

A Shallow Model of the UTP in Isabelle/HOL

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

February 1, 2016

Contents

1	UTP variables	2
1.1	Deep UTP variables	5
1.2	Cardinalities	5
1.3	Injection functions	6
1.4	Deep variables	8
2	UTP expressions	10
3	Unrestriction	14
4	Substitution	15
4.1	Substitution definitions	15
4.2	Substitution laws	16
5	Lifting expressions	18
5.1	Lifting definitions	18
5.2	Lifting laws	19
6	Alphabetised Predicates	19
6.1	Predicate syntax	20
6.2	Predicate operators	21
6.3	Proof support	23
6.4	Unrestriction Laws	23
6.5	Substitution Laws	25
6.6	Predicate Laws	25
6.7	Quantifier lifting	28
7	Alphabetised relations	28
7.1	Unrestriction Laws	30
7.2	Substitution laws	31
7.3	Relation laws	31
7.4	Converse laws	34
7.5	Weakest precondition calculus	36
8	UTP Theories	37
9	Example UTP theory: Boyle's laws	37

10 Designs	38
10.1 Definitions	39
10.2 Design laws	40
10.3 H1: No observation is allowed before initiation	42
10.4 H2: A specification cannot require non-termination	44
10.5 H3: The design assumption is a precondition	46
10.6 H4: Feasibility	48
11 Concurrent programming	48
12 Reactive processes	49
12.1 Preliminaries	49

1 UTP variables

```

theory utp-var
imports
  ../contrib/Kleene-Algebras/Quantales
  ../utils/cardinals
  ../utils/Continuum
  ../utils/finite-bijection
  ~~ /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 :: ( $'a \Rightarrow 'a$ )  $\Rightarrow ' \alpha \Rightarrow ' \alpha$ 

```

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

```

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

```

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

```

syntax -VAR ::  $id \Rightarrow ('a, 'r)$  uvar (VAR -)
translations VAR  $x \Rightarrow$  ( $\mid$  var-lookup =  $x$ , var-update = -update-name  $x$   $\mid$ )

```

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 uvar =
  fixes  $x :: ('a, 'r) \text{ uvar}$ 
  — Application of two updates should correspond to the composition of update functions
  assumes var-update-comp:  $\text{var-update } x \ f \ (\text{var-update } x \ g \ \sigma) = \text{var-update } x \ (f \circ g) \ \sigma$ 
  — Looking a variable up after updating it corresponds to updating the variable's prior valuation
  and var-update-lookup:  $\text{var-lookup } x \ (\text{var-update } x \ f \ \sigma) = f \ (\text{var-lookup } x \ \sigma)$ 
  — Updating a variable's value to the one it already has is ineffectual
  and var-update-eta:  $\text{var-update } x \ (\lambda\cdot. \text{var-lookup } x \ \sigma) \ \sigma = \sigma$ 

  declare uvar.var-update-comp [simp]
  declare uvar.var-update-lookup [simp]
  declare uvar.var-update-eta [simp]

```

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 $\text{uvar-indep} :: ('a, 'r) \text{ uvar} \Rightarrow ('b, 'r) \text{ uvar} \Rightarrow \text{bool}$ (**infix** \bowtie 50) **where**
 $x \bowtie y \iff (\forall \ f \ g \ \sigma. \text{var-update } x \ f \ (\text{var-update } y \ g \ \sigma) = \text{var-update } y \ g \ (\text{var-update } x \ f \ \sigma))$

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 $\text{uvar } x \ x \bowtie y$
shows $\text{var-lookup } x \ (\text{var-update } y \ f \ \sigma) = \text{var-lookup } x \ \sigma$
proof –
have $\text{var-lookup } x \ (\text{var-update } y \ f \ \sigma) = \text{var-lookup } x \ (\text{var-update } y \ f \ (\text{var-update } x \ (\lambda\cdot. \text{var-lookup } x \ \sigma) \ \sigma))$
by (*simp add: assms(1)*)
also have $\dots = \text{var-lookup } x \ (\text{var-update } x \ (\lambda\cdot. \text{var-lookup } x \ \sigma) \ (\text{var-update } y \ f \ \sigma))$
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 $in-var :: ('a, 'α) uvar \Rightarrow ('a, 'α \times 'β) uvar$ **where**
 $in-var\ x = (\mid var-lookup = var-lookup\ x \circ fst, var-update = (\lambda f\ (A, A'). (var-update\ x\ f\ A, A')) \mid)$

definition $out-var :: ('a, 'β) uvar \Rightarrow ('a, 'α \times 'β) uvar$ **where**
 $out-var\ x = (\mid var-lookup = var-lookup\ x \circ snd, var-update = (\lambda f\ (A, A'). (A, var-update\ x\ f\ A')) \mid)$

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

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

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

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 ($simp\ add: uvar-indep-def\ in-var-def\ out-var-def$)

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 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 $univ-alpha :: ('α, 'α) uvar\ (\Sigma)$ **where**
 $univ-alpha = (\mid var-lookup = id, var-update = id \mid)$

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 $uvar\text{-}comp :: ('a, 'α) uvar \Rightarrow ('b, 'α) uvar \Rightarrow ('a \times 'b, 'α) uvar$ (**infix** \circ_v 35) **where**
 $uvar\text{-}comp\ x\ y = (\mid\ var\text{-}lookup = \lambda\ A.\ (var\text{-}lookup\ x\ A,\ var\text{-}lookup\ y\ A)$
 $\quad,\ var\text{-}update = \lambda\ f.\ var\text{-}update\ x\ (\lambda\ a.\ fst\ (f\ (a,\ undefined))) \circ$
 $\quad\quad\quad var\text{-}update\ y\ (\lambda\ b.\ snd\ (f\ (undefined,\ b))) \mid)$

nonterminal $svar$

syntax

$-svar \quad :: id \Rightarrow svar\ (-\ [999]\ 999)$
 $-spvar \quad :: id \Rightarrow svar\ (\&- \ [999]\ 999)$
 $-sinvar \quad :: id \Rightarrow svar\ (\$- \ [999]\ 999)$
 $-soutvar \quad :: id \Rightarrow svar\ (\$-' \ [999]\ 999)$

translations

$-svar\ x ==> x$
 $-spvar\ x ==> x$
 $-sinvar\ x == CONST\ in\text{-}var\ x$
 $-soutvar\ x == CONST\ out\text{-}var\ x$

end

1.1 Deep UTP variables

theory $utp\text{-}dvar$

imports $utp\text{-}var$

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 $ucard = fin\ nat \mid aleph0\ (\aleph_0) \mid cont\ (\mathfrak{c})$

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

type-synonym *uuniv* = *nat set*

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

fun *uuniv* :: *ucard* \Rightarrow *uuniv set* (*U'*(-)) **where**
U(*fin* *n*) = $\{\{x\} \mid x. x \leq n\}$ |
U(\aleph_0) = $\{\{x\} \mid x. \text{True}\}$ |
U(*c*) = *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-of-finite* [*simp*]:

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

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

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

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

uncountable (*UNIV* :: '*a set*') \implies *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::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* :: *string* \Rightarrow (*'a::continuum*) *dvar*
is $\lambda n. \langle \text{dname-name} = n, \text{dname-card} = UCARD('a) \rangle$
by *auto*

lift-definition *dvar-name* :: (*'a::continuum*) *dvar* \Rightarrow *string* **is** *dname-name* .

lift-definition *dvar-card* :: (*'a::continuum*) *dvar* \Rightarrow *ucard* **is** *dname-card* .

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

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

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

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

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

lemma *vstore-upd-eta [simp]*: *vstore-upd* *x* ($\lambda -. \text{vstore-lookup } x s$) *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 (λ -. get-vstore s) s = s*
and *upd-store-param*: *upd-vstore f s = upd-vstore (λ -. f (get-vstore s)) s*

definition *dvar-lift* :: '*a*::*continuum* *dvar* \Rightarrow ('*a*, '*α*::*vst*) *uvar* (-↑ [999] 999)
where *dvar-lift x* = (| *var-lookup* = λ *v*. *vstore-lookup x (get-vstore v)*
, *var-update* = λ *f s*. *upd-vstore (vstore-upd x f) s*
|)

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 *uvar-dvar*: *uvar (x↑)*
apply (*unfold-locales, simp-all add: dvar-lift-def*)
apply (*subst upd-store-param*)
apply (*simp*)
done

Deep variables with different names are independent

lemma *dvar-indep-diff-name*:
assumes *dvar-name x* \neq *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

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 adhoc-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)

