Isabelle/UTP: Mechanised reasoning for the UTP

Simon Foster Frank Zeyda

December 8, 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/Positive
../utils/ttrace
../utils/Library-extra/Pfun
../utils/Library-extra/Ffun
../utils/Library-extra/Derivative-extra
../utils/Library-extra/List-lexord-alt
../utils/Library-extra/Monoid-extra
~~/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*)

declare *fst-vwb-lens* [*simp*]
declare *snd-vwb-lens* [*simp*]
declare *lens-indep-left-comp* [*simp*]
declare *comp-vwb-lens* [*simp*]
declare *lens-indep-left-ext* [*simp*]
declare *lens-indep-right-ext* [*simp*]

This theory describes the foundational structure of UTP variables, upon which the rest of our model rests. We start by defining alphabets, which following [3, 4] in this shallow model are simply represented as types, though by convention usually a record type where each field corresponds to a variable.

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

UTP variables carry two type parameters, $'a$ that corresponds to the variable's type and $'\alpha$ that corresponds to alphabet of which the variable is a type. There is 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 [3] is a syntactic translations that allows to retrieve a variable given its name, assuming the variable is a field in a record.

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

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

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

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

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

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

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

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

lemma *in-out-indep* [*simp*]:
in-var x ⋈ *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 ⋈ *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 ⋈ *y* ⟹ *in-var x* ⋈ *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 ⋈ *y* ⟹ *out-var x* ⋈ *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* :: (′α, ′α) *uvar* (Σ) **where**
univ-alpha ≡ 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*)
-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*
⟩⟩

named-theorems *uvar-defs*

named-theorems *alpha-splits*

end

1.1 Deep UTP variables

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

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

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

1.2 Cardinalities

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

datatype *ucard* = *fin nat* | *aleph0* ($\aleph_0$) | *cont* (c)

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

type-synonym *uuniv* = *nat set*

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

fun *uuniv* :: *ucard* ⇒ *uuniv set* ($\mathcal{U}$′(-′)) **where**
$\mathcal{U}(fin\ n) = \{\{x\} \mid x.\ x \leq n\}$ |
$\mathcal{U}(\aleph_0) = \{\{x\} \mid x.\ True\}$ |
$\mathcal{U}(\mathrm{c}) = UNIV$

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

definition *ucard-of* :: ′*a*::*continuum itself* ⇒ *ucard* **where**
ucard-of x = (*if* (*finite* (*UNIV* :: ′*a set*))
    then fin(*card*(*UNIV* :: ′*a set*) − *1*)
  else if (*countable* (*UNIV* :: ′*a set*))
    then $\aleph_0$
  else c)

syntax
 -ucard :: *type* ⇒ *ucard* (*UCARD*′(-′))

translations
 UCARD(′*a*) == *CONST ucard-of* (*TYPE*(′*a*))

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

lemma *ucard-of-finite* [*simp*]:
 finite (*UNIV* :: ′*a*::*continuum set*) ⟹ *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*) ⟧ ⟹ *UCARD*(′*a*) = $\aleph_0$
 by (*simp add*: *ucard-of-def*)

lemma *ucard-of-uncountably-infinite* [*simp*]:
 uncountable (*UNIV* :: ′*a set*) ⟹ *UCARD*(′*a* :: *continuum*) = c
 apply (*simp add*: *ucard-of-def*)
 using *countable-finite* **apply** *blast*
done

1.3 Injection functions

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

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

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

definition *uinject* :: ′*a*::*continuum* ⇒ *uuniv* **where**

uinject x = (*if* (*finite* (*UNIV* :: *'a set*))
then {*to-nat-fin* x}
else if (*countable* (*UNIV* :: *'a set*))
then {*to-nat-on* (*UNIV* :: *'a set*) x}
else to-nat-set x)

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

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

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

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

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

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

1.4 Deep variables

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

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

declare *dname.splits* [*alpha-splits*]

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

typedef $vstore = \{f :: dname \Rightarrow uuniv.\ \forall\ \ x.\ f(x) \in \mathcal{U}(dname\text{-}card\ x)\}$
  apply (*rule-tac* $x{=}\lambda\ x.\ \{0\}$ **in** *exI*)
  apply (*auto*)
  apply (*rename-tac x*)
  apply (*case-tac dname-card x*)
  apply (*simp-all*)
done

setup-lifting *type-definition-vstore*

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

setup-lifting *type-definition-dvar*

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

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

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

term *fun-lens*

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

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

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

by (*auto*)

