

A Shallow Model of the UTP in Isabelle/HOL

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

March 2, 2016

Contents

1	UTP variables	2
1.1	Deep UTP variables	6
1.2	Cardinalities	6
1.3	Injection functions	7
1.4	Deep variables	9
2	UTP expressions	12
2.1	Evaluation laws for expressions	18
3	Unrestriction	18
4	Substitution	20
4.1	Substitution definitions	20
4.2	Substitution laws	21
5	Lifting expressions	23
5.1	Lifting definitions	23
5.2	Lifting laws	24
6	Alphabetised Predicates	25
6.1	Predicate syntax	25
6.2	Predicate operators	26
6.3	Proof support	28
6.4	Unrestriction Laws	29
6.5	Substitution Laws	30
6.6	Predicate Laws	31
6.7	Quantifier lifting	34
7	Alphabetised relations	35
7.1	Unrestriction Laws	36
7.2	Substitution laws	37
7.3	Lifting laws	38
7.4	Relation laws	39
7.5	Converse laws	42
7.6	Weakest precondition calculus	45
8	UTP Theories	45

9 Example UTP theory: Boyle's laws	46
10 Designs	47
10.1 Definitions	47
10.2 Design laws	49
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
11 Concurrent programming	59
11.1 Design parallel composition	59
11.2 Parallel by merge	60
12 Reactive processes	61
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/Library-extra/Map-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

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.

record ($'a$, $'\alpha$) *uvar* =
var-lookup :: $'\alpha \Rightarrow 'a$
var-update :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

The *var-assign* function uses the *var-update* function of a variable to update its value.

abbreviation $\text{var-assign} :: ('a, 'α) \text{uvar} \Rightarrow 'a \Rightarrow 'α \Rightarrow 'α$
where $\text{var-assign } f \ v \equiv \text{var-update } f \ (\lambda - . v)$

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.

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

translations $\text{VAR } x \Rightarrow \llbracket \text{var-lookup} = x, \text{var-update} = \text{-update-name } x \rrbracket$

In order to allow reasoning about variables generically, we introduce a locale called *uvar*, that axiomatises properties of a valid variable, that should be satisfied for any record field. When a UTP alphabet record is created it will be necessary to prove these properties for each variable field, though this will always be automatic. The locale effectively describes the relationship between the functions *var-update* and *var-lookup*, and thus prevents one from having arbitrary functions as variables. Moreover, these properties allow us to prove several important UTP laws, such as the assignment laws in the theory of alphabetised relations.

locale *semi-uvar* =

fixes $x :: ('a, 'r) \text{uvar}$

— Application of two updates should correspond to the composition of update functions

assumes *var-update-comp*: $\text{var-update } x \ f \ (\text{var-update } x \ g \ \sigma) = \text{var-update } x \ (f \circ g) \ \sigma$

— Updating a variable's value to the one it already has is ineffectual

and *var-update-eta*: $\text{var-update } x \ (\lambda -. \text{var-lookup } x \ \sigma) \ \sigma = \sigma$

locale *uvar* = *semi-uvar* +

assumes *var-update-lookup*: $\text{var-lookup } x \ (\text{var-update } x \ f \ \sigma) = f \ (\text{var-lookup } x \ \sigma)$

and *var-state-eq-iff*: $\llbracket \text{var-lookup } x \ \sigma = \text{var-lookup } x \ \varrho; \text{var-assign } x \ v \ \sigma = \text{var-assign } x \ v \ \varrho \rrbracket \Longrightarrow \sigma = \varrho$

declare *semi-uvar.var-update-comp* [simp]

declare *uvar.var-update-lookup* [simp]

declare *semi-uvar.var-update-eta* [simp]

lemma *var-assign-inject*: $\llbracket \text{uvar } x; \text{var-assign } x \ u \ \sigma = \text{var-assign } x \ v \ \sigma \rrbracket \Longrightarrow u = v$
by (*metis uvar.var-update-lookup*)

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

lemma *var-assign-eq*:

$\llbracket \text{uvar } x; \text{var-lookup } x \ b = k; \text{var-assign } x \ u \ b = \text{var-assign } x \ v \ a \rrbracket \Longrightarrow \text{var-assign } x \ k \ a = b$

apply (*rule uvar.var-state-eq-iff [of x - v]*)

apply (*simp-all add: comp-def*)

apply (*metis uvar.var-update-lookup*)

done

In addition to defining the validity of variable, we also need to show how two variables are related. Since variables are pairs of functions and have no identifying name that we can reason about, and moreover will often have different types, we cannot use the usual HOL inequalities to reason about them. Thus we define a weaker notion of inequality called *independence* – two variables are independent if their update functions commute. That is to say, updates to the variables do not have any effect on each other. This assumes they are also valid variables.

definition *uvar-indep* :: $('a, 'r) \text{uvar} \Rightarrow ('b, 'r) \text{uvar} \Rightarrow \text{bool}$ (**infix** \bowtie 50) **where**

$x \bowtie y \longleftrightarrow (\forall f \ g \ \sigma. \text{var-update } x \ f \ (\text{var-update } y \ g \ \sigma) = \text{var-update } y \ g \ (\text{var-update } x \ f \ \sigma))$

We can now demonstrate some useful properties about the variable independence relation.

lemma *uvar-indep-sym*: $x \bowtie y \implies y \bowtie x$
by (*simp add: uvar-indep-def*)

lemma *uvar-indep-comm*:
assumes $x \bowtie y$
shows $\text{var-update } x \ f \ (\text{var-update } y \ g \ \sigma) = \text{var-update } y \ g \ (\text{var-update } x \ f \ \sigma)$
using *assms* **by** (*simp add: uvar-indep-def*)

The following property states that looking up the value of a variable is unaffected by an update to an independent variable.

lemma *uvar-indep-lookup-upd* [*simp*]:
assumes $\text{uvar } x \bowtie y$
shows $\text{var-lookup } x \ (\text{var-update } y \ f \ \sigma) = \text{var-lookup } x \ \sigma$
proof –
have $\text{var-lookup } x \ (\text{var-update } y \ f \ \sigma) = \text{var-lookup } x \ (\text{var-update } y \ f \ (\text{var-update } x \ (\lambda-. \text{var-lookup } x \ \sigma) \ \sigma))$
by (*simp add: assms(1)*)
also have $\dots = \text{var-lookup } x \ (\text{var-update } x \ (\lambda-. \text{var-lookup } x \ \sigma) \ (\text{var-update } y \ f \ \sigma))$
using *assms(2)* **by** (*auto simp add: uvar-indep-def*)
also have $\dots = \text{var-lookup } x \ \sigma$
by (*simp add: assms(1)*)
finally show ?thesis .
qed

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* :: $('a, 'a) \text{uvar} \Rightarrow ('a, 'a \times 'a) \text{uvar}$ **where**
 $\text{in-var } x = \langle \text{var-lookup} = \text{var-lookup } x \circ \text{fst}, \text{var-update} = (\lambda f \ (A, A'). (\text{var-update } x \ f \ A, A')) \rangle$

definition *out-var* :: $('a, 'a) \text{uvar} \Rightarrow ('a, 'a \times 'a) \text{uvar}$ **where**
 $\text{out-var } x = \langle \text{var-lookup} = \text{var-lookup } x \circ \text{snd}, \text{var-update} = (\lambda f \ (A, A'). (A, \text{var-update } x \ f \ A')) \rangle$

We show that lifted input and output variables are both valid variables, and that input and output variables are always independent.

lemma *in-var-semi-uvar* [*simp*]:
assumes *semi-uvar* x
shows *semi-uvar* $(\text{in-var } x)$
using *assms*
by (*unfold-locales, auto simp add: in-var-def*)

lemma *out-var-semi-uvar* [*simp*]:
assumes *semi-uvar* x
shows *semi-uvar* $(\text{out-var } x)$
using *assms*
by (*unfold-locales, auto simp add: out-var-def*)

lemma *in-var-uvar* [*simp*]:
assumes *uvar* x
shows *uvar* $(\text{in-var } x)$
using *assms*
by (*unfold-locales, auto intro: uvar.var-state-eq-iff simp add: in-var-def*)

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

```

assumes uvar x
shows uvar (out-var x)
using assms
by (unfold-locales, auto intro: uvar.var-state-eq-iff simp add: out-var-def)

```

```

lemma in-out-indep [simp]:
  in-var x  $\bowtie$  out-var y
by (simp add: uvar-indep-def in-var-def out-var-def)

```

```

lemma out-in-indep [simp]:
  out-var x  $\bowtie$  in-var y
by (simp add: uvar-indep-def in-var-def out-var-def)

```

```

lemma in-var-indep [simp]:
  x  $\bowtie$  y  $\implies$  in-var x  $\bowtie$  in-var y
by (simp add: uvar-indep-def in-var-def)

```

```

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

```

We also define some lookup abstraction simplifications.

```

lemma var-lookup-in [simp]: var-lookup (in-var x) (A, A') = var-lookup x A
by (simp add: in-var-def)

```

```

lemma var-lookup-out [simp]: var-lookup (out-var x) (A, A') = var-lookup x A'
by (simp add: out-var-def)

```

```

lemma var-update-in [simp]: var-update (in-var x) f (A, A') = (var-update x f A, A')
by (simp add: in-var-def)

```

```

lemma var-update-out [simp]: var-update (out-var x) f (A, A') = (A, var-update x f A')
by (simp add: out-var-def)

```

Variables can also be used to effectively define sets of variables. Here we define the the universal alphabet (Σ) 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 = ( $\lambda$  var-lookup = id, var-update = id)

```

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  $\circ_v$  35) where
uvar-comp x y = ( $\lambda$  var-lookup =  $\lambda$  A. (var-lookup x A, var-lookup y A)
  , var-update =  $\lambda$  f. var-update x ( $\lambda$  a. fst (f (a, undefined)))  $\circ$ 
  var-update y ( $\lambda$  b. snd (f (undefined, b)))  $\lambda$ )

```

nonterminal *svar*

```

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

```

```

-spvar  :: id ⇒ svar (&- [999] 999)
-sinvar :: id ⇒ svar ($- [999] 999)
-soutvar :: id ⇒ svar ($-' [999] 999)

```

consts

```

svar :: 'v ⇒ 'e
ivar :: 'v ⇒ 'e
ovar :: 'v ⇒ 'e

```

ad hoc-overloading

```

ivar in-var and over 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 (ℵ₀) | cont (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
 $\mathcal{U}(\text{fin } n) = \{\{x\} \mid x. x \leq n\} \mid$ 
 $\mathcal{U}(\aleph_0) = \{\{x\} \mid x. \text{True}\} \mid$ 
 $\mathcal{U}(c) = \text{UNIV}$ 
```

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 c)
```

syntax

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

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]:

```
 $\llbracket \text{countable}(UNIV :: 'a::continuum \text{ set}); \text{infinite}(UNIV :: 'a \text{ set}) \rrbracket \Longrightarrow UCARD('a) = \aleph_0$ 
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*) \implies *uinject* = ($\lambda x :: 'a. \{to-nat-fin\ x\}$)
by (*rule ext, auto simp add: uinject-def*)

lemma *uinject-uncountable*:
uncountable (*UNIV* :: *'a::continuum set*) \implies (*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 $x < \text{card } (UNIV :: 'a \text{ set}) \longleftrightarrow x \leq \text{card } (UNIV :: 'a \text{ set}) - \text{Suc } 0$
proof –
have $\text{card } (UNIV :: 'a \text{ set}) > 0$
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]*:
 $x \in \mathcal{U}(UCARD('a::\text{continuum})) \implies \text{uinject } ((\text{uproject} :: \text{nat set} \Rightarrow 'a) \ x) = x$
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* = $\{f :: \text{dname} \Rightarrow \text{univ}. \forall x. f(x) \in \mathcal{U}(\text{dname-card } x)\}$
apply (*rule-tac* $x = \lambda x. \{0\}$ **in** *exI*)
apply (*auto*)
apply (*rename-tac* *x*)
apply (*case-tac* *dname-card* *x*)
apply (*simp-all*)
done

setup-lifting *type-definition-vstore*

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

setup-lifting *type-definition-dvar*

lift-definition *mk-dvar* :: $\text{string} \Rightarrow ('a::\text{continuum}) \text{dvar} ([_]_d)$
is $\lambda n. ([_]_{\text{dname-name} = n, \text{dname-card} = UCARD('a)})$
by *auto*

lift-definition *dvar-name* :: $'a::\text{continuum} \text{dvar} \Rightarrow \text{string}$ **is** *dname-name* .

lift-definition *dvar-card* :: $'a::\text{continuum} \text{dvar} \Rightarrow \text{ucard}$ **is** *dname-card* .

lemma *dvar-name [simp]*: $\text{dvar-name } [x]_d = x$
by (*transfer*, *simp*)

lift-definition *vstore-lookup* :: $('a::\text{continuum}) \text{dvar} \Rightarrow \text{vstore} \Rightarrow 'a$
is $\lambda x s. (\text{uproject} :: \text{univ} \Rightarrow 'a) (s(x))$.

lift-definition *vstore-put* :: $('a::\text{continuum}) \text{dvar} \Rightarrow 'a \Rightarrow \text{vstore} \Rightarrow \text{vstore}$
is $\lambda (x :: \text{dname}) (v :: 'a) f . f(x := \text{uinject } v)$
by (*auto*)

definition *vstore-upd* :: $('a::\text{continuum}) \text{dvar} \Rightarrow ('a \Rightarrow 'a) \Rightarrow \text{vstore} \Rightarrow \text{vstore}$
where *vstore-upd* $x f s = \text{vstore-put } x (f (\text{vstore-lookup } x s)) s$

lemma *vstore-upd-comp [simp]*:
 $\text{vstore-upd } x f (\text{vstore-upd } x g s) = \text{vstore-upd } x (f \circ g) s$
by (*simp add: vstore-upd-def, transfer, simp*)

lemma *vstore-lookup-upd [simp]*: $\text{vstore-lookup } x (\text{vstore-upd } x f s) = f (\text{vstore-lookup } x s)$
by (*simp add: vstore-upd-def, transfer, simp*)

```

lemma vstore-upd-eta [simp]: vstore-upd x ( $\lambda$  -. 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  $\neq$  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  $\neq$  dvar-name y
  shows vstore-put x u (vstore-put y v s) = vstore-put y v (vstore-put x u s)
  using assms
  by (transfer, fastforce)

```

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 upd-vstore :: (vstore  $\Rightarrow$  vstore)  $\Rightarrow$  'a  $\Rightarrow$  'a
  assumes get-upd-vstore [simp]: get-vstore (upd-vstore f s) = f (get-vstore s)
  and upd-vstore-comp [simp]: upd-vstore f (upd-vstore g s) = upd-vstore (f  $\circ$  g) s
  and upd-vstore-eta [simp]: upd-vstore ( $\lambda$  -. get-vstore s) s = s
  and upd-store-param: upd-vstore f s = upd-vstore ( $\lambda$  -. f (get-vstore s)) s

```

```

definition dvar-lift :: 'a::continuum dvar  $\Rightarrow$  ('a, 'a::vst) uvar ( $\neg$  [999] 999)
where dvar-lift x = ( $\mid$  var-lookup =  $\lambda$  v. vstore-lookup x (get-vstore v)
  , var-update =  $\lambda$  f s. upd-vstore (vstore-upd x f) s
   $\mid$ )

```

```

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

```

adhoc-overloading

```

ivar in-dvar and ovar out-dvar

```

```

lemma vstore-upd-compose [simp]: vstore-upd x f  $\circ$  vstore-upd x g = vstore-upd x (f  $\circ$  g)
  by (rule ext, simp add: vstore-upd-def, transfer, auto)

```

```

lemma update-vstore-appl: upd-vstore ( $\lambda$  x. f (g x)) s = upd-vstore f (upd-vstore g s)
  by (metis comp-apply upd-store-param upd-vstore-comp)

```

```

lemma vstore-upd-appl: vstore-upd x ( $\lambda$  x. f (g x)) s = vstore-upd x f (vstore-upd x g s)
  by (metis (no-types, lifting) vstore-lookup-upd vstore-upd-comp vstore-upd-def)

```

```

lemma uvar-dvar: uvar (x $\uparrow$ )

```

proof

```

  fix f g :: 'a  $\Rightarrow$  'a and  $\sigma$  :: 'b
  show var-update (x $\uparrow$ ) f (var-update (x $\uparrow$ ) g  $\sigma$ ) = var-update (x $\uparrow$ ) (f  $\circ$  g)  $\sigma$ 
    by (simp add: dvar-lift-def)
  show var-assign (x $\uparrow$ ) (var-lookup (x $\uparrow$ )  $\sigma$ )  $\sigma$  =  $\sigma$ 
    by (simp add: dvar-lift-def, subst upd-store-param, simp)

```

```

show var-lookup (x↑) (var-update (x↑) f σ) = f (var-lookup (x↑) σ)
  by (simp add: dvar-lift-def)
fix ρ :: 'b and v :: 'a
show var-lookup (x↑) σ = var-lookup (x↑) ρ  $\implies$  var-assign (x↑) v σ = var-assign (x↑) v ρ  $\implies$  σ =
ρ
proof –
  assume vl: var-lookup (x↑) σ = var-lookup (x↑) ρ and va: var-assign (x↑) v σ = var-assign (x↑) v
ρ
  have get-vstore σ = get-vstore ρ
    by (metis (no-types, lifting) dvar-lift-def get-upd-vstore uvar.select-convs(1) uvar.select-convs(2))
  va vl vstore-lookup-upd vstore-upd-comp vstore-upd-def vstore-upd-eta)

  moreover from va have upd-vstore (λ-. get-vstore ρ) σ = ρ
    by (simp add: dvar-lift-def, metis upd-store-parm upd-vstore-eta update-vstore-appl)

  ultimately show ?thesis
    by (metis upd-vstore-eta)
qed
qed

```

Deep variables with different names are independent

lemma dvar-indep-diff-name:

assumes dvar-name x ≠ dvar-name y

shows x↑ \bowtie y↑

proof –

from assms **have** $\bigwedge f g. \text{vstore-upd } x f \circ \text{vstore-upd } y g = \text{vstore-upd } y g \circ \text{vstore-upd } x f$

apply (auto simp add: comp-def vstore-upd-def)

apply (rule ext, subst vstore-put-commute, auto)

done

thus ?thesis

by (auto simp add: uvar-indep-def dvar-name-def dvar-card-def dvar-lift-def vstore-upd-def)

qed

lemma dvar-indep-diff-name' [simp]:

$x \neq y \implies [x]_d \uparrow \bowtie [y]_d \uparrow$

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]: get-vstore-vstore-d-ext = vstore

definition [simp]: upd-vstore-vstore-d-ext = vstore-update

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 ⇒ ('t, 'α × '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 ⇒ ('t, 'α) uexpr
is λ 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

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

adhoc-overloading

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

syntax

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

translations

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

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 $plus\text{-}uexpr\text{-}def: u + v = bop \ (op \ +) \ u \ v$

instance ..

end

Instantiating uminus also provides negation for predicates later

instantiation $uexpr :: (uminus, type) uminus$

begin

```

definition uminus-uepr-def:  $- u = uop\ uminus\ u$ 
instance ..
end

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

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

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

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

instantiation uepr :: (one, type) one
begin
  definition one-uepr-def:  $1 = lit\ 1$ 
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 uepr :: (semigroup-add, type) semigroup-add
  by (intro-classes) (simp add: plus-uepr-def zero-uepr-def, transfer, simp add: add.assoc) +

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

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

Set up automation for numerals

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

```

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

definition *eq-upred* :: $('a, 'α) \text{ uepr} \Rightarrow ('a, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$
where *eq-upred* $x \ y = \text{bop HOL.eq } x \ y$

adhoc-overloading

ueq eq-upred

abbreviation *seq-filter* :: $'a \text{ set} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
seq-filter $A \equiv \text{filter } (\lambda x. x \in A)$

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

syntax

-ucoerce :: $('a, 'α) \text{ uepr} \Rightarrow \text{type} \Rightarrow ('a, 'α) \text{ uepr}$ (**infix** :_u 50)
-unil :: $('a \text{ list}, 'α) \text{ uepr}$ ($\langle \rangle$)
-ulist :: $\text{args} \Rightarrow ('a \text{ list}, 'α) \text{ uepr}$ ($\langle (-) \rangle$)
-uappend :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow ('a \text{ list}, 'α) \text{ uepr} \Rightarrow ('a \text{ list}, 'α) \text{ uepr}$ (**infixr** ^_u 80)
-ulast :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow ('a, 'α) \text{ uepr}$ ($\text{last}_u '(-)$)
-ufront :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow ('a, 'α) \text{ uepr}$ ($\text{front}_u '(-)$)
-uhead :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow ('a, 'α) \text{ uepr}$ ($\text{head}_u '(-)$)
-utail :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow ('a \text{ list}, 'α) \text{ uepr}$ ($\text{tail}_u '(-)$)
-ulength :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow (\text{nat}, 'α) \text{ uepr}$ ($\text{length}_u '(-)$)
-ufilter :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr} \Rightarrow ('a \text{ list}, 'α) \text{ uepr}$ (**infixl** |_u 75)
-uelems :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr}$ ($\text{elems}_u '(-)$)
-usorted :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ ($\text{sorted}_u '(-)$)
-udistinct :: $('a \text{ list}, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ ($\text{distinct}_u '(-)$)
-uless :: $('a, 'α) \text{ uepr} \Rightarrow ('a, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ (**infix** <_u 50)
-uless :: $('a, 'α) \text{ uepr} \Rightarrow ('a, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ (**infix** ≤_u 50)
-ugreat :: $('a, 'α) \text{ uepr} \Rightarrow ('a, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ (**infix** >_u 50)
-ugeq :: $('a, 'α) \text{ uepr} \Rightarrow ('a, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ (**infix** ≥_u 50)
-uempset :: $('a \text{ set}, 'α) \text{ uepr}$ ($\{\}_u$)
-uset :: $\text{args} \Rightarrow ('a \text{ set}, 'α) \text{ uepr}$ ($\{(-)\}_u$)
-uunion :: $('a \text{ set}, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr}$ (**infixl** ∪_u 65)
-uinter :: $('a \text{ set}, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr}$ (**infixl** ∩_u 70)
-umem :: $('a, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ (**infix** ∈_u 50)
-unmem :: $('a, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ (**infix** ∉_u 50)
-usubset :: $('a \text{ set}, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ (**infix** ⊆_u 50)
-usubseteq :: $('a \text{ set}, 'α) \text{ uepr} \Rightarrow ('a \text{ set}, 'α) \text{ uepr} \Rightarrow (\text{bool}, 'α) \text{ uepr}$ (**infix** ⊆_u 50)
-utuple :: $('a, 'α) \text{ uepr} \Rightarrow \text{utuple-args} \Rightarrow ('a * 'b, 'α) \text{ uepr}$ ($(1'(-)/ -)_u$)
-utuple-arg :: $('a, 'α) \text{ uepr} \Rightarrow \text{utuple-args } (-)$
-utuple-args :: $('a, 'α) \text{ uepr} \Rightarrow \text{utuple-args} \Rightarrow \text{utuple-args}$ $(-, / -)$
-uunit :: $('a, 'α) \text{ uepr}$ ($(')_u$)
-ufst :: $('a \times 'b, 'α) \text{ uepr} \Rightarrow ('a, 'α) \text{ uepr}$ ($\pi_1 '(-)$ 90)
-usnd :: $('a \times 'b, 'α) \text{ uepr} \Rightarrow ('b, 'α) \text{ uepr}$ ($\pi_2 '(-)$)
-uapply :: $('a \Rightarrow 'b, 'α) \text{ uepr} \Rightarrow \text{utuple-args} \Rightarrow ('b, 'α) \text{ uepr}$ $(-\langle - \rangle)_u [999, 0] 999$)
-ulambda :: $\text{pttrn} \Rightarrow \text{logic} \Rightarrow \text{logic}$ $(\lambda - \cdot - [0, 10] 10)$
-udom :: $\text{logic} \Rightarrow \text{logic}$ ($\text{dom}_u '(-)$)
-uran :: $\text{logic} \Rightarrow \text{logic}$ ($\text{ran}_u '(-)$)
-uinl :: $\text{logic} \Rightarrow \text{logic}$ ($\text{inl}_u '(-)$)
-uinr :: $\text{logic} \Rightarrow \text{logic}$ ($\text{inr}_u '(-)$)
-umap-empty :: $('a \rightarrow 'b, 'α) \text{ uepr}$ ($\llbracket \rrbracket_u$)
-umap-apply :: $\text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic}$ $(-\langle - \rangle)_m [999, 0] 999$)
-umap-plus :: $\text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic}$ (**infixl** ⊕_m 85)