```

```

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 $idvar :: 't::continuum\ dvar \Rightarrow ('t, ' \alpha::vst \times ' \beta) \ uexpr$
where $idvar\ x = var\ (in-var\ (dvar-lift\ x))$

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

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

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

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

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

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

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

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

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

syntax

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

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

```

Instantiating uminus also provides negation for predicates later

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

Set up automation for numerals

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

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

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

adhoc-overloading

ueq eq-upred

nonterminal *utuple-args*

syntax

-unil :: ('a list, 'α) uexpr ($\langle \rangle$)
 -ulist :: args => ('a list, 'α) uexpr ($\langle \langle (-) \rangle \rangle$)
 -uappend :: ('a list, 'α) uexpr \Rightarrow ('a list, 'α) uexpr \Rightarrow ('a list, 'α) uexpr (**infixr** \hat{u} 80)
 -uless :: ('a, 'α) uexpr \Rightarrow ('a, 'α) uexpr \Rightarrow (bool, 'α) uexpr (**infix** $<_u$ 50)
 -uleq :: ('a, 'α) uexpr \Rightarrow ('a, 'α) uexpr \Rightarrow (bool, 'α) uexpr (**infix** \leq_u 50)
 -ugreat :: ('a, 'α) uexpr \Rightarrow ('a, 'α) uexpr \Rightarrow (bool, 'α) uexpr (**infix** $>_u$ 50)
 -ugeq :: ('a, 'α) uexpr \Rightarrow ('a, 'α) uexpr \Rightarrow (bool, 'α) uexpr (**infix** \geq_u 50)
 -uempset :: ('a set, 'α) uexpr ($\{ \}_u$)
 -uset :: args => ('a set, 'α) uexpr ($\{ \langle (-) \rangle \}_u$)
 -uunion :: ('a set, 'α) uexpr \Rightarrow ('a set, 'α) uexpr \Rightarrow ('a set, 'α) uexpr (**infixl** \cup_u 65)
 -uinter :: ('a set, 'α) uexpr \Rightarrow ('a set, 'α) uexpr \Rightarrow ('a set, 'α) uexpr (**infixl** \cap_u 70)
 -umem :: ('a, 'α) uexpr \Rightarrow ('a set, 'α) uexpr \Rightarrow (bool, 'α) uexpr (**infix** \in_u 50)
 -unmem :: ('a, 'α) uexpr \Rightarrow ('a set, 'α) uexpr \Rightarrow (bool, 'α) uexpr (**infix** \notin_u 50)
 -utuple :: ('a, 'α) uexpr \Rightarrow utuple-args \Rightarrow ('a * 'b, 'α) uexpr ($\langle \langle 1'(-, / -)_u \rangle \rangle$)
 -utuple-arg :: ('a, 'α) uexpr \Rightarrow utuple-args (-)
 -utuple-args :: ('a, 'α) uexpr => utuple-args \Rightarrow utuple-args (-, / -)

translations

$\langle \rangle$ == $\llbracket \rangle \rrbracket$
 $\langle x, xs \rangle$ == $\text{CONST bop } (op \ \#) \ x \ \langle xs \rangle$
 $\langle x \rangle$ == $\text{CONST bop } (op \ \#) \ x \ \llbracket \rangle \rrbracket$
 $x \hat{u} y$ == $\text{CONST bop } (op \ @) \ x \ y$
 $x <_u y$ == $\text{CONST bop } (op \ <) \ x \ y$
 $x \leq_u y$ == $\text{CONST bop } (op \ \leq) \ x \ y$
 $x >_u y$ == $y <_u x$
 $x \geq_u y$ == $y \leq_u x$
 $\{ \}_u$ == $\llbracket \{ \} \rrbracket$
 $\{ x, xs \}_u$ == $\text{CONST bop } (\text{CONST insert}) \ x \ \{ xs \}_u$
 $\{ x \}_u$ == $\text{CONST bop } (\text{CONST insert}) \ x \ \llbracket \{ \} \rrbracket$
 $A \cup_u B$ == $\text{CONST bop } \text{Set.union} \ A \ B$
 $A \cap_u B$ == $\text{CONST bop } \text{Set.inter} \ A \ B$
 $x \in_u A$ == $\text{CONST bop } (op \ \in) \ x \ A$
 $x \notin_u A$ == $\text{CONST bop } (op \ \notin) \ x \ A$
 $(x, y)_u$ == $\text{CONST bop } (\text{CONST Pair}) \ x \ y$
 $\text{-utuple } x \ (\text{-utuple-args } y \ z)$ == $\text{-utuple } x \ (\text{-utuple-arg } (\text{-utuple } y \ z))$

```

lemmas uexpr-defs =
  iuvar-def
  ouvar-def
  zero-uexpr-def
  one-uexpr-def
  plus-uexpr-def
  uminus-uexpr-def
  minus-uexpr-def
  times-uexpr-def
  div-uexpr-def
  mod-uexpr-def
  eq-upred-def
  numeral-uexpr-simp

```

```

lemma var-in-var:  $\text{var } (\text{in-var } x) = \$x$ 
  by (simp add: iuvar-def)

```

```

lemma var-out-var:  $\text{var } (\text{out-var } x) = \$x'$ 
  by (simp add: ouvar-def)

```

```

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  $\lambda x e. \forall b v. e (\text{var-update } x v b) = e b$  .

