A Shallow Model of the UTP in Isabelle/HOL

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

February 5, 2016

Contents

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 is this shallow model are simple 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 a thus a strong link between alphabets and variables in this model. Variable 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* $f\ (\lambda\ \text{-}\ .\ v)$

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

syntax *-VAR* :: *id* ⇒ (*'a*, *'r*) *uvar* (*VAR -*)
translations *VAR x* => (| *var-lookup* = *x*, *var-update* = *-update-name x* |)

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

locale *semi-uvar* =
  fixes *x* :: (*'a*, *'r*) *uvar*
  — Application of two updates should correspond to the composition of update functions
  assumes *var-update-comp*: *var-update x f* (*var-update x g σ*) = *var-update x* (*f* ∘ *g*) *σ*
  — Updating a variable's value to the one it already has is ineffectual
  and *var-update-eta*: *var-update x* (λ-. *var-lookup x σ*) *σ* = *σ*

locale *uvar* = *semi-uvar* +
  assumes *var-update-lookup*: *var-lookup x* (*var-update x f σ*) = *f* (*var-lookup x σ*)

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

lemma *uvar-semi-var* [*simp*]: *uvar x* ⟹ *semi-uvar x*
  by (*simp add*: *uvar-def*)

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

definition *uvar-indep* :: (*'a*, *'r*) *uvar* ⇒ (*'b*, *'r*) *uvar* ⇒ *bool* (**infix** ⋈ *50*) **where**
x ⋈ *y* ⟷ (∀ *f g σ*. *var-update x f* (*var-update y g σ*) = *var-update y g* (*var-update x f σ*))

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

lemma *uvar-indep-sym*: *x* ⋈ *y* ⟹ *y* ⋈ *x*
  by (*simp add*: *uvar-indep-def*)

lemma *uvar-indep-comm*:
  assumes *x* ⋈ *y*
  shows *var-update x f* (*var-update y g σ*) = *var-update y g* (*var-update x f σ*)
  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 *uvar x x* ⋈ *y*
  shows *var-lookup x* (*var-update y f σ*) = *var-lookup x σ*
proof –
  have *var-lookup x* (*var-update y f σ*) = *var-lookup x* (*var-update y f* (*var-update x* (λ-. *var-lookup x σ*) *σ*))

by (*simp add*: *assms*(*1*))
also have ... = *var-lookup* x (*var-update* x (λ-. *var-lookup* x σ) (*var-update* y f σ))
using *assms*(*2*) **by** (*auto simp add*:*uvar-indep-def*)
also have ... = *var-lookup* x σ
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* :: $('a, '\alpha)$ *uvar* $\Rightarrow$ $('a, '\alpha \times '\beta)$ *uvar* **where**
in-var x = ⦇ *var-lookup* = *var-lookup* $x \circ$ *fst*, *var-update* = (λ f (A, A'). (*var-update* x f A, A')) ⦈

definition *out-var* :: $('a, '\beta)$ *uvar* $\Rightarrow$ $('a, '\alpha \times '\beta)$ *uvar* **where**
out-var x = ⦇ *var-lookup* = *var-lookup* $x \circ$ *snd*, *var-update* = (λ f (A, A'). (A, *var-update* x f A')) ⦈

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

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

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

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 the universal alphabet (Σ) to be a variable with identity for both the lookup and update functions. Effectively this is just a function directly on the alphabet type.

definition *univ-alpha* :: ($'\alpha$, $'\alpha$) *uvar* (Σ) **where**
univ-alpha = (| *var-lookup* = *id*, *var-update* = *id* |)

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

definition *uvar-comp* :: ($'a$, $'\alpha$) *uvar* $\Rightarrow$ ($'b$, $'\alpha$) *uvar* $\Rightarrow$ ($'a \times 'b$, $'\alpha$) *uvar* (**infix** $\circ_v$ *35*) **where**
uvar-comp x y = (| *var-lookup* = λ *A*. (*var-lookup x A*, *var-lookup y A*)
  , *var-update* = λ *f*. *var-update x* (λ *a*. *fst* (*f* (*a*, *undefined*))) $\circ$
    var-update y (λ *b*. *snd* (*f* (*undefined*, *b*))) |)

nonterminal *svar*

syntax
  -svar :: *id* $\Rightarrow$ *svar* (- [*999*] *999*)
  -spvar :: *id* $\Rightarrow$ *svar* (&- [*999*] *999*)
  -sinvar :: *id* $\Rightarrow$ *svar* ($- [*999*] *999*)
  -soutvar :: *id* $\Rightarrow$ *svar* ($-´ [*999*] *999*)

translations
  -svar x => *x*
  -spvar x => *x*
  -sinvar x == *CONST in-var x*
  -soutvar x == *CONST out-var x*

end

1.1 Deep UTP variables

theory *utp-dvar*
  imports *utp-var*
begin

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

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

1.2 Cardinalities

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

datatype $\mathit{ucard} = \mathit{fin}\ \mathit{nat} \mid \mathit{aleph0}\ (\aleph_0) \mid \mathit{cont}\ (\mathrm{c})$

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

type-synonym $\mathit{uuniv} = \mathit{nat}\ \mathit{set}$

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

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

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

syntax
$\mathit{\text{-}ucard} :: \mathit{type} \Rightarrow \mathit{ucard}\ (\mathit{UCARD}'(\text{-}'))$

translations
$\mathit{UCARD}({'a}) == \mathit{CONST}\ \mathit{ucard\text{-}of}\ (\mathit{TYPE}({'a}))$

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

lemma *ucard-of-countably-infinite* [*simp*]:
$[\![\ \mathit{countable}(\mathit{UNIV} :: {'a}{::}\mathit{continuum}\ \mathit{set});\ \mathit{infinite}(\mathit{UNIV} :: {'a}\ \mathit{set})\ ]\!] \Longrightarrow \mathit{UCARD}({'a}) = \aleph_0$
by (*simp add*: *ucard-of-def*)

lemma *ucard-of-uncountably-infinite* [*simp*]:
$\mathit{uncountable}\ (\mathit{UNIV} :: {'a}\ \mathit{set}) \Longrightarrow \mathit{UCARD}({'a} :: \mathit{continuum}) = \mathrm{c}$
apply (*simp add*: *ucard-of-def*)

using *countable-finite* **apply** *blast*
done

1.3 Injection functions

definition *uinject-finite* :: *'a::finite* ⇒ *uuniv* **where**
uinject-finite x = {*to-nat-fin x*}

definition *uinject-aleph0* :: *'a::{countable, infinite}* ⇒ *uuniv* **where**
uinject-aleph0 x = {*to-nat-bij x*}

definition *uinject-continuum* :: *'a::{continuum, infinite}* ⇒ *uuniv* **where**
uinject-continuum x = *to-nat-set-bij x*

definition *uinject* :: *'a::continuum* ⇒ *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* ⇒ *'a::continuum* **where**
uproject = *inv uinject*

lemma *uinject-finite*:
  finite (*UNIV* :: *'a::continuum set*) ⟹ *uinject* = (λ *x* :: *'a*. {*to-nat-fin x*})
  by (*rule ext*, *auto simp add*: *uinject-def*)

lemma *uinject-uncountable*:
  uncountable (*UNIV* :: *'a::continuum set*) ⟹ (*uinject* :: *'a* ⇒ *uuniv*) = *to-nat-set*
  by (*rule ext*, *auto simp add*: *uinject-def countable-finite*)

lemma *card-finite-lemma*:
  assumes *finite* (*UNIV* :: *'a set*)
  shows *x* < *card* (*UNIV* :: *'a set*) ⟷ *x* ≤ *card* (*UNIV* :: *'a set*) − *Suc 0*
proof −
  have *card* (*UNIV* :: *'a set*) > *0*
    by (*simp add*: *assms finite-UNIV-card-ge-0*)
  thus *?thesis*
    by *linarith*
qed

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

lemma *uinject-bij*:
  bij-betw (*uinject* :: *'a::continuum* ⇒ *uuniv*) *UNIV* 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*]:
$x \in \mathcal{U}(UCARD('a::continuum)) \Longrightarrow$ *uinject* ((*uproject* :: *nat set* $\Rightarrow$ $'a$) x) $= x$
by (*metis bij-betw-inv-into-right uinject-bij uproject-def*)

1.4 Deep variables

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

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

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

typedef *vstore* = $\{f$ :: *dname* $\Rightarrow$ *uuniv*. $\forall\ x.\ f(x) \in \mathcal{U}($*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* = $\{x$ :: *dname*. *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 $\lambda\ n.$ (| *dname-name* $= n$, *dname-card* $= UCARD('a)$ |)
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 λ x s. (*uproject* :: *uuniv* $\Rightarrow$ $'a$) $(s(x))$.

lift-definition *vstore-put* :: $('a::continuum)$ *dvar* $\Rightarrow$ $'a$ $\Rightarrow$ *vstore* $\Rightarrow$ *vstore*
is λ (x :: *dname*) (v :: $'a$) f . $f(x :=$ *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 (λ -. *vstore-lookup* x s) s = s
  apply (*simp add*: *vstore-upd-def*, *transfer*, *auto*)
  apply (*metis Domainp-iff dvar.domain fun-upd-idem-iff uproject-inv*)
done

lemma *vstore-lookup-put-diff-var* [*simp*]:
  assumes *dvar-name* $x \neq$ *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-parm*: *upd-vstore* f s = *upd-vstore* (λ -. f (*get-vstore* s)) s

definition *dvar-lift* :: $'a$::*continuum dvar* $\Rightarrow$ $('a, '\alpha::vst)$ *uvar* (-$\uparrow$ [*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* ($x\uparrow$)
  apply (*unfold-locales*, *simp-all add*: *dvar-lift-def*)
  apply (*subst upd-store-parm*)

apply (*simp*)
done

Deep variables with different names are independent

lemma *dvar-indep-diff-name*:
assumes *dvar-name* $x \neq$ *dvar-name* y
shows $x\uparrow \bowtie y\uparrow$
proof −
from *assms* **have** $\bigwedge f\ g.$ *vstore-upd* $x\ f \circ$ *vstore-upd* $y\ g =$ *vstore-upd* $y\ g \circ$ *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 $('t, '\alpha)$ *uexpr* = *UNIV* :: $('\alpha$ *alphabet* $\Rightarrow 't)$ *set* **..**

notation *Rep-uexpr* ($[\![\text{-}]\!]_e$)

lemma *uexpr-eq-iff*:
$e = f \longleftrightarrow (\forall\ b.\ [\![e]\!]_e\ b = [\![f]\!]_e\ b)$
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* :: $('t, '\alpha)$ *uvar* $\Rightarrow$ $('t, '\alpha)$ *uexpr* **is** *var-lookup* **.**

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

definition *dvar-exp* :: $'t$::*continuum dvar* $\Rightarrow$ $('t, '\alpha::vst)$ *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* :: $('t, '\alpha)$ *uvar* $\Rightarrow$ $('t, '\alpha \times '\beta)$ *uexpr*
where *iuvar* x = *var* (*in-var* x)

definition *ouvar* :: $('t, '\beta)$ *uvar* $\Rightarrow$ $('t, '\alpha \times '\beta)$ *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 λ v b. v **.**