definition *dvar-lens* :: (*'a::continuum*) *dvar* $\Rightarrow$ (*'a* $\Longrightarrow$ *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* $\bowtie$ *dvar-lens y* $\longleftrightarrow$ (*dvar-dname x* $\neq$ *dvar-dname y*)
proof $-$
obtain *v1 v2* :: *'b*::{*continuum*,*two*} **where** *v*:*v1* $\neq$ *v2*
using *two-diff* **by** *auto*
obtain *u* :: *'a*::{*continuum*,*two*} **and** *v* :: *'b*::{*continuum*,*two*}
where *uv*: *uinject u* $\neq$ *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 *y* :: *dname*
assume *a1*: *ucard-of* (*TYPE*(*'b*)::*'b itself*) = *ucard-of* (*TYPE*(*'a*)::*'a itself*)
assume *dname-card y* = *ucard-of* (*TYPE*(*'a*)::*'a itself*)
assume *a2*:
$\forall\ \sigma.\ (\forall x.\ \sigma\ x \in \mathcal{U}(\mathit{dname\text{-}card}\ x)) \longrightarrow (\forall\ v\ u.\ \sigma(y := \mathit{uinject}\ (u::'a)) = \sigma(y := \mathit{uinject}\ (v::'b)))$
$\forall\ \sigma.\ (\forall x.\ \sigma\ x \in \mathcal{U}(\mathit{dname\text{-}card}\ x)) \longrightarrow (\forall\ v.\ (\mathit{uproject}\ (\mathit{uinject}\ v)::'a) = \mathit{uproject}\ (\sigma\ y))$
$\forall\ \sigma.\ (\forall x.\ \sigma\ x \in \mathcal{U}(\mathit{dname\text{-}card}\ x)) \longrightarrow (\forall\ u.\ (\mathit{uproject}\ (\mathit{uinject}\ u)::'b) = \mathit{uproject}\ (\sigma\ y))$
obtain *NN* :: *vstore* $\Rightarrow$ *dname* $\Rightarrow$ *nat set* **where**
$\bigwedge v.\ \forall d.\ \mathit{NN}\ v\ d \in \mathcal{U}(\mathit{dname\text{-}card}\ d)$
by (*metis* (*lifting*) *Abs-vstore-cases mem-Collect-eq*)
then show *False*
using *a2 a1* **by** (*metis fun-upd-same 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* $\Longrightarrow$ *'a* ($\mathcal{V}$)
assumes *vstore-vwb-lens* [*simp*]: *vwb-lens vstore-lens*

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

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

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

lemma *uvar-dvar*: *vwb-lens* $(x\uparrow)$
  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 $x\uparrow \bowtie y\uparrow \longleftrightarrow$ *dvar-dname* $x \neq$ *dvar-dname* y
proof −
  have $x\uparrow \bowtie y\uparrow \longleftrightarrow$ *dvar-lens* $x \bowtie$ *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 ... $\longleftrightarrow$ *dvar-dname* $x \neq$ *dvar-dname* y
    by (*simp add*: *dvar-lens-indep-iff*)
  finally show *?thesis* **.**
qed

lemma *dvar-indep-diff-name'* [*simp*]:
  $x \neq y \Longrightarrow \lceil x\rceil_d\uparrow \bowtie \lceil y\rceil_d\uparrow$
  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*]: *vwb-lens* (*MkDVar* x)
  by (*simp add*: *MkDVar-def uvar-dvar*)

lemma *MkDVar-indep* [*simp*]: $x \neq y \Longrightarrow$ *MkDVar* $x \bowtie$ *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 (<->_d)
-dvar-tyd :: id ⇒ type ⇒ logic (<-::->_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
  Profiling
begin

Before building the predicate model, we will build a model of expressions that generalise alphabetised predicates. Expressions are represented semantically as mapping from the alphabet to the expression's type. This general model will allow us to unify all constructions under one type. All definitions in the file are given using the *lifting* package.

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

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

notation *Rep-uexpr* (⟦-⟧$_e$)

lemma *uexpr-eq-iff*:
  $e = f \longleftrightarrow (\forall\ b.\ [\![e]\!]_e\ b = [\![f]\!]_e\ b)$
  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, 'α) *uexpr* ⇒ ('α, 'α) *lens* (α'(-')) **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 :: $'t \Rightarrow 'e$ ($\ll$-$\gg$)
 ueq :: $'a \Rightarrow 'a \Rightarrow 'b$ (**infixl** $=_u$ *50*)

adhoc-overloading
 ulit lit

syntax
 -uuvar :: *svar* $\Rightarrow$ *logic*

translations
 -uuvar x == *CONST var* x

syntax
 -uuvar :: *svar* $\Rightarrow$ *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* ⇒ ('*a*, '*b*) *uexpr* ⇒ ('*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* ⇒ ('*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* :: (*cancel-monoid-add*, *type*) *cancel-monoid-add*
by (*intro-classes*, *simp-all add*: *plus-uexpr-def minus-uexpr-def zero-uexpr-def*) (*transfer*, *auto*)+

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* $P\ Q = (P \leq Q \land \neg\ Q \leq P)$
instance proof
fix $x\ y\ z$ :: $('a, 'b)$ *uexpr*
show $(x < y) = (x \leq y \land \neg\ y \leq x)$ **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 ab-group-add-class.ab-diff-conv-add-uminus abs-ge-minus-self abs-ge-self add-mono-thms-linordered-semiri*
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*: $[\![numeral\ x]\!]_e\ b = numeral\ x$
by (*induct x*, *simp-all add*: *plus-uexpr-def one-uexpr-def numeral.simps lit.rep-eq bop.rep-eq*)

