Isabelle/UTP: Mechanised reasoning for the UTP

Simon Foster Frank Zeyda

July 6, 2016

Contents

1 UTP variables

theory *utp-var*
imports
../contrib/Kleene-Algebra/Quantales
../contrib/HOL−Algebra2/Complete-Lattice
../utils/cardinals
../utils/Continuum
../utils/finite-bijection
../utils/Lenses
../utils/Library-extra/Pfun
../utils/Library-extra/Ffun
../utils/Library-extra/Derivative-extra
../utils/Library-extra/List-lexord-alt
~~/src/HOL/Library/Prefix-Order
~~/src/HOL/Library/Char-ord
~~/src/HOL/Library/Adhoc-Overloading
~~/src/HOL/Library/Monad-Syntax
~~/src/HOL/Library/Countable
~~/src/HOL/Eisbach/Eisbach
utp-parser-utils
begin

no-notation *inner* (**infix** · *70*)

no-notation *le* (**infixl** $\sqsubseteq_1$ *50*)

no-notation
Set.member (*op* :) **and**
Set.member ((-/ : -) [*51*, *51*] *50*)

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.

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

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

syntax *-VAR* :: *id* $\Rightarrow$ $('a, 'r)$ *uvar* (*VAR* -)
translations *VAR x* => *FLDLENS x*

abbreviation *semi-uvar* ≡ *mwb-lens*

abbreviation *uvar* ≡ *vwb-lens*

We also define some lifting functions for variables to create input and output variables. These simply lift the alphabet to a tuple type since relations will ultimately be defined to a tuple alphabet.

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

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

definition *pr-var* :: $('a, '\beta)$ *uvar* $\Rightarrow$ $('a, '\beta)$ *uvar* **where**
[*simp*]: *pr-var* $x = x$

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

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

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

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

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

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

lemma *in-var-indep* [*simp*]:
$x \bowtie y \Longrightarrow$ *in-var* $x \bowtie$ *in-var* y
by (*simp add*: *in-var-def out-var-def fst-vwb-lens lens-indep-left-comp*)

lemma *out-var-indep* [*simp*]:
$x \bowtie y \Longrightarrow$ *out-var* $x \bowtie$ *out-var* y
by (*simp add*: *lens-indep-left-comp out-var-def snd-vwb-lens*)

We also define some lookup abstraction simplifications.

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

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

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

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

Variables can also be used to effectively define sets of variables. Here we define the the universal alphabet (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.

abbreviation (*input*) *univ-alpha* :: ($'\alpha$, $'\alpha$) *uvar* (Σ) **where**
univ-alpha $\equiv$ 1_L

nonterminal *svid* **and** *svar* **and** *salpha*

syntax
-salphaid :: *id* ⇒ *salpha* (- [998] 998)
-salphavar :: *svar* ⇒ *salpha* (- [998] 998)
-salphacomp :: *salpha* ⇒ *salpha* ⇒ *salpha* (**infixr** , 75)
-salphacomp :: *salpha* ⇒ *salpha* ⇒ *salpha* (**infixr** ; 75)
-svid :: *id* ⇒ *svid* (- [999] 999)
-svid-alpha :: *svid* (Σ)
-svid-empty :: *svid* (∅)
-svid-dot :: *svid* ⇒ *svid* ⇒ *svid* (-:- [999,998] 999)
-spvar :: *svid* ⇒ *svar* (&- [998] 998)
-sinvar :: *svid* ⇒ *svar* ($- [998] 998)
-soutvar :: *svid* ⇒ *svar* ($-´ [998] 998)

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

adhoc-overloading
svar pr-var **and** *ivar in-var* **and** *ovar out-var*

translations
-salphaid x => *x*
-salphacomp x y => $x +_L y$
-salphavar x => *x*
-svid-alpha == Σ
-svid-empty == 0_L
-svid-dot x y => $y ;_L x$
-svid x => *x*
-sinvar (*-svid-dot x y*) <= *CONST ivar* (*CONST lens-comp y x*)
-soutvar (*-svid-dot x y*) <= *CONST ovar* (*CONST lens-comp y x*)
-spvar x == *CONST svar x*
-sinvar x == *CONST ivar x*
-soutvar x == *CONST ovar x*

Syntactic function to construct a uvar type given a return type

syntax
-uvar-ty :: *type* ⇒ *type* ⇒ *type*

parse-translation ⟪
```
let
  fun uvar-ty-tr [ty] = Syntax.const @{type-syntax uvar} $ ty $ Syntax.const @{type-syntax dummy}
    | uvar-ty-tr ts = raise TERM (uvar-ty-tr, ts);
in [(@{syntax-const -uvar-ty}, K uvar-ty-tr)] end
```
⟫

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

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

translations

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

lemma *ucard-non-empty*:
$\mathcal{U}(x) \neq \{\}$
by (*induct x*, *auto*)

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

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

lemma *ucard-of-uncountably-infinite* [*simp*]:
$uncountable\ (UNIV :: 'a\ set) \Longrightarrow UCARD('a :: continuum) = \mathrm{c}$
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*:
$finite\ (UNIV :: 'a::continuum\ set) \Longrightarrow uinject = (\lambda\ x :: 'a.\ \{to\text{-}nat\text{-}fin\ x\})$
by (*rule ext*, *auto simp add*: *uinject-def*)

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

lemma *card-finite-lemma*:
assumes *finite* (*UNIV* :: *'a set*)
shows $x < card\ (UNIV :: 'a\ set) \longleftrightarrow x \leq card\ (UNIV :: 'a\ set) - Suc\ 0$
proof −
have $card\ (UNIV :: 'a\ set) > 0$
by (*simp add*: *assms finite-UNIV-card-ge-0*)
thus *?thesis*

by *linarith*
qed

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

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

lemma *uinject-card* [*simp*]: *uinject* (*x* :: *'a::continuum*) ∈ $\mathcal{U}$(*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 ∈ $\mathcal{U}$(*UCARD*(*'a::continuum*)) ⟹ *uinject* ((*uproject* :: *nat set* ⇒ *'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* ⇒ *uuniv*. ∀ *x*. *f*(*x*) ∈ $\mathcal{U}$(*dname-card x*)}
  apply (*rule-tac x*=λ *x*. {*0*} **in** *exI*)
  apply (*auto*)
  apply (*rename-tac x*)
  apply (*case-tac dname-card x*)

apply (*simp-all*)
done

setup-lifting *type-definition-vstore*

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

setup-lifting *type-definition-dvar*

lift-definition *mk-dvar* :: *string* ⇒ ($'a$::{*continuum*,*two*}) *dvar* ($\lceil$-$\rceil_d$)
is λ *n*. (| *dname-name* = *n*, *dname-card* = *UCARD*($'a$) |)
by *auto*

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

lemma *dvar-name* [*simp*]: *dvar-name* $\lceil x \rceil_d$ = *x*
by (*transfer*, *simp*)

term *fun-lens*

setup-lifting *type-definition-lens-ext*

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

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

definition *dvar-lens* :: ($'a$::*continuum*) *dvar* ⇒ ($'a$ ⟹ *vstore*) **where**
dvar-lens x = (| *lens-get* = *dvar-get x*, *lens-put* = *dvar-put x* |)

lemma *vstore-vwb-lens* [*simp*]:
vwb-lens (*dvar-lens x*)
apply (*unfold-locales*)
apply (*simp-all add*: *dvar-lens-def*)
apply (*transfer*, *auto*)
apply (*transfer*)
apply (*metis fun-upd-idem uproject-inv*)
apply (*transfer*, *simp*)
done

lemma *dvar-lens-indep-iff*:
fixes *x* :: $'a$::{*continuum*,*two*} *dvar* **and** *y* :: $'b$::{*continuum*,*two*} *dvar*
shows *dvar-lens x* ⋈ *dvar-lens y* ⟷ (*dvar-dname x* ≠ *dvar-dname y*)
proof –
obtain *v1 v2* :: $'b$::{*continuum*,*two*} **where** *v*:*v1* ≠ *v2*
using *two-diff* **by** *auto*
obtain *u* :: $'a$::{*continuum*,*two*} **and** *v* :: $'b$::{*continuum*,*two*}
where *uv*: *uinject u* ≠ *uinject v*
by (*metis* (*full-types*) *uinject-inv v*)
show *?thesis*

proof (*simp add*: *dvar-lens-def lens-indep-def*, *transfer*, *auto simp add*: *fun-upd-twist*)
fix *ya* :: *dname*
assume *a1*: *ucard-of* (*TYPE*('*b*)::'*b itself*) = *ucard-of* (*TYPE*('*a*)::'*a itself*)
assume *dname-card ya* = *ucard-of* (*TYPE*('*a*)::'*a itself*)
assume *a2*: ∀ *u v σ*. (∀ *x*. *σ x* ∈ $\mathcal{U}$(*dname-card x*)) ⟶ *σ*(*ya* := *uinject* (*u*::'*a*)) = *σ*(*ya* := *uinject* (*v*::'*b*)) ∧ (*uproject* (*uinject v*)::'*a*) = *uproject* (*σ ya*) ∧ (*uproject* (*uinject u*)::'*b*) = *uproject* (*σ ya*)
obtain *NN* :: *vstore* ⇒ *dname* ⇒ *nat set* **where**
⋀*v*. ∀ *d*. *NN v d* ∈ $\mathcal{U}$(*dname-card d*)
by (*metis* (*lifting*) *Abs-vstore-cases mem-Collect-eq*)
then show *False*
using *a2 a1* **by** (*metis uinject-card uproject-inv uv*)
qed
qed

The vst class provides the location of the store in a larger type via a lens

class *vst* =
fixes *vstore-lens* :: *vstore* ⟹ '*a* ($\mathcal{V}$)
assumes *vstore-vwb-lens* [*simp*]: *vwb-lens vstore-lens*

definition *dvar-lift* :: '*a*::*continuum dvar* ⇒ ('*a*, '*α*::*vst*) *uvar* (-↑ [*999*] *999*) **where**
dvar-lift x = *dvar-lens x* $;_L$ *vstore-lens*

definition [*simp*]: *in-dvar x* = *in-var* (*x*↑)
definition [*simp*]: *out-dvar x* = *out-var* (*x*↑)

adhoc-overloading
ivar in-dvar **and** *ovar out-dvar* **and** *svar dvar-lift*

lemma *uvar-dvar*: *uvar* (*x*↑)
by (*auto intro*: *comp-vwb-lens simp add*: *dvar-lift-def*)

Deep variables with different names are independent

lemma *dvar-lift-indep-iff*:
fixes *x* :: '*a*::{*continuum*,*two*} *dvar* **and** *y* :: '*b*::{*continuum*,*two*} *dvar*
shows *x*↑ ⋈ *y*↑ ⟷ *dvar-dname x* ≠ *dvar-dname y*
proof −
have *x*↑ ⋈ *y*↑ ⟷ *dvar-lens x* ⋈ *dvar-lens y*
by (*metis dvar-lift-def lens-comp-indep-cong-left lens-indep-left-comp vst-class*.*vstore-vwb-lens vwb-lens-mwb*)
also have ... ⟷ *dvar-dname x* ≠ *dvar-dname y*
by (*simp add*: *dvar-lens-indep-iff*)
finally show *?thesis* **.**
qed

lemma *dvar-indep-diff-name'* [*simp*]:
x ≠ *y* ⟹ $\lceil x\rceil_d$↑ ⋈ $\lceil y\rceil_d$↑
by (*simp add*: *dvar-lift-indep-iff mk-dvar*.*rep-eq*)

A basic record structure for vstores

record *vstore-d* =
vstore :: *vstore*

instantiation *vstore-d-ext* :: (*type*) *vst*
begin
definition *vstore-lens-vstore-d-ext* = *VAR vstore*
instance

by (*intro-classes*, *unfold-locales*, *simp-all add*: *vstore-lens-vstore-d-ext-def*)
end

syntax
-*sin-dvar* :: *id* ⇒ *svar* (%- [*999*] *999*)
-*sout-dvar* :: *id* ⇒ *svar* (%-´ [*999*] *999*)

translations
-*sin-dvar x* => *CONST in-dvar* (*CONST mk-dvar IDSTR*(*x*))
-*sout-dvar x* => *CONST out-dvar* (*CONST mk-dvar IDSTR*(*x*))

definition *MkDVar x* = $\lceil x \rceil_d \uparrow$

lemma *uvar-MkDVar* [*simp*]: *uvar* (*MkDVar x*)
by (*simp add*: *MkDVar-def uvar-dvar*)

lemma *MkDVar-indep* [*simp*]: $x \neq y \Longrightarrow$ *MkDVar x* ⋈ *MkDVar y*
apply (*rule lens-indepI*)
apply (*simp-all add*: *MkDVar-def*)
apply (*meson dvar-indep-diff-name′ lens-indep-comm*)
done

lemma *MkDVar-put-comm* [*simp*]:
$m <_l n \Longrightarrow put_{MkDVar\ n}\ (put_{MkDVar\ m}\ s\ u)\ v = put_{MkDVar\ m}\ (put_{MkDVar\ n}\ s\ v)\ u$
by (*simp add*: *lens-indep-comm*)

Set up parsing and pretty printing for deep variables

syntax
-*dvar* :: *id* ⇒ *svid* (<->)
-*dvar-ty* :: *id* ⇒ *type* ⇒ *svid* (<-::->)
-*dvard* :: *id* ⇒ *logic* ($<\text{-}>_d$)
-*dvar-tyd* :: *id* ⇒ *type* ⇒ *logic* ($<\text{-::-}>_d$)

translations
-*dvar x* => *CONST MkDVar IDSTR*(*x*)
-*dvar-ty x a* => -*constrain* (*CONST MkDVar IDSTR*(*x*)) (-*uvar-ty a*)
-*dvard x* => *CONST MkDVar IDSTR*(*x*)
-*dvar-tyd x a* => -*constrain* (*CONST MkDVar IDSTR*(*x*)) (-*uvar-ty a*)

