simp add: lens-indep-def*)

**lemma** *unrest-all-same* [*unrest*]:  
 $uvar\ x \implies x \# (\forall x \cdot P)$   
**by** *pred-tac*

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

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

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

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

## 6.5 Substitution Laws

**lemma** *subst-true* [*usubst*]:  $\sigma \dagger true = true$   
**by** (*pred-tac*)

**lemma** *subst-false* [*usubst*]:  $\sigma \dagger false = false$   
**by** (*pred-tac*)

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

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

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

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

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

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

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

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

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

**lemma** *subst-ex-same* [*usubst*]:  
 assumes *uvar x*  
 shows  $(\exists x \cdot P)[v/x] = (\exists x \cdot P)$   
 by (*simp add: assms id-subst subst-unrest unrest-ex-same*)

**lemma** *subst-ex-indep* [*usubst*]:  
 assumes  $x \bowtie y \ y \nmid v$   
 shows  $(\exists y \cdot P)[v/x] = (\exists y \cdot P[v/x])$   
 using *assms*  
 by (*pred-tac, auto simp add: lens-indep-def*)

**lemma** *subst-all-same* [*usubst*]:  
 assumes *uvar x*  
 shows  $(\forall x \cdot P)[v/x] = (\forall x \cdot P)$   
 by (*simp add: assms id-subst subst-unrest unrest-all-same*)

**lemma** *subst-all-indep* [*usubst*]:  
 assumes  $x \bowtie y \ y \nmid v$   
 shows  $(\forall y \cdot P)[v/x] = (\forall y \cdot P[v/x])$   
 using *assms*  
 by (*pred-tac, auto simp add: lens-indep-def*)

## 6.6 Predicate Laws

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**lemma** *true-not-false* [simp]:  $\text{true} \neq \text{false} \text{ false} \neq \text{true}$   
 by *pred-tac*+

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

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

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

**lemma** *impl-alt-def*:  $(P \Rightarrow Q) = (\neg P \vee Q)$   
 by *pred-tac*

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

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

**lemma** *conj-eq-in-var-subst*:  
 fixes  $x :: ('a, 'α) \text{ uvar}$   
 assumes  $\text{uvar } x$   
 shows  $(P \wedge \$x =_u v) = (P \llbracket v / \$x \rrbracket \wedge \$x =_u v)$   
 using *assms*  
 by (*pred-tac*, (*metis vwb-lens-wb wb-lens.get-put*))+)

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

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

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

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

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

**lemma** *one-point*:



**assumes**  $uvar\ x\ x \# v$   
**shows**  $(\exists\ x \cdot (P \wedge (var\ x =_u v))) = P[v/x]$   
**using** *assms*  
**by** (*simp add: upred-defs, transfer, auto*)

**lemma** *uvar-assign-exists*:  
 $uvar\ x \implies \exists\ v. b = var\text{-}assign\ x\ v\ b$   
**by** (*rule-tac x=var-lookup x b in exI, simp*)

**lemma** *uvar-obtain-assign*:  
**assumes**  $uvar\ x$   
**obtains**  $v$  **where**  $b = var\text{-}assign\ x\ v\ b$   
**using** *assms*  
**by** (*drule-tac uvar-assign-exists[of - b], auto*)

**lemma** *taut-split-subst*:  
**assumes**  $uvar\ x$   
**shows**  $\langle P \rangle \longleftrightarrow (\forall\ v. \langle P[v/x] \rangle)$   
**using** *assms*  
**by** (*pred-tac, metis uvar-assign-exists*)

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

**lemma** *subst-bool-split*:  
**assumes**  $uvar\ x$   
**shows**  $\langle P \rangle = \langle (P[valse/x] \wedge P[true/x]) \rangle$   
**proof** –  
**from** *assms* **have**  $\langle P \rangle = (\forall\ v. \langle P[v/x] \rangle)$   
**by** (*subst taut-split-subst[of x], auto*)  
**also have**  $\dots = (\langle P[True/x] \rangle \wedge \langle P[False/x] \rangle)$   
**by** (*metis (mono-tags, lifting)*)  
**also have**  $\dots = \langle (P[valse/x] \wedge P[true/x]) \rangle$   
**by** (*pred-tac*)  
**finally show** *?thesis* .  
**qed**

**lemma** *taut-iff-eq*:  
 $\langle P \Leftrightarrow Q \rangle \longleftrightarrow (P = Q)$   
**by** *pred-tac*

**lemma** *subst-eq-replace*:  
**fixes**  $x :: ('a, 'a) uvar$   
**shows**  $(p[u/x] \wedge u =_u v) = (p[v/x] \wedge u =_u v)$   
**by** *pred-tac*

**lemma** *exists-twice*:  $uvar\ x \implies (\exists\ x \cdot \exists\ x \cdot P) = (\exists\ x \cdot P)$   
**by** (*pred-tac*)

**lemma** *all-twice*:  $uvar\ x \implies (\forall\ x \cdot \forall\ x \cdot P) = (\forall\ x \cdot P)$   
**by** (*pred-tac*)

**lemma** *ex-commute*:  
**assumes**  $x \bowtie y$   
**shows**  $(\exists x \cdot \exists y \cdot P) = (\exists y \cdot \exists x \cdot P)$   
**using** *assms*  
**by** (*pred-tac, auto simp add: lens-indep-def*)

**lemma** *all-commute*:  
**assumes**  $x \bowtie y$   
**shows**  $(\forall x \cdot \forall y \cdot P) = (\forall y \cdot \forall x \cdot P)$   
**using** *assms*  
**by** (*pred-tac, auto simp add: lens-indep-def*)

## 6.7 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-tac*

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

**end**

## 7 Alphabetised relations

**theory** *utp-rel*

**imports**

*utp-pred*

**begin**

**default-sort** *type*

**named-theorems** *urel-defs*

**consts**

*useq*  $:: 'a \Rightarrow 'b \Rightarrow 'c$  (**infixr** ;; 15)

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

**definition** *in $\alpha$*   $:: ('a, 'a \times 'b) \text{ wvar}$  **where**  
 $\text{in}\alpha = \llbracket \text{lens-get} = \text{fst}, \text{lens-put} = \lambda (A, A') v. (v, A') \rrbracket$

**definition** *out $\alpha$*   $:: ('b, 'a \times 'b) \text{ wvar}$  **where**  
 $\text{out}\alpha = \llbracket \text{lens-get} = \text{snd}, \text{lens-put} = \lambda (A, A') v. (A, v) \rrbracket$

**declare** *in $\alpha$ -def* [*urel-defs*]

**declare** *out $\alpha$ -def* [*urel-defs*]

**lemma** *alpha-in-out*:

$\Sigma = \text{in}\alpha \circ_v \text{out}\alpha$

**by** (*auto simp add: in $\alpha$ -def out $\alpha$ -def univ-alpha-def id-lens-def wvar-comp-def prod-lens-def*)

**type-synonym** *'a condition*  $= 'a \text{ upred}$

**type-synonym**  $(\alpha, \beta) \text{ relation} = (\alpha \times \beta) \text{ upred}$   
**type-synonym**  $\alpha \text{ hrelation} = (\alpha \times \alpha) \text{ upred}$

**definition**  $\text{cond}::(\alpha, \beta) \text{ relation} \Rightarrow (\alpha, \beta) \text{ relation} \Rightarrow (\alpha, \beta) \text{ relation} \Rightarrow (\alpha, \beta) \text{ relation}$   
 $((\beta \triangleleft - \triangleright / -) [14,0,15] 14)$

**where**  $(P \triangleleft b \triangleright Q) \equiv (b \wedge P) \vee ((\neg b) \wedge Q)$

**abbreviation**  $\text{rcond}::(\alpha, \beta) \text{ relation} \Rightarrow \alpha \text{ condition} \Rightarrow (\alpha, \beta) \text{ relation} \Rightarrow (\alpha, \beta) \text{ relation}$   
 $((\beta \triangleleft - \triangleright_r / -) [14,0,15] 14)$

**where**  $(P \triangleleft b \triangleright_r Q) \equiv (P \triangleleft [b]_{<} \triangleright Q)$

**lift-definition**  $\text{seqr}::((\alpha \times \beta) \text{ upred}) \Rightarrow ((\beta \times \gamma) \text{ upred}) \Rightarrow (\alpha \times \gamma) \text{ upred}$   
**is**  $\lambda P Q r. r : (\{p. P p\} O \{q. Q q\})$  .

**lift-definition**  $\text{conv-r}::(\alpha, \alpha \times \beta) \text{ uexpr} \Rightarrow (\alpha, \beta \times \alpha) \text{ uexpr} (- [999] 999)$   
**is**  $\lambda e (b1, b2). e (b2, b1)$  .

**lift-definition**  $\text{assigns-r}::\alpha \text{ usubst} \Rightarrow \alpha \text{ hrelation} (\langle - \rangle_a)$   
**is**  $\lambda \sigma (A, A'). A' = \sigma(A)$  .

**definition**  $\text{skip-r}::\alpha \text{ hrelation}$  **where**  
 $\text{skip-r} = \text{assigns-r id}$

**abbreviation**  $\text{assign-r}::(\alpha, \alpha) \text{ uvar} \Rightarrow (\alpha, \alpha) \text{ uexpr} \Rightarrow \alpha \text{ hrelation}$   
**where**  $\text{assign-r } x v \equiv \text{assigns-r } [x \mapsto_s v]$

**abbreviation**  $\text{assign-2-r}::$   
 $(\alpha1, \alpha) \text{ uvar} \Rightarrow (\alpha2, \alpha) \text{ uvar} \Rightarrow (\alpha1, \alpha) \text{ uexpr} \Rightarrow (\alpha2, \alpha) \text{ uexpr} \Rightarrow \alpha \text{ hrelation}$   
**where**  $\text{assign-2-r } x y u v \equiv \text{assigns-r } [x \mapsto_s u, y \mapsto_s v]$

**nonterminal**

$\text{id-list}$  **and**  $\text{uexpr-list}$

**syntax**

$\text{-id-unit}::\text{id} \Rightarrow \text{id-list } (-)$   
 $\text{-id-list}::\text{id} \Rightarrow \text{id-list} \Rightarrow \text{id-list } (-, / -)$   
 $\text{-uexpr-unit}::(\alpha, \alpha) \text{ uexpr} \Rightarrow \text{uexpr-list } (- [40] 40)$   
 $\text{-uexpr-list}::(\alpha, \alpha) \text{ uexpr} \Rightarrow \text{uexpr-list} \Rightarrow \text{uexpr-list } (-, / - [40,40] 40)$   
 $\text{-assignment}::\text{svars} \Rightarrow \text{uexprs} \Rightarrow \alpha \text{ hrelation } (\text{infixr} := 35)$   
 $\text{-mk-usubst}::\text{svars} \Rightarrow \text{uexpr-list} \Rightarrow \alpha \text{ usubst}$

**translations**

$\text{-mk-usubst } (-\text{svar } x) (-\text{uexpr-unit } v) == [x \mapsto_s v]$   
 $\text{-mk-usubst } (-\text{id-list } xs) (-\text{uexpr-list } v vs) == (\text{-mk-usubst } xs vs)(x \mapsto_s v)$   
 $\text{-assignment } xs vs => \text{CONST assigns-r } (-\text{psubst } (\text{CONST id}) xs vs)$   
 $x := v <= \text{CONST assign-r } x v$   
 $x, y := u, v <= \text{CONST assign-2-r } x y u v$

**ad hoc-overloading**

$\text{useq seqr}$  **and**  
 $\text{uskip skip-r}$

**method**  $\text{rel-tac} = ((\text{simp add: upred-defs urel-defs})?, (\text{transfer, (rule-tac ext)})?, \text{auto simp add: urel-defs relcomp-unfold fun-eq-iff})?$

A test is like a precondition, except that it identifies to the postcondition. It forms the basis

for Kleene Algebra with Tests (KAT).