```

```

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

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

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

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

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

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 $'\alpha \text{ usubst} = 'a \text{ alphabet} \Rightarrow 'a \text{ alphabet}$

lift-definition *subst* :: $'\alpha \text{ usubst} \Rightarrow ('a, '\alpha) \text{ uexpr} \Rightarrow ('a, '\alpha) \text{ uexpr}$ **is**
 $\lambda \sigma \ e \ b. \ e \ (\sigma \ b)$.

adhoc-overloading
usubst *subst*

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

consts *subst-upd* :: $'\alpha \text{ usubst} \Rightarrow 'v \Rightarrow ('a, '\alpha) \text{ uexpr} \Rightarrow 'a \text{ usubst}$

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

definition *subst-upd-dvar* :: $'\alpha \text{ usubst} \Rightarrow 'a::\text{continuum} \text{ dvar} \Rightarrow ('a, '\alpha::\text{vst}) \text{ uexpr} \Rightarrow 'a \text{ usubst}$ **where**
subst-upd-dvar $\sigma \ x \ v = (\lambda \ b. \text{var-assign } (\text{dvar-lift } x) \ (\llbracket v \rrbracket_e b) \ (\sigma \ b))$

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* :: $'\alpha$ *usubst* \Rightarrow $('a, '\alpha)$ *uvar* \Rightarrow $('a, '\alpha)$ *uexpr* $(\langle \cdot \rangle_s)$
is $\lambda \sigma x b. \text{var-lookup } x (\sigma b)$.

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

definition *usubst-rel-lift* :: $'\alpha$ *usubst* \Rightarrow $(' \alpha \times ' \beta)$ *usubst* $([\cdot]_s)$ **where**
 $[\sigma]_s = (\lambda (A, A'). (\sigma A, A'))$

definition *usubst-rel-drop* :: $(' \alpha \times ' \alpha)$ *usubst* \Rightarrow $' \alpha$ *usubst* $([\cdot]_s)$ **where**
 $[\sigma]_s = (\lambda A. \text{fst } (\sigma (A, A)))$

nonterminal *smaplet* and *smaplets*

syntax

-smaplet :: $[svar, 'a] \Rightarrow \text{smaplet}$ $(- \mapsto_s / -)$
 :: *smaplet* $\Rightarrow \text{smaplets}$ $(-)$
-SMaplets :: $[smaplet, smaplets] \Rightarrow \text{smaplets}$ $(-, / -)$
-SubstUpd :: $[m \text{ usubst}, smaplets] \Rightarrow m \text{ usubst}$ $(-/'(-) [900, 0] 900)$
-Subst :: *smaplets* $\Rightarrow 'a \leadsto \Rightarrow 'b$ $((1[\cdot]))$

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

lemma *subst-unrest* [*usubst*] : $x \# P \implies \sigma(x \mapsto_s v) \dagger P = \sigma \dagger P$
 by (*simp add: subst-upd-uvar-def, transfer, auto*)

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

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

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

lemma *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*)

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

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

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

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

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

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

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

lemma *subst-upd-comp* [*usubst*] :
 fixes $x :: ('a, 'α) \text{uvar}$
 shows $\varrho(x \mapsto_s v) \circ \sigma = (\varrho \circ \sigma)(x \mapsto_s \sigma \dagger v)$
 by (*rule ext, simp add: 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]:  $\lfloor \lceil \sigma \rceil_s \rfloor_s = \sigma$ 
  by (simp add: usubst-rel-lift-def usubst-rel-drop-def)

lemma subst-lift-upd [usubst]:  $\lceil \sigma(x \mapsto_s v) \rceil_s = \lceil \sigma \rceil_s(\$x \mapsto_s \lceil v \rceil_<)$ 
  by (simp add: usubst-rel-lift-def subst-upd-uvar-def, transfer, auto)

lemma subst-drop-upd [usubst]:  $\lfloor \sigma(\$x \mapsto_s v) \rfloor_s = \lfloor \sigma \rfloor_s(x \mapsto_s \lfloor v \rfloor_<)$ 
  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  $\sigma$  A)
  apply (case-tac  $\sigma$  (A, A), simp)
done

```

nonterminal *uexprs* and *svars*

syntax

```

-psubst :: [ $\alpha$  usubst, svars, uexprs]  $\Rightarrow$  logic
-subst :: ( $'a$ ,  $'\alpha$ ) uexpr  $\Rightarrow$  uexprs  $\Rightarrow$  svars  $\Rightarrow$  ( $'a$ ,  $'\alpha$ ) uexpr (( $\lceil$ -/ $\rfloor$ ) [999,999] 1000)
-uexprs :: [ $'a$ ,  $'\alpha$ ] uexpr, uexprs]  $\Rightarrow$  uexprs (-, / -)
  :: ( $'a$ ,  $'\alpha$ ) uexpr  $\Rightarrow$  uexprs (-)
-svars :: [svar, svars]  $\Rightarrow$  svars (-, / -)
  :: svar  $\Rightarrow$  svars (-)

```

translations

```

-subst P es vs  $\Rightarrow$  CONST subst (-psubst (CONST id) vs es) P
-psubst m (-svar x) v  $\Rightarrow$  CONST subst-upd m x v
-psubst m (-spvar x) v  $\Rightarrow$  CONST subst-upd m x v
-psubst m (-sinvar x) v  $\Rightarrow$  CONST subst-upd m (CONST in-var x) v
-psubst m (-soutvar x) v  $\Rightarrow$  CONST subst-upd m (CONST out-var x) v
-psubst m (-svars x xs) (-uexprs v vs)  $\Rightarrow$  -psubst (-psubst m x v) xs vs

```

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* :: ($'a$, $'\alpha$) *uexpr* \Rightarrow ($'a$, $'\alpha \times '\beta$) *uexpr* (\lceil - $\rfloor_<$)
is $\lambda p (A, A'). p A$.

lift-definition *drop-pre* :: ($'a$, $'\alpha \times '\alpha$) *uexpr* \Rightarrow ($'a$, $'\alpha$) *uexpr* (\lfloor - $\rfloor_<$)
is $\lambda p A. p (A, A)$.

lift-definition *lift-post* :: ($'a$, $'\beta$) *uexpr* \Rightarrow ($'a$, $'\alpha \times '\beta$) *uexpr* (\lceil - $\rfloor_>$)
is $\lambda p (A, A'). p A'$.

abbreviation *drop-post* :: ($'a$, $'\alpha \times '\alpha$) *uexpr* \Rightarrow ($'a$, $'\alpha$) *uexpr* (\lfloor - $\rfloor_>$)

where $[b]_> \equiv [b]_<$

5.2 Lifting laws

lemma *lift-pre-var* [*simp*]:

$\lceil \text{var } x \rceil_< = \x

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

lemma *lift-post-var* [*simp*]:

$\lceil \text{var } x \rceil_> = \x'

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

lemma *lift-pre-lit* [*simp*]:

$\lceil \ll v \gg \rceil_< = \ll v \gg$

by (*transfer, auto*)

lemma *lift-post-lit* [*simp*]:

$\lceil \ll v \gg \rceil_> = \ll v \gg$

by (*transfer, auto*)

lemma *lift-pre-uop* [*simp*]:

$\lceil \text{uop } f \ v \rceil_< = \text{uop } f \ \lceil v \rceil_<$

by (*transfer, auto*)

lemma *lift-post-uop* [*simp*]:

$\lceil \text{uop } f \ v \rceil_> = \text{uop } f \ \lceil v \rceil_>$

by (*transfer, auto*)

lemma *lift-pre-bop* [*simp*]:

$\lceil \text{bop } f \ u \ v \rceil_< = \text{bop } f \ \lceil u \rceil_< \ \lceil v \rceil_<$

by (*transfer, auto*)

lemma *lift-post-bop* [*simp*]:

$\lceil \text{bop } f \ u \ v \rceil_> = \text{bop } f \ \lceil u \rceil_> \ \lceil v \rceil_>$

by (*transfer, auto*)

lemma *lift-pre-trop* [*simp*]:

$\lceil \text{trop } f \ u \ v \ w \rceil_< = \text{trop } f \ \lceil u \rceil_< \ \lceil v \rceil_< \ \lceil w \rceil_<$

by (*transfer, auto*)

lemma *lift-post-trop* [*simp*]:

$\lceil \text{trop } f \ u \ v \ w \rceil_> = \text{trop } f \ \lceil u \rceil_> \ \lceil v \rceil_> \ \lceil w \rceil_>$

by (*transfer, auto*)

end

6 Alphabetised Predicates

theory *utp-pred*

imports

utp-expr

utp-subst

begin

An alphabetised predicate is simply a boolean valued expression

type-synonym $'\alpha$ *upred* = (*bool*, $'\alpha$) *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** \wedge 35) **and**
disj (**infixr** \vee 30) **and**
Not (\neg - [40] 40)

consts

uttrue :: $'a$ (*true*)
ufalse :: $'a$ (*false*)
uconj :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** \wedge 35)
udisj :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** \vee 30)
wimpl :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** \Rightarrow 25)
wiff :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** \Leftrightarrow 25)
unot :: $'a \Rightarrow 'a$ (\neg - [40] 40)
uex :: ($'a$, $'\alpha$) *uvar* $\Rightarrow 'p \Rightarrow 'p$
uall :: ($'a$, $'\alpha$) *uvar* $\Rightarrow 'p \Rightarrow 'p$
ushEx :: $['a \Rightarrow 'p] \Rightarrow 'p$
ushAll :: $['a \Rightarrow 'p] \Rightarrow 'p$

adhoc-overloading

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

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

syntax

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

translations

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

$\forall x. P == \text{CONST } \text{ushAll } (\lambda x. P)$

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* \Rightarrow *y* \leq *x* \Rightarrow *x* = *y*

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

qed

end

We also trivially instantiate our refinement class

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

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

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

begin

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

```

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

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

```

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

```

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

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

```

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

```

definition true-upred = (top :: 'α upred)
definition false-upred = (bot :: 'α upred)
definition conj-upred = (inf :: 'α upred  $\Rightarrow$  '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::'α upred  $\Rightarrow$  'α upred  $\Rightarrow$  'α upred is
 $\lambda P Q A. P A \longrightarrow Q A .$ 

lift-definition iff-upred ::'α upred  $\Rightarrow$  'α upred  $\Rightarrow$  'α upred is
 $\lambda P Q A. P A \longleftrightarrow Q A .$ 

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

lift-definition shEx :: ['β  $\Rightarrow$  'α upred]  $\Rightarrow$  'α upred is
 $\lambda P A. \exists x. (P x) A .$ 

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

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

ad hoc-overloading

ut *true* *true-upred* **and**
uf *false* *false-upred* **and**
un *not* *not-upred* **and**
uc *conj* *conj-upred* **and**
ud *disj* *disj-upred* **and**
ui *impl* *impl* **and**
ui *iff* *iff-upred* **and**
ue *ex* *ex* **and**
ua *all* *all* **and**
ushEx *shEx* *shEx* **and**
ushAll *shAll*

6.3 Proof support

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

method *pred-tac* = ((*simp only: upred-defs*)? ; (*transfer, (rule-tac ext)?, auto*)?)

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 *ue* *expr-defs* [*upred-defs*]
declare *usubst-rel-lift-def* [*upred-defs*]
declare *usubst-rel-drop-def* [*upred-defs*]

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

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

6.4 Unrestriction Laws

lemma *unrest-true* [*unrest*]: $x \# \text{true}$

by (*pred-tac*)

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

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

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

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

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

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

lemma *unrest-ex-same* [*unrest*]:
 $\text{uvar } x \implies x \# (\exists x \cdot P)$
 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*]:
 $\text{uvar } x \implies x \# (\forall x \cdot P)$
 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*

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*: $P = R \Longrightarrow ((P::'\alpha \text{ upred}) \wedge Q) = (R \wedge Q)$

by *pred-tac*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 *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 $\text{uvar } x \text{ } x \# v$
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* (*full-types*) *uvar.var-update-eta*)

lemma *eq-split*:
assumes $'P \Rightarrow Q' \text{ } 'Q \Rightarrow P'$
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 $\dots = ('P[\llbracket \text{True} \rrbracket/x]' \wedge 'P[\llbracket \text{False} \rrbracket/x])'$
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 $x :: ('a, 'a) \text{ uvar}$
shows $(p[\llbracket u \rrbracket/x] \wedge u =_u v) = (p[\llbracket v \rrbracket/x] \wedge u =_u v)$
by *pred-tac*

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

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

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

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

declare *in α -def* [*urel-defs*]

declare *out α -def* [*urel-defs*]

type-synonym ' α *condition* = ' α *upred*

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

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

definition *cond*::(' α , ' β) *relation* \Rightarrow ' α *condition* \Rightarrow (' α , ' β) *relation* \Rightarrow (' α , ' β) *relation*
 $((\exists - \triangleleft - \triangleright / -) [14,0,15] 14)$

where (*P* \triangleleft *b* \triangleright *Q*) \equiv ($\lceil b \rceil_{<} \wedge P$) \vee ($(\neg \lceil b \rceil_{<}) \wedge Q$)

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

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

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

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

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

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

definition *skip-r* :: ' α *hrelation* **where**

skip-r = *assigns-r id*

abbreviation *assign-r* :: ('*t*, ' α) *uvar* \Rightarrow ('*t*, ' α) *uexpr* \Rightarrow ' α *hrelation*

where *assign-r* *x v* \equiv *assigns-r* [*x* \mapsto_s *v*]

abbreviation *assign-2-r* ::

('*t1*, ' α) *uvar* \Rightarrow ('*t2*, ' α) *uvar* \Rightarrow ('*t1*, ' α) *uexpr* \Rightarrow ('*t2*, ' α) *uexpr* \Rightarrow ' α *hrelation*

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

nonterminal

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

syntax

-id-unit :: *id* \Rightarrow *id-list* (-)

-id-list :: *id* \Rightarrow *id-list* \Rightarrow *id-list* (-, / -)

-uexpr-unit :: ('*a*, ' α) *uexpr* \Rightarrow *uexpr-list* (- [40] 40)

-uexpr-list :: ('*a*, ' α) *uexpr* \Rightarrow *uexpr-list* \Rightarrow *uexpr-list* (-, / - [40,40] 40)

-assignment :: *id-list* \Rightarrow *uexpr-list* \Rightarrow ' α *hrelation* (**infixr** := 35)

-mk-usubst :: *id-list* \Rightarrow *uexpr-list* \Rightarrow ' α *usubst*

translations

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

ad hoc overloading

$useq \ seqr \ and$
 $uskip \ skip-r$

method $rel-tac = ((simp \ add: \ upred-defs \ urel-defs)?, (transfer, (rule-tac \ ext)?, auto \ simp \ add: \ urel-defs \ relcomp-unfold)?)$

A test is like a precondition, except that it identifies to the postcondition. It forms the basis for Kleene Algebra with Tests (KAT).

definition $lift-test :: 'a \ condition \Rightarrow 'a \ hrelation \ ([\cdot]_t)$
where $[\cdot]_t = ([\cdot]_< \wedge II)$

declare $cond-def \ [urel-defs]$
declare $skip-r-def \ [urel-defs]$

7.1 Unrestriction Laws

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

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

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

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

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

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

lemma $unrest-pre-out\alpha \ [unrest]: \ out\alpha \ \# \ [\cdot]_<$
by $(transfer, \ auto \ simp \ add: \ out\alpha-def)$

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

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

7.2 Substitution laws

It should be possible to substantially generalise the following two laws

lemma *usubst-seq-left* [*usubst*]:
 $\llbracket \text{uvar } x; \text{out}\alpha \# v \rrbracket \implies (P ;; Q) \llbracket v/\$x \rrbracket = ((P \llbracket v/\$x \rrbracket) ;; Q)$
 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*]:
 $\llbracket \text{uvar } x; \text{in}\alpha \# v \rrbracket \implies (P ;; Q) \llbracket v/\$x' \rrbracket = (P ;; Q \llbracket v/\$x' \rrbracket)$
 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

7.3 Relation laws

Homogeneous relations form a quantale

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

abbreviation *falsr* :: ' α hrelation (*false_h*) **where**
falsr \equiv *false*

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

where *times* = *seqr* **and** *one* = *skip-r* **and** *Sup* = *Sup* **and** *Inf* = *Inf* **and** *inf* = *inf* **and** *less-eq* =
less-eq **and** *less* = *less*
and *sup* = *sup* **and** *bot* = *bot* **and** *top* = *top*
 apply (*unfold-locales*)
 apply (*rel-tac*)