```

-umap-minus :: logic ⇒ logic ⇒ logic (infixl  $\ominus_m$  85)
-umap-subseteq :: logic ⇒ logic ⇒ logic (infixl  $\subseteq_M$  85)
-umaplet    :: [logic, logic] => umaplet (- / $\mapsto_u$ / -)
              :: umaplet => umaplets      (-)
-UMaplets   :: [umaplet, umaplets] => umaplets (-,/ -)
-UMapUpd    :: [logic, umaplets] => logic (-/'(-) [900,0] 900)
-UMap       :: umaplets => logic ((1[-]))

```

consts *uapply* :: 'f ⇒ 'k ⇒ 'v

definition *fun-apply* f x = f x

declare *fun-apply-def* [simp]

definition *map-upd* = (λ f x v. *fun-upd* f x (Some v))

definition *map-empty* :: 'a \rightarrow 'b ($[]_m$) **where**

map-empty = Map.empty

ad hoc-overloading

uapply *fun-apply* **and** *uapply* *nth*

translations

```

x :u 'a == x :: ('a, -) uexpr
⟨⟩      == <<[]>>
⟨x, xs⟩ == CONST bop (op #) x ⟨xs⟩
⟨x⟩     == CONST bop (op #) x <<[]>>
x ^u y  == CONST bop (op @) x y
lastu(xs) == CONST uop CONST last xs
frontu(xs) == CONST uop CONST butlast xs
headu(xs) == CONST uop CONST hd xs
tailu(xs) == CONST uop CONST tl xs
lengthu(xs) == CONST uop CONST length xs
elemsu(xs) == CONST uop CONST set xs
sortedu(xs) == CONST uop CONST sorted xs
distinctu(xs) == CONST uop CONST distinct xs
xs  $\downarrow_u$  A == CONST bop CONST seq-filter A xs
x <u y == CONST bop (op <) x y
x ≤u y == CONST bop (op ≤) x y
x >u y == y <u x
x ≥u y == y ≤u x
{ }u == <<{ }>>
{x, xs}u == CONST bop (CONST insert) x {xs}u
{x}u == CONST bop (CONST insert) x <<{ }>>
A  $\cup_u$  B == CONST bop (op  $\cup$ ) A B
A  $\cap_u$  B == CONST bop (op  $\cap$ ) A B
x  $\in_u$  A == CONST bop (op  $\in$ ) x A
x  $\notin_u$  A == CONST bop (op  $\notin$ ) x A
A  $\subset_u$  B == CONST bop (op  $\subset$ ) A B
A  $\subseteq_u$  B == CONST bop (op  $\subseteq$ ) A B
()u == <<()>>
(x, y)u == CONST bop (CONST Pair) x y
-utuple x (-utuple-args y z) == -utuple x (-utuple-arg (-utuple y z))
 $\pi_1$ (x) == CONST uop CONST fst x

```


$\pi_2(x) == \text{CONST } \text{uop } \text{CONST } \text{snd } x$
 $f(\downarrow x)_u == \text{CONST } \text{bop } \text{CONST } \text{uapply } f \ x$
 $\lambda x \cdot p == \text{CONST } \text{ulambda } (\lambda x. p)$
 $\text{dom}_u(f) == \text{CONST } \text{uop } \text{CONST } \text{dom } f$
 $\text{ran}_u(f) == \text{CONST } \text{uop } \text{CONST } \text{ran } f$
 $\text{inl}_u(x) == \text{CONST } \text{uop } \text{CONST } \text{Inl } x$
 $\text{inr}_u(x) == \text{CONST } \text{uop } \text{CONST } \text{Inr } x$
 $f(\downarrow x)_m == \text{CONST } \text{bop } \text{CONST } \text{map-apply } f \ x$
 $f \oplus_m g == \text{CONST } \text{bop } \text{CONST } \text{map-add } f \ g$
 $f \ominus_m g == \text{CONST } \text{bop } \text{CONST } \text{map-minus } f \ g$
 $f \subseteq_M g == \text{CONST } \text{bop } \text{op } \subseteq_m f \ g$
 $\downarrow_u == \ll \text{CONST } \text{map-empty} \gg$
 $\text{-UMapUpd } m \ (\text{-UMaplets } xy \ ms) == \text{-UMapUpd } (\text{-UMapUpd } m \ xy) \ ms$
 $\text{-UMapUpd } m \ (\text{-umaplet } x \ y) == \text{CONST } \text{trop } \text{CONST } \text{map-upd } m \ x \ y$
 $\text{-UMap } ms == \text{-UMapUpd } \downarrow_u \ ms$
 $\text{-UMap } (\text{-UMaplets } ms1 \ ms2) \leq \text{-UMapUpd } (\text{-UMap } ms1) \ ms2$
 $\text{-UMaplets } ms1 \ (\text{-UMaplets } ms2 \ ms3) \leq \text{-UMaplets } (\text{-UMaplets } ms1 \ ms2) \ ms3$
 $f(\downarrow x, y)_u == \text{CONST } \text{bop } \text{CONST } \text{fun-apply } f \ (x, y)_u$

Lifting set intervals

syntax

$\text{-uset-atLeastAtMost} :: ('a, 'α) \text{uepr} \Rightarrow ('a, 'α) \text{uepr} \Rightarrow ('a \text{ set}, 'α) \text{uepr} ((1\{-..\}_{u}))$
 $\text{-uset-atLeastLessThan} :: ('a, 'α) \text{uepr} \Rightarrow ('a, 'α) \text{uepr} \Rightarrow ('a \text{ set}, 'α) \text{uepr} ((1\{-..<\}_{u}))$
 $\text{-uset-compr} :: id \Rightarrow ('a \text{ set}, 'α) \text{uepr} \Rightarrow (bool, 'α) \text{uepr} \Rightarrow ('b, 'α) \text{uepr} \Rightarrow ('b \text{ set}, 'α) \text{uepr} ((1\{-\cdot\}_{u}))$

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

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

translations

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

Lifting limits

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

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

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

syntax

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

translations

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

lemmas $\text{uepr-defs} =$

iuvar-def
 ouvar-def
 zero-uepr-def
 one-uepr-def

```

plus-uepr-def
uminus-uepr-def
minus-uepr-def
times-uepr-def
div-uepr-def
mod-uepr-def
eq-upred-def
numeral-uepr-simp
map-empty-def
map-upd-def
ulim-left-def
ulim-right-def
ucont-on-def

```

```

lemma var-in-var: var (in-var x) = $x
by (simp add: iuvar-def)

```

```

lemma var-out-var: var (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*

lift-definition $unrest-upred :: ('a, 'α) uvar \Rightarrow ('b, 'α) uexpr \Rightarrow bool$

is $\lambda x\ e.\ \forall\ b\ v.\ e\ (var-update\ x\ v\ b) = e\ b$.

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

ad hoc-overloading

$unrest\ unrest-upred$

lemma $unrest-lit\ [unrest]:\ x\ \# \ll v \gg$

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 $unrest-var\ [unrest]:\ \ll uvar\ x;\ x\ \bowtie\ y \gg \Longrightarrow y\ \# var\ x$

by (*transfer, auto*)

lemma $unrest-iuvar\ [unrest]:\ \ll uvar\ x;\ x\ \bowtie\ y \gg \Longrightarrow \$y\ \# \$x$

by (*metis in-var-indep in-var-uvar unrest-var var-in-var*)

lemma $unrest-ouvar\ [unrest]:\ \ll uvar\ x;\ x\ \bowtie\ y \gg \Longrightarrow \$y'\ \# \$x'$

by (*metis out-var-indep out-var-uvar unrest-var var-out-var*)

lemma $unrest-iuvar-ouvar\ [unrest]:$

fixes $x :: ('a, 'α)\ uvar$

assumes $uvar\ y$

shows $\$x\ \# \y'

by (*metis assms out-in-indep out-var-uvar unrest-var var-out-var*)

lemma $unrest-ouvar-iuvar\ [unrest]:$

fixes $x :: ('a, 'α)\ uvar$

assumes $uvar\ y$

shows $\$x'\ \# \y

by (*metis assms in-out-indep in-var-uvar unrest-var var-in-var*)

lemma $unrest-uop\ [unrest]:\ x\ \# e \Longrightarrow x\ \# uop\ f\ e$

by (*transfer, simp*)

lemma $unrest-bop\ [unrest]:\ \ll x\ \# u;\ x\ \# v \gg \Longrightarrow x\ \# bop\ f\ u\ v$

by (*transfer, simp*)

lemma $unrest-trop\ [unrest]:\ \ll x\ \# u;\ x\ \# v;\ x\ \# w \gg \Longrightarrow x\ \# trop\ f\ u\ v\ w$

by (*transfer, simp*)

lemma $unrest-eq\ [unrest]:\ \ll x\ \# u;\ x\ \# v \gg \Longrightarrow x\ \# u =_u v$

by (*simp add: eq-upred-def, transfer, simp*)

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) uexpr  $\Rightarrow$  ('a, 'α) uexpr is
λ σ e b. e (σ b) .

```

```

ad hoc-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) uexpr  $\Rightarrow$  'α usubst

```