**definition**  $\text{lift-test} :: 'a \text{ condition} \Rightarrow 'a \text{ hrelation} \ (\llbracket - \rrbracket_t)$   
**where**  $\llbracket b \rrbracket_t = (\llbracket b \rrbracket_{<} \wedge II)$

**declare**  $\text{cond-def} \ [\text{urel-defs}]$   
**declare**  $\text{skip-r-def} \ [\text{urel-defs}]$

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

**definition**  $\text{rel-var-res} :: 'a \text{ hrelation} \Rightarrow ('a, 'a) \text{ uvar} \Rightarrow 'a \text{ hrelation} \ (\text{infix } \downarrow_\alpha \ 80) \text{ where}$   
 $P \downarrow_\alpha x = (\exists \$x \cdot \exists \$x' \cdot P)$

**declare**  $\text{rel-var-res-def} \ [\text{urel-defs}]$

## 7.1 Unrestriction Laws

**lemma**  $\text{unrest-iuvar} \ [\text{unrest}]: \text{uvar } x \Longrightarrow \text{out}\alpha \ \# \ \$x$   
**by**  $(\text{simp add: out}\alpha\text{-def iuvar-def, transfer, auto})$

**lemma**  $\text{unrest-ouvar} \ [\text{unrest}]: \text{uvar } x \Longrightarrow \text{in}\alpha \ \# \ \$x'$   
**by**  $(\text{simp add: in}\alpha\text{-def ouvar-def, transfer, auto})$

**lemma**  $\text{unrest-in}\alpha\text{-var} \ [\text{unrest}]:$   
 $\llbracket \text{uvar } x; \text{in}\alpha \ \# \ P \rrbracket \Longrightarrow \$x \ \# \ P$   
**by**  $(\text{pred-tac, simp add: in}\alpha\text{-def})$

**lemma**  $\text{unrest-out}\alpha\text{-var} \ [\text{unrest}]:$   
 $\llbracket \text{uvar } x; \text{out}\alpha \ \# \ P \rrbracket \Longrightarrow \$x' \ \# \ P$   
**by**  $(\text{pred-tac, simp add: out}\alpha\text{-def})$

**lemma**  $\text{in}\alpha\text{-uvar} \ [\text{simp}]: \text{uvar } \text{in}\alpha$   
**by**  $(\text{unfold-locales, auto simp add: in}\alpha\text{-def})$

**lemma**  $\text{out}\alpha\text{-uvar} \ [\text{simp}]: \text{uvar } \text{out}\alpha$   
**by**  $(\text{unfold-locales, auto simp add: out}\alpha\text{-def})$

**lemma**  $\text{unrest-pre-out}\alpha \ [\text{unrest}]: \text{out}\alpha \ \# \ \llbracket b \rrbracket_{<}$   
**by**  $(\text{transfer, auto simp add: out}\alpha\text{-def})$

**lemma**  $\text{unrest-post-in}\alpha \ [\text{unrest}]: \text{in}\alpha \ \# \ \llbracket b \rrbracket_{>}$   
**by**  $(\text{transfer, auto simp add: in}\alpha\text{-def})$

**lemma**  $\text{unrest-pre-in-var} \ [\text{unrest}]:$   
 $x \ \# \ p1 \Longrightarrow \$x \ \# \ \llbracket p1 \rrbracket_{<}$   
**by**  $(\text{transfer, simp})$

**lemma**  $\text{unrest-post-out-var} \ [\text{unrest}]:$   
 $x \ \# \ p1 \Longrightarrow \$x' \ \# \ \llbracket p1 \rrbracket_{>}$   
**by**  $(\text{transfer, simp})$

**lemma**  $\text{unrest-convr-out}\alpha \ [\text{unrest}]:$   
 $\text{in}\alpha \ \# \ p \Longrightarrow \text{out}\alpha \ \# \ p^-$   
**by**  $(\text{transfer, auto simp add: in}\alpha\text{-def out}\alpha\text{-def})$

**lemma**  $\text{unrest-convr-in}\alpha \ [\text{unrest}]:$   
 $\text{out}\alpha \ \# \ p \Longrightarrow \text{in}\alpha \ \# \ p^-$

by (transfer, auto simp add: in $\alpha$ -def out $\alpha$ -def)

**lemma** unrest-in-rel-var-res [unrest]:  
 $uvar\ x \Longrightarrow \$x \# (P \upharpoonright_{\alpha} x)$   
 by (simp add: rel-var-res-def unrest)

**lemma** unrest-out-rel-var-res [unrest]:  
 $uvar\ x \Longrightarrow \$x' \# (P \upharpoonright_{\alpha} x)$   
 by (simp add: rel-var-res-def unrest)

## 7.2 Substitution laws

It should be possible to substantially generalise the following two laws

**lemma** usubst-seq-left [usubst]:  
 $\llbracket uvar\ x; out\alpha \# v \rrbracket \Longrightarrow (P ;; Q) \llbracket v / \$x \rrbracket = ((P \llbracket v / \$x \rrbracket) ;; Q)$   
 apply (rel-tac)  
 apply (rename-tac x v P Q a y ya)  
 apply (rule-tac x=ya in exI)  
 apply (simp)  
 apply (drule-tac x=a in spec)  
 apply (drule-tac x=y in spec)  
 apply (drule-tac x=ya in spec)  
 apply (simp)  
 apply (rename-tac x v P Q a ba y)  
 apply (rule-tac x=y in exI)  
 apply (drule-tac x=a in spec)  
 apply (drule-tac x=y in spec)  
 apply (drule-tac x=ba in spec)  
 apply (simp)  
 done

**lemma** usubst-seq-right [usubst]:  
 $\llbracket uvar\ x; in\alpha \# v \rrbracket \Longrightarrow (P ;; Q) \llbracket v / \$x' \rrbracket = (P ;; Q \llbracket v / \$x' \rrbracket)$   
 apply (rel-tac)  
 apply (rename-tac x v P Q b xa ya)  
 apply (rule-tac x=ya in exI)  
 apply (simp)  
 apply (drule-tac x=ya in spec)  
 apply (drule-tac x=b in spec)  
 apply (drule-tac x=xa in spec)  
 apply (simp)  
 apply (rename-tac x v P Q b aa y)  
 apply (rule-tac x=y in exI)  
 apply (simp)  
 apply (drule-tac x=aa in spec)  
 apply (drule-tac x=b in spec)  
 apply (drule-tac x=y in spec)  
 apply (simp)  
 done

**lemma** usubst-condr [usubst]:  
 $\sigma \dagger (P \triangleleft b \triangleright Q) = (\sigma \dagger P \triangleleft \sigma \dagger b \triangleright \sigma \dagger Q)$   
 by rel-tac

**lemma** subst-skip-r [usubst]:

fixes  $x :: ('a, 'α) \text{ uvar}$   
 shows  $II \llbracket v \rrbracket_{<} / \$x = (x := v)$   
 by (*rel-tac*)

### 7.3 Lifting laws

**lemma** *lift-pre-conj* [*ulift*]:  $\llbracket p \wedge q \rrbracket_{<} = (\llbracket p \rrbracket_{<} \wedge \llbracket q \rrbracket_{<})$   
 by (*pred-tac*)

**lemma** *lift-post-conj* [*ulift*]:  $\llbracket p \wedge q \rrbracket_{>} = (\llbracket p \rrbracket_{>} \wedge \llbracket q \rrbracket_{>})$   
 by (*pred-tac*)

**lemma** *lift-pre-disj* [*ulift*]:  $\llbracket p \vee q \rrbracket_{<} = (\llbracket p \rrbracket_{<} \vee \llbracket q \rrbracket_{<})$   
 by (*pred-tac*)

**lemma** *lift-post-disj* [*ulift*]:  $\llbracket p \vee q \rrbracket_{>} = (\llbracket p \rrbracket_{>} \vee \llbracket q \rrbracket_{>})$   
 by (*pred-tac*)

**lemma** *lift-pre-not* [*ulift*]:  $\llbracket \neg p \rrbracket_{<} = (\neg \llbracket p \rrbracket_{<})$   
 by (*pred-tac*)

**lemma** *lift-post-not* [*ulift*]:  $\llbracket \neg p \rrbracket_{>} = (\neg \llbracket p \rrbracket_{>})$   
 by (*pred-tac*)

### 7.4 Relation laws

Homogeneous relations form a quantale

**abbreviation** *truer* ::  $'α \text{ hrelation } (true_h)$  **where**  
*truer*  $\equiv true$

**abbreviation** *false* ::  $'α \text{ hrelation } (false_h)$  **where**  
*false*  $\equiv false$

**interpretation** *upred-quantale*: *unital-quantale-plus*

**where** *times* = *seqr* **and** *one* = *skip-r* **and** *Sup* = *Sup* **and** *Inf* = *Inf* **and** *inf* = *inf* **and** *less-eq* =  
*less-eq* **and** *less* = *less*

**and** *sup* = *sup* **and** *bot* = *bot* **and** *top* = *top*

**apply** (*unfold-locales*)

**apply** (*rel-tac*)

**apply** (*unfold SUP-def, transfer, auto*)

**apply** (*unfold SUP-def, transfer, auto*)

**apply** (*unfold INF-def, transfer, auto*)

**apply** (*unfold INF-def, transfer, auto*)

**apply** (*rel-tac*)

**apply** (*rel-tac*)

**done**

**lemma** *drop-pre-inv* [*simp*]:  $\llbracket out\alpha \# p \rrbracket \implies \llbracket p \rrbracket_{<}_{<} = p$   
 by (*pred-tac, auto simp add: outα-def*)

**abbreviation** *ustar* ::  $'α \text{ hrelation} \Rightarrow 'α \text{ hrelation } (^*_u [999] 999)$  **where**  
 $P^*_u \equiv \text{unital-quantale.qstar } II \text{ op } ;; \text{ Sup } P$

**definition** *while* ::  $'α \text{ condition} \Rightarrow 'α \text{ hrelation} \Rightarrow 'α \text{ hrelation } (while - do - od)$  **where**  
 $while \ b \ do \ P \ od = ((\llbracket b \rrbracket_{<} \wedge P)^*_u \wedge (\neg \llbracket b \rrbracket_{>}))$

**declare** *while-def* [*urel-defs*]

**lemma** *cond-idem*:  $(P \triangleleft b \triangleright P) = P$  **by** *rel-tac*

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

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

**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** *rel-tac*

**lemma** *cond-unit-T*:  $(P \triangleleft \text{true} \triangleright Q) = P$  **by** *rel-tac*

**lemma** *cond-unit-F*:  $(P \triangleleft \text{false} \triangleright Q) = Q$  **by** *rel-tac*

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

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

**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** *rel-tac*

**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** *rel-tac*

**lemma** *cond-imp-distr*:

$((P \Rightarrow Q) \triangleleft b \triangleright (R \Rightarrow S)) = ((P \triangleleft b \triangleright R) \Rightarrow (Q \triangleleft b \triangleright S))$  **by** *rel-tac*

**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** *rel-tac*

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

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

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

**lemma** *comp-cond-left-distr*:

$((P \triangleleft b \triangleright_r Q) ;; R) = ((P ;; R) \triangleleft b \triangleright_r (Q ;; R))$

**by** *rel-tac*

These laws may seem to duplicate quantale laws, but they don't – they are applicable to non-homogeneous relations as well, which will become important later.

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

**by** *rel-tac*

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

$(II ;; P) = P$

**by** *rel-tac*

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

$(P ;; II) = P$

**by** *rel-tac*

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

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

**by** *pred-tac*

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

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

**by** *pred-tac*

**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-tac*, *blast*)

**lemma** *pre-skip-post*:  $(\lceil b \rceil_{<} \wedge II) = (II \wedge \lceil b \rceil_{>})$

**by** (*rel-tac*)

**lemma** *seqr-exists-left*:

$uvar\ x \implies ((\exists \$x \cdot P) ;; Q) = (\exists \$x \cdot (P ;; Q))$

**by** (*rel-tac*, *auto simp add: comp-def*)

**lemma** *seqr-exists-right*:

$uvar\ x \implies (P ;; (\exists \$x' \cdot Q)) = (\exists \$x' \cdot (P ;; Q))$

**by** (*rel-tac*, *auto simp add: comp-def*)

We should be able to generalise this law to arbitrary assignments at some point, but that requires additional conversion operators for substitutions that act only on *in* $\alpha$ .

**lemma** *assign-subst* [*usubst*]:

$\llbracket uvar\ x; uvar\ y \rrbracket \implies [\$x \mapsto_s [u]_{<}] \dagger (y := v) = (x, y := u, [x \mapsto_s u] \dagger v)$

**by** *rel-tac*

**lemma** *assigns-idem*:  $uvar\ x \implies (x, x := u, v) = (x := v)$

**by** (*simp add: usubst*)

**lemma** *assigns-comp*:  $(\text{assigns-r}\ f ;; \text{assigns-r}\ g) = \text{assigns-r}\ (g \circ f)$

**by** (*transfer*, *auto simp add: relcomp-unfold*)

**lemma** *assigns-r-comp*:  $uvar\ x \implies (\langle \sigma \rangle_a ;; P) = (\lceil \sigma \rceil_s \dagger P)$

**by** *rel-tac*

**lemma** *assign-r-comp*:  $uvar\ x \implies (x := u ;; P) = ([\$x \mapsto_s [u]_{<}] \dagger P)$

**by** (*simp add: assigns-r-comp usubst*)

**lemma** *assign-test*:  $uvar\ x \implies (x := \langle u \rangle ;; x := \langle v \rangle) = (x := \langle v \rangle)$

**by** (*simp add: assigns-comp subst-upd-comp subst-lit usubst-upd-idem*)

**lemma** *skip-r-unfold*:

$uvar\ x \implies II = (\$x' =_u \$x \wedge II \upharpoonright_{\alpha} x)$

**by** (*rel-tac*, *blast*, *metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens.get-put*)

**lemma** *assign-unfold*:

$uvar\ x \implies (x := v) = (\$x' =_u [v]_{<} \wedge II \upharpoonright_{\alpha} x)$

**apply** (*rel-tac*, *auto simp add: comp-def*)

**using** *vwb-lens.put-eq* **by** *fastforce*

**lemma** *seqr-or-distl*:

$((P \vee Q) ;; R) = ((P ;; R) \vee (Q ;; R))$

**by** *rel-tac*

**lemma** *seqr-or-distr*:



$(P ;; (Q \vee R)) = ((P ;; Q) \vee (P ;; R))$   
 by *rel-tac*

**lemma** *seqr-middle*:

**assumes** *uvar x*  
**shows**  $(P ;; Q) = (\exists v \cdot P[\llbracket v \rrbracket / \$x'] ;; Q[\llbracket v \rrbracket / \$x])$   
**using** *assms*  
**apply** (*rel-tac*)  
**apply** (*rename-tac xa P Q a b y*)  
**apply** (*rule-tac x=var-lookup xa y in exI*)  
**apply** (*rule-tac x=y in exI*)  
**apply** (*simp*)  
**done**

**theorem** *precond-equiv*:

$P = (P ;; \text{true}) \longleftrightarrow (\text{out}\alpha \# P)$   
 by (*rel-tac*)

**theorem** *postcond-equiv*:

$P = (\text{true} ;; P) \longleftrightarrow (\text{in}\alpha \# P)$   
 by (*rel-tac*)

**lemma** *precond-right-unit*:  $\text{out}\alpha \# p \implies (p ;; \text{true}) = p$   
 by (*metis precondition-equiv*)