print-translation ⟪
```
let fun MkDVar-tr′ - [name] =
      Const (@{syntax-const -dvar}, dummyT) $
        Name-Utils.mk-id (HOLogic.dest-string (Name-Utils.deep-unmark-const name))
  | MkDVar-tr′ - - = raise Match in
 [(@{const-syntax MkDVar}, MkDVar-tr′)]
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* $([\![-]\!]_e)$

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*)

named-theorems *ueval*

setup-lifting *type-definition-uexpr*

Get the alphabet of an expression

definition *alpha-of* :: $('a, '\alpha)$ *uexpr* $\Rightarrow ('\alpha, '\alpha)$ *lens* $(\alpha'(-'))$ **where**
alpha-of $e = 1_L$

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

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

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)

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

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

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

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

We also define a UTP expression version of function abstract

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

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

consts
  ulit :: $'t \Rightarrow 'e$ (≪-≫)
  ueq :: $'a \Rightarrow 'a \Rightarrow 'b$ (**infixl** $=_u$ *50*)

adhoc-overloading
  ulit lit

syntax
  -uuvar :: *svar* ⇒ *logic*

translations
  -uuvar x == *CONST var x*

syntax
  -uuvar :: *svar* ⇒ *logic* (-)

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

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

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

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

begin
  definition *inverse-uexpr* :: $('a, 'b)$ *uexpr* $\Rightarrow$ $('a, 'b)$ *uexpr*
  where *inverse-uexpr u* = *uop inverse u*
instance ..
end

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

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

instantiation *uexpr* :: (*abs*, *type*) *abs*
begin
  definition *abs-uexpr-def*: *abs u* = *uop abs u*
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* :: (*ab-semigroup-add*, *type*) *ab-semigroup-add*
  by (*intro-classes*) (*simp add*: *plus-uexpr-def*, *transfer*, *simp add*: *add.commute*)+

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

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

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

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

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

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

instance *uexpr* :: (*ordered-ab-group-add-abs*, *type*) *ordered-ab-group-add-abs*
apply (*intro-classes*)
apply (*simp add*: *abs-uexpr-def zero-uexpr-def plus-uexpr-def uminus-uexpr-def*, *transfer*, *simp add*: *abs-ge-self abs-le-iff abs-triangle-ineq*)+
apply (*metis abs-ge-self abs-le-iff abs-minus-cancel abs-triangle-ineq4 add-mono*)
done

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

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

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

Set up automation for numerals

lemma *numeral-uexpr-rep-eq*: $[\![\mathit{numeral}\ x]\!]_e\ b = \mathit{numeral}\ x$
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*, 'α) *uexpr* $\Rightarrow$ ('*a*, 'α) *uexpr* $\Rightarrow$ (*bool*, 'α) *uexpr*
where *eq-upred x y* = *bop HOL.eq x y*

adhoc-overloading
ueq eq-upred

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

consts
uempty :: '*f*
uapply :: '*f* $\Rightarrow$ '*k* $\Rightarrow$ '*v*
uupd :: '*f* $\Rightarrow$ '*k* $\Rightarrow$ '*v* $\Rightarrow$ '*f*
udom :: '*f* $\Rightarrow$ '*a set*
uran :: '*f* $\Rightarrow$ '*b set*
udomres :: '*a set* $\Rightarrow$ '*f* $\Rightarrow$ '*f*
uranres :: '*f* $\Rightarrow$ '*b set* $\Rightarrow$ '*f*
ucard :: '*f* $\Rightarrow$ *nat*

definition *LNil* = *Nil*
definition *LZero* = *0*

adhoc-overloading
uempty LZero **and** *uempty LNil* **and**
uapply fun-apply **and** *uapply nth* **and** *uapply pfun-app* **and** *uapply ffun-app* **and**
uupd pfun-upd **and** *uupd ffun-upd* **and** *uupd list-update* **and**
udom Domain **and** *udom pdom* **and** *udom fdom* **and** *udom seq-dom* **and**
udom Range **and** *uran pran* **and** *uran fran* **and** *uran set* **and**
udomres pdom-res **and** *udomres fdom-res* **and**
uranres pran-res **and** *udomres fran-res* **and**
ucard card **and** *ucard pcard* **and** *ucard length*

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

syntax
-ucoerce :: ('*a*, 'α) *uexpr* $\Rightarrow$ *type* $\Rightarrow$ ('*a*, 'α) *uexpr* (**infix** $:_u$ *50*)
-unil :: ('*a list*, 'α) *uexpr* ($\langle\rangle$)
-ulist :: *args* => ('*a list*, 'α) *uexpr* ($\langle$(-)$\rangle$)
-uappend :: ('*a list*, 'α) *uexpr* $\Rightarrow$ ('*a list*, 'α) *uexpr* $\Rightarrow$ ('*a list*, 'α) *uexpr* (**infixr** $\hat{}_u$ *80*)
-ulast :: ('*a list*, 'α) *uexpr* $\Rightarrow$ ('*a*, 'α) *uexpr* ($last_u$'(-'))
-ufront :: ('*a list*, 'α) *uexpr* $\Rightarrow$ ('*a list*, 'α) *uexpr* ($front_u$'(-'))
-uhead :: ('*a list*, 'α) *uexpr* $\Rightarrow$ ('*a*, 'α) *uexpr* ($head_u$'(-'))
-utail :: ('*a list*, 'α) *uexpr* $\Rightarrow$ ('*a list*, 'α) *uexpr* ($tail_u$'(-'))
-ucard :: ('*a list*, 'α) *uexpr* $\Rightarrow$ (*nat*, 'α) *uexpr* ($\#_u$'(-'))
-ufilter :: ('*a list*, 'α) *uexpr* $\Rightarrow$ ('*a set*, 'α) *uexpr* $\Rightarrow$ ('*a list*, 'α) *uexpr* (**infixl** $\restriction_u$ *75*)
-uextract :: ('*a set*, 'α) *uexpr* $\Rightarrow$ ('*a list*, 'α) *uexpr* $\Rightarrow$ ('*a list*, 'α) *uexpr* (**infixl** $\upharpoonleft_u$ *75*)
-uelems :: ('*a list*, 'α) *uexpr* $\Rightarrow$ ('*a set*, 'α) *uexpr* ($elems_u$'(-'))
-usorted :: ('*a list*, 'α) *uexpr* $\Rightarrow$ (*bool*, 'α) *uexpr* ($sorted_u$'(-'))
-udistinct :: ('*a list*, 'α) *uexpr* $\Rightarrow$ (*bool*, 'α) *uexpr* ($distinct_u$'(-'))
-uless :: ('*a*, 'α) *uexpr* $\Rightarrow$ ('*a*, 'α) *uexpr* $\Rightarrow$ (*bool*, 'α) *uexpr* (**infix** $<_u$ *50*)
-uleq :: ('*a*, 'α) *uexpr* $\Rightarrow$ ('*a*, 'α) *uexpr* $\Rightarrow$ (*bool*, 'α) *uexpr* (**infix** $\leq_u$ *50*)
-ugreat :: ('*a*, 'α) *uexpr* $\Rightarrow$ ('*a*, 'α) *uexpr* $\Rightarrow$ (*bool*, 'α) *uexpr* (**infix** $>_u$ *50*)
-ugeq :: ('*a*, 'α) *uexpr* $\Rightarrow$ ('*a*, 'α) *uexpr* $\Rightarrow$ (*bool*, 'α) *uexpr* (**infix** $\geq_u$ *50*)

$\textit{-uempset} \quad :: ('a\ set,\ '\alpha)\ uexpr\ (\{\}_u)$
$\textit{-uset} \quad :: args => ('a\ set,\ '\alpha)\ uexpr\ (\{(\text{-})\}_u)$
$\textit{-uunion} \quad :: ('a\ set,\ '\alpha)\ uexpr \Rightarrow ('a\ set,\ '\alpha)\ uexpr \Rightarrow ('a\ set,\ '\alpha)\ uexpr\ (\textbf{infixl} \cup_u\ 65)$
$\textit{-uinter} \quad :: ('a\ set,\ '\alpha)\ uexpr \Rightarrow ('a\ set,\ '\alpha)\ uexpr \Rightarrow ('a\ set,\ '\alpha)\ uexpr\ (\textbf{infixl} \cap_u\ 70)$
$\textit{-umem} \quad :: ('a,\ '\alpha)\ uexpr \Rightarrow ('a\ set,\ '\alpha)\ uexpr \Rightarrow (bool,\ '\alpha)\ uexpr\ (\textbf{infix} \in_u\ 50)$
$\textit{-usubset} \quad :: ('a\ set,\ '\alpha)\ uexpr \Rightarrow ('a\ set,\ '\alpha)\ uexpr \Rightarrow (bool,\ '\alpha)\ uexpr\ (\textbf{infix} \subset_u\ 50)$
$\textit{-usubseteq} \quad :: ('a\ set,\ '\alpha)\ uexpr \Rightarrow ('a\ set,\ '\alpha)\ uexpr \Rightarrow (bool,\ '\alpha)\ uexpr\ (\textbf{infix} \subseteq_u\ 50)$
$\textit{-utuple} \quad :: ('a,\ '\alpha)\ uexpr \Rightarrow utuple\text{-}args \Rightarrow ('a * 'b,\ '\alpha)\ uexpr\ ((1'(\text{-},/\ \text{-}')_u))$
$\textit{-utuple-arg} \quad :: ('a,\ '\alpha)\ uexpr \Rightarrow utuple\text{-}args\ (\text{-})$
$\textit{-utuple-args} :: ('a,\ '\alpha)\ uexpr => utuple\text{-}args \Rightarrow utuple\text{-}args \quad (\text{-},/\ \text{-})$
$\textit{-uunit} \quad :: ('a,\ '\alpha)\ uexpr\ ('(')_u)$
$\textit{-ufst} \quad :: ('a \times 'b,\ '\alpha)\ uexpr \Rightarrow ('a,\ '\alpha)\ uexpr\ (\pi_1'(\text{-}'))$
$\textit{-usnd} \quad :: ('a \times 'b,\ '\alpha)\ uexpr \Rightarrow ('b,\ '\alpha)\ uexpr\ (\pi_2'(\text{-}'))$
$\textit{-uapply} \quad :: ('a \Rightarrow 'b,\ '\alpha)\ uexpr \Rightarrow utuple\text{-}args \Rightarrow ('b,\ '\alpha)\ uexpr\ (\text{-}(\!|\text{-}|\!)_u\ [999,0]\ 999)$
$\textit{-ulamba} \quad :: pttrn \Rightarrow logic \Rightarrow logic\ (\lambda\ \text{-}\ \cdot\ \text{-}\ [0,\ 10]\ 10)$
$\textit{-udom} \quad :: logic \Rightarrow logic\ (dom_u'(\text{-}'))$
$\textit{-uran} \quad :: logic \Rightarrow logic\ (ran_u'(\text{-}'))$
$\textit{-uinl} \quad :: logic \Rightarrow logic\ (inl_u'(\text{-}'))$
$\textit{-uinr} \quad :: logic \Rightarrow logic\ (inr_u'(\text{-}'))$
$\textit{-umap-empty} :: logic\ ([]_u)$
$\textit{-umap-plus} \quad :: logic \Rightarrow logic \Rightarrow logic\ (\textbf{infixl} \oplus_u\ 85)$
$\textit{-umap-minus} :: logic \Rightarrow logic \Rightarrow logic\ (\textbf{infixl} \ominus_u\ 85)$
$\textit{-udom-res} \quad :: logic \Rightarrow logic \Rightarrow logic\ (\textbf{infixl} \lhd_u\ 85)$
$\textit{-uran-res} \quad :: logic \Rightarrow logic \Rightarrow logic\ (\textbf{infixl} \rhd_u\ 85)$
$\textit{-umaplet} \quad :: [logic,\ logic] => umaplet\ (\text{-}\ /\mapsto/\ \text{-})$
$\quad :: umaplet => umaplets \quad (\text{-})$
$\textit{-UMaplets} \quad :: [umaplet,\ umaplets] => umaplets\ (\text{-},/\ \text{-})$
$\textit{-UMapUpd} \quad :: [logic,\ umaplets] => logic\ (\text{-}/'(\text{-}')_u\ [900,0]\ 900)$
$\textit{-UMap} \quad :: umaplets => logic\ ((1[\text{-}]_u))$

translations
$f(\!|v|\!)_u <= CONST\ uapply\ f\ v$
$dom_u(f) <= CONST\ udom\ f$
$ran_u(f) <= CONST\ uran\ f$
$A \lhd_u f <= CONST\ udomres\ A\ f$
$f \rhd_u A <= CONST\ uranres\ f\ A$
$\#_u(f) <= CONST\ ucard\ f$
$f(k \mapsto v)_u <= CONST\ uupd\ f\ k\ v$

translations
$x :_u 'a == x :: ('a,\ \text{-})\ uexpr$
$\langle\rangle \quad == \ll[]\gg$
$\langle x,\ xs\rangle \quad == CONST\ bop\ (op\ \#)\ x\ \langle xs\rangle$
$\langle x\rangle \quad == CONST\ bop\ (op\ \#)\ x \ll[]\gg$
$x\ \hat{}_u\ y \quad == CONST\ bop\ (op\ @)\ x\ y$
$last_u(xs) == CONST\ uop\ CONST\ last\ xs$
$front_u(xs) == CONST\ uop\ CONST\ butlast\ xs$
$head_u(xs) == CONST\ uop\ CONST\ hd\ xs$
$tail_u(xs) == CONST\ uop\ CONST\ tl\ xs$
$\#_u(xs) == CONST\ uop\ CONST\ ucard\ xs$
$elems_u(xs) == CONST\ uop\ CONST\ set\ xs$
$sorted_u(xs) == CONST\ uop\ CONST\ sorted\ xs$
$distinct_u(xs) == CONST\ uop\ CONST\ distinct\ xs$
$xs \upharpoonleft_u A \quad == CONST\ bop\ CONST\ seq\text{-}filter\ xs\ A$
$A \upharpoonright_u xs \quad == CONST\ bop\ (op \upharpoonright_l)\ A\ xs$

$x <_u y \ \ == CONST\ bop\ (op <)\ x\ y$
$x \leq_u y \ \ == CONST\ bop\ (op \leq)\ x\ y$
$x >_u y \ \ == y <_u x$
$x \geq_u y \ \ == y \leq_u x$
$\{\}_u \ \ == \ll\{\}\gg$
$\{x,\ xs\}_u == CONST\ bop\ (CONST\ insert)\ x\ \{xs\}_u$
$\{x\}_u \ \ == CONST\ bop\ (CONST\ insert)\ x \ll\{\}\gg$
$A \cup_u B \ \ == CONST\ bop\ (op \cup)\ A\ B$
$A \cap_u B \ \ == CONST\ bop\ (op \cap)\ A\ B$
$f \oplus_u g \ \ => (f :: ((\text{-},\ \text{-})\ pfun,\ \text{-})\ uexpr) + g$
$f \ominus_u g \ \ => (f :: ((\text{-},\ \text{-})\ pfun,\ \text{-})\ uexpr) - g$
$x \in_u A \ \ == CONST\ bop\ (op \in)\ x\ A$
$A \subset_u B \ \ == CONST\ bop\ (op <)\ A\ B$
$A \subset_u B \ \ <= CONST\ bop\ (op \subset)\ A\ B$
$f \subset_u g \ \ <= CONST\ bop\ (op \subset_p)\ f\ g$
$f \subset_u g \ \ <= CONST\ bop\ (op \subset_f)\ f\ g$
$A \subseteq_u B \ \ == CONST\ bop\ (op \leq)\ A\ B$
$A \subseteq_u B \ \ <= CONST\ bop\ (op \subseteq)\ A\ B$
$f \subseteq_u g \ \ <= CONST\ bop\ (op \subseteq_p)\ f\ g$
$f \subseteq_u g \ \ <= CONST\ bop\ (op \subseteq_f)\ f\ g$
$()_u \ \ == \ll()\gg$
$(x,\ y)_u \ \ == CONST\ bop\ (CONST\ Pair)\ x\ y$
$\text{-}utuple\ x\ (\text{-}utuple\text{-}args\ y\ z) == \text{-}utuple\ x\ (\text{-}utuple\text{-}arg\ (\text{-}utuple\ y\ z))$
$\pi_1(x) \ \ == CONST\ uop\ CONST\ fst\ x$
$\pi_2(x) \ \ == CONST\ uop\ CONST\ snd\ x$
$f(\!|x|\!)_u \ \ == CONST\ bop\ CONST\ uapply\ f\ x$
$\lambda\ x \cdot p == CONST\ ulambda\ (\lambda\ x.\ p)$
$dom_u(f) == CONST\ uop\ CONST\ udom\ f$
$ran_u(f) == CONST\ uop\ CONST\ uran\ f$
$inl_u(x) == CONST\ uop\ CONST\ Inl\ x$
$inr_u(x) == CONST\ uop\ CONST\ Inr\ x$
$[]_u \ \ == \ll CONST\ uempty\gg$
$A \lhd_u f == CONST\ bop\ (CONST\ udomres)\ A\ f$
$f \rhd_u A == CONST\ bop\ (CONST\ uranres)\ f\ A$
$\text{-}UMapUpd\ m\ (\text{-}UMaplets\ xy\ ms) == \text{-}UMapUpd\ (\text{-}UMapUpd\ m\ xy)\ ms$
$\text{-}UMapUpd\ m\ (\text{-}umaplet\ \ x\ y) \ \ == CONST\ trop\ CONST\ uupd\ m\ x\ y$
$\text{-}UMap\ ms \ \ == \text{-}UMapUpd\ []_u\ ms$
$\text{-}UMap\ (\text{-}UMaplets\ ms1\ ms2) \ \ <= \text{-}UMapUpd\ (\text{-}UMap\ ms1)\ ms2$
$\text{-}UMaplets\ ms1\ (\text{-}UMaplets\ ms2\ ms3) <= \text{-}UMaplets\ (\text{-}UMaplets\ ms1\ ms2)\ ms3$
$f(\!|x,y|\!)_u \ \ == CONST\ bop\ CONST\ uapply\ f\ (x,y)_u$

Lifting set intervals

syntax
$\text{-}uset\text{-}atLeastAtMost :: ('a,\ '\alpha)\ uexpr \Rightarrow ('a,\ '\alpha)\ uexpr \Rightarrow ('a\ set,\ '\alpha)\ uexpr\ ((1\{\text{-}..\text{-}\}_u))$
$\text{-}uset\text{-}atLeastLessThan :: ('a,\ '\alpha)\ uexpr \Rightarrow ('a,\ '\alpha)\ uexpr \Rightarrow ('a\ set,\ '\alpha)\ uexpr\ ((1\{\text{-}..<\text{-}\}_u))$
$\text{-}uset\text{-}compr :: id \Rightarrow ('a\ set,\ '\alpha)\ uexpr \Rightarrow (bool,\ '\alpha)\ uexpr \Rightarrow ('b,\ '\alpha)\ uexpr \Rightarrow ('b\ set,\ '\alpha)\ uexpr\ ((1\{\text{-} :/\ \text{-}\ |/\ \text{-} \cdot/\ \text{-}\}_u))$

lift-definition *ZedSetCompr* ::
$('a\ set,\ '\alpha)\ uexpr \Rightarrow ('a \Rightarrow (bool,\ '\alpha)\ uexpr \times ('b,\ '\alpha)\ uexpr) \Rightarrow ('b\ set,\ '\alpha)\ uexpr$
is $\lambda\ A\ PF\ b.\ \{\ snd\ (PF\ x)\ b \mid x.\ x \in A\ b \land fst\ (PF\ x)\ b\}$ **.**

translations
$\{x..y\}_u == CONST\ bop\ CONST\ atLeastAtMost\ x\ y$
$\{x..<y\}_u == CONST\ bop\ CONST\ atLeastLessThan\ x\ y$

$\{x : A \mid P \cdot F\}_u$ == $CONST\ ZedSetCompr\ A\ (\lambda\ x.\ (P,\ F))$

Lifting limits

definition *ulim-left* = $(\lambda\ p\ f.\ Lim\ (at\text{-}left\ p)\ f)$
definition *ulim-right* = $(\lambda\ p\ f.\ Lim\ (at\text{-}right\ p)\ f)$
definition *ucont-on* = $(\lambda\ f\ A.\ continuous\text{-}on\ A\ f)$

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

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

lemmas *uexpr-defs* =
alpha-of-def
zero-uexpr-def
one-uexpr-def
plus-uexpr-def
uminus-uexpr-def
minus-uexpr-def
times-uexpr-def
inverse-uexpr-def
divide-uexpr-def
sgn-uexpr-def
abs-uexpr-def
mod-uexpr-def
eq-upred-def
numeral-uexpr-simp
ulim-left-def
ulim-right-def
ucont-on-def
LNil-def
LZero-def

2.1 Evaluation laws for expressions

lemma *lit-ueval* [*ueval*]: $[\![\ll x \gg]\!]_e b = x$
by (*transfer*, *simp*)

lemma *var-ueval* [*ueval*]: $[\![var\ x]\!]_e b = get_x\ b$
by (*transfer*, *simp*)

lemma *uop-ueval* [*ueval*]: $[\![uop\ f\ x]\!]_e b = f\ ([\![x]\!]_e b)$
by (*transfer*, *simp*)

lemma *bop-ueval* [*ueval*]: $[\![bop\ f\ x\ y]\!]_e b = f\ ([\![x]\!]_e b)\ ([\![y]\!]_e b)$
by (*transfer*, *simp*)

lemma *trop-ueval* [*ueval*]: $[\![trop\ f\ x\ y\ z]\!]_e b = f\ ([\![x]\!]_e b)\ ([\![y]\!]_e b)\ ([\![z]\!]_e b)$
by (*transfer*, *simp*)

declare *uexpr-defs* [*ueval*]

2.2 Misc laws

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

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

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

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

named-theorems *unrest*

lift-definition *unrest-upred* :: $('a, '\alpha)$ *uvar* $\Rightarrow$ $('b, '\alpha)$ *uexpr* $\Rightarrow$ *bool*
is $\lambda\ x\ e.\ \forall\ b\ v.\ e\ (put_x\ b\ v) = e\ b$ **.**

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

adhoc-overloading
unrest unrest-upred

lemma *unrest-var-comp* [*unrest*]:
$[\![\ x\ \sharp\ P;\ y\ \sharp\ P\ ]\!] \Longrightarrow x,y\ \sharp\ P$
by (*transfer*, *simp add*: *lens-defs*)

lemma *unrest-lit* [*unrest*]: $x\ \sharp\ \ll v \gg$
by (*transfer*, *simp*)

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

lemma *unrest-var* [*unrest*]: $[\![\ uvar\ x;\ x \bowtie y\ ]\!] \Longrightarrow y\ \sharp\ var\ x$
by (*transfer*, *auto*)

lemma *unrest-iuvar* [*unrest*]: $[\![\ uvar\ x;\ x \bowtie y\ ]\!] \Longrightarrow \$y\ \sharp\ \$x$
by (*metis in-var-indep in-var-uvar unrest-var*)

lemma *unrest-ouvar* [*unrest*]: $[\![\ uvar\ x;\ x \bowtie y\ ]\!] \Longrightarrow \$y\acute{}\ \sharp\ \$x\acute{}$
by (*metis out-var-indep out-var-uvar unrest-var*)

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

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

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

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

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

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

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

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

lemma *unrest-numeral* [*unrest*]: $x \sharp$ (*numeral n*)
  by (*simp add*: *numeral-uexpr-simp unrest-lit*)

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

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

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

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

lemma *unrest-minus* [*unrest*]: $[\![x \sharp u; x \sharp v]\!] \Longrightarrow x \sharp u - v$
  by (*simp add*: *minus-uexpr-def unrest*)

lemma *unrest-times* [*unrest*]: $[\![x \sharp u; x \sharp v]\!] \Longrightarrow x \sharp u * v$
  by (*simp add*: *times-uexpr-def unrest*)

lemma *unrest-divide* [*unrest*]: $[\![x \sharp u; x \sharp v]\!] \Longrightarrow x \sharp u / v$
  by (*simp add*: *divide-uexpr-def unrest*)

end

4 Substitution

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

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

definition $subst\text{-}upd\text{-}dvar :: '\alpha\ usubst \Rightarrow 'a{::}continuum\ dvar \Rightarrow ('a, '\alpha{::}vst)\ uexpr \Rightarrow '\alpha\ usubst$ **where**
$subst\text{-}upd\text{-}dvar\ \sigma\ x\ v = subst\text{-}upd\text{-}uvar\ \sigma\ (x\uparrow)\ v$

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

Lookup the expression associated with a variable in a substitution

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

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

definition $unrest\text{-}usubst :: ('a, '\alpha)\ uvar \Rightarrow '\alpha\ usubst \Rightarrow bool$
where $unrest\text{-}usubst\ x\ \sigma = (\forall\ \varrho\ v.\ \sigma\ (put_x\ \varrho\ v) = put_x\ (\sigma\ \varrho)\ v)$

adhoc-overloading
  unrest unrest-usubst

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

syntax
  -smaplet :: [*salpha*, *'a*] => *smaplet*   (- /$\mapsto_s$/ -)

:: *smaplet* => *smaplets* (-)
-*SMaplets* :: [*smaplet*, *smaplets*] => *smaplets* (-,/ -)
-*SubstUpd* :: ['*m usubst*, *smaplets*] => '*m usubst* (-/'(-') [*900*,*0*] *900*)
-*Subst* :: *smaplets* => '*a* ⇀ '*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*

Deletion of a substitution maplet

definition *subst-del* :: $'\alpha$ *usubst* ⇒ ($'a$, $'\alpha$) *uvar* ⇒ $'\alpha$ *usubst* (**infix** $-_s$ *85*) **where**
subst-del σ x = $\sigma(x \mapsto_s \&x)$

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 *semi-uvar x*
shows $\langle\sigma(x \mapsto_s v)\rangle_s\ x = v$
using *assms*
by (*simp add*: *subst-upd-uvar-def*, *transfer*) (*simp*)

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

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

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

lemma *swap-usubst-inj*:
fixes *x y* :: ($'a$, $'\alpha$) *uvar*
assumes *uvar x uvar y* $x \bowtie y$
shows *inj* $[x \mapsto_s \&y, y \mapsto_s \&x]$
using *assms*
apply (*auto simp add*: *inj-on-def subst-upd-uvar-def*)
apply (*smt lens-indep-get lens-indep-sym var.rep-eq vwb-lens.put-eq vwb-lens-wb wb-lens-weak weak-lens.put-get*)
done

lemma *usubst-upd-var-id* [*usubst*]:
  $uvar\ x \Longrightarrow [x \mapsto_s var\ x] = id$
  apply (*simp add*: *subst-upd-uvar-def*)
  apply (*transfer*)
  apply (*rule ext*)
  apply (*auto*)
done

lemma *usubst-upd-comm-dash* [*usubst*]:
  fixes $x :: ('a, '\alpha)\ uvar$
  shows $\sigma(\$x\acute{} \mapsto_s v, \$x \mapsto_s u) = \sigma(\$x \mapsto_s u, \$x\acute{} \mapsto_s v)$
  using *in-out-indep usubst-upd-comm* **by** *force*

lemma *usubst-lookup-upd-indep* [*usubst*]:
  assumes *semi-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 *usubst-apply-unrest* [*usubst*]:
  $\llbracket\ uvar\ x;\ x \sharp \sigma\ \rrbracket \Longrightarrow \langle \sigma \rangle_s\ x = var\ x$
  by (*simp add*: *unrest-usubst-def*, *transfer*, *auto simp add*: *fun-eq-iff*, *metis vwb-lens-wb wb-lens.get-put wb-lens-weak weak-lens.put-get*)

lemma *subst-del-id* [*usubst*]:
  $uvar\ x \Longrightarrow id -_s x = id$
  by (*simp add*: *subst-del-def subst-upd-uvar-def*, *transfer*, *auto*)

lemma *subst-del-upd-same* [*usubst*]:
  $semi\text{-}uvar\ x \Longrightarrow \sigma(x \mapsto_s v) -_s x = \sigma -_s x$
  by (*simp add*: *subst-del-def subst-upd-uvar-def*)

lemma *subst-del-upd-diff* [*usubst*]:
  $x \bowtie y \Longrightarrow \sigma(y \mapsto_s v) -_s x = (\sigma -_s x)(y \mapsto_s v)$
  by (*simp add*: *subst-del-def subst-upd-uvar-def lens-indep-comm*)

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 *subst-compose-upd* [*usubst*]: $\llbracket\ uvar\ x;\ x \sharp \sigma\ \rrbracket \Longrightarrow \sigma \circ \varrho(x \mapsto_s v) = (\sigma \circ \varrho)(x \mapsto_s v)$
  by (*simp add*: *subst-upd-uvar-def*, *transfer*, *auto simp add*: *unrest-usubst-def*)

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 *unrest-usubst-del* [*unrest*]: $\llbracket\ uvar\ x;\ x \sharp (\langle \sigma \rangle_s\ x);\ x \sharp \sigma -_s x\ \rrbracket \Longrightarrow\ x \sharp (\sigma \dagger P)$
  by (*simp add*: *subst-del-def subst-upd-uvar-def unrest-upred-def unrest-usubst-def subst.rep-eq usubst-lookup.rep-eq*)
    (*metis vwb-lens.put-eq*)

We set up a purely syntactic order on variable lenses which is useful for the substitution normal form.

definition *var-name-ord* :: $('a, '\alpha)$ *uvar* $\Rightarrow$ $('b, '\alpha)$ *uvar* $\Rightarrow$ *bool* **where**
[*no-atp*]: *var-name-ord* x y = *True*

syntax
-var-name-ord :: *salpha* $\Rightarrow$ *salpha* $\Rightarrow$ *bool* (**infix** $\prec_v$ *65*)

translations
-var-name-ord x y == *CONST var-name-ord* x y

lemma *usubst-upd-comm-ord* [*usubst*]:
assumes $x \bowtie y$ $y \prec_v x$
shows $\sigma(x \mapsto_s u,\ y \mapsto_s v) = \sigma(y \mapsto_s v,\ x \mapsto_s u)$
by (*simp add*: *assms*(*1*) *usubst-upd-comm*)

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

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

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

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

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

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

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

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*)

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

syntax
-psubst :: [*logic*, *svars*, *uexprs*] ⇒ *logic*
-subst :: *logic* ⇒ *uexprs* ⇒ *salphas* ⇒ *logic* ((-⟦-'/-⟧) [*999*,*999*] *1000*)
-uexprs :: [*logic*, *uexprs*] => *uexprs* (-,/ -)
:: *logic* => *uexprs* (-)
-svars :: [*svar*, *svars*] => *svars* (-,/ -)
:: *svar* => *svars* (-)
-salphas :: [*salpha*, *salpha*] => *salphas* (-,/ -)
:: *salpha* => *salphas* (-)

translations
-subst P es vs => *CONST subst* (*-psubst* (*CONST id*) *vs es*) *P*
-psubst m (*-salphas x xs*) (*-uexprs v vs*) => *-psubst* (*-psubst m x v*) *xs vs*
-psubst m x v => *CONST subst-upd m x v*
P⟦*v*/$*x*⟧ <= *CONST usubst* (*CONST subst-upd* (*CONST id*) (*CONST ivar x*) *v*) *P*
P⟦*v*/$*x*´⟧ <= *CONST usubst* (*CONST subst-upd* (*CONST id*) (*CONST ovar x*) *v*) *P*

4.3 Unrestriction laws

lemma *unrest-usubst-single* [*unrest*]:
⟦ *semi-uvar x*; $x \sharp v$ ⟧ $\Longrightarrow x \sharp P⟦v/x⟧$
by (*transfer*, *auto simp add*: *subst-upd-uvar-def unrest-upred-def*)

lemma *unrest-usubst-id* [*unrest*]:
semi-uvar x $\Longrightarrow x \sharp id$
by (*simp add*: *unrest-usubst-def*)

lemma *unrest-usubst-upd* [*unrest*]:
⟦ $x \bowtie y$; $x \sharp \sigma$; $x \sharp v$ ⟧ $\Longrightarrow x \sharp \sigma(y \mapsto_s v)$
by (*simp add*: *subst-upd-uvar-def unrest-usubst-def unrest-upred.rep-eq lens-indep-comm*)

lemma *unrest-subst* [*unrest*]:
⟦ $x \sharp P$; $x \sharp \sigma$ ⟧ $\Longrightarrow x \sharp (\sigma \dagger P)$
by (*transfer*, *simp add*: *unrest-usubst-def*)

end

5 Alphabet manipulation

theory *utp-alphabet*
imports
utp-pred
begin

named-theorems *alpha*

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

5.1 Alphabet extension

Extend an alphabet by application of a lens that demonstrates how the smaller alphabet (β) injects into the larger alphabet (α).

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