```

apply (unfold SUP-def, transfer, auto)
apply (unfold SUP-def, transfer, auto)
apply (unfold INF-def, transfer, auto)
apply (unfold INF-def, transfer, auto)
apply (rel-tac)
apply (rel-tac)
done

```

```

lemma drop-pre-inv [simp]:  $\llbracket \text{out}\alpha \# p \rrbracket \implies \llbracket p \rrbracket_{<} = p$ 
  apply (pred-tac, auto simp add: out $\alpha$ -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= $\lambda$  -. a in spec)
  apply (simp)
done

```

```

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

```

```

lemma cond-eq-distr:
   $((P \iff Q) \triangleleft b \triangleright (R \iff S)) = ((P \triangleleft b \triangleright R) \iff (Q \triangleleft b \triangleright S))$  by rel-tac

```

```

lemma comp-cond-left-distr:
   $((P \triangleleft b \triangleright Q) ;; R) = ((P ;; R) \triangleleft b \triangleright (Q ;; R))$ 
  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]:
   $(\text{II} ;; P) = P$ 
  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*

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

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

lemma *assigns-idem*: $uvar\ x \Longrightarrow (x, x := u, v) = (x := u)$
 by (*simp add: usubst*)

lemma *assigns-comp*: $(assigns-r\ f ;; assigns-r\ g) = assigns-r\ (g \circ f)$
 by (*transfer, auto simp add: relcomp-unfold*)

lemma *assigns-r-comp*: $uvar\ x \Longrightarrow (\langle\sigma\rangle_a ;; P) = ([\sigma]_s \dagger P)$
 by *rel-tac*

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

lemma *assign-test*: $uvar\ x \Longrightarrow (x := \langle u \rangle ;; x := \langle v \rangle) = (x := \langle v \rangle)$
 by (*simp add: assigns-comp subst-upd-comp subst-lit usubst-upd-idem*)

lemma *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 $(P ;; Q) = (\exists\ v \cdot P[\langle v \rangle / \$x'] ;; Q[\langle v \rangle / \$x])$
 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*:
 $P = (P ;; true) \longleftrightarrow (out\alpha \# P)$

```

apply (rel-tac)
apply (metis case-prodI)
apply (metis case-prodI)
apply (rule ext)
apply (auto)
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 (metis case-prodI)
apply (metis case-prodI)
apply (rule ext)
apply (auto)
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

```

```

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

```

```

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

```

```

theorem precond-left-zero:
  assumes  $out\alpha \# p \ p \neq false$ 
  shows  $(true ;; p) = true$ 
  using assms
  apply (simp add: outα-def upred-defs)
  apply (transfer, auto simp add: relcomp-unfold, rule ext, auto)
  apply (rename-tac p b)
  apply (subgoal-tac ∃ 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=λ -. b in spec)
  apply (simp)
done

```

7.4 Converse laws

```

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

```

```

lemma lit-convr [simp]:  $\ll v \gg^- = \ll v \gg$ 
by pred-tac

```

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

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

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

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

lemma *uop-convr* [*simp*]: $(\text{uop } f \ u)^- = \text{uop } f \ (u^-)$

by (*pred-tac*)

lemma *bop-convr* [*simp*]: $(\text{bop } f \ u \ v)^- = \text{bop } f \ (u^-) \ (v^-)$

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*: $\text{in}\alpha \nmid q \implies ((P \wedge q) ;; R) = (P ;; (q^- \wedge R))$

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*: $\text{in}\alpha \nmid r \implies (P ;; (Q \wedge r)) = ((P ;; Q) \wedge r)$

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*: $\text{out}\alpha \nmid q \implies (P ;; (q \wedge R)) = (P \wedge q^- ;; R)$

by (*simp* *add*: *seqr-pre-transfer* *unrest-convr-in* α)

lemma *seqr-pre-out*: $\text{out}\alpha \nmid p \implies ((p \wedge Q) ;; R) = (p \wedge (Q ;; R))$

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=λ-.y in spec, simp)
done

```

```

lemma seqr-true-lemma:
  (P = (¬ (¬ P ;; true))) = (P = (P ;; true))
  apply (rel-tac)
  apply (rule ext)
  apply (auto)
  apply (metis case-prodI)
  apply (rule ext)
  apply (auto)
  apply (metis case-prodI)
done

```

```

lemma shEx-lift-seq [uquant-lift]:
  ((∃ x · P(x)) ;; (∃ y · Q(y))) = (∃ x · ∃ y · P(x) ;; Q(y))
  by pred-tac

```

end

7.5 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 ⇒ 'b ⇒ 'c (infix wp 60)

```

```

definition wp-upred :: ('α, 'β) relation ⇒ 'β condition ⇒ 'α condition where
wp-upred Q r = [¬ (Q ;; ¬ [r]_<)]_<

```

```

ad hoc overloading
  wwp wp-upred

```

```

declare wp-upred-def [urel-defs]

```

```

theorem wp-assigns-r [wp]:
  (assigns-r σ) wp r = σ † r
  by rel-tac

```

```

theorem wp-skip-r [wp]:
  If wp r = r
  by rel-tac

```

```

theorem wp-true [wp]:
  r ≠ true ⇒ true wp r = false
  by rel-tac

```

```

theorem wp-conj [wp]:

```

$P \text{ wp } (q \wedge r) = (P \text{ wp } q \wedge P \text{ wp } r)$
by *rel-tac*

theorem *wp-seq-r* [*wp*]: $(P ;; Q) \text{ wp } r = P \text{ wp } (Q \text{ wp } r)$
by *rel-tac*

theorem *wp-cond* [*wp*]: $(P \triangleleft b \triangleright Q) \text{ wp } r = ((b \Rightarrow P \text{ wp } r) \wedge ((\neg b) \Rightarrow Q \text{ wp } r))$
by *rel-tac*

end

8 UTP Theories

theory *utp-theory*
imports *utp-rel*
begin

type-synonym $'\alpha$ *Healthiness-condition* = $'\alpha \text{ upred} \Rightarrow '\alpha \text{ upred}$

definition
 $\text{Healthy}::'\alpha \text{ upred} \Rightarrow '\alpha \text{ Healthiness-condition} \Rightarrow \text{bool}$ (**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 $k = \text{VAR } \text{boyle-}k$
definition $p = \text{VAR } \text{boyle-}p$
definition $V = \text{VAR } \text{boyle-}V$

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

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

lemma $B\text{-idempotent}$:

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

by $pred\text{-tac}$

lemma $B\text{-monotone}$:

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

by $pred\text{-tac}$

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 simplification of its definitional equation and then application of the predicate tactic.

lemma $B\text{-}\varphi_1$: φ_1 is B

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

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: $\varphi_2\text{-def}$*)