```

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

```

```

ad hoc-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) uexpr ( $\langle - \rangle_s$ )
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
 $\lfloor \sigma \rfloor_s = (\lambda A. fst (\sigma (A, A)))$ 

```

```

nonterminal smaplet and smaplets

```

```

syntax

```

```

-smaplet :: [svar, 'a] => smaplet          (- / $\mapsto_s$ / -)
          :: smaplet => smaplets           (-)
-SMaplets :: [smaplet, smaplets] => smaplets (-, / -)
-SubstUpd :: ['m usubst, smaplets] => 'm usubst (-/'(-) [900,0] 900)
-Subst    :: smaplets => 'a  $\sim$  => '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 *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 uvar-indep-comm*)

lemma *usubst-upd-comm-dash* [*usubst*]:
fixes $x :: ('a, 'a) \text{ uvar}$
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 $\text{uvar } x \bowtie y$
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*] : $x \nmid P \implies \sigma(x \mapsto_s v) \dagger P = \sigma \dagger P$
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*]: $\sigma \dagger \text{var } x = \langle \sigma \rangle_s x$
by (*transfer*, *simp*)

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

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

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

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

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

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

lemma *subst-lift-upd* [*usubst*]:
fixes $x :: ('a, 'α) \text{uvar}$
shows $\lfloor \sigma(x \mapsto_s v) \rfloor_s = \lfloor \sigma \rfloor_s(x \mapsto_s \lfloor v \rfloor_<)$
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 (*rename-tac x v σ A*)
apply (*case-tac σ (A, A), simp*)
done

nonterminal *uexprs* and *svars*

syntax

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

translations

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

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* :: $('a, 'α) \text{uexpr} \Rightarrow ('a, 'α \times 'β) \text{uexpr} (\lfloor - \rfloor_<)$

is $\lambda p (A, A^{\wedge}). p A$.

lift-definition *drop-pre* :: ('a, 'α × 'α) uexpr ⇒ ('a, 'α) uexpr ([-]<)
is λ p A. p (A, A) .

lift-definition *lift-post* :: ('a, 'β) uexpr ⇒ ('a, 'α × 'β) uexpr ([-]>)
is λ p (A, A'). p A' .

abbreviation *drop-post* :: ('a, 'α × 'α) uexpr ⇒ ('a, 'α) uexpr ([-]>)
where [b]> ≡ [b]<

named-theorems *ulift*

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

5.2 Lifting laws

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

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

lemma *lift-pre-lit* [simp]:
 $[<<v>>]_< = <<v>>$
by (transfer, auto)

lemma *lift-post-lit* [simp]:
 $[<<v>>]_> = <<v>>$
by (transfer, auto)

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

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

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

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

lemma *lift-pre-trop* [simp]:
 $[trop\ f\ u\ v\ w]_< = trop\ f\ [u]_<\ [v]_<\ [w]_<$
by (transfer, auto)

lemma *lift-post-trop* [simp]:
 $[trop\ f\ u\ v\ w]_> = trop\ f\ [u]_>\ [v]_>\ [w]_>$
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) uexpr
```

```
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 ∧ 35) and
disj (infixr ∨ 30) and
Not (¬ - [40] 40)
```

consts

```
utruel :: 'a (true)
ufalse :: 'a (false)
uconj :: 'a ⇒ 'a ⇒ 'a (infixr ∧ 35)
udisj :: 'a ⇒ 'a ⇒ 'a (infixr ∨ 30)
uimpl :: 'a ⇒ 'a ⇒ 'a (infixr ⇒ 25)
uiff :: 'a ⇒ 'a ⇒ 'a (infixr ⇔ 25)
unot :: 'a ⇒ 'a (¬ - [40] 40)
uex :: ('a, 'α) uvar ⇒ 'p ⇒ 'p
uall :: ('a, 'α) uvar ⇒ 'p ⇒ 'p
ushEx :: ['a ⇒ 'p] ⇒ 'p
ushAll :: ['a ⇒ 'p] ⇒ '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 ⇒ logic ⇒ logic (∃ - - - [0, 10] 10)
-uall :: svar ⇒ logic ⇒ logic (∀ - - - [0, 10] 10)
-ushEx :: idt ⇒ logic ⇒ logic (∃ - - - [0, 10] 10)
-ushAll :: idt ⇒ logic ⇒ logic (∀ - - - [0, 10] 10)
-ushBEx :: idt ⇒ logic ⇒ logic ⇒ logic (∃ - ∈ - - - [0, 0, 10] 10)
-ushBAll :: idt ⇒ logic ⇒ logic ⇒ logic (∀ - ∈ - - - [0, 0, 10] 10)
```

translations

```

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

```

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 \sqcup 70)

notation *sup* (infixl \sqcap 65)

notation *Inf* (\bigsqcup - [900] 900)

notation *Sup* (\bigsqcap - [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  $x \leq y \implies y \leq x \implies x = y$ 
    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  $\Rightarrow$  'α upred  $\Rightarrow$  'α upred)
definition disj-upred = (sup :: 'α upred  $\Rightarrow$  'α upred  $\Rightarrow$  'α upred)
definition not-upred = (uminus :: 'α upred  $\Rightarrow$  'α upred)
definition diff-upred = (minus :: 'α upred  $\Rightarrow$  'α upred  $\Rightarrow$  'α upred)

```

We also define the other predicate operators

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

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

lift-definition *ex* :: $('a, '\alpha) \text{ uvar} \Rightarrow '\alpha \text{ upred} \Rightarrow '\alpha \text{ upred}$ **is**
 $\lambda x P b. (\exists v. P(\text{var-assign } x v b))$.

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

lift-definition *all* :: $('a, '\alpha) \text{ uvar} \Rightarrow '\alpha \text{ upred} \Rightarrow '\alpha \text{ upred}$ **is**
 $\lambda x P b. (\forall v. P(\text{var-assign } x v b))$.

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

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

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

adhoc-overloading

utru *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* = $((\text{simp only: upred-defs})? ; (\text{transfer, (rule-tac ext)}?, \text{auto simp add: fun-eq-iff})?)$

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*: $true = \ll True \gg$
by (*pred-tac*)

lemma *false-alt-def*: $false = \ll False \gg$
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*]: $\ll x \# P; x \# Q \gg \implies x \# P \wedge Q$
by (*pred-tac*)

lemma *unrest-disj* [*unrest*]: $\ll x \# P; x \# Q \gg \implies x \# P \vee Q$
by (*pred-tac*)

lemma *unrest-impl* [*unrest*]: $\ll x \# P; x \# Q \gg \implies x \# P \Rightarrow Q$
by (*pred-tac*)

lemma *unrest-iff* [*unrest*]: $\ll x \# P; x \# Q \gg \implies x \# P \Leftrightarrow Q$
by (*pred-tac*)

lemma *unrest-not* [*unrest*]: $x \# P \implies x \# (\neg P)$
by (*pred-tac*)

lemma *unrest-ex-same* [*unrest*]:
 $uvar\ x \implies x \# (\exists x \cdot P)$
by (*pred-tac*, *auto simp add: comp-def*)

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

lemma *unrest-all-same* [*unrest*]:
 $uvar\ x \implies x \# (\forall x \cdot P)$
by (*pred-tac*, *auto simp add: comp-def*)

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: uvar-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*]: $\sigma \dagger \text{true} = \text{true}$
by (*pred-tac*)

lemma *subst-false* [*usubst*]: $\sigma \dagger \text{false} = \text{false}$
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*]: $\sigma \dagger (\forall x \cdot P(x)) = (\forall x \cdot \sigma \dagger P(x))$
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 $x \bowtie y \ y \# v$
shows $(\exists y \cdot P)[v/x] = (\exists y \cdot P[v/x])$
using *assms*
by (*pred-tac, auto simp add: uvar-indep-def*)

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

lemma *subst-all-indep* [*usubst*]:
assumes $x \bowtie y \ y \nmid v$
shows $(\forall y \cdot P) \llbracket v/x \rrbracket = (\forall y \cdot P \llbracket v/x \rrbracket)$
using *assms*
by (*pred-tac, auto simp add: uvar-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*]: $\text{true} \neq \text{false} \quad \text{false} \neq \text{true}$
 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 $(P \wedge \$x =_u v) = (P[v/\$x] \wedge \$x =_u v)$
 using *assms*
 by (*pred-tac*, (*metis semi-uvar.var-update-eta uvar-semi-var*)+)

lemma *conj-eq-out-var-subst*:
 fixes $x :: ('a, 'α) \text{uvar}$
 assumes $\text{uvar } x$
 shows $(P \wedge \$x' =_u v) = (P[v/\$x'] \wedge \$x' =_u v)$
 using *assms*
 by (*pred-tac*, (*metis semi-uvar.var-update-eta uvar-semi-var*)+)

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

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

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

lemma *taut-split-subst*:

assumes $uvar\ x$
shows $'P' \longleftrightarrow (\forall\ v.\ 'P[\llbracket v \rrbracket/x]')$
using $assms$
by $(pred-tac, metis\ uvar-assign-exists)$

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

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

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

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

lemma $exists-twice$: $uvar\ x \Longrightarrow (\exists\ x \cdot \exists\ x \cdot P) = (\exists\ x \cdot P)$
by $(pred-tac, auto\ simp\ add: comp-def)$

lemma $all-twice$: $uvar\ x \Longrightarrow (\forall\ x \cdot \forall\ x \cdot P) = (\forall\ x \cdot P)$
by $(pred-tac, auto\ simp\ add: comp-def)$

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 $(pred-tac, auto\ simp\ add: uvar-indep-def)$

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 $(pred-tac, auto\ simp\ add: uvar-indep-def)$

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 α* :: $(' \alpha, ' \alpha \times ' \beta)$ *uvar* **where**

in α = (λ *var-lookup* = *fst*, *var-update* = $\lambda f (A, A'). (f A, A')$)

definition *out α* :: $(' \beta, ' \alpha \times ' \beta)$ *uvar* **where**

out α = (λ *var-lookup* = *snd*, *var-update* = $\lambda f (A, A'). (A, f A')$)

declare *in α -def* [*urel-defs*]

declare *out α -def* [*urel-defs*]

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

type-synonym $(' \alpha, ' \beta)$ *relation* = $(' \alpha \times ' \beta)$ *upred*

type-synonym $' \alpha$ *hrelation* = $(' \alpha \times ' \alpha)$ *upred*

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

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

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

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

lift-definition *seqr*:: $((' \alpha \times ' \beta)$ *upred*) \Rightarrow $((' \beta \times ' \gamma)$ *upred*) \Rightarrow $(' \alpha \times ' \gamma)$ *upred*

is $\lambda P Q r. r : (\{p. P p\} O \{q. Q q\})$.

lift-definition *conv-r* :: $('a, ' \alpha \times ' \beta)$ *uexpr* \Rightarrow $('a, ' \beta \times ' \alpha)$ *uexpr* ($-$ [999] 999)

is $\lambda e (b1, b2). e (b2, b1)$.

lift-definition *assigns-r* :: $' \alpha$ *usubst* \Rightarrow $' \alpha$ *hrelation* $(\langle \cdot \rangle_a)$

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

definition *skip-r* :: $' \alpha$ *hrelation* **where**

skip-r = *assigns-r id*

abbreviation *assign-r* :: ('t, 'α) *uvar* ⇒ ('t, 'α) *uexpr* ⇒ 'α *hrelation*
where *assign-r* *x v* ≡ *assigns-r* [*x* ↦_s *v*]

abbreviation *assign-2-r* ::