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

lemma *aext-lit* [*alpha*]: $\ll v \gg \oplus_p a = \ll v \gg$
by (*pred-tac*)

lemma *aext-uop* [*alpha*]: $uop\ f\ u \oplus_p a = uop\ f\ (u \oplus_p a)$
by (*pred-tac*)

lemma *aext-bop* [*alpha*]: $bop\ f\ u\ v \oplus_p a = bop\ f\ (u \oplus_p a)\ (v \oplus_p a)$
by (*pred-tac*)

lemma *aext-trop* [*alpha*]: $trop\ f\ u\ v\ w \oplus_p a = trop\ f\ (u \oplus_p a)\ (v \oplus_p a)\ (w \oplus_p a)$
by (*pred-tac*)

lemma *aext-plus* [*alpha*]:
$(x + y) \oplus_p a = (x \oplus_p a) + (y \oplus_p a)$
by (*pred-tac*)

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

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

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

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

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

lemma *aext-true* [*alpha*]: $true \oplus_p a = true$
by (*pred-tac*)

lemma *aext-false* [*alpha*]: $false \oplus_p a = false$
by (*pred-tac*)

lemma *aext-not* [*alpha*]: $(\neg P) \oplus_p x = (\neg (P \oplus_p x))$

by (*pred-tac*)

lemma *aext-and* [*alpha*]: $(P \wedge Q) \oplus_p x = (P \oplus_p x \wedge Q \oplus_p x)$
by (*pred-tac*)

lemma *aext-or* [*alpha*]: $(P \vee Q) \oplus_p x = (P \oplus_p x \vee Q \oplus_p x)$
by (*pred-tac*)

lemma *aext-imp* [*alpha*]: $(P \Rightarrow Q) \oplus_p x = (P \oplus_p x \Rightarrow Q \oplus_p x)$
by (*pred-tac*)

lemma *aext-iff* [*alpha*]: $(P \Leftrightarrow Q) \oplus_p x = (P \oplus_p x \Leftrightarrow Q \oplus_p x)$
by (*pred-tac*)

lemma *unrest-aext* [*unrest*]:
$[\![$ *mwb-lens* a; $x \sharp p$ $]\!] \Longrightarrow$ *unrest* $(x ;_L a)$ $(p \oplus_p a)$
by (*transfer*, *simp add*: *lens-comp-def*)

lemma *unrest-aext-indep* [*unrest*]:
$a \bowtie b \Longrightarrow b \sharp (p \oplus_p a)$
by *pred-tac*

5.2 Alphabet restriction

Restrict an alphabet by application of a lens that demonstrates how the smaller alphabet (β) injects into the larger alphabet (α). Unlike extension, this operation can lose information if the expressions refers to variables in the larger alphabet.

lift-definition *arestr* :: $('a, '\alpha)$ *uexpr* $\Rightarrow$ $('\beta, '\alpha)$ *lens* $\Rightarrow$ $('a, '\beta)$ *uexpr* (**infixr** $\restriction_p$ *90*)
is λ P x b. P $(create_x\ b)$ **.**

lemma *arestr-id* [*alpha*]: $P \restriction_p 1_L = P$
by (*pred-tac*)

lemma *arestr-aext* [*alpha*]: *mwb-lens* $a \Longrightarrow (P \oplus_p a) \restriction_p a = P$
by (*pred-tac*)

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

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

lemma *arestr-lit* [*alpha*]: $\ll v \gg \restriction_p a = \ll v \gg$
by (*pred-tac*)

lemma *arestr-var* [*alpha*]:
  var $x \upharpoonright_p a$ = *var* $(x /_L a)$
  by (*pred-tac*)

lemma *arestr-true* [*alpha*]: *true* $\upharpoonright_p a$ = *true*
  by (*pred-tac*)

lemma *arestr-false* [*alpha*]: *false* $\upharpoonright_p a$ = *false*
  by (*pred-tac*)

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

lemma *arestr-and* [*alpha*]: $(P \wedge Q)\upharpoonright_p x = (P\upharpoonright_p x \wedge Q\upharpoonright_p x)$
  by (*pred-tac*)

lemma *arestr-or* [*alpha*]: $(P \vee Q)\upharpoonright_p x = (P\upharpoonright_p x \vee Q\upharpoonright_p x)$
  by (*pred-tac*)

lemma *arestr-imp* [*alpha*]: $(P \Rightarrow Q)\upharpoonright_p x = (P\upharpoonright_p x \Rightarrow Q\upharpoonright_p x)$
  by (*pred-tac*)

5.3 Alphabet lens laws

lemma *alpha-in-var* [*alpha*]: $x ;_L fst_L$ = *in-var* x
  by (*simp add*: *in-var-def*)

lemma *alpha-out-var* [*alpha*]: $x ;_L snd_L$ = *out-var* x
  by (*simp add*: *out-var-def*)

5.4 Alphabet coercion

definition *id-on* :: $('a \Longrightarrow '\alpha) \Rightarrow '\alpha \Rightarrow '\alpha$ **where**
[*upred-defs*]: *id-on* $x = (\lambda\ s.$ *undefined* $\oplus_L s$ *on* $x)$

definition *alpha-coerce* :: $('a \Longrightarrow '\alpha) \Rightarrow '\alpha$ *upred* $\Rightarrow '\alpha$ *upred*
where [*upred-defs*]: *alpha-coerce* $x\ P$ = *id-on* $x \dagger P$

syntax
  -alpha-coerce :: *salpha* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($!_\alpha$ - • - [*0*, *10*] *10*)

translations
  -alpha-coerce $P\ x$ == *CONST alpha-coerce* $P\ x$

5.5 Substitution alphabet extension

definition *subst-ext* :: $'\alpha$ *usubst* $\Rightarrow ('\alpha \Longrightarrow '\beta) \Rightarrow '\beta$ *usubst* (**infix** $\oplus_s$ *65*) **where**
[*upred-defs*]: $\sigma \oplus_s x = (\lambda\ s.\ put_x\ s\ (\sigma\ (get_x\ s)))$

lemma *id-subst-ext* [*usubst*,*alpha*]:
  uvar $x \Longrightarrow id \oplus_s x = id$
  by *pred-tac*

lemma *upd-subst-ext* [*alpha*]:
  uvar $x \Longrightarrow \sigma(y \mapsto_s v) \oplus_s x = (\sigma \oplus_s x)(\&x{:}y \mapsto_s v \oplus_p x)$
  by *pred-tac*

lemma *apply-subst-ext* [*alpha*]:
  $uvar\ x \Longrightarrow (\sigma \dagger e) \oplus_p x = (\sigma \oplus_s x) \dagger (e \oplus_p x)$
  by (*pred-tac*)

lemma *aext-upred-eq* [*alpha*]:
  $((e =_u f) \oplus_p a) = ((e \oplus_p a) =_u (f \oplus_p a))$
  by (*pred-tac*)

5.6 Substitution alphabet restriction

definition *subst-res* :: $'\alpha\ usubst \Rightarrow ('\beta \Longrightarrow '\alpha) \Rightarrow '\beta\ usubst$ (**infix** $\restriction_s$ *65*) **where**
[*upred-defs*]: $\sigma \restriction_s x = (\lambda\ s.\ get_x\ (\sigma\ (create_x\ s)))$

lemma *id-subst-res* [*alpha*,*usubst*]:
  $semi\text{-}uvar\ x \Longrightarrow id \restriction_s x = id$
  by *pred-tac*

lemma *upd-subst-res* [*alpha*]:
  $uvar\ x \Longrightarrow \sigma(\&x{:}y \mapsto_s v) \restriction_s x = (\sigma \restriction_s x)(\&y \mapsto_s v \restriction_p x)$
  by (*pred-tac*)

lemma *subst-ext-res* [*alpha*,*usubst*]:
  $uvar\ x \Longrightarrow (\sigma \oplus_s x) \restriction_s x = \sigma$
  by (*pred-tac*)

lemma *unrest-subst-alpha-ext* [*unrest*]:
  $x \bowtie y \Longrightarrow x \sharp (P \oplus_s y)$
  by (*pred-tac*, *auto simp add*: *unrest-usubst-def*, *metis lens-indep-def*)

end

6 Lifting expressions

theory *utp-lift*
  imports
    utp-alphabet
begin

6.1 Lifting definitions

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

abbreviation *lift-pre* :: $('a, '\alpha)\ uexpr \Rightarrow ('a, '\alpha \times '\beta)\ uexpr\ (\lceil\text{-}\rceil_<)$
where $\lceil P \rceil_< \equiv P \oplus_p fst_L$

abbreviation *drop-pre* :: $('\alpha \times '\alpha)\ upred \Rightarrow '\alpha\ upred\ (\lfloor\text{-}\rfloor_<)$
where $\lfloor P \rfloor_< \equiv P \restriction_p fst_L$

abbreviation *lift-post* :: $('a, '\beta)\ uexpr \Rightarrow ('a, '\alpha \times '\beta)\ uexpr\ (\lceil\text{-}\rceil_>)$
where $\lceil P \rceil_> \equiv P \oplus_p snd_L$

abbreviation *drop-post* :: $('\alpha \times '\alpha)\ upred \Rightarrow '\alpha\ upred\ (\lfloor\text{-}\rfloor_>)$
where $\lfloor P \rfloor_> \equiv P \restriction_p snd_L$

6.2 Lifting laws

lemma *lift-pre-var* [*simp*]:
  $\lceil var\ x \rceil_< = \$x$
  by (*alpha-tac*)

lemma *lift-post-var* [*simp*]:
  $\lceil var\ x \rceil_> = \$x'$
  by (*alpha-tac*)

6.3 Unrestriction laws

lemma *unrest-dash-var-pre* [*unrest*]:
  fixes $x :: ('a, '\alpha)\ uvar$
  shows $\$x' \sharp \lceil p \rceil_<$
  by (*pred-tac*)

end

7 Alphabetised Predicates

theory *utp-pred*
imports
  utp-expr
  utp-subst
begin

An alphabetised predicate is a simply a boolean valued expression

type-synonym $'\alpha\ upred = (bool, '\alpha)\ uexpr$

translations
  $(type)\ '\alpha\ upred <= (type)\ (bool, '\alpha)\ uexpr$

named-theorems *upred-defs*

7.1 Predicate syntax

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

no-notation
  conj (**infixr** $\wedge$ *35*) **and**
  disj (**infixr** $\vee$ *30*) **and**
  Not ($\neg$ - [*40*] *40*)

consts
  utrue :: $'a$ (*true*)
  ufalse :: $'a$ (*false*)
  uconj :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\wedge$ *35*)
  udisj :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\vee$ *30*)

$uimpl :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\Rightarrow$ *25*)
$uiff :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\Leftrightarrow$ *25*)
$unot :: 'a \Rightarrow 'a$ ($\neg$ - [*40*] *40*)
$uex :: ('a, '\alpha)\ uvar \Rightarrow 'p \Rightarrow 'p$
$uall :: ('a, '\alpha)\ uvar \Rightarrow 'p \Rightarrow 'p$
$ushEx :: ['a \Rightarrow 'p] \Rightarrow 'p$
$ushAll :: ['a \Rightarrow 'p] \Rightarrow 'p$

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
$\text{-}uex :: salpha \Rightarrow logic \Rightarrow logic$ ($\exists$ - $\cdot$ - [*0*, *10*] *10*)
$\text{-}uall :: salpha \Rightarrow logic \Rightarrow logic$ ($\forall$ - $\cdot$ - [*0*, *10*] *10*)
$\text{-}ushEx :: idt \Rightarrow logic \Rightarrow logic$ ($\exists$ - $\cdot$ - [*0*, *10*] *10*)
$\text{-}ushAll :: idt \Rightarrow logic \Rightarrow logic$ ($\forall$ - $\cdot$ - [*0*, *10*] *10*)
$\text{-}ushBEx :: idt \Rightarrow logic \Rightarrow logic \Rightarrow logic$ ($\exists$ - $\in$ - $\cdot$ - [*0*, *0*, *10*] *10*)
$\text{-}ushBAll :: idt \Rightarrow logic \Rightarrow logic \Rightarrow logic$ ($\forall$ - $\in$ - $\cdot$ - [*0*, *0*, *10*] *10*)
$\text{-}ushGAll :: idt \Rightarrow logic \Rightarrow logic \Rightarrow logic$ ($\forall$ - | - $\cdot$ - [*0*, *0*, *10*] *10*)

translations
$\text{-}uex\ x\ P$ == $CONST\ uex\ x\ P$
$\text{-}uall\ x\ P$ == $CONST\ uall\ x\ P$
$\exists\ x \cdot P$ == $CONST\ ushEx\ (\lambda\ x.\ P)$
$\exists\ x \in A \cdot P$ => $\exists\ x \cdot \ll x \gg \in_u A \wedge P$
$\forall\ x \cdot P$ == $CONST\ ushAll\ (\lambda\ x.\ P)$
$\forall\ x \in A \cdot P$ => $\forall\ x \cdot \ll x \gg \in_u A \Rightarrow P$
$\forall\ x\ |\ P \cdot Q$ => $\forall\ x \cdot P \Rightarrow Q$

7.2 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**
$P \sqsubseteq Q \equiv less\text{-}eq\ Q\ P$

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.

no-notation *inf* (**infixl** $\sqcap$ *70*)
notation *inf* (**infixl** $\sqcup$ *70*)
no-notation *sup* (**infixl** $\sqcup$ *65*)

notation *sup* (**infixl** ⊓ *65*)

no-notation *Inf* (⨅- [*900*] *900*)
notation *Inf* (⨆- [*900*] *900*)
no-notation *Sup* (⨆- [*900*] *900*)
notation *Sup* (⨅- [*900*] *900*)

no-notation *bot* (⊥)
notation *bot* (⊤)
no-notation *top* (⊤)
notation *top* (⊥)

no-syntax
  -INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨅-./ -) [*0*, *10*] *10*)
  -INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨅-∈-./ -) [*0*, *0*, *10*] *10*)
  -SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨆-./ -) [*0*, *10*] *10*)
  -SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨆-∈-./ -) [*0*, *0*, *10*] *10*)

syntax
  -INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨆-./ -) [*0*, *10*] *10*)
  -INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨆-∈-./ -) [*0*, *0*, *10*] *10*)
  -SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨅-./ -) [*0*, *10*] *10*)
  -SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨅-∈-./ -) [*0*, *0*, *10*] *10*)

We 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 *λP Q A. sup* (*P A*) (*Q A*) **.**
  lift-definition *inf-uexpr* :: (*'a*, *'b*) *uexpr* ⇒ (*'a*, *'b*) *uexpr* ⇒ (*'a*, *'b*) *uexpr*
  is *λP 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** *λ A. bot* **.**
  lift-definition *top-uexpr* :: (*'a*, *'b*) *uexpr* **is** *λ 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 *λ PS A. INF P*:*PS*. *P*(*A*) **.**

lift-definition *Sup-uexpr* :: $('a, 'b)$ *uexpr set* $\Rightarrow$ $('a, 'b)$ *uexpr*
is λ *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* :: $'\alpha$ *upred*)
definition *false-upred* = (*bot* :: $'\alpha$ *upred*)
definition *conj-upred* = (*inf* :: $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred*)
definition *disj-upred* = (*sup* :: $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred*)
definition *not-upred* = (*uminus* :: $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred*)
definition *diff-upred* = (*minus* :: $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred*)

lift-definition *USUP* :: $('a \Rightarrow '\alpha$ *upred*$)$ $\Rightarrow$ $('a \Rightarrow ('b$::*complete-lattice*, $'\alpha)$ *uexpr*$)$ $\Rightarrow$ $('b, '\alpha)$ *uexpr*
is λ *P F b. Sup* $\{[\![F\ x]\!]_e b \mid x.\ [\![P\ x]\!]_e b\}$ **.**

lift-definition *UINF* :: $('a \Rightarrow '\alpha$ *upred*$)$ $\Rightarrow$ $('a \Rightarrow ('b$::*complete-lattice*, $'\alpha)$ *uexpr*$)$ $\Rightarrow$ $('b, '\alpha)$ *uexpr*
is λ *P F b. Inf* $\{[\![F\ x]\!]_e b \mid x.\ [\![P\ x]\!]_e b\}$ **.**

declare *USUP-def* [*upred-defs*]
declare *UINF-def* [*upred-defs*]

syntax
-*USup* :: *idt* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigsqcap$ - $\cdot$ - [*0*, *10*] *10*)
-*USup-mem* :: *idt* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigsqcap$ - $\in$ - $\cdot$ - [*0*, *10*] *10*)
-*USUP* :: *idt* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigsqcap$ - $\mid$ - $\cdot$ - [*0*, *0*, *10*] *10*)
-*UInf* :: *idt* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigsqcup$ - $\cdot$ - [*0*, *10*] *10*)
-*UInf-mem* :: *idt* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigsqcup$ - $\in$ - $\cdot$ - [*0*, *10*] *10*)
-*UINF* :: *idt* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigsqcup$ - $\mid$ - $\cdot$ - [*0*, *10*] *10*)

translations
$\bigsqcap\ x \mid P \cdot F$ => *CONST USUP* $(\lambda\ x.\ P)$ $(\lambda\ x.\ F)$
$\bigsqcap\ x \cdot F$ == $\bigsqcap\ x \mid$ *true* $\cdot\ F$
$\bigsqcap\ x \cdot F$ == $\bigsqcap\ x \mid$ *true* $\cdot\ F$
$\bigsqcap\ x \in A \cdot F$ => $\bigsqcap\ x \mid \ll x \gg \in_u \ll A \gg \cdot F$
$\bigsqcap\ x \mid P \cdot F$ <= *CONST USUP* $(\lambda\ x.\ P)$ $(\lambda\ y.\ F)$
$\bigsqcup\ x \mid P \cdot F$ => *CONST UINF* $(\lambda\ x.\ P)$ $(\lambda\ x.\ F)$
$\bigsqcup\ x \cdot F$ == $\bigsqcup\ x \mid$ *true* $\cdot\ F$
$\bigsqcup\ x \in A \cdot F$ => $\bigsqcup\ x \mid \ll x \gg \in_u \ll A \gg \cdot F$
$\bigsqcup\ x \mid P \cdot F$ <= *CONST UINF* $(\lambda\ x.\ P)$ $(\lambda\ y.\ F)$

We also define the other predicate operators

lift-definition *impl*::$'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* **is**
λ *P Q A. P A* $\longrightarrow$ *Q A* **.**

lift-definition *iff-upred* ::$'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* **is**
λ *P Q A. P A* $\longleftrightarrow$ *Q A* **.**

lift-definition *ex* :: $('a, '\alpha)$ *uvar* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* **is**
λ *x P b.* $(\exists\ v.\ P(put_x\ b\ v))$ **.**

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**
$\lambda\ x\ P\ b.\ (\forall\ v.\ P(put_x\ b\ v))$.

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* $([-]_u)$ **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

syntax
-uneq :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infixl** $\neq_u$ *50*)
-unmem :: $('a, '\alpha)$ *uexpr* $\Rightarrow$ $('a\ set, '\alpha)$ *uexpr* $\Rightarrow$ $(bool, '\alpha)$ *uexpr* (**infix** $\notin_u$ *50*)

translations
$x \neq_u y == CONST\ unot\ (x =_u y)$
$x \notin_u A == CONST\ unot\ (CONST\ bop\ (op \in)\ x\ A)$

7.3 Proof support

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

method *pred-tac* = ((*simp only*: *upred-defs*)? ; (*transfer*, (*rule-tac ext*)?, *auto simp add*: *lens-defs fun-eq-iff prod.case-eq-if*)?)

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*]

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

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

7.4 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*]: $[\![x \sharp (P :: '\alpha\ upred); x \sharp Q]\!] \Longrightarrow x \sharp P \wedge Q$
by (*pred-tac*)

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

lemma *unrest-USUP* [*unrest*]:
$[\![(\bigwedge i.\ x \sharp P(i)); (\bigwedge i.\ x \sharp Q(i))]\!] \Longrightarrow x \sharp (\bigsqcap\ i \mid P(i) \cdot Q(i))$
by (*simp add*: *USUP-def*, *pred-tac*)

lemma *unrest-UINF* [*unrest*]:
$[\![(\bigwedge i.\ x \sharp P(i)); (\bigwedge i.\ x \sharp Q(i))]\!] \Longrightarrow x \sharp (\bigsqcup\ i \mid P(i) \cdot Q(i))$
by (*simp add*: *UINF-def*, *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*]: $x \sharp (P :: '\alpha\ upred) \Longrightarrow x \sharp (\neg P)$
by (*pred-tac*)

The sublens proviso can be thought of as membership below.

lemma *unrest-ex-in* [*unrest*]:
$[\![semi\text{-}uvar\ y; x \subseteq_L y]\!] \Longrightarrow x \sharp (\exists\ y \cdot P)$
by (*pred-tac*)

declare *sublens-refl* [*simp*]
declare *lens-plus-ub* [*simp*]
declare *lens-plus-right-sublens* [*simp*]
declare *comp-wb-lens* [*simp*]
declare *comp-mwb-lens* [*simp*]
declare *plus-mwb-lens* [*simp*]

lemma *unrest-ex-diff* [*unrest*]:
assumes $x \bowtie y\ y \sharp P$
shows $y \sharp (\exists\ x \cdot P)$
using *assms*
apply (*pred-tac*)

using *lens-indep-comm* **apply** *fastforce+*
done

lemma *unrest-all-in* [*unrest*]:
$\llbracket$ *semi-uvar* y; $x \subseteq_L y$ $\rrbracket \Longrightarrow x \sharp (\forall\ y \cdot P)$
by *pred-tac*

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

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*

7.5 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-sup* [*usubst*]: $\sigma \dagger (P \sqcap Q) = (\sigma \dagger P \sqcap \sigma \dagger Q)$
by (*pred-tac*)

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

lemma *subst-USUP* [*usubst*]: $\sigma \dagger (\bigsqcap\ i \mid P(i) \cdot Q(i)) = (\bigsqcap\ i \mid (\sigma \dagger P(i)) \cdot (\sigma \dagger Q(i)))$
by (*simp add*: *USUP-def*, *pred-tac*)

lemma *subst-UINF* [*usubst*]: $\sigma \dagger (\bigsqcup\ i \mid P(i) \cdot Q(i)) = (\bigsqcup\ i \mid (\sigma \dagger P(i)) \cdot (\sigma \dagger Q(i)))$
by (*simp add*: *UINF-def*, *pred-tac*)

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

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

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

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

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

lemma *subst-ex-indep* [*usubst*]:
assumes $x \bowtie y$ $y \sharp v$
shows $(\exists\ y \cdot P)[\![v/x]\!] = (\exists\ y \cdot P[\![v/x]\!])$
using *assms*
apply (*pred-tac*)
using *lens-indep-comm* **apply** *fastforce+*
done

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

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

7.6 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*: $P \sqsubseteq Q = `Q \Rightarrow P`$
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+*

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 *USUP-cong-eq*:
$[\![\bigwedge x.\ P_1(x) = P_2(x);\ \bigwedge x.\ `P_1(x) \Rightarrow Q_1(x) =_u Q_2(x)`]\!] \Longrightarrow$
$(\bigsqcap\ x \mid P_1(x) \cdot Q_1(x)) = (\bigsqcap\ x \mid P_2(x) \cdot Q_2(x))$
by (*simp add*: *USUP-def*, *pred-tac*, *metis*)

lemma *USUP-as-Sup*: $(\bigsqcap\ P \in \mathcal{P} \cdot P) = \bigsqcap\ \mathcal{P}$
apply (*simp add*: *upred-defs bop.rep-eq lit.rep-eq Sup-uexpr-def*)
apply (*pred-tac*)
apply (*unfold SUP-def*)
apply (*rule cong*[*of Sup*])
apply (*auto*)
done

lemma *USUP-as-Sup-collect*: $(\bigsqcap P{\in}A \cdot f(P)) = (\bigsqcap P{\in}A.\ f(P))$
apply (*simp add*: *upred-defs bop.rep-eq lit.rep-eq Sup-uexpr-def*)
apply (*unfold SUP-def*)
apply (*pred-tac*)
apply (*simp add*: *Setcompr-eq-image*)
done

lemma *USUP-as-Sup-image*: $(\bigsqcap\ P \mid \ll P \gg \in_u \ll A \gg \cdot f(P)) = \bigsqcap\ (f\ `\ A)$
apply (*simp add*: *upred-defs bop.rep-eq lit.rep-eq Sup-uexpr-def*)
apply (*pred-tac*)
apply (*unfold SUP-def*)
apply (*rule cong*[*of Sup*])
apply (*auto*)

done

lemma *UINF-as-Inf*: $(\bigsqcup\ P \in \mathcal{P} \cdot P) = \bigsqcup\ \mathcal{P}$
apply (*simp add*: *upred-defs bop.rep-eq lit.rep-eq Inf-uexpr-def*)
apply (*pred-tac*)
apply (*unfold INF-def*)
apply (*rule cong*[*of Inf*])
apply (*auto*)
done

lemma *UINF-as-Inf-collect*: $(\bigsqcup P{\in}A \cdot f(P)) = (\bigsqcup P{\in}A.\ f(P))$
apply (*simp add*: *upred-defs bop.rep-eq lit.rep-eq Sup-uexpr-def*)
apply (*unfold INF-def*)
apply (*pred-tac*)
apply (*simp add*: *Setcompr-eq-image*)
done

lemma *UINF-as-Inf-image*: $(\bigsqcup\ P \in \mathcal{P} \cdot f(P)) = \bigsqcup\ (f\ \text{‘}\ \mathcal{P})$
apply (*simp add*: *upred-defs bop.rep-eq lit.rep-eq Inf-uexpr-def*)
apply (*pred-tac*)
apply (*unfold INF-def*)
apply (*rule cong*[*of Inf*])
apply (*auto*)
done

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

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

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

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

lemma *eq-cong-left*:
assumes *uvar x* $\$x \sharp Q\ \$x' \sharp Q\ \$x \sharp R\ \$x' \sharp R$
shows $((\$x' =_u \$x \wedge Q) = (\$x' =_u \$x \wedge R)) \longleftrightarrow (Q = R)$
using *assms*
by (*pred-tac*, (*meson mwb-lens-def vwb-lens-mwb weak-lens-def*)+)

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

lemma *conj-eq-out-var-subst*:
fixes $x :: ('a, '\alpha)$ *uvar*
assumes *uvar x*
shows $(P \wedge \$x' =_u v) = (P[\![v/\$x']\!] \wedge \$x' =_u v)$
using *assms*

by (*pred-tac*, (*metis vwb-lens-wb wb-lens.get-put*)+)

lemma *conj-pos-var-subst*:
assumes *uvar* x
shows $(\$x \wedge Q) = (\$x \wedge Q[\![true/\$x]\!])$
using *assms*
by (*pred-tac*, *metis* (*full-types*) *vwb-lens-wb wb-lens.get-put*, *metis* (*full-types*) *vwb-lens-wb wb-lens.get-put*)

lemma *conj-neg-var-subst*:
assumes *uvar* x
shows $(\neg \$x \wedge Q) = (\neg \$x \wedge Q[\![false/\$x]\!])$
using *assms*
by (*pred-tac*, *metis* (*full-types*) *vwb-lens-wb wb-lens.get-put*, *metis* (*full-types*) *vwb-lens-wb wb-lens.get-put*)

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

lemma *shEx-unbound* [*simp*]: $(\exists\ x \cdot P) = P$
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 *conj-var-subst*:
assumes *uvar* x
shows $(P \wedge var\ x =_u v) = (P[\![v/x]\!] \wedge var\ x =_u v)$
using *assms*
by (*pred-tac*, (*metis* (*full-types*) *vwb-lens-def wb-lens.get-put*)+)

lemma *one-point*:
assumes *semi-uvar* x $x \sharp v$
shows $(\exists\ x \cdot P \wedge \&x =_u v) = P[\![v/x]\!]$
using *assms*
by (*pred-tac*)

lemma *uvar-assign-exists*:
uvar $x \Longrightarrow \exists\ v.\ b = put_x\ b\ v$
by (*rule-tac* x=get_x b **in** *exI*, *simp*)

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

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

lemma *eq-split-substI*:
assumes *uvar x* $\bigwedge v.\ P[\![\ll v \gg/x]\!] = Q[\![\ll v \gg/x]\!]$
shows $P = Q$
using *assms*(*1*) *assms*(*2*) *eq-split-subst* **by** *blast*

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 $`P` = `(P[\![false/x]\!] \wedge P[\![true/x]\!])`$
proof −
from *assms* **have** $`P` = (\forall\ v.\ `P[\![\ll v \gg/x]\!]`)$
by (*subst taut-split-subst*[*of x*], *auto*)
also have ... $= (`P[\![\ll True \gg/x]\!]` \wedge `P[\![\ll False \gg/x]\!]`)$
by (*metis* (*mono-tags*, *lifting*))
also have ... $= `(P[\![false/x]\!] \wedge P[\![true/x]\!])`$
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 $x :: ('a, '\alpha)$ *uvar*
shows $(p[\![u/x]\!] \wedge u =_u v) = (p[\![v/x]\!] \wedge u =_u v)$
by *pred-tac*

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

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

lemma *exists-sub*: $[\![$ *mwb-lens y*; $x \subseteq_L y\ ]\!] \Longrightarrow (\exists\ x \cdot \exists\ y \cdot P) = (\exists\ y \cdot P)$
by *pred-tac*

lemma *all-sub*: $\llbracket$ *mwb-lens* y; $x \subseteq_L y$ $\rrbracket \Longrightarrow (\forall\ x \cdot \forall\ y \cdot P) = (\forall\ y \cdot P)$
  by *pred-tac*

lemma *ex-commute*:
  assumes $x \bowtie y$
  shows $(\exists\ x \cdot \exists\ y \cdot P) = (\exists\ y \cdot \exists\ x \cdot P)$
  using *assms*
  apply (*pred-tac*)
  using *lens-indep-comm* **apply** *fastforce+*
done

lemma *all-commute*:
  assumes $x \bowtie y$
  shows $(\forall\ x \cdot \forall\ y \cdot P) = (\forall\ y \cdot \forall\ x \cdot P)$
  using *assms*
  apply (*pred-tac*)
  using *lens-indep-comm* **apply** *fastforce+*
done

lemma *ex-equiv*:
  assumes $x \approx_L y$
  shows $(\exists\ x \cdot P) = (\exists\ y \cdot P)$
  using *assms*
  by (*pred-tac*, *metis* (*no-types*, *lifting*) *lens.select-convs*(*2*))

lemma *all-equiv*:
  assumes $x \approx_L y$
  shows $(\forall\ x \cdot P) = (\forall\ y \cdot P)$
  using *assms*
  by (*pred-tac*, *metis* (*no-types*, *lifting*) *lens.select-convs*(*2*))

lemma *ex-zero*:
  $(\exists\ \&\emptyset \cdot P) = P$
  by *pred-tac*

lemma *all-zero*:
  $(\forall\ \&\emptyset \cdot P) = P$
  by *pred-tac*

lemma *ex-plus*:
  $(\exists\ y{,}x \cdot P) = (\exists\ x \cdot \exists\ y \cdot P)$
  by *pred-tac*

lemma *all-plus*:
  $(\forall\ y{,}x \cdot P) = (\forall\ x \cdot \forall\ y \cdot P)$
  by *pred-tac*

lemma *closure-all*:
  $[P]_u = (\forall\ \&\Sigma \cdot P)$
  by *pred-tac*

lemma *unrest-as-exists*:
  vwb-lens $x \Longrightarrow (x \sharp P) \longleftrightarrow ((\exists\ x \cdot P) = P)$
  by (*pred-tac*, *metis vwb-lens.put-eq*)

7.7 Cylindric algebra

lemma *C1*: $(\exists\ x \cdot false) = false$
by $(pred\text{-}tac)$

lemma *C2*: $wb\text{-}lens\ x \Longrightarrow `P \Rightarrow (\exists\ x \cdot P)`$
by $(pred\text{-}tac,\ metis\ wb\text{-}lens.get\text{-}put)$

lemma *C3*: $mwb\text{-}lens\ x \Longrightarrow (\exists\ x \cdot (P \wedge (\exists\ x \cdot Q))) = ((\exists\ x \cdot P) \wedge (\exists\ x \cdot Q))$
by $(pred\text{-}tac)$

lemma *C4a*: $x \approx_L y \Longrightarrow (\exists\ x \cdot \exists\ y \cdot P) = (\exists\ y \cdot \exists\ x \cdot P)$
by $(pred\text{-}tac,\ metis\ (no\text{-}types,\ lifting)\ lens.select\text{-}convs(2))+$

lemma *C4b*: $x \bowtie y \Longrightarrow (\exists\ x \cdot \exists\ y \cdot P) = (\exists\ y \cdot \exists\ x \cdot P)$
using *ex-commute* **by** *blast*

lemma *C5*:
fixes $x :: ('a,\ '\alpha)\ uvar$
shows $(\&x =_u \&x) = true$
by *pred-tac*

lemma *C6*:
assumes $wb\text{-}lens\ x\ x \bowtie y\ x \bowtie z$
shows $(\&y =_u \&z) = (\exists\ x \cdot \&y =_u \&x \wedge \&x =_u \&z)$
using *assms*
by $(pred\text{-}tac,\ (metis\ lens\text{-}indep\text{-}def)+)$

lemma *C7*:
assumes $weak\text{-}lens\ x\ x \bowtie y$
shows $((\exists\ x \cdot \&x =_u \&y \wedge P) \wedge (\exists\ x \cdot \&x =_u \&y \wedge \neg\ P)) = false$
using *assms*
by $(pred\text{-}tac,\ simp\ add{:}\ lens\text{-}indep\text{-}sym)$

7.8 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

8 Alphabetised relations

theory *utp-rel*
imports
utp-pred
utp-lift
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**
$in\alpha = (\!|\ lens\text{-}get = fst,\ lens\text{-}put = \lambda\ (A,\ A')\ v.\ (v,\ A')\ |\!)$

definition $out\alpha$ :: $('\beta,\ '\alpha \times '\beta)$ *uvar* **where**
$out\alpha = (\!|\ lens\text{-}get = snd,\ lens\text{-}put = \lambda\ (A,\ A')\ v.\ (A,\ v)\ |\!)$

declare *inα-def* [*urel-defs*]
declare *outα-def* [*urel-defs*]

The alphabet of a relation consists of the input and output portions

lemma *alpha-in-out*:
  $\Sigma \approx_L in\alpha +_L out\alpha$
  by (*metis fst-lens-def fst-snd-id-lens inα-def lens-equiv-refl outα-def snd-lens-def*)

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*- $\triangleleft$ - $\triangleright$/ -) [*14*,*0*,*15*] *14*)
where $(P \triangleleft b \triangleright Q) \equiv (b \wedge P) \vee ((\neg\ b) \wedge Q)$

abbreviation *rcond*::$('\alpha,\ '\beta)$ *relation* $\Rightarrow$ $'\alpha$ *condition* $\Rightarrow$ $('\alpha,\ '\beta)$ *relation* $\Rightarrow$ $('\alpha,\ '\beta)$ *relation*
  ((*3*- $\triangleleft$ - $\triangleright_r$ / -) [*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 $\lambda\ P\ Q\ r.\ r \in (\{p.\ P\ p\}\ O\ \{q.\ Q\ q\})$ **.**

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

definition *skip-ra* :: $('\beta,\ '\alpha)$ *lens* $\Rightarrow$$'\alpha$ *hrelation* **where**
[*urel-defs*]: *skip-ra* $v = (\$v\acute{} =_u \$v)$

syntax
  -skip-ra :: *salpha* $\Rightarrow$ *logic* ($II_{\text{-}}$)

translations
  -skip-ra v == *CONST skip-ra v*

abbreviation *usubst-rel-lift* :: $'\alpha$ *usubst* $\Rightarrow$ $('\alpha \times '\beta)$ *usubst* $(\lceil\text{-}\rceil_s)$ **where**
$\lceil \sigma \rceil_s \equiv \sigma \oplus_s in\alpha$

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

definition *assigns-ra* :: $'\alpha$ *usubst* ⇒ ($'\beta$, $'\alpha$) *lens* ⇒ $'\alpha$ *hrelation* (⟨-⟩-) **where**
$\langle\sigma\rangle_a = (\lceil\sigma\rceil_s \dagger II_a)$

lift-definition *assigns-r* :: $'\alpha$ *usubst* ⇒ $'\alpha$ *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* ⇒ ($'t$, $'\alpha$) *uexpr* ⇒ $'\alpha$ *hrelation*
where *assign-r x v* ≡ *assigns-r* [$x \mapsto_s v$]

abbreviation *assign-2-r* ::
($'t1$, $'\alpha$) *uvar* ⇒ ($'t2$, $'\alpha$) *uvar* ⇒ ($'t1$, $'\alpha$) *uexpr* ⇒ ($'t2$, $'\alpha$) *uexpr* ⇒ $'\alpha$ *hrelation*
where *assign-2-r x y u v* ≡ *assigns-r* [$x \mapsto_s u$, $y \mapsto_s v$]

nonterminal
svid-list **and** *uexpr-list*

syntax
-svid-unit :: *svid* ⇒ *svid-list* (-)
-svid-list :: *svid* ⇒ *svid-list* ⇒ *svid-list* (-,/ -)
-uexpr-unit :: ($'a$, $'\alpha$) *uexpr* ⇒ *uexpr-list* (- [*40*] *40*)
-uexpr-list :: ($'a$, $'\alpha$) *uexpr* ⇒ *uexpr-list* ⇒ *uexpr-list* (-,/ - [*40*,*40*] *40*)
-assignment :: *svid-list* ⇒ *uexprs* ⇒ $'\alpha$ *hrelation* (**infixr** := *55*)
-mk-usubst :: *svid-list* ⇒ *uexprs* ⇒ $'\alpha$ *usubst*

translations
-mk-usubst σ (*-svid-unit x*) *v* == σ(&*x* $\mapsto_s$ *v*)
-mk-usubst σ (*-svid-list x xs*) (*-uexprs v vs*) == (*-mk-usubst* (σ(&*x* $\mapsto_s$ *v*)) *xs vs*)
-assignment xs vs => *CONST assigns-r* (*-mk-usubst* (*CONST id*) *xs vs*)
x := *v* <= *CONST assigns-r* (*CONST subst-upd* (*CONST id*) (*CONST svar x*) *v*)
x,*y* := *u*,*v* <= *CONST assigns-r* (*CONST subst-upd* (*CONST subst-upd* (*CONST id*) (*CONST svar x*) *u*) (*CONST svar y*) *v*)

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

method *rel-simp* = ((*simp add*: *upred-defs urel-defs*)?, (*transfer*, (*rule-tac ext*)?, *simp-all add*: *lens-defs urel-defs relcomp-unfold fun-eq-iff prod.case-eq-if*)?)
method *rel-tac* = ((*simp add*: *upred-defs urel-defs*)?, (*transfer*, (*rule-tac ext*)?, *auto simp add*: *lens-defs urel-defs relcomp-unfold fun-eq-iff prod.case-eq-if*)?)

We describe some properties of relations

definition *ufunctional* :: ($'a$, $'b$) *relation* ⇒ *bool*
where *ufunctional R* ⟷ ($II \sqsubseteq (R^- \ ;;\ R)$)

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

definition *uinj* :: ($'a$, $'b$) *relation* ⇒ *bool*
where *uinj R* ⟷ $II \sqsubseteq (R\ ;;\ R^-)$

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

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* :: $'\alpha$ *condition* $\Rightarrow$ $'\alpha$ *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*]

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

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

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

8.1 Unrestriction Laws

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

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

lemma *unrest-semir-undash* [*unrest*]:
  fixes x :: $('a, '\alpha)$ *uvar*
  assumes $\$x \sharp P$
  shows $\$x \sharp (P ;; Q)$
  using *assms* **by** (*rel-tac*)

lemma *unrest-semir-dash* [*unrest*]:
  fixes x :: $('a, '\alpha)$ *uvar*
  assumes $\$x' \sharp Q$
  shows $\$x' \sharp (P ;; Q)$
  using *assms* **by** (*rel-tac*)

lemma *unrest-cond* [*unrest*]:
  $[\![x \sharp P; x \sharp b; x \sharp Q]\!] \Longrightarrow x \sharp (P \triangleleft b \triangleright Q)$
  by (*rel-tac*)

lemma *unrest-inα-var* [*unrest*]:
  $[\![$ *semi-uvar* x; *in*$\alpha \sharp (P :: ('\alpha, '\beta)$ *relation*$)]\!] \Longrightarrow \$x \sharp P$
  by (*pred-tac*, *simp add*: *inα-def*, *blast*, *metis inα-def lens.select-convs*(*2*) *old.prod.case*)

lemma *unrest-outα-var* [*unrest*]:
  $[\![$ *semi-uvar* x; *out*$\alpha \sharp (P :: ('\alpha, '\beta)$ *relation*$)]\!] \Longrightarrow \$x' \sharp P$
  by (*pred-tac*, *simp add*: *outα-def*, *blast*, *metis lens.select-convs*(*2*) *old.prod.case outα-def*)

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

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

lemma *unrest-pre-outα* [*unrest*]: *out*$\alpha \sharp \lceil b \rceil_<$
  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*)

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

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

8.2 Substitution laws

It should be possible to substantially generalise the following two laws

lemma *usubst-seq-left* [*usubst*]:
$[\![semi\text{-}uvar\ x;\ out\alpha \sharp v]\!] \Longrightarrow (P\ ;;\ Q)[\![v/\$x]\!] = ((P[\![v/\$x]\!])\ ;;\ Q)$
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=ya* **in** *spec*)
apply (*simp*)
apply (*rename-tac x v P Q a ba y*)
apply (*rule-tac x=y* **in** *exI*)
apply (*drule-tac x=a* **in** *spec*)
apply (*drule-tac x=y* **in** *spec*)
apply (*drule-tac x=ba* **in** *spec*)
apply (*simp*)
done

lemma *usubst-seq-right* [*usubst*]:
$[\![semi\text{-}uvar\ x;\ in\alpha \sharp v]\!] \Longrightarrow (P\ ;;\ Q)[\![v/\$x']\!] = (P\ ;;\ Q[\![v/\$x']\!])$
by (*rel-tac*, *metis+*)

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

lemma *subst-skip-r* [*usubst*]:
fixes x :: ($'a$, $'\alpha$) *uvar*
shows $II[\![\lceil v \rceil_< / \$x]\!] = (x := v)$
by (*rel-tac*)

lemma *usubst-upd-in-comp* [*usubst*]:
$\sigma(\&in\alpha{:}x \mapsto_s v) = \sigma(\$x \mapsto_s v)$
by (*simp add*: *fst-lens-def inα-def in-var-def*)

lemma *usubst-upd-out-comp* [*usubst*]:
$\sigma(\&out\alpha{:}x \mapsto_s v) = \sigma(\$x' \mapsto_s v)$
by (*simp add*: *outα-def out-var-def snd-lens-def*)

lemma *subst-lift-upd* [*usubst*]:
fixes x :: ($'a$, $'\alpha$) *uvar*
shows $\lceil \sigma(x \mapsto_s v) \rceil_s = \lceil \sigma \rceil_s(\$x \mapsto_s \lceil v \rceil_<)$
by (*simp add*: *alpha usubst*, *simp add*: *fst-lens-def inα-def in-var-def*)

lemma *subst-lift-pre* [*usubst*]: $\lceil \sigma \rceil_s \dagger \lceil b \rceil_< = \lceil \sigma \dagger b \rceil_<$
by (*metis apply-subst-ext fst-lens-def fst-vwb-lens inα-def*)

lemma *unrest-usubst-lift-in* [*unrest*]:
$x \sharp P \Longrightarrow \$x \sharp \lceil P \rceil_s$
by (*pred-tac*, *auto simp add*: *unrest-usubst-def inα-def*)

lemma *unrest-usubst-lift-out* [*unrest*]:
fixes x :: ($'a$, $'\alpha$) *uvar*
shows $\$x' \sharp \lceil P \rceil_s$
by (*pred-tac*, *auto simp add*: *unrest-usubst-def inα-def*)

8.3 Relation laws

Homogeneous relations form a quantale

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

abbreviation *falser* :: $'\alpha$ *hrelation* ($false_h$) **where**
falser ≡ *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*]: $[\![out\alpha \sharp p]\!] \Longrightarrow \lceil \lfloor p \rfloor_< \rceil_< = p$

by (*pred-tac*, *auto simp add*: *outα-def lens-create-def fst-lens-def prod.case-eq-if*)

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

definition *while* :: *'α condition* ⇒ *'α hrelation* ⇒ *'α 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*]

While loops with invariant decoration

definition *while-inv* :: *'α condition* ⇒ *'α condition* ⇒ *'α hrelation* ⇒ *'α hrelation* (*while* - *invr* - *do* - *od*) **where**
while b invr p do S od = *while b do S od*

declare *while-inv-def*

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-and-T-integrate*:
$((P \wedge b) \vee (Q \triangleleft b \triangleright R)) = ((P \vee Q) \triangleleft b \triangleright R)$
by (*rel-tac*)

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

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

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

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

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

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

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

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

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

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

by *rel-tac*

lemma *cond-var-subst-left*:
assumes *uvar* x
shows $(P \triangleleft \$x \triangleright Q) = (P[\![true/\$x]\!] \triangleleft \$x \triangleright Q)$
using *assms* **by** (*metis cond-def conj-pos-var-subst*)

lemma *cond-var-subst-right*:
assumes *uvar* x
shows $(P \triangleleft \$x \triangleright Q) = (P \triangleleft \$x \triangleright Q[\![false/\$x]\!])$
using *assms* **by** (*metis cond-def conj-neg-var-subst*)

lemma *cond-seq-left-distr*:
$out\alpha \sharp b \Longrightarrow ((P \triangleleft b \triangleright Q) ;; R) = ((P ;; R) \triangleleft b \triangleright (Q ;; R))$
by (*rel-tac*, *blast+*)

lemma *cond-seq-right-distr*:
$in\alpha \sharp b \Longrightarrow (P ;; (Q \triangleleft b \triangleright R)) = ((P ;; Q) \triangleleft b \triangleright (P ;; R))$
by (*rel-tac*, *blast+*)

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

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

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

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

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

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

lemma *seqr-mono*:
$[\![P_1 \sqsubseteq P_2; Q_1 \sqsubseteq Q_2]\!] \Longrightarrow (P_1 ;; Q_1) \sqsubseteq (P_2 ;; Q_2)$
by (*rel-tac*, *blast*)

lemma *spec-refine*:
$Q \sqsubseteq (P \wedge R) \Longrightarrow (P \Rightarrow Q) \sqsubseteq R$
by (*rel-tac*)

lemma *cond-skip*: $out\alpha \sharp b \Longrightarrow (b \wedge II) = (II \wedge b^-)$
by (*rel-tac*)

lemma *pre-skip-post*: $(\lceil b \rceil_< \wedge II) = (II \wedge \lceil b \rceil_>)$
by (*rel-tac*)

lemma *skip-var*:
fixes x :: (*bool*, $'\alpha$) *uvar*
shows $(\$x \wedge II) = (II \wedge \$x')$
by (*rel-tac*)

lemma *seqr-exists-left*:
semi-uvar $x \Longrightarrow ((\exists\ \$x \cdot P)\ ;;\ Q) = (\exists\ \$x \cdot (P\ ;;\ Q))$
by (*rel-tac*)

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

lemma *assigns-subst* [*usubst*]:
$\lceil\sigma\rceil_s \dagger \langle\varrho\rangle_a = \langle\varrho \circ \sigma\rangle_a$
by (*rel-tac*)

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

lemma *assigns-r-feasible*:
$(\langle\sigma\rangle_a\ ;;\ true) = true$
by (*rel-tac*)

lemma *assign-subst* [*usubst*]:
$[\![$ *semi-uvar* x; *semi-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*: *semi-uvar* $x \Longrightarrow (x,x := u,v) = (x := v)$
by (*simp add*: *usubst*)

lemma *assigns-comp*: $(\langle f\rangle_a\ ;;\ \langle g\rangle_a) = \langle g \circ f\rangle_a$
by (*simp add*: *assigns-r-comp usubst*)

lemma *assigns-r-conv*:
bij $f \Longrightarrow \langle f\rangle_a{}^- = \langle inv\ f\rangle_a$
by (*rel-tac*, *simp-all add*: *bij-is-inj bij-is-surj surj-f-inv-f*)

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

lemma *assign-test*: *semi-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 *assign-twice*: $[\![$ *uvar* x; $x \sharp f$ $]\!] \Longrightarrow (x := e\ ;;\ x := f) = (x := f)$
by (*simp add*: *assigns-comp usubst*)

lemma *assign-commute*:
assumes $x \bowtie y$ $x \sharp f$ $y \sharp e$
shows $(x := e\ ;;\ y := f) = (y := f\ ;;\ x := e)$
using *assms*
by (*rel-tac*, *simp-all add*: *lens-indep-comm*)

lemma *assign-cond*:
fixes x :: ($'a$, $'\alpha$) *uvar*

assumes $out\alpha \sharp b$
shows $(x := e ;; (P \triangleleft b \triangleright Q)) = ((x := e ;; P) \triangleleft (b\llbracket\lceil e\rceil_{<}/\$x\rrbracket) \triangleright (x := e ;; Q))$
by *rel-tac*

lemma *assign-rcond*:
fixes $x :: ('a, '\alpha)$ *uvar*
shows $(x := e ;; (P \triangleleft b \triangleright_r Q)) = ((x := e ;; P) \triangleleft (b\llbracket e/x\rrbracket) \triangleright_r (x := e ;; Q))$
by *rel-tac*

lemma *assign-r-alt-def*:
fixes $x :: ('a, '\alpha)$ *uvar*
shows $x := v = II\llbracket\lceil v\rceil_{<}/\$x\rrbracket$
by *rel-tac*

lemma *assigns-r-ufunc*: *ufunctional* $\langle f\rangle_a$
by (*rel-tac*)

lemma *assigns-r-uinj*: *inj f* $\Longrightarrow$ *uinj* $\langle f\rangle_a$
by (*rel-tac*, *simp add*: *inj-eq*)

lemma *assigns-r-swap-uinj*:
$\llbracket$ *uvar x*; *uvar y*; $x \bowtie y$ $\rrbracket \Longrightarrow$ *uinj* $(x,y := \&y,\&x)$
using *assigns-r-uinj swap-usubst-inj* **by** *auto*

lemma *skip-r-unfold*:
uvar x $\Longrightarrow II = (\$x\acute{} =_u \$x \wedge II\restriction_\alpha x)$
by (*rel-tac*, *blast*, *metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens.get-put*)

lemma *skip-r-alpha-eq*:
$II = (\$\Sigma\acute{} =_u \$\Sigma)$
by (*rel-tac*)

lemma *skip-ra-unfold*:
$II_{x,y} = (\$x\acute{} =_u \$x \wedge II_y)$
by (*rel-tac*)

lemma *skip-res-as-ra*:
$\llbracket$ *vwb-lens y*; $x +_L y \approx_L 1_L$; $x \bowtie y$ $\rrbracket \Longrightarrow II\restriction_\alpha x = II_y$
apply (*rel-tac*)
apply (*metis* (*no-types*, *lifting*) *lens-indep-def*)
apply (*metis vwb-lens.put-eq*)
done

lemma *assign-unfold*:
uvar x $\Longrightarrow (x := v) = (\$x\acute{} =_u \lceil v\rceil_{<} \wedge II\restriction_\alpha x)$
apply (*rel-tac*, *auto simp add*: *comp-def*)
using *vwb-lens.put-eq* **by** *fastforce*

lemma *seqr-or-distl*:
$((P \vee Q) ;; R) = ((P ;; R) \vee (Q ;; R))$
by *rel-tac*

lemma *seqr-or-distr*:
$(P ;; (Q \vee R)) = ((P ;; Q) \vee (P ;; R))$
by *rel-tac*

lemma *seqr-and-distr-ufunc*:
  ufunctional $P \Longrightarrow (P\ ;;\ (Q \land R)) = ((P\ ;;\ Q) \land (P\ ;;\ R))$
  by *rel-tac*

lemma *seqr-and-distl-uinj*:
  uinj $R \Longrightarrow ((P \land Q)\ ;;\ R) = ((P\ ;;\ R) \land (Q\ ;;\ R))$
  by (*rel-tac*, *metis*)

lemma *seqr-unfold*:
  $(P\ ;;\ Q) = (\exists\ v \cdot P[\![\ll v\gg/\$\Sigma']\!] \land Q[\![\ll v\gg/\$\Sigma]\!])$
  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* $x{=}get_{xa}\ y$ **in** *exI*)
  apply (*rule-tac* $x{=}y$ **in** *exI*)
  apply (*simp*)
done

lemma *seqr-left-one-point*:
  assumes *uvar x*
  shows $(P \land (\$x' =_u \ll v\gg)\ ;;\ Q) = (P[\![\ll v\gg/\$x']\!]\ ;;\ Q[\![\ll v\gg/\$x]\!])$
  using *assms*
  by (*rel-tac*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-right-one-point*:
  assumes *uvar x*
  shows $(P\ ;;\ (\$x =_u \ll v\gg) \land Q) = (P[\![\ll v\gg/\$x']\!]\ ;;\ Q[\![\ll v\gg/\$x]\!])$
  using *assms*
  by (*rel-tac*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-insert-ident*:
  assumes *uvar x* $\$x' \sharp P$ $\$x \sharp Q$
  shows $((\$x' =_u \$x \land P)\ ;;\ Q) = (P\ ;;\ Q)$
  using *assms*
  by (*rel-tac*, *meson vwb-lens-wb wb-lens-weak weak-lens.put-get*)

lemma *seq-var-ident-lift*:
  assumes *uvar x* $\$x' \sharp P$ $\$x \sharp Q$
  shows $((\$x' =_u \$x \land P)\ ;;\ (\$x' =_u \$x) \land Q) = (\$x' =_u \$x \land (P\ ;;\ Q))$
  using *assms* **apply** (*rel-tac*)
  by (*metis* (*no-types*, *lifting*) *vwb-lens-wb wb-lens-weak weak-lens.put-get*)

theorem *precond-equiv*:
  $P = (P\ ;;\ true) \longleftrightarrow (out\alpha \sharp P)$
  by (*rel-tac*)

theorem *postcond-equiv*:
  $P = (true\ ;;\ P) \longleftrightarrow (in\alpha \sharp P)$
  by (*rel-tac*)

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

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

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

8.4 Converse laws

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

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

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

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

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

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

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

lemma *not-convr* [*simp*]: $(\neg p)^{-} = (\neg p^{-})$
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*

lemma *pre-convr* [*simp*]: $\lceil p \rceil_<{}^- = \lceil p \rceil_>$
by (*rel-tac*)

lemma *post-convr* [*simp*]: $\lceil p \rceil_>{}^- = \lceil p \rceil_<$
by (*rel-tac*)

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

theorem *seqr-post-out*: $in\alpha \sharp r \Longrightarrow (P \;;; (Q \wedge r)) = ((P \;;; Q) \wedge r)$
by (*rel-tac*, *blast+*)

lemma *seqr-post-var-out*:
fixes x :: (*bool*, $'\alpha$) *uvar*
shows $(P \;;; (Q \wedge \$x\acute{})) = ((P \;;; Q) \wedge \$x\acute{})$
by (*rel-tac*)

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*: $out\alpha \sharp p \Longrightarrow ((p \wedge Q) \;;; R) = (p \wedge (Q \;;; R))$
by (*rel-tac*, *blast+*)

lemma *seqr-pre-var-out*:
fixes x :: (*bool*, $'\alpha$) *uvar*
shows $((\$x \wedge P) \;;; Q) = (\$x \wedge (P \;;; Q))$
by (*rel-tac*)

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

lemma *shEx-lift-seq-1* [*uquant-lift*]:
$((\exists\ x \cdot P\ x) \;;; Q) = (\exists\ x \cdot (P\ x \;;; Q))$
by *pred-tac*

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

Frame and antiframe

definition *frame* :: $('a, '\alpha)$ *lens* $\Rightarrow$ $'\alpha$ *hrelation* $\Rightarrow$ $'\alpha$ *hrelation* **where**
[*urel-defs*]: *frame* x $P = (II_x \wedge P)$

definition *antiframe* :: $('a, '\alpha)$ *lens* $\Rightarrow$ $'\alpha$ *hrelation* $\Rightarrow$ $'\alpha$ *hrelation* **where**
[*urel-defs*]: *antiframe* x $P = (II{\restriction_\alpha}x \wedge P)$

syntax
-frame :: *salpha* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (-:⟦-⟧ [*64*,*0*] *80*)
-antiframe :: *salpha* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (-:[-] [*64*,*0*] *80*)

translations
-frame x P == *CONST frame x P*
-antiframe x P == *CONST antiframe x P*

lemma *frame-disj*: $(x:\llbracket P\rrbracket \vee x:\llbracket Q\rrbracket) = x:\llbracket P \vee Q\rrbracket$
  by (*rel-tac*)

lemma *frame-conj*: $(x:\llbracket P\rrbracket \wedge x:\llbracket Q\rrbracket) = x:\llbracket P \wedge Q\rrbracket$
  by (*rel-tac*)

lemma *frame-seq*:
  $\llbracket$ *uvar x*; $\$x' \sharp P$; $\$x \sharp Q$ $\rrbracket \Longrightarrow (x:\llbracket P\rrbracket ;; x:\llbracket Q\rrbracket) = x:\llbracket P ;; Q\rrbracket$
  by (*rel-tac*, *metis vwb-lens-def wb-lens-weak weak-lens.put-get*)

lemma *antiframe-to-frame*:
  $\llbracket x \bowtie y; x +_L y = 1_L \rrbracket \Longrightarrow x:[P] = y:\llbracket P\rrbracket$
  by (*rel-tac*, *metis lens-indep-def*, *metis lens-indep-def surj-pair*)

While loop laws

lemma *while-cond-true*:
  $((while\ b\ do\ P\ od) \wedge \lceil b\rceil_<) = ((P \wedge \lceil b\rceil_<) ;; while\ b\ do\ P\ od)$
proof −
  have $(while\ b\ do\ P\ od \wedge \lceil b\rceil_<) = (((\lceil b\rceil_< \wedge P)^\star{}_u \wedge (\neg \lceil b\rceil_>)) \wedge \lceil b\rceil_<)$
    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 ... $= ((P \wedge \lceil b\rceil_<) ;; while\ b\ do\ P\ od)$
    by (*simp add*: *utp-pred.inf-commute while-def*)
  finally show *?thesis* **.**
qed

lemma *while-cond-false*:
  $((while\ b\ do\ P\ od) \wedge (\neg \lceil b\rceil_<)) = (II \wedge \neg \lceil b\rceil_<)$
proof −
  have $(while\ b\ do\ P\ od \wedge (\neg \lceil b\rceil_<)) = (((\lceil b\rceil_< \wedge P)^\star{}_u \wedge (\neg \lceil b\rceil_>)) \wedge (\neg \lceil b\rceil_<))$
    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-outα utp-pred.inf.assoc*)
  also have ... $= (((II \wedge \neg \lceil b\rceil_>) \wedge \neg \lceil b\rceil_<) \vee (((false ;; ((\lceil b\rceil_< \wedge P)^\star{}_u)) \wedge \neg \lceil b\rceil_>)))$
    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* ;; *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*)

8.5 Relational unrestriction

Relational unrestriction states that a variable is unchanged by a relation. Eventually I'd also like to have it state that the relation also does not depend on the variable's initial value, but I'm not sure how to state that yet. For now we represent this by the parametric healthiness condition RID.

definition *RID* :: ($'a$, $'\alpha$) *uvar* $\Rightarrow$ $'\alpha$ *hrelation* $\Rightarrow$ $'\alpha$ *hrelation*
where *RID x P* = $((\exists \ \$x \cdot \exists \ \$x' \cdot P) \wedge \$x' =_u \$x)$

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

lemma *RID-idem*:
semi-uvar x $\Longrightarrow$ $RID(x)(RID(x)(P)) = RID(x)(P)$
by *rel-tac*

lemma *RID-mono*:
$P \sqsubseteq Q \Longrightarrow RID(x)(P) \sqsubseteq RID(x)(Q)$
by *rel-tac*

lemma *RID-skip-r*:
uvar x $\Longrightarrow$ $RID(x)(II) = II$
apply *rel-tac*
using *vwb-lens.put-eq* **apply** *fastforce*
by *auto*

lemma *RID-disj*:
$RID(x)(P \vee Q) = (RID(x)(P) \vee RID(x)(Q))$
by *rel-tac*

lemma *RID-conj*:
uvar x $\Longrightarrow$ $RID(x)(RID(x)(P) \wedge RID(x)(Q)) = (RID(x)(P) \wedge RID(x)(Q))$
by *rel-tac*

lemma *RID-assigns-r-diff*:
$[\![$ *uvar x*; $x \sharp \sigma$ $]\!] \Longrightarrow RID(x)(\langle\sigma\rangle_a) = \langle\sigma\rangle_a$
apply (*rel-tac*)
apply (*auto simp add*: *unrest-usubst-def*)
apply (*metis vwb-lens.put-eq*)
apply (*metis vwb-lens-wb wb-lens.get-put wb-lens-weak weak-lens.put-get*)
done

lemma *RID-assign-r-same*:

uvar x $\Longrightarrow$ $RID(x)(x := v) = II$
apply (*rel-tac*)
using *vwb-lens.put-eq* **apply** *fastforce*
apply *blast*
done

lemma *RID-seq-left*:
assumes *uvar x*
shows $RID(x)(RID(x)(P) ;; Q) = (RID(x)(P) ;; RID(x)(Q))$
proof –
have $RID(x)(RID(x)(P) ;; Q) = ((\exists \ \$x \cdot \exists \ \$x' \cdot (\exists \ \$x \cdot \exists \ \$x' \cdot P) \wedge \$x' =_u \$x ;; Q) \wedge \$x' =_u \$x)$
by (*simp add*: *RID-def usubst*)
also from *assms* **have** ... $= (((\exists \ \$x \cdot \exists \ \$x' \cdot P) \wedge (\exists \ \$x \cdot \$x' =_u \$x) ;; (\exists \ \$x' \cdot Q)) \wedge \$x' =_u \$x)$
by (*rel-tac*)
also from *assms* **have** ... $= (((\exists \ \$x \cdot \exists \ \$x' \cdot P) ;; (\exists \ \$x \cdot \exists \ \$x' \cdot Q)) \wedge \$x' =_u \$x)$
apply (*rel-tac*)
apply (*metis vwb-lens.put-eq*)