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

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

adhoc-overloading
ueq eq-upred

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* ⇒ $'f \Rightarrow 'f$
uranres :: $'f \Rightarrow 'b$ *set* ⇒ $'f$
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** *uapply cgf-apply* **and** *uapply tt-apply* **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, $'\alpha$) uexpr $\Rightarrow$ type $\Rightarrow$ ('a, $'\alpha$) uexpr (**infix** $:_u$ 50)
-unil :: ('a list, $'\alpha$) uexpr ($\langle\rangle$)
-ulist :: args => ('a list, $'\alpha$) uexpr ($\langle$(-)$\rangle$)
-uappend :: ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr (**infixr** $\hat{}_u$ 80)
-ulast :: ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a, $'\alpha$) uexpr ($last_u$'(-'))
-ufront :: ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr ($front_u$'(-'))
-uhead :: ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a, $'\alpha$) uexpr ($head_u$'(-'))
-utail :: ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr ($tail_u$'(-'))
-utake :: (nat, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr ($take_u$'(-,/ -'))
-udrop :: (nat, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr ($drop_u$'(-,/ -'))
-ucard :: ('a list, $'\alpha$) uexpr $\Rightarrow$ (nat, $'\alpha$) uexpr ($\#_u$'(-'))
-ufilter :: ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a set, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr (**infixl** $\upharpoonleft_u$ 75)
-uextract :: ('a set, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a list, $'\alpha$) uexpr (**infixl** $\upharpoonright_u$ 75)
-uelems :: ('a list, $'\alpha$) uexpr $\Rightarrow$ ('a set, $'\alpha$) uexpr ($elems_u$'(-'))
-usorted :: ('a list, $'\alpha$) uexpr $\Rightarrow$ (bool, $'\alpha$) uexpr ($sorted_u$'(-'))
-udistinct :: ('a list, $'\alpha$) uexpr $\Rightarrow$ (bool, $'\alpha$) uexpr ($distinct_u$'(-'))
-uless :: ('a, $'\alpha$) uexpr $\Rightarrow$ ('a, $'\alpha$) uexpr $\Rightarrow$ (bool, $'\alpha$) uexpr (**infix** $<_u$ 50)
-uleq :: ('a, $'\alpha$) uexpr $\Rightarrow$ ('a, $'\alpha$) uexpr $\Rightarrow$ (bool, $'\alpha$) uexpr (**infix** $\leq_u$ 50)
-ugreat :: ('a, $'\alpha$) uexpr $\Rightarrow$ ('a, $'\alpha$) uexpr $\Rightarrow$ (bool, $'\alpha$) uexpr (**infix** $>_u$ 50)
-ugeq :: ('a, $'\alpha$) uexpr $\Rightarrow$ ('a, $'\alpha$) uexpr $\Rightarrow$ (bool, $'\alpha$) uexpr (**infix** $\geq_u$ 50)
-umin :: logic $\Rightarrow$ logic $\Rightarrow$ logic (min_u'(-, -'))
-umax :: logic $\Rightarrow$ logic $\Rightarrow$ logic (max_u'(-, -'))
-ugcd :: logic $\Rightarrow$ logic $\Rightarrow$ logic (gcd_u'(-, -'))
-ufinite :: logic $\Rightarrow$ logic ($finite_u$'(-'))
-uempset :: ('a set, $'\alpha$) uexpr ($\{\}_u$)
-uset :: args => ('a set, $'\alpha$) uexpr ($\{$(-)$\}_u$)
-uunion :: ('a set, $'\alpha$) uexpr $\Rightarrow$ ('a set, $'\alpha$) uexpr $\Rightarrow$ ('a set, $'\alpha$) uexpr (**infixl** $\cup_u$ 65)
-uinter :: ('a set, $'\alpha$) uexpr $\Rightarrow$ ('a set, $'\alpha$) uexpr $\Rightarrow$ ('a set, $'\alpha$) uexpr (**infixl** $\cap_u$ 70)
-umem :: ('a, $'\alpha$) uexpr $\Rightarrow$ ('a set, $'\alpha$) uexpr $\Rightarrow$ (bool, $'\alpha$) uexpr (**infix** $\in_u$ 50)
-usubset :: ('a set, $'\alpha$) uexpr $\Rightarrow$ ('a set, $'\alpha$) uexpr $\Rightarrow$ (bool, $'\alpha$) uexpr (**infix** $\subset_u$ 50)
-usubseteq :: ('a set, $'\alpha$) uexpr $\Rightarrow$ ('a set, $'\alpha$) uexpr $\Rightarrow$ (bool, $'\alpha$) uexpr (**infix** $\subseteq_u$ 50)
-utuple :: ('a, $'\alpha$) uexpr $\Rightarrow$ utuple-args $\Rightarrow$ ('a $*$ 'b, $'\alpha$) uexpr ((1'(-,/ -')$_u$))
-utuple-arg :: ('a, $'\alpha$) uexpr $\Rightarrow$ utuple-args (-)
-utuple-args :: ('a, $'\alpha$) uexpr => utuple-args $\Rightarrow$ utuple-args (-,/ -)
-uunit :: ('a, $'\alpha$) uexpr ('(')$_u$)
-ufst :: ('a $\times$ 'b, $'\alpha$) uexpr $\Rightarrow$ ('a, $'\alpha$) uexpr (π_1'(-'))
-usnd :: ('a $\times$ 'b, $'\alpha$) uexpr $\Rightarrow$ ('b, $'\alpha$) uexpr (π_2'(-'))
-uapply :: ('a $\Rightarrow$ 'b, $'\alpha$) uexpr $\Rightarrow$ utuple-args $\Rightarrow$ ('b, $'\alpha$) uexpr (-$(\!|$-$|\!)_u$ [999,0] 999)
-ulamba :: pttrn $\Rightarrow$ logic $\Rightarrow$ logic (λ - $\cdot$ - [0, 10] 10)
-udom :: logic $\Rightarrow$ logic (dom_u'(-'))