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

lift-definition *uop* :: $('a \Rightarrow 'b) \Rightarrow ('a, '\alpha)$ *uexpr* $\Rightarrow ('b, '\alpha)$ *uexpr*
 is λ 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 λ 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 λ 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 :: $'a \Rightarrow 'a \Rightarrow 'b$ (**infixl** $=_u$ *50*)
 ueuvar :: $'v \Rightarrow 'p$
 uiuvar :: $'v \Rightarrow 'p$
 uouvar :: $'v \Rightarrow 'p$

adhoc-overloading
 ulit lit **and**
 ueuvar var **and**
 ueuvar dvar-exp **and**

uiuvar iuvar **and**
uiuvar idvar **and**
uouvar ouvar **and**
uouvar odvar

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

translations
&*x* == *CONST ueuvar x*
$*x* == *CONST uiuvar x*
$*x*´ == *CONST uouvar x*

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

instantiation *uexpr* :: (*plus*, *type*) *plus*
begin
definition *plus-uexpr-def*: *u* + *v* = *bop* (*op* +) *u v*
instance ..
end

Instantiating uminus also provides negation for predicates later

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

instantiation *uexpr* :: (*minus*, *type*) *minus*
begin
definition *minus-uexpr-def*: *u* − *v* = *bop* (*op* −) *u v*
instance ..
end

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

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

instantiation *uexpr* :: (*zero*, *type*) *zero*
begin
definition *zero-uexpr-def*: *0* = *lit 0*
instance ..
end

instantiation *uexpr* :: (*one*, *type*) *one*
begin
  definition *one-uexpr-def*: *1* = *lit 1*
instance ..

end

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

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

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

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

instance *uexpr* :: (*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*: $[\![numeral\ x]\!]_e\ b = numeral\ x$
  by (*induct x*, *simp-all add*: *plus-uexpr-def one-uexpr-def numeral.simps lit.rep-eq bop.rep-eq*)

lemma *numeral-uexpr-simp*: *numeral x* = «*numeral x*»
  by (*simp add*: *uexpr-eq-iff numeral-uexpr-rep-eq lit.rep-eq*)

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

adhoc-overloading
  ueq eq-upred

abbreviation *seq-filter* :: $'a$ *set* ⇒ $'a$ *list* ⇒ $'a$ *list* **where**
seq-filter A ≡ *filter* (λ *x*. *x* ∈ *A*)

nonterminal *utuple-args*

syntax
  -unil :: ($'a$ *list*, $'\alpha$) *uexpr* (⟨⟩)
  -ulist :: *args* => ($'a$ *list*, $'\alpha$) *uexpr* (⟨(-)⟩)
  -uappend :: ($'a$ *list*, $'\alpha$) *uexpr* ⇒ ($'a$ *list*, $'\alpha$) *uexpr* ⇒ ($'a$ *list*, $'\alpha$) *uexpr* (**infixr** $\hat{}_u$ *80*)
  -ulast :: ($'a$ *list*, $'\alpha$) *uexpr* ⇒ ($'a$, $'\alpha$) *uexpr* ($last_u$'(-'))
  -ufilter :: ($'a$ *list*, $'\alpha$) *uexpr* ⇒ ($'a$ *set*, $'\alpha$) *uexpr* ⇒ ($'a$ *list*, $'\alpha$) *uexpr* (**infixl** $\upharpoonright_u$ *75*)
  -uless :: ($'a$, $'\alpha$) *uexpr* ⇒ ($'a$, $'\alpha$) *uexpr* ⇒ (*bool*, $'\alpha$) *uexpr* (**infix** $<_u$ *50*)
  -uleq :: ($'a$, $'\alpha$) *uexpr* ⇒ ($'a$, $'\alpha$) *uexpr* ⇒ (*bool*, $'\alpha$) *uexpr* (**infix** $\leq_u$ *50*)
  -ugreat :: ($'a$, $'\alpha$) *uexpr* ⇒ ($'a$, $'\alpha$) *uexpr* ⇒ (*bool*, $'\alpha$) *uexpr* (**infix** $>_u$ *50*)
  -ugeq :: ($'a$, $'\alpha$) *uexpr* ⇒ ($'a$, $'\alpha$) *uexpr* ⇒ (*bool*, $'\alpha$) *uexpr* (**infix** $\geq_u$ *50*)
  -uempset :: ($'a$ *set*, $'\alpha$) *uexpr* ($\{\}_u$)
  -uset :: *args* => ($'a$ *set*, $'\alpha$) *uexpr* ($\{(-)\}_u$)
  -uunion :: ($'a$ *set*, $'\alpha$) *uexpr* ⇒ ($'a$ *set*, $'\alpha$) *uexpr* ⇒ ($'a$ *set*, $'\alpha$) *uexpr* (**infixl** $\cup_u$ *65*)
  -uinter :: ($'a$ *set*, $'\alpha$) *uexpr* ⇒ ($'a$ *set*, $'\alpha$) *uexpr* ⇒ ($'a$ *set*, $'\alpha$) *uexpr* (**infixl** $\cap_u$ *70*)
  -umem :: ($'a$, $'\alpha$) *uexpr* ⇒ ($'a$ *set*, $'\alpha$) *uexpr* ⇒ (*bool*, $'\alpha$) *uexpr* (**infix** $\in_u$ *50*)

-unmem :: ('a, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** $\notin_u$ 50)
-usubset :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** $\subset_u$ 50)
-usubseteq :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** $\subseteq_u$ 50)
-utuple :: ('a, 'α) uexpr ⇒ utuple-args ⇒ ('a * 'b, 'α) uexpr ((1'(-,/ -')$_u$))
-utuple-arg :: ('a, 'α) uexpr ⇒ utuple-args (-)
-utuple-args :: ('a, 'α) uexpr => utuple-args ⇒ utuple-args (-,/ -)
-uunit :: ('a, 'α) uexpr ('(')$_u$)
-ufst :: ('a × 'b, 'α) uexpr ⇒ ('a, 'α) uexpr (π_1'(-'))
-usnd :: ('a × 'b, 'α) uexpr ⇒ ('b, 'α) uexpr (π_2'(-'))
-uapply :: ('a ⇒ 'b, 'α) uexpr ⇒ utuple-args ⇒ ('b, 'α) uexpr (-(|-|)$_u$ [999,0] 999)

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

translations
⟨⟩ == ≪[]≫
⟨x, xs⟩ == CONST bop (op #) x ⟨xs⟩
⟨x⟩ == CONST bop (op #) x ≪[]≫
x ^$_u$ y == CONST bop (op @) x y
last$_u$(xs) == CONST uop CONST last xs
xs $\upharpoonright_u$ A == CONST bop CONST seq-filter A xs
x $<_u$ y == CONST bop (op <) x y
x $\leq_u$ y == CONST bop (op ≤) x y
x $>_u$ y == y $<_u$ x
x $\geq_u$ y == y $\leq_u$ x
{}$_u$ == ≪{}≫
{x, xs}$_u$ == CONST bop (CONST insert) x {xs}$_u$
{x}$_u$ == CONST bop (CONST insert) x ≪{}≫
A $\cup_u$ B == CONST bop Set.union A B
A $\cap_u$ B == CONST bop Set.inter A B
x $\in_u$ A == CONST bop (op ∈) x A
x $\notin_u$ A == CONST bop (op ∉) x A
A $\subset_u$ B == CONST bop (op ⊂) A B
A $\subseteq_u$ B == CONST bop (op ⊆) A B
()$_u$ == ≪()≫
(x, y)$_u$ == CONST bop (CONST Pair) x y
-utuple x (-utuple-args y z) == -utuple x (-utuple-arg (-utuple y z))
π_1(x) == CONST uop CONST fst x
π_2(x) == CONST uop CONST snd x
f(|x|)$_u$ == CONST bop CONST fun-apply f x
f(|x,y|)$_u$ == CONST bop CONST fun-apply f (x,y)$_u$

Lifting set intervals

syntax
-uset-atLeastLessThan :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ ('a set, 'α) uexpr ((1{-..<-}$_u$))
-uset-compr :: id ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr ⇒ ('b, 'α) uexpr ⇒ ('b set, 'α) uexpr ((1{-
:/ - |/ - •/ -}$_u$))

lift-definition *ZedSetCompr* ::
('a set, 'α) uexpr ⇒ ('a ⇒ (bool, 'α) uexpr × ('b, 'α) uexpr) ⇒ ('b set, 'α) uexpr
is λ A PF b. { snd (PF x) b | x. x ∈ A b ∧ fst (PF x) b} **.**

translations
{x..<y}$_u$ == CONST bop CONST atLeastLessThan x y
{x : A | P • F}$_u$ == CONST ZedSetCompr A (λ x. (P, F))

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

lemma *var-out-var*: *var* (*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 $x \sharp p$, if altering the value of x has no effect on the valuation of p. This is a sufficient notion to prove many laws that would ordinarily rely on an *fv* function.

consts
  unrest :: *'a* ⇒ *'b* ⇒ *bool*

syntax
  -unrest :: *svar* ⇒ *logic* ⇒ *logic* ⇒ *logic* (**infix** ♯ *20*)

translations
  -unrest x p == *CONST unrest x p*

named-theorems *unrest*

lift-definition *unrest-upred* :: (*'a*, *'α*) *uvar* ⇒ (*'b*, *'α*) *uexpr* ⇒ *bool*
is λ *x e*. ∀ *b v*. *e* (*var-update x v b*) = *e b* **.**

adhoc-overloading
  unrest unrest-upred

lemma *unrest-lit* [*unrest*]: *x* ♯ ≪*v*≫
  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*]: $\llbracket$ *uvar x*; $x \bowtie y$ $\rrbracket \Longrightarrow y \sharp$ *var x*
  by (*transfer*, *auto*)

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

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

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

lemma *unrest-eq* [*unrest*]: $\llbracket x \sharp u;\ x \sharp v \rrbracket \Longrightarrow x \sharp u =_u v$
  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** † *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 $'\alpha$ *usubst* = $'\alpha$ *alphabet* $\Rightarrow$ $'\alpha$ *alphabet*

lift-definition *subst* :: $'\alpha$ *usubst* $\Rightarrow$ $('a, '\alpha)$ *uexpr* $\Rightarrow$ $('a, '\alpha)$ *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* :: $'\alpha$ *usubst* $\Rightarrow$ $'v \Rightarrow$ $('a, '\alpha)$ *uexpr* $\Rightarrow$ $'\alpha$ *usubst*

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

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

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

Lookup the expression associated with a variable in a substitution

lift-definition *usubst-lookup* :: $'\alpha$ *usubst* $\Rightarrow$ $('a, '\alpha)$ *uvar* $\Rightarrow$ $('a, '\alpha)$ *uexpr* $(\langle\text{-}\rangle_s)$
is $\lambda\ \sigma\ x\ b.$ *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* $(\lceil\text{-}\rceil_s)$ **where**
$\lceil\sigma\rceil_s = (\lambda\ (A, A').\ (\sigma\ A, A'))$