apply (*metis mwb-lens.put-put vwb-lens-mwb*)
done
also from *assms* **have** ... $= ((((\exists \ \$x \cdot \exists \ \$x' \cdot P) \wedge \$x' =_u \$x) ;; (\exists \ \$x \cdot \exists \ \$x' \cdot Q)) \wedge \$x' =_u \$x)$
by (*rel-tac*, *metis* (*full-types*) *mwb-lens.put-put vwb-lens-def wb-lens-weak weak-lens.put-get*)
also have ... $= ((((\exists \ \$x \cdot \exists \ \$x' \cdot P) \wedge \$x' =_u \$x) ;; ((\exists \ \$x \cdot \exists \ \$x' \cdot Q) \wedge \$x' =_u \$x)) \wedge \$x' =_u \$x)$
by (*rel-tac*, *fastforce*)
also have ... $= ((((\exists \ \$x \cdot \exists \ \$x' \cdot P) \wedge \$x' =_u \$x) ;; ((\exists \ \$x \cdot \exists \ \$x' \cdot Q) \wedge \$x' =_u \$x)))$
by *rel-tac*
also have ... $= (RID(x)(P) ;; RID(x)(Q))$
by *rel-tac*
finally show *?thesis* **.**
qed

lemma *RID-seq-right*:
assumes *uvar x*
shows $RID(x)(P ;; RID(x)(Q)) = (RID(x)(P) ;; RID(x)(Q))$
proof –
have $RID(x)(P ;; RID(x)(Q)) = ((\exists \ \$x \cdot \exists \ \$x' \cdot P ;; (\exists \ \$x \cdot \exists \ \$x' \cdot Q) \wedge \$x' =_u \$x) \wedge \$x' =_u \$x)$
by (*simp add*: *RID-def usubst*)
also from *assms* **have** ... $= (((\exists \ \$x \cdot \ P) ;; (\exists \ \$x \cdot \exists \ \$x' \cdot Q) \wedge (\exists \ \$x' \cdot \$x' =_u \$x)) \wedge \$x' =_u \$x)$
by (*rel-tac*)
also from *assms* **have** ... $= (((\exists \ \$x \cdot \exists \ \$x' \cdot P) ;; (\exists \ \$x \cdot \exists \ \$x' \cdot Q)) \wedge \$x' =_u \$x)$
apply (*rel-tac*)
apply (*metis vwb-lens.put-eq*)
apply (*metis mwb-lens.put-put vwb-lens-mwb*)
done
also from *assms* **have** ... $= ((((\exists \ \$x \cdot \exists \ \$x' \cdot P) \wedge \$x' =_u \$x) ;; (\exists \ \$x \cdot \exists \ \$x' \cdot Q)) \wedge \$x' =_u \$x)$
by (*rel-tac*, *metis* (*full-types*) *mwb-lens.put-put vwb-lens-def wb-lens-weak weak-lens.put-get*)
also have ... $= ((((\exists \ \$x \cdot \exists \ \$x' \cdot P) \wedge \$x' =_u \$x) ;; ((\exists \ \$x \cdot \exists \ \$x' \cdot Q) \wedge \$x' =_u \$x)) \wedge \$x' =_u \$x)$
by (*rel-tac*, *fastforce*)
also have ... $= ((((\exists \ \$x \cdot \exists \ \$x' \cdot P) \wedge \$x' =_u \$x) ;; ((\exists \ \$x \cdot \exists \ \$x' \cdot Q) \wedge \$x' =_u \$x)))$
by *rel-tac*
also have ... $= (RID(x)(P) ;; RID(x)(Q))$

by *rel-tac*
finally show *?thesis* **.**
qed

definition *unrest-relation* :: $('a, '\alpha)$ *uvar* $\Rightarrow$ $'\alpha$ *hrelation* $\Rightarrow$ *bool* (**infix** $\sharp\sharp$ *20*)
where $(x \sharp\sharp P) \longleftrightarrow (P = RID(x)(P))$

declare *unrest-relation-def* [*urel-defs*]

lemma *skip-r-runrest* [*unrest*]:
uvar $x \Longrightarrow x \sharp\sharp II$
by (*simp add*: *RID-skip-r unrest-relation-def*)

lemma *assigns-r-runrest*:
$[\![$ *uvar* x; $x \sharp \sigma$ $]\!] \Longrightarrow x \sharp\sharp \langle\sigma\rangle_a$
by (*simp add*: *RID-assigns-r-diff unrest-relation-def*)

lemma *seq-r-runrest* [*unrest*]:
assumes *uvar* x $x \sharp\sharp P$ $x \sharp\sharp Q$
shows $x \sharp\sharp (P ;; Q)$
by (*metis RID-seq-left assms unrest-relation-def*)

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

lemma *and-runrest* [*unrest*]: $[\![$ *uvar* x; $x \sharp\sharp P$; $x \sharp\sharp Q$ $]\!] \Longrightarrow x \sharp\sharp (P \wedge Q)$
by (*metis RID-conj unrest-relation-def*)

lemma *or-runrest* [*unrest*]: $[\![$ $x \sharp\sharp P$; $x \sharp\sharp Q$ $]\!] \Longrightarrow x \sharp\sharp (P \vee Q)$
by (*simp add*: *RID-disj unrest-relation-def*)

8.6 Alphabet laws

lemma *aext-cond* [*alpha*]:
$(P \triangleleft b \triangleright Q) \oplus_p a = ((P \oplus_p a) \triangleleft (b \oplus_p a) \triangleright (Q \oplus_p a))$
by *rel-tac*

lemma *aext-seq* [*alpha*]:
wb-lens $a \Longrightarrow ((P ;; Q) \oplus_p (a \times_L a)) = ((P \oplus_p (a \times_L a)) ;; (Q \oplus_p (a \times_L a)))$
by (*rel-tac*, *metis wb-lens-weak weak-lens.put-get*)

8.7 Relation algebra laws

theorem *RA1*: $(P ;; (Q ;; R)) = ((P ;; Q) ;; R)$
using *seqr-assoc* **by** *auto*

theorem *RA2*: $(P ;; II) = P$ $(II ;; P) = P$
by *simp-all*

theorem *RA3*: $P^{--} = P$
by *simp*

theorem *RA4*: $(P ;; Q)^{-} = (Q^{-} ;; P^{-})$
by *simp*

theorem *RA5*: $(P \vee Q)^{-} = (P^{-} \vee Q^{-})$

by *rel-tac*

theorem *RA6*: $((P \vee Q) ;; R) = ((P;;R) \vee (Q;;R))$
using *seqr-or-distl* **by** *blast*

theorem *RA7*: $((P^{-} ;; (\neg(P ;; Q))) \vee (\neg Q)) = (\neg Q)$
by (*rel-tac*)

8.8 Relational alphabet extension

lift-definition *rel-alpha-ext* :: $'\beta$ *hrelation* $\Rightarrow$ $('\beta \Longrightarrow '\alpha) \Rightarrow '\alpha$ *hrelation* (**infix** $\oplus_R$ *65*)
is λ P x $(b1, b2)$. P $(get_x\ b1, get_x\ b2) \wedge (\forall\ b.\ b1 \oplus_L b\ on\ x = b2 \oplus_L b\ on\ x)$ **.**

lemma *rel-alpha-ext-alt-def*:
assumes *uvar* y $x +_L y \approx_L 1_L$ $x \bowtie y$
shows $P \oplus_R x = (P \oplus_p (x \times_L x) \wedge \$y\acute{} =_u \$y)$
using *assms*
apply (*rel-tac*, *simp-all add*: *lens-override-def*)
apply (*metis lens-indep-get lens-indep-sym*)
apply (*metis vwb-lens-def wb-lens.get-put wb-lens-def weak-lens.put-get*)
done

8.9 Program values

abbreviation *prog-val* :: $'\alpha$ *hrelation* $\Rightarrow$ $('\alpha$ *hrelation*, $'\alpha)$ *uexpr* $(\{\!|\text{-}|\!\}_u)$
where $\{\!|P|\!\}_u \equiv \ll P \gg$

lift-definition *call* :: $('\alpha$ *hrelation*, $'\alpha)$ *uexpr* $\Rightarrow$ $'\alpha$ *hrelation*
is λ P b. P (*fst* b) b **.**

lemma *call-prog-val*: *call* $\{\!|P|\!\}_u = P$
by (*simp add*: *call-def urel-defs lit.rep-eq Rep-uexpr-inverse*)

end

8.10 Relational Hoare calculus

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

named-theorems *hoare*

definition *hoare-r* :: $'\alpha$ *condition* $\Rightarrow$ $'\alpha$ *hrelation* $\Rightarrow$ $'\alpha$ *condition* $\Rightarrow$ *bool* $(\{\!|\text{-}|\!\}\text{-}\{\!|\text{-}|\!\}_u)$ **where**
$\{\!|p|\!\}Q\{\!|r|\!\}_u = ((\lceil p \rceil_< \Rightarrow \lceil r \rceil_>) \sqsubseteq Q)$

declare *hoare-r-def* [*upred-defs*]

lemma *hoare-r-conj* [*hoare*]: $[\![\ \{\!|p|\!\}Q\{\!|r|\!\}_u; \{\!|p|\!\}Q\{\!|s|\!\}_u\ ]\!] \Longrightarrow \{\!|p|\!\}Q\{\!|r \wedge s|\!\}_u$
by *rel-tac*

lemma *hoare-r-conseq* [*hoare*]: $[\![\ `p_1 \Rightarrow p_2`; \{\!|p_2|\!\}S\{\!|q_2|\!\}_u; `q_2 \Rightarrow q_1`\ ]\!] \Longrightarrow \{\!|p_1|\!\}S\{\!|q_1|\!\}_u$
by *rel-tac*

lemma *assigns-hoare-r* [*hoare*]: $\sigma \dagger q = p \Longrightarrow \{\!|p|\!\}\langle\sigma\rangle_a\{\!|q|\!\}_u$
by *rel-tac*

lemma *skip-hoare-r* [*hoare*]: $\{\!|p|\!\}II\{\!|p|\!\}_u$
by *rel-tac*

lemma *seq-hoare-r* [*hoare*]: $[\![\ \{\!|p|\!\}Q_1\{\!|s|\!\}_u\ ;\ \{\!|s|\!\}Q_2\{\!|r|\!\}_u\ ]\!] \Longrightarrow \{\!|p|\!\}Q_1\ ;;\ Q_2\{\!|r|\!\}_u$
by *rel-tac*

lemma *cond-hoare-r* [*hoare*]: $[\![\ \{\!|b \wedge p|\!\}S\{\!|q|\!\}_u\ ;\ \{\!|\neg b \wedge p|\!\}T\{\!|q|\!\}_u\ ]\!] \Longrightarrow \{\!|p|\!\}S \triangleleft b \triangleright_r\ T\{\!|q|\!\}_u$
by *rel-tac*

lemma *while-hoare-r* [*hoare*]:
assumes $\{\!|p \wedge b|\!\}S\{\!|p|\!\}_u$
shows $\{\!|p|\!\}$*while b do S od*$\{\!|\neg b \wedge p|\!\}_u$
proof −
from *assms* **have** $(\lceil p\rceil_< \Rightarrow \lceil p\rceil_>) \sqsubseteq (II \sqcap ((\lceil b\rceil_< \wedge S)\ ;;\ (\lceil p\rceil_< \Rightarrow \lceil p\rceil_>)))$
by (*simp add*: *hoare-r-def*) (*rel-tac*)
hence *p*: $(\lceil p\rceil_< \Rightarrow \lceil p\rceil_>) \sqsubseteq (\lceil b\rceil_< \wedge S)^\star{}_u$
by (*rule upred-quantale.star-inductl-one*[*rule-format*])
have $(\neg\lceil b\rceil_> \wedge \lceil p\rceil_>) \sqsubseteq ((\lceil p\rceil_< \wedge (\lceil p\rceil_< \Rightarrow \lceil p\rceil_>)) \wedge (\neg\lceil b\rceil_>))$
by (*rel-tac*)
with *p* **have** $(\neg\lceil b\rceil_> \wedge \lceil p\rceil_>) \sqsubseteq ((\lceil p\rceil_< \wedge (\lceil b\rceil_< \wedge S)^\star{}_u) \wedge (\neg\lceil b\rceil_>))$
by (*meson order-refl order-trans utp-pred.inf-mono*)
thus *?thesis*
unfolding *hoare-r-def while-def*
by (*auto intro*: *spec-refine simp add*: *alpha utp-pred.conj-assoc*)
qed

lemma *while-invr-hoare-r* [*hoare*]:
assumes $\{\!|p \wedge b|\!\}S\{\!|p|\!\}_u$ ‘*pre* ⇒ *p*‘ ‘(¬*b* ∧ *p*) ⇒ *post*‘
shows $\{\!|pre|\!\}$*while b invr p do S od*$\{\!|post|\!\}_u$
by (*metis assms hoare-r-conseq while-hoare-r while-inv-def*)

end

8.11 Weakest precondition calculus

theory *utp-wp*
imports *utp-hoare*
begin

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

named-theorems *wp*

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

consts
uwp :: ′*a* ⇒ ′*b* ⇒ ′*c* (**infix** *wp 60*)

definition *wp-upred* :: (′α, ′β) *relation* ⇒ ′β *condition* ⇒ ′α *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*]:
  $\langle\sigma\rangle_a$ *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*

theorem *wp-hoare-link*:
  $\{\!|p|\!\}Q\{\!|r|\!\}_u \longleftrightarrow (Q$ *wp* $r \sqsubseteq p)$
  by *rel-tac*

end

9 Relational operational semantics

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

fun *trel* :: $'\alpha$ *usubst* $\times$ $'\alpha$ *hrelation* $\Rightarrow$ $'\alpha$ *usubst* $\times$ $'\alpha$ *hrelation* $\Rightarrow$ *bool* (**infix** $\rightarrow_u$ *85*) **where**
$(\sigma, P) \rightarrow_u (\varrho, Q) \longleftrightarrow (\langle\sigma\rangle_a\ ;;\ P) \sqsubseteq (\langle\varrho\rangle_a\ ;;\ Q)$

lemma *trans-trel*:
  $[\![\ (\sigma, P) \rightarrow_u (\varrho, Q); (\varrho, Q) \rightarrow_u (\varphi, R)\ ]\!] \Longrightarrow (\sigma, P) \rightarrow_u (\varphi, R)$
  by *auto*

lemma *skip-trel*: $(\sigma, II) \rightarrow_u (\sigma, II)$
  by *simp*

lemma *assigns-trel*: $(\sigma, \langle\varrho\rangle_a) \rightarrow_u (\varrho \circ \sigma, II)$
  by (*simp add*: *assigns-comp*)

lemma *assign-trel*:
  fixes x :: $('a, '\alpha)$ *uvar*
  assumes *uvar* x
  shows $(\sigma, x := v) \rightarrow_u (\sigma(x \mapsto_s \sigma \dagger v), II)$
  by (*simp add*: *assigns-comp subst-upd-comp*)

lemma *seq-trel*:
  assumes $(\sigma, P) \rightarrow_u (\varrho, Q)$

shows $(\sigma,\ P\ ;;\ R) \rightarrow_u (\varrho,\ Q\ ;;\ R)$
by (*metis* (*no-types*, *lifting*) *assms seqr-assoc trel.simps upred-quantale.mult-isor*)

lemma *seq-skip-trel*:
$(\sigma,\ II\ ;;\ P) \rightarrow_u (\sigma,\ P)$
by *simp*

lemma *nondet-left-trel*:
$(\sigma,\ P \sqcap Q) \rightarrow_u (\sigma,\ P)$
by (*simp add*: *upred-quantale.subdistl*)

lemma *nondet-right-trel*:
$(\sigma,\ P \sqcap Q) \rightarrow_u (\sigma,\ Q)$
using *nondet-left-trel* **by** *force*

lemma *rcond-true-trel*:
assumes $\sigma \dagger b = true$
shows $(\sigma,\ P \triangleleft b \triangleright_r Q) \rightarrow_u (\sigma,\ P)$
using *assms*
by (*simp add*: *assigns-r-comp usubst aext-true cond-unit-T*)

lemma *rcond-false-trel*:
assumes $\sigma \dagger b = false$
shows $(\sigma,\ P \triangleleft b \triangleright_r Q) \rightarrow_u (\sigma,\ Q)$
using *assms*
by (*simp add*: *assigns-r-comp usubst aext-false cond-unit-F*)

lemma *while-true-trel*:
assumes $\sigma \dagger b = true$
shows $(\sigma,\ while\ b\ do\ P\ od) \rightarrow_u (\sigma,\ P\ ;;\ while\ b\ do\ P\ od)$
by (*metis assms rcond-true-trel while-unfold*)

lemma *while-false-trel*:
assumes $\sigma \dagger b = false$
shows $(\sigma,\ while\ b\ do\ P\ od) \rightarrow_u (\sigma,\ II)$
by (*metis assms rcond-false-trel while-unfold*)

declare *trel.simps* [*simp del*]

end

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

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

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

abbreviation *Healthy-carrier* :: $'\alpha$ *Healthiness-condition* $\Rightarrow$ $'\alpha$ *upred set* ($[\![$-$]\!]$)
where $[\![H]\!] \equiv \{P.\ P\ \mathit{is}\ H\}$

definition $\mathit{Idempotent}(H) \longleftrightarrow (\forall\ P.\ H(H(P)) = H(P))$

definition $\mathit{Monotonic}(H) \longleftrightarrow (\forall\ P\ Q.\ Q \sqsubseteq P \longrightarrow (H(Q) \sqsubseteq H(P)))$

definition $\mathit{IMH}(H) \longleftrightarrow \mathit{Idempotent}(H) \wedge \mathit{Monotonic}(H)$

definition $\mathit{Antitone}(H) \longleftrightarrow (\forall\ P\ Q.\ Q \sqsubseteq P \longrightarrow (H(P) \sqsubseteq H(Q)))$

definition $\mathit{NM} : \mathit{NM}(P) = (\neg\ P \wedge \mathit{true})$

lemma $\mathit{Monotonic}(\mathit{NM})$
  apply (*simp add*:*Monotonic-def*)
  nitpick
  oops

lemma $\mathit{Antitone}(\mathit{NM})$
  by (*simp add*:*Antitone-def NM*)

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

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

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

lemma *Conjunctive-conj*:
  assumes $\mathit{Conjunctive}(HC)$
  shows $HC(P \wedge Q) = (HC(P) \wedge Q)$
  using *assms* **unfolding** *Conjunctive-def*
  by (*metis utp-pred.inf.assoc utp-pred.inf.commute*)

lemma *Conjunctive-distr-conj*:
  assumes $\mathit{Conjunctive}(HC)$
  shows $HC(P \wedge Q) = (HC(P) \wedge HC(Q))$
  using *assms* **unfolding** *Conjunctive-def*
  by (*metis Conjunctive-conj assms utp-pred.inf.assoc utp-pred.inf-right-idem*)

lemma *Conjunctive-distr-disj*:
  assumes $\mathit{Conjunctive}(HC)$
  shows $HC(P \vee Q) = (HC(P) \vee HC(Q))$
  using *assms* **unfolding** *Conjunctive-def*
  using *utp-pred.inf-sup-distrib2* **by** *fastforce*

lemma *Conjunctive-distr-cond*:
  assumes *Conjunctive*(HC)
  shows $HC(P \triangleleft b \triangleright Q) = (HC(P) \triangleleft b \triangleright HC(Q))$
  using *assms* **unfolding** *Conjunctive-def*
  by (*metis cond-conj-distr utp-pred.inf-commute*)

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

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

lemma *FunctionalConjunctive-Monotonic*:
  FunctionalConjunctive(H) $\Longrightarrow$ *Monotonic*(H)
  unfolding *FunctionalConjunctive-def* **by** (*metis Monotonic-def utp-pred.inf-mono*)

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

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

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

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

10.1 UTP theory hierarchy

Unfortunately we can currently only characterise UTP theories of homogeneous relations; this is due to restrictions in the instantiation of Isabelle's polymorphic constants.

consts
  utp-hcond :: $('\mathcal{T} \times '\alpha)$ *itself* $\Rightarrow$ $('\alpha \times '\alpha)$ *Healthiness-condition* ($\mathcal{H}$1)
  utp-unit :: $('\mathcal{T} \times '\alpha)$ *itself* $\Rightarrow$ $'\alpha$ *hrelation* ($\mathcal{II}$1)

definition *utp-order* :: $('\mathcal{T} \times '\alpha)$ *itself* $\Rightarrow$ $'\alpha$ *hrelation gorder* **where**
utp-order T = $(\!|$ *carrier* = $\{P.\ P$ *is* $\mathcal{H}_T\}$, *eq* = (*op* =), *le* = *op* $\sqsubseteq$ $|\!)$

locale *utp-theory* =
  fixes $\mathcal{T}$ :: $('\mathcal{T} \times '\alpha)$ *itself* (**structure**)
  assumes *HCond-Idem*: $\mathcal{H}(\mathcal{H}(P)) = \mathcal{H}(P)$
begin
  sublocale *partial-order utp-order* $\mathcal{T}$
    by (*unfold-locales*, *simp-all add*: *utp-order-def*)
end

locale *utp-theory-lattice* = *utp-theory* $\mathcal{T}$ + *complete-lattice utp-order* $\mathcal{T}$ **for** $\mathcal{T}$ :: $('\mathcal{T} \times '\alpha)$ *itself* (**structure**)

locale *utp-theory-left-unital* =
  utp-theory +
  assumes *Healthy-Left-Unit*: $\mathcal{II}$ *is* $\mathcal{H}$
  and *Left-Unit*: P *is* $\mathcal{H} \Longrightarrow (\mathcal{II} \;;;\; P) = P$

locale *utp-theory-right-unital* =
  utp-theory +
  assumes *Healthy-Right-Unit*: $\mathcal{II}$ *is* $\mathcal{H}$
  and *Right-Unit*: P *is* $\mathcal{H} \Longrightarrow (P \;;;\; \mathcal{II}) = P$

locale *utp-theory-unital* =
  utp-theory +
  assumes *Healthy-Unit*: $\mathcal{II}$ *is* $\mathcal{H}$
  and *Unit-Left*: P *is* $\mathcal{H} \Longrightarrow (\mathcal{II} \;;;\; P) = P$
  and *Unit-Right*: P *is* $\mathcal{H} \Longrightarrow (P \;;;\; \mathcal{II}) = P$

sublocale *utp-theory-unital* ⊆ *utp-theory-left-unital*
  by (*simp add*: *Healthy-Unit Unit-Left utp-theory-axioms utp-theory-left-unital-axioms-def utp-theory-left-unital-def*)

sublocale *utp-theory-unital* ⊆ *utp-theory-right-unital*
  by (*simp add*: *Healthy-Unit Unit-Right utp-theory-axioms utp-theory-right-unital-axioms-def utp-theory-right-unital-def*)

typedef *REL* = *UNIV* :: *unit set* **..**

abbreviation *REL* ≡ *TYPE*(*REL* × $'\alpha$)

overloading
  rel-hcond == *utp-hcond* :: (*REL* × $'\alpha$) *itself* ⇒ ($'\alpha$ × $'\alpha$) *Healthiness-condition*
  rel-unit == *utp-unit* :: (*REL* × $'\alpha$) *itself* ⇒ $'\alpha$ *hrelation*
begin
  definition *rel-hcond* :: (*REL* × $'\alpha$) *itself* ⇒ ($'\alpha$ × $'\alpha$) *upred* ⇒ ($'\alpha$ × $'\alpha$) *upred* **where**
  rel-hcond T = *id*

  definition *rel-unit* :: (*REL* × $'\alpha$) *itself* ⇒ $'\alpha$ *hrelation* **where**
  rel-unit T = *II*
end

interpretation *rel-theory*: *utp-theory-unital REL*
  by (*unfold-locales*, *simp-all add*: *rel-hcond-def rel-unit-def Healthy-def*)

end

11 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 $k = VAR\ boyle\text{-}k$
definition $p = VAR\ boyle\text{-}p$
definition $V = VAR\ boyle\text{-}V$

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

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

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

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

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

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

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

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

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

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

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

lemma $B\text{-}\varphi_1$: φ_1 *is* B
  by (*simp add*: φ_1*-def*, *pred-tac*)

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

lemma $B\text{-}\varphi_2$: $B(\varphi_2) = \varphi_1$
proof –
  have $B(\varphi_2) = B((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 100))$
    by (*simp add*: φ_2*-def*)
  also have ... $= ((\exists\ k \cdot (\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 100)) \wedge (\&k =_u \&p * \&V))$
    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

12 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.

12.1 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*)

lemma *ok-ord* [*usubst*]:
  $\$ok \prec_v \$ok´$
  by (*simp add*: *var-name-ord-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*

definition *des-lens* :: $('\alpha, '\alpha$ *alphabet-d*$)$ *lens* (Σ_D) **where**
des-lens = (| *lens-get* = *more*, *lens-put* = *fld-put more-update* |)

syntax
  -svid-alpha-d :: *svid* (Σ_D)

translations
  -svid-alpha-d => Σ_D

declare *des-lens-def* [*upred-defs*]

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

lemma *ok-indep-des-lens* [*simp*]: $ok \bowtie des\text{-}lens\ des\text{-}lens \bowtie ok$
  by (*rule lens-indepI*, *simp-all add*: *ok-def des-lens-def*)+

lemma *ok-des-bij-lens*: *bij-lens* ($ok +_L des\text{-}lens$)
  by (*unfold-locales*, *simp-all add*: *ok-def des-lens-def lens-plus-def prod.case-eq-if*)

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...

abbreviation *lift-desr* :: $('\alpha, '\beta)$ *relation* $\Rightarrow$ $('\alpha, '\beta)$ *relation-d* $(\lceil\text{-}\rceil_D)$
where $\lceil P\rceil_D \equiv P \oplus_p (des\text{-}lens \times_L des\text{-}lens)$

abbreviation *drop-desr* :: $('\alpha, '\beta)$ *relation-d* $\Rightarrow$ $('\alpha, '\beta)$ *relation* $(\lfloor\text{-}\rfloor_D)$
where $\lfloor P\rfloor_D \equiv P \restriction_p (des\text{-}lens \times_L des\text{-}lens)$

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

syntax
  -assignmentd :: *svid-list* $\Rightarrow$ *uexprs* $\Rightarrow$ *logic* (**infixr** $:=_D$ *55*)

translations
  -assignmentd xs vs => *CONST assigns-d* (*-mk-usubst* (*CONST id*) *xs vs*)

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

syntax

-ok-f :: *logic* ⇒ *logic* (*-^{f}* [*1000*] *1000*)
-ok-t :: *logic* ⇒ *logic* (*-^{t}* [*1000*] *1000*)
-top-d :: *logic* ($\top_D$)
-bot-d :: *logic* ($\bot_D$)

translations
P^f ⇌ *CONST usubst* (*CONST subst-upd CONST id* (*CONST ovar CONST ok*) *false*) *P*
P^t ⇌ *CONST usubst* (*CONST subst-upd CONST id* (*CONST ovar CONST ok*) *true*) *P*
$\top_D$ => *CONST not-upred* (*CONST var* (*CONST ivar CONST ok*))
$\bot_D$ => *true*

definition *pre-design* :: (*'α*, *'β*) *relation-d* ⇒ (*'α*, *'β*) *relation* (pre_D*'*(*-'*)) **where**
$pre_D(P) = \lfloor \neg\ P[\![true,false/\$ok,\$ok']\!] \rfloor_D$

definition *post-design* :: (*'α*, *'β*) *relation-d* ⇒ (*'α*, *'β*) *relation* ($post_D$*'*(*-'*)) **where**
$post_D(P) = \lfloor P[\![true,true/\$ok,\$ok']\!] \rfloor_D$

definition *wp-design* :: (*'α*, *'β*) *relation-d* ⇒ *'β condition* ⇒ *'α 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 *assigns-d-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 (*simp add*: *arestr-aext prod-mwb-lens*)

lemma *lift-desr-inv*:
fixes *P* :: (*'α*, *'β*) *relation-d*
assumes \$*ok* ♯ *P* \$*ok´* ♯ *P*
shows $\lceil \lfloor P \rfloor_D \rceil_D = P$
proof −
have *bij-lens* (*des-lens* $\times_L$ *des-lens* $+_L$ (*in-var ok* $+_L$ *out-var ok*) :: (*-*, *'α alpha-d-scheme* × *'β alpha-d-scheme*) *lens*)
(**is** *bij-lens* (*?P*))
proof −
have *?P* $\approx_L$ (*ok* $+_L$ *des-lens*) $\times_L$ (*ok* $+_L$ *des-lens*) (**is** *?P* $\approx_L$ *?Q*)
apply (*simp add*: *in-var-def out-var-def prod-as-plus*)
apply (*simp add*: *prod-as-plus*[*THEN sym*])
apply (*meson lens-equiv-sym lens-equiv-trans lens-indep-prod lens-plus-comm lens-plus-prod-exchange ok-indep-des-lens*)
done
moreover have *bij-lens ?Q*
by (*simp add*: *ok-des-bij-lens prod-bij-lens*)
ultimately show *?thesis*

by (*metis bij-lens-equiv lens-equiv-sym*)
qed

with *assms* **show** *?thesis*
apply (*rule-tac aext-arestr*[*of - in-var ok* $+_L$ *out-var ok*])
apply (*simp add*: *prod-mwb-lens*)
apply (*simp*)
apply (*metis alpha-in-var lens-indep-prod lens-indep-sym ok-indep-des-lens out-var-def prod-as-plus*)
using *unrest-var-comp* **apply** *blast*
done
qed

12.2 Design laws

lemma *prod-lens-indep-in-var* [*simp*]:
$a \bowtie x \Longrightarrow a \times_L b \bowtie \mathit{in\text{-}var}\ x$
by (*metis in-var-def in-var-indep out-in-indep out-var-def plus-pres-lens-indep prod-as-plus*)

lemma *prod-lens-indep-out-var* [*simp*]:
$b \bowtie x \Longrightarrow a \times_L b \bowtie \mathit{out\text{-}var}\ x$
by (*metis in-out-indep in-var-def out-var-def out-var-indep plus-pres-lens-indep prod-as-plus*)

lemma *unrest-out-des-lift* [*unrest*]: $out\alpha \sharp p \Longrightarrow out\alpha \sharp \lceil p \rceil_D$
by (*pred-tac*, *auto simp add*: *outα-def des-lens-def prod-lens-def*)

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*))

lemma *lift-des-skip-dr-unit-unrest*: $\$ok' \sharp P \Longrightarrow (P ;; \lceil II \rceil_D) = P$
by (*rel-tac*, *metis alpha-d.surjective alpha-d.update-convs*(*1*))

lemma *true-is-design*:
$(false \vdash true) = true$
by *rel-tac*

lemma *true-is-rdesign*:
$(false \vdash_r true) = true$
by *rel-tac*

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*)
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*:
  $\neg (P \vdash Q)^f = (\$ok \wedge P^f)$
  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)

declare *des-lens-def* [*upred-defs*]
declare *lens-create-def* [*upred-defs*]
declare *prod-lens-def* [*upred-defs*]
declare *in-var-def* [*upred-defs*]

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

theorem *rdesign-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-top-left-zero*: $(\top_D \;;; (P \vdash Q)) = \top_D$
  by (*rel-tac*, *meson alpha-d.select-convs*(*1*))

theorem *design-choice*:
  $(P_1 \vdash P_2) \sqcap (Q_1 \vdash Q_2) = ((P_1 \wedge Q_1) \vdash (P_2 \vee Q_2))$
  by *rel-tac*

theorem *design-inf*:
$(P_1 \vdash P_2) \sqcup (Q_1 \vdash Q_2) = ((P_1 \vee Q_1) \vdash ((P_1 \Rightarrow P_2) \wedge (Q_1 \Rightarrow Q_2)))$
by *rel-tac*

theorem *rdesign-choice*:
$(P_1 \vdash_r P_2) \sqcap (Q_1 \vdash_r Q_2) = ((P_1 \wedge Q_1) \vdash_r (P_2 \vee Q_2))$
by *rel-tac*

theorem *design-condr*:
$((P_1 \vdash P_2) \triangleleft b \triangleright (Q_1 \vdash Q_2)) = ((P_1 \triangleleft b \triangleright Q_1) \vdash (P_2 \triangleleft b \triangleright Q_2))$
by *rel-tac*

lemma *design-top*:
$(P \vdash Q) \sqsubseteq \top_D$
by *rel-tac*

lemma *design-bottom*:
$\bot_D \sqsubseteq (P \vdash Q)$
by *simp*

lemma *design-USUP*:
assumes $A \neq \{\}$
shows $(\bigsqcap\ i \in A \cdot P(i) \vdash Q(i)) = (\bigsqcup\ i \in A \cdot P(i)) \vdash (\bigsqcap\ i \in A \cdot Q(i))$
using *assms* **by** *rel-tac*

lemma *design-UINF*:
$(\bigsqcup\ i \in A \cdot P(i) \vdash Q(i)) = (\bigsqcap\ i \in A \cdot P(i)) \vdash (\bigsqcup\ i \in A \cdot P(i) \Rightarrow Q(i))$
by *rel-tac*

theorem *design-composition-subst*:
assumes
$\$ok' \sharp P1$ $\$ok \sharp P2$
shows $((P1 \vdash Q1) ;; (P2 \vdash Q2)) =$
$(((\neg ((\neg P1) ;; true)) \wedge \neg (Q1[\![true/\$ok']\!] ;; (\neg P2))) \vdash (Q1[\![true/\$ok']\!] ;; Q2[\![true/\$ok]\!]))$
proof −
have $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (\exists\ ok_0 \cdot ((P1 \vdash Q1)[\![\guillemotleft ok_0\guillemotright/\$ok']\!] ;; (P2 \vdash Q2)[\![\guillemotleft ok_0\guillemotright/\$ok]\!]))$
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 ... $= (((\$ok \wedge P1 \Rightarrow Q1[\![true/\$ok']\!]) ;; (P2 \Rightarrow \$ok' \wedge Q2[\![true/\$ok]\!])) \vee ((\neg (\$ok \wedge P1)) ;; true))$
by (*simp add*: *design-def usubst unrest*, *pred-tac*)
also have ... $= ((\neg\$ok ;; true_h) \vee (\neg P1 ;; true) \vee (Q1[\![true/\$ok']\!] ;; \neg P2) \vee (\$ok' \wedge (Q1[\![true/\$ok']\!] ;; Q2[\![true/\$ok]\!])))$
by (*rel-tac*)
also have ... $= (((\neg ((\neg P1) ;; true)) \wedge \neg (Q1[\![true/\$ok']\!] ;; (\neg P2))) \vdash (Q1[\![true/\$ok']\!] ;; Q2[\![true/\$ok]\!]))$
by (*simp add*: *precond-right-unit design-def unrest*, *rel-tac*)
finally show *?thesis* **.**
qed

lemma *design-export-ok*:

$P \vdash Q = (P \vdash (\$ok \wedge Q))$
by (*rel-tac*)

lemma *design-export-ok'*:
$P \vdash Q = (P \vdash (\$ok\acute{} \wedge Q))$
by (*rel-tac*)

theorem *design-composition*:
assumes
$\$ok\acute{} \sharp P1$ $\$ok \sharp P2$ $\$ok\acute{} \sharp Q1$ $\$ok \sharp Q2$
shows $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (((\neg ((\neg P1) ;; true)) \wedge \neg (Q1 ;; (\neg P2))) \vdash (Q1 ;; Q2))$
using *assms* **by** (*simp add*: *design-composition-subst usubst*)

lemma *runrest-ident-var*:
assumes $x \sharp\sharp P$
shows $(\$x \wedge P) = (P \wedge \$x\acute{})$
proof −
have $P = (\$x\acute{} =_u \$x \wedge P)$
by (*metis* (*no-types*, *lifting*) *RID-def assms conj-idem unrest-relation-def utp-pred.inf.left-commute*)
moreover have $(\$x\acute{} =_u \$x \wedge (\$x \wedge P)) = (\$x\acute{} =_u \$x \wedge (P \wedge \$x\acute{}))$
by (*rel-tac*)
ultimately show *?thesis*
by (*metis utp-pred.inf.assoc utp-pred.inf-left-commute*)
qed

theorem *design-composition-runrest*:
assumes
$\$ok\acute{} \sharp P1$ $\$ok \sharp P2$ $ok \sharp\sharp Q1$ $ok \sharp\sharp Q2$
shows $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (((\neg ((\neg P1) ;; true)) \wedge \neg (Q1^t ;; (\neg P2))) \vdash (Q1 ;; Q2))$
proof −
have $(\$ok \wedge \$ok\acute{} \wedge (Q1^t ;; Q2[\![true/\$ok]\!])) = (\$ok \wedge \$ok\acute{} \wedge (Q1 ;; Q2))$
proof −
have $(\$ok \wedge \$ok\acute{} \wedge (Q1 ;; Q2)) = (\$ok \wedge Q1 ;; Q2 \wedge \$ok\acute{})$
by (*metis* (*no-types*, *hide-lams*) *seqr-post-out seqr-pre-out utp-pred.inf.commute utp-rel.unrest-iuvar utp-rel.unrest-ouvar uvar-ok vwb-lens-mwb*)
also have ... $= (Q1 \wedge \$ok\acute{} ;; \$ok \wedge Q2)$
by (*simp add*: *assms*(*3*) *assms*(*4*) *runrest-ident-var*)
also have ... $= (Q1^t ;; Q2[\![true/\$ok]\!])$
by (*metis seqr-left-one-point seqr-post-transfer true-alt-def uivar-convr upred-eq-true utp-pred.inf.cobounded2 utp-pred.inf.orderE utp-rel.unrest-iuvar uvar-ok vwb-lens-mwb*)
finally show *?thesis*
by (*metis utp-pred.inf.left-commute utp-pred.inf-left-idem*)
qed
moreover have $(\neg (\neg P1 ;; true) \wedge \neg (Q1^t ;; \neg P2)) \vdash (Q1^t ;; Q2[\![true/\$ok]\!]) =$
$(\neg (\neg P1 ;; true) \wedge \neg (Q1^t ;; \neg P2)) \vdash (\$ok \wedge \$ok\acute{} \wedge (Q1^t ;; Q2[\![true/\$ok]\!]))$
by (*metis design-export-ok design-export-ok'*)
ultimately show *?thesis* **using** *assms*
by (*simp add*: *design-composition-subst usubst*, *metis design-export-ok design-export-ok'*)
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 alpha*)

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
$out\alpha \sharp p1$ $\$ok \sharp P2$ $\$ok' \sharp Q1$ $\$ok \sharp Q2$
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 alpha*)

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*

lemma *assigns-d-id* [*simp*]: $\langle id \rangle_D = II_D$
by (*rel-tac*)

lemma *assign-d-left-comp*:
$(\langle f \rangle_D ;; (P \vdash_r Q)) = (\lceil f \rceil_s \dagger P \vdash_r \lceil f \rceil_s \dagger Q)$
by (*simp add*: *assigns-d-def rdesign-composition assigns-r-comp subst-not*)

lemma *assign-d-right-comp*:
$((P \vdash_r Q) ;; \langle f \rangle_D) = ((\neg (\neg P ;; true)) \vdash_r (Q ;; \langle f \rangle_a))$
by (*simp add*: *assigns-d-def rdesign-composition*)

lemma *assigns-d-comp*:
$(\langle f \rangle_D ;; \langle g \rangle_D) = \langle g \circ f \rangle_D$
using *assms*
by (*simp add*: *assigns-d-def rdesign-composition assigns-comp*)

12.3 Design preconditions

lemma *design-pre-choice* [*simp*]:
$pre_D(P \sqcap Q) = (pre_D(P) \wedge pre_D(Q))$
by (*rel-tac*)

lemma *design-post-choice* [*simp*]:
$post_D(P \sqcap Q) = (post_D(P) \vee post_D(Q))$
by (*rel-tac*)

lemma *design-pre-condr* [*simp*]:
$pre_D(P \triangleleft \lceil b \rceil_D \triangleright Q) = (pre_D(P) \triangleleft b \triangleright pre_D(Q))$
by (*rel-tac*)

lemma *design-post-condr* [*simp*]:
$post_D(P \triangleleft \lceil b \rceil_D \triangleright Q) = (post_D(P) \triangleleft b \triangleright post_D(Q))$
by (*rel-tac*)

12.4 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-below-top*:
$H1(P) \sqsubseteq \top_D$
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 (*pred-tac*, *metis alpha-d.select-convs(1)*)

theorem *H1-left-zero*:
assumes *P is H1*
shows $(true ;; P) = true$
proof −
from *assms* **have** $(true ;; P) = (true ;; (\$ok \Rightarrow P))$
by (*simp add*: *H1-def Healthy-def'*)

also from *assms* **have** ... = $(true ;; (\neg \$ok \vee P))$ (**is** - = (*?true* ;; -))
by (*simp add*: *impl-alt-def*)
also from *assms* **have** ... = $((?true ;; \neg \$ok) \vee (?true ;; P))$
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 :: *'α 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

lemma *H1-design*:
$H1(P \vdash Q) = (P \vdash Q)$
by (*rel-tac*)

lemma *H1-rdesign*:
$H1(P \vdash_r Q) = (P \vdash_r Q)$
by (*rel-tac*)

lemma *H1-choice-closed*:
$[\![P$ *is H1*; Q *is H1* $]\!] \Longrightarrow P \sqcap Q$ *is H1*
by (*simp add*: *H1-def Healthy-def′ disj-upred-def impl-alt-def semilattice-sup-class.sup-left-commute*)

lemma *H1-inf-closed*:
$[\![P$ *is H1*; Q *is H1* $]\!] \Longrightarrow P \sqcup Q$ *is H1*
by (*rel-tac*, *blast+*)

lemma *H1-USUP*:
assumes $A \neq \{\}$
shows $H1(\prod i \in A \cdot P(i)) = (\prod i \in A \cdot H1(P(i)))$
using *assms* **by** (*rel-tac*)

lemma *H1-Sup*:
assumes $A \neq \{\}$ $\forall P \in A.$ P *is H1*
shows $(\prod A)$ *is H1*
proof −
from *assms*(*2*) **have** *H1* ‘ $A = A$

by (*auto simp add*: *Healthy-def rev-image-eqI*)
with *H1-USUP*[*of A id*, *OF assms*(*1*)] **show** *?thesis*
by (*simp add*: *USUP-as-Sup-image Healthy-def*)
qed

lemma *H1-UINF*:
shows $H1(\bigsqcup\ i \in A \cdot P(i)) = (\bigsqcup\ i \in A \cdot H1(P(i)))$
by (*rel-tac*)

lemma *H1-Inf*:
assumes $\forall\ P \in A.\ P$ *is H1*
shows $(\bigsqcup\ A)$ *is H1*
proof −
from *assms* **have** $H1\ `\ A = A$
by (*auto simp add*: *Healthy-def rev-image-eqI*)
with *H1-UINF*[*of A id*] **show** *?thesis*
by (*simp add*: *UINF-as-Inf-image Healthy-def*)
qed

12.5 H2: A specification cannot require non-termination

lemma *J-split*:
shows $(P\ ;;\ J) = (P^f \vee (P^t \wedge \$ok'))$
proof −
have $(P\ ;;\ J) = (P\ ;;\ ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D))$
by (*simp add*: *H2-def J-def design-def*)
also have ... $= (P\ ;;\ ((\$ok \Rightarrow \$ok \wedge \$ok') \wedge \lceil II \rceil_D))$
by *rel-tac*
also have ... $= ((P\ ;;\ (\neg \$ok \wedge \lceil II \rceil_D)) \vee (P\ ;;\ (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))))$
by *rel-tac*
also have ... $= (P^f \vee (P^t \wedge \$ok'))$
proof −
have $(P\ ;;\ (\neg \$ok \wedge \lceil II \rceil_D)) = P^f$
proof −
have $(P\ ;;\ (\neg \$ok \wedge \lceil II \rceil_D)) = ((P \wedge \neg \$ok')\ ;;\ \lceil II \rceil_D)$
by *rel-tac*
also have ... $= (\exists\ \$ok' \cdot P \wedge \$ok' =_u$ *false*$)$
by (*rel-tac*, *metis* (*mono-tags*, *lifting*) *alpha-d.surjective alpha-d.update-convs*(*1*))
also have ... $= P^f$
by (*metis C1 one-point out-var-uvar pr-var-def unrest-as-exists uvar-ok vwb-lens-mwb*)
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 $\$ok' \sharp P$ $\$ok' \sharp Q$
  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*)
  also have ... $= (\neg \$ok)$
    by *pred-tac*
  finally show *?thesis* **.**
qed

lemma *H2-choice-closed*:

$\llbracket$ *P is H2*; *Q is H2* $\rrbracket \Longrightarrow P \sqcap Q$ *is H2*
by (*metis H2-def Healthy-def′ disj-upred-def seqr-or-distl*)

lemma *H2-inf-closed*:
assumes *P is H2 Q is H2*
shows $P \sqcup Q$ *is H2*
proof −
have $P \sqcup Q = (P^f \vee P^t \wedge \$ok') \sqcup (Q^f \vee Q^t \wedge \$ok')$
by (*metis H2-def Healthy-def J-split assms*(*1*) *assms*(*2*))
moreover have *H2*(...) = ...
by (*simp add*: *H2-split usubst*, *pred-tac*)
ultimately show *?thesis*
by (*simp add*: *Healthy-def*)
qed

lemma *H2-USUP*:
shows $H2(\bigsqcap\ i \in A \cdot P(i)) = (\bigsqcap\ i \in A \cdot H2(P(i)))$
using *assms* **by** (*rel-tac*)

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* (*mono-tags*, *lifting*) *alpha-d.surjective alpha-d.update-convs*(*1*) *alpha-d.update-convs*(*2*))+

lemma *ok-post*: $(\$ok \wedge \lceil post_D(P)\rceil_D) = (\$ok \wedge (P^t))$
by (*pred-tac*)
(*metis alpha-d.cases-scheme alpha-d.ext-inject alpha-d.select-convs*(*1*) *alpha-d.select-convs*(*2*) *alpha-d.update-convs*(*1*)
alpha-d.update-convs(*2*))+

theorem *H1-H2-is-design*:
assumes *P is H1 P is H2*
shows $P = (\neg P^f) \vdash P^t$
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 ... = $(\neg P^f) \vdash P^t$
by *pred-tac*
finally show *?thesis* **.**
qed

lemma *H1-H2-eq-design*:
$H1\ (H2\ P) = (\neg P^f) \vdash P^t$
apply (*subst H1-H2-is-design*)
apply (*simp-all add*: *Healthy-def H1-idem H2-idem H1-H2-commute*)
apply (*simp add*: *H2-split H1-def usubst*)
apply (*rel-tac*)
done

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

lemma *design-is-H1-H2*:
$[\![\ \$ok' \sharp P; \$ok' \sharp Q\ ]\!] \Longrightarrow (P \vdash Q)$ *is H1-H2*
by (*simp add*: *H1-design H2-design Healthy-def′*)

lemma *rdesign-is-H1-H2*:
$(P \vdash_r Q)$ *is H1-H2*
by (*simp add*: *Healthy-def H1-rdesign H2-rdesign*)

lemma *seq-r-H1-H2-closed*:
assumes *P is H1-H2 Q is H1-H2*
shows $(P ;; Q)$ *is H1-H2*
proof −
obtain P_1 P_2 **where** $P = P_1 \vdash_r P_2$
by (*metis H1-H2-commute H1-H2-is-rdesign H2-idem Healthy-def assms*(*1*))
moreover obtain Q_1 Q_2 **where** $Q = Q_1 \vdash_r Q_2$
by (*metis H1-H2-commute H1-H2-is-rdesign H2-idem Healthy-def assms*(*2*))

moreover have $((P_1 \vdash_r P_2) ;; (Q_1 \vdash_r Q_2))$ *is H1-H2*
  by (*simp add*: *rdesign-composition rdesign-is-H1-H2*)
  ultimately show *?thesis* **by** *simp*
qed

lemma *assigns-d-comp-ext*:
  fixes P :: $'\alpha$ *hrelation-d*
  assumes P *is H1-H2*
  shows $(\langle\sigma\rangle_D ;; P) = \lceil\sigma \oplus_s \Sigma_D\rceil_s \dagger P$
proof –
  have $(\langle\sigma\rangle_D ;; P) = (\langle\sigma\rangle_D ;; pre_D(P) \vdash_r post_D(P))$
    by (*metis H1-H2-commute H1-H2-is-rdesign H2-idem Healthy-def′ assms*)
  also have ... $= \lceil\sigma\rceil_s \dagger pre_D(P) \vdash_r \lceil\sigma\rceil_s \dagger post_D(P)$
    by (*simp add*: *assign-d-left-comp*)
  also have ... $= \lceil\sigma \oplus_s \Sigma_D\rceil_s \dagger (pre_D(P) \vdash_r post_D(P))$
    by (*rel-tac*)
  also have ... $= \lceil\sigma \oplus_s \Sigma_D\rceil_s \dagger P$
    by (*metis H1-H2-commute H1-H2-is-rdesign H2-idem Healthy-def′ assms*)
  finally show *?thesis* **.**
qed

lemma *USUP-H1-H2-closed*:
  assumes $A \neq \{\}$ $\forall\ P \in A.\ P$ *is H1-H2*
  shows $(\bigsqcap A)$ *is H1-H2*
proof –
  from *assms* **have** A: $A = $ *H1-H2* ‘ A
    by (*auto simp add*: *Healthy-def rev-image-eqI*)
  also have $(\bigsqcap ...) = (\bigsqcap P \in A.\ H1\text{-}H2(P))$
    by *auto*
  also have ... $= (\bigsqcap P \in A \cdot H1\text{-}H2(P))$
    by (*simp add*: *USUP-as-Sup-collect*)
  also have ... $= (\bigsqcap P \in A \cdot (\neg P^f) \vdash P^t)$
    by (*meson H1-H2-eq-design*)
  also have ... $= (\bigsqcup P \in A \cdot \neg P^f) \vdash (\bigsqcap P \in A \cdot P^t)$
    by (*simp add*: *design-USUP assms*)
  also have ... *is H1-H2*
    by (*simp add*: *design-is-H1-H2 unrest*)
  finally show *?thesis* **.**
qed

definition *design-sup* :: $('\alpha, '\beta)$ *relation-d set* $\Rightarrow$ $('\alpha, '\beta)$ *relation-d* $(\bigsqcap_D$- [*900*] *900*) **where**
$\bigsqcap_D A = (if\ (A = \{\})\ then\ \top_D\ else\ \bigsqcap A)$

lemma *design-sup-H1-H2-closed*:
  assumes $\forall\ P \in A.\ P$ *is H1-H2*
  shows $(\bigsqcap_D A)$ *is H1-H2*
  apply (*auto simp add*: *design-sup-def*)
  apply (*simp add*: *H1-def H2-not-okay Healthy-def impl-alt-def*)
  using *USUP-H1-H2-closed assms* **apply** *blast*
done

lemma *design-sup-empty* [*simp*]: $\bigsqcap_D \{\} = \top_D$
  by (*simp add*: *design-sup-def*)

lemma *design-sup-non-empty* [*simp*]: $A \neq \{\} \Longrightarrow \bigsqcap_D A = \bigsqcap A$

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

lemma *UINF-H1-H2-closed*:
assumes $\forall\ P \in A.\ P$ *is H1-H2*
shows $(\bigsqcup\ A)$ *is H1-H2*
proof –
from *assms* **have** A: $A = \mathit{H1\text{-}H2}\ `\ A$
by (*auto simp add*: *Healthy-def rev-image-eqI*)
also have $(\bigsqcup\ ...) = (\bigsqcup\ P \in A.\ \mathit{H1\text{-}H2}(P))$
by *auto*
also have $... = (\bigsqcup\ P \in A \cdot \mathit{H1\text{-}H2}(P))$
by (*simp add*: *UINF-as-Inf-collect*)
also have $... = (\bigsqcup\ P \in A \cdot (\neg\ P^f) \vdash P^t)$
by (*meson H1-H2-eq-design*)
also have $... = (\bigsqcap\ P \in A \cdot \neg\ P^f) \vdash (\bigsqcup\ P \in A \cdot \neg\ P^f \Rightarrow P^t)$
by (*simp add*: *design-UINF*)
also have ... *is H1-H2*
by (*simp add*: *design-is-H1-H2 unrest*)
finally show *?thesis* **.**
qed

abbreviation *design-inf* :: $('\alpha,\ '\beta)$ *relation-d set* $\Rightarrow$ $('\alpha,\ '\beta)$ *relation-d* $(\bigsqcup_D$- [*900*] *900*) **where**
$\bigsqcup_D\ A \equiv \bigsqcup\ A$

12.6 H3: The design assumption is a precondition

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

theorem *design-condition-is-H3*:
assumes $out\alpha \sharp p$
shows $(p \vdash Q)$ *is H3*
proof –
have $((p \vdash Q)\ ;;\ II_D) = (\neg\ (\neg\ p\ ;;\ \mathit{true})) \vdash (Q^t\ ;;\ II[\![\mathit{true}/\$ok]\!])$
by (*simp add*: *skip-d-alt-def design-composition-subst unrest assms*)
also have $... = p \vdash (Q^t\ ;;\ II[\![\mathit{true}/\$ok]\!])$
using *assms precond-equiv seqr-true-lemma* **by** *force*
also have $... = p \vdash Q$
by (*rel-tac*, *metis* (*full-types*) *alpha-d.cases-scheme alpha-d.select-convs*(*1*) *alpha-d.update-convs*(*1*))
finally show *?thesis*
by (*simp add*: *H3-def Healthy-def'*)
qed

theorem *rdesign-H3-iff-pre*:
$P \vdash_r Q$ *is H3* $\longleftrightarrow P = (P\ ;;\ \mathit{true})$
proof –
have $(P \vdash_r Q\ ;;\ II_D) = (P \vdash_r Q\ ;;\ \mathit{true} \vdash_r II)$
by (*simp add*: *skip-d-def*)
also from *assms* **have** $... = (\neg\ (\neg\ P\ ;;\ \mathit{true}) \wedge \neg\ (Q\ ;;\ \neg\ \mathit{true})) \vdash_r (Q\ ;;\ II)$
by (*simp add*: *rdesign-composition*)
also from *assms* **have** $... = (\neg\ (\neg\ P\ ;;\ \mathit{true}) \wedge \neg\ (Q\ ;;\ \neg\ \mathit{true})) \vdash_r Q$
by *simp*
also have $... = (\neg\ (\neg\ P\ ;;\ \mathit{true})) \vdash_r Q$
by *pred-tac*
finally have $P \vdash_r Q$ *is H3* $\longleftrightarrow P \vdash_r Q = (\neg\ (\neg\ P\ ;;\ \mathit{true})) \vdash_r Q$

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 aext-true*)
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 *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 vwb-lens-mwb*)

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-design*:
assumes *P is H1 P is H3*
shows $P = (\neg P^f) \vdash P^t$
by (*metis H1-H2-eq-design H2-H3-absorb Healthy-def' assms*(*1*) *assms*(*2*))

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

lemma *H1-H3-impl-H2*: P *is H1-H3* $\Longrightarrow$ P *is H1-H2*
by (*metis H1-H2-commute H1-idem H2-H3-absorb Healthy-def'*)

lemma *H1-H3-eq-design-d-comp*: $H1\ (H3\ P) = ((\neg P^f) \vdash P^t\ ;;\ II_D)$
by (*metis H1-H2-eq-design H1-H3-commute H3-H2-absorb H3-def*)

lemma *H1-H3-eq-design*: $H1\ (H3\ P) = (\neg (P^f\ ;;\ true)) \vdash P^t$
apply (*simp add*: *H1-H3-eq-design-d-comp skip-d-alt-def*)
apply (*subst design-composition-subst*)
apply (*simp-all add*: *usubst unrest*)
apply (*rel-tac*)
done

lemma *H3-unrest-out-alpha-nok* [*unrest*]:
assumes *P is H1-H3*
shows $out\alpha \sharp P^f$
proof −
have $P = (\neg (P^f\ ;;\ true)) \vdash P^t$
by (*metis H1-H3-eq-design Healthy-def assms*)
also have $out\alpha \sharp (...^f)$
by (*simp add*: *design-def usubst unrest*, *rel-tac*)
finally show *?thesis* **.**
qed

lemma *H3-unrest-out-alpha* [*unrest*]: P *is H1-H3* $\Longrightarrow$ *out*α $\sharp$ $pre_D(P)$
by (*metis H1-H3-commute H1-H3-is-rdesign H1-idem Healthy-def′ precond-equiv rdesign-H3-iff-pre*)

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

12.7 H4: Feasibility

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

lemma *is-H4-alt-def*:
P *is H4* $\longleftrightarrow$ $(P\ ;;\ true) = true$
by (*rel-tac*)

lemma *H4-assigns-d*: $\langle\sigma\rangle_D$ *is H4*
proof −
have $(\langle\sigma\rangle_D\ ;;\ (false \vdash_r true_h)) = (false \vdash_r true)$
by (*simp add*: *assigns-d-def rdesign-composition assigns-r-feasible*)
moreover have ... = *true*
by (*rel-tac*)
ultimately show *?thesis*
using *is-H4-alt-def* **by** *auto*
qed

12.8 UTP theories

typedef *DES* = *UNIV* :: *unit set* **by** *simp*
typedef *NDES* = *UNIV* :: *unit set* **by** *simp*

abbreviation *DES* ≡ *TYPE*(*DES* × ′α *alphabet-d*)
abbreviation *NDES* ≡ *TYPE*(*NDES* × ′α *alphabet-d*)

overloading
des-hcond == *utp-hcond* :: (*DES* × ′α *alphabet-d*) *itself* ⇒ (′α *alphabet-d* × ′α *alphabet-d*) *Healthiness-condition*
des-unit == *utp-unit* :: (*DES* × ′α *alphabet-d*) *itself* ⇒ ′α *hrelation-d*

ndes-hcond == *utp-hcond* :: (*NDES* × ′α *alphabet-d*) *itself* ⇒ (′α *alphabet-d* × ′α *alphabet-d*) *Healthiness-condition*
ndes-unit == *utp-unit* :: (*NDES* × ′α *alphabet-d*) *itself* ⇒ ′α *hrelation-d*

begin
definition *des-hcond* :: (*DES* × ′α *alphabet-d*) *itself* ⇒ (′α *alphabet-d* × ′α *alphabet-d*) *Healthiness-condition* **where**
des-hcond t = *H1-H2*

definition *des-unit* :: (*DES* × ′α *alphabet-d*) *itself* ⇒ ′α *hrelation-d* **where**
des-unit t = II_D

definition *ndes-hcond* :: (*NDES* × ′α *alphabet-d*) *itself* ⇒ (′α *alphabet-d* × ′α *alphabet-d*) *Healthiness-condition* **where**

ndes-hcond t = *H1-H3*

definition *ndes-unit* :: (*NDES* × $'\alpha$ *alphabet-d*) *itself* ⇒ $'\alpha$ *hrelation-d* **where**
ndes-unit t = II_D

end

interpretation *des-utp-theory*: *utp-theory TYPE*(*DES* × $'\alpha$ *alphabet-d*)
by (*simp add*: *H1-H2-commute H1-idem H2-idem des-hcond-def utp-theory-def*)

interpretation *ndes-utp-theory*: *utp-theory TYPE*(*NDES* × $'\alpha$ *alphabet-d*)
by (*simp add*: *H1-H3-commute H1-idem H3-idem ndes-hcond-def utp-theory.intro*)

interpretation *des-left-unital*: *utp-theory-left-unital TYPE*(*DES* × $'\alpha$ *alphabet-d*)
apply (*unfold-locales*)
apply (*simp-all add*: *des-hcond-def des-unit-def*)
apply (*simp add*: *rdesign-is-H1-H2 skip-d-def*)
apply (*metis H1-idem H1-left-unit Healthy-def'*)
done

interpretation *ndes-unital*: *utp-theory-unital TYPE*(*NDES* × ($'\alpha$ *alphabet-d*))
apply (*unfold-locales*, *simp-all add*: *ndes-hcond-def ndes-unit-def*)
apply (*metis H1-rdesign H3-def Healthy-def' design-skip-idem skip-d-def*)
apply (*metis H1-idem H1-left-unit Healthy-def'*)
apply (*metis H1-H3-commute H3-def H3-idem Healthy-def'*)
done

interpretation *design-complete-lattice*: *utp-theory-lattice TYPE*(*DES* × $'\alpha$ *alphabet-d*)
rewrites *carrier* (*utp-order DES*) = ⟦*H1-H2*⟧
apply (*unfold-locales*)
apply (*simp-all add*: *des-hcond-def utp-order-def H1-idem H2-idem*)
apply (*rule-tac x*=$\bigsqcup_D$ *A* **in** *exI*)
apply (*auto simp add*: *least-def Upper-def*)
using *Inf-lower* **apply** *blast*
apply (*simp add*: *Ball-Collect UINF-H1-H2-closed*)
apply (*meson Ball-Collect Inf-greatest*)
apply (*rule-tac x*=$\bigsqcap_D$ *A* **in** *exI*)
apply (*case-tac A* = {})
apply (*auto simp add*: *greatest-def Lower-def*)
using *design-sup-H1-H2-closed* **apply** *fastforce*
apply (*metis H1-below-top Healthy-def'*)
using *Sup-upper* **apply** *blast*
apply (*metis* (*no-types*) *USUP-H1-H2-closed contra-subsetD emptyE mem-Collect-eq*)
apply (*meson Ball-Collect Sup-least*)
done

abbreviation *design-lfp* :: - ⇒ - (μ_D) **where**
μ_D $F \equiv \mu_{\textit{utp-order DES}}$ F

abbreviation *design-gfp* :: - ⇒ - (ν_D) **where**
ν_D $F \equiv \nu_{\textit{utp-order DES}}$ F

end

13 Concurrent programming

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

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

13.1 Design parallel composition

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

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

lemma *design-par-is-H1-H2*: $(P \parallel Q)$ *is H1-H2*
  by (*simp add*: *design-par-def rdesign-is-H1-H2*)

lemma *design-par-skip-d-distl*:
  assumes *P is H1-H2 Q is H1-H2*
  shows $((P \;;;\; II_D) \parallel (Q \;;;\; II_D)) = ((P \parallel Q) \;;;\; II_D)$
proof –
  obtain P_1 P_2 **where** P: $P = P_1 \vdash_r P_2$
    by (*metis H1-H2-commute H1-H2-is-rdesign H2-idem Healthy-def assms*(*1*))
  moreover obtain Q_1 Q_2 **where** Q: $Q = Q_1 \vdash_r Q_2$
   by (*metis H1-H2-commute H1-H2-is-rdesign H2-idem Healthy-def assms*(*2*))
  moreover have $(((P_1 \vdash_r P_2) \;;;\; II_D) \parallel ((Q_1 \vdash_r Q_2) \;;;\; II_D)) = (((P_1 \vdash_r P_2) \parallel (Q_1 \vdash_r Q_2)) \;;;\; II_D)$
    by (*simp add*: *design-par-def skip-d-def rdesign-composition*, *rel-tac*)
  ultimately show *?thesis*
    by *simp*
qed

lemma *design-par-H3-closure*:
  assumes *P is H1-H3 Q is H1-H3*
  shows $(P \parallel Q)$ *is H3*
  using *assms*
  by (*simp add*: *H3-unrest-out-alpha design-par-def precond-right-unit rdesign-H3-iff-pre seqr-pre-out*)

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

lemma *parallel-choice-distr*:
  $(P \sqcap Q) \parallel R = ((P \parallel R) \sqcap (Q \parallel R))$
  by (*simp add*: *design-par-def rdesign-choice conj-assoc inf-left-commute inf-sup-distrib2*)

lemma *parallel-condr-distr*:
  $(P \triangleleft \lceil b \rceil_D \triangleright Q) \parallel R = ((P \parallel R) \triangleleft \lceil b \rceil_D \triangleright (Q \parallel R))$
  by (*simp add*: *design-par-def rdesign-def alpha cond-conj-distr conj-comm design-condr*)

13.2 Parallel by merge

We describe the partition of a state space into two pieces.

type-synonym $'\alpha$ *partition* $= '\alpha \times '\alpha$

definition *left-uvar* $x = x ;_L fst_L ;_L snd_L$

definition *right-uvar* $x = x ;_L snd_L ;_L snd_L$

declare *left-uvar-def* [*upred-defs*]

declare *right-uvar-def* [*upred-defs*]

Extract the ith element of the second part

definition *ind-uvar* i $x = x ;_L$ *list-lens* $i ;_L snd_L ;_L$ *des-lens*

definition *pre-uvar* $x = x ;_L fst_L$

definition *in-ind-uvar* i x = *in-var* (*ind-uvar* i x)

definition *out-ind-uvar* i x = *out-var* (*ind-uvar* i x)

definition *in-pre-uvar* x = *in-var* (*pre-uvar* x)

definition *out-pre-uvar x* = *out-var* (*pre-uvar x*)

definition *in-ind-uexpr i x* = *var* (*in-ind-uvar i x*)

definition *out-ind-uexpr i x* = *var* (*out-ind-uvar i x*)

definition *in-pre-uexpr x* = *var* (*in-pre-uvar x*)

definition *out-pre-uexpr x* = *var* (*out-pre-uvar x*)

declare *ind-uvar-def* [*upred-defs*]
declare *pre-uvar-def* [*upred-defs*]

declare *in-ind-uvar-def* [*upred-defs*]
declare *out-ind-uvar-def* [*upred-defs*]

declare *in-ind-uexpr-def* [*upred-defs*]
declare *out-ind-uexpr-def* [*upred-defs*]

declare *in-pre-uexpr-def* [*upred-defs*]
declare *out-pre-uexpr-def* [*upred-defs*]

lemma *left-uvar-indep-right-uvar* [*simp*]:
left-uvar x $\bowtie$ *right-uvar y*
apply (*simp add*: *left-uvar-def right-uvar-def lens-comp-assoc*[*THEN sym*])
apply (*metis in-out-indep in-var-def lens-indep-left-comp out-var-def out-var-indep uvar-des-lens vwb-lens-mwb*)
done

lemma *right-uvar-indep-left-uvar* [*simp*]:
right-uvar x $\bowtie$ *left-uvar y*
by (*simp add*: *lens-indep-sym*)

lemma *left-uvar* [*simp*]: *uvar x* $\Longrightarrow$ *uvar* (*left-uvar x*)
by (*simp add*: *left-uvar-def comp-vwb-lens fst-vwb-lens snd-vwb-lens*)

lemma *right-uvar* [*simp*]: *uvar x* $\Longrightarrow$ *uvar* (*right-uvar x*)
by (*simp add*: *right-uvar-def comp-vwb-lens fst-vwb-lens snd-vwb-lens*)