-uran :: logic ⇒ logic (ran_u'(-'))
-uinl :: logic ⇒ logic (inl_u'(-'))
-uinr :: logic ⇒ logic (inr_u'(-'))
-umap-empty :: logic ($[]_u$)
-umap-plus :: logic ⇒ logic ⇒ logic (**infixl** $\oplus_u$ 85)
-umap-minus :: logic ⇒ logic ⇒ logic (**infixl** $\ominus_u$ 85)
-udom-res :: logic ⇒ logic ⇒ logic (**infixl** $\lhd_u$ 85)
-uran-res :: logic ⇒ logic ⇒ logic (**infixl** $\rhd_u$ 85)
-umaplet :: [logic, logic] => umaplet (- /↦/ -)
:: umaplet => umaplets (-)
-UMaplets :: [umaplet, umaplets] => umaplets (-,/ -)
-UMapUpd :: [logic, umaplets] => logic (-/'(-')$_u$ [900,0] 900)
-UMap :: umaplets => logic ((1[-]$_u$))

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, -) uexpr
⟨⟩ == ≪[]≫
⟨x, xs⟩ == CONST bop (op #) x ⟨xs⟩
⟨x⟩ == CONST bop (op #) x ≪[]≫
$x\ \hat{}_u\ y$ == 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
$drop_u(n,xs)$ == CONST bop CONST drop n xs
$take_u(n,xs)$ == CONST bop CONST take n 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
$xs \upharpoonright_u A$ == CONST bop CONST seq-filter xs A
$A \upharpoonleft_u xs$ == CONST bop (op $\upharpoonleft_l$) A xs
$x <_u y$ == CONST bop (op <) x y
$x \leq_u y$ == CONST bop (op ≤) x y
$x >_u y$ == $y <_u x$
$x \geq_u y$ == $y \leq_u x$
$min_u(x, y)$ == CONST bop (CONST min) x y
$max_u(x, y)$ == CONST bop (CONST max) x y
$gcd_u(x, y)$ == CONST bop (CONST gcd) x y
$finite_u(x)$ == CONST uop (CONST finite) x
$\{\}_u$ == ≪{}≫
$\{x, xs\}_u$ == CONST bop (CONST insert) x $\{xs\}_u$
$\{x\}_u$ == CONST bop (CONST insert) x ≪{}≫
$A \cup_u B$ == CONST bop (op ∪) A B
$A \cap_u B$ == CONST bop (op ∩) A B
$f \oplus_u g$ => (f :: ((-, -) pfun, -) uexpr) + g

$f \ominus_u g$ => (f :: ((-, -) pfun, -) uexpr) − g
$x \in_u A$ == CONST bop (op ∈) x A
$A \subset_u B$ == CONST bop (op <) A B
$A \subset_u B$ <= CONST bop (op ⊂) 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 ≤) A B
$A \subseteq_u B$ <= CONST bop (op ⊆) A B
$f \subseteq_u g$ <= CONST bop (op $\subseteq_p$) f g
$f \subseteq_u g$ <= CONST bop (op $\subseteq_f$) f g
$()_u$ == ≪()≫
$(x, y)_u$ == CONST bop (CONST Pair) x y
-utuple x (-utuple-args y z) == -utuple x (-utuple-arg (-utuple y z))
$\pi_1(x)$ == CONST uop CONST fst x
$\pi_2(x)$ == CONST uop CONST snd x
$f(\!|x|\!)_u$ == CONST bop CONST uapply f x
λ x · p == CONST ulambda (λ 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
$[]_u$ == ≪CONST uempty≫
$A \lhd_u f$ == CONST bop (CONST udomres) A f
$f \rhd_u A$ == CONST bop (CONST uranres) f A
-UMapUpd m (-UMaplets xy ms) == -UMapUpd (-UMapUpd m xy) ms
-UMapUpd m (-umaplet x y) == CONST trop CONST uupd m x y
-UMap ms == -UMapUpd $[]_u$ ms
-UMap (-UMaplets ms1 ms2) <= -UMapUpd (-UMap ms1) ms2
-UMaplets ms1 (-UMaplets ms2 ms3) <= -UMaplets (-UMaplets ms1 ms2) ms3
$f(\!|x,y|\!)_u$ == CONST bop CONST uapply f $(x,y)_u$

Lifting set intervals

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

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

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

Lifting limits

definition ulim-left = (λ p f. Lim (at-left p) f)
definition ulim-right = (λ p f. Lim (at-right p) f)
definition ucont-on = (λ f A. continuous-on A f)

syntax
-ulim-left :: id ⇒ logic ⇒ logic ⇒ logic (lim_u'(- → -$^-$')'(-'))
-ulim-right :: id ⇒ logic ⇒ logic ⇒ logic (lim_u'(- → -$^+$')'(-'))

-ucont-on :: *logic* ⇒ *logic* ⇒ *logic* (**infix** *cont*−*on*$_u$ *90*)

translations
$lim_u(x \to p^-)(e)$ == *CONST bop CONST ulim-left p* (λ *x* · *e*)
$lim_u(x \to p^+)(e)$ == *CONST bop CONST ulim-right p* (λ *x* · *e*)
f cont−*on*$_u$ *A* == *CONST bop CONST continuous-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
plus-list-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*)

lemma *lit-num-simps*: $\ll 0 \gg = 0$ $\ll 1 \gg = 1$ $\ll numeral\ n \gg = numeral\ n$ $\ll -\ x \gg = -\ \ll x \gg$
by (*simp-all add*: *ueval*, *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* ⇒ *logic* ⇒ *logic* ⇒ *logic* (**infix** ♯ *20*)