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

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

syntax
  -smaplet :: [*svar*, *'a*] => *smaplet* (- /$\mapsto_s$/ -)
  :: *smaplet* => *smaplets* (-)
  -SMaplets :: [*smaplet*, *smaplets*] => *smaplets* (-,/ -)
  -SubstUpd :: [*'m usubst*, *smaplets*] => *'m usubst* (-/'(-') [*900*,*0*] *900*)
  -Subst :: *smaplets* => *'a* ~=> *'b* ((*1*[-]))

translations
  -SubstUpd m (*-SMaplets xy ms*) == *-SubstUpd* (*-SubstUpd m xy*) *ms*
  -SubstUpd m (*-smaplet x y*) == *CONST subst-upd m x y*
  -Subst ms == *-SubstUpd* (*CONST id*) *ms*
  -Subst (*-SMaplets ms1 ms2*) <= *-SubstUpd* (*-Subst ms1*) *ms2*
  -SMaplets ms1 (*-SMaplets ms2 ms3*) <= *-SMaplets* (*-SMaplets ms1 ms2*) *ms3*

4.2 Substitution laws

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *subst-lift-upd* [*usubst*]: $\lceil\sigma(x \mapsto_s v)\rceil_s = \lceil\sigma\rceil_s(\$x \mapsto_s \lceil v\rceil_<)$
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* σ *A*)
apply (*case-tac* σ (*A*, *A*), *simp*)
done

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

syntax
-psubst :: [$'\alpha$ *usubst*, *svars*, *uexprs*] ⇒ *logic*
-subst :: ($'a$, $'\alpha$) *uexpr* ⇒ *uexprs* ⇒ *svars* ⇒ ($'a$, $'\alpha$) *uexpr* ((-⟦-'/-⟧) [*999*,*999*] *1000*)
-uexprs :: [($'a$, $'\alpha$) *uexpr*, *uexprs*] => *uexprs* (-,/ -)
:: ($'a$, $'\alpha$) *uexpr* => *uexprs* (-)
-svars :: [*svar*, *svars*] => *svars* (-,/ -)
:: *svar* => *svars* (-)

translations
-subst P es vs => *CONST subst* (*-psubst* (*CONST id*) *vs es*) *P*
-psubst m (*-svar x*) *v* => *CONST subst-upd m x v*
-psubst m (*-spvar x*) *v* => *CONST subst-upd m x v*
-psubst m (*-sinvar x*) *v* => *CONST subst-upd m* (*CONST in-var x*) *v*
-psubst m (*-soutvar x*) *v* => *CONST subst-upd m* (*CONST out-var x*) *v*
-psubst m (*-svars x xs*) (*-uexprs v vs*) => *-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* ⇒ ($'a$, $'\alpha \times '\beta$) *uexpr* ($\lceil$-$\rceil_<$)
is λ *p* (*A*, *A'*). *p A* **.**

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

lift-definition *lift-post* :: ($'a$, $'\beta$) *uexpr* ⇒ ($'a$, $'\alpha \times '\beta$) *uexpr* ($\lceil$-$\rceil_>$)
is λ *p* (*A*, *A'*). *p A'* **.**

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

where $\lfloor b \rfloor_{>} \equiv \lfloor b \rfloor_{<}$

named-theorems *ulift*

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

5.2 Lifting laws

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

lemma *lift-post-var* [*simp*]:
$\lceil var\ x \rceil_{>} = \$x'$
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*]:
$\lceil uop\ f\ v \rceil_{<} = uop\ f\ \lceil v \rceil_{<}$
by (*transfer*, *auto*)

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

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

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

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

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

end

6 Alphabetised Predicates

theory *utp-pred*
imports
utp-expr

utp-subst
begin

An alphabetised predicate is a 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** ∧ *35*) **and**
  disj (**infixr** ∨ *30*) **and**
  Not (¬ - [*40*] *40*)

consts
  utrue :: $'a$ (*true*)
  ufalse :: $'a$ (*false*)
  uconj :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** ∧ *35*)
  udisj :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** ∨ *30*)
  uimpl :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** ⇒ *25*)
  uiff :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** ⇔ *25*)
  unot :: $'a \Rightarrow 'a$ (¬ - [*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* ⇒ *logic* ⇒ *logic* (∃ - · - [*0*, *10*] *10*)
  -uall :: *svar* ⇒ *logic* ⇒ *logic* (∀ - · - [*0*, *10*] *10*)
  -ushEx :: *idt* ⇒ *logic* ⇒ *logic* (∃ - · - [*0*, *10*] *10*)
  -ushAll :: *idt* ⇒ *logic* ⇒ *logic* (∀ - · - [*0*, *10*] *10*)
  -ushBEx :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∃ - ∈ - · - [*0*, *0*, *10*] *10*)
  -ushBAll :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∀ - ∈ - · - [*0*, *0*, *10*] *10*)

translations
  ∃ &x · P => *CONST uex x P*
  ∃ \$x · P == *CONST uex* (*CONST in-var x*) *P*
  ∃ \$x´ · P == *CONST uex* (*CONST out-var x*) *P*

$\exists\ x \cdot P$ == *CONST uex x P*
$\forall\ \&x \cdot P$ => *CONST uall x P*
$\forall\ \$x \cdot P$ == *CONST uall* (*CONST in-var x*) *P*
$\forall\ \$x' \cdot P$ == *CONST uall* (*CONST out-var x*) *P*
$\forall\ x \cdot P$ == *CONST uall x P*
$\exists\ x \cdot P$ == *CONST ushEx* (λ *x*. *P*)
$\exists\ x \in A \cdot P$ => $\exists\ x \cdot «x» \in_u A \wedge P$
$\forall\ x \cdot P$ == *CONST ushAll* (λ *x*. *P*)
$\forall\ x \in A \cdot P$ => $\forall\ x \cdot «x» \in_u A \Rightarrow P$

6.2 Predicate operators

We chose to maximally reuse definitions and laws built into HOL. For this reason, when introducing the core operators we proceed by lifting operators from the polymorphic algebraic hiearchy 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* ($\bot$)

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 λ *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 $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
    by (*transfer*, *blast intro*:*order.trans*)
  show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
    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* ⇒ ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr*
  is $\lambda P\ Q\ A.\ sup\ (P\ A)\ (Q\ A)$ **.**
  lift-definition *inf-uexpr* :: ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr*
  is $\lambda P\ Q\ A.\ inf\ (P\ A)\ (Q\ A)$ **.**
instance
  by (*intro-classes*) (*transfer*, *auto*)+
end

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

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

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

We also define the other predicate operators

lift-definition *impl*::'α *upred* ⇒ 'α *upred* ⇒ 'α *upred* **is**
$\lambda\ P\ Q\ A.\ P\ A \longrightarrow Q\ A$ **.**

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

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

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

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

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

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

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

6.3 Proof support

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

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

declare *true-upred-def* [*upred-defs*]
declare *false-upred-def* [*upred-defs*]
declare *conj-upred-def* [*upred-defs*]
declare *disj-upred-def* [*upred-defs*]
declare *not-upred-def* [*upred-defs*]
declare *diff-upred-def* [*upred-defs*]
declare *subst-upd-uvar-def* [*upred-defs*]
declare *subst-upd-dvar-def* [*upred-defs*]
declare *uexpr-defs* [*upred-defs*]
declare *usubst-rel-lift-def* [*upred-defs*]
declare *usubst-rel-drop-def* [*upred-defs*]

lemma *true-alt-def*: $true = \ll True \gg$

by (*pred-tac*)

lemma *false-alt-def*: *false* = ≪*False*≫
by (*pred-tac*)

6.4 Unrestriction Laws

lemma *unrest-true* [*unrest*]: $x \sharp$ *true*
by (*pred-tac*)

lemma *unrest-false* [*unrest*]: $x \sharp$ *false*
by (*pred-tac*)

lemma *unrest-conj* [*unrest*]: $[\![\ x \sharp P;\ x \sharp Q\ ]\!] \Longrightarrow x \sharp P \wedge Q$
by (*pred-tac*)

lemma *unrest-disj* [*unrest*]: $[\![\ x \sharp P;\ x \sharp Q\ ]\!] \Longrightarrow x \sharp P \vee Q$
by (*pred-tac*)

lemma *unrest-impl* [*unrest*]: $[\![\ x \sharp P;\ x \sharp Q\ ]\!] \Longrightarrow x \sharp P \Rightarrow Q$
by (*pred-tac*)

lemma *unrest-iff* [*unrest*]: $[\![\ x \sharp P;\ x \sharp Q\ ]\!] \Longrightarrow x \sharp P \Leftrightarrow Q$
by (*pred-tac*)

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

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

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

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

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

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

lemma *unrest-shAll* [*unrest*]:
assumes $\bigwedge\ y.\ x \sharp P(y)$
shows $x \sharp (\forall\ y \cdot P(y))$

using *assms* **by** *pred-tac*

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

6.5 Substitution Laws

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

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

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

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

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

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

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

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

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

lemma *subst-shAll* [*usubst*]: $\sigma \dagger (\forall\ x \cdot P(x)) = (\forall\ x \cdot \sigma \dagger P(x))$
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* $\leq$ *op* $<$ *disj-upred false-upred true-upred*
by (*unfold-locales*, *pred-tac+*)

lemma *refBy-order*: $P \sqsubseteq Q = \text{‘}Q \Rightarrow P\text{‘}$
by (*transfer*, *auto*)

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

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

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

by *pred-tac*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *true-not-false* [*simp*]: $true \neq false\ false \neq true$
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*]: $(P \Leftrightarrow true) = P$
by *pred-tac*

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

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

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

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

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

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

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

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

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

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

lemma *subst-bool-split*:
  assumes *uvar x*
  shows $`P` = `(P[\![\mathit{false}/x]\!] \wedge P[\![\mathit{true}/x]\!])`$
proof –
  from *assms* **have** $`P` = (\forall\ v.\ `P[\![\ll v\gg/x]\!]`)$
    by (*subst taut-split-subst*[*of x*], *auto*)
  also have ... $= (`P[\![\ll \mathit{True}\gg/x]\!]` \wedge `P[\![\ll \mathit{False}\gg/x]\!]`)$
    by (*metis* (*mono-tags*, *lifting*))
  also have ... $= `(P[\![\mathit{false}/x]\!] \wedge P[\![\mathit{true}/x]\!])`$
    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 $x :: ('a, '\alpha)\ \mathit{uvar}$
  shows $(p[\![u/x]\!] \wedge u =_u v) = (p[\![v/x]\!] \wedge u =_u v)$
  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 $in\alpha$ :: $('\alpha,\ '\alpha \times '\beta)$ *uvar* **where**
$in\alpha = (\!|\ var\text{-}lookup = fst,\ var\text{-}update = \lambda\ f\ (A,\ A').\ (f\ A,\ A')\ |\!)$

definition $out\alpha$ :: $('\beta,\ '\alpha \times '\beta)$ *uvar* **where**
$out\alpha = (\!|\ var\text{-}lookup = snd,\ var\text{-}update = \lambda\ f\ (A,\ A').\ (A,\ f\ A')\ |\!)$