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

**theorem** *precond-left-zero*:

**assumes**  $\text{out}\alpha \# p \neq \text{false}$   
**shows**  $(\text{true} ;; p) = \text{true}$   
**using** *assms*  
**apply** (*simp add: outα-def upred-defs*)  
**apply** (*transfer, auto simp add: relcomp-unfold, rule ext, auto*)  
**apply** (*rename-tac p b*)  
**apply** (*subgoal-tac  $\exists b1 b2. p (b1, b2)$* )  
**apply** (*auto*)  
**done**

## 7.5 Converse laws

**lemma** *convr-invol* [*simp*]:  $p^{--} = p$   
 by *pred-tac*

**lemma** *lit-convr* [*simp*]:  $\llbracket v \rrbracket^- = \llbracket v \rrbracket$   
 by *pred-tac*

**lemma** *uivar-convr* [*simp*]:

**fixes**  $x :: ('a, 'α) \text{uvar}$   
**shows**  $(\$x)^- = \$x'$   
 by *pred-tac*

**lemma** *uovar-convr* [*simp*]:

**fixes**  $x :: ('a, 'α) \text{uvar}$   
**shows**  $(\$x')^- = \$x$   
 by *pred-tac*

**lemma** *uop-convr* [*simp*]:  $(uop\ f\ u)^- = uop\ f\ (u^-)$   
 by (*pred-tac*)

**lemma** *bop-convr* [*simp*]:  $(bop\ f\ u\ v)^- = bop\ f\ (u^-)\ (v^-)$   
 by (*pred-tac*)

**lemma** *eq-convr* [*simp*]:  $(p =_u q)^- = (p^- =_u q^-)$   
 by (*pred-tac*)

**lemma** *disj-convr* [*simp*]:  $(p \vee q)^- = (q^- \vee p^-)$   
 by (*pred-tac*)

**lemma** *conj-convr* [*simp*]:  $(p \wedge q)^- = (q^- \wedge p^-)$   
 by (*pred-tac*)

**lemma** *seqr-convr* [*simp*]:  $(p ;; q)^- = (q^- ;; p^-)$   
 by (*rel-tac*)

**theorem** *seqr-pre-transfer*:  $in\alpha \# q \implies ((P \wedge q) ;; R) = (P ;; (q^- \wedge R))$   
 by (*rel-tac*)

**theorem** *seqr-post-out*:  $in\alpha \# r \implies (P ;; (Q \wedge r)) = ((P ;; Q) \wedge r)$   
 by (*rel-tac*)

**theorem** *seqr-post-transfer*:  $out\alpha \# q \implies (P ;; (q \wedge R)) = (P \wedge q^- ;; R)$   
 by (*simp add: seqr-pre-transfer unrest-convr-in\alpha*)

**lemma** *seqr-pre-out*:  $out\alpha \# p \implies ((p \wedge Q) ;; R) = (p \wedge (Q ;; R))$   
 by (*rel-tac*)

**lemma** *seqr-true-lemma*:  
 $(P = (\neg (\neg P ;; true))) = (P = (P ;; true))$   
 by (*rel-tac*)

**lemma** *shEx-lift-seq* [*uquant-lift*]:  
 $((\exists x \cdot P(x)) ;; (\exists y \cdot Q(y))) = (\exists x \cdot \exists y \cdot P(x) ;; Q(y))$   
 by (*pred-tac*)

While loop laws

**lemma** *while-cond-true*:  
 $((while\ b\ do\ P\ od) \wedge [b]_{<}) = ((P \wedge [b]_{<}) ;; while\ b\ do\ P\ od)$

**proof** –

**have**  $(while\ b\ do\ P\ od \wedge [b]_{<}) = ((([b]_{<} \wedge P)^*_u \wedge (\neg [b]_{>})) \wedge [b]_{<})$   
 by (*simp add: while-def*)  
**also have**  $\dots = (((II \vee ([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u)) \wedge \neg [b]_{>} \wedge [b]_{<})$   
 by (*simp add: disj-upred-def*)  
**also have**  $\dots = (([b]_{<} \wedge (II \vee ([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u))) \wedge (\neg [b]_{>}))$   
 by (*simp add: conj-comm utp-pred.inf.left-commute*)  
**also have**  $\dots = ((([b]_{<} \wedge II) \vee ([b]_{<} \wedge ([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u))) \wedge (\neg [b]_{>}))$   
 by (*simp add: conj-disj-distr*)  
**also have**  $\dots = ((([b]_{<} \wedge II) \vee ([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u))) \wedge (\neg [b]_{>}))$   
 by (*subst seqr-pre-out[THEN sym], simp add: unrest, rel-tac*)  
**also have**  $\dots = (((II \wedge [b]_{>}) \vee ([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u))) \wedge (\neg [b]_{>}))$   
 by (*simp add: pre-skip-post*)

```

also have ... = ((II ∧ [b]₃ ∧ ¬ [b]₃) ∨ ((([b]₃ ∧ P) ;; ([b]₃ ∧ P)*u)) ∧ (¬ [b]₃))
  by (simp add: utp-pred.inf.assoc utp-pred.inf-sup-distrib2)
also have ... = ((([b]₃ ∧ P) ;; ([b]₃ ∧ P)*u)) ∧ (¬ [b]₃)
  by simp
also have ... = (([b]₃ ∧ P) ;; ((([b]₃ ∧ P)*u)) ∧ (¬ [b]₃))
  by (simp add: segr-post-out unrest)
also have ... = ((P ∧ [b]₃) ;; while b do P od)
  by (simp add: utp-pred.inf-commute while-def)
finally show ?thesis .
qed

```

**lemma** *while-cond-false*:

$((\text{while } b \text{ do } P \text{ od}) \wedge (\neg [b]_<)) = (II \wedge \neg [b]_<)$

**proof** –

```

have (while b do P od ∧ (¬ [b]₃)) = ((([b]₃ ∧ P)*u ∧ (¬ [b]₃)) ∧ (¬ [b]₃))
  by (simp add: while-def)
also have ... = (((II ∨ ([b]₃ ∧ P) ;; ([b]₃ ∧ P)*u)) ∧ ¬ [b]₃) ∧ (¬ [b]₃)
  by (simp add: disj-upred-def)
also have ... = (((II ∧ ¬ [b]₃) ∧ ¬ [b]₃) ∨ ((¬ [b]₃) ∧ ((([b]₃ ∧ P) ;; ([b]₃ ∧ P)*u)) ∧ ¬ [b]₃))
  by (simp add: conj-disj-distr utp-pred.inf.commute)
also have ... = (((II ∧ ¬ [b]₃) ∧ ¬ [b]₃) ∨ (((¬ [b]₃) ∧ ([b]₃ ∧ P) ;; ([b]₃ ∧ P)*u)) ∧ ¬ [b]₃))
  by (simp add: segr-pre-out unrest-not unrest-pre-outα utp-pred.inf.assoc)
also have ... = (((II ∧ ¬ [b]₃) ∧ ¬ [b]₃) ∨ ((false ;; ([b]₃ ∧ P)*u)) ∧ ¬ [b]₃))
  by (simp add: conj-comm utp-pred.inf.left-commute)
also have ... = ((II ∧ ¬ [b]₃) ∧ ¬ [b]₃)
  by simp
also have ... = (II ∧ ¬ [b]₃)
  by rel-tac
finally show ?thesis .
qed

```

**theorem** *while-unfold*:

$\text{while } b \text{ do } P \text{ od} = ((P ;; \text{while } b \text{ do } P \text{ od}) \triangleleft b \triangleright_r II)$

**by** (*metis* (*no-types*, *hide-lams*) *bounded-semilattice-sup-bot-class.sup-bot.left-neutral comp-cond-left-distr cond-def cond-idem disj-comm disj-upred-def segr-right-zero upred-quantale.bot-zero utp-pred.inf-bot-right utp-pred.inf-commute while-cond-false while-cond-true*)

**end**

## 7.6 Weakest precondition calculus

**theory** *utp-wp*

**imports** *utp-rel*

**begin**

A very quick implementation of wp – more laws still needed!

**named-theorems** *wp*

**method** *wp-tac* = (*simp add: wp*)

**consts**

*wp* :: 'a ⇒ 'b ⇒ 'c (**infix** *wp* 60)

**definition** *wp-upred* :: ('α, 'β) relation ⇒ 'β condition ⇒ 'α condition **where**

*wp-upred* *Q* *r* = [¬ (*Q* ;; ¬ [r]₃)]₃

**adhoc-overloading**

*wp wp-upred*

**declare** *wp-upred-def* [*urel-defs*]

**theorem** *wp-assigns-r* [*wp*]:

*(assigns-r  $\sigma$ ) wp r =  $\sigma \dagger r$*

**by** *rel-tac*

**theorem** *wp-skip-r* [*wp*]:

*II wp r = r*

**by** *rel-tac*

**theorem** *wp-true* [*wp*]:

*r  $\neq$  true  $\implies$  true wp r = false*

**by** *rel-tac*

**theorem** *wp-conj* [*wp*]:

*P wp (q  $\wedge$  r) = (P wp q  $\wedge$  P wp r)*

**by** *rel-tac*

**theorem** *wp-seq-r* [*wp*]: *(P ;; Q) wp r = P wp (Q wp r)*

**by** *rel-tac*

**theorem** *wp-cond* [*wp*]: *(P  $\triangleleft$  b  $\triangleright_r$  Q) wp r = ((b  $\implies$  P wp r)  $\wedge$  (( $\neg$  b)  $\implies$  Q wp r))*

**by** *rel-tac*

**end**

## 8 UTP Theories

**theory** *utp-theory*

**imports** *utp-rel*

**begin**

**type-synonym** *' $\alpha$  Healthiness-condition* = *' $\alpha$  upred  $\implies$  ' $\alpha$  upred*

**definition**

*Healthy::'α upred  $\implies$  ' $\alpha$  Healthiness-condition  $\implies$  bool* (**infix** *is 30*)

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

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

**unfolding** *Healthy-def* **by** *auto*

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

**definition** *Idempotent*(*H*)  $\longleftrightarrow$  ( $\forall P. H(H(P)) = H(P)$ )

**definition** *Monotonic*(*H*)  $\longleftrightarrow$  ( $\forall P Q. Q \sqsubseteq P \longrightarrow (H(Q) \sqsubseteq H(P))$ )

**definition** *IMH*(*H*)  $\longleftrightarrow$  *Idempotent*(*H*)  $\wedge$  *Monotonic*(*H*)

**definition** *Antitone*(*H*)  $\longleftrightarrow$  ( $\forall P Q. Q \sqsubseteq P \longrightarrow (H(P) \sqsubseteq H(Q))$ )

**definition**  $NM : NM(P) = (\neg P \wedge true)$

**lemma** *Monotonic(NM)*  
**apply** (*simp add:Monotonic-def*)  
**nitpick**  
**oops**

**lemma** *Antitone(NM)*  
**by** (*simp add:Antitone-def NM*)

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

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

**lemma** *Conjunctive-Monotonic*:  
 $Conjunctive(H) \Longrightarrow Monotonic(H)$   
**unfolding** *Conjunctive-def Monotonic-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.inf.assoc utp-pred.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.inf.assoc utp-pred.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.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.inf-commute*)

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

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

**lemma** *FunctionalConjunctive-Monotonic*:  
 $FunctionalConjunctive(H) \Longrightarrow Monotonic(H)$

```

unfolding FunctionalConjunctive-def by (metis Monotonic-def utp-pred.inf-mono)

lemma WeakConjunctive-Refinement:
  assumes WeakConjunctive(HC)
  shows  $P \sqsubseteq HC(P)$ 
  using assms unfolding WeakConjunctive-def by (metis utp-pred.inf.cobounded1)

lemma WeakCojunctive-Healthy-Refinement:
  assumes WeakConjunctive(HC) and P is HC
  shows  $HC(P) \sqsubseteq P$ 
  using assms unfolding WeakConjunctive-def Healthy-def by simp

lemma WeakConjunctive-implies-WeakConjunctive:
  Conjunctive(H)  $\implies$  WeakConjunctive(H)
  unfolding WeakConjunctive-def Conjunctive-def by pred-tac

declare Conjunctive-def [upred-defs]
declare Monotonic-def [upred-defs]

end

```

## 9 Example UTP theory: Boyle's laws

```

theory utp-boyle
imports utp-theory
begin

```

Boyle's law states that  $k = p * V$  is invariant. We here encode this as a simple UTP theory. We first create a record to represent the alphabet of the theory consisting of the three variables  $k$ ,  $p$  and  $V$ .

```

record alpha-boyle =
  boyle-k :: real
  boyle-p :: real
  boyle-V :: real

```

For now we have to explicitly cast the fields to UTP variables using the VAR syntactic transformation function – in future we'd like to automate this. We also have to add the definition equations for these variables to the simplification set for predicates to enable automated proof through our tactics.

```

definition k = VAR boyle-k
definition p = VAR boyle-p
definition V = VAR boyle-V

```

```

declare k-def [upred-defs] and p-def [upred-defs] and V-def [upred-defs]

```

Next we state Boyle's law using the healthiness condition B and likewise add it to the UTP predicate definitional equation set. The syntax differs a little from UTP; we try not to override HOL constants and so UTP predicate equality is subscripted. Moreover to distinguish variables standing for a predicate (like  $\phi$ ) from variables standing for UTP variables we have to prepend the latter with an ampersand.

```

definition  $B(\varphi) = ((\exists k \cdot \varphi) \wedge (\&k =_u \&p * \&V))$ 

```

```

declare B-def [upred-defs]

```

We can then prove that  $B$  is both idempotent and monotone simply by application of the predicate tactic.