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

named-theorems *unrest*

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

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

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

adhoc-overloading
unrest unrest-upred

lemma *unrest-var-comp* [*unrest*]:
⟦ $x \sharp P$; $y \sharp P$ ⟧ ⟹ $x;y \sharp P$
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*]: ⟦ *vwb-lens x*; $x \bowtie y$ ⟧ ⟹ $y \sharp$ *var x*
by (*transfer*, *auto*)

lemma *unrest-iuvar* [*unrest*]: ⟦ *vwb-lens x*; $x \bowtie y$ ⟧ ⟹ $\$y \sharp \$x$
by (*metis in-var-indep in-var-uvar unrest-var*)

lemma *unrest-ouvar* [*unrest*]: ⟦ *vwb-lens x*; $x \bowtie y$ ⟧ ⟹ $\$y' \sharp \$x'$
by (*metis out-var-indep out-var-uvar unrest-var*)

lemma *unrest-iuvar-ouvar* [*unrest*]:
fixes *x* :: ($'a$, $'\alpha$) *uvar*

assumes *vwb-lens 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 *vwb-lens 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*)

lemma *unrest-ulambda* [*unrest*]:
$[\![$ *uvar v*; $\bigwedge x.\ v \sharp F\ x]\!] \Longrightarrow v \sharp (\lambda\ x \cdot F\ x)$
by (*transfer*, *simp*)

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
  $usubst :: 's \Rightarrow 'a \Rightarrow 'b$ (**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 $('\alpha,'\beta)\ psubst = '\alpha\ alphabet \Rightarrow '\beta\ alphabet$
type-synonym $'\alpha\ usubst = '\alpha\ alphabet \Rightarrow '\alpha\ alphabet$

lift-definition $subst :: ('\alpha,\ '\beta)\ psubst \Rightarrow ('a,\ '\beta)\ uexpr \Rightarrow ('a,\ '\alpha)\ uexpr$ **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 $subst\text{-}upd :: ('\alpha,'\beta)\ psubst \Rightarrow 'v \Rightarrow ('a,\ '\alpha)\ uexpr \Rightarrow ('\alpha,'\beta)\ psubst$

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

definition $subst\text{-}upd\text{-}dvar :: ('\alpha,'\beta::vst)\ psubst \Rightarrow 'a::continuum\ dvar \Rightarrow ('a,\ '\alpha)\ uexpr \Rightarrow ('\alpha,'\beta)\ psubst$
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 $usubst\text{-}lookup :: ('\alpha,'\beta)\ psubst \Rightarrow ('a,\ '\beta)\ uvar \Rightarrow ('a,\ '\alpha)\ uexpr\ (\langle\text{-}\rangle_s)$
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* $\rightharpoonup$ *'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* $\Rightarrow$ ($'a$, $'\alpha$) *uvar* $\Rightarrow$ $'\alpha$ *usubst* (**infix** $-_s$ *85*) **where**
subst-del $\sigma\ x = \sigma(x \mapsto_s \&x)$

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 *mwb-lens 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 *mwb-lens 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** *mwb-lens 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 *vwb-lens x vwb-lens 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*]:
vwb-lens $x \Longrightarrow [x \mapsto_s \mathit{var}\ x] = \mathit{id}$
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 *mwb-lens 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*]:
$\llbracket$ *vwb-lens* $x;\ x \sharp \sigma\ \rrbracket \Longrightarrow \langle\sigma\rangle_s\ x = \mathit{var}\ x$
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*]:
vwb-lens $x \Longrightarrow \mathit{id} -_s x = \mathit{id}$
by (*simp add*: *subst-del-def subst-upd-uvar-def*, *transfer*, *auto*)

lemma *subst-del-upd-same* [*usubst*]:
mwb-lens $x \Longrightarrow \sigma(x \mapsto_s v) -_s x = \sigma -_s x$
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*]: $x \sharp \sigma \Longrightarrow \sigma \circ \varrho(x \mapsto_s v) = (\sigma \circ \varrho)(x \mapsto_s v)$
by (*simp add*: *subst-upd-uvar-def*, *transfer*, *auto simp add*: *unrest-usubst-def*)

lemma *id-subst* [*usubst*]: $\mathit{id} \dagger v = v$
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*]: $[\![$ *vwb-lens x*; $x \sharp (\langle\sigma\rangle_s\ x)$; $x \sharp \sigma -_s x$ $]\!] \Longrightarrow x \sharp (\sigma \dagger P)$
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*, 'α) *uvar* $\Rightarrow$ ('*b*, 'α) *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-mod* [*usubst*]: $\sigma \dagger (x\ mod\ y) = \sigma \dagger x\ mod\ \sigma \dagger y$
by (*simp add*: *mod-uexpr-def usubst*)