('t1, 'α) *uvar* ⇒ ('t2, 'α) *uvar* ⇒ ('t1, 'α) *uexpr* ⇒ ('t2, 'α) *uexpr* ⇒ 'α *hrelation*
where *assign-2-r* *x y u v* ≡ *assigns-r* [*x* ↦_s *u*, *y* ↦_s *v*]

nonterminal

id-list **and** *uexpr-list*

syntax

-id-unit :: *id* ⇒ *id-list* (-)
-id-list :: *id* ⇒ *id-list* ⇒ *id-list* (-, / -)
-uexpr-unit :: ('a, 'α) *uexpr* ⇒ *uexpr-list* (- [40] 40)
-uexpr-list :: ('a, 'α) *uexpr* ⇒ *uexpr-list* ⇒ *uexpr-list* (-, / - [40,40] 40)
-assignment :: *svars* ⇒ *uexprs* ⇒ 'α *hrelation* (**infixr** := 35)
-mk-usubst :: *svars* ⇒ *uexpr-list* ⇒ 'α *usubst*

translations

-mk-usubst (-*svar* *x*) (-*uexpr-unit* *v*) == [*x* ↦_s *v*]
-mk-usubst (-*id-list* *xs*) (-*uexpr-list* *v* *vs*) == (-*mk-usubst* *xs* *vs*)(*x* ↦_s *v*)
-assignment *xs* *vs* => *CONST assigns-r* (-*psubst* (*CONST id*) *xs* *vs*)
x := *v* <= *CONST assign-r* *x v*
x, y := *u, v* <= *CONST assign-2-r* *x y u v*

ad hoc-overloading

useq *seqr* **and**
uskip *skip-r*

method *rel-tac* = ((*simp add: upred-defs urel-defs*)?, (*transfer*, (*rule-tac ext*)?, *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 *lift-test* :: 'α *condition* ⇒ 'α *hrelation* ([-]ₜ)
where [b]ₜ = ([b]₍ < ∧ II)

declare *cond-def* [*urel-defs*]
declare *skip-r-def* [*urel-defs*]

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

definition *rel-var-res* :: 'α *hrelation* ⇒ ('a, 'α) *uvar* ⇒ 'α *hrelation* (**infix** ⊢_α 80) **where**
P ⊢_α *x* = (∃ \$x · ∃ \$x' · *P*)

declare *rel-var-res-def* [*urel-defs*]

7.1 Unrestriction Laws

lemma *unrest-iuvar* [*unrest*]: *uvar* *x* ⇒ *outα* # \$*x*
by (*simp add: outα-def iuvar-def, transfer, auto*)

lemma *unrest-ouvar* [*unrest*]: *uvar* *x* ⇒ *inα* # \$*x*'
by (*simp add: inα-def ouvar-def, transfer, auto*)

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

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

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

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

lemma *unrest-pre-out α* [*unrest*]: *out $\alpha \# \lceil b \rceil_<$*
by (*transfer*, *auto simp add: out α -def*)

lemma *unrest-post-in α* [*unrest*]: *in $\alpha \# \lceil b \rceil_>$*
by (*transfer*, *auto simp add: in α -def*)

lemma *unrest-pre-in-var* [*unrest*]:
 $x \# p1 \Longrightarrow \$x \# \lceil p1 \rceil_<$
by (*transfer*, *simp*)

lemma *unrest-post-out-var* [*unrest*]:
 $x \# p1 \Longrightarrow \$x' \# \lceil p1 \rceil_>$
by (*transfer*, *simp*)

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

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

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

lemma *unrest-out-rel-var-res* [*unrest*]:
 $\text{uvar } x \Longrightarrow \$x' \# (P \vdash_\alpha x)$
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*]:
 $\llbracket \text{uvar } x; \text{out}\alpha \# v \rrbracket \Longrightarrow (P ;; Q) \llbracket v / \$x \rrbracket = ((P \llbracket v / \$x \rrbracket) ;; Q)$
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=λ-.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=λ-.ba in spec)
apply (simp)
done

```

```

lemma usubst-seq-right [usubst]:
   $\llbracket \text{uvar } x; \text{in}\alpha \# v \rrbracket \Longrightarrow (P ;; Q) \llbracket v/\$x' \rrbracket = (P ;; Q \llbracket v/\$x' \rrbracket)$ 
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=λ-.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=λ-.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) uvar
  shows  $II \llbracket [v]_{<}/\$x \rrbracket = (x := v)$ 
by (rel-tac)

```

7.3 Lifting laws

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

7.4 Relation laws

Homogeneous relations form a quantale

abbreviation *truer* :: $'\alpha$ *hrelation* (*true_h*) **where**
truer \equiv *true*

abbreviation *false_r* :: $'\alpha$ *hrelation* (*false_h*) **where**
false_r \equiv *false*

interpretation *upred-quantale*: *unital-quantale-plus*
where *times* = *segr* **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*]: $\llbracket \text{out}\alpha \# p \rrbracket \implies \lceil [p]_{<} \rceil_{<} = p$
apply (*pred-tac*, *auto simp add: out α -def*)
apply (*rename-tac* *p a b*)
apply (*drule-tac* *x=a in spec*)
apply (*drule-tac* *x=b in spec*)
apply (*drule-tac* *x= λ -. a in spec*)
apply (*simp*)
done

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

definition *while* :: $'\alpha$ *condition* \Rightarrow $'\alpha$ *hrelation* \Rightarrow $'\alpha$ *hrelation* (*while* - *do* - *od*) **where**
while b do P od = $((\lceil b \rceil_{<} \wedge P)^*_u \wedge (\neg \lceil b \rceil_{>}))$

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 *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*]:

$(false ;; P) = false$

by *pred-tac*

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

$(P ;; false) = false$

by *pred-tac*

lemma *seqr-mono*:

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

by (*rel-tac*, *blast*)

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

by (*rel-tac*)

lemma *seqr-exists-left*:

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

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

lemma *seqr-exists-right*:

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

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*α.

lemma *assign-subst* [*usubst*]:
 $\llbracket \text{uvar } x; \text{uvar } y \rrbracket \Longrightarrow [\$x \mapsto_s [u]_<] \dagger (y := v) = (x, y := u, [x \mapsto_s u] \dagger v)$
by *rel-tac*

lemma *assigns-idem*: $\text{uvar } x \Longrightarrow (x, x := u, v) = (x := v)$
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*: $\text{uvar } x \Longrightarrow (\langle \sigma \rangle_a ;; P) = ([\sigma]_s \dagger P)$
by *rel-tac*

lemma *assign-r-comp*: $\text{uvar } x \Longrightarrow (x := u ;; P) = ([\$x \mapsto_s [u]_<] \dagger P)$
by (*simp add: assigns-r-comp usubst*)

lemma *assign-test*: $\text{uvar } x \Longrightarrow (x := \llbracket u \rrbracket ;; x := \llbracket v \rrbracket) = (x := \llbracket v \rrbracket)$
by (*simp add: assigns-comp subst-upd-comp subst-lit usubst-upd-idem*)

lemma *skip-r-unfold*:
 $\text{uvar } x \Longrightarrow H = (\$x' =_u \$x \wedge H|_\alpha x)$
by (*rel-tac, blast, metis uvar.var-state-eq-iff uvar.var-update-lookup*)

lemma *assign-unfold*:
 $\text{uvar } x \Longrightarrow (x := v) = (\$x' =_u [v]_< \wedge H|_\alpha x)$
apply (*rel-tac, auto simp add: comp-def*)
using *var-assign-eq* **apply** *fastforce*
done

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*:
 $P = (P ;; \text{true}) \longleftrightarrow (\text{out}\alpha \nmid P)$
apply (*rel-tac*)
apply (*rename-tac P a b y*)
apply (*drule-tac x=a in spec*)
apply (*drule-tac x=b in spec*)
apply (*drule-tac x=λ -.y in spec*)

apply (simp)
 done

theorem *postcond-equiv*:
 $P = (true ;; P) \longleftrightarrow (in\alpha \# P)$
 apply (rel-tac)
 apply (rename-tac $P \ a \ b \ y$)
 apply (drule-tac $x=a$ in spec)
 apply (drule-tac $x=b$ in spec)
 apply (drule-tac $x=\lambda \ .y$ in spec)
 apply (simp)
 done

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

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

theorem *precond-left-zero*:
 assumes $out\alpha \# p \ p \neq false$
 shows $(true ;; p) = true$
 using *assms*
 apply (simp add: *out α -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)
 apply (rule-tac $x=b1$ in *exI*)
 apply (drule-tac $x=b1$ in spec)
 apply (drule-tac $x=b2$ in spec)
 apply (drule-tac $x=\lambda \ . \ b$ in spec)
 apply (simp)
 done

7.5 Converse laws

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

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

lemma *uivar-convr* [*simp*]:
 fixes $x :: ('a, 'a) \ uvar$
 shows $(\$x)^- = \x'
 by *pred-tac*

lemma *uovar-convr* [*simp*]:
 fixes $x :: ('a, 'a) \ uvar$
 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*: $in\alpha \# q \implies ((P \wedge q) ;; R) = (P ;; (q^- \wedge R))$
 apply (*rel-tac*)
 apply (*rename-tac* *q* *P* *R* *a* *b* *y*)
 apply (*rule-tac* $x=y$ **in** *exI*, *simp*)
 apply (*drule-tac* $x=b$ **in** *spec*, *drule-tac* $x=y$ **in** *spec*, *drule-tac* $x=\lambda-.a$ **in** *spec*, *simp*)
 apply (*rename-tac* *q* *P* *R* *a* *b* *y*)
 apply (*rule-tac* $x=y$ **in** *exI*, *simp*)
 apply (*drule-tac* $x=a$ **in** *spec*, *drule-tac* $x=y$ **in** *spec*, *drule-tac* $x=\lambda-.b$ **in** *spec*, *simp*)
 done

theorem *seqr-post-out*: $in\alpha \# r \implies (P ;; (Q \wedge r)) = ((P ;; Q) \wedge r)$
 apply (*rel-tac*)
 apply (*rename-tac* *r* *P* *Q* *a* *b* *y*)
 apply (*drule-tac* $x=a$ **in** *spec*, *drule-tac* $x=b$ **in** *spec*, *drule-tac* $x=\lambda-.y$ **in** *spec*, *simp*)
 apply (*rename-tac* *r* *P* *Q* *a* *b* *y*)
 apply (*rule-tac* $x=y$ **in** *exI*)
 apply (*simp*, *drule-tac* $x=a$ **in** *spec*, *drule-tac* $x=b$ **in** *spec*, *drule-tac* $x=\lambda-.y$ **in** *spec*, *simp*)
 done

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* α)

lemma *seqr-pre-out*: $out\alpha \# p \implies ((p \wedge Q) ;; R) = (p \wedge (Q ;; R))$
 apply (*rel-tac*)
 apply (*rename-tac* *p* *Q* *R* *a* *b* *y*)
 apply (*drule-tac* $x=a$ **in** *spec*, *drule-tac* $x=b$ **in** *spec*, *drule-tac* $x=\lambda-.y$ **in** *spec*, *simp*)
 apply (*rename-tac* *p* *Q* *R* *a* *b* *y*)
 apply (*rule-tac* $x=y$ **in** *exI*)
 apply (*simp*, *drule-tac* $x=a$ **in** *spec*, *drule-tac* $x=b$ **in** *spec*, *drule-tac* $x=\lambda-.y$ **in** *spec*, *simp*)
 done

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*:

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

proof –

have $(\text{while } b \text{ do } P \text{ od} \wedge [b]_{<}) = ((([b]_{<} \wedge P)^*_u \wedge (\neg [b]_{>})) \wedge [b]_{<})$
 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 $\dots = ((II \wedge [b]_{>} \wedge \neg [b]_{>}) \vee ((([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u)) \wedge (\neg [b]_{>}))$
 by (simp add: utp-pred.inf.assoc utp-pred.inf-sup-distrib2)
 also have $\dots = ((([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u)) \wedge (\neg [b]_{>}))$
 by simp
 also have $\dots = (([b]_{<} \wedge P) ;; ((([b]_{<} \wedge P)^*_u) \wedge (\neg [b]_{>})))$
 by (simp add: seqr-post-out unrest)
 also have $\dots = ((P \wedge [b]_{<}) ;; \text{while } b \text{ do } P \text{ od})$
 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 $(\text{while } b \text{ do } P \text{ od} \wedge (\neg [b]_{<})) = ((([b]_{<} \wedge P)^*_u \wedge (\neg [b]_{>})) \wedge (\neg [b]_{<}))$
 by (simp add: while-def)
 also have $\dots = (((II \vee ([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u)) \wedge \neg [b]_{>} \wedge (\neg [b]_{<}))$
 by (simp add: disj-upred-def)
 also have $\dots = (((II \wedge \neg [b]_{>}) \wedge \neg [b]_{<}) \vee ((\neg [b]_{<}) \wedge ((([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u)) \wedge \neg [b]_{>}))$
 by (simp add: conj-disj-distr utp-pred.inf.commute)
 also have $\dots = (((II \wedge \neg [b]_{>}) \wedge \neg [b]_{<}) \vee (((\neg [b]_{<}) \wedge ([b]_{<} \wedge P) ;; ([b]_{<} \wedge P)^*_u)) \wedge \neg [b]_{>}))$
 by (simp add: seqr-pre-out unrest-not unrest-pre-outα utp-pred.inf.assoc)
 also have $\dots = (((II \wedge \neg [b]_{>}) \wedge \neg [b]_{<}) \vee ((\text{false} ;; ([b]_{<} \wedge P)^*_u)) \wedge \neg [b]_{>}))$
 by (simp add: conj-comm utp-pred.inf.left-commute)
 also have $\dots = ((II \wedge \neg [b]_{>}) \wedge \neg [b]_{<})$
 by simp
 also have $\dots = (II \wedge \neg [b]_{<})$
 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 seqr-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
```

```
  wwp :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'c (infix wp 60)
```

```
definition wp-upred :: (' $\alpha$ , ' $\beta$ ) relation  $\Rightarrow$  ' $\beta$  condition  $\Rightarrow$  ' $\alpha$  condition where
wp-upred Q r =  $\lfloor \neg (Q \;; \neg \lceil r \rceil_{<}) \rfloor_{<}$ 
```

```
adhoc-overloading
```

```
  wwp wp-upred
```