**lemma** *B-idempotent*:

$$B(B(P)) = B(P)$$

**by** *pred-tac*

**lemma** *B-monotone*:

$$X \sqsubseteq Y \implies B(X) \sqsubseteq B(Y)$$

**by** *pred-tac*

We also create some example observations; the first satisfies Boyle's law and the second doesn't.

**definition**  $\varphi_1 = ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 50))$

**definition**  $\varphi_2 = ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 100))$

We prove that  $\varphi_1$  satisfied by Boyle's law by simplication of its definitional equation and then application of the predicate tactic.

**lemma** *B- $\varphi_1$ :  $\varphi_1$  is B*

**by** (*simp add:  $\varphi_1$ -def, pred-tac*)

We prove that  $\varphi_2$  does not satisfy Boyle's law by showing it's in fact equal to  $\varphi_1$ . We do this via an automated Isar proof.

**lemma** *B- $\varphi_2$ :  $B(\varphi_2) = \varphi_1$*

**proof** –

**have**  $B(\varphi_2) = B((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 100))$

**by** (*simp add:  $\varphi_2$ -def*)

**also have**  $\dots = ((\exists k \cdot (\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 100)) \wedge (\&k =_u \&p * \&V))$

**by** *pred-tac*

**also have**  $\dots = ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u \&p * \&V))$

**by** *pred-tac*

**also have**  $\dots = ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 50))$

**by** *pred-tac*

**also have**  $\dots = \varphi_1$

**by** (*simp add:  $\varphi_1$ -def*)

**finally show** *?thesis* .

**qed**

**end**

## 10 Designs

**theory** *utp-designs*

**imports**

*utp-rel*

*utp-wp*

*utp-theory*

**begin**

In UTP, in order to explicitly record the termination of a program, a subset of alphabetized relations is introduced. These relations are called designs and their alphabet should contain the special boolean observational variable *ok*. It is used to record the start and termination of a program.

## 10.1 Definitions

In the following, the definitions of designs alphabets, designs and healthiness (well-formedness) conditions are given. The healthiness conditions of designs are defined by  $H1$ ,  $H2$ ,  $H3$  and  $H4$ .

**record**  $\alpha$ -d = des-ok::bool

The ok variable is defined using the syntactic translation  $VAR$

**definition**  $ok = VAR \text{ des-ok}$

**declare**  $ok\text{-def}$  [ $upred\text{-defs}$ ]

**lemma**  $uvar\text{-ok}$  [ $simp$ ]:  $uvar \text{ ok}$   
**by** ( $unfold\text{-locales}$ ,  $simp\text{-all add: ok-def}$ )

**type-synonym**  $'\alpha$  alphabet-d =  $'\alpha$  alpha-d-scheme alphabet

**type-synonym**  $('a, '\alpha)$  uvar-d =  $('a, '\alpha \text{ alphabet-d})$  uvar

**type-synonym**  $(' \alpha, ' \beta)$  relation-d =  $(' \alpha \text{ alphabet-d}, ' \beta \text{ alphabet-d})$  relation

**type-synonym**  $' \alpha$  hrelation-d =  $' \alpha \text{ alphabet-d hrelation}$

**definition**  $des\text{-lens} :: (' \alpha, ' \alpha \text{ alphabet-d}) \text{ lens}$  **where**  
 $des\text{-lens} = (\mid \text{ lens-get} = \text{more}, \text{ lens-put} = \text{rec-put more-update} \mid)$

**lemma**  $uvar\text{-des-lens: uvar des-lens}$   
**apply** ( $unfold\text{-locales}$ )  
**apply** ( $simp\text{-all add: des-lens-def}$ )  
**done**

It would be nice to be able to prove some general distributivity properties about these lifting operators. I don't know if that's possible somehow...

**lift-definition**  $lift\text{-desr} :: (' \alpha, ' \beta) \text{ relation} \Rightarrow (' \alpha, ' \beta) \text{ relation-d } (\lceil - \rceil_D)$  **is**  
 $\lambda P (A, A'). P (\text{more } A, \text{more } A') .$

**lift-definition**  $drop\text{-desr} :: (' \alpha, ' \beta) \text{ relation-d} \Rightarrow (' \alpha, ' \beta) \text{ relation } (\lfloor - \rfloor_D)$  **is**  
 $\lambda P (A, A'). P (\mid \text{ des-ok} = \text{True}, \dots = A \mid), (\mid \text{ des-ok} = \text{True}, \dots = A' \mid) .$

**definition**  $design :: (' \alpha, ' \beta) \text{ relation-d} \Rightarrow (' \alpha, ' \beta) \text{ relation-d} \Rightarrow (' \alpha, ' \beta) \text{ relation-d}$  (**infixl**  $\vdash_{60}$ )  
**where**  $P \vdash Q = (\$ok \wedge P \Rightarrow \$ok' \wedge Q)$

An rdesign is a design that uses the Isabelle type system to prevent reference to ok in the assumption and commitment.

**definition**  $rdesign :: (' \alpha, ' \beta) \text{ relation} \Rightarrow (' \alpha, ' \beta) \text{ relation} \Rightarrow (' \alpha, ' \beta) \text{ relation-d}$  (**infixl**  $\vdash_r$   $60$ )  
**where**  $(P \vdash_r Q) = (\lceil P \rceil_D \vdash \lceil Q \rceil_D)$

An ndesign is a normal design, i.e. where the assumption is a condition

**definition**  $ndesign :: ' \alpha \text{ condition} \Rightarrow (' \alpha, ' \beta) \text{ relation} \Rightarrow (' \alpha, ' \beta) \text{ relation-d}$  (**infixl**  $\vdash_n$   $60$ )  
**where**  $(p \vdash_n Q) = (\lceil p \rceil_{<} \vdash_r Q)$

**definition**  $skip\text{-d} :: ' \alpha \text{ hrelation-d } (II_D)$   
**where**  $II_D \equiv (\text{true} \vdash_r II)$

**definition**  $assigns\text{-d} :: ' \alpha \text{ usubst} \Rightarrow ' \alpha \text{ hrelation-d}$   
**where**  $assigns\text{-d } \sigma = (\text{true} \vdash_r assigns\text{-r } \sigma)$

At some point assignment should be generalised to multiple variables and maybe also for selectors.



**abbreviation**  $assign-d :: ('a, 'α) uvar \Rightarrow ('a, 'α) uexpr \Rightarrow 'α \text{ hrelation-d}$  (**infix**  $:=_D$  40)  
**where**  $assign-d\ x\ v \equiv assigns-d\ [x \mapsto_s v]$

**definition**  $J :: 'α \text{ hrelation-d}$   
**where**  $J = ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D)$

**definition**  $H1\ (P) \equiv \$ok \Rightarrow P$

**definition**  $H2\ (P) \equiv P ;; J$

**definition**  $H3\ (P) \equiv P ;; II_D$

**definition**  $H4\ (P) \equiv ((P;;true) \Rightarrow P)$

**abbreviation**  $\sigma f :: ('α, 'β) \text{ relation-d} \Rightarrow ('α, 'β) \text{ relation-d}$  ( $^-^f$  [1000] 1000)  
**where**  $\sigma f\ D \equiv D \llbracket false/\$ok' \rrbracket$

**abbreviation**  $\sigma t :: ('α, 'β) \text{ relation-d} \Rightarrow ('α, 'β) \text{ relation-d}$  ( $^-^t$  [1000] 1000)  
**where**  $\sigma t\ D \equiv D \llbracket true/\$ok' \rrbracket$

**definition**  $pre\text{-}design :: ('α, 'β) \text{ relation-d} \Rightarrow ('α, 'β) \text{ relation}$  ( $pre_D'(-)$ ) **where**  
 $pre_D(P) = \lfloor \neg P^f \rfloor_D$

**definition**  $post\text{-}design :: ('α, 'β) \text{ relation-d} \Rightarrow ('α, 'β) \text{ relation}$  ( $post_D'(-)$ ) **where**  
 $post_D(P) = \lfloor P^t \rfloor_D$

**definition**  $wp\text{-}design :: ('α, 'β) \text{ relation-d} \Rightarrow 'β \text{ condition} \Rightarrow 'α \text{ condition}$  (**infix**  $wp_D$  60) **where**  
 $Q\ wp_D\ r = (\lfloor pre_D(Q) \rfloor ;; true)_{<} \wedge (post_D(Q)\ wp\ r)$

**declare**  $design\text{-}def$  [ $upred\text{-}defs$ ]  
**declare**  $rdesign\text{-}def$  [ $upred\text{-}defs$ ]  
**declare**  $skip\text{-}d\text{-}def$  [ $upred\text{-}defs$ ]  
**declare**  $J\text{-}def$  [ $upred\text{-}defs$ ]  
**declare**  $pre\text{-}design\text{-}def$  [ $upred\text{-}defs$ ]  
**declare**  $post\text{-}design\text{-}def$  [ $upred\text{-}defs$ ]  
**declare**  $wp\text{-}design\text{-}def$  [ $upred\text{-}defs$ ]

**declare**  $H1\text{-}def$  [ $upred\text{-}defs$ ]  
**declare**  $H2\text{-}def$  [ $upred\text{-}defs$ ]  
**declare**  $H3\text{-}def$  [ $upred\text{-}defs$ ]  
**declare**  $H4\text{-}def$  [ $upred\text{-}defs$ ]

**lemma**  $drop\text{-}desr\text{-}inv$  [ $simp$ ]:  $\lfloor \lceil P \rceil_D \rfloor_D = P$   
**by** ( $transfer$ ,  $simp$ )

**lemma**  $lift\text{-}desr\text{-}inv$ :  
 $\llbracket \$ok \# P; \$ok' \# P \rrbracket \Longrightarrow \lfloor \lceil P \rceil_D \rfloor_D = P$   
**apply** ( $rel\text{-}tac$ )  
**apply** ( $rename\text{-}tac\ P\ a\ b$ )  
**apply** ( $drule\text{-}tac\ x=a\ \text{in}\ spec$ )  
**apply** ( $drule\text{-}tac\ x=b\ \text{in}\ spec$ )  
**apply** ( $drule\text{-}tac\ x=True\ \text{in}\ spec$ )  
**apply** ( $metis\ alpha\text{-}d.\text{surjective}\ alpha\text{-}d.\text{update}\text{-}convs(1)$ )  
**apply** ( $drule\text{-}tac\ x=a\ \text{in}\ spec$ )  
**apply** ( $drule\text{-}tac\ x=b\ \text{in}\ spec$ )

```

  apply (drule-tac x=True in spec)
  apply (metis alpha-d.surjective alpha-d.update-convs(1))
done

```

## 10.2 Design laws

```

lemma lift-desr-unrest-ok [unrest]:
  $ok \# \lceil P \rceil_D $ok' \# \lceil P \rceil_D
  by (transfer, simp add: ok-def)+

```

```

lemma unrest-out-des-lift [unrest]: out\alpha \# p \implies out\alpha \# \lceil p \rceil_D
  by (pred-tac, auto simp add: out\alpha-def)

```

```

lemma lift-dists [simp]:
  \lceil true \rceil_D = true
  \lceil \neg P \rceil_D = (\neg \lceil P \rceil_D)
  \lceil P \wedge Q \rceil_D = (\lceil P \rceil_D \wedge \lceil Q \rceil_D)
  by (pred-tac)+

```

```

lemma lift-dist-seq [simp]:
  \lceil P ;; Q \rceil_D = (\lceil P \rceil_D ;; \lceil Q \rceil_D)
  by (rel-tac, metis alpha-d.select-convs(2))

```

**theorem** *design-refinement*:

```

  assumes
    $ok \# P1 $ok' \# P1 $ok \# P2 $ok' \# P2
    $ok \# Q1 $ok' \# Q1 $ok \# Q2 $ok' \# Q2
  shows (P1 \vdash Q1 \sqsubseteq P2 \vdash Q2) \longleftrightarrow ('P1 \Rightarrow P2' \wedge 'P1 \wedge Q2 \Rightarrow Q1')
proof -
  have (P1 \vdash Q1) \sqsubseteq (P2 \vdash Q2) \longleftrightarrow '($ok \wedge P2 \Rightarrow $ok' \wedge Q2) \Rightarrow ($ok \wedge P1 \Rightarrow $ok' \wedge Q1)'
  by pred-tac
  also with assms have ... = '(P2 \Rightarrow $ok' \wedge Q2) \Rightarrow (P1 \Rightarrow $ok' \wedge Q1)'
  by (subst subst-bool-split[of in-var ok], simp-all, subst-tac)
  also with assms have ... = '(\neg P2 \Rightarrow \neg P1) \wedge ((P2 \Rightarrow Q2) \Rightarrow P1 \Rightarrow Q1)'
  by (subst subst-bool-split[of out-var ok], simp-all, subst-tac)
  also have ... \longleftrightarrow '(P1 \Rightarrow P2)' \wedge 'P1 \wedge Q2 \Rightarrow Q1'
  by (pred-tac)
  finally show ?thesis .
qed

```

**theorem** *rdesign-refinement*:

```

(P1 \vdash_r Q1 \sqsubseteq P2 \vdash_r Q2) \longleftrightarrow ('P1 \Rightarrow P2' \wedge 'P1 \wedge Q2 \Rightarrow Q1')
  apply (simp add: rdesign-def)
  apply (subst design-refinement)
  apply (simp-all add: unrest)
  apply (pred-tac)
  apply (metis alpha-d.select-convs(2))+
done

```

**lemma** *design-refine-intro*:

```

  assumes 'P1 \Rightarrow P2' 'P1 \wedge Q2 \Rightarrow Q1'
  shows P1 \vdash Q1 \sqsubseteq P2 \vdash Q2
  using assms unfolding upred-defs
  by pred-tac