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

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*,*0*,*0*] *1000*)
  -uexprs :: [*logic*, *uexprs*] => *uexprs* (-,/ -)
    :: *logic* => *uexprs* (-)
  -svars :: [*svar*, *svars*] => *svars* (-,/ -)
    :: *svar* => *svars* (-)
  -salphas :: [*salpha*, *salphas*] => *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*
  P⟦*v*/*x*⟧ <= *CONST usubst* (*CONST subst-upd* (*CONST id*) *x v*) *P*

lemma *subst-singleton*:
  fixes x :: ($'a$, $'\alpha$) *uvar*
  assumes $x \sharp \sigma$
  shows $\sigma(x \mapsto_s v) \dagger P = (\sigma \dagger P)[\![v/x]\!]$
  using *assms*
  by (*simp add*: *usubst*)

lemmas *subst-to-singleton* = *subst-singleton id-subst*

4.3 Unrestriction laws

lemma *unrest-usubst-single* [*unrest*]:
  $[\![\ mwb\text{-}lens\ x;\ x \sharp v\ ]\!] \Longrightarrow x \sharp P[\![v/x]\!]$
  by (*transfer*, *auto simp add*: *subst-upd-uvar-def unrest-upred-def*)

lemma *unrest-usubst-id* [*unrest*]:

$mwb\text{-}lens\ x \Longrightarrow x \sharp id$
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 $\lambda\ P\ x\ b.\ P\ (get_x\ b)$ **.**

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

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

lemma *aext-zero* [*alpha*]: $0 \oplus_p a = 0$
by (*pred-auto*)

lemma *aext-one* [*alpha*]: $1 \oplus_p a = 1$
by (*pred-auto*)

lemma *aext-numeral* [*alpha*]: $numeral\ n \oplus_p a = numeral\ n$
by (*pred-auto*)

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

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

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-auto*)

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

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

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

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

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

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

lemma *aext-true* [*alpha*]: $\mathit{true} \oplus_p a = \mathit{true}$
by (*pred-auto*)

lemma *aext-false* [*alpha*]: $\mathit{false} \oplus_p a = \mathit{false}$
by (*pred-auto*)

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

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

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

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

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

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

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

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-auto*)

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

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-auto*)
    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-auto*)

lemma *arestr-zero* [*alpha*]: $0 \restriction_p a = 0$
  by (*pred-auto*)

lemma *arestr-one* [*alpha*]: $1 \restriction_p a = 1$
  by (*pred-auto*)

lemma *arestr-numeral* [*alpha*]: *numeral n* $\restriction_p a =$ *numeral n*
  by (*pred-auto*)

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

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

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

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

lemma *arestr-and* [*alpha*]: $(P \wedge Q)\restriction_p x = (P\restriction_p x \wedge Q\restriction_p x)$

by (*pred-auto*)

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

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

5.3 Alphabet lens laws

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

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

lemma *in-var-prod-lens* [*alpha*]:
$wb\text{-}lens\ Y \Longrightarrow in\text{-}var\ x ;_L (X \times_L Y) = in\text{-}var\ (x ;_L X)$
by (*simp add*: *in-var-def prod-as-plus lens-comp-assoc fst-lens-plus*)

lemma *out-var-prod-lens* [*alpha*]:
$wb\text{-}lens\ X \Longrightarrow out\text{-}var\ x ;_L (X \times_L Y) = out\text{-}var\ (x ;_L Y)$
apply (*simp add*: *out-var-def prod-as-plus lens-comp-assoc*)
apply (*subst snd-lens-prod*)
using *comp-wb-lens fst-vwb-lens vwb-lens-wb* **apply** *blast*
apply (*simp add*: *alpha-in-var alpha-out-var*)
apply (*simp*)
done

5.4 Alphabet coercion

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

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

syntax
-alpha-coerce :: *salpha* ⇒ *logic* ⇒ *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*]:
$vwb\text{-}lens\ x \Longrightarrow id \oplus_s x = id$
by *pred-auto*

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

lemma *apply-subst-ext* [*alpha*]:
$vwb\text{-}lens\ x \Longrightarrow (\sigma \dagger e) \oplus_p x = (\sigma \oplus_s x) \dagger (e \oplus_p x)$
by (*pred-auto*)

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

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*]:
$mwb\text{-}lens\ x \Longrightarrow id \restriction_s x = id$
by *pred-auto*

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

lemma *subst-ext-res* [*alpha*,*usubst*]:
$vwb\text{-}lens\ x \Longrightarrow (\sigma \oplus_s x) \restriction_s x = \sigma$
by (*pred-auto*)

lemma *unrest-subst-alpha-ext* [*unrest*]:
$x \bowtie y \Longrightarrow x \sharp (P \oplus_s y)$
by (*pred-auto*, *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* :: $('a, '\alpha \times '\beta)\ uexpr \Rightarrow ('a, '\alpha)\ uexpr$ ($\lfloor\text{-}\rfloor_<$)
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* :: $('a, '\alpha \times '\beta)\ uexpr \Rightarrow ('a, '\beta)\ uexpr$ ($\lfloor\text{-}\rfloor_>$)
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-auto*)

end

7 Alphabetised Predicates

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

An alphabetised predicate is a simply a boolean valued expression

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

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

7.1 Automatic Tactics

named-theorems *upred-defs*

We set up several automatic tactics that recast theorems on UTP predicates into equivalent HOL predicates, eliminating artefacts of the mechanisation as much as this is possible. Our approach is first to unfold all relevant definition of the UTP predicate model, then perform a transfer, and finally simplify by using lens and variable definitions, the split laws of alphabet records, and interpretation laws to convert record-based state spaces into products. The definition of the methods is facilitated by the Eisbach tool.