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

by $pred\text{-tac}$

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

by $pred\text{-tac}$

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

by $pred\text{-tac}$

also have ... = φ_1

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

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*)

type-synonym *'α alphabet-d* = *'α alpha-d-scheme alphabet*

type-synonym (*'a*, *'α*) *uvar-d* = (*'a*, *'α alphabet-d*) *uvar*

type-synonym (*'α*, *'β*) *relation-d* = (*'α alphabet-d*, *'β alphabet-d*) *relation*

type-synonym *'α hrelation-d* = *'α 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* :: (*'α*, *'β*) *relation* \Rightarrow (*'α*, *'β*) *relation-d* ($\lceil _ \rceil_D$) **is**
 $\lambda P (A, A'). P \text{ (more } A, \text{ more } A') .$

lift-definition *drop-desr* :: (*'α*, *'β*) *relation-d* \Rightarrow (*'α*, *'β*) *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*::(*'α*, *'β*) *relation* \Rightarrow (*'α*, *'β*) *relation* \Rightarrow (*'α*, *'β*) *relation-d* (**infixl** $\vdash 60$)
where (*P* \vdash *Q*) = ($\$ok \wedge \lceil P \rceil_D \Rightarrow \$ok' \wedge \lceil Q \rceil_D$)

definition *skip-d* :: *'α hrelation-d* (*II*_{*D*})
where *II*_{*D*} \equiv (*true* \vdash *II*)

definition *assigns-d* :: *'α usubst* \Rightarrow *'α hrelation-d*
where *assigns-d* σ = (*true* \vdash *assigns-r* σ)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10.2 Design laws

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

lemma $lift\text{-}dists$ [$simp$]:
 $\lfloor true \rfloor_D = true$
 $\lfloor \neg P \rfloor_D = (\neg \lfloor P \rfloor_D)$
 $\lfloor P \wedge Q \rfloor_D = (\lfloor P \rfloor_D \wedge \lfloor Q \rfloor_D)$
by ($pred\text{-}tac$) $+$

lemma $lift\text{-}dist\text{-}seq$ [$simp$]:
 $\lfloor P ;; Q \rfloor_D = (\lfloor P \rfloor_D ;; \lfloor Q \rfloor_D)$
by ($rel\text{-}tac$, $metis$ $alpha\text{-}d.select\text{-}conv(2)$)

lemma $design\text{-}refine$:
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*]: $(P \vdash Q) \llbracket \text{false}/\$ok \rrbracket = \text{true}$
 by (*simp add: design-def usubst*)

theorem *design-pre* [*simp*]: $\text{pre}_D(P \vdash Q) = P$
 by *pred-tac*

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

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

proof –

have $(\text{true} ;; (P \vdash Q)) = (\exists \text{ ok}_0 \cdot \text{true} \llbracket \llcorner \text{ok}_0 \gg / \$ok' \rrbracket ;; (P \vdash Q) \llbracket \llcorner \text{ok}_0 \gg / \$ok \rrbracket)$
 by (*subst segr-middle[of ok], simp-all*)
 also have $\dots = ((\text{true} \llbracket \text{false}/\$ok' \rrbracket ;; (P \vdash Q) \llbracket \text{false}/\$ok \rrbracket) \vee (\text{true} \llbracket \text{true}/\$ok' \rrbracket ;; (P \vdash Q) \llbracket \text{true}/\$ok \rrbracket))$
 by (*simp add: disj-comm false-alt-def true-alt-def*)
 also have $\dots = ((\text{true} \llbracket \text{false}/\$ok' \rrbracket ;; \text{true}_h) \vee (\text{true} ;; ((P \vdash Q) \llbracket \text{true}/\$ok \rrbracket)))$
 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*:

$((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (((\neg (\neg P1) ;; \text{true})) \wedge \neg (Q1 ;; (\neg P2))) \vdash (Q1 ;; Q2))$
 using *assms*

proof –

have $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (\exists \text{ ok}_0 \cdot ((P1 \vdash Q1) \llbracket \llcorner \text{ok}_0 \gg / \$ok' \rrbracket ;; (P2 \vdash Q2) \llbracket \llcorner \text{ok}_0 \gg / \$ok \rrbracket))$
 by (*rule segr-middle, simp*)
 also have \dots
 $= (((P1 \vdash Q1) \llbracket \text{false}/\$ok' \rrbracket ;; (P2 \vdash Q2) \llbracket \text{false}/\$ok \rrbracket) \vee ((P1 \vdash Q1) \llbracket \text{true}/\$ok' \rrbracket ;; (P2 \vdash Q2) \llbracket \text{true}/\$ok \rrbracket))$
 by (*simp add: true-alt-def false-alt-def, pred-tac*)
 also from *assms*
 have $\dots = (((\$ok \wedge \lceil P1 \rceil_D \Rightarrow \lceil Q1 \rceil_D) ;; (\lceil P2 \rceil_D \Rightarrow \$ok' \wedge \lceil Q2 \rceil_D)) \vee ((\neg (\$ok \wedge \lceil P1 \rceil_D)) ;; \text{true}))$
 by (*simp add: design-def usubst unrest, pred-tac*)
 also have $\dots = ((\neg \$ok ;; \text{true}_h) \vee (\neg \lceil P1 \rceil_D ;; \text{true}) \vee (\lceil Q1 \rceil_D ;; \neg \lceil P2 \rceil_D) \vee (\$ok' \wedge (\lceil Q1 \rceil_D ;; \lceil Q2 \rceil_D)))$
 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 *design-skip-idem* [*simp*]:

$(\Pi_D ;; \Pi_D) = \Pi_D$
 by (*simp add: skip-d-def urel-defs, pred-tac*)

theorem *design-composition-cond*:

assumes *outα* $\nmid p1$
 shows $((p1 \vdash Q1) ;; (P2 \vdash Q2)) = ((p1 \wedge \neg (Q1 ;; (\neg P2))) \vdash (Q1 ;; Q2))$
 using *assms*

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

theorem *design-composition-wp*:

fixes $Q1\ Q2 :: 'a\ hrelation$

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

using *assms*

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

theorem *design-wp [wp]*:

$([p]_{<} \vdash Q)\ wp_D\ r = (p \wedge Q\ wp\ r)$

by *rel-tac*

theorem *wpd-seq-r*:

fixes $Q1\ Q2 :: 'a\ hrelation$

shows $([p1]_{<} \vdash Q1 ;; [p2]_{<} \vdash Q2)\ wp_D\ r = ([p1]_{<} \vdash Q1)\ wp_D\ ([p2]_{<} \vdash Q2)\ wp_D\ r$

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

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

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

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

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

assumes $out\alpha \nVdash p$

shows $(p \vdash Q ;; II_D) = (p \vdash Q)$

using *assms*

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

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

$([P]_D ;; [II]_D) = [P]_D$

$([II]_D ;; [P]_D) = [P]_D$

by *rel-tac rel-tac*

10.3 H1: No observation is allowed before initiation

lemma *H1-idem*:

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

by *pred-tac*

lemma *H1-monotone*:

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

by *pred-tac*

lemma *H1-design-skip*:

$H1\ (II) = II_D$

by *rel-tac*

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

theorem *H1-algebraic-intro*:

assumes

$(true_h ;; R) = true_h$

$(II_D ;; R) = R$

shows R is *H1*

proof –

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

```

also have ... = (H1(II) ;; R)
  by (simp add: H1-design-skip)
also have ... = ($ok  $\Rightarrow$  II) ;; R)
  by (simp add: H1-def)
also have ... = (( $\neg$  $ok ;; R)  $\vee$  R)
  by (simp add: impl-alt-def seqr-or-distl)
also have ... = ((( $\neg$  $ok ;; trueh) ;; R)  $\vee$  R)
  by (simp add: precondition-right-unit unrest)
also have ... = (( $\neg$  $ok ;; trueh)  $\vee$  R)
  by (metis assms(1) seqr-assoc)
also have ... = ($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 (simp add: ok-def, pred-tac, simp add: in-var-def, metis alpha-d.select-convs(1) fst-conv)