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

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

definition *cond*::$('\alpha,\ '\beta)$ *relation* $\Rightarrow ('\alpha,\ '\beta)$ *relation* $\Rightarrow ('\alpha,\ '\beta)$ *relation* $\Rightarrow ('\alpha,\ '\beta)$ *relation*
$((3\text{-} \triangleleft \text{-} \triangleright / \text{-})\ [14,0,15]\ 14)$
where $(P \triangleleft b \triangleright Q) \equiv (b \wedge P) \vee ((\neg b) \wedge Q)$

abbreviation *rcond*::$('\alpha,\ '\beta)$ *relation* $\Rightarrow '\alpha$ *condition* $\Rightarrow ('\alpha,\ '\beta)$ *relation* $\Rightarrow ('\alpha,\ '\beta)$ *relation*
$((3\text{-} \triangleleft \text{-} \triangleright_r / \text{-})\ [14,0,15]\ 14)$
where $(P \triangleleft b \triangleright_r Q) \equiv (P \triangleleft \lceil b \rceil_< \triangleright Q)$

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

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

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

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

abbreviation *assign-r* :: $('t,\ '\alpha)$ *uvar* $\Rightarrow ('t,\ '\alpha)$ *uexpr* $\Rightarrow '\alpha$ *hrelation*
where *assign-r x v* $\equiv$ *assigns-r* $[x \mapsto_s v]$

abbreviation *assign-2-r* ::
  $('t1,\ '\alpha)$ *uvar* $\Rightarrow ('t2,\ '\alpha)$ *uvar* $\Rightarrow ('t1,\ '\alpha)$ *uexpr* $\Rightarrow ('t2,\ '\alpha)$ *uexpr* $\Rightarrow '\alpha$ *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* ⇒ *id-list* (-)
-id-list :: *id* ⇒ *id-list* ⇒ *id-list* (-,/ -)
-uexpr-unit :: ('*a*, '*α*) *uexpr* ⇒ *uexpr-list* (- [*40*] *40*)
-uexpr-list :: ('*a*, '*α*) *uexpr* ⇒ *uexpr-list* ⇒ *uexpr-list* (-,/ - [*40*,*40*] *40*)
-assignment :: *svars* ⇒ *uexprs* ⇒ '*α hrelation* (**infixr** := *35*)
-mk-usubst :: *svars* ⇒ *uexpr-list* ⇒ '*α usubst*

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

adhoc-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* :: '*α condition* ⇒ '*α hrelation* ($\lceil$-$\rceil_t$)
where $\lceil b \rceil_t = (\lceil b \rceil_< \wedge II)$

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

7.1 Unrestriction Laws

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

lemma *unrest-ouvar* [*unrest*]: *uvar x* ⟹ *inα* ♯ $x´
by (*simp add*: *inα-def ouvar-def*, *transfer*, *auto*)

lemma *unrest-inα-var* [*unrest*]:
⟦ *uvar x*; *inα* ♯ *P* ⟧ ⟹ $x ♯ *P*
by (*pred-tac*, *simp add*: *inα-def*)

lemma *unrest-outα-var* [*unrest*]:
⟦ *uvar x*; *outα* ♯ *P* ⟧ ⟹ $x´ ♯ *P*
by (*pred-tac*, *simp add*: *outα-def*)

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

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

lemma *unrest-pre-outα* [*unrest*]: *outα* ♯ $\lceil b \rceil_<$

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

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

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

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

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

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

7.2 Substitution laws

It should be possible to substantially generalise the following two laws

lemma *usubst-seq-left* [*usubst*]:
$[\![uvar\ x;\ out\alpha \sharp v]\!] \Longrightarrow (P ;; Q)[\![v/\$x]\!] = ((P[\![v/\$x]\!]) ;; Q)$
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*]:
$[\![uvar\ x;\ in\alpha \sharp v]\!] \Longrightarrow (P ;; Q)[\![v/\$x']\!] = (P ;; Q[\![v/\$x']\!])$
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 Lifting laws

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

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

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

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

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

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

7.4 Relation laws

Homogeneous relations form a quantale

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

abbreviation *falser* :: $'\alpha$ *hrelation* ($false_h$) **where**
falser $\equiv$ *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*]: $[\![$ *out*α $\sharp$ p $]\!] \Longrightarrow \lceil\lfloor p\rfloor_<\rceil_< = p$
apply (*pred-tac*, *auto simp add*: *out*α*-def*)
apply (*rename-tac* p a b)
apply (*drule-tac* x=a **in** *spec*)
apply (*drule-tac* x=b **in** *spec*)
apply (*drule-tac* x=λ -. a **in** *spec*)

apply (*simp*)
done

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *seqr-left-unit* [*simp*]:
$(II$;; $P) = P$
by *rel-tac*

lemma *seqr-right-unit* [*simp*]:
$(P$;; $II) = P$
by *rel-tac*

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

$(false\ ;;\ P) = false$
by *pred-tac*

lemma *seqr-right-zero* [*simp*]:
$(P\ ;;\ false) = false$
by *pred-tac*

lemma *pre-skip-post*: $(\lceil b \rceil_< \wedge II) = (II \wedge \lceil b \rceil_>)$
by (*rel-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*]:
$[\![\ uvar\ x;\ uvar\ y\ ]\!] \Longrightarrow [\$x \mapsto_s \lceil u \rceil_<] \dagger (y := v) = (x,\ y := u,\ [x \mapsto_s u] \dagger v)$
by *rel-tac*

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

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

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

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

lemma *assign-test*: $uvar\ x \Longrightarrow (x := \ll u \gg\ ;;\ x := \ll v \gg) = (x := \ll v \gg)$
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 $(P\ ;;\ Q) = (\exists\ v \cdot P[\![\ll v \gg/\$x']\!]\ ;;\ Q[\![\ll v \gg/\$x]\!])$
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*:
$P = (P\ ;;\ true) \longleftrightarrow (out\alpha \sharp P)$
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* $x{=}\lambda$ *-.y* **in** *spec*)
apply (*simp*)
done

theorem *postcond-equiv*:
$P = (true\ ;;\ P) \longleftrightarrow (in\alpha \sharp P)$
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* $x{=}\lambda$ *-.y* **in** *spec*)
apply (*simp*)
done

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

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

theorem *precond-left-zero*:
assumes $out\alpha \sharp p$ $p \neq false$
shows $(true\ ;;\ p) = true$
using *assms*
apply (*simp add*: $out\alpha$*-def upred-defs*)
apply (*transfer*, *auto simp add*: *relcomp-unfold*, *rule ext*, *auto*)
apply (*rename-tac p b*)
apply (*subgoal-tac* $\exists$ *b1 b2. p* (*b1*, *b2*))
apply (*auto*)
apply (*rule-tac x=b1* **in** *exI*)
apply (*drule-tac x=b1* **in** *spec*)
apply (*drule-tac x=b2* **in** *spec*)
apply (*drule-tac* $x{=}\lambda$ *-. b* **in** *spec*)
apply (*simp*)
done

7.5 Converse laws

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

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

lemma *uivar-convr* [*simp*]:
fixes $x :: ('a, '\alpha)$ *uvar*
shows $(\$x)^{-} = \$x´$

by *pred-tac*

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

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

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

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

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

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

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

theorem *seqr-pre-transfer*: $in\alpha \sharp q \Longrightarrow ((P \wedge q)\ ;;\ R) = (P\ ;;\ (q^{-} \wedge R))$
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=λ-.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=λ-.b* **in** *spec*, *simp*)
done

theorem *seqr-post-out*: $in\alpha \sharp r \Longrightarrow (P\ ;;\ (Q \wedge r)) = ((P\ ;;\ Q) \wedge r)$
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=λ-.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=λ-.y* **in** *spec*, *simp*)
done

theorem *seqr-post-transfer*: $out\alpha \sharp q \Longrightarrow (P\ ;;\ (q \wedge R)) = (P \wedge q^{-}\ ;;\ R)$
by (*simp add*: *seqr-pre-transfer unrest-convr-inα*)

lemma *seqr-pre-out*: $out\alpha \sharp p \Longrightarrow ((p \wedge Q)\ ;;\ R) = (p \wedge (Q\ ;;\ R))$
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=λ-.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*:
$(P = (\neg (\neg P ;; \mathit{true}))) = (P = (P ;; \mathit{true}))$
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*]:
$((\exists\ x \bullet P(x)) ;; (\exists\ y \bullet Q(y))) = (\exists\ x \bullet \exists\ y \bullet P(x) ;; Q(y))$
by *pred-tac*

While loop laws

lemma *while-cond-true*:
$((\mathit{while}\ b\ \mathit{do}\ P\ \mathit{od}) \wedge \lceil b \rceil_<) = ((P \wedge \lceil b \rceil_<) ;; \mathit{while}\ b\ \mathit{do}\ P\ \mathit{od})$
proof −
have $(\mathit{while}\ b\ \mathit{do}\ P\ \mathit{od} \wedge \lceil b \rceil_<) = (((\lceil b \rceil_< \wedge P)^{\star}{}_u \wedge (\neg \lceil b \rceil_>)) \wedge \lceil b \rceil_<)$
by (*simp add*: *while-def*)
also have ... $= (((II \vee ((\lceil b \rceil_< \wedge P) ;; (\lceil b \rceil_< \wedge P)^{\star}{}_u)) \wedge \neg \lceil b \rceil_>) \wedge \lceil b \rceil_<)$
by (*simp add*: *disj-upred-def*)
also have ... $= ((\lceil b \rceil_< \wedge (II \vee ((\lceil b \rceil_< \wedge P) ;; (\lceil b \rceil_< \wedge P)^{\star}{}_u))) \wedge (\neg \lceil b \rceil_>))$
by (*simp add*: *conj-comm utp-pred.inf.left-commute*)
also have ... $= (((\lceil b \rceil_< \wedge II) \vee (\lceil b \rceil_< \wedge ((\lceil b \rceil_< \wedge P) ;; (\lceil b \rceil_< \wedge P)^{\star}{}_u))) \wedge (\neg \lceil b \rceil_>))$
by (*simp add*: *conj-disj-distr*)
also have ... $= ((((\lceil b \rceil_< \wedge II) \vee ((\lceil b \rceil_< \wedge P) ;; (\lceil b \rceil_< \wedge P)^{\star}{}_u))) \wedge (\neg \lceil b \rceil_>))$
by (*subst seqr-pre-out*[*THEN sym*], *simp add*: *unrest*, *rel-tac*)
also have ... $= ((((II \wedge \lceil b \rceil_>) \vee ((\lceil b \rceil_< \wedge P) ;; (\lceil b \rceil_< \wedge P)^{\star}{}_u))) \wedge (\neg \lceil b \rceil_>))$
by (*simp add*: *pre-skip-post*)
also have ... $= ((II \wedge \lceil b \rceil_> \wedge \neg \lceil b \rceil_>) \vee (((\lceil b \rceil_< \wedge P) ;; ((\lceil b \rceil_< \wedge P)^{\star}{}_u)) \wedge (\neg \lceil b \rceil_>)))$
by (*simp add*: *utp-pred.inf.assoc utp-pred.inf-sup-distrib2*)
also have ... $= (((\lceil b \rceil_< \wedge P) ;; ((\lceil b \rceil_< \wedge P)^{\star}{}_u)) \wedge (\neg \lceil b \rceil_>))$
by *simp*
also have ... $= ((\lceil b \rceil_< \wedge P) ;; (((\lceil b \rceil_< \wedge P)^{\star}{}_u) \wedge (\neg \lceil b \rceil_>)))$
by (*simp add*: *seqr-post-out unrest*)
also have ... $= ((P \wedge \lceil b \rceil_<) ;; \mathit{while}\ b\ \mathit{do}\ P\ \mathit{od})$
by (*simp add*: *utp-pred.inf-commute while-def*)
finally show *?thesis* **.**
qed