Without re-interpretation of lens types in state spaces (legacy).

method *pred-simp'* = (
  (*unfold upred-defs*)?,
  (*transfer*),
  (*simp add*: *fun-eq-iff*
    lens-defs uvar-defs upred-defs alpha-splits Product-Type.split-beta)?,
  (*clarsimp*)?)

Variations that adjoin *pred-simp'* with automatic tactics.

method *pred-auto'* = (*pred-simp'*, *auto?*)
method *pred-blast'* = (*pred-simp'*; *blast*)

With reinterpretation of lens types in state spaces (default).

method *pred-simp* = (
  (*unfold upred-defs*)?,
  (*transfer*),
  (*simp add*: *fun-eq-iff*
    lens-defs uvar-defs upred-defs alpha-splits Product-Type.split-beta)?,
  (*simp add*: *lens-interp-laws*)?,
  (*clarsimp*)?)

Variations that adjoin *pred-simp* with automatic tactics.

method *pred-auto* = (*pred-simp*, *auto?*)
method *pred-blast* = (*pred-simp*; *blast*)

— TODO: Rename *pred-auto* into *pred-auto*.

7.2 Predicate syntax

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

no-notation
  conj (**infixr** ∧ *35*) **and**
  disj (**infixr** ∨ *30*) **and**
  Not (¬ - [*40*] *40*)

consts
  utrue :: *'a* (*true*)
  ufalse :: *'a* (*false*)
  uconj :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ∧ *35*)
  udisj :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ∨ *30*)
  uimpl :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ⇒ *25*)
  uiff :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ⇔ *25*)
  unot :: *'a* ⇒ *'a* (¬ - [*40*] *40*)
  uex :: (*'a*, *'α*) *uvar* ⇒ *'p* ⇒ *'p*
  uall :: (*'a*, *'α*) *uvar* ⇒ *'p* ⇒ *'p*
  ushEx :: [*'a* ⇒ *'p*] ⇒ *'p*
  ushAll :: [*'a* ⇒ *'p*] ⇒ *'p*

adhoc-overloading
  uconj conj **and**
  udisj disj **and**
  unot Not

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

syntax
  -uex :: *salpha* ⇒ *logic* ⇒ *logic* (∃ - · - [*0*, *10*] *10*)
  -uall :: *salpha* ⇒ *logic* ⇒ *logic* (∀ - · - [*0*, *10*] *10*)
  -ushEx :: *idt* ⇒ *logic* ⇒ *logic* (∃ - · - [*0*, *10*] *10*)
  -ushAll :: *idt* ⇒ *logic* ⇒ *logic* (∀ - · - [*0*, *10*] *10*)
  -ushBEx :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∃ - ∈ - · - [*0*, *0*, *10*] *10*)
  -ushBAll :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∀ - ∈ - · - [*0*, *0*, *10*] *10*)
  -ushGAll :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∀ - | - · - [*0*, *0*, *10*] *10*)

translations
  -uex x P == *CONST uex x P*
  -uall x P == *CONST uall x P*
  ∃ *x* · *P* == *CONST ushEx* (λ *x*. *P*)
  ∃ *x* ∈ *A* · *P* => ∃ *x* · «*x*» $\in_u$ *A* ∧ *P*
  ∀ *x* · *P* == *CONST ushAll* (λ *x*. *P*)
  ∀ *x* ∈ *A* · *P* => ∀ *x* · «*x*» $\in_u$ *A* ⇒ *P*
  ∀ *x* | *P* · *Q* => ∀ *x* · *P* ⇒ *Q*

7.3 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* ⇒ *'a* ⇒ *bool* (**infix** ⊑ *50*) **where**
P ⊑ *Q* ≡ *less-eq Q P*

Since, on the whole, lattices in UTP are the opposite way up to the standard definitions in HOL, we syntactically invert the lattice operators. This is the one exception where we do steal HOL syntax, but I think it makes sense for UTP.

no-notation *inf* (**infixl** ⊓ *70*)
notation *inf* (**infixl** ⊔ *70*)
no-notation *sup* (**infixl** ⊔ *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 λ*P Q A*. *sup* (*P A*) (*Q A*) **.**
lift-definition *inf-uexpr* :: (*'a*, *'b*) *uexpr* ⇒ (*'a*, *'b*) *uexpr* ⇒ (*'a*, *'b*) *uexpr*
is λ*P Q A*. *inf* (*P A*) (*Q A*) **.**
instance
by (*intro-classes*) (*transfer*, *auto*)+
end

instantiation *uexpr* :: (*bounded-lattice*, *type*) *bounded-lattice*
begin
lift-definition *bot-uexpr* :: (*'a*, *'b*) *uexpr* **is** 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*
apply (*intro-classes*, *unfold uexpr-defs*; *transfer*, *rule ext*)
apply (*simp-all add*: *sup-inf-distrib1 diff-eq*)
done

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

lift-definition *USUP* :: (*'a* ⇒ *'α upred*) ⇒ (*'a* ⇒ (*'b*::*complete-lattice*, *'α*) *uexpr*) ⇒ (*'b*, *'α*) *uexpr*

is $\lambda\ P\ F\ b.\ Sup\ \{[\![F\ x]\!]_e b \mid x.\ [\![P\ x]\!]_e b\}$ **.**