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

```
theorem wp-assigns-r [wp]:
  (assigns-r  $\sigma$ ) wp r =  $\sigma \upharpoonright r$ 
by rel-tac
```

```
theorem wp-skip-r [wp]:
   $\Pi$  wp r = r
by rel-tac
```

```
theorem wp-true [wp]:
   $r \neq \text{true} \Longrightarrow \text{true wp } r = \text{false}$ 
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  $\Rightarrow$  P wp r)  $\wedge$  (( $\neg$  b)  $\Rightarrow$  Q wp r))
by rel-tac
```

```
end
```

8 UTP Theories

```
theory utp-theory
imports utp-rel
begin
```

```
type-synonym ' $\alpha$  Healthiness-condition = ' $\alpha$  upred  $\Rightarrow$  ' $\alpha$  upred
```

```
definition
```

Healthy:: $\alpha \text{ upred} \Rightarrow \alpha \text{ Healthiness-condition} \Rightarrow \text{bool}$ (**infix** is 30)
where $P \text{ is } H \equiv (P = H P)$

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

declare *Healthy-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 $k = \text{VAR } \textit{boyle-k}$
definition $p = \text{VAR } \textit{boyle-p}$
definition $V = \text{VAR } \textit{boyle-V}$

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 ϕ) 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*:
 $X \sqsubseteq Y \Longrightarrow B(X) \sqsubseteq B(Y)$
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 φ_1 satisfied by Boyle's law by simplication of its definitional equation and then application of the predicate tactic.

lemma $B\text{-}\varphi_1$: φ_1 is B

by (*simp add: φ_1 -def, pred-tac*)

We prove that φ_2 does not satisfy Boyle's law by showing it's in fact equal to φ_1 . We do this via an automated Isar proof.

lemma $B\text{-}\varphi_2$: $B(\varphi_2) = \varphi_1$

proof –

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

by (*simp add: φ_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: φ_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 des-ok*

declare *ok-def* [*upred-defs*]

lemma *uvar-ok* [*simp*]: *uvar ok*

by (*unfold-locales*, *simp-all add: ok-def*, *metis alpha-d.ext-inject alpha-d.surjective alpha-d.update-convs(1)*)

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

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

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

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

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-desr* :: $(' \alpha, '\beta)$ *relation* \Rightarrow $(' \alpha, '\beta)$ *relation-d* ($\lceil _ \rceil_D$) **is**

$\lambda P (A, A'). P (\text{more } A, \text{more } A') .$

lift-definition *drop-desr* :: $(' \alpha, '\beta)$ *relation-d* \Rightarrow $(' \alpha, '\beta)$ *relation* ($\lfloor _ \rfloor_D$) **is**

$\lambda P (A, A'). P (\lfloor \text{des-ok} = \text{True}, \dots = A \rfloor, \lfloor \text{des-ok} = \text{True}, \dots = A' \rfloor) .$

definition *design*:: $(' \alpha, '\beta)$ *relation-d* \Rightarrow $(' \alpha, '\beta)$ *relation-d* \Rightarrow $(' \alpha, '\beta)$ *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*:: $(' \alpha, '\beta)$ *relation* \Rightarrow $(' \alpha, '\beta)$ *relation* \Rightarrow $(' \alpha, '\beta)$ *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*:: $'\alpha$ *condition* \Rightarrow $(' \alpha, '\beta)$ *relation* \Rightarrow $(' \alpha, '\beta)$ *relation-d* (**infixl** \vdash_n 60)

where $(p \vdash_n Q) = (\lceil p \rceil_{<} \vdash_r Q)$

definition *skip-d* :: $'\alpha$ *hrelation-d* (II_D)

where $II_D \equiv (\text{true} \vdash_r II)$

definition *assigns-d* :: $'\alpha$ *usubst* \Rightarrow $'\alpha$ *hrelation-d*

where *assigns-d* $\sigma = (\text{true} \vdash_r \text{assigns-r } \sigma)$

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

abbreviation *assign-d* :: $('a, '\alpha)$ *uvar* \Rightarrow $('a, '\alpha)$ *uexpr* \Rightarrow $'\alpha$ *hrelation-d* (**infix** $:=_D$ 40)

where *assign-d* $x v \equiv \text{assigns-d } [x \mapsto_s v]$

definition *J* :: $'\alpha$ *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) $\equiv ((P;;\text{true}) \Rightarrow P)$

abbreviation *sf*:: $(' \alpha, '\beta)$ *relation-d* \Rightarrow $(' \alpha, '\beta)$ *relation-d* ($-^f$ [1000] 1000)

where $\sigma f D \equiv D \llbracket \text{false}/\$ok' \rrbracket$

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

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

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

definition $\text{wp-design} :: (' \alpha, ' \beta) \text{ relation-}d \Rightarrow ' \beta \text{ condition} \Rightarrow ' \alpha \text{ condition } (\text{infix } \text{wp}_D 60)$ **where**
 $Q \text{wp}_D r = (\lfloor \text{pre}_D(Q) \rfloor ; \text{true} \rfloor_{<} \wedge (\text{post}_D(Q) \text{wp } r))$

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

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

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

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

10.2 Design laws

lemma $\text{lift-desr-unrest-ok} \text{ } [unrest]$:
 $\$ok \# \lfloor P \rfloor_D \text{ } \$ok' \# \lfloor P \rfloor_D$
by $(\text{transfer}, \text{simp add: ok-def})+$

lemma $\text{unrest-out-des-lift} \text{ } [unrest]: \text{out} \alpha \# p \Longrightarrow \text{out} \alpha \# \lfloor p \rfloor_D$
apply (pred-tac)
apply $(\text{auto simp add: out}\alpha\text{-def})$
apply $(\text{rename-tac } p \text{ } b \text{ } v \text{ } x)$
apply $(\text{drule-tac } x=\alpha\text{-d.more } x \text{ in spec})$
apply $(\text{drule-tac } x=\alpha\text{-d.more } b \text{ in spec})$
apply $(\text{drule-tac } x=\lambda \text{ } -. \alpha\text{-d.more } (v \text{ } b) \text{ in spec})$

```

apply (simp)
apply (rename-tac p b v x)
apply (drule-tac x=alpha-d.more x in spec)
apply (drule-tac x=alpha-d.more b in spec)
apply (drule-tac x=λ -. alpha-d.more (v b) in spec)
apply (simp)
done

```

```

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

```

```

lemma lift-dist-seq [simp]:
   $[P ;; Q]_D = ([P]_D ;; [Q]_D)$ 
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  $\dots = '(P2 \Rightarrow \$ok' \wedge Q2) \Rightarrow (P1 \Rightarrow \$ok' \wedge Q1)'$ 
    by (subst subst-bool-split[of in-var ok], simp-all, subst-tac)
  also with assms have  $\dots = '(\neg P2 \Rightarrow \neg P1) \wedge ((P2 \Rightarrow Q2) \Rightarrow P1 \Rightarrow Q1)'$ 
    by (subst subst-bool-split[of out-var ok], simp-all, subst-tac)
  also have  $\dots \longleftrightarrow '(P1 \Rightarrow P2)' \wedge 'P1 \wedge Q2 \Rightarrow Q1'$ 
    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*:

```

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

```

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]: $\text{pre}_D(P \vdash_r Q) = P$
 by pred-tac

theorem design-post [simp]: $\text{post}_D(P \vdash_r Q) = (P \Rightarrow Q)$
 by pred-tac

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

proof –

have $(\text{true} ;; (P \vdash Q)) = (\exists \text{ ok}_0 \cdot \text{true}[\llbracket \llbracket \text{ok}_0 \rrbracket / \$\text{ok}' \rrbracket ;; (P \vdash Q)[\llbracket \llbracket \text{ok}_0 \rrbracket / \$\text{ok} \rrbracket])$
 by (subst segr-middle[of ok], simp-all)
 also have $\dots = ((\text{true}[\llbracket \text{false} / \$\text{ok}' \rrbracket ;; (P \vdash Q)[\llbracket \text{false} / \$\text{ok} \rrbracket]) \vee (\text{true}[\llbracket \text{true} / \$\text{ok}' \rrbracket ;; (P \vdash Q)[\llbracket \text{true} / \$\text{ok} \rrbracket]))$
 by (simp add: disj-comm false-alt-def true-alt-def)
 also have $\dots = ((\text{true}[\llbracket \text{false} / \$\text{ok}' \rrbracket ;; \text{true}_h] \vee (\text{true} ;; ((P \vdash Q)[\llbracket \text{true} / \$\text{ok} \rrbracket]))$
 by (subst-tac, rel-tac)
 also have $\dots = \text{true}$
 by (subst-tac, simp add: precondition-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 $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (((\neg (P1)) ;; \text{true})) \wedge \neg (Q1 ;; (\neg P2))) \vdash (Q1 ;; Q2)$

proof –

have $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (\exists \text{ ok}_0 \cdot ((P1 \vdash Q1)[\llbracket \llbracket \text{ok}_0 \rrbracket / \$\text{ok}' \rrbracket ;; (P2 \vdash Q2)[\llbracket \llbracket \text{ok}_0 \rrbracket / \$\text{ok} \rrbracket])$
 by (rule segr-middle, simp)
 also have \dots
 $= (((P1 \vdash Q1)[\llbracket \text{false} / \$\text{ok}' \rrbracket ;; (P2 \vdash Q2)[\llbracket \text{false} / \$\text{ok} \rrbracket])$
 $\vee ((P1 \vdash Q1)[\llbracket \text{true} / \$\text{ok}' \rrbracket ;; (P2 \vdash Q2)[\llbracket \text{true} / \$\text{ok} \rrbracket]))$
 by (simp add: true-alt-def false-alt-def, pred-tac)
 also from assms
 have $\dots = (((\$ok \wedge P1 \Rightarrow Q1) ;; (P2 \Rightarrow \$ok' \wedge Q2)) \vee ((\neg (\$ok \wedge P1)) ;; \text{true}))$
 by (simp add: design-def usubst unrest, pred-tac)
 also have $\dots = ((\neg \$ok ;; \text{true}_h) \vee (\neg P1 ;; \text{true}) \vee (Q1 ;; \neg P2) \vee (\$ok' \wedge (Q1 ;; Q2)))$
 by (rel-tac)
 also have $\dots = ((\neg (\neg P1 ;; \text{true}) \wedge \neg (Q1 ;; \neg P2)) \vdash (Q1 ;; Q2))$
 by (simp add: precondition-right-unit design-def unrest, rel-tac)
 finally show ?thesis .

qed

theorem rdesign-composition:

$((P1 \vdash_r Q1) ;; (P2 \vdash_r Q2)) = (((\neg (P1)) ;; \text{true})) \wedge \neg (Q1 ;; (\neg P2))) \vdash_r (Q1 ;; Q2)$
 by (simp add: rdesign-def design-composition unrest)

lemma skip-d-alt-def: $\text{ID} = \text{true} \vdash \text{ID}$
 by (rel-tac)

theorem design-skip-idem [simp]:

$(\text{ID} ;; \text{ID}) = \text{ID}$
 by (simp add: skip-d-def urel-defs, pred-tac)

theorem *design-composition-cond*:

assumes

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

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

shows $((p1 \vdash Q1) ;; (P2 \vdash Q2)) = ((p1 \wedge \neg (Q1 ;; (\neg P2))) \vdash (Q1 ;; Q2))$

using *assms*

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

theorem *rdesign-composition-cond*:

assumes $\text{out}\alpha \# p1$

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 $Q1 \ Q2 :: 'a \text{ hrelation-d}$

assumes

$ok \# p1 \ ok \# p2$

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

shows $((\lceil p1 \rceil_{<} \vdash Q1) ;; (\lceil p2 \rceil_{<} \vdash Q2)) = ((\lceil p1 \wedge Q1 \text{ wp } p2 \rceil_{<} \vdash (Q1 ;; Q2))$

using *assms*

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

theorem *rdesign-composition-wp*:

fixes $Q1 \ Q2 :: 'a \text{ hrelation}$

shows $((\lceil p1 \rceil_{<} \vdash_r Q1) ;; (\lceil p2 \rceil_{<} \vdash_r Q2)) = ((\lceil p1 \wedge Q1 \text{ wp } p2 \rceil_{<} \vdash_r (Q1 ;; Q2))$

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

theorem *rdesign-wp [wp]*:

$(\lceil p \rceil_{<} \vdash_r Q) \text{ wp}_D r = (p \wedge Q \text{ wp } r)$

by *rel-tac*

theorem *wpd-seq-r*:

fixes $Q1 \ Q2 :: 'a \text{ hrelation}$

shows $(\lceil p1 \rceil_{<} \vdash_r Q1 ;; \lceil p2 \rceil_{<} \vdash_r Q2) \text{ wp}_D r = (\lceil p1 \rceil_{<} \vdash_r Q1) \text{ wp}_D ((\lceil p2 \rceil_{<} \vdash_r Q2) \text{ wp}_D r)$

apply (*simp add: wp*)

apply (*subst rdesign-composition-wp*)

apply (*simp only: wp*)

apply (*rel-tac*)

done

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

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

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

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

assumes $\text{out}\alpha \# p$

shows $(p \vdash_r Q ;; \Pi_D) = (p \vdash_r Q)$

using *assms*

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

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

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

$(\lceil H \rceil_D ;; \lceil P \rceil_D) = \lceil P \rceil_D$
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*:

$H1(\Pi) = \Pi_D$
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
 $(true_h ;; R) = true_h$
 $(\Pi_D ;; R) = R$
shows *R is H1*

proof –

have $R = (\Pi_D ;; R)$ **by** (*simp add: assms(2)*)
also have $\dots = (H1(\Pi) ;; R)$
by (*simp add: H1-design-skip*)
also have $\dots = (\$ok \Rightarrow \Pi) ;; R$
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: precondition-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 *nok-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** $\dots = (true_h ;; (\neg \$ok \vee P))$
by (*simp add: impl-alt-def*)
also from *assms* **have** $\dots = ((true_h ;; \neg \$ok) \vee (true_h ;; P))$
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 is *H1* $\longleftrightarrow (true_h ;; P) = true_h \wedge (II_D ;; P) = P$

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  $\dots = (\exists \$ok' \cdot P \wedge \$ok' =_u \text{false})$ 
    by  $(\text{rel-tac}, \text{metis } (\text{mono-tags}, \text{lifting}) \text{ alpha-d.surjective alpha-d.update-convs}(1))$ 
  also have  $\dots = P^f$ 
    by  $(\text{metis one-point out-var-uvar ouvar-def unrest-false uvar-ok})$ 
  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  $(\text{rel-tac}, \text{metis alpha-d.equality})$ 
  also have  $\dots = (P^t \wedge \$ok')$ 
    by  $(\text{rel-tac}, \text{metis } (\text{full-types}) \text{ alpha-d.surjective alpha-d.update-convs}(1))$ 
  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 $(\text{simp add: H2-def J-split})$

theorem *H2-equivalence*:

$P \text{ is } H2 \iff 'P^f \Rightarrow P^t'$