lemma *while-cond-false*:
$((\mathit{while}\ b\ \mathit{do}\ P\ \mathit{od}) \wedge (\neg \lceil b \rceil_<)) = (II \wedge \neg \lceil b \rceil_<)$
proof −
have $(\mathit{while}\ b\ \mathit{do}\ P\ \mathit{od} \wedge (\neg \lceil b \rceil_<)) = (((\lceil b \rceil_< \wedge P)^{\star}{}_u \wedge (\neg \lceil b \rceil_>)) \wedge (\neg \lceil b \rceil_<))$
by (*simp add*: *while-def*)
also have ... $= (((II \vee ((\lceil b \rceil_< \wedge P) ;; (\lceil b \rceil_< \wedge P)^{\star}{}_u)) \wedge \neg \lceil b \rceil_>) \wedge (\neg \lceil b \rceil_<))$
by (*simp add*: *disj-upred-def*)
also have ... $= (((II \wedge \neg \lceil b \rceil_>) \wedge \neg \lceil b \rceil_<) \vee ((\neg \lceil b \rceil_<) \wedge (((\lceil b \rceil_< \wedge P) ;; ((\lceil b \rceil_< \wedge P)^{\star}{}_u)) \wedge \neg \lceil b \rceil_>)))$
by (*simp add*: *conj-disj-distr utp-pred.inf.commute*)
also have ... $= (((II \wedge \neg \lceil b \rceil_>) \wedge \neg \lceil b \rceil_<) \vee ((((\neg \lceil b \rceil_<) \wedge (\lceil b \rceil_< \wedge P) ;; ((\lceil b \rceil_< \wedge P)^{\star}{}_u)) \wedge \neg \lceil b \rceil_>)))$
by (*simp add*: *seqr-pre-out unrest-not unrest-pre-out*α *utp-pred.inf.assoc*)
also have ... $= (((II \wedge \neg \lceil b \rceil_>) \wedge \neg \lceil b \rceil_<) \vee (((\mathit{false} ;; ((\lceil b \rceil_< \wedge P)^{\star}{}_u)) \wedge \neg \lceil b \rceil_>)))$

by (*simp add*: *conj-comm utp-pred.inf.left-commute*)
also have ... = $((II \wedge \neg \lceil b \rceil_{>}) \wedge \neg \lceil b \rceil_{<})$
by *simp*
also have ... = $(II \wedge \neg \lceil b \rceil_{<})$
by *rel-tac*
finally show *?thesis* **.**
qed

theorem *while-unfold*:
while b do P od = $((P\ ;;\ \textit{while b do P od}) \triangleleft b \triangleright_r II)$
by (*metis* (*no-types*, *hide-lams*) *bounded-semilattice-sup-bot-class.sup-bot.left-neutral comp-cond-left-distr cond-def cond-idem disj-comm disj-upred-def seqr-right-zero upred-quantale.bot-zerol utp-pred.inf-bot-right utp-pred.inf-commute while-cond-false while-cond-true*)

end

7.6 Weakest precondition calculus

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

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

named-theorems *wp*

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

consts
uwp :: $'a \Rightarrow 'b \Rightarrow 'c$ (**infix** *wp 60*)

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

adhoc-overloading
uwp wp-upred

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

theorem *wp-assigns-r* [*wp*]:
(*assigns-r* σ) *wp r* = $\sigma \dagger r$
by *rel-tac*

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

theorem *wp-true* [*wp*]:
$r \neq$ *true* $\Longrightarrow$ *true wp r* = *false*
by *rel-tac*

theorem *wp-conj* [*wp*]:
P wp $(q \wedge r)$ = (*P wp q* $\wedge$ *P wp r*)
by *rel-tac*

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

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

end

8 UTP Theories

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

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

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

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

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

end

9 Example UTP theory: Boyle's laws

theory *utp-boyle*
imports *utp-theory*
begin

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

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

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

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

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

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

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

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

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

lemma *B-idempotent*:
  $B(B(P)) = B(P)$
  by *pred-tac*

lemma *B-monotone*:
  $X \sqsubseteq Y \Longrightarrow B(X) \sqsubseteq B(Y)$
  by *pred-tac*

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

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

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

We prove that φ_1 satisfied by Boyle's law by simplication of its definitional equation and then application of the predicate tactic.

lemma *B-*φ_1: φ_1 *is B*
  by (*simp add*: φ_1*-def*, *pred-tac*)

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

lemma *B-*φ_2: $B(\varphi_2) = \varphi_1$
proof –
  have $B(\varphi_2) = B((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 100))$
    by (*simp add*: φ_2*-def*)
  also have ... $= ((\exists\ k \cdot (\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 100)) \wedge (\&k =_u \&p * \&V))$
    by *pred-tac*
  also have ... $= ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u \&p * \&V))$
    by *pred-tac*
  also have ... $= ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 50))$
    by *pred-tac*
  also have ... $= \varphi_1$
    by (*simp add*: φ_1*-def*)
  finally show *?thesis* **.**
qed

end

10 Designs

theory *utp-designs*
imports
  utp-rel
  utp-wp
  utp-theory
begin

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

10.1 Definitions

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

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

The ok variable is defined using the syntactic translation VAR

definition *ok* = *VAR des-ok*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

abbreviation σf::$('\alpha, '\beta)$ *relation-d* $\Rightarrow$ $('\alpha, '\beta)$ *relation-d* (-^f *[1000] 1000*)
where σf $D \equiv D[\![false/\$ok']\!]$

abbreviation σt::$('\alpha, '\beta)$ *relation-d* $\Rightarrow$ $('\alpha, '\beta)$ *relation-d* (-^t *[1000] 1000*)
where σt $D \equiv D[\![true/\$ok']\!]$

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

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

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

declare *design-def* [*upred-defs*]
declare *rdesign-def* [*upred-defs*]
declare *skip-d-def* [*upred-defs*]
declare *J-def* [*upred-defs*]
declare *pre-design-def* [*upred-defs*]
declare *post-design-def* [*upred-defs*]
declare *wp-design-def* [*upred-defs*]

declare *H1-def* [*upred-defs*]
declare *H2-def* [*upred-defs*]
declare *H3-def* [*upred-defs*]
declare *H4-def* [*upred-defs*]

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

lemma *lift-desr-inv*:
  $[\![\ \$ok \sharp P; \$ok' \sharp P\ ]\!] \Longrightarrow \lceil \lfloor P \rfloor_D \rceil_D = P$
  apply (*rel-tac*)
  apply (*rename-tac P a b*)
  apply (*drule-tac x=a* **in** *spec*)
  apply (*drule-tac x=b* **in** *spec*)
  apply (*drule-tac x=*λ -. *True* **in** *spec*)
  apply (*metis alpha-d.surjective alpha-d.update-convs*(*1*))
  apply (*drule-tac x=a* **in** *spec*)
  apply (*drule-tac x=b* **in** *spec*)

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

10.2 Design laws

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

lemma *unrest-out-des-lift* [*unrest*]: $out\alpha \sharp p \Longrightarrow out\alpha \sharp \lceil p \rceil_D$
apply (*pred-tac*)
apply (*auto simp add*: *out*α*-def*)
apply (*rename-tac p b v x*)
apply (*drule-tac x=alpha-d.more x* **in** *spec*)
apply (*drule-tac x=alpha-d.more b* **in** *spec*)
apply (*drule-tac* $x=\lambda$ -. *alpha-d.more* (*v b*) **in** *spec*)
apply (*simp*)
apply (*rename-tac p b v x*)
apply (*drule-tac x=alpha-d.more x* **in** *spec*)
apply (*drule-tac x=alpha-d.more b* **in** *spec*)
apply (*drule-tac* $x=\lambda$ -. *alpha-d.more* (*v b*) **in** *spec*)
apply (*simp*)
done

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

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

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

theorem *rdesign-refinement*:
$(P1 \vdash_r Q1 \sqsubseteq P2 \vdash_r Q2) \longleftrightarrow (`P1 \Rightarrow P2` \wedge `P1 \wedge Q2 \Rightarrow Q1`)$
apply (*simp add*: *rdesign-def*)
apply (*subst design-refinement*)

apply (*simp-all add*: *unrest*)
apply (*pred-tac*)
apply (*metis alpha-d.select-convs*(*2*))+
done

lemma *design-refine-intro*:
assumes ‘$P1 \Rightarrow P2$‘ ‘$P1 \wedge Q2 \Rightarrow Q1$‘
shows $P1 \vdash Q1 \sqsubseteq P2 \vdash Q2$
using *assms* **unfolding** *upred-defs*
by *pred-tac*

theorem *design-ok-false* [*usubst*]: $(P \vdash Q)[\![false/\$ok]\!] = true$
by (*simp add*: *design-def usubst*)

theorem *design-pre*:
$\$ok' \sharp P \Longrightarrow \neg (P \vdash Q)^f = (\$ok \wedge P^f)$
by (*simp add*: *design-def*, *subst-tac*)
(*metis* (*no-types*, *hide-lams*) *not-conj-deMorgans true-not-false*(*2*) *utp-pred.compl-top-eq utp-pred.sup.idem utp-pred.sup-compl-top var-in-var*)