```

theorem *H1-left-zero*:

```

assumes P is H1
shows (trueh ;; P) = trueh

```

proof –

```

from assms have (trueh ;; P) = (trueh ;; ($ok  $\Rightarrow$  P))
  by (simp add: H1-def Healthy-def')
also from assms have ... = (trueh ;; ( $\neg$  $ok  $\vee$  P))
  by (simp add: impl-alt-def)
also from assms have ... = ((trueh ;;  $\neg$  $ok)  $\vee$  (trueh ;; 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 :: 'α hrelation-d
assumes P is H1
shows (IID ;; P) = P

```

proof –

```

have (IID ;; 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 ;; trueh) ;; P)  $\vee$  P)
  by (simp add: precondition-right-unit unrest)
also have ... = (( $\neg$  $ok ;; (trueh ;; 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  $\longleftrightarrow$  (trueh ;; P) = trueh  $\wedge$  (IID ;; 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 $\dots = ((\neg \$ok) ;; true_h)$

by (*metis H1-left-zero assms seqr-assoc*)

also have $\dots = (\neg \$ok)$

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 $\dots = (P ;; ((\$ok \Rightarrow \$ok \wedge \$ok') \wedge \lceil II \rceil_D))$

by *rel-tac*

also have $\dots = ((P ;; (\neg \$ok \wedge \lceil II \rceil_D)) \vee (P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))))$

by *rel-tac*

also have $\dots = (P^f \vee (P^t \wedge \$ok'))$

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 (*rel-tac, metis (mono-tags, lifting) alpha-d.surjective alpha-d.update-convs(1)*)

also have $\dots = P^f$

by (*metis one-point out-var-uvar ouvar-def unrest-false uvar-ok*)

finally show *?thesis* .

qed

moreover have $(P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))) = (P^t \wedge \$ok')$

proof –

have $(P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))) = (P ;; (\$ok \wedge II))$

by (*rel-tac, metis alpha-d.equality*)

also have $\dots = (P^t \wedge \$ok')$

by (*rel-tac, metis (full-types) alpha-d.surjective alpha-d.update-convs(1)*)

finally show *?thesis* .

qed

ultimately show *?thesis*

by *simp*

qed

finally show *?thesis* .

qed

lemma *H2-split*:

shows $H2(P) = (P^f \vee (P^t \wedge \$ok'))$

by (*simp add: H2-def J-split*)

theorem *H2-equivalence*:

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

proof –

have $'P \Leftrightarrow (P ;; J)'$ $\iff 'P \Leftrightarrow (P^f \vee (P^t \wedge \$ok'))'$

by (*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 (*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 (*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*:

$H2(P \vdash Q) = P \vdash Q$

using *assms*

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

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 seqr-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, simp add: iuvar-def*)

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 *seqr-or-distl* **by** *blast*

also have ... = $((H2 (\neg \$ok)) \vee H2(P))$

by (simp add: H2-def)

also have ... = $((\neg \$ok) \vee H2(P))$

by (simp add: H2-not-okay)

also have ... = $H1(H2(P))$

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-design:

assumes P is H1 P is H2

shows $P = pre_D(P) \vdash post_D(P)$

proof –

from assms have $P = (\$ok \Rightarrow H2(P))$

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

also have ... = $(\$ok \Rightarrow (P^f \vee (P^t \wedge \$ok')))$

by (metis H2-split)

also have ... = $(\$ok \wedge (\neg P^f) \Rightarrow \$ok' \wedge P^t)$

by pred-tac

also have ... = $(\$ok \wedge (\neg P^f) \Rightarrow \$ok' \wedge \$ok \wedge P^t)$

by pred-tac

also have ... = $(\$ok \wedge \lceil pre_D(P) \rceil_D \Rightarrow \$ok' \wedge \$ok \wedge \lceil post_D(P) \rceil_D)$

by (simp add: ok-post ok-pre)

also have ... = $(\$ok \wedge \lceil pre_D(P) \rceil_D \Rightarrow \$ok' \wedge \lceil post_D(P) \rceil_D)$

by pred-tac

also from assms have ... = $pre_D(P) \vdash post_D(P)$

by (simp add: 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 H3-iff-pre:

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

proof –

have $(P \vdash Q ;; II_D) = (P \vdash Q ;; true \vdash II)$

by (simp add: skip-d-def)

also have ... = $(\neg (\neg P ;; true) \wedge \neg (Q ;; \neg true)) \vdash (Q ;; II)$

by (fact design-composition)

also have ... = $(\neg (\neg P ;; true) \wedge \neg (Q ;; \neg true)) \vdash Q$

by simp

also have ... = $(\neg (\neg P ;; true)) \vdash Q$

by pred-tac

finally have $P \vdash Q \text{ is } H3 \iff P \vdash Q = (\neg (\neg P ;; true)) \vdash Q$

by (metis H3-def Healthy-def')
 also have ... $\longleftrightarrow P = (\neg (\neg P ;; true))$
 by (metis design-pre)
 also have ... $\longleftrightarrow P = (P ;; true)$
 by (simp add: segr-true-lemma)
 finally show ?thesis .
 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-design 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-def)
 also have ... $= (\exists ok_0 \cdot ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D) \llbracket \ll ok_0 \gg / \$ok' \rrbracket ;; (true \vdash II) \llbracket \ll ok_0 \gg / \$ok \rrbracket)$
 by (subst segr-middle[of ok], simp-all)
 also have ... $= (((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D) \llbracket false / \$ok' \rrbracket ;; (true \vdash II) \llbracket false / \$ok \rrbracket)$
 $\vee (((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D) \llbracket true / \$ok' \rrbracket ;; (true \vdash II) \llbracket true / \$ok \rrbracket)$
 by (simp add: disj-comm false-alt-def true-alt-def)
 also have ... $= ((\neg \$ok \wedge \lceil II \rceil_D ;; true) \vee (\lceil II \rceil_D ;; \$ok' \wedge \lceil II \rceil_D))$
 by (simp add: usubst unrest design-def iuvar-def ouvar-def)
 also have ... $= II_D$
 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 $out\alpha \nVdash p$
 shows $H3(p \vdash Q) = p \vdash Q$
 using assms
 by (simp add: H3-def)

theorem H1-H3-is-design:
 assumes P is $H1 P$ is $H3$
 shows $P = pre_D(P) \vdash post_D(P)$
 by (metis H1-H2-is-design H2-H3-absorb Healthy-def' assms)

theorem *H1-H3-is-normal-design*:

assumes *P is H1 P is H3*

shows $P = \lceil \lfloor \text{pre}_D(P) \rfloor \rfloor \vdash \text{post}_D(P)$

by (*metis H1-H3-is-design H3-iff-pre assms(1) assms(2) drop-pre-inv precond-equiv*)

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 $(P ;; Q)\ \text{wp}_D\ r = P\ \text{wp}_D\ (Q\ \text{wp}_D\ r)$

by (*metis (no-types, lifting) H1-H3-commute H1-H3-is-normal-design H1-idem Healthy-def' assms(1) assms(2) 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)

We describe the partition of a state space into a left and right part for parallel composition. If we want n-ary partitions this could alternatively use a list. But then the type-system would not record the number of state-spaces present, but perhaps we don't want that ...

record *'a partition* =

left-alpha :: *'a*

right-alpha :: *'a*