```

**theorem** *design-ok-false* [usubst]: (P \vdash Q) \llbracket false/\$ok \rrbracket = true

by (simp add: design-def usubst)

**theorem** *design-pre*:

$\$ok' \# P \implies \neg (P \vdash Q)^f = (\$ok \wedge P^f)$

by (simp add: design-def, subst-tac)

(metis (no-types, hide-lams) not-conj-deMorgans true-not-false(2) utp-pred.compl-top-eq  
utp-pred.sup.idem utp-pred.sup-compl-top var-in-var)

**theorem** *rdesign-pre* [simp]:  $pre_D(P \vdash_r Q) = P$

by pred-tac

**theorem** *design-post* [simp]:  $post_D(P \vdash_r Q) = (P \Rightarrow Q)$

by pred-tac

**theorem** *design-true-left-zero*:  $(true ;; (P \vdash Q)) = true$

**proof** –

have  $(true ;; (P \vdash Q)) = (\exists ok_0 \cdot true \llbracket \llcorner ok_0 \gg / \$ok' \rrbracket ;; (P \vdash Q) \llbracket \llcorner ok_0 \gg / \$ok \rrbracket)$

by (subst segr-middle[of ok], simp-all)

also have  $\dots = ((true \llbracket false / \$ok' \rrbracket ;; (P \vdash Q) \llbracket false / \$ok \rrbracket) \vee (true \llbracket true / \$ok' \rrbracket ;; (P \vdash Q) \llbracket true / \$ok \rrbracket))$

by (simp add: disj-comm false-alt-def true-alt-def)

also have  $\dots = ((true \llbracket false / \$ok' \rrbracket ;; true_h) \vee (true ;; ((P \vdash Q) \llbracket true / \$ok \rrbracket)))$

by (subst-tac, rel-tac)

also have  $\dots = true$

by (subst-tac, simp add: precond-right-unit unrest)

finally show ?thesis .

qed

**theorem** *design-composition*:

assumes

$\$ok \# P1 \ \$ok' \# P1 \ \$ok \# P2 \ \$ok' \# P2$

$\$ok \# Q1 \ \$ok' \# Q1 \ \$ok \# Q2 \ \$ok' \# Q2$

shows  $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (((\neg (\neg P1) ;; true)) \wedge \neg (Q1 ;; (\neg P2))) \vdash (Q1 ;; Q2))$

**proof** –

have  $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (\exists ok_0 \cdot ((P1 \vdash Q1) \llbracket \llcorner ok_0 \gg / \$ok' \rrbracket ;; (P2 \vdash Q2) \llbracket \llcorner ok_0 \gg / \$ok \rrbracket))$

by (rule segr-middle, simp)

also have  $\dots$

$= (((P1 \vdash Q1) \llbracket false / \$ok' \rrbracket ;; (P2 \vdash Q2) \llbracket false / \$ok \rrbracket) \vee ((P1 \vdash Q1) \llbracket true / \$ok' \rrbracket ;; (P2 \vdash Q2) \llbracket true / \$ok \rrbracket))$

by (simp add: true-alt-def false-alt-def, pred-tac)

also from *assms*

have  $\dots = (((\$ok \wedge P1 \Rightarrow Q1) ;; (P2 \Rightarrow \$ok' \wedge Q2)) \vee ((\neg (\$ok \wedge P1)) ;; true))$

by (simp add: design-def usubst unrest, pred-tac)

also have  $\dots = ((\neg \$ok ;; true_h) \vee (\neg P1 ;; true) \vee (Q1 ;; \neg P2) \vee (\$ok' \wedge (Q1 ;; Q2)))$

by (rel-tac)

also have  $\dots = ((\neg (\neg P1 ;; true) \wedge \neg (Q1 ;; \neg P2)) \vdash (Q1 ;; Q2))$

by (simp add: precond-right-unit design-def unrest, rel-tac)

finally show ?thesis .

qed

**theorem** *rdesign-composition*:

$((P1 \vdash_r Q1) ;; (P2 \vdash_r Q2)) = (((\neg (\neg P1) ;; true)) \wedge \neg (Q1 ;; (\neg P2))) \vdash_r (Q1 ;; Q2))$

by (simp add: rdesign-def design-composition unrest)

**lemma** *skip-d-alt-def*:  $II_D = true \vdash II$

by (rel-tac)

**theorem** *design-skip-idem* [simp]:

$$(II_D ;; II_D) = II_D$$

by (simp add: skip-d-def urel-defs, pred-tac)

**theorem** *design-composition-cond*:

assumes

$$\text{\$ok} \# p1 \text{ out}\alpha \# p1 \text{ \$ok} \# P2 \text{ \$ok}' \# P2$$

$$\text{\$ok} \# Q1 \text{ \$ok}' \# Q1 \text{ \$ok} \# Q2 \text{ \$ok}' \# Q2$$

$$\text{shows } ((p1 \vdash_r Q1) ;; (P2 \vdash_r Q2)) = ((p1 \wedge \neg (Q1 ;; (\neg P2))) \vdash_r (Q1 ;; Q2))$$

using *assms*

by (simp add: design-composition unrest precondition-right-unit)

**theorem** *rdesign-composition-cond*:

assumes *out* $\alpha \# p1$

$$\text{shows } ((p1 \vdash_r Q1) ;; (P2 \vdash_r Q2)) = ((p1 \wedge \neg (Q1 ;; (\neg P2))) \vdash_r (Q1 ;; Q2))$$

using *assms*

by (simp add: rdesign-def design-composition-cond unrest)

**theorem** *design-composition-wp*:

fixes *Q1 Q2* :: 'a *hrelation-d*

assumes

$$\text{ok} \# p1 \text{ ok} \# p2$$

$$\text{\$ok} \# Q1 \text{ \$ok}' \# Q1 \text{ \$ok} \# Q2 \text{ \$ok}' \# Q2$$

$$\text{shows } (([p1]_{<} \vdash_r Q1) ;; ([p2]_{<} \vdash_r Q2)) = (([p1 \wedge Q1 \text{ wp } p2]_{<} \vdash_r (Q1 ;; Q2))$$

using *assms*

by (simp add: design-composition-cond unrest, rel-tac)

**theorem** *rdesign-composition-wp*:

fixes *Q1 Q2* :: 'a *hrelation*

$$\text{shows } (([p1]_{<} \vdash_r Q1) ;; ([p2]_{<} \vdash_r Q2)) = (([p1 \wedge Q1 \text{ wp } p2]_{<} \vdash_r (Q1 ;; Q2))$$

by (simp add: rdesign-composition-cond unrest, rel-tac)

**theorem** *rdesign-wp* [wp]:

$$([p]_{<} \vdash_r Q) \text{ wp}_D r = (p \wedge Q \text{ wp } r)$$

by *rel-tac*

**theorem** *wpd-seq-r*:

fixes *Q1 Q2* :: 'a *hrelation*

$$\text{shows } ([p1]_{<} \vdash_r Q1 ;; [p2]_{<} \vdash_r Q2) \text{ wp}_D r = ([p1]_{<} \vdash_r Q1) \text{ wp}_D ([p2]_{<} \vdash_r Q2) \text{ wp}_D r$$

apply (simp add: wp)

apply (subst rdesign-composition-wp)

apply (simp only: wp)

apply (rel-tac)

done

**theorem** *design-left-unit* [simp]:

$$(II_D ;; P \vdash_r Q) = (P \vdash_r Q)$$

by (simp add: skip-d-def urel-defs, pred-tac)

**theorem** *design-right-cond-unit* [simp]:

assumes *out* $\alpha \# p$

$$\text{shows } (p \vdash_r Q ;; II_D) = (p \vdash_r Q)$$

using *assms*

by (simp add: skip-d-def redesign-composition-cond)

**lemma** *lift-des-skip-dr-unit* [simp]:

$(\lceil P \rceil_D ;; \lceil II \rceil_D) = \lceil P \rceil_D$

$(\lceil II \rceil_D ;; \lceil P \rceil_D) = \lceil P \rceil_D$

by *rel-tac rel-tac*

### 10.3 H1: No observation is allowed before initiation

**lemma** *H1-idem*:

$H1(H1 P) = H1(P)$

by *pred-tac*

**lemma** *H1-monotone*:

$P \sqsubseteq Q \implies H1(P) \sqsubseteq H1(Q)$

by *pred-tac*

**lemma** *H1-design-skip*:

$H1(II) = II_D$

by *rel-tac*

The H1 algebraic laws are valid only when  $\alpha(R)$  is homogeneous. This should maybe be generalised.

**theorem** *H1-algebraic-intro*:

**assumes**

$(true_h ;; R) = true_h$

$(II_D ;; R) = R$

**shows** *R is H1*

**proof** –

**have**  $R = (II_D ;; R)$  **by** (simp add: *assms*(2))

**also have**  $\dots = (H1(II) ;; R)$

**by** (simp add: *H1-design-skip*)

**also have**  $\dots = (\$ok \Rightarrow II) ;; R$

**by** (simp add: *H1-def*)

**also have**  $\dots = ((\neg \$ok ;; R) \vee R)$

**by** (simp add: *impl-alt-def seqr-or-distl*)

**also have**  $\dots = (((\neg \$ok ;; true_h) ;; R) \vee R)$

**by** (simp add: *precond-right-unit unrest*)

**also have**  $\dots = ((\neg \$ok ;; true_h) \vee R)$

**by** (*metis* *assms*(1) *seqr-assoc*)

**also have**  $\dots = (\$ok \Rightarrow R)$

**by** (simp add: *impl-alt-def precondition-right-unit unrest*)

**finally show** *?thesis* **by** (*metis* *H1-def Healthy-def'*)

**qed**

**lemma** *not-not-false*:

$(\neg \$ok) \neq false$

by (*pred-tac*, *metis* *alpha-d.select-convs*(1))

**theorem** *H1-left-zero*:

**assumes** *P is H1*

**shows**  $(true_h ;; P) = true_h$

**proof** –

**from** *assms* **have**  $(true_h ;; P) = (true_h ;; (\$ok \Rightarrow P))$

**by** (simp add: *H1-def Healthy-def'*)

also from *assms* have ... = ( $true_h ;; (\neg \$ok \vee P)$ )  
 by (*simp add: impl-alt-def*)  
 also from *assms* have ... = ( $(true_h ;; \neg \$ok) \vee (true_h ;; P)$ )  
 using *seqr-or-distr* by *blast*  
 also from *assms* have ... = ( $true \vee (true ;; P)$ )  
 by (*simp add: nok-not-false precondition-left-zero unrest*)  
 finally show *?thesis* by *rel-tac*  
 qed

**theorem** *H1-left-unit:*

fixes  $P :: 'a \text{ hrelation-}d$

assumes *P is H1*

shows  $(II_D ;; P) = P$

**proof** –

have  $(II_D ;; P) = ((\$ok \Rightarrow II) ;; P)$   
 by (*metis H1-def H1-design-skip*)  
 also have ... =  $((\neg \$ok ;; P) \vee P)$   
 by (*simp add: impl-alt-def seqr-or-distl*)  
 also from *assms* have ... =  $((\neg \$ok ;; true_h) ;; P) \vee P$   
 by (*simp add: precondition-right-unit unrest*)  
 also have ... =  $((\neg \$ok ;; (true_h ;; P)) \vee P)$   
 by (*simp add: seqr-assoc*)  
 also from *assms* have ... =  $(\$ok \Rightarrow P)$   
 by (*simp add: H1-left-zero impl-alt-def precondition-right-unit unrest*)  
 finally show *?thesis* using *assms*  
 by (*simp add: H1-def Healthy-def'*)  
 qed

**theorem** *H1-algebraic:*

$P \text{ is } H1 \iff (true_h ;; P) = true_h \wedge (II_D ;; P) = P$

using *H1-algebraic-intro H1-left-unit H1-left-zero* by *blast*

**theorem** *H1-nok-left-zero:*

fixes  $P :: 'a \text{ hrelation-}d$

assumes *P is H1*

shows  $(\neg \$ok ;; P) = (\neg \$ok)$

**proof** –

have  $(\neg \$ok ;; P) = ((\neg \$ok ;; true_h) ;; P)$   
 by (*simp add: precondition-right-unit unrest*)  
 also have ... =  $((\neg \$ok) ;; true_h)$   
 by (*metis H1-left-zero assms seqr-assoc*)  
 also have ... =  $(\neg \$ok)$   
 by (*simp add: precondition-right-unit unrest*)  
 finally show *?thesis* .  
 qed

## 10.4 H2: A specification cannot require non-termination

**lemma** *J-split:*

shows  $(P ;; J) = (P^f \vee (P^t \wedge \$ok'))$

**proof** –

have  $(P ;; J) = (P ;; ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D))$   
 by (*simp add: H2-def J-def design-def*)  
 also have ... =  $(P ;; ((\$ok \Rightarrow \$ok \wedge \$ok') \wedge \lceil II \rceil_D))$   
 by *rel-tac*  
 also have ... =  $((P ;; (\neg \$ok \wedge \lceil II \rceil_D)) \vee (P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))))$

```

  by rel-tac
also have ... = ( $P^f \vee (P^t \wedge \$ok')$ )
proof -
  have ( $P ;; (\neg \$ok \wedge \lceil II \rceil_D)$ ) =  $P^f$ 
  proof -
    have ( $P ;; (\neg \$ok \wedge \lceil II \rceil_D)$ ) = ( $(P \wedge \neg \$ok') ;; \lceil II \rceil_D$ )
    by rel-tac
    also have ... = ( $\exists \$ok' \cdot P \wedge \$ok' =_u false$ )
    by (rel-tac, metis (mono-tags, lifting) alpha-d.surjective alpha-d.update-convs(1))
    also have ... =  $P^f$ 
    by (metis one-point out-var-uvar ouvar-def unrest-false uvar-ok)
  finally show ?thesis .
qed
moreover have ( $P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))$ ) = ( $P^t \wedge \$ok'$ )
proof -
  have ( $P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))$ ) = ( $P ;; (\$ok \wedge II)$ )
  by (rel-tac, metis alpha-d.equality)
  also have ... = ( $P^t \wedge \$ok'$ )
  by (rel-tac, metis (full-types) alpha-d.surjective alpha-d.update-convs(1))+
  finally show ?thesis .
qed
ultimately show ?thesis
  by simp
qed
finally show ?thesis .
qed