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

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

theorem *design-true-left-zero*: $(true ;; (P \vdash Q)) = true$
proof −
have $(true ;; (P \vdash Q)) = (\exists\ ok_0 \cdot true[\![\ll ok_0 \gg/\$ok']\!] ;; (P \vdash Q)[\![\ll ok_0 \gg/\$ok]\!])$
by (*subst seqr-middle*[*of ok*], *simp-all*)
also have ... $= ((true[\![false/\$ok']\!] ;; (P \vdash Q)[\![false/\$ok]\!]) \vee (true[\![true/\$ok']\!] ;; (P \vdash Q)[\![true/\$ok]\!]))$
by (*simp add*: *disj-comm false-alt-def true-alt-def*)
also have ... $= ((true[\![false/\$ok']\!] ;; true_h) \vee (true ;; ((P \vdash Q)[\![true/\$ok]\!])))$
by (*subst-tac*, *rel-tac*)
also have ... $= true$
by (*subst-tac*, *simp add*: *precond-right-unit unrest*)
finally show *?thesis* **.**
qed

theorem *design-composition*:
assumes
$\$ok \sharp P1$ $\$ok' \sharp P1$ $\$ok \sharp P2$ $\$ok' \sharp P2$
$\$ok \sharp Q1$ $\$ok' \sharp Q1$ $\$ok \sharp Q2$ $\$ok' \sharp Q2$
shows $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (((\neg ((\neg P1) ;; true)) \wedge \neg (Q1 ;; (\neg P2))) \vdash (Q1 ;; Q2))$
proof −
have $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (\exists\ ok_0 \cdot ((P1 \vdash Q1)[\![\ll ok_0 \gg/\$ok']\!] ;; (P2 \vdash Q2)[\![\ll ok_0 \gg/\$ok]\!]))$
by (*rule seqr-middle*, *simp*)
also have ...
$= (((P1 \vdash Q1)[\![false/\$ok']\!] ;; (P2 \vdash Q2)[\![false/\$ok]\!])$
$\vee ((P1 \vdash Q1)[\![true/\$ok']\!] ;; (P2 \vdash Q2)[\![true/\$ok]\!]))$
by (*simp add*: *true-alt-def false-alt-def*, *pred-tac*)
also from *assms*
have ... $= (((\$ok \wedge P1 \Rightarrow Q1) ;; (P2 \Rightarrow \$ok' \wedge Q2)) \vee ((\neg (\$ok \wedge P1)) ;; true))$
by (*simp add*: *design-def usubst unrest*, *pred-tac*)
also have ... $= ((\neg \$ok ;; true_h) \vee (\neg P1 ;; true) \vee (Q1 ;; \neg P2) \vee (\$ok' \wedge (Q1 ;; Q2)))$

by (*rel-tac*)
also have ... = $(\neg\ (\neg\ P1\ ;;\ true) \wedge \neg\ (Q1\ ;;\ \neg\ P2)) \vdash (Q1\ ;;\ Q2)$
by (*simp add*: *precond-right-unit design-def unrest*, *rel-tac*)
finally show *?thesis* **.**
qed

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

lemma *skip-d-alt-def*: $II_D = true \vdash II$
by (*rel-tac*)

theorem *design-skip-idem* [*simp*]:
$(II_D\ ;;\ II_D) = II_D$
by (*simp add*: *skip-d-def urel-defs*, *pred-tac*)

theorem *design-composition-cond*:
assumes
$\$ok \sharp p1\ out\alpha \sharp p1\ \$ok \sharp P2\ \$ok' \sharp P2$
$\$ok \sharp Q1\ \$ok' \sharp Q1\ \$ok \sharp Q2\ \$ok' \sharp Q2$
shows $((p1 \vdash Q1)\ ;;\ (P2 \vdash Q2)) = ((p1 \wedge \neg\ (Q1\ ;;\ (\neg\ P2))) \vdash (Q1\ ;;\ Q2))$
using *assms*
by (*simp add*: *design-composition unrest precond-right-unit*)

theorem *rdesign-composition-cond*:
assumes $out\alpha \sharp p1$
shows $((p1 \vdash_r Q1)\ ;;\ (P2 \vdash_r Q2)) = ((p1 \wedge \neg\ (Q1\ ;;\ (\neg\ P2))) \vdash_r (Q1\ ;;\ Q2))$
using *assms*
by (*simp add*: *rdesign-def design-composition-cond unrest*)

theorem *design-composition-wp*:
fixes *Q1 Q2* :: *'a hrelation-d*
assumes
$ok \sharp p1\ ok \sharp p2$
$\$ok \sharp Q1\ \$ok' \sharp Q1\ \$ok \sharp Q2\ \$ok' \sharp Q2$
shows $((\lceil p1 \rceil_< \vdash Q1)\ ;;\ (\lceil p2 \rceil_< \vdash Q2)) = ((\lceil p1 \wedge Q1\ wp\ p2 \rceil_<) \vdash (Q1\ ;;\ Q2))$
using *assms*
by (*simp add*: *design-composition-cond unrest*, *rel-tac*)

theorem *rdesign-composition-wp*:
fixes *Q1 Q2* :: *'a hrelation*
shows $((\lceil p1 \rceil_< \vdash_r Q1)\ ;;\ (\lceil p2 \rceil_< \vdash_r Q2)) = ((\lceil p1 \wedge Q1\ wp\ p2 \rceil_<) \vdash_r (Q1\ ;;\ Q2))$
by (*simp add*: *rdesign-composition-cond unrest*, *rel-tac*)

theorem *rdesign-wp* [*wp*]:
$(\lceil p \rceil_< \vdash_r Q)\ wp_D\ r = (p \wedge Q\ wp\ r)$
by *rel-tac*

theorem *wpd-seq-r*:
fixes *Q1 Q2* :: *'α hrelation*
shows $(\lceil p1 \rceil_< \vdash_r Q1\ ;;\ \lceil p2 \rceil_< \vdash_r Q2)\ wp_D\ r = (\lceil p1 \rceil_< \vdash_r Q1)\ wp_D\ ((\lceil p2 \rceil_< \vdash_r Q2)\ wp_D\ r)$
apply (*simp add*: *wp*)
apply (*subst rdesign-composition-wp*)

apply (*simp only*: *wp*)
apply (*rel-tac*)
done

theorem *design-left-unit* [*simp*]:
$(II_D \;;; P \vdash_r Q) = (P \vdash_r Q)$
by (*simp add*: *skip-d-def urel-defs*, *pred-tac*)

theorem *design-right-cond-unit* [*simp*]:
assumes $out\alpha \sharp p$
shows $(p \vdash_r Q \;;; II_D) = (p \vdash_r Q)$
using *assms*
by (*simp add*: *skip-d-def rdesign-composition-cond*)

lemma *lift-des-skip-dr-unit* [*simp*]:
$(\lceil P \rceil_D \;;; \lceil II \rceil_D) = \lceil P \rceil_D$
$(\lceil II \rceil_D \;;; \lceil P \rceil_D) = \lceil P \rceil_D$
by *rel-tac rel-tac*

10.3 H1: No observation is allowed before initiation

lemma *H1-idem*:
$H1\ (H1\ P) = H1(P)$
by *pred-tac*

lemma *H1-monotone*:
$P \sqsubseteq Q \Longrightarrow H1(P) \sqsubseteq H1(Q)$
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 ... $= (((\neg \$ok \;;; true_h) \;;; R) \vee R)$
by (*simp add*: *precond-right-unit unrest*)
also have ... $= ((\neg \$ok \;;; true_h) \vee R)$
by (*metis assms*(*1*) *seqr-assoc*)
also have ... $= (\$ok \Rightarrow R)$
by (*simp add*: *impl-alt-def precond-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 $(true_h \;;\; P) = true_h$
proof −
  from *assms* **have** $(true_h \;;\; P) = (true_h \;;\; (\$ok \Rightarrow P))$
    by (*simp add*: *H1-def Healthy-def'*)
  also from *assms* **have** ... $= (true_h \;;\; (\neg \$ok \vee P))$
    by (*simp add*: *impl-alt-def*)
  also from *assms* **have** ... $= ((true_h \;;\; \neg \$ok) \vee (true_h \;;\; P))$
    using *seqr-or-distr* **by** *blast*
  also from *assms* **have** ... $= (true \vee (true \;;\; P))$
    by (*simp add*: *nok-not-false precond-left-zero unrest*)
  finally show *?thesis* **by** *rel-tac*
qed

theorem *H1-left-unit*:
  fixes $P :: {'}\alpha$ *hrelation-d*
  assumes *P is H1*
  shows $(II_D \;;\; P) = P$
proof −
  have $(II_D \;;\; P) = ((\$ok \Rightarrow II) \;;\; P)$
    by (*metis H1-def H1-design-skip*)
  also have ... $= ((\neg \$ok \;;\; P) \vee P)$
    by (*simp add*: *impl-alt-def seqr-or-distl*)
  also from *assms* **have** ... $= (((\neg \$ok \;;\; true_h) \;;\; P) \vee P)$
    by (*simp add*: *precond-right-unit unrest*)
  also have ... $= ((\neg \$ok \;;\; (true_h \;;\; P)) \vee P)$
    by (*simp add*: *seqr-assoc*)
  also from *assms* **have** ... $= (\$ok \Rightarrow P)$
    by (*simp add*: *H1-left-zero impl-alt-def precond-right-unit unrest*)
  finally show *?thesis* **using** *assms*
    by (*simp add*: *H1-def Healthy-def'*)
qed

theorem *H1-algebraic*:
  P *is H1* $\longleftrightarrow (true_h \;;\; P) = true_h \wedge (II_D \;;\; P) = P$
  using *H1-algebraic-intro H1-left-unit H1-left-zero* **by** *blast*

theorem *H1-nok-left-zero*:
  fixes $P :: {'}\alpha$ *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*: *precond-right-unit unrest*)
  also have ... $= ((\neg \$ok) \;;\; true_h)$
    by (*metis H1-left-zero assms seqr-assoc*)
  also have ... $= (\neg \$ok)$
    by (*simp add*: *precond-right-unit unrest*)

finally show *?thesis* **.**
qed

10.4 H2: A specification cannot require non-termination

lemma *J-split*:
  shows $(P ;; J) = (P^f \vee (P^t \wedge \$ok'))$
proof –
  have $(P ;; J) = (P ;; ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D))$
    by (*simp add*: *H2-def J-def design-def*)
  also have ... $= (P ;; ((\$ok \Rightarrow \$ok \wedge \$ok') \wedge \lceil II \rceil_D))$
    by *rel-tac*
  also have ... $= ((P ;; (\neg \$ok \wedge \lceil II \rceil_D)) \vee (P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))))$
    by *rel-tac*
  also have ... $= (P^f \vee (P^t \wedge \$ok'))$
  proof –
    have $(P ;; (\neg \$ok \wedge \lceil II \rceil_D)) = P^f$
    proof –
      have $(P ;; (\neg \$ok \wedge \lceil II \rceil_D)) = ((P \wedge \neg \$ok') ;; \lceil II \rceil_D)$
        by *rel-tac*
      also have ... $= (\exists\ \$ok' \cdot P \wedge \$ok' =_u \mathit{false})$
        by (*rel-tac*, *metis* (*mono-tags*, *lifting*) *alpha-d.surjective alpha-d.update-convs*(*1*))
      also have ... $= P^f$
        by (*metis one-point out-var-uvar ouvar-def unrest-false uvar-ok*)
     finally show *?thesis* **.**
    qed
    moreover have $(P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))) = (P^t \wedge \$ok')$
    proof –
      have $(P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))) = (P ;; (\$ok \wedge II))$
        by (*rel-tac*, *metis alpha-d.equality*)
      also have ... $= (P^t \wedge \$ok')$
        by (*rel-tac*, *metis* (*full-types*) *alpha-d.surjective alpha-d.update-convs*(*1*))+
      finally show *?thesis* **.**
    qed
    ultimately show *?thesis*
      by *simp*
  qed
  finally show *?thesis* **.**
qed