definition *design-par* :: $('a, 'b)\ relation-d \Rightarrow ('a, 'b)\ relation-d \Rightarrow ('a, 'b)\ relation-d$ (**infixr** \parallel 85)

where

$P \parallel Q = ((\text{pre}_D(P) \wedge \text{pre}_D(Q)) \vdash (\text{post}_D(P) \wedge \text{post}_D(Q)))$

declare *design-par-def* [*upred-defs*]

lemma *parallel-zero*: $P \parallel \text{true} = \text{true}$

proof –

have $P \parallel \text{true} = (\text{pre}_D(P) \wedge \text{pre}_D(\text{true})) \vdash (\text{post}_D(P) \wedge \text{post}_D(\text{true}))$

by (*simp add: design-par-def*)

also have $\dots = (\text{pre}_D(P) \wedge \text{false}) \vdash (\text{post}_D(P) \wedge \text{true})$

by *rel-tac*

also have $\dots = \text{true}$

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-design assms conj-idem design-par-def*)

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.

type-synonym $('α, 'β)$ *merge-d* = $('α$ *partition*, $'β)$ *relation-d*

lift-definition $U0 :: ('α, 'α$ *partition*) *relation-d* **is**
 $\lambda (A, A'). \text{des-ok } A' = \text{des-ok } A \wedge \text{left-alpha } (\text{alpha-d.more } A') = \text{alpha-d.more } A .$

lift-definition $U1 :: ('α, 'α$ *partition*) *relation-d* **is**
 $\lambda (A, A'). \text{des-ok } A' = \text{des-ok } A \wedge \text{right-alpha } (\text{alpha-d.more } A') = \text{alpha-d.more } A .$

Parallel by merge

definition *design-par-by-merge* ::
 $('α, 'β)$ *relation-d* $\Rightarrow ('β, 'γ)$ *merge-d* $\Rightarrow ('α, 'β)$ *relation-d* $\Rightarrow ('α, 'γ)$ *relation-d* (**infixr** \parallel - 85)
where $P \parallel_M Q = (((P ;; U0) \parallel (Q ;; U1)) ;; M)$

end

12 Reactive processes

theory *utp-reactive*
imports *utp-concurrency*
begin

Following the way of UTP to describe reactive processes, more observational variables are needed to record the interaction with the environment. Three observational variables are defined for this subset of relations: *wait*, *tr* and *ref*. The boolean variable *wait* records if the process is waiting for an interaction or has terminated. *tr* records the list (trace) of interactions the process has performed so far. The variable *ref* contains the set of interactions (events) the process may refuse to perform.

In this section, we introduce first some preliminary notions, useful for trace manipulations. The definitions of reactive process alphabets and healthiness conditions are also given. Finally, proved lemmas and theorems are listed.

12.1 Preliminaries

type-synonym $'α$ *trace* = $'α$ *list*

fun *list-diff*:: $'α$ *list* $\Rightarrow 'α$ *list* $\Rightarrow 'α$ *list option* **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 *uvar-wait* [simp]: uvar wait
by (unfold-locales, simp-all add: wait-def)

lemma *uvar-tr* [simp]: uvar tr
by (unfold-locales, simp-all add: tr-def)

lemma *uvar-ref* [simp]: uvar ref
by (unfold-locales, simp-all add: ref-def)

Note that we define here the class of UTP alphabets that contain *wait*, *tr* and *ref*, or, in other words, we define here the class of reactive process alphabets.

type-synonym $(' \vartheta, ' \alpha)$ alphabet-rp = $(' \vartheta, ' \alpha)$ alpha-rp-scheme alphabet

type-synonym $(' \vartheta, ' \alpha, ' \beta)$ relation-rp = $((' \vartheta, ' \alpha)$ alphabet-rp, $(' \vartheta, ' \beta)$ alphabet-rp) relation

type-synonym $(' \vartheta, ' \sigma)$ predicate-rp = $(' \vartheta, ' \sigma)$ alphabet-rp upred

definition *R1-def* [upred-defs]: $R1 (P) = (P \wedge (\$tr \leq_u \$tr'))$

definition *R2-def* [upred-defs]: $R2 (P) = (P[\langle \rangle / \$tr][(\$tr' - \$tr) / \$tr'] \wedge (\$tr \leq_u \$tr'))$

definition *skip-rea-def* [urel-defs]: $II_r = (II \vee (\neg \$ok \wedge \$tr \leq_u \$tr'))$

There are two versions of $R3$ in the UTP book. Here we opt for the version that works for CSP

definition *R3-def* [urel-defs]: $R3c (P) = (II_r \triangleleft \&wait \triangleright P)$

definition $R(P) = R1(R2(R3c(P)))$

lemma $R1\text{-idem}$: $R1(R1(P)) = R1(P)$
by *pred-tac*

lemma $R2\text{-idem}$: $R2(R2(P)) = R2(P)$
by (*pred-tac*)

lemma $tr\text{-prefix-as-concat}$: $(xs \leq_u ys) = (\exists zs \cdot ys =_u xs \hat{\ }_u \ll zs \gg)$
by (*rel-tac*, *simp add: less-eq-list-def prefixeq-def*)

lemma $R2\text{-form}$:
 $R2(P) = (\exists tt \cdot P[\langle \rangle / \$tr][\ll tt \gg / \$tr']) \wedge \$tr' =_u \$tr \hat{\ }_u \ll tt \gg$
by (*rel-tac*, *metis prefix-subst strict-prefixE*)

lemma $uconc\text{-left-unit}$ [*simp*]: $\langle \rangle \hat{\ }_u e = e$
by *pred-tac*

lemma $uconc\text{-right-unit}$ [*simp*]: $e \hat{\ }_u \langle \rangle = e$
by *pred-tac*

This laws is proven only for homogeneous relations, can it be generalised?

lemma $R2\text{-segr-form}$:
fixes $P Q :: ('\theta, '\alpha, '\alpha) \text{ relation-rp}$
shows $(R2(P) ;; R2(Q)) =$
 $(\exists tt_1 \cdot \exists tt_2 \cdot ((P[\langle \rangle / \$tr][\ll tt_1 \gg / \$tr']) ;; (Q[\langle \rangle / \$tr][\ll tt_2 \gg / \$tr'])))$
 $\wedge (\$tr' =_u \$tr \hat{\ }_u \ll tt_1 \gg \hat{\ }_u \ll tt_2 \gg))$

proof –

have $(R2(P) ;; R2(Q)) = (\exists tr_0 \cdot (R2(P))[\ll tr_0 \gg / \$tr']) ;; (R2(Q))[\ll tr_0 \gg / \$tr'])$
by (*subst segr-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: segr-pre-out segr-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 $R2\text{-segr-distribute}$:
fixes $P Q :: (''\theta, '\alpha, '\alpha) \text{ relation-rp}$

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: R2-def usubst unrest, rel-tac*)
also have ... =
 $((\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] ;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr'])(\langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle) / \$tr')$
 $\wedge \$tr' - \$tr =_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr)$
by (*subst subst-eq-replace, simp*)
also have ... =
 $((\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] ;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr']$
 $\wedge \$tr' - \$tr =_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr)$
by (*simp add: usubst unrest*)
also have ... =
 $(\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] ;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr']$
 $\wedge (\$tr' - \$tr =_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle \wedge \$tr' \geq_u \$tr))$
by *pred-tac*
also have ... =
 $((\exists tt_1 \cdot \exists tt_2 \cdot (P[\langle \rangle / \$tr][\langle tt_1 \rangle / \$tr'] ;; Q[\langle \rangle / \$tr][\langle tt_2 \rangle / \$tr']$
 $\wedge \$tr' =_u \$tr \hat{^}_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle))$
proof –
have $\bigwedge tt_1 tt_2. (((\$tr' - \$tr =_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr) :: ('v, 'a, 'a) \text{ relation-rp})$
 $= (\$tr' =_u \$tr \hat{^}_u \langle tt_1 \rangle \hat{^}_u \langle tt_2 \rangle)$
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 *R3c-idem*: $R3c(R3c(P)) = R3c(P)$
by *rel-tac*

end