```

lemma *H2-split*:

```

shows  $H2(P) = (P^f \vee (P^t \wedge \$ok'))$ 
by (simp add: H2-def J-split)

```

theorem *H2-equivalence*:

$P$  is *H2*  $\longleftrightarrow$  ' $P^f \Rightarrow P^t$ '

proof -

```

have ' $P \Leftrightarrow (P ;; J)$ '  $\longleftrightarrow$  ' $P \Leftrightarrow (P^f \vee (P^t \wedge \$ok'))$ '
  by (simp add: J-split)
also from assms have ...  $\longleftrightarrow$  ' $(P \Leftrightarrow P^f \vee P^t \wedge \$ok')^f \wedge (P \Leftrightarrow P^f \vee P^t \wedge \$ok')^t$ '
  by (simp add: subst-bool-split)
also from assms have ... = ' $(P^f \Leftrightarrow P^f) \wedge (P^t \Leftrightarrow P^f \vee P^t)$ '
  by subst-tac
also have ... = ' $P^t \Leftrightarrow (P^f \vee P^t)$ '
  by pred-tac
also have ... = ' $(P^f \Rightarrow P^t)$ '
  by pred-tac
finally show ?thesis using assms
  by (metis H2-def Healthy-def' taut-iff-eq)
qed

```

lemma *H2-equiv*:

$P$  is *H2*  $\longleftrightarrow$   $P^t \sqsubseteq P^f$

using *H2-equivalence* *refBy-order* by blast

lemma *H2-design*:

```

assumes  $\$ok \# P \ \$ok' \# P \ \$ok \# Q \ \$ok' \# Q$ 
shows  $H2(P \vdash Q) = P \vdash Q$ 

```

**using** *assms*  
**by** (*simp add: H2-split design-def usubst unrest, pred-tac*)

**lemma** *H2-rdesign:*  
 $H2(P \vdash_r Q) = P \vdash_r Q$   
**by** (*simp add: H2-design unrest rdesign-def*)

**theorem** *J-idem:*  
 $(J ;; J) = J$   
**by** (*simp add: J-def urel-defs, pred-tac*)

**theorem** *H2-idem:*  
 $H2(H2(P)) = H2(P)$   
**by** (*metis H2-def J-idem segr-assoc*)

**theorem** *H2-not-okay:*  $H2(\neg \$ok) = (\neg \$ok)$   
**proof** –  
**have**  $H2(\neg \$ok) = ((\neg \$ok)^f \vee ((\neg \$ok)^t \wedge \$ok'))$   
**by** (*simp add: H2-split*)  
**also have**  $\dots = (\neg \$ok \vee (\neg \$ok) \wedge \$ok')$   
**by** (*subst-tac*)  
**also have**  $\dots = (\neg \$ok)$   
**by** *pred-tac*  
**finally show** *?thesis* .  
**qed**

**theorem** *H1-H2-commute:*  
 $H1(H2 P) = H2(H1 P)$   
**proof** –  
**have**  $H2(H1 P) = (\$ok \Rightarrow P) ;; J$   
**by** (*simp add: H1-def H2-def*)  
**also from** *assms* **have**  $\dots = ((\neg \$ok \vee P) ;; J)$   
**by** *rel-tac*  
**also have**  $\dots = ((\neg \$ok ;; J) \vee (P ;; J))$   
**using** *segr-or-distl* **by** *blast*  
**also have**  $\dots = ((H2(\neg \$ok)) \vee H2(P))$   
**by** (*simp add: H2-def*)  
**also have**  $\dots = ((\neg \$ok) \vee H2(P))$   
**by** (*simp add: H2-not-okay*)  
**also have**  $\dots = H1(H2(P))$   
**by** *rel-tac*  
**finally show** *?thesis* **by** *simp*  
**qed**

**lemma** *ok-pre:*  $(\$ok \wedge \lceil pre_D(P) \rceil_D) = (\$ok \wedge (\neg P^f))$   
**by** (*pred-tac, metis (full-types) alpha-d.surjective alpha-d.update-convs(1)+*)

**lemma** *ok-post:*  $(\$ok \wedge \lceil post_D(P) \rceil_D) = (\$ok \wedge (P^t))$   
**by** (*pred-tac, metis (full-types) alpha-d.surjective alpha-d.update-convs(1)+*)

**theorem** *H1-H2-is-rdesign:*  
**assumes** *P is H1 P is H2*  
**shows**  $P = pre_D(P) \vdash_r post_D(P)$   
**proof** –  
**from** *assms* **have**  $P = (\$ok \Rightarrow H2(P))$



by (*simp add: H1-def Healthy-def'*)  
 also have ... = ( $\$ok \Rightarrow (P^f \vee (P^t \wedge \$ok'))$ )  
 by (*metis H2-split*)  
 also have ... = ( $\$ok \wedge (\neg P^f) \Rightarrow \$ok' \wedge P^t$ )  
 by *pred-tac*  
 also have ... = ( $\$ok \wedge (\neg P^f) \Rightarrow \$ok' \wedge \$ok \wedge P^t$ )  
 by *pred-tac*  
 also have ... = ( $\$ok \wedge [pre_D(P)]_D \Rightarrow \$ok' \wedge \$ok \wedge [post_D(P)]_D$ )  
 by (*simp add: ok-post ok-pre*)  
 also have ... = ( $\$ok \wedge [pre_D(P)]_D \Rightarrow \$ok' \wedge [post_D(P)]_D$ )  
 by *pred-tac*  
 also from *assms* have ... =  $pre_D(P) \vdash_r post_D(P)$   
 by (*simp add: rdesign-def design-def*)  
 finally show ?thesis .  
 qed

abbreviation  $H1\text{-}H2\ P \equiv H1\ (H2\ P)$

## 10.5 H3: The design assumption is a precondition

theorem *H3-idem*:

$H3(H3(P)) = H3(P)$   
 by (*metis H3-def design-skip-idem seqr-assoc*)

theorem *rdesign-H3-iff-pre*:

$P \vdash_r Q \text{ is } H3 \iff P = (P ;; true)$

proof –

have  $(P \vdash_r Q ;; II_D) = (P \vdash_r Q ;; true \vdash_r II)$   
 by (*simp add: skip-d-def*)  
 also from *assms* have ... =  $(\neg (\neg P ;; true) \wedge \neg (Q ;; \neg true)) \vdash_r (Q ;; II)$   
 by (*simp add: rdesign-composition*)  
 also from *assms* have ... =  $(\neg (\neg P ;; true) \wedge \neg (Q ;; \neg true)) \vdash_r Q$   
 by *simp*  
 also have ... =  $(\neg (\neg P ;; true)) \vdash_r Q$   
 by *pred-tac*  
 finally have  $P \vdash_r Q \text{ is } H3 \iff P \vdash_r Q = (\neg (\neg P ;; true)) \vdash_r Q$   
 by (*metis H3-def Healthy-def'*)  
 also have ...  $\iff P = (\neg (\neg P ;; true))$   
 by (*metis rdesign-pre*)  
 also have ...  $\iff P = (P ;; true)$   
 by (*simp add: seqr-true-lemma*)  
 finally show ?thesis .  
 qed

theorem *design-H3-iff-pre*:

assumes  $\$ok \# P\ \$ok' \# P\ \$ok \# Q\ \$ok' \# Q$   
 shows  $P \vdash Q \text{ is } H3 \iff P = (P ;; true)$

proof –

have  $P \vdash Q = [P]_D \vdash_r [Q]_D$   
 by (*simp add: assms lift-desr-inv rdesign-def*)  
 moreover hence  $[P]_D \vdash_r [Q]_D \text{ is } H3 \iff [P]_D = ([P]_D ;; true)$   
 using *rdesign-H3-iff-pre* by *blast*  
 ultimately show ?thesis  
 by (*metis assms drop-desr-inv lift-desr-inv lift-dist-seq lift-dists(1)*)  
 qed

**theorem** *H1-H3-commute:*

$H1 (H3 P) = H3 (H1 P)$

by *rel-tac*

**lemma** *skip-d-absorb-J-1:*

$(II_D ;; J) = II_D$

by (*metis H2-def H2-rdesign skip-d-def*)

**lemma** *skip-d-absorb-J-2:*

$(J ;; II_D) = II_D$

**proof** –

have  $(J ;; II_D) = ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D ;; true \vdash II)$

by (*simp add: J-def skip-d-alt-def*)

also have  $\dots = (\exists ok_0 \cdot ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D) \llbracket \ll ok_0 \gg / \$ok' \rrbracket ;; (true \vdash II) \llbracket \ll ok_0 \gg / \$ok \rrbracket)$

by (*subst segr-middle[of ok], simp-all*)

also have  $\dots = (((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D) \llbracket false / \$ok' \rrbracket ;; (true \vdash II) \llbracket false / \$ok \rrbracket)$

$\vee (((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D) \llbracket true / \$ok' \rrbracket ;; (true \vdash II) \llbracket true / \$ok \rrbracket)$

by (*simp add: disj-comm false-alt-def true-alt-def*)

also have  $\dots = ((\neg \$ok \wedge \lceil II \rceil_D ;; true) \vee (\lceil II \rceil_D ;; \$ok' \wedge \lceil II \rceil_D))$

by *rel-tac*

also have  $\dots = II_D$

by *rel-tac*

finally show *?thesis* .

qed

**lemma** *H2-H3-absorb:*

$H2 (H3 P) = H3 P$

by (*metis H2-def H3-def segr-assoc skip-d-absorb-J-1*)

**lemma** *H3-H2-absorb:*

$H3 (H2 P) = H3 P$

by (*metis H2-def H3-def segr-assoc skip-d-absorb-J-2*)

**theorem** *H2-H3-commute:*

$H2 (H3 P) = H3 (H2 P)$

by (*simp add: H2-H3-absorb H3-H2-absorb*)

**theorem** *H3-design-pre:*

assumes  $\$ok \# p \text{ out}\alpha \# p \ \$ok \# Q \ \$ok' \# Q$

shows  $H3(p \vdash Q) = p \vdash Q$

using *assms*

by (*metis Healthy-def' design-H3-iff-pre precondition-right-unit unrest-out $\alpha$ -var uvar-ok*)

**theorem** *H3-rdesign-pre:*

assumes  $\text{out}\alpha \# p$

shows  $H3(p \vdash_r Q) = p \vdash_r Q$

using *assms*

by (*simp add: H3-def*)

**theorem** *H1-H3-is-rdesign:*

assumes  $P \text{ is } H1 P \text{ is } H3$

shows  $P = \text{pre}_D(P) \vdash_r \text{post}_D(P)$

by (*metis H1-H2-is-rdesign H2-H3-absorb Healthy-def' assms*)

**theorem** *H1-H3-is-normal-design:*

**assumes**  $P$  is  $H1$   $P$  is  $H3$   
**shows**  $P = \lfloor pre_D(P) \rfloor_{<} \vdash_n post_D(P)$   
**by** (*metis H1-H3-is-rdesign assms drop-pre-inv ndesign-def precond-equiv rdesign-H3-iff-pre*)

**abbreviation**  $H1-H3\ p \equiv H1\ (H3\ p)$

**theorem** *wpd-seq-r-H1-H2* [*wp*]:

**fixes**  $P\ Q :: 'a\ hrelation-d$

**assumes**  $P$  is  $H1-H3$   $Q$  is  $H1-H3$

**shows**  $(P ;; Q)\ wp_D\ r = P\ wp_D\ (Q\ wp_D\ r)$

**by** (*smt H1-H3-commute H1-H3-is-rdesign H1-idem Healthy-def' assms(1) assms(2) drop-pre-inv precond-equiv rdesign-H3-iff-pre wpd-seq-r*)

## 10.6 H4: Feasibility

**theorem** *H4-idem*:

$H4(H4(P)) = H4(P)$

**by** *pred-tac*

**end**

# 11 Concurrent programming

**theory** *utp-concurrency*

**imports** *utp-designs*

**begin**

**no-notation**

*Sublist.parallel* (**infixl**  $\parallel$  50)

## 11.1 Design parallel composition

**definition** *design-par* ::  $('a, 'b)\ relation-d \Rightarrow ('a, 'b)\ relation-d \Rightarrow ('a, 'b)\ relation-d$  (**infixr**  $\parallel$  85)

**where**

$P \parallel Q = ((pre_D(P) \wedge pre_D(Q)) \vdash_r (post_D(P) \wedge post_D(Q)))$

**declare** *design-par-def* [*upred-defs*]

**lemma** *parallel-zero*:  $P \parallel true = true$

**proof** –

**have**  $P \parallel true = (pre_D(P) \wedge pre_D(true)) \vdash_r (post_D(P) \wedge post_D(true))$

**by** (*simp add: design-par-def*)

**also have**  $\dots = (pre_D(P) \wedge false) \vdash_r (post_D(P) \wedge true)$

**by** *rel-tac*

**also have**  $\dots = true$

**by** *rel-tac*

**finally show** *?thesis* .

**qed**

**lemma** *parallel-assoc*:  $P \parallel Q \parallel R = (P \parallel Q) \parallel R$

**by** *rel-tac*

**lemma** *parallel-comm*:  $P \parallel Q = Q \parallel P$

**by** *pred-tac*

**lemma** *parallel-idem*:  
**assumes**  $P$  is  $H1$   $P$  is  $H2$   
**shows**  $P \parallel P = P$   
**by** (*metis H1-H2-is-rdesign assms conj-idem design-par-def*)

**lemma** *parallel-mono-1*:  
**assumes**  $P_1 \sqsubseteq P_2$   $P_1$  is  $H1-H2$   $P_2$  is  $H1-H2$   
**shows**  $P_1 \parallel Q \sqsubseteq P_2 \parallel Q$

**proof** –

**have**  $pre_D(P_1) \vdash_r post_D(P_1) \sqsubseteq pre_D(P_2) \vdash_r post_D(P_2)$   
**by** (*metis H1-H2-commute H1-H2-is-rdesign H1-idem Healthy-def' assms*)  
**hence**  $(pre_D(P_1) \vdash_r post_D(P_1)) \parallel Q \sqsubseteq (pre_D(P_2) \vdash_r post_D(P_2)) \parallel Q$   
**by** (*auto simp add: rdesign-refinement design-par-def*) (*pred-tac+*)  
**thus** *?thesis*  
**by** (*metis H1-H2-commute H1-H2-is-rdesign H1-idem Healthy-def' assms*)

**qed**

**lemma** *parallel-mono-2*:  
**assumes**  $Q_1 \sqsubseteq Q_2$   $Q_1$  is  $H1-H2$   $Q_2$  is  $H1-H2$   
**shows**  $P \parallel Q_1 \sqsubseteq P \parallel Q_2$   
**by** (*metis assms parallel-comm parallel-mono-1*)

## 11.2 Parallel by merge

We describe the partition of a state space into a  $n$  pieces through the use of a list.