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

theorem *H2-equivalence*:
  P *is H2* $\longleftrightarrow$ ‘$P^f \Rightarrow P^t$‘
proof –
  have ‘$P \Leftrightarrow (P ;; J)$‘ $\longleftrightarrow$ ‘$P \Leftrightarrow (P^f \vee (P^t \wedge \$ok'))$‘
    by (*simp add*: *J-split*)
  also from *assms* **have** ... $\longleftrightarrow$ ‘$(P \Leftrightarrow P^f \vee P^t \wedge \$ok')^f \wedge (P \Leftrightarrow P^f \vee P^t \wedge \$ok')^t$‘
    by (*simp add*: *subst-bool-split*)
  also from *assms* **have** ... = ‘$(P^f \Leftrightarrow P^f) \wedge (P^t \Leftrightarrow P^f \vee P^t)$‘
    by *subst-tac*
  also have ... = ‘$P^t \Leftrightarrow (P^f \vee P^t)$‘
    by *pred-tac*
  also have ... = ‘$(P^f \Rightarrow P^t)$‘

by *pred-tac*
finally show *?thesis* **using** *assms*
by (*metis H2-def Healthy-def′ taut-iff-eq*)
qed

lemma *H2-equiv*:
P *is* $H2 \longleftrightarrow P^t \sqsubseteq P^f$
using *H2-equivalence refBy-order* **by** *blast*

lemma *H2-design*:
assumes $\$ok \sharp P$ $\$ok' \sharp P$ $\$ok \sharp Q$ $\$ok' \sharp Q$
shows $H2(P \vdash Q) = P \vdash Q$
using *assms*
by (*simp add*: *H2-split design-def usubst unrest*, *pred-tac*)

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

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

theorem *H2-idem*:
$H2(H2(P)) = H2(P)$
by (*metis H2-def J-idem 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 ... $= (\neg\ \$ok \vee (\neg\ \$ok) \wedge \$ok')$
by (*subst-tac*, *simp add*: *iuvar-def*)
also have ... $= (\neg\ \$ok)$
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** ... $= ((\neg\ \$ok \vee P) \;;;\; J)$
by *rel-tac*
also have ... $= ((\neg\ \$ok \;;;\; J) \vee (P \;;;\; J))$
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-rdesign*:
  assumes P *is H1* P *is H2*
  shows $P = pre_D(P) \vdash_r post_D(P)$
proof –
  from *assms* **have** $P = (\$ok \Rightarrow H2(P))$
    by (*simp add*: *H1-def Healthy-def'*)
  also have ... $= (\$ok \Rightarrow (P^f \vee (P^t \wedge \$ok')))$
    by (*metis H2-split*)
  also have ... $= (\$ok \wedge (\neg P^f) \Rightarrow \$ok' \wedge P^t)$
    by *pred-tac*
  also have ... $= (\$ok \wedge (\neg P^f) \Rightarrow \$ok' \wedge \$ok \wedge P^t)$
    by *pred-tac*
  also have ... $= (\$ok \wedge \lceil pre_D(P) \rceil_D \Rightarrow \$ok' \wedge \$ok \wedge \lceil post_D(P) \rceil_D)$
    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** ... $= pre_D(P) \vdash_r post_D(P)$
    by (*simp add*: *rdesign-def design-def*)
  finally show *?thesis* **.**
qed

abbreviation *H1-H2* $P \equiv H1\ (H2\ P)$

10.5 H3: The design assumption is a precondition

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

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

theorem *design-H3-iff-pre*:
assumes $\$ok \sharp P$ $\$ok' \sharp P$ $\$ok \sharp Q$ $\$ok' \sharp Q$
shows $P \vdash Q$ *is H3* $\longleftrightarrow P = (P \;;;\; true)$
proof –
have $P \vdash Q = \lfloor P \rfloor_D \vdash_r \lfloor Q \rfloor_D$
by (*simp add*: *assms lift-desr-inv rdesign-def*)
moreover hence $\lfloor P \rfloor_D \vdash_r \lfloor Q \rfloor_D$ *is H3* $\longleftrightarrow \lfloor P \rfloor_D = (\lfloor P \rfloor_D \;;;\; true)$
using *rdesign-H3-iff-pre* **by** *blast*
ultimately show *?thesis*
by (*metis assms drop-desr-inv lift-desr-inv lift-dist-seq lift-dists*(*1*))
qed

theorem *H1-H3-commute*:
$H1\ (H3\ P) = H3\ (H1\ P)$
by *rel-tac*

lemma *skip-d-absorb-J-1*:
$(II_D \;;;\; J) = II_D$
by (*metis H2-def H2-rdesign skip-d-def*)

lemma *skip-d-absorb-J-2*:
$(J \;;;\; II_D) = II_D$
proof –
have $(J \;;;\; II_D) = ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D \;;;\; true \vdash II)$
by (*simp add*: *J-def skip-d-alt-def*)
also have ... $= (\exists\ ok_0 \cdot ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D)[\![\ll ok_0 \gg/\$ok']\!] \;;;\; (true \vdash II)[\![\ll ok_0 \gg/\$ok]\!])$
by (*subst seqr-middle*[*of ok*], *simp-all*)
also have ... $= ((((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D)[\![false/\$ok']\!] \;;;\; (true \vdash II)[\![false/\$ok]\!])$
$\vee (((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D)[\![true/\$ok']\!] \;;;\; (true \vdash II)[\![true/\$ok]\!]))$
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*, *rel-tac*)
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 seqr-assoc skip-d-absorb-J-1*)

lemma *H3-H2-absorb*:
$H3\ (H2\ P) = H3\ P$
by (*metis H2-def H3-def seqr-assoc skip-d-absorb-J-2*)

theorem *H2-H3-commute*:
$H2\ (H3\ P) = H3\ (H2\ P)$
by (*simp add*: *H2-H3-absorb H3-H2-absorb*)

theorem *H3-design-pre*:
assumes $\$ok \sharp p$ $out\alpha \sharp p$ $\$ok \sharp Q$ $\$ok' \sharp Q$
shows $H3(p \vdash Q) = p \vdash Q$
using *assms*
by (*metis Healthy-def' design-H3-iff-pre precond-right-unit unrest-outα-var uvar-ok*)

theorem *H3-rdesign-pre*:
  assumes $out\alpha \sharp p$
  shows $H3(p \vdash_r Q) = p \vdash_r Q$
  using *assms*
  by (*simp add*: *H3-def*)

theorem *H1-H3-is-rdesign*:
  assumes *P is H1 P is H3*
  shows $P = pre_D(P) \vdash_r post_D(P)$
  by (*metis H1-H2-is-rdesign H2-H3-absorb Healthy-def′ assms*)

theorem *H1-H3-is-normal-design*:
  assumes *P is H1 P is H3*
  shows $P = \lfloor pre_D(P) \rfloor_< \vdash_n post_D(P)$
  by (*metis H1-H3-is-rdesign assms drop-pre-inv ndesign-def precond-equiv rdesign-H3-iff-pre*)

abbreviation $H1\text{-}H3\ p \equiv H1\ (H3\ p)$

theorem *wpd-seq-r-H1-H2* [*wp*]:
  fixes $P\ Q :: {'}\alpha\ hrelation\text{-}d$
  assumes *P is H1-H3 Q is H1-H3*
  shows $(P ;; Q)\ wp_D\ r = P\ wp_D\ (Q\ wp_D\ r)$
  by (*smt H1-H3-commute H1-H3-is-rdesign H1-idem Healthy-def′ assms*(*1*) *assms*(*2*) *drop-pre-inv precond-equiv rdesign-H3-iff-pre wpd-seq-r*)

10.6 H4: Feasibility

theorem *H4-idem*:
  $H4(H4(P)) = H4(P)$
  by *pred-tac*

end

11 Concurrent programming

theory *utp-concurrency*
  imports *utp-designs*
begin

no-notation
  Sublist.parallel (**infixl** $\|$ *50*)

11.1 Design parallel composition

definition $design\text{-}par :: ({'}\alpha, {'}\beta)\ relation\text{-}d \Rightarrow ({'}\alpha, {'}\beta)\ relation\text{-}d \Rightarrow ({'}\alpha, {'}\beta)\ relation\text{-}d$ (**infixr** $\|$ *85*)
where
$P \| Q = ((pre_D(P) \wedge pre_D(Q)) \vdash_r (post_D(P) \wedge post_D(Q)))$

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

lemma *parallel-zero*: $P \| true = true$
proof −
  have $P \| true = (pre_D(P) \wedge pre_D(true)) \vdash_r (post_D(P) \wedge post_D(true))$
    by (*simp add*: *design-par-def*)
  also have ... $= (pre_D(P) \wedge false) \vdash_r (post_D(P) \wedge true)$

by *rel-tac*
also have ... = *true*
by *rel-tac*
finally show *?thesis* **.**
qed

lemma *parallel-assoc*: $P \parallel Q \parallel R = (P \parallel Q) \parallel R$
by *rel-tac*

lemma *parallel-comm*: $P \parallel Q = Q \parallel P$
by *pred-tac*

lemma *parallel-idem*:
assumes *P is H1 P is H2*
shows $P \parallel P = P$
by (*metis H1-H2-is-rdesign assms conj-idem design-par-def*)

lemma *parallel-mono-1*:
assumes $P_1 \sqsubseteq P_2$ P_1 *is H1-H2* P_2 *is H1-H2*
shows $P_1 \parallel Q \sqsubseteq P_2 \parallel Q$
proof −
have $pre_D(P_1) \vdash_r post_D(P_1) \sqsubseteq pre_D(P_2) \vdash_r post_D(P_2)$
by (*metis H1-H2-commute H1-H2-is-rdesign H1-idem Healthy-def' assms*)
hence $(pre_D(P_1) \vdash_r post_D(P_1)) \parallel Q \sqsubseteq (pre_D(P_2) \vdash_r post_D(P_2)) \parallel Q$
by (*auto simp add*: *rdesign-refinement design-par-def*) (*pred-tac*+)
thus *?thesis*
by (*metis H1-H2-commute H1-H2-is-rdesign H1-idem Healthy-def' assms*)
qed

lemma *parallel-mono-2*:
assumes $Q_1 \sqsubseteq Q_2$ Q_1 *is H1-H2* Q_2 *is H1-H2*
shows $P \parallel Q_1 \sqsubseteq P \parallel Q_2$
by (*metis assms parallel-comm parallel-mono-1*)

11.2 Parallel by merge

We describe the partition of a state space into a n pieces through the use of a list.

type-synonym *'a partition* = *'a list*

A merge relation is a design that describes how a partitioned state-space should be merged into a third state-space. For now the state-spaces for two merged processes should have the same type. This could potentially be generalised, but that might have an effect on our reasoning capabilities.

definition *ind-uvar* :: *nat* ⇒ (*'a*, *'α alphabet-d*) *uvar* ⇒ (*'a*, (*'α* × *'α partition*) *alphabet-d*) *uvar* **where**
ind-uvar i x = (| *var-lookup* = *var-lookup x* ∘ (λ *A*. (| *des-ok* = *des-ok A*, ... = *snd* (*more A*) ! *i* |))
, *var-update* = *undefined*
|)

definition *pre-uvar* :: (*'a*, *'α alphabet-d*) *uvar* ⇒ (*'a*, (*'α* × *'α partition*) *alphabet-d*) *uvar* **where**
pre-uvar x = (| *var-lookup* = *var-lookup x* ∘ (λ *A*. (| *des-ok* = *des-ok A*, ... = *fst* (*more A*) |))
, *var-update* = *undefined* |)