lemma *ind-uvar-indep* [*simp*]:
$[\![$*mwb-lens x*; $i \neq j]\!] \Longrightarrow$ *ind-uvar i x* $\bowtie$ *ind-uvar j x*
apply (*simp add*: *ind-uvar-def lens-comp-assoc*[*THEN sym*])
apply (*metis lens-indep-left-comp lens-indep-right-comp list-lens-indep out-var-def out-var-indep uvar-des-lens vwb-lens-mwb*)
done

lemma *ind-uvar-semi-uvar* [*simp*]:
semi-uvar x $\Longrightarrow$ *semi-uvar* (*ind-uvar i x*)
by (*auto intro*!: *comp-mwb-lens list-mwb-lens simp add*: *ind-uvar-def snd-vwb-lens*)

lemma *in-ind-uvar-semi-uvar* [*simp*]:
semi-uvar x $\Longrightarrow$ *semi-uvar* (*in-ind-uvar i x*)
by (*simp add*: *in-ind-uvar-def*)

lemma *out-ind-uvar-semi-uvar* [*simp*]:
semi-uvar x $\Longrightarrow$ *semi-uvar* (*out-ind-uvar i x*)

by (*simp add*: *out-ind-uvar-def*)

declare *id-vwb-lens* [*simp*]

syntax
  -svarpre :: *svid* ⇒ *svid* (*-*$_<$ [*999*] *999*)
  -svarleft :: *svid* ⇒ *svid* (*0*−*-* [*999*] *999*)
  -svarright :: *svid* ⇒ *svid* (*1*−*-* [*999*] *999*)

translations
  -svarpre x == *CONST pre-uvar x*
  -svarleft x == *CONST left-uvar x*
  -svarright x == *CONST right-uvar x*

type-synonym $'\alpha$ *merge* = ($'\alpha$ × $'\alpha$ *partition*, $'\alpha$) *relation-d*

Separating simulations. I assume that the value of ok' should track the value of n.ok'.

definition *U0* = (*true* $\vdash_r$ ($\$0{-}\Sigma\,' =_u \$\Sigma \wedge \$\Sigma_<\,' =_u \$\Sigma$))

definition *U1* = (*true* $\vdash_r$ ($\$1{-}\Sigma\,' =_u \$\Sigma \wedge \$\Sigma_<\,' =_u \$\Sigma$))

declare *U0-def* [*upred-defs*]
declare *U1-def* [*upred-defs*]

The following implementation of parallel by merge is less general than the book version, in that it does not properly partition the alphabet into two disjoint segments. We could actually achieve this specifying lenses into the larger alphabet, but this would complicate the definition of programs. May reconsider later.

definition *par-by-merge* ::
  $'\alpha$ *hrelation-d* ⇒ $'\alpha$ *merge* ⇒ $'\alpha$ *hrelation-d* ⇒ $'\alpha$ *hrelation-d* (**infixr** $\|_-$ *85*)
where $P \parallel_M Q$ = ((((P ;; *U0*) ‖ (Q ;; *U1*))) ;; M)

definition $swap_m$ = *true* $\vdash_r$ ($0{-}\Sigma, 1{-}\Sigma := \&1{-}\Sigma, \&0{-}\Sigma$)

declare *One-nat-def* [*simp del*]

declare $swap_m$*-def* [*upred-defs*]

lemma *U0-H1-H2*: *U0 is H1-H2*
  by (*simp add*: *U0-def rdesign-is-H1-H2*)

lemma *U0-swap*: (*U0* ;; $swap_m$) = *U1*
  apply (*simp add*: *U0-def* $swap_m$*-def rdesign-composition*)
  apply (*subst seqr-and-distl-uinj*)
  using *assigns-r-swap-uinj id-vwb-lens left-uvar right-uvar* **apply** *fastforce*
  apply (*rel-tac*)
  apply (*metis prod.collapse*)+
done

lemma *U1-H1-H2*: *U1 is H1-H2*
  by (*simp add*: *U1-def rdesign-is-H1-H2*)

lemma *U1-swap*: (*U1* ;; $swap_m$) = *U0*
  apply (*simp add*: *U1-def* $swap_m$*-def rdesign-composition*)
  apply (*subst seqr-and-distl-uinj*)

using *assigns-r-swap-uinj id-vwb-lens left-uvar right-uvar* **apply** *fastforce*
apply (*rel-tac*)
apply (*metis prod.collapse*)+
done

lemma *swap-merge-par-distl*:
assumes P *is H1-H2* Q *is H1-H2*
shows $((P \parallel Q) ;; swap_m) = (P ;; swap_m) \parallel (Q ;; swap_m)$
proof −
obtain P_1 P_2 **where** P: $P = P_1 \vdash_r P_2$
by (*metis H1-H2-commute H1-H2-is-rdesign H2-idem Healthy-def assms*(*1*))
obtain Q_1 Q_2 **where** Q: $Q = Q_1 \vdash_r Q_2$
by (*metis H1-H2-commute H1-H2-is-rdesign H2-idem Healthy-def assms*(*2*))
have $(((P_1 \vdash_r P_2) \parallel (Q_1 \vdash_r Q_2)) ;; swap_m) =$
$(\neg (\neg P_1 \vee \neg Q_1 ;; true)) \vdash_r ((P_1 \Rightarrow P_2) \wedge (Q_1 \Rightarrow Q_2) ;; \langle[\&0{-}\Sigma \mapsto_s \&1{-}\Sigma, \&1{-}\Sigma \mapsto_s \&0{-}\Sigma]\rangle_a)$
by (*simp add*: *design-par-def* $swap_m$*-def rdesign-composition*)
also have ... = $(\neg (\neg P_1 \vee \neg Q_1 ;; true)) \vdash_r (((P_1 \Rightarrow P_2) ;; \langle[\&0{-}\Sigma \mapsto_s \&1{-}\Sigma, \&1{-}\Sigma \mapsto_s \&0{-}\Sigma]\rangle_a) \wedge ((Q_1 \Rightarrow Q_2) ;; \langle[\&0{-}\Sigma \mapsto_s \&1{-}\Sigma, \&1{-}\Sigma \mapsto_s \&0{-}\Sigma]\rangle_a))$
apply (*subst seqr-and-distl-uinj*)
using *assigns-r-swap-uinj id-vwb-lens left-uvar right-uvar* **apply** *fastforce*
apply (*simp*)
done

also have ... = $((P_1 \vdash_r P_2) ;; swap_m) \parallel ((Q_1 \vdash_r Q_2) ;; swap_m)$
by (*simp add*: *design-par-def* $swap_m$*-def rdesign-composition*, *rel-tac*)

finally show *?thesis*
using P Q **by** *blast*
qed

lemma *par-by-merge-left-zero*:
assumes M *is H1*
shows $true \parallel_M P = true$
proof −
have $true \parallel_M P = ((true ;; U0) \parallel (P ;; U1) ;; M)$ (**is** - = $((?P \parallel ?Q) ;; ?M)$)
by (*simp add*: *par-by-merge-def*)
moreover have $?P = true$
by (*rel-tac*, *meson alpha-d.select-convs*(*1*))
ultimately show *?thesis*
by (*metis H1-left-zero assms parallel-comm parallel-zero*)
qed

lemma *par-by-merge-right-zero*:
assumes M *is H1*
shows $P \parallel_M true = true$
proof −
have $P \parallel_M true = ((P ;; U0) \parallel (true ;; U1) ;; M)$ (**is** - = $((?P \parallel ?Q) ;; ?M)$)
by (*simp add*: *par-by-merge-def*)
moreover have $?Q = true$
by (*rel-tac*, *meson alpha-d.select-convs*(*1*))
ultimately show *?thesis*
by (*metis H1-left-zero assms parallel-comm parallel-zero*)
qed

lemma *par-by-merge-commute*:
assumes P *is H1-H2* Q *is H1-H2* $M = (swap_m\ ;;\ M)$
shows $P \parallel_M Q = Q \parallel_M P$
proof −
have $P \parallel_M Q = (((P\ ;;\ U0) \parallel (Q\ ;;\ U1))\ ;;\ M)$
by (*simp add*: *par-by-merge-def*)
also have ... $= ((((P\ ;;\ U0) \parallel (Q\ ;;\ U1))\ ;;\ swap_m)\ ;;\ M)$
by (*metis assms*(*3*) *seqr-assoc*)
also have ... $= (((P\ ;;\ U0\ ;;\ swap_m) \parallel (Q\ ;;\ U1\ ;;\ swap_m))\ ;;\ M)$
by (*simp add*: *U0-def U1-def assms*(*1*) *assms*(*2*) *rdesign-is-H1-H2 seq-r-H1-H2-closed seqr-assoc swap-merge-par-distl*)
also have ... $= (((P\ ;;\ U1) \parallel (Q\ ;;\ U0))\ ;;\ M)$
by (*simp add*: *U0-swap U1-swap*)
also have ... $= Q \parallel_M P$
by (*simp add*: *par-by-merge-def parallel-comm*)
finally show *?thesis* **.**
qed

lemma *par-by-merge-mono-1*:
assumes $P_1 \sqsubseteq P_2$ P_1 *is H1-H2* P_2 *is H1-H2*
shows $P_1 \parallel_M Q \sqsubseteq P_2 \parallel_M Q$
using *assms*
by (*auto intro*:*seqr-mono parallel-mono-1 seq-r-H1-H2-closed U0-H1-H2 U1-H1-H2 simp add*: *par-by-merge-def*)

lemma *par-by-merge-mono-2*:
assumes $Q_1 \sqsubseteq Q_2$ Q_1 *is H1-H2* Q_2 *is H1-H2*
shows $(P \parallel_M Q_1) \sqsubseteq (P \parallel_M Q_2)$
using *assms*
by (*auto intro*:*seqr-mono parallel-mono-2 seq-r-H1-H2-closed U0-H1-H2 U1-H1-H2 simp add*: *par-by-merge-def*)

end

14 Reactive processes

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

14.1 Preliminaries

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

fun *list-diff* $::'\alpha$ *list* $\Rightarrow$ $'\alpha$ *list* $\Rightarrow$ $'\alpha$ *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*]: $l\ @\ t = m \Longrightarrow m - l = t$
by (*auto*)

lemma *prefix-subst1* [*simp*]: $m = l\ @\ t \Longrightarrow m - l = t$
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'* = *rp-wait* :: *bool*
rp-tr :: *'ϑ trace*
rp-ref :: *'ϑ refusal*

type-synonym (*'ϑ*, *'α*) *alpha-rp-scheme* = (*'ϑ*, *'α*) *alpha-rp'-scheme alpha-d-scheme*

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

definition $wait_r$ = *VAR rp-wait*
definition tr_r = *VAR rp-tr*
definition ref_r = *VAR rp-ref*
definition [*upred-defs*]: Σ_r = *VAR more*

declare $wait_r$*-def* [*upred-defs*]
declare tr_r*-def* [*upred-defs*]
declare ref_r*-def* [*upred-defs*]
declare Σ_r*-def* [*upred-defs*]

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

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

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

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

definition *wait* = $(wait_r ;_L \Sigma_D)$
definition *tr* = $(tr_r ;_L \Sigma_D)$
definition *ref* = $(ref_r ;_L \Sigma_D)$
definition [*upred-defs*]: $\Sigma_R = (\Sigma_r ;_L \Sigma_D)$

lemma *wait-uvar* [*simp*]: *uvar wait*
by (*simp add*: *comp-vwb-lens wait-def*)

lemma *tr-uvar* [*simp*]: *uvar tr*
by (*simp add*: *comp-vwb-lens tr-def*)

lemma *ref-uvar* [*simp*]: *uvar ref*
by (*simp add*: *comp-vwb-lens ref-def*)

lemma *rea-lens-uvar* [*simp*]: *uvar* Σ_R
by (*simp add*: Σ_R*-def comp-vwb-lens*)

lemma *rea-lens-under-des-lens*: $\Sigma_R \subseteq_L \Sigma_D$
by (*simp add*: Σ_R*-def lens-comp-lb*)

lemma *rea-lens-indep-ok* [*simp*]: $\Sigma_R \bowtie$ *ok ok* $\bowtie \Sigma_R$
using *ok-indep-des-lens*(*2*) *rea-lens-under-des-lens sublens-pres-indep* **apply** *blast*
using *lens-indep-sym ok-indep-des-lens*(*2*) *rea-lens-under-des-lens sublens-pres-indep* **apply** *blast*
done

declare *wait-def* [*upred-defs*]
declare *tr-def* [*upred-defs*]
declare *ref-def* [*upred-defs*]

lemma *tr-ok-indep* [*simp*]: *tr* $\bowtie$ *ok ok* $\bowtie$ *tr*
by (*simp-all add*: *lens-indep-left-ext lens-indep-sym tr-def*)

lemma *wait-ok-indep* [*simp*]: *wait* $\bowtie$ *ok ok* $\bowtie$ *wait*
by (*simp-all add*: *lens-indep-left-ext lens-indep-sym wait-def*)

lemma *ref-ok-indep* [*simp*]: *ref* $\bowtie$ *ok ok* $\bowtie$ *ref*
by (*simp-all add*: *lens-indep-left-ext lens-indep-sym ref-def*)

lemma tr_r*-*$wait_r$*-indep* [*simp*]: $tr_r \bowtie wait_r$ $wait_r \bowtie tr_r$
by (*auto intro*!:*lens-indepI simp add*: tr_r*-def* $wait_r$*-def*)

lemma *tr-wait-indep* [*simp*]: *tr* $\bowtie$ *wait wait* $\bowtie$ *tr*
by (*auto intro*: *lens-indep-left-comp simp add*: *tr-def wait-def*)

lemma ref_r*-*$wait_r$*-indep* [*simp*]: $ref_r \bowtie wait_r$ $wait_r \bowtie ref_r$
by (*auto intro*!:*lens-indepI simp add*: ref_r*-def* $wait_r$*-def*)

lemma *ref-wait-indep* [*simp*]: *ref* $\bowtie$ *wait wait* $\bowtie$ *ref*
by (*auto intro*: *lens-indep-left-comp simp add*: *ref-def wait-def*)

lemma tr_r*-*ref_r*-indep* [*simp*]: $ref_r \bowtie tr_r$ $tr_r \bowtie ref_r$
by (*auto intro*!:*lens-indepI simp add*: ref_r*-def* tr_r*-def*)

lemma *tr-ref-indep* [*simp*]: *ref* $\bowtie$ *tr tr* $\bowtie$ *ref*
by (*auto intro*: *lens-indep-left-comp simp add*: *ref-def tr-def*)

lemma *rea-indep-wait* [*simp*]: $\Sigma_r \bowtie wait_r$ $wait_r \bowtie \Sigma_r$
by (*auto intro*!:*lens-indepI simp add*: $wait_r$*-def* Σ_r*-def*)

lemma *rea-lens-indep-wait* [*simp*]: $\Sigma_R \bowtie$ *wait wait* $\bowtie \Sigma_R$
by (*auto intro*: *lens-indep-left-comp simp add*: *wait-def* Σ_R*-def*)

lemma *rea-indep-tr* [*simp*]: $\Sigma_r \bowtie tr_r$ $tr_r \bowtie \Sigma_r$
by (*auto intro*!:*lens-indepI simp add*: tr_r*-def* Σ_r*-def*)

lemma *rea-lens-indep-tr* [*simp*]: $\Sigma_R \bowtie$ *tr tr* $\bowtie \Sigma_R$
by (*auto intro*: *lens-indep-left-comp simp add*: *tr-def* Σ_R*-def*)

lemma *rea-indep-ref* [*simp*]: $\Sigma_r \bowtie ref_r$ $ref_r \bowtie \Sigma_r$

by (*auto intro*!:*lens-indepI simp add*: ref_r*-def* Σ_r*-def*)

lemma *rea-lens-indep-ref* [*simp*]: $\Sigma_R \bowtie ref$ $ref \bowtie \Sigma_R$
by (*auto intro*: *lens-indep-left-comp simp add*: *ref-def* Σ_R*-def*)

lemma *rea-var-ords* [*usubst*]:
$\$tr \prec_v \$tr´$ $\$wait \prec_v \$wait´$ $\$ref \prec_v \$ref´$
$\$ok \prec_v \$tr$ $\$ok´ \prec_v \$tr´$ $\$ok \prec_v \$tr´$ $\$ok´ \prec_v \$tr$
$\$ok \prec_v \$ref$ $\$ok´ \prec_v \$ref´$ $\$ok \prec_v \$ref´$ $\$ok´ \prec_v \$ref$
$\$ok \prec_v \$wait$ $\$ok´ \prec_v \$wait´$ $\$ok \prec_v \$wait´$ $\$ok´ \prec_v \$wait$
$\$tr \prec_v \$wait$ $\$tr´ \prec_v \$wait´$ $\$tr \prec_v \$wait´$ $\$tr´ \prec_v \$wait$
by (*simp-all add*: *var-name-ord-def*)

instantiation *alpha-rp'-ext* :: (*type*, *vst*) *vst*
begin
definition *vstore-lens-alpha-rp'-ext* :: *vstore* $\Longrightarrow$ ('*a*, '*b*) *alpha-rp'-scheme*
where *vstore-lens-alpha-rp'-ext* = $\mathcal{V}$ $;_L$ Σ_r
instance
by (*intro-classes*, *simp add*: *vstore-lens-alpha-rp'-ext-def comp-vwb-lens*)
end

abbreviation *wait-f*::('ϑ, 'α, 'β) *relation-rp* $\Rightarrow$ ('ϑ, 'α, 'β) *relation-rp*
where *wait-f R* $\equiv$ $R[\![false/\$wait]\!]$

abbreviation *wait-t*::('ϑ, 'α, 'β) *relation-rp* $\Rightarrow$ ('ϑ, 'α, 'β) *relation-rp*
where *wait-t R* $\equiv$ $R[\![true/\$wait]\!]$

syntax
-wait-f :: *logic* $\Rightarrow$ *logic* (-$_f$ [*1000*] *1000*)
-wait-t :: *logic* $\Rightarrow$ *logic* (-$_t$ [*1000*] *1000*)

translations
$P\ _f$ $\rightleftharpoons$ *CONST usubst* (*CONST subst-upd CONST id* (*CONST ivar CONST wait*) *false*) *P*
$P\ _t$ $\rightleftharpoons$ *CONST usubst* (*CONST subst-upd CONST id* (*CONST ivar CONST wait*) *true*) *P*

definition *lift-rea* :: ('α, 'β) *relation* $\Rightarrow$ ('ϑ, 'α, 'β) *relation-rp* ($\lceil$-$\rceil_R$) **where**
[*upred-defs*]: $\lceil P\rceil_R = P \oplus_p (\Sigma_R \times_L \Sigma_R)$

definition *drop-rea* :: ('ϑ, 'α, 'β) *relation-rp* $\Rightarrow$ ('α, 'β) *relation* ($\lfloor$-$\rfloor_R$) **where**
[*upred-defs*]: $\lfloor P\rfloor_R = P \restriction_p (\Sigma_R \times_L \Sigma_R)$

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

14.2 Reactive lemmas

lemma *unrest-tr-lift-rea* [*unrest*]:
$\$tr \sharp \lceil P\rceil_R$ $\$tr´ \sharp \lceil P\rceil_R$
by (*pred-tac*)+

lemma *tr'-minus-tr-prefix* [*simp*]:
$(\$tr´ - \$tr =_u []_u) = (\$tr =_u \$tr´)$
apply (*pred-tac*)
using *list-minus-anhil* **apply** *fastforce*
done

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*)

14.3 R1: Events cannot be undone

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

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

lemma *R1-mono*: $P \sqsubseteq Q \Longrightarrow R1(P) \sqsubseteq R1(Q)$
by *pred-tac*

lemma *R1-conj*: $R1(P \land Q) = (R1(P) \land R1(Q))$
by *pred-tac*

lemma *R1-disj*: $R1(P \lor Q) = (R1(P) \lor R1(Q))$
by *pred-tac*

lemma *R1-USUP*:
$R1(\bigsqcap\ i \in A \cdot P(i)) = (\bigsqcap\ i \in A \cdot R1(P(i)))$
by (*rel-tac*)

lemma *R1-UINF*:
assumes $A \neq \{\}$
shows $R1(\bigsqcup\ i \in A \cdot P(i)) = (\bigsqcup\ i \in A \cdot R1(P(i)))$
using *assms* **by** (*rel-tac*)

lemma *R1-extend-conj*: $R1(P \land Q) = (R1(P) \land Q)$
by *pred-tac*

lemma *R1-extend-conj'*: $R1(P \land Q) = (P \land R1(Q))$
by *pred-tac*

lemma *R1-cond*: $R1(P \triangleleft b \triangleright Q) = (R1(P) \triangleleft b \triangleright R1(Q))$
by *rel-tac*

lemma *R1-negate-R1*: $R1(\neg R1(P)) = R1(\neg P)$
by *pred-tac*

lemma *R1-wait-true*: $(R1\ P)_t = R1(P)_t$
by *pred-tac*

lemma *R1-wait-false*: $(R1\ P)\ _f = R1(P)\ _f$
by *pred-tac*

lemma *R1-skip*: $R1(II) = II$
by *rel-tac*

lemma *R1-skip-rea*: $R1(II_r) = II_r$
by *rel-tac*

lemma *R1-by-refinement*:
P *is* $R1 \longleftrightarrow ((\$tr \leq_u \$tr') \sqsubseteq P)$
by *rel-tac*

lemma *tr-le-trans*:
$(\$tr \leq_u \$tr' ;; \$tr \leq_u \$tr') = (\$tr \leq_u \$tr')$
by (*rel-tac*, *metis alpha-d.select-convs*(*2*) *alpha-rp'.select-convs*(*2*) *eq-refl*)

lemma *R1-seqr*:
$R1(R1(P) ;; R1(Q)) = (R1(P) ;; R1(Q))$
by (*rel-tac*)

lemma *R1-seqr-closure*:
assumes *P is R1 Q is R1*
shows (*P* ;; *Q*) *is R1*
using *assms* **unfolding** *R1-by-refinement*
by (*metis seqr-mono tr-le-trans*)

lemma *R1-ok'-true*: $(R1(P))^t = R1(P^t)$
by *pred-tac*

lemma *R1-ok'-false*: $(R1(P))^f = R1(P^f)$
by *pred-tac*

lemma *R1-ok-true*: $(R1(P))\llbracket true/\$ok \rrbracket = R1(P\llbracket true/\$ok \rrbracket)$
by *pred-tac*

lemma *R1-ok-false*: $(R1(P))\llbracket false/\$ok \rrbracket = R1(P\llbracket false/\$ok \rrbracket)$
by *pred-tac*

lemma *seqr-R1-true-right*: $((P ;; R1(true)) \vee P) = (P ;; (\$tr \leq_u \$tr'))$
by *rel-tac*

lemma *R1-extend-conj-unrest*: $\llbracket \$tr \sharp Q; \$tr' \sharp Q \rrbracket \Longrightarrow R1(P \wedge Q) = (R1(P) \wedge Q)$
by *pred-tac*

lemma *R1-extend-conj-unrest'*: $\llbracket \$tr \sharp P; \$tr' \sharp P \rrbracket \Longrightarrow R1(P \wedge Q) = (P \wedge R1(Q))$
by *pred-tac*

lemma *R1-tr'-eq-tr*: $R1(\$tr' =_u \$tr) = (\$tr' =_u \$tr)$
by (*rel-tac*)

lemma *R1-H2-commute*: $R1(H2(P)) = H2(R1(P))$
by (*simp add*: *H2-split R1-def usubst*, *rel-tac*)

14.4 R2

definition *R2a-def* [*upred-defs*]: $R2a\ (P) = (\bigsqcap\ s \cdot P\llbracket \ll s \gg, \ll s \gg \hat{\ }_u(\$tr' - \$tr)/\$tr,\$tr' \rrbracket)$
definition *R2s-def* [*upred-defs*]: $R2s\ (P) = (P\llbracket \langle\rangle/\$tr \rrbracket\llbracket (\$tr' - \$tr)/\$tr' \rrbracket)$
definition *R2-def* [*upred-defs*]: $R2(P) = R1(R2s(P))$
definition *R2c-def* [*upred-defs*]: $R2c(P) = (R2s(P) \triangleleft R1(true) \triangleright P)$

lemma *R2a-R2s*: $R2a(R2s(P)) = R2s(P)$
by *rel-tac*

lemma *R2s-R2a*: $R2s(R2a(P)) = R2a(P)$
by *rel-tac*

lemma *R2a-equiv-R2s*: P *is R2a* $\longleftrightarrow$ P *is R2s*
by (*metis Healthy-def' R2a-R2s R2s-R2a*)

lemma *R2s-idem*: $R2s(R2s(P)) = R2s(P)$
by (*pred-tac*)

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

lemma *R2-mono*: $P \sqsubseteq Q \Longrightarrow R2(P) \sqsubseteq R2(Q)$
by (*pred-tac*)

lemma *R2s-conj*: $R2s(P \wedge Q) = (R2s(P) \wedge R2s(Q))$
by (*pred-tac*)

lemma *R2-conj*: $R2(P \wedge Q) = (R2(P) \wedge R2(Q))$
by (*pred-tac*)

lemma *R2s-disj*: $R2s(P \vee Q) = (R2s(P) \vee R2s(Q))$
by *pred-tac*

lemma *R2s-USUP*:
$R2s(\bigsqcap\ i \in A \cdot P(i)) = (\bigsqcap\ i \in A \cdot R2s(P(i)))$
by (*simp add*: *R2s-def usubst*)

lemma *R2s-UINF*:
$R2s(\bigsqcup\ i \in A \cdot P(i)) = (\bigsqcup\ i \in A \cdot R2s(P(i)))$
by (*simp add*: *R2s-def usubst*)

lemma *R2-disj*: $R2(P \vee Q) = (R2(P) \vee R2(Q))$
by (*pred-tac*)

lemma *R2s-not*: $R2s(\neg\ P) = (\neg\ R2s(P))$
by *pred-tac*

lemma *R2s-condr*: $R2s(P \triangleleft b \triangleright Q) = (R2s(P) \triangleleft R2s(b) \triangleright R2s(Q))$
by *rel-tac*

lemma *R2-condr*: $R2(P \triangleleft b \triangleright Q) = (R2(P) \triangleleft R2(b) \triangleright R2(Q))$
by *rel-tac*

lemma *R2-condr'*: $R2(P \triangleleft b \triangleright Q) = (R2(P) \triangleleft R2s(b) \triangleright R2(Q))$
by *rel-tac*

lemma *R2s-ok*: $R2s(\$ok) = \$ok$
by *rel-tac*

lemma *R2s-ok'*: $R2s(\$ok\acute{}) = \$ok\acute{}$
by *rel-tac*

lemma *R2s-wait*: $R2s(\$wait) = \$wait$
by *rel-tac*

lemma *R2s-wait'*: $R2s(\$wait\acute{}) = \$wait\acute{}$
by *rel-tac*

lemma *R2s-tr'-eq-tr*: $R2s(\$tr\acute{} =_u \$tr) = (\$tr\acute{} =_u \$tr)$

apply (*pred-tac*)
using *list-minus-anhil* **apply** *blast*
done

lemma *R2s-true*: $R2s(true) = true$
by *pred-tac*

lemma *true-is-R2s*:
true is R2s
by (*simp add*: *Healthy-def R2s-true*)

lemma *R2s-lift-rea*: $R2s(\lceil P \rceil_R) = \lceil P \rceil_R$
by (*simp add*: *R2s-def usubst unrest*)

lemma *R2s-skip-r*: $R2s(II) = II$
proof –
have $R2s(II) = R2s(\$tr\acute{} =_u \$tr \wedge II \restriction_\alpha tr)$
by (*subst skip-r-unfold*[*of tr*], *simp-all*)
also have ... $= (R2s(\$tr\acute{} =_u \$tr) \wedge II \restriction_\alpha tr)$
by (*simp add*: *R2s-def usubst unrest*)
also have ... $= (\$tr\acute{} =_u \$tr \wedge II \restriction_\alpha tr)$
by (*simp add*: *R2s-tr′-eq-tr*)
finally show *?thesis*
by (*subst skip-r-unfold*[*of tr*], *simp-all*)
qed

lemma *R2-skip*: $R2(II) = II$
by (*simp add*: *R1-skip R2-def R2s-skip-r*)

lemma *R2-skip-rea*: $R2(II_r) = II_r$
apply (*simp add*: *skip-rea-def R2-disj R2-skip*)
apply (*simp add*: *R2-def R2s-conj R2s-not R2s-ok R1-extend-conj′*)
apply (*rel-tac*)
done

lemma *R2-tr-prefix*: $R2(\$tr \le_u \$tr\acute{}) = (\$tr \le_u \$tr\acute{})$
by (*pred-tac*)

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*

lemma *R2-seqr-form*:
shows $(R2(P) ;; R2(Q)) =$
$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P[\![\langle\rangle/\$tr]\!][\![\ll tt_1 \gg/\$tr\acute{}]\!]) ;; (Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2 \gg/\$tr\acute{}]\!]))$
$\wedge (\$tr\acute{} =_u \$tr\ \hat{}_u \ll tt_1 \gg\ \hat{}_u \ll tt_2 \gg))$
proof –
have $(R2(P) ;; R2(Q)) = (\exists\ tr_0 \cdot (R2(P))[\![\ll tr_0 \gg/\$tr\acute{}]\!] ;; (R2(Q))[\![\ll tr_0 \gg/\$tr]\!])$
by (*subst seqr-middle*[*of tr*], *simp-all*)

also have ... =
$(\exists\ tr_0 \cdot \exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!] \wedge \ll tr_0\gg =_u \$tr\ \hat{}_u \ll tt_1\gg)\ ;;$
$(Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!] \wedge \$tr\acute{} =_u \ll tr_0\gg\ \hat{}_u \ll tt_2\gg)))$
by (*simp add*: *R2-form usubst unrest uquant-lift*, *rel-tac*)
also have ... =
$(\exists\ tr_0 \cdot \exists\ tt_1 \cdot \exists\ tt_2 \cdot ((\ll tr_0\gg =_u \$tr\ \hat{}_u \ll tt_1\gg \wedge P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!])\ ;;$
$(Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!] \wedge \$tr\acute{} =_u \ll tr_0\gg\ \hat{}_u \ll tt_2\gg)))$
by (*simp add*: *conj-comm*)
also have ... =
$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot \exists\ tr_0 \cdot ((P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!])\ ;;\ (Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!]))$
$\wedge \ll tr_0\gg =_u \$tr\ \hat{}_u \ll tt_1\gg \wedge \$tr\acute{} =_u \ll tr_0\gg\ \hat{}_u \ll tt_2\gg)$
by *rel-tac*
also have ... =
$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!])\ ;;\ (Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!]))$
$\wedge (\exists\ tr_0 \cdot \ll tr_0\gg =_u \$tr\ \hat{}_u \ll tt_1\gg \wedge \$tr\acute{} =_u \ll tr_0\gg\ \hat{}_u \ll tt_2\gg))$
by *rel-tac*
also have ... =
$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!])\ ;;\ (Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!]))$
$\wedge (\$tr\acute{} =_u \$tr\ \hat{}_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg))$
by *rel-tac*
finally show *?thesis* **.**
qed

lemma *R2-seqr-distribute*:
fixes P :: $('\vartheta,'\alpha,'\beta)$ *relation-rp* **and** Q :: $('\vartheta,'\beta,'\gamma)$ *relation-rp*
shows $R2(R2(P)\ ;;\ R2(Q)) = (R2(P)\ ;;\ R2(Q))$
proof −
have $R2(R2(P)\ ;;\ R2(Q)) =$
$((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!]\ ;;\ Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!])[\![(\$tr\acute{} - \$tr)/\$tr\acute{}]\!]$
$\wedge \$tr\acute{} - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg) \wedge \$tr\acute{} \geq_u \$tr)$
by (*simp add*: *R2-seqr-form*, *simp add*: *R2s-def usubst unrest*, *rel-tac*)
also have ... =
$((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!]\ ;;\ Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!])[\![(\ll tt_1\gg\ \hat{}_u \ll tt_2\gg)/\$tr\acute{}]\!]$
$\wedge \$tr\acute{} - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg) \wedge \$tr\acute{} \geq_u \$tr)$
by (*subst subst-eq-replace*, *simp*)
also have ... =
$((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!]\ ;;\ Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!])$
$\wedge \$tr\acute{} - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg) \wedge \$tr\acute{} \geq_u \$tr)$
by (*rel-tac*)
also have ... =
$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!]\ ;;\ Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!])$
$\wedge (\$tr\acute{} - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg \wedge \$tr\acute{} \geq_u \$tr))$
by *pred-tac*
also have ... =
$((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![\langle\rangle/\$tr]\!][\![\ll tt_1\gg/\$tr\acute{}]\!]\ ;;\ Q[\![\langle\rangle/\$tr]\!][\![\ll tt_2\gg/\$tr\acute{}]\!])$
$\wedge \$tr\acute{} =_u \$tr\ \hat{}_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg))$
proof −
have $\bigwedge\ tt_1\ tt_2.\ (((\$tr\acute{} - \$tr =_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg) \wedge \$tr\acute{} \geq_u \$tr) :: ('\vartheta,'\alpha,'\gamma)$ *relation-rp*)
$= (\$tr\acute{} =_u \$tr\ \hat{}_u \ll tt_1\gg\ \hat{}_u \ll tt_2\gg)$
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 *R2-seqr-closure*:
  assumes P *is* $R2$ Q *is* $R2$
  shows $(P\ ;;\ Q)$ *is* $R2$
  by (*metis Healthy-def′ R2-seqr-distribute assms*(*1*) *assms*(*2*)))

lemma *R1-R2-commute*:
  $R1(R2(P)) = R2(R1(P))$
  by *pred-tac*

lemma *R2-R1-form*: $R2(R1(P)) = R1(R2s(P))$
  by (*rel-tac*)

lemma *R2s-H1-commute*:
  $R2s(H1(P)) = H1(R2s(P))$
  by *rel-tac*

lemma *R2s-H2-commute*:
  $R2s(H2(P)) = H2(R2s(P))$
  by (*simp add*: *H2-split R2s-def usubst*)

lemma *R2-R1-seq-drop-left*:
  $R2(R1(P)\ ;;\ R1(Q)) = R2(P\ ;;\ R1(Q))$
  by *rel-tac*

lemma *R2c-and*: $R2c(P \wedge Q) = (R2c(P) \wedge R2c(Q))$
  by (*rel-tac*)

lemma *R2c-disj*: $R2c(P \vee Q) = (R2c(P) \vee R2c(Q))$
  by (*rel-tac*)

lemma *R2c-not*: $R2c(\neg\ P) = (\neg\ R2c(P))$
  by (*rel-tac*)

lemma *R2c-ok*: $R2c(\$ok) = (\$ok)$
  by (*rel-tac*)

lemma *R2c-wait*: $R2c(\$wait) = \$wait$
  by (*rel-tac*)

lemma *R2c-idem*: $R2c(R2c(P)) = R2c(P)$
  by (*rel-tac*)

lemma *R1-R2c-commute*: $R1(R2c(P)) = R2c(R1(P))$
  by (*rel-tac*)

lemma *R1-R2c-is-R2*: $R1(R2c(P)) = R2(P)$
  by (*rel-tac*)

lemma *R2c-seq*: $R2c(R2(P)\ ;;\ R2(Q)) = (R2(P)\ ;;\ R2(Q))$
  by (*metis R1-R2c-commute R1-R2c-is-R2 R2-seqr-distribute R2c-idem*)

lemma *R2c-tr′-minus-tr*: $R2c(\$tr\acute{} =_u \$tr) = (\$tr\acute{} =_u \$tr)$