**type-synonym**  $'\alpha$  *partition* =  $'\alpha$  *list*

A merge relation is a design that describes how a partitioned state-space should be merged into a third state-space. For now the state-spaces for two merged processes should have the same type. This could potentially be generalised, but that might have an effect on our reasoning capabilities.

Extract the  $i$ th element of the second part

**definition** *ind-uvar*  $i$   $x = x ;_l des\_lens ;_l list\_lens\ i ;_l snd_l$

**definition** *pre-uvar*  $x = x ;_l des\_lens ;_l fst_l$

**lemma** *ind-uvar-semi-uvar*:  
 $semi\_uvar\ x \implies semi\_uvar\ (ind\_uvar\ i\ x)$   
**apply** (*unfold-locales*)  
**apply** (*simp-all add:ind-uvar-def*)  
**oops**

**syntax**

*-uprevar*  $:: ('t, '\alpha)\ uvar \Rightarrow logic\ (\$<- [0,999]\ 999)$   
*-udotvar*  $:: nat \Rightarrow ('t, '\alpha)\ uvar \Rightarrow logic\ (\&-.- [0,999]\ 999)$   
*-uidotvar*  $:: nat \Rightarrow ('t, '\alpha)\ uvar \Rightarrow logic\ (\$.-.- [0,999]\ 999)$

$-uodotvar :: nat \Rightarrow ('t, 'a) uvar \Rightarrow logic (\$.-' [999] 999)$   
 $-sdotvar :: nat \Rightarrow logic \Rightarrow svar (\&.- [0,999] 999)$   
 $-sin-dotvar :: nat \Rightarrow logic \Rightarrow svar (\$.-)$   
 $-sout-dotvar :: nat \Rightarrow logic \Rightarrow svar (\$.-')$

#### translations

$-uprevar x == CONST var (CONST in-var (CONST pre-uvar x))$   
 $-udotvar n x == CONST var (CONST ind-uvar n x)$   
 $-uidotvar n x == CONST var (CONST in-var (CONST ind-uvar n x))$   
 $-uidotvar n x == CONST var (CONST out-var (CONST ind-uvar n x))$   
 $-sdotvar n x == CONST ind-uvar n x$   
 $-sin-dotvar n x == CONST in-var (CONST ind-uvar n x)$   
 $-sout-dotvar n x == CONST out-var (CONST ind-uvar n x)$   
 $-psubst m (-sdotvar n x) v => CONST subst-upd m (CONST ind-uvar n x) v$

**type-synonym**  $'a \text{ merge} = ('a \text{ alphabet-d} \times 'a \text{ alphabet-d partition}, 'a) \text{ relation-d}$

**term**  $\$ok' =_u (\$0.ok \wedge \$1.ok)$

**term**  $(\$0.ok \wedge \$1.ok)$

#### Separating simulations

**lift-definition**  $sep\text{-}sim :: nat \Rightarrow ('a, ('a \text{ alphabet-d}) \text{ partition}) \text{ relation-d } (U'(-)) \text{ is}$   
 $\lambda n (A, A'). des\text{-}ok A' = des\text{-}ok A \wedge length (\alpha\text{-d.more } A') > n \wedge \alpha\text{-d.more } A' ! n = A .$

**lift-definition**  $\alpha\text{-ext} :: ('a, 'b) \text{ relation-d} \Rightarrow ('a, 'a \text{ alphabet-d} \times 'b) \text{ relation-d } (-_+ [999] 999) \text{ is}$   
 $\lambda P (A, A'). P (A, \langle des\text{-}ok = des\text{-}ok A', \dots = snd (more A') \rangle) \wedge des\text{-}ok A' = des\text{-}ok A \wedge fst (more A') = A .$

#### Parallel by merge

**term**  $((P ;; U(0)) \parallel (Q ;; U(1)))_+$

**definition**  $design\text{-}par\text{-}by\text{-}merge ::$

$'a \text{ hrelation-d} \Rightarrow 'a \text{ merge} \Rightarrow 'a \text{ hrelation-d} \Rightarrow 'a \text{ hrelation-d } (\mathbf{infixr} \parallel - 85)$   
**where**  $P \parallel_M Q = (((P ;; U(0)) \parallel (Q ;; U(1)))_+ ;; M)$

**end**

## 12 Reactive processes

**theory**  $utp\text{-}reactive$

**imports**

$utp\text{-}concurrency$

$utp\text{-}event$

**begin**

### 12.1 Preliminaries

**type-synonym**  $'a \text{ trace} = 'a \text{ list}$

**fun**  $list\text{-}diff :: 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list option}$  **where**

$list\text{-}diff l [] = Some l$   
 $| list\text{-}diff [] l = None$   
 $| list\text{-}diff (x \# xs) (y \# ys) = (if (x = y) then (list\text{-}diff xs ys) else None)$

**lemma** *list-diff-empty* [simp]: the (list-diff l []) = l  
**by** (cases l) auto

**lemma** *prefix-subst* [simp]:  $l @ t = m \implies m - l = t$   
**by** (auto)

**lemma** *prefix-subst1* [simp]:  $m = l @ t \implies m - l = t$   
**by** (auto)

The definitions of reactive process alphabets and healthiness conditions are given in the following. The healthiness conditions of reactive processes are defined by *R1*, *R2*, *R3* and their composition *R*.

**type-synonym** 'v refusal = 'v set

**record** 'v alpha-rp = alpha-d +  
                     rp-wait :: bool  
                     rp-tr  :: 'v trace  
                     rp-ref  :: 'v refusal

**definition** wait = VAR rp-wait

**definition** tr  = VAR rp-tr

**definition** ref = VAR rp-ref

**declare** wait-def [upred-defs]

**declare** tr-def  [upred-defs]

**declare** ref-def [upred-defs]

**lemma** *tr-ok-indep* [simp]:  $tr \bowtie ok \, ok \bowtie tr$   
**by** (simp add: lens-indep-def, pred-tac)+

**lemma** *wait-ok-indep* [simp]:  $wait \bowtie ok \, ok \bowtie wait$   
**by** (simp add: lens-indep-def, pred-tac)+

**lemma** *ref-ok-indep* [simp]:  $ref \bowtie ok \, ok \bowtie ref$   
**by** (simp add: lens-indep-def, pred-tac)+

**lemma** *tr-wait-indep* [simp]:  $tr \bowtie wait \, wait \bowtie tr$   
**by** (simp add: lens-indep-def, pred-tac)+

**lemma** *ref-wait-indep* [simp]:  $ref \bowtie wait \, wait \bowtie ref$   
**by** (simp add: lens-indep-def, pred-tac)+

**lemma** *tr-ref-indep* [simp]:  $ref \bowtie tr \, tr \bowtie ref$   
**by** (simp add: lens-indep-def, pred-tac)+

**term** put-vstore

**term** alpha-rp.more-update ( $\lambda\cdot$ . put-vstore x s)

**term** alpha-d.more

**term** alpha-rp.more-update

**term** alpha-d.extend

**instantiation** alpha-rp-ext :: (type, vst) vst  
**begin**

**definition** *get-vstore-alpha-rp-ext* :: ('a, 'b) *alpha-rp-ext*  $\Rightarrow$  *vstore*  
**where** [simp]: *get-vstore-alpha-rp-ext* *x* = *get-vstore* (*alpha-rp.more* (*alpha-d.extend undefined x*))  
**definition** *put-vstore-alpha-rp-ext* :: ('a, 'b) *alpha-rp-ext*  $\Rightarrow$  *vstore*  $\Rightarrow$  ('a, 'b) *alpha-rp-ext*  
**where** [simp]: *put-vstore-alpha-rp-ext* *s x* = *alpha-d.more* (*alpha-rp.more-update* ( $\lambda v.$  *put-vstore v x*)  
(*alpha-d.extend undefined s*))  
**instance**  
**apply** (*intro-classes*, *auto simp add: alpha-rp.defs alpha-d.defs*)  
**apply** (*metis alpha-d.select-convs*(2) *alpha-rp.select-convs*(4) *alpha-rp.surjective alpha-rp.update-convs*(4)  
*put-get-vstore*)  
**apply** (*metis (no-types, lifting) alpha-d.select-convs*(2) *alpha-rp.surjective alpha-rp.update-convs*(4)  
*get-put-vstore*)  
**apply** (*metis (no-types, lifting) alpha-d.select-convs*(2) *alpha-rp.surjective alpha-rp.update-convs*(4)  
*put-put-vstore*)  
**done**  
**end**

**lemma** *uvar-wait* [simp]: *uvar wait*  
**by** (*unfold-locales, simp-all add: wait-def*)

**lemma** *uvar-tr* [simp]: *uvar tr*  
**by** (*unfold-locales, simp-all add: tr-def*)

**lemma** *uvar-ref* [simp]: *uvar ref*  
**by** (*unfold-locales, simp-all add: ref-def*)

Note that we define here the class of UTP alphabets that contain *wait*, *tr* and *ref*, or, in other words, we define here the class of reactive process alphabets.

**type-synonym** (' $\vartheta$ , ' $\alpha$ ) *alphabet-rp* = (' $\vartheta$ , ' $\alpha$ ) *alpha-rp-scheme alphabet*  
**type-synonym** (' $\vartheta$ , ' $\alpha$ , ' $\beta$ ) *relation-rp* = ((' $\vartheta$ , ' $\alpha$ ) *alphabet-rp*, (' $\vartheta$ , ' $\beta$ ) *alphabet-rp*) *relation*  
**type-synonym** (' $\vartheta$ , ' $\alpha$ ) *hrelation-rp* = ((' $\vartheta$ , ' $\alpha$ ) *alphabet-rp*, (' $\vartheta$ , ' $\alpha$ ) *alphabet-rp*) *relation*  
**type-synonym** (' $\vartheta$ , ' $\sigma$ ) *predicate-rp* = (' $\vartheta$ , ' $\sigma$ ) *alphabet-rp upred*

**abbreviation** *wait-f*::(' $\vartheta$ , ' $\alpha$ , ' $\beta$ ) *relation-rp*  $\Rightarrow$  (' $\vartheta$ , ' $\alpha$ , ' $\beta$ ) *relation-rp* ( $-_f$  [1000] 1000)  
**where** *wait-f* *R*  $\equiv R \llbracket \text{false} / \$\text{wait} \rrbracket$

**abbreviation** *wait-t*::(' $\vartheta$ , ' $\alpha$ , ' $\beta$ ) *relation-rp*  $\Rightarrow$  (' $\vartheta$ , ' $\alpha$ , ' $\beta$ ) *relation-rp* ( $-_t$  [1000] 1000)  
**where** *wait-t* *R*  $\equiv R \llbracket \text{true} / \$\text{wait} \rrbracket$

**lift-definition** *lift-rea* :: (' $\alpha$ , ' $\beta$ ) *relation*  $\Rightarrow$  (' $\vartheta$ , ' $\alpha$ , ' $\beta$ ) *relation-rp* ( $[-]_R$ ) **is**  
 $\lambda P (A, A'). P \text{ (more } A, \text{ more } A')$  .