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

syntax
  -uprevar :: ('t, 'α) *uvar* ⇒ *logic* ($\$_{<}$- [999] 999)
  -udotvar :: *nat* ⇒ ('t, 'α) *uvar* ⇒ *logic* (&-.- [0,999] 999)
  -uidotvar :: *nat* ⇒ ('t, 'α) *uvar* ⇒ *logic* ($-.- [0,999] 999)
  -uodotvar :: *nat* ⇒ ('t, 'α) *uvar* ⇒ *logic* ($-.-´ [999] 999)
  -sdotvar :: *nat* ⇒ *id* ⇒ *svar* (&-.- [0,999] 999)
  -sin-dotvar :: *nat* ⇒ *id* ⇒ *svar* ($-.-)
  -sout-dotvar :: *nat* ⇒ *id* ⇒ *svar* ($-.-´)

translations
  -uprevar x == *CONST var* (*CONST in-var* (*CONST pre-uvar x*))
  -udotvar n x == *CONST var* (*CONST ind-uvar n x*)
  -uidotvar n x == *CONST var* (*CONST in-var* (*CONST ind-uvar n x*))
  -uidotvar n x == *CONST var* (*CONST out-var* (*CONST ind-uvar n x*))
  -sdotvar n x == *CONST ind-uvar n x*
  -sin-dotvar n x == *CONST in-var* (*CONST ind-uvar n x*)
  -sout-dotvar n x == *CONST out-var* (*CONST ind-uvar n x*)
  -psubst m (*-sdotvar n x*) *v* => *CONST subst-upd m* (*CONST ind-uvar n x*) *v*

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

Separating simulations

lift-definition *sep-sim* :: *nat* ⇒ ('α, 'α *partition*) *relation-d* (*U*'(-')) **is**
λ *n* (*A*, *A*'). *des-ok A*' = *des-ok A* ∧ *length* (*alpha-d.more A*') > *n* ∧ *alpha-d.more A*' ! *n* = *alpha-d.more A* .

lift-definition *alpha-ext* :: ('α, 'β) *relation-d* ⇒ ('α, 'α × 'β) *relation-d* ($-_{+}$ [999] 999) **is**
λ *P* (*A*, *A*'). *P* (*A*, (| *des-ok* = *des-ok A*', ... = *snd* (*more A*') |)) ∧ *des-ok A*' = *des-ok A* ∧ *fst* (*more A*') = *more A* .

Parallel by merge

definition *design-par-by-merge* ::
  'α *hrelation-d* ⇒ 'α *merge* ⇒ 'α *hrelation-d* ⇒ 'α *hrelation-d* (**infixr** $\|_{-}$ 85)
where $P \|_{M} Q = (((P ;; U(0)) \| (Q ;; U(1)))_{+} ;; M)$

end

12 Reactive processes

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

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 ⇒ 'α list ⇒ '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 ⟹ m − l = t*
by *(auto)*

lemma *prefix-subst1 [simp]: m = l @ t ⟹ 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 *'ϑ refusal = 'ϑ set*

record *'ϑ alpha-rp = alpha-d +*
rp-wait :: bool
rp-tr :: 'ϑ trace
rp-ref :: 'ϑ 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]*

instantiation *alpha-rp-ext :: (type, vst) vst*
begin
definition *get-vstore-alpha-rp-ext :: ('a, 'b) alpha-rp-ext ⇒ vstore*
where *[simp]: get-vstore-alpha-rp-ext x = get-vstore (alpha-rp.more (alpha-d.extend undefined x))*
definition *upd-vstore-alpha-rp-ext :: (vstore ⇒ vstore) ⇒ ('a, 'b) alpha-rp-ext ⇒ ('a, 'b) alpha-rp-ext*
where *[simp]: upd-vstore-alpha-rp-ext f x = alpha-d.more (alpha-rp.more-update (upd-vstore f) (alpha-d.extend undefined x))*
instance
apply *(intro-classes, auto simp add: upd-store-parm[THEN sym] alpha-rp.defs alpha-d.defs)*
apply *(metis (no-types, lifting) alpha-d.ext-inject alpha-d.surjective alpha-rp.select-convs(4) alpha-rp.surjective alpha-rp.update-convs(4) get-upd-vstore)*

apply (*smt alpha-d.select-convs*(*2*) *alpha-rp.surjective alpha-rp.update-convs*(*4*) *upd-vstore-comp*)
apply (*metis alpha-d.select-convs*(*2*) *alpha-rp.surjective alpha-rp.update-convs*(*4*) *upd-vstore-eta*)
apply (*metis alpha-rp.unfold-congs*(*5*) *upd-store-parm*)
done
end

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

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

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

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

type-synonym $('\vartheta,'\alpha)$ *alphabet-rp* $= ('\vartheta,'\alpha)$ *alpha-rp-scheme alphabet*
type-synonym $('\vartheta,'\alpha,'\beta)$ *relation-rp* $= (('\vartheta,'\alpha)$ *alphabet-rp*, $('\vartheta,'\beta)$ *alphabet-rp*) *relation*
type-synonym $('\vartheta,'\alpha)$ *hrelation-rp* $= (('\vartheta,'\alpha)$ *alphabet-rp*, $('\vartheta,'\alpha)$ *alphabet-rp*) *relation*
type-synonym $('\vartheta,'\sigma)$ *predicate-rp* $= ('\vartheta,'\sigma)$ *alphabet-rp upred*

lift-definition *lift-rea* :: $('\alpha,\ '\beta)$ *relation* $\Rightarrow$ $('\vartheta,\ '\alpha,\ '\beta)$ *relation-rp* $(\lceil\text{-}\rceil_R)$ **is**
$\lambda\ P\ (A,\ A').\ P\ (more\ A,\ more\ A')$ **.**

lift-definition *drop-rea* :: $('\vartheta,\ '\alpha,\ '\beta)$ *relation-rp* $\Rightarrow$ $('\alpha,\ '\beta)$ *relation* $(\lfloor\text{-}\rfloor_R)$ **is**
$\lambda\ P\ (A,\ A').\ P\ ((\!|\ des\text{-}ok = True,\ rp\text{-}wait = True,\ rp\text{-}tr = [],\ rp\text{-}ref = \{\},\ \ldots = A\ |\!)$,
$(\!|\ des\text{-}ok = True,\ rp\text{-}wait = True,\ rp\text{-}tr = [],\ rp\text{-}ref = \{\},\ \ldots = A'\ |\!))$ **.**

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

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

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

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 $RH(P) = R1(R2(R3c(P)))$

lemma *R1-idem*: $R1(R1(P)) = R1(P)$
by *pred-tac*

lemma *R2-idem*: $R2(R2(P)) = R2(P)$
by (*pred-tac*)

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

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

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

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

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

lemma *R2-seqr-form*:
  fixes $P\ Q$ :: $('\vartheta,'\alpha,'\alpha)$ *relation-rp*
  shows $(R2(P)\ ;;\ R2(Q)) =$
    $(\exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket)\ ;;\ (Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket))$
      $\wedge\ (\$tr' =_u \$tr\ \hat{}_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg))$
proof −
  have $(R2(P)\ ;;\ R2(Q)) = (\exists\ tr_0 \cdot (R2(P))\llbracket\ll tr_0\gg/\$tr'\rrbracket\ ;;\ (R2(Q))\llbracket\ll tr_0\gg/\$tr\rrbracket)$
    by (*subst seqr-middle*[*of tr*], *simp-all*)
  also have ... =
    $(\exists\ tr_0 \cdot \exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket \wedge \ll tr_0\gg =_u \$tr\ \hat{}_u \ll tt_1\gg)\ ;;$
      $(Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket \wedge \$tr' =_u \ll tr_0\gg\ \hat{}_u \ll tt_2\gg)))$
    by (*simp add*: *R2-form usubst unrest uquant-lift var-in-var var-out-var*, *rel-tac*)
  also have ... =
    $(\exists\ tr_0 \cdot \exists\ tt_1 \cdot \exists\ tt_2 \cdot ((\ll tr_0\gg =_u \$tr\ \hat{}_u \ll tt_1\gg \wedge P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket)\ ;;$
      $(Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket \wedge \$tr' =_u \ll tr_0\gg\ \hat{}_u \ll tt_2\gg)))$
    by (*simp add*: *conj-comm*)
  also have ... =
    $(\exists\ tt_1 \cdot \exists\ tt_2 \cdot \exists\ tr_0 \cdot ((P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket)\ ;;\ (Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket))$
      $\wedge \ll tr_0\gg =_u \$tr\ \hat{}_u \ll tt_1\gg \wedge \$tr' =_u \ll tr_0\gg\ \hat{}_u \ll tt_2\gg)$
    by (*simp add*: *seqr-pre-out seqr-post-out unrest*, *rel-tac*)
  also have ... =
    $(\exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket)\ ;;\ (Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket))$
      $\wedge\ (\exists\ tr_0 \cdot \ll tr_0\gg =_u \$tr\ \hat{}_u \ll tt_1\gg \wedge \$tr' =_u \ll tr_0\gg\ \hat{}_u \ll tt_2\gg))$
    by *rel-tac*
  also have ... =
    $(\exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket)\ ;;\ (Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket))$
      $\wedge\ (\$tr' =_u \$tr\ \hat{}_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg))$
    by *rel-tac*
  finally show *?thesis* **.**
qed

lemma *R2-seqr-distribute*:
  fixes $P\ Q$ :: $('\vartheta,'\alpha,'\alpha)$ *relation-rp*
  shows $R2(R2(P)\ ;;\ R2(Q)) = (R2(P)\ ;;\ R2(Q))$
proof −
  have $R2(R2(P)\ ;;\ R2(Q)) =$
    $((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket\ ;;\ Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket)\llbracket(\$tr' - \$tr)/\$tr'\rrbracket$
      $\wedge\ \$tr' - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg) \wedge \$tr' \geq_u \$tr)$
    by (*simp add*: *R2-seqr-form*, *simp add*: *R2-def usubst unrest*, *rel-tac*)
  also have ... =
    $((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket\ ;;\ Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket)\llbracket(\ll tt_1\gg\ \hat{}_u \ll tt_2\gg)/\$tr'\rrbracket$
      $\wedge\ \$tr' - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg) \wedge \$tr' \geq_u \$tr)$
      by (*subst subst-eq-replace*, *simp*)
  also have ... =
    $((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket\ ;;\ Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket)$
      $\wedge\ \$tr' - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg) \wedge \$tr' \geq_u \$tr)$
      by (*simp add*: *usubst unrest*)
  also have ... =
    $(\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_1\gg/\$tr'\rrbracket\ ;;\ Q\llbracket\langle\rangle/\$tr\rrbracket\llbracket\ll tt_2\gg/\$tr'\rrbracket)$
      $\wedge\ (\$tr' - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg \wedge \$tr' \geq_u \$tr))$

by *pred-tac*
also have ... =
$((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr']\!]\ ;;\ Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr']\!])$
$\wedge\ \$tr' =_u \$tr\ \hat{}_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg))$
proof −
have $\bigwedge tt_1\ tt_2.\ (((\$tr' - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg) \wedge \$tr' \geq_u \$tr) :: ('\vartheta,'\alpha,'\alpha)\ \textit{relation-rp})$
$= (\$tr' =_u \$tr\ \hat{}_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg)$
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