```

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

```

lemma *H2-equiv*:

$P \text{ is } H2 \iff P^t \sqsubseteq P^f$

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 $(\text{simp add: H2-split design-def usubst unrest, pred-tac})$

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 $\dots = (\$ok \Rightarrow (P^f \vee (P^t \wedge \$ok')))$
 by (metis H2-split)
 also have $\dots = (\$ok \wedge (\neg P^f) \Rightarrow \$ok' \wedge P^t)$

by *pred-tac*
 also have $\dots = (\$ok \wedge (\neg P^f) \Rightarrow \$ok' \wedge \$ok \wedge P^t)$
 by *pred-tac*
 also have $\dots = (\$ok \wedge [pre_D(P)]_D \Rightarrow \$ok' \wedge \$ok \wedge [post_D(P)]_D)$
 by (*simp add: ok-post ok-pre*)
 also have $\dots = (\$ok \wedge [pre_D(P)]_D \Rightarrow \$ok' \wedge [post_D(P)]_D)$
 by *pred-tac*
 also from *assms* have $\dots = pre_D(P) \vdash_r post_D(P)$
 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 $(P \vdash_r Q ;; H_D) = (P \vdash_r Q ;; true \vdash_r H)$

by (*simp add: skip-d-def*)

also from *assms* have $\dots = (\neg(\neg P ;; true) \wedge \neg(Q ;; \neg true)) \vdash_r (Q ;; H)$

by (*simp add: rdesign-composition*)

also from *assms* have $\dots = (\neg(\neg P ;; true) \wedge \neg(Q ;; \neg true)) \vdash_r Q$

by *simp*

also have $\dots = (\neg(\neg P ;; true)) \vdash_r Q$

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 $\dots \iff P = (\neg(\neg P ;; true))$

by (*metis rdesign-pre*)

also have $\dots \iff P = (P ;; true)$

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-outα-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 $P \text{ is } H1 \ P \text{ is } H3$

shows $P = \lfloor \text{pre}_D(P) \rfloor_{<} \vdash_n \text{post}_D(P)$

by (metis *H1-H3-is-rdesign assms drop-pre-inv ndesign-def precondition-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 precondition-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 $\text{pre}_D(P_1) \vdash_r \text{post}_D(P_1) \sqsubseteq \text{pre}_D(P_2) \vdash_r \text{post}_D(P_2)$
by (*metis H1-H2-commute H1-H2-is-rdesign H1-idem Healthy-def' assms*)
hence $(\text{pre}_D(P_1) \vdash_r \text{post}_D(P_1)) \parallel Q \sqsubseteq (\text{pre}_D(P_2) \vdash_r \text{post}_D(P_2)) \parallel Q$
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 'a partition = 'a 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.

definition *ind-uvar* :: $\text{nat} \Rightarrow ('a, 'a \text{ alphabet-d}) \text{ uvar} \Rightarrow ('a, ('a \times 'a \text{ partition}) \text{ alphabet-d}) \text{ uvar}$ **where**
ind-uvar i $x = (\text{var-lookup} = \text{var-lookup } x \circ (\lambda A. \text{des-ok} = \text{des-ok } A, \dots = \text{snd}(\text{more } A) ! i))$
 $, \text{var-update} = \text{undefined}$
 []

definition *pre-uvar* :: $('a, 'a \text{ alphabet-d}) \text{ uvar} \Rightarrow ('a, ('a \times 'a \text{ partition}) \text{ alphabet-d}) \text{ uvar}$ **where**
pre-uvar $x = (\text{var-lookup} = \text{var-lookup } x \circ (\lambda A. \text{des-ok} = \text{des-ok } A, \dots = \text{fst}(\text{more } A)))$
 $, \text{var-update} = \text{undefined} \text{ []}$

lemma *ind-uvar-semi-uvar*:
 $\text{semi-uvar } x \Longrightarrow \text{semi-uvar } (\text{ind-uvar } i \ x)$
apply (*unfold-locales*)
apply (*simp-all add:ind-uvar-def*)
oops

syntax

-uprevar :: $('t, 'a) \text{ uvar} \Rightarrow \text{logic } (\$<- [999] \ 999)$
-udotvar :: $\text{nat} \Rightarrow ('t, 'a) \text{ uvar} \Rightarrow \text{logic } (\&- [0,999] \ 999)$
-uidotvar :: $\text{nat} \Rightarrow ('t, 'a) \text{ uvar} \Rightarrow \text{logic } (\$- [0,999] \ 999)$
-uodotvar :: $\text{nat} \Rightarrow ('t, 'a) \text{ uvar} \Rightarrow \text{logic } (\$- ' [999] \ 999)$
-sdotvar :: $\text{nat} \Rightarrow \text{id} \Rightarrow \text{svar } (\&- [0,999] \ 999)$
-sin-dotvar :: $\text{nat} \Rightarrow \text{id} \Rightarrow \text{svar } (\$-)$
-sout-dotvar :: $\text{nat} \Rightarrow \text{id} \Rightarrow \text{svar } (\$-')$

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 $'\alpha\ merge = (' \alpha \times ' \alpha\ partition, ' \alpha)\ relation-d$

Separating simulations

lift-definition $sep-sim :: nat \Rightarrow (' \alpha, ' \alpha\ partition)\ relation-d\ (U'(-))\ is$

$\lambda\ n\ (A, A').\ des-ok\ A' = des-ok\ A \wedge length\ (\alpha-d.more\ A') > n \wedge \alpha-d.more\ A'!\ n = \alpha-d.more\ A.$

lift-definition $\alpha-ext :: (' \alpha, ' \beta)\ relation-d \Rightarrow (' \alpha, ' \alpha \times ' \beta)\ relation-d\ (-_+ [999]\ 999)\ is$

$\lambda\ P\ (A, A').\ P\ (A, \llbracket des-ok = des-ok\ A', \dots = snd\ (more\ A') \rrbracket) \wedge des-ok\ A' = des-ok\ A \wedge fst\ (more\ A') = more\ A.$

Parallel by merge

definition $design-par-by-merge ::$

$' \alpha\ hrelation-d \Rightarrow ' \alpha\ merge \Rightarrow ' \alpha\ hrelation-d \Rightarrow ' \alpha\ hrelation-d\ (\mathbf{infixr}\ \|\ -\ 85)$

where $P\ \|_M\ Q = (((P\ ;;\ U(0))\ \|\ (Q\ ;;\ U(1)))_+ \;;\ M)$

end

12 Reactive processes

theory $utp-reactive$

imports

$utp-concurrency$

$utp-event$

begin

12.1 Preliminaries

type-synonym $' \alpha\ trace = ' \alpha\ list$

fun $list-diff :: ' \alpha\ list \Rightarrow ' \alpha\ list \Rightarrow ' \alpha\ list\ option\ \mathbf{where}$

$list-diff\ l\ [] = Some\ l$

$| list-diff\ []\ l = None$

$| list-diff\ (x\#xs)\ (y\#ys) = (if\ (x = y)\ then\ (list-diff\ xs\ ys)\ else\ None)$

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 $'\vartheta$ *refusal* = $'\vartheta$ *set*

record $'\vartheta$ *alpha-rp* = *alpha-d* +
 rp-wait :: *bool*
 rp-tr :: $'\vartheta$ *trace*
 rp-ref :: $'\vartheta$ *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: uvar-indep-def, pred-tac*) +

lemma *wait-ok-indep* [*simp*]: $wait \bowtie ok \ ok \bowtie wait$
 by (*simp add: uvar-indep-def, pred-tac*) +

lemma *ref-ok-indep* [*simp*]: $ref \bowtie ok \ ok \bowtie ref$
 by (*simp add: uvar-indep-def, pred-tac*) +

lemma *tr-wait-indep* [*simp*]: $tr \bowtie wait \ wait \bowtie tr$
 by (*simp add: uvar-indep-def, pred-tac*) +

lemma *ref-wait-indep* [*simp*]: $ref \bowtie wait \ wait \bowtie ref$
 by (*simp add: uvar-indep-def, pred-tac*) +

lemma *tr-ref-indep* [*simp*]: $ref \bowtie tr \ tr \bowtie ref$
 by (*simp add: uvar-indep-def, pred-tac*) +

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 *upd-vstore-alpha-rp-ext* :: (*vstore* \Rightarrow *vstore*) \Rightarrow ($'a$, $'b$) *alpha-rp-ext* \Rightarrow ($'a$, $'b$) *alpha-rp-ext*

where [*simp*]: *upd-vstore-alpha-rp-ext* f x = *alpha-d.more* (*alpha-rp.more-update* (*upd-vstore* f) (*alpha-d.extend* *undefined* x))

instance

apply (*intro-classes*, *auto simp add: upd-store-parm* [*THEN sym*] *alpha-rp.defs* *alpha-d.defs*)

apply (*metis* (*no-types*, *lifting*) *alpha-d.ext-inject* *alpha-d.surjective* *alpha-rp.select-convs* (4) *alpha-rp.surjective* *alpha-rp.update-convs* (4) *get-upd-vstore*)

apply (*smt* *alpha-d.select-convs* (2) *alpha-rp.surjective* *alpha-rp.update-convs* (4) *upd-vstore-comp*)

apply (*metis* *alpha-d.select-convs* (2) *alpha-rp.surjective* *alpha-rp.update-convs* (4) *upd-vstore-eta*)

apply (*metis* *alpha-rp.unfold-congs* (5) *upd-store-parm*)

done

end

lemma *uvar-wait* [*simp*]: *uvar wait*

by (*unfold-locales*, *simp-all add: wait-def*)

 (*metis* (*no-types*, *lifting*) *alpha-d.ext-inject* *alpha-rp.ext-inject* *alpha-rp.surjective* *alpha-rp.update-convs* (1))

lemma *uvar-tr* [*simp*]: *uvar tr*
 by (*unfold-locales*, *simp-all add: tr-def*)
 (*metis alpha-d.ext-inject alpha-rp.ext-inject alpha-rp.surjective alpha-rp.update-convs*(2))

lemma *uvar-ref* [*simp*]: *uvar ref*
 by (*unfold-locales*, *simp-all add: ref-def*)
 (*metis alpha-d.ext-inject alpha-rp.ext-inject alpha-rp.surjective alpha-rp.update-convs*(3))

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 (ϑ, α) *alphabet-rp* = (ϑ, α) *alpha-rp-scheme alphabet*
type-synonym (ϑ, α, β) *relation-rp* = ((ϑ, α) *alphabet-rp*, (ϑ, β) *alphabet-rp*) *relation*
type-synonym (ϑ, α) *hrelation-rp* = ((ϑ, α) *alphabet-rp*, (ϑ, α) *alphabet-rp*) *relation*
type-synonym (ϑ, σ) *predicate-rp* = (ϑ, σ) *alphabet-rp upred*

abbreviation *wait-f*::(ϑ, α, β) *relation-rp* \Rightarrow (ϑ, α, β) *relation-rp* ($-_f$ [1000] 1000)
where *wait-f* $R \equiv R \llbracket \text{false} / \$\text{wait} \rrbracket$

abbreviation *wait-t*::(ϑ, α, β) *relation-rp* \Rightarrow (ϑ, α, β) *relation-rp* ($-_t$ [1000] 1000)
where *wait-t* $R \equiv R \llbracket \text{true} / \$\text{wait} \rrbracket$

lift-definition *lift-rea* :: (α, β) *relation* \Rightarrow (ϑ, α, β) *relation-rp* ($[-]_R$) **is**
 $\lambda P (A, A'). P \text{ (more } A, \text{ more } A')$.

lift-definition *drop-rea* :: (ϑ, α, β) *relation-rp* \Rightarrow (α, β) *relation* ($[-]_R$) **is**
 $\lambda P (A, A'). P \text{ (} \langle \text{des-ok} = \text{True}, \text{rp-wait} = \text{True}, \text{rp-tr} = [], \text{rp-ref} = \{\}, \dots = A \rangle,$
 $\langle \text{des-ok} = \text{True}, \text{rp-wait} = \text{True}, \text{rp-tr} = [], \text{rp-ref} = \{\}, \dots = A' \rangle \text{)}$.

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*:
 $P\ is\ R1 \iff (\$tr \leq_u \$tr') \sqsubseteq P$
 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 $P\ is\ R1\ Q\ is\ R1$
 shows $(P \;;\ Q)\ is\ R1$
 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 $P Q :: ('t, 'a, 'a) \text{ relation-rp}$
 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 ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot \exists tr_0 \cdot ((P[\langle \rangle / \$tr][\ll tt_1 \gg / \$tr'] ;; (Q[\langle \rangle / \$tr][\ll tt_2 \gg / \$tr'])))$
 $\wedge \ll tr_0 \gg =_u \$tr \hat{\ }_u \ll tt_1 \gg \wedge \$tr' =_u \ll tr_0 \gg \hat{\ }_u \ll tt_2 \gg)$

by (*simp add: seqr-pre-out seqr-post-out unrest*, *rel-tac*)

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot ((P[\langle \rangle / \$tr][\ll tt_1 \gg / \$tr'] ;; (Q[\langle \rangle / \$tr][\ll tt_2 \gg / \$tr'])))$
 $\wedge (\exists tr_0 \cdot \ll tr_0 \gg =_u \$tr \hat{\ }_u \ll tt_1 \gg \wedge \$tr' =_u \ll tr_0 \gg \hat{\ }_u \ll tt_2 \gg))$

by *rel-tac*

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot ((P[\langle \rangle / \$tr][\ll tt_1 \gg / \$tr'] ;; (Q[\langle \rangle / \$tr][\ll tt_2 \gg / \$tr'])))$
 $\wedge (\$tr' =_u \$tr \hat{\ }_u \ll tt_1 \gg \hat{\ }_u \ll tt_2 \gg))$

by *rel-tac*

finally show *?thesis* .

qed

lemma *R2-seqr-distribute*:
 fixes $P Q :: ('t, 'a, 'a) \text{ relation-rp}$
 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-seqr-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 ... =
 $((\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] ;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])(\langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr)$
by (*simp add: usubst unrest*)
also have ... =
 $(\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] ;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])(\$tr' - \$tr =_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle \wedge \$tr' \geq_u \$tr))$
by *pred-tac*
also have ... =
 $((\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] ;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])(\$tr' =_u \$tr \hat{^}_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle))$
proof –
have $\bigwedge tt_1 tt_2. (((\$tr' - \$tr =_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr) :: ('\vartheta, '\alpha, '\alpha) \text{ relation-rp})$
 $= (\$tr' =_u \$tr \hat{^}_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle)$
by (*rel-tac, metis prefix-subst strict-prefixE*)
thus *?thesis* **by** *simp*
qed
also have ... = $(R2(P) ;; R2(Q))$
by (*simp add: R2-seqr-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*]: $\Pi_r = (\Pi \vee (\neg \$ok \wedge \$tr \leq_u \$tr'))$

definition *R3-def* [*upred-defs*]: $R3(P) = (\Pi \triangleleft \$wait \triangleright P)$

definition *R3c-def* [*upred-defs*]: $R3c(P) = (\Pi_r \triangleleft \$wait \triangleright P)$

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 $(metis \textit{Healthy-def}' R3-semir-form)$

lemma *R1-R3-commute*: $R1(R3(P)) = R3(R1(P))$
by *rel-tac*

lemma *R2-R3-commute*: $R2(R3(P)) = R3(R2(P))$
by $(rel-tac, (metis (no-types, lifting) \alpha-rp.surjective \alpha-rp.update-convs(2) \textit{append-Nil2 prefix-subst strict-prefixE})+)$

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 $(metis (no-types, lifting) \alpha-rp.surjective \alpha-rp.update-convs(2) \textit{append-Nil2 prefix-subst strict-prefixE})$
done

end