  apply (*rel-tac*) **using** *list-minus-anhil* **by** *blast*

lemma *R2c-condr*: $R2c(P \triangleleft b \triangleright Q) = (R2c(P) \triangleleft R2c(b) \triangleright R2c(Q))$
by (*rel-tac*)

lemma *R2c-skip-r*: $R2c(II) = II$
proof –
have $R2c(II) = R2c(\$tr' =_u \$tr \wedge II\restriction_\alpha tr)$
by (*subst skip-r-unfold*[*of tr*], *simp-all*)
also have ... $= (R2c(\$tr' =_u \$tr) \wedge II\restriction_\alpha tr)$
by (*simp add*: *R2c-def R2s-def usubst unrest*,
metis LNil-def cond-idem eq-upred-sym tr′-minus-tr-prefix)
also have ... $= (\$tr' =_u \$tr \wedge II\restriction_\alpha tr)$
by (*simp add*: *R2c-tr′-minus-tr*)
finally show *?thesis*
by (*subst skip-r-unfold*[*of tr*], *simp-all*)
qed

14.5 R3

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

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

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

lemma *R3-mono*: $P \sqsubseteq Q \Longrightarrow R3(P) \sqsubseteq R3(Q)$
by *rel-tac*

lemma *R3-conj*: $R3(P \wedge Q) = (R3(P) \wedge R3(Q))$
by *rel-tac*

lemma *R3-disj*: $R3(P \vee Q) = (R3(P) \vee R3(Q))$
by *rel-tac*

lemma *R3-USUP*:
assumes $A \neq \{\}$
shows $R3(\bigsqcap\ i \in A \cdot P(i)) = (\bigsqcap\ i \in A \cdot R3(P(i)))$
using *assms* **by** (*rel-tac*)

lemma *R3-UINF*:
assumes $A \neq \{\}$
shows $R3(\bigsqcup\ i \in A \cdot P(i)) = (\bigsqcup\ i \in A \cdot R3(P(i)))$
using *assms* **by** (*rel-tac*)

lemma *R3-condr*: $R3(P \triangleleft b \triangleright Q) = (R3(P) \triangleleft b \triangleright R3(Q))$
by *rel-tac*

lemma *R3-skipr*: $R3(II) = II$
by *rel-tac*

lemma *R3-form*: $R3(P) = ((\$wait \wedge II) \vee (\neg\ \$wait \wedge P))$
by *rel-tac*

lemma *R3-semir-form*:
$(R3(P)\ ;;\ R3(Q)) = R3(P\ ;;\ R3(Q))$

by *rel-tac*

lemma *R3-semir-closure*:
assumes *P is R3 Q is R3*
shows $(P \;;;\; Q)$ *is R3*
using *assms*
by (*metis Healthy-def′ R3-semir-form*)

lemma *R3c-semir-form*:
$(R3c(P) \;;;\; R3c(R1(Q))) = R3c(P \;;;\; R3c(R1(Q)))$
by *rel-tac*

lemma *R3c-seq-closure*:
assumes *P is R3c Q is R3c Q is R1*
shows $(P \;;;\; Q)$ *is R3c*
by (*metis Healthy-def′ R3c-semir-form assms*)

lemma *R3c-subst-wait*: $R3c(P) = R3c(P\ _f)$
by (*metis R3c-def cond-var-subst-right wait-uvar*)

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

lemma *R1-R3c-commute*: $R1(R3c(P)) = R3c(R1(P))$
by *rel-tac*

lemma *R2-R3-commute*: $R2(R3(P)) = R3(R2(P))$
by (*rel-tac*, (*smt alpha-d.surjective alpha-d.update-convs*(*2*) *alpha-rp′.surjective alpha-rp′.update-convs*(*2*) *append-Nil2 append-minus strict-prefixE*)+)

lemma *R2-R3c-commute*: $R2(R3c(P)) = R3c(R2(P))$
by (*rel-tac*, (*smt alpha-d.surjective alpha-d.update-convs*(*2*) *alpha-rp′.surjective alpha-rp′.update-convs*(*2*) *append-minus append-self-conv strict-prefixE*)+)

lemma *R1-H1-R3c-commute*:
$R1(H1(R3c(P))) = R3c(R1(H1(P)))$
by *rel-tac*

lemma *R3c-H2-commute*: $R3c(H2(P)) = H2(R3c(P))$
apply (*simp add*: *H2-split R3c-def usubst*, *rel-tac*)
apply (*metis* (*mono-tags*, *lifting*) *alpha-d.surjective alpha-d.update-convs*(*1*))+
done

lemma *R3c-idem*: $R3c(R3c(P)) = R3c(P)$
by *rel-tac*

14.6 RH laws

definition *RH-def* [*upred-defs*]: $RH(P) = R1(R2s(R3c(P)))$

lemma *RH-alt-def*:
$RH(P) = R1(R2(R3c(P)))$
by (*simp add*: *R1-idem R2-def RH-def*)

lemma *RH-alt-def′*:
$RH(P) = R2(R3c(P))$

by (*simp add*: *R2-def RH-def*)

lemma *RH-idem*:
$RH(RH(P)) = RH(P)$
by (*metis R2-R3c-commute R2-def R2-idem R3c-idem RH-def*)

lemma *RH-monotone*:
$P \sqsubseteq Q \Longrightarrow RH(P) \sqsubseteq RH(Q)$
by *rel-tac*

lemma *RH-intro*:
$[\![\ P\ is\ R1;\ P\ is\ R2;\ P\ is\ R3c\ ]\!] \Longrightarrow P\ is\ RH$
by (*simp add*: *Healthy-def′ R2-def RH-def*)

lemma *RH-seq-closure*:
assumes *P is RH Q is RH*
shows (*P* ;; *Q*) *is RH*
proof (*rule RH-intro*)
show (*P* ;; *Q*) *is R1*
by (*metis Healthy-def′ R1-seqr-closure R2-def RH-alt-def RH-def assms*(*1*) *assms*(*2*))
show (*P* ;; *Q*) *is R2*
by (*metis Healthy-def′ R2-def R2-idem R2-seqr-closure RH-def assms*(*1*) *assms*(*2*))
show (*P* ;; *Q*) *is R3c*
by (*metis Healthy-def′ R2-R3c-commute R2-def R3c-idem R3c-seq-closure RH-alt-def RH-def assms*(*1*) *assms*(*2*))
qed

lemma *RH-R2c-def*: $RH(P) = R3c(R1(R2c(P)))$
by (*simp add*: *R1-R2c-is-R2 R2-R3c-commute RH-alt-def′*)

lemma *RH-absorbs-R2c*: $RH(R2c(P)) = RH(P)$
by (*metis R1-R2-commute R1-R2c-is-R2 R1-R3c-commute R2-R3c-commute R2-idem RH-alt-def RH-alt-def′*)

end

15 Reactive designs

theory *utp-rea-designs*
imports *utp-reactive*
begin

definition *wait′-cond* :: - ⇒ - ⇒ - (**infix** ⋄ *65*) **where**
[*upred-defs*]: $P \diamond Q = (P \triangleleft \$wait' \triangleright Q)$

lemma *wait-false-design*:
$(P \vdash Q)_{\ f} = ((P_{\ f}) \vdash (Q_{\ f}))$
by (*rel-tac*)

lemma *wait′-cond-subst* [*usubst*]:
$\$wait' \sharp \sigma \Longrightarrow \sigma \dagger (P \diamond Q) = (\sigma \dagger P) \diamond (\sigma \dagger Q)$
by (*simp add*: *wait′-cond-def usubst unrest*)

lemma *wait′-cond-left-false*: $false \diamond P = (\neg \$wait' \wedge P)$
by (*rel-tac*)

lemma *wait′-cond-seq*: $((P \diamond Q) ;; R) = ((P ;; \$wait \wedge R) \vee (Q ;; \neg\$wait \wedge R))$
by (*simp add*: *wait′-cond-def cond-def seqr-or-distl*, *rel-tac*)

lemma *wait′-cond-true*: $(P \diamond Q \wedge \$wait') = (P \wedge \$wait')$
by (*rel-tac*)

lemma *wait′-cond-false*: $(P \diamond Q \wedge (\neg\$wait')) = (Q \wedge (\neg\$wait'))$
by (*rel-tac*)

lemma *subst-wait′-cond-true* [*usubst*]: $(P \diamond Q)[\![true/\$wait']\!] = P[\![true/\$wait']\!]$
by *rel-tac*

lemma *subst-wait′-cond-false* [*usubst*]: $(P \diamond Q)[\![false/\$wait']\!] = Q[\![false/\$wait']\!]$
by *rel-tac*

lemma *subst-wait′-left-subst*: $(P[\![true/\$wait']\!] \diamond Q) = (P \diamond Q)$
by (*metis wait′-cond-def cond-def conj-comm conj-eq-out-var-subst upred-eq-true wait-uvar*)

lemma *subst-wait′-right-subst*: $(P \diamond Q[\![false/\$wait']\!]) = (P \diamond Q)$
by (*metis cond-def conj-eq-out-var-subst upred-eq-false utp-pred.inf.commute wait′-cond-def wait-uvar*)

lemma *H2-R1-comm*: $H2(R1(P)) = R1(H2(P))$
by (*simp add*: *H2-split R1-def usubst*, *rel-tac*)

lemma *H2-R2s-comm*: $H2(R2s(P)) = R2s(H2(P))$
by (*simp add*: *H2-split R2s-def usubst*, *rel-tac*)

lemma *H2-R2-comm*: $H2(R2(P)) = R2(H2(P))$
by (*simp add*: *H2-R1-comm H2-R2s-comm R2-def*)

lemma *H2-R3-comm*: $H2(R3c(P)) = R3c(H2(P))$
by (*simp add*: *R3c-H2-commute*)

lemma *R3c-via-H1*: $R1(R3c(H1(P))) = R1(H1(R3(P)))$
by *rel-tac*

lemma *skip-rea-via-H1*: $II_r = R1(H1(R3(II)))$
by *rel-tac*

Pedro's proof for R1 design composition

lemma *R1-design-composition*:
fixes $P\ Q :: ('\vartheta,'\alpha,'\beta)$ *relation-rp*
and $R\ S :: ('\vartheta,'\beta,'\gamma)$ *relation-rp*
assumes $\$ok' \sharp P\ \$ok' \sharp Q\ \$ok \sharp R\ \$ok \sharp S$
shows
$(R1(P \vdash Q) ;; R1(R \vdash S)) =$
$R1((\neg (R1(\neg P) ;; R1(true)) \wedge \neg (R1(Q) ;; R1(\neg R))) \vdash (R1(Q) ;; R1(S)))$
proof −
have $(R1(P \vdash Q) ;; R1(R \vdash S)) = (\exists\ ok_0 \cdot (R1(P \vdash Q))[\![\ll ok_0\gg/\$ok']\!] ;; (R1(R \vdash S))[\![\ll ok_0\gg/\$ok]\!])$
using *seqr-middle uvar-ok* **by** *blast*
also from *assms* **have** ... $= (\exists\ ok_0 \cdot R1((\$ok \wedge P) \Rightarrow (\ll ok_0\gg \wedge Q)) ;; R1((\ll ok_0\gg\ \wedge R) \Rightarrow (\$ok'$
$\wedge S)))$
by (*simp add*: *design-def R1-def usubst unrest*)
also from *assms* **have** ... $= ((R1((\$ok \wedge P) \Rightarrow (true \wedge Q)) ;; R1((true \wedge R) \Rightarrow (\$ok' \wedge S)))$

$\vee (R1((\$ok \wedge P) \Rightarrow (false \wedge Q)) ;; R1((false \wedge R) \Rightarrow (\$ok' \wedge S))))$
by (*simp add*: *false-alt-def true-alt-def*)
also from *assms* **have** ... $= ((R1((\$ok \wedge P) \Rightarrow Q) ;; R1(R \Rightarrow (\$ok' \wedge S)))$
$\vee (R1(\neg (\$ok \wedge P)) ;; R1(true)))$
by *simp*
also from *assms* **have** ... $= ((R1(\neg \$ok \vee \neg P \vee Q) ;; R1(\neg R \vee (\$ok' \wedge S)))$
$\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$
by (*simp add*: *impl-alt-def utp-pred.sup.assoc*)
also from *assms* **have** ... $= (((R1(\neg \$ok \vee \neg P) \vee R1(Q)) ;; R1(\neg R \vee (\$ok' \wedge S)))$
$\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$
by (*simp add*: *R1-disj utp-pred.disj-assoc*)
also from *assms* **have** ... $= ((R1(\neg \$ok \vee \neg P) ;; R1(\neg R \vee (\$ok' \wedge S)))$
$\vee (R1(Q) ;; R1(\neg R \vee (\$ok' \wedge S)))$
$\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$
by (*simp add*: *seqr-or-distl utp-pred.sup.assoc*)
also from *assms* **have** ... $= ((R1(Q) ;; R1(\neg R \vee (\$ok' \wedge S)))$
$\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$
by *rel-tac*
also from *assms* **have** ... $= ((R1(Q) ;; (R1(\neg R) \vee R1(S) \wedge \$ok'))$
$\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$
by (*simp add*: *R1-disj R1-extend-conj utp-pred.inf-commute*)
also have ... $= ((R1(Q) ;; (R1(\neg R) \vee R1(S) \wedge \$ok'))$
$\vee ((R1(\neg \$ok) :: ('\vartheta,'\alpha,'\beta)\ relation\text{-}rp) ;; R1(true))$
$\vee (R1(\neg P) ;; R1(true)))$
by (*simp add*: *R1-disj seqr-or-distl*)
also have ... $= ((R1(Q) ;; (R1(\neg R) \vee R1(S) \wedge \$ok'))$
$\vee (R1(\neg \$ok))$
$\vee (R1(\neg P) ;; R1(true)))$
proof −
have $((R1(\neg \$ok) :: ('\vartheta,'\alpha,'\beta)\ relation\text{-}rp) ;; R1(true)) =$
$(R1(\neg \$ok) :: ('\vartheta,'\alpha,'\gamma)\ relation\text{-}rp)$
by (*rel-tac*, *metis alpha-d.select-convs*(2) *alpha-rp'.select-convs*(2) *order-refl*)
thus *?thesis*
by *simp*
qed
also have ... $= ((R1(Q) ;; (R1(\neg R) \vee (R1(S \wedge \$ok'))))$
$\vee R1(\neg \$ok)$
$\vee (R1(\neg P) ;; R1(true)))$
by (*simp add*: *R1-extend-conj*)
also have ... $= ((R1(Q) ;; (R1\ (\neg R)))$
$\vee (R1(Q) ;; (R1(S \wedge \$ok')))$
$\vee R1(\neg \$ok)$
$\vee (R1(\neg P) ;; R1(true)))$
by (*simp add*: *seqr-or-distr utp-pred.sup.assoc*)
also have ... $= R1((R1(Q) ;; (R1\ (\neg R)))$
$\vee (R1(Q) ;; (R1(S \wedge \$ok')))$
$\vee (\neg \$ok)$
$\vee (R1(\neg P) ;; R1(true)))$
by (*simp add*: *R1-disj R1-seqr*)
also have ... $= R1((R1(Q) ;; (R1\ (\neg R)))$
$\vee ((R1(Q) ;; R1(S)) \wedge \$ok')$
$\vee (\neg \$ok)$
$\vee (R1(\neg P) ;; R1(true)))$
by (*rel-tac*)
also have ... $= R1(\neg(\$ok \wedge \neg (R1(\neg P) ;; R1(true)) \wedge \neg (R1(Q) ;; (R1\ (\neg R))))$

$\vee\ \ ((R1(Q)\ ;;\ R1(S)) \wedge \$ok')$)
by (*rel-tac*)
also have ... $= R1((\$ok \wedge \neg\ (R1(\neg\ P)\ ;;\ R1(true)) \wedge \neg\ (R1(Q)\ ;;\ (R1\ (\neg\ R))))$
$\Rightarrow (\$ok' \wedge (R1(Q)\ ;;\ R1(S))))$
by (*simp add*: *impl-alt-def utp-pred.inf-commute*)
also have ... $= R1((\neg\ (R1(\neg\ P)\ ;;\ R1(true)) \wedge \neg\ (R1(Q)\ ;;\ R1(\neg\ R))) \vdash (R1(Q)\ ;;\ R1(S)))$
by (*simp add*: *design-def*)
finally show *?thesis* .
qed

definition [*upred-defs*]: $R3c\text{-}pre(P) = (true \triangleleft \$wait \triangleright P)$

definition [*upred-defs*]: $R3c\text{-}post(P) = (\lceil II \rceil_D \triangleleft \$wait \triangleright P)$

lemma *R3c-pre-conj*: $R3c\text{-}pre(P \wedge Q) = (R3c\text{-}pre(P) \wedge R3c\text{-}pre(Q))$
by *rel-tac*

lemma *R3c-pre-seq*:
$(true\ ;;\ Q) = true \Longrightarrow R3c\text{-}pre(P\ ;;\ Q) = (R3c\text{-}pre(P)\ ;;\ Q)$
by (*rel-tac*)

lemma *R2s-design*: $R2s(P \vdash Q) = (R2s(P) \vdash R2s(Q))$
by (*simp add*: *R2s-def design-def usubst*)

lemma *R1-R3c-design*:
$R1(R3c(P \vdash Q)) = R1(R3c\text{-}pre(P) \vdash R3c\text{-}post(Q))$
by (*rel-tac*, *simp-all add*: *alpha-d.equality*)

lemma *unrest-ok-R2s* [*unrest*]: $\$ok \sharp P \Longrightarrow \$ok \sharp R2s(P)$
by (*simp add*: *R2s-def unrest*)

lemma *unrest-ok'-R2s* [*unrest*]: $\$ok' \sharp P \Longrightarrow \$ok' \sharp R2s(P)$
by (*simp add*: *R2s-def unrest*)

lemma *unrest-ok-R3c-pre* [*unrest*]: $\$ok \sharp P \Longrightarrow \$ok \sharp R3c\text{-}pre(P)$
by (*simp add*: *R3c-pre-def cond-def unrest*)

lemma *unrest-ok'-R3c-pre* [*unrest*]: $\$ok' \sharp P \Longrightarrow \$ok' \sharp R3c\text{-}pre(P)$
by (*simp add*: *R3c-pre-def cond-def unrest*)

lemma *unrest-ok-R3c-post* [*unrest*]: $\$ok \sharp P \Longrightarrow \$ok \sharp R3c\text{-}post(P)$
by (*simp add*: *R3c-post-def cond-def unrest*)

lemma *unrest-ok-R3c-post'* [*unrest*]: $\$ok' \sharp P \Longrightarrow \$ok' \sharp R3c\text{-}post(P)$
by (*simp add*: *R3c-post-def cond-def unrest*)

lemma *R3c-R1-design-composition*:
assumes $\$ok' \sharp P\ \$ok' \sharp Q\ \$ok \sharp R\ \$ok \sharp S$
shows $(R3c(R1(P \vdash Q))\ ;;\ R3c(R1(R \vdash S))) =$
$R3c(R1((\neg\ (R1(\neg\ P)\ ;;\ R1(true)) \wedge \neg\ ((R1(Q) \wedge \neg\ \$wait')\ ;;\ R1(\neg\ R)))$
$\vdash (R1(Q)\ ;;\ (\lceil II \rceil_D \triangleleft \$wait \triangleright R1(S)))))$
proof −
have *1*:$(\neg\ (R1\ (\neg\ R3c\text{-}pre\ P)\ ;;\ R1\ true)) = (R3c\text{-}pre\ (\neg\ (R1\ (\neg\ P)\ ;;\ R1\ true)))$
by (*rel-tac*)
have *2*:$(\neg\ (R1\ (R3c\text{-}post\ Q)\ ;;\ R1\ (\neg\ R3c\text{-}pre\ R))) = R3c\text{-}pre(\neg\ (R1\ Q \wedge \neg\ \$wait'\ ;;\ R1\ (\neg\ R)))$

by (*rel-tac*)
have *3*:(*R1* (*R3c-post Q*) ;; *R1* (*R3c-post S*)) = *R3c-post* (*R1 Q* ;; ($\lceil II \rceil_D$ ◁ \$*wait* ▷ *R1 S*))
by (*rel-tac*)
show *?thesis*
apply (*simp add*: *R3c-semir-form R1-R3c-commute*[*THEN sym*] *R1-R3c-design unrest*)
apply (*subst R1-design-composition*)
apply (*simp-all add*: *unrest assms R3c-pre-conj 1 2 3*)
done
qed

lemma *R2c-design*: *R2c*(*P* ⊢ *Q*) = *R2c*(*P*) ⊢ *R2c*(*Q*)
by (*rel-tac*)

lemma *R1-des-lift-skip*: *R1*($\lceil II \rceil_D$) = $\lceil II \rceil_D$
by (*rel-tac*)

lemma *R2s-subst-wait-true* [*usubst*]:
(*R2s*(*P*))⟦*true*/\$*wait*⟧ = *R2s*(*P*⟦*true*/\$*wait*⟧)
by (*simp add*: *R2s-def usubst unrest*)

lemma *R2s-subst-wait′-true* [*usubst*]:
(*R2s*(*P*))⟦*true*/\$*wait´*⟧ = *R2s*(*P*⟦*true*/\$*wait´*⟧)
by (*simp add*: *R2s-def usubst unrest*)

lemma *R2-subst-wait-true* [*usubst*]:
(*R2*(*P*))⟦*true*/\$*wait*⟧ = *R2*(*P*⟦*true*/\$*wait*⟧)
by (*simp add*: *R2-def R1-def R2s-def usubst unrest*)

lemma *R2-subst-wait′-true* [*usubst*]:
(*R2*(*P*))⟦*true*/\$*wait´*⟧ = *R2*(*P*⟦*true*/\$*wait´*⟧)
by (*simp add*: *R2-def R1-def R2s-def usubst unrest*)

lemma *R2-subst-wait-false* [*usubst*]:
(*R2*(*P*))⟦*false*/\$*wait*⟧ = *R2*(*P*⟦*false*/\$*wait*⟧)
by (*simp add*: *R2-def R1-def R2s-def usubst unrest*)

lemma *R2-subst-wait′-false* [*usubst*]:
(*R2*(*P*))⟦*false*/\$*wait´*⟧ = *R2*(*P*⟦*false*/\$*wait´*⟧)
by (*simp add*: *R2-def R1-def R2s-def usubst unrest*)

lemma *R2-des-lift-skip*:
R2($\lceil II \rceil_D$) = $\lceil II \rceil_D$
by (*rel-tac*, *metis* (*no-types*, *lifting*) *alpha-rp′.surjective alpha-rp′.update-convs*(*2*) *append-Nil2 append-minus strict-prefixE*)

lemma *RH-design-composition*:
assumes \$*ok´* ♯ *P* \$*ok´* ♯ *Q* \$*ok* ♯ *R* \$*ok* ♯ *S*
shows (*RH*(*P* ⊢ *Q*) ;; *RH*(*R* ⊢ *S*)) =
RH((¬ (*R1* (¬ *R2s P*) ;; *R1 true*) ∧ ¬ (*R1* (*R2s Q*) ∧ ¬ \$*wait´* ;; *R1* (¬ *R2s R*))) ⊢
(*R1* (*R2s Q*) ;; ($\lceil II \rceil_D$ ◁ \$*wait* ▷ *R1* (*R2s S*))))
proof −
have *1*: *R2c* (*R1* (¬ *R2s P*) ;; *R1 true*) = (*R1* (¬ *R2s P*) ;; *R1 true*)
proof −
have *1*:(*R1* (¬ *R2s P*) ;; *R1 true*) = (*R1*(*R2* (¬ *P*) ;; *R2 true*))
by (*rel-tac*)

have $R2c(R1(R2\ (\neg\ P)\ ;;\ R2\ true)) = R2c(R1(R2\ (\neg\ P)\ ;;\ R2\ true))$
using *R2c-not* **by** *blast*
also have ... $= R2(R2\ (\neg\ P)\ ;;\ R2\ true)$
by (*metis R1-R2c-commute R1-R2c-is-R2*)
also have ... $= (R2\ (\neg\ P)\ ;;\ R2\ true)$
by (*simp add*: *R2-seqr-distribute*)
also have ... $= (R1\ (\neg\ R2s\ P)\ ;;\ R1\ true)$
by (*simp add*: *R2-def R2s-not R2s-true*)
finally show *?thesis*
by (*simp add*: *1*)
qed

have $2{:}R2c\ (R1\ (R2s\ Q) \wedge \neg\ \$wait' ;;\ R1\ (\neg\ R2s\ R)) = (R1\ (R2s\ Q) \wedge \neg\ \$wait' ;;\ R1\ (\neg\ R2s\ R))$
proof –
have $(R1\ (R2s\ Q) \wedge \neg\ \$wait' ;;\ R1\ (\neg\ R2s\ R)) = R1\ (R2\ (Q \wedge \neg\ \$wait')\ ;;\ R2\ (\neg\ R))$
by (*rel-tac*)
hence $R2c\ (R1\ (R2s\ Q) \wedge \neg\ \$wait' ;;\ R1\ (\neg\ R2s\ R)) = (R2\ (Q \wedge \neg\ \$wait')\ ;;\ R2\ (\neg\ R))$
by (*metis R1-R2c-commute R1-R2c-is-R2 R2-seqr-distribute*)
also have ... $= (R1\ (R2s\ Q) \wedge \neg\ \$wait' ;;\ R1\ (\neg\ R2s\ R))$
by *rel-tac*
finally show *?thesis* **.**
qed

have $3{:}R2c((R1\ (R2s\ Q)\ ;;\ (\lceil II\rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S)))) = (R1\ (R2s\ Q)\ ;;\ (\lceil II\rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S)))$
proof –
have $R2c(((R1\ (R2s\ Q))[\![true/\$wait']\!]\ ;;\ (\lceil II\rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![true/\$wait]\!]))$
$= ((R1\ (R2s\ Q))[\![true/\$wait']\!]\ ;;\ (\lceil II\rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![true/\$wait]\!])$
proof –
have $R2c(((R1\ (R2s\ Q))[\![true/\$wait']\!]\ ;;\ (\lceil II\rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![true/\$wait]\!])) =$
$R2c(R1\ (R2s\ (Q[\![true/\$wait']\!]))\ ;;\ \lceil II\rceil_D[\![true/\$wait]\!])$
by (*simp add*: *usubst cond-unit-T R1-def R2s-def*, *rel-tac*)
also have ... $= R2c(R2(Q[\![true/\$wait']\!])\ ;;\ R2(\lceil II\rceil_D[\![true/\$wait]\!]))$
by (*metis R2-def R2-des-lift-skip R2-subst-wait-true*)
also have ... $= (R2(Q[\![true/\$wait']\!])\ ;;\ R2(\lceil II\rceil_D[\![true/\$wait]\!]))$
using *R2c-seq* **by** *blast*
also have ... $= ((R1\ (R2s\ Q))[\![true/\$wait']\!]\ ;;\ (\lceil II\rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![true/\$wait]\!])$
apply (*simp add*: *usubst cond-unit-T R2-des-lift-skip*)
apply (*metis R2-def R2-des-lift-skip R2-subst-wait′-true R2-subst-wait-true*)
done
finally show *?thesis* **.**
qed
moreover have $R2c(((R1\ (R2s\ Q))[\![false/\$wait']\!]\ ;;\ (\lceil II\rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![false/\$wait]\!]))$
$= ((R1\ (R2s\ Q))[\![false/\$wait']\!]\ ;;\ (\lceil II\rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![false/\$wait]\!])$
by (*simp add*: *usubst cond-unit-F*, *metis R2-R1-form R2-subst-wait′-false R2-subst-wait-false R2c-seq*)
ultimately show *?thesis*
by (*smt R2-R1-form R2-condr′ R2-des-lift-skip R2c-seq R2s-wait*)
qed

have $(R1(R2s(R3c(P \vdash Q)))\ ;;\ R1(R2s(R3c(R \vdash S)))) =$
$((R3c(R1(R2s(P) \vdash R2s(Q))))\ ;;\ R3c(R1(R2s(R) \vdash R2s(S))))$
by (*metis R2-R3c-commute R2-def R2s-design*)
also have ... $= R3c\ (R1\ ((\neg\ (R1\ (\neg\ R2s\ P)\ ;;\ R1\ true) \wedge \neg\ (R1\ (R2s\ Q) \wedge \neg\ \$wait' ;;\ R1\ (\neg\ R2s\ R))) \vdash$

$(R1\ (R2s\ Q)\ ;;\ (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S)))))$
by (*simp add*: *R3c-R1-design-composition assms unrest*)
also have ... $= R3c(R1(R2c((\neg\ (R1\ (\neg\ R2s\ P)\ ;;\ R1\ true) \wedge \neg\ (R1\ (R2s\ Q) \wedge \neg\ \$wait\acute{}\ ;;\ R1\ (\neg\ R2s\ R))) \vdash$
$(R1\ (R2s\ Q)\ ;;\ (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))))))$
by (*simp add*: *R2c-design R2c-and R2c-not 1 2 3*)
finally show *?thesis*
by (*metis RH-R2c-def RH-def*)
qed

Marcel's proof for reactive design composition

lemma *reactive-design-composition*:
assumes $out\alpha \sharp p_1$ p_1 *is R2s* P_2 *is R2s* Q_1 *is R2s* Q_2 *is R2s*
shows
$(RH(p_1 \vdash Q_1)\ ;;\ RH(P_2 \vdash Q_2)) =$
$RH((p_1 \wedge \neg\ ((\$ok\acute{} \wedge \neg\ \$wait\acute{} \wedge Q_1)\ ;;\ R1\ (\neg\ P_2)))$
$\vdash (((\$wait\acute{} \wedge Q_1) \vee ((\$ok\acute{} \wedge \neg\ \$wait\acute{} \wedge Q_1)\ ;;\ R1(Q_2)))))$ (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = *RH*(*?lhs*)
by (*metis Healthy-def′ RH-idem RH-seq-closure*)
also have ... $= RH\ ((R2 \circ R1)\ (p_1 \vdash Q_1)\ ;;\ RH\ (P_2 \vdash Q_2))$
by (*metis R1-R2-commute R1-idem R2-R3c-commute R2-def R3c-idem R3c-semir-form RH-def comp-apply*)
also have ... $= RH\ (R1\ ((\neg\ \$ok \vee R2s\ (\neg\ p_1)) \vee \$ok\acute{} \wedge R2s\ Q_1)\ ;;\ RH(P_2 \vdash Q_2))$
by (*simp add*: *design-def R2-R1-form impl-alt-def R2s-not R2s-ok R2s-disj R2s-conj R2s-ok′*)
also have ... $= RH(((\neg\ \$ok \wedge \$tr \le_u \$tr\acute{})\ ;;\ RH(P_2 \vdash Q_2))$
$\vee ((\neg\ R2s(p_1) \wedge \$tr \le_u \$tr\acute{})\ ;;\ RH(P_2 \vdash Q_2))$
$\vee ((\$ok\acute{} \wedge R2s(Q_1) \wedge \$tr \le_u \$tr\acute{})\ ;;\ RH(P_2 \vdash Q_2)))$
by (*smt R1-conj R1-def R1-disj R1-negate-R1 R2-def R2s-not seqr-or-distl utp-pred.conj-assoc utp-pred.inf.commute utp-pred.sup.assoc*)
also have ... $= RH(((\neg\ \$ok \wedge \$tr \le_u \$tr\acute{})\ ;;\ RH(P_2 \vdash Q_2))$
$\vee ((\neg\ p_1 \wedge \$tr \le_u \$tr\acute{})\ ;;\ RH(P_2 \vdash Q_2))$
$\vee ((\$ok\acute{} \wedge Q_1 \wedge \$tr \le_u \$tr\acute{})\ ;;\ RH(P_2 \vdash Q_2)))$
by (*metis Healthy-def′ assms*(*2*) *assms*(*4*))

also have ... $= RH((\neg\ \$ok \wedge \$tr \le_u \$tr\acute{})$
$\vee (\neg\ p_1 \wedge \$tr \le_u \$tr\acute{})$
$\vee ((\$ok\acute{} \wedge Q_1 \wedge \$tr \le_u \$tr\acute{})\ ;;\ RH(P_2 \vdash Q_2)))$
proof −
have $((\neg\ \$ok \wedge \$tr \le_u \$tr\acute{})\ ;;\ RH(P_2 \vdash Q_2)) = (\neg\ \$ok \wedge \$tr \le_u \$tr\acute{})$
by (*rel-tac*, *metis alpha-d.select-convs*(*1*) *alpha-d.select-convs*(*2*) *order-refl*)
moreover have $(((\neg\ p_1\ ;;\ true) \wedge \$tr \le_u \$tr\acute{})\ ;;\ RH(P_2 \vdash Q_2)) = ((\neg\ p_1\ ;;\ true) \wedge \$tr \le_u \$tr\acute{})$
by (*rel-tac*, *metis alpha-d.select-convs*(*1*) *alpha-d.select-convs*(*2*) *order-refl*)
ultimately show *?thesis*
by (*smt assms*(*1*) *precond-right-unit unrest-not*)
qed

also have ... $= RH((\neg\ \$ok \wedge \$tr \le_u \$tr\acute{})$
$\vee (\neg\ p_1 \wedge \$tr \le_u \$tr\acute{})$
$\vee ((\$ok\acute{} \wedge Q_1 \wedge \$tr \le_u \$tr\acute{})\ ;;\ (\$wait \wedge \$ok\acute{} \wedge II))$
$\vee ((\$ok\acute{} \wedge Q_1 \wedge \$tr \le_u \$tr\acute{})\ ;;\ (\neg\ \$wait \wedge R1(\neg\ P_2) \wedge \$tr \le_u \$tr\acute{}))$
$\vee ((\$ok\acute{} \wedge Q_1 \wedge \$tr \le_u \$tr\acute{})\ ;;\ (\neg\ \$wait \wedge \$ok\acute{} \wedge R2(Q_2) \wedge \$tr \le_u \$tr\acute{})))$
proof −
have *1*:$RH(P_2 \vdash Q_2) = ((\$wait \wedge \neg\ \$ok \wedge \$tr \le_u \$tr\acute{})$
$\vee (\$wait \wedge \$ok\acute{} \wedge II)$

$\vee$ $(\neg\ \$wait \wedge \neg\ \$ok \wedge \$tr \leq_u \$tr')$
$\vee$ $(\neg\ \$wait \wedge R2(\neg\ P_2) \wedge \$tr \leq_u \$tr')$
$\vee$ $(\neg\ \$wait \wedge \$ok' \wedge R2(Q_2) \wedge \$tr \leq_u \$tr'))$
by (*simp add*: *RH-alt-def' R2-condr' R2s-wait R2-skip-rea R3c-def usubst*, *rel-tac*)
have *2*:$((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr') \;;\; (\$wait \wedge \neg\ \$ok \wedge \$tr \leq_u \$tr')) = false$
by *rel-tac*
have *3*:$((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr') \;;\; (\neg\ \$wait \wedge \neg\ \$ok \wedge \$tr \leq_u \$tr')) = false$
by *rel-tac*
have *4*:$R2(\neg\ P_2) = R1(\neg\ P_2)$
by (*metis Healthy-def' R1-negate-R1 R2-def R2s-not assms*(*3*))
show *?thesis*
by (*simp add*: *1 2 3 4 seqr-or-distr*)
qed

also have ... $= RH((\neg\ \$ok) \vee (\neg\ p_1)$
$\vee ((\$ok' \wedge Q_1) \;;\; (\$wait \wedge \$ok' \wedge II))$
$\vee ((\$ok' \wedge Q_1) \;;\; (\neg\ \$wait \wedge R1(\neg\ P_2)))$
$\vee ((\$ok' \wedge Q_1) \;;\; (\neg\ \$wait \wedge \$ok' \wedge R2(Q_2))))$
by (*rel-tac*)

also have ... $= RH((\neg\ \$ok) \vee (\neg\ p_1)$
$\vee (\$ok' \wedge \$wait' \wedge Q_1)$
$\vee ((\$ok' \wedge Q_1) \;;\; (\neg\ \$wait \wedge R1(\neg\ P_2)))$
$\vee ((\$ok' \wedge Q_1) \;;\; (\neg\ \$wait \wedge \$ok' \wedge R1(Q_2))))$
proof –
have $((\$ok' \wedge Q_1) \;;\; (\$wait \wedge \$ok' \wedge II)) = (\$ok' \wedge \$wait' \wedge Q_1)$
by (*rel-tac*)
moreover have $R2(Q_2) = R1(Q_2)$
by (*metis Healthy-def' R2-def assms*(*5*))
ultimately show *?thesis* **by** *simp*
qed

also have ... $= RH((\neg\ \$ok) \vee (\neg\ p_1)$
$\vee (\$ok' \wedge \$wait' \wedge Q_1)$
$\vee ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; (R1(\neg\ P_2)))$
$\vee ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; (\$ok' \wedge R1(Q_2))))$
by *rel-tac*

also have ... $= RH((\neg\ \$ok) \vee (\neg\ p_1) \vee ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; R1(\neg\ P_2))$
$\vee (\$ok' \wedge ((\$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; R1(Q_2)))))$
by *rel-tac*

also have ... $= RH(\neg\ (\$ok \wedge p_1 \wedge \neg\ ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; R1(\neg\ P_2)))$
$\vee (\$ok' \wedge ((\$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; R1(Q_2)))))$
by *rel-tac*

also have ... = *?rhs*
proof –
have $(\neg\ (\$ok \wedge p_1 \wedge \neg\ ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; R1(\neg\ P_2)))$
$\vee (\$ok' \wedge ((\$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; R1(Q_2)))))$
$= ((\$ok \wedge (p_1 \wedge \neg\ ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; R1(\neg\ P_2)))) \Rightarrow$
$(\$ok' \wedge ((\$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg\ \$wait' \wedge Q_1) \;;\; R1(Q_2)))))$
by *pred-tac*
thus *?thesis*
by (*simp add*: *design-def*)

qed

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

end