**lift-definition** *drop-rea* :: (' $\vartheta$ , ' $\alpha$ , ' $\beta$ ) *relation-rp*  $\Rightarrow$  (' $\alpha$ , ' $\beta$ ) *relation* ( $[-]_R$ ) **is**  
 $\lambda P (A, A'). P \text{ (} \llbracket \text{des-ok} = \text{True, rp-wait} = \text{True, rp-tr} = [], \text{rp-ref} = \{\}, \dots = A \rrbracket,$   
 $\llbracket \text{des-ok} = \text{True, rp-wait} = \text{True, rp-tr} = [], \text{rp-ref} = \{\}, \dots = A' \rrbracket \text{ )} .$

## 12.2 R1: Events cannot be undone

**definition** *R1-def* [*upred-defs*]: *R1* (*P*) = (*P*  $\wedge$  ( $\$tr \leq_u \$tr'$ ))

**lemma** *R1-idem*: *R1*(*R1*(*P*)) = *R1*(*P*)  
**by** *pred-tac*

**lemma** *R1-mono*: *P*  $\sqsubseteq$  *Q*  $\Longrightarrow$  *R1*(*P*)  $\sqsubseteq$  *R1*(*Q*)  
**by** *pred-tac*

**lemma** *R1-conj*:  $R1(P \wedge Q) = (R1(P) \wedge R1(Q))$   
 by *pred-tac*

**lemma** *R1-disj*:  $R1(P \vee Q) = (R1(P) \vee R1(Q))$   
 by *pred-tac*

**lemma** *R1-extend-conj*:  $R1(P \wedge Q) = (R1(P) \wedge Q)$   
 by *pred-tac*

**lemma** *R1-cond*:  $R1(P \triangleleft b \triangleright Q) = (R1(P) \triangleleft b \triangleright R1(Q))$   
 by *rel-tac*

**lemma** *R1-negate-R1*:  $R1(\neg R1(P)) = R1(\neg P)$   
 by *pred-tac*

**lemma** *R1-wait-true*:  $(R1\ P)_t = R1(P)_t$   
 by *pred-tac*

**lemma** *R1-wait-false*:  $(R1\ P)_f = R1(P)_f$   
 by *pred-tac*

**lemma** *R1-skip*:  $R1(II) = II$   
 by *rel-tac*

**lemma** *R1-by-refinement*:  
 $P \text{ is } R1 \longleftrightarrow ((\$tr \leq_u \$tr') \sqsubseteq P)$   
 by *rel-tac*

**lemma** *tr-le-trans*:  
 $(\$tr \leq_u \$tr' ;; \$tr \leq_u \$tr') = (\$tr \leq_u \$tr')$   
 by (*rel-tac*, *metis alpha-rp.select-convs(2) order-refl*)

**lemma** *R1-seqr-closure*:  
 assumes  $P \text{ is } R1$   $Q \text{ is } R1$   
 shows  $(P ;; Q) \text{ is } R1$   
 using *assms unfolding R1-by-refinement*  
 by (*metis seqr-mono tr-le-trans*)

**lemma** *R1-ok'-true*:  $(R1(P))^t = R1(P^t)$   
 by *pred-tac*

**lemma** *R1-ok'-false*:  $(R1(P))^f = R1(P^f)$   
 by *pred-tac*

**lemma** *R1-ok-true*:  $(R1(P))\llbracket true/\$ok \rrbracket = R1(P\llbracket true/\$ok \rrbracket)$   
 by *pred-tac*

**lemma** *R1-ok-false*:  $(R1(P))\llbracket false/\$ok \rrbracket = R1(P\llbracket false/\$ok \rrbracket)$   
 by *pred-tac*

**lemma** *seqr-R1-true-right*:  $((P ;; R1(true)) \vee P) = (P ;; (\$tr \leq_u \$tr'))$   
 by *rel-tac*

## 12.3 R2

**definition** *R2s-def* [*upred-defs*]:  $R2s\ (P) = (P\llbracket \langle \rangle / \$tr \rrbracket\llbracket (\$tr' - \$tr) / \$tr' \rrbracket)$



**definition** *R2-def* [*upred-defs*]:  $R2(P) = R1(R2s(P))$

**lemma** *R2s-idem*:  $R2s(R2s(P)) = R2s(P)$   
by (*pred-tac*)

**lemma** *R2-idem*:  $R2(R2(P)) = R2(P)$   
by (*pred-tac*)

**lemma** *R2-mono*:  $P \sqsubseteq Q \implies R2(P) \sqsubseteq R2(Q)$   
by (*pred-tac*)

**lemma** *R2s-conj*:  $R2s(P \wedge Q) = (R2s(P) \wedge R2s(Q))$   
by (*pred-tac*)

**lemma** *R2-conj*:  $R2(P \wedge Q) = (R2(P) \wedge R2(Q))$   
by (*pred-tac*)

**lemma** *R2s-condr*:  $R2s(P \triangleleft b \triangleright Q) = (R2s(P) \triangleleft R2s(b) \triangleright R2s(Q))$   
by (*rel-tac*)

**lemma** *R2-condr*:  $R2(P \triangleleft b \triangleright Q) = (R2(P) \triangleleft R2(b) \triangleright R2(Q))$   
by (*rel-tac*)

**lemma** *tr-prefix-as-concat*:  $(xs \leq_u ys) = (\exists zs \cdot ys =_u xs \hat{\ }_u \ll zs \gg)$   
by (*rel-tac*, *simp add: less-eq-list-def prefixeq-def*)

**lemma** *R2-form*:  
 $R2(P) = (\exists tt \cdot P[\langle \rangle / \$tr][\ll tt \gg / \$tr'] \wedge \$tr' =_u \$tr \hat{\ }_u \ll tt \gg)$   
by (*rel-tac*, *metis prefix-subst strict-prefixE*)

**lemma** *uconc-left-unit* [*simp*]:  $\langle \rangle \hat{\ }_u e = e$   
by (*pred-tac*)

**lemma** *uconc-right-unit* [*simp*]:  $e \hat{\ }_u \langle \rangle = e$   
by (*pred-tac*)

This laws is proven only for homogeneous relations, can it be generalised?

**lemma** *R2-seqr-form*:  
fixes  $P Q :: ('\theta, '\alpha, '\alpha) \text{ relation-rp}$   
shows  $(R2(P) ;; R2(Q)) =$   
 $(\exists tt_1 \cdot \exists tt_2 \cdot ((P[\langle \rangle / \$tr][\ll tt_1 \gg / \$tr'] ;; (Q[\langle \rangle / \$tr][\ll tt_2 \gg / \$tr'])))$   
 $\wedge (\$tr' =_u \$tr \hat{\ }_u \ll tt_1 \gg \hat{\ }_u \ll tt_2 \gg))$

**proof** –

have  $(R2(P) ;; R2(Q)) = (\exists tr_0 \cdot (R2(P))[\ll tr_0 \gg / \$tr'] ;; (R2(Q))[\ll tr_0 \gg / \$tr])$   
by (*subst seqr-middle[of tr], simp-all*)

also have ... =

$(\exists tr_0 \cdot \exists tt_1 \cdot \exists tt_2 \cdot ((P[\langle \rangle / \$tr][\ll tt_1 \gg / \$tr'] \wedge \ll tr_0 \gg =_u \$tr \hat{\ }_u \ll tt_1 \gg) ;;$   
 $(Q[\langle \rangle / \$tr][\ll tt_2 \gg / \$tr'] \wedge \$tr' =_u \ll tr_0 \gg \hat{\ }_u \ll tt_2 \gg)))$

by (*simp add: R2-form usubst unrest uquant-lift var-in-var var-out-var, rel-tac*)

also have ... =

$(\exists tr_0 \cdot \exists tt_1 \cdot \exists tt_2 \cdot ((\ll tr_0 \gg =_u \$tr \hat{\ }_u \ll tt_1 \gg \wedge P[\langle \rangle / \$tr][\ll tt_1 \gg / \$tr']) ;;$   
 $(Q[\langle \rangle / \$tr][\ll tt_2 \gg / \$tr'] \wedge \$tr' =_u \ll tr_0 \gg \hat{\ }_u \ll tt_2 \gg)))$

by (*simp add: conj-comm*)

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot \exists tr_0 \cdot ((P[\langle \rangle / \$tr][\ll tt_1 \gg / \$tr'] ;; (Q[\langle \rangle / \$tr][\ll tt_2 \gg / \$tr'])))$

$\wedge \langle tr_0 \rangle =_u \$tr \hat{ }_u \langle tt_1 \rangle \wedge \$tr' =_u \langle tr_0 \rangle \hat{ }_u \langle tt_2 \rangle$ )

by (*simp add: segr-pre-out segr-post-out unrest, rel-tac*)

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot ((P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] \;; (Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])))$   
 $\wedge (\exists tr_0 \cdot \langle tr_0 \rangle =_u \$tr \hat{ }_u \langle tt_1 \rangle \wedge \$tr' =_u \langle tr_0 \rangle \hat{ }_u \langle tt_2 \rangle))$

by *rel-tac*

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot ((P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] \;; (Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])))$   
 $\wedge (\$tr' =_u \$tr \hat{ }_u \langle tt_1 \rangle \hat{ }_u \langle tt_2 \rangle))$

by *rel-tac*

finally show *?thesis* .

qed

**lemma** *R2-segr-distribute*:

fixes  $P Q :: ('\vartheta, '\alpha, '\alpha)$  *relation-rp*

shows  $R2(R2(P) \;; R2(Q)) = (R2(P) \;; R2(Q))$

**proof** –

have  $R2(R2(P) \;; R2(Q)) =$

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] \;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])(\$tr' - \$tr) / \$tr')$   
 $\wedge \$tr' - \$tr =_u \langle tt_1 \rangle \hat{ }_u \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr)$

by (*simp add: R2-segr-form, simp add: R2s-def usubst unrest, rel-tac, blast+*)

also have ... =

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] \;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])(\langle tt_1 \rangle \hat{ }_u \langle tt_2 \rangle) / \$tr')$   
 $\wedge \$tr' - \$tr =_u \langle tt_1 \rangle \hat{ }_u \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr)$

by (*subst subst-eq-replace, simp*)

also have ... =

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] \;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])))$   
 $\wedge \$tr' - \$tr =_u \langle tt_1 \rangle \hat{ }_u \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr)$

by (*simp add: usubst unrest*)

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] \;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])))$   
 $\wedge (\$tr' - \$tr =_u \langle tt_1 \rangle \hat{ }_u \langle tt_2 \rangle \wedge \$tr' \geq_u \$tr)$

by *pred-tac*

also have ... =

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] \;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])))$   
 $\wedge \$tr' =_u \$tr \hat{ }_u \langle tt_1 \rangle \hat{ }_u \langle tt_2 \rangle))$

**proof** –

have  $\bigwedge tt_1 tt_2. (((\$tr' - \$tr =_u \langle tt_1 \rangle \hat{ }_u \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr) :: ('\vartheta, '\alpha, '\alpha)$  *relation-rp*)

$= (\$tr' =_u \$tr \hat{ }_u \langle tt_1 \rangle \hat{ }_u \langle tt_2 \rangle)$

by (*rel-tac, metis prefix-subst strict-prefixE*)

thus *?thesis* by *simp*

qed

also have ... =  $(R2(P) \;; R2(Q))$

by (*simp add: R2-segr-form*)

finally show *?thesis* .

qed

**lemma** *R1-R2-commute*:

$R1(R2(P)) = R2(R1(P))$

by *pred-tac*

## 12.4 R3

**definition** *skip-rea-def* [*urel-defs*]:  $II_r = (II \vee (\neg \$ok \wedge \$tr \leq_u \$tr'))$

**definition** *R3-def* [*upred-defs*]:  $R3(P) = (II \triangleleft \$wait \triangleright P)$

**definition** *R3c-def* [*upred-defs*]:  $R3c(P) = (II_r \triangleleft \$wait \triangleright P)$

**definition** *RH-def* [*upred-defs*]:  $RH(P) = R1(R2(R3c(P)))$

**lemma** *R3-idem*:  $R3(R3(P)) = R3(P)$   
by *rel-tac*

**lemma** *R3-mono*:  $P \sqsubseteq Q \implies R3(P) \sqsubseteq R3(Q)$   
by *rel-tac*

**lemma** *R3-conj*:  $R3(P \wedge Q) = (R3(P) \wedge R3(Q))$   
by *rel-tac*

**lemma** *R3-disj*:  $R3(P \vee Q) = (R3(P) \vee R3(Q))$   
by *rel-tac*

**lemma** *R3-condr*:  $R3(P \triangleleft b \triangleright Q) = (R3(P) \triangleleft b \triangleright R3(Q))$   
by *rel-tac*

**lemma** *R3-skipr*:  $R3(II) = II$   
by *rel-tac*

**lemma** *R3-form*:  $R3(P) = ((\$wait \wedge II) \vee (\neg \$wait \wedge P))$   
by *rel-tac*

**lemma** *R3-semir-form*:  
 $(R3(P) ;; R3(Q)) = R3(P ;; R3(Q))$   
by *rel-tac*

**lemma** *R3-semir-closure*:  
assumes *P is R3 Q is R3*  
shows  $(P ;; Q)$  is *R3*  
using *assms*  
by (*metis Healthy-def' R3-semir-form*)

**lemma** *R1-R3-commute*:  $R1(R3(P)) = R3(R1(P))$   
by *rel-tac*

**lemma** *R2-R3-commute*:  $R2(R3(P)) = R3(R2(P))$   
by (*rel-tac*, (*metis* (*no-types*, *lifting*) *alpha-rp.surjective alpha-rp.update-convs(2) append-Nil2 prefix-subst strict-prefixE*)+)

**lemma** *R2-R3c-commute*:  $R2(R3c(P)) = R3c(R2(P))$   
by (*rel-tac*, (*metis* (*no-types*, *lifting*) *alpha-rp.surjective alpha-rp.update-convs(2) append-Nil2 append-minus strict-prefixE*)+)

**lemma** *R3c-idem*:  $R3c(R3c(P)) = R3c(P)$   
by *rel-tac*

**lemma** *R1-skip-rea*:  $R1(II_r) = II_r$   
by *rel-tac*

**lemma** *R2-skip-rea*:  $R2(II_r) = II_r$   
apply (*rel-tac*)

```

apply (metis (no-types, lifting) alpha-rp.surjective alpha-rp.update-convs(2) append-Nil2 prefix-subst
strict-prefixE)
done

end

```