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

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

syntax
-USup :: *idt* ⇒ *logic* ⇒ *logic* ($\bigsqcap$ - · - [*0*, *10*] *10*)
-USup-mem :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigsqcap$ - ∈ - · - [*0*, *10*] *10*)
-USUP :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigsqcap$ - | - · - [*0*, *0*, *10*] *10*)
-UInf :: *idt* ⇒ *logic* ⇒ *logic* ($\bigsqcup$ - · - [*0*, *10*] *10*)
-UInf-mem :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigsqcup$ - ∈ - · - [*0*, *10*] *10*)
-UINF :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigsqcup$ - | - · - [*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**
$\lambda\ P\ Q\ A.\ P\ A \longrightarrow Q\ A$ **.**

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

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

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

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

adhoc-overloading

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

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

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

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 *unrest-usubst-def* [*upred-defs*]
declare *uexpr-defs* [*upred-defs*]

lemma *true-alt-def*: *true* = «*True*»
by (*pred-auto*)

lemma *false-alt-def*: *false* = «*False*»
by (*pred-auto*)

7.4 Unrestriction Laws

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

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

lemma *unrest-conj* [*unrest*]: $⟦ x \sharp (P :: 'α\ upred); x \sharp Q ⟧ \Longrightarrow x \sharp P \wedge Q$
by (*pred-auto*)

lemma *unrest-disj* [*unrest*]: $⟦ x \sharp (P :: 'α\ upred); x \sharp Q ⟧ \Longrightarrow x \sharp P \vee Q$
by (*pred-auto*)

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 *pred-auto*

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 *pred-auto*

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

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

lemma *unrest-not* [*unrest*]: $x \sharp (P :: '\alpha\ upred) \Longrightarrow x \sharp (\neg\ P)$
by (*pred-auto*)

The sublens proviso can be thought of as membership below.

lemma *unrest-ex-in* [*unrest*]:
$[\![$ *mwb-lens* $y;\ x \subseteq_L y]\!] \Longrightarrow x \sharp (\exists\ y \cdot P)$
by (*pred-auto*)

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

lemma *unrest-all-in* [*unrest*]:
$[\![$ *mwb-lens* $y;\ x \subseteq_L y]\!] \Longrightarrow x \sharp (\forall\ y \cdot P)$
by *pred-auto*

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

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

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

7.5 Substitution Laws

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

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

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

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

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

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

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

lemma *subst-sup* [*usubst*]: $\sigma \dagger (P \sqcap Q) = (\sigma \dagger P \sqcap \sigma \dagger Q)$
by (*pred-auto*)

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

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-auto*)

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-auto*)

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

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

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

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

lemma *subst-ex-same* [*usubst*]:
assumes *mwb-lens 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-auto*)
using *lens-indep-comm* **apply** *fastforce+*

done

lemma *subst-all-same* [*usubst*]:
assumes *mwb-lens* 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-auto*, *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-auto*+)

lemma *taut-true* [*simp*]: ‘*true*‘
by (*pred-auto*)

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

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

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

lemma *conj-comm*: $((P::'\alpha\ upred) \wedge Q) = (Q \wedge P)$
by *pred-auto*

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

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

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

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

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

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

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

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

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

lemma *true-disj-zero* [*simp*]:
$(P \vee true) = true$ $(true \vee P) = true$
by *pred-auto*

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

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

lemma *imp-true* [*simp*]: $(p \Rightarrow true) = true$
by *pred-auto*

lemma *true-imp* [*simp*]: $(true \Rightarrow p) = p$
by *pred-auto*

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

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

lemma *p-imp-p* [*simp*]: $(P \Rightarrow P) = true$
by *pred-auto*

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

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

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

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

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

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

lemma *true-not-false* [*simp*]: $true \neq false$ $false \neq true$
by *pred-auto+*

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

by *pred-auto*

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

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-auto*, *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-auto*)
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-auto*)
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-auto*)
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-auto*)
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-auto*)
apply (*simp add*: *Setcompr-eq-image*)
done

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

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

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

lemma *eq-upred-refl* [*simp*]: $(x =_u x) = true$
by *pred-auto*

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

lemma *eq-cong-left*:
assumes *vwb-lens* 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-auto*, (*meson mwb-lens-def vwb-lens-mwb weak-lens-def*)+)

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

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

lemma *conj-pos-var-subst*:
assumes *vwb-lens* x
shows $(\$x \wedge Q) = (\$x \wedge Q[\![true/\$x]\!])$
using *assms*
by (*pred-auto*, *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 *vwb-lens* x
shows $(\neg \$x \wedge Q) = (\neg \$x \wedge Q[\![false/\$x]\!])$
using *assms*
by (*pred-auto*, *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-auto*)

lemma *shEx-unbound* [*simp*]: $(\exists\ x \cdot P) = P$
by *pred-auto*

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

lemma *shEx-cong*: $\llbracket \bigwedge x.\ P\ x = Q\ x\ \rrbracket \Longrightarrow shEx\ P = shEx\ Q$
by (*pred-auto*)

lemma *shAll-unbound* [*simp*]: $(\forall\ x \cdot P) = P$
by *pred-auto*

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

lemma *shAll-cong*: $\llbracket \bigwedge x.\ P\ x = Q\ x\ \rrbracket \Longrightarrow shAll\ P = shAll\ Q$
by (*pred-auto*)

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

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

lemma *conj-var-subst*:
assumes *vwb-lens x*
shows $(P \wedge var\ x =_u v) = (P\llbracket v/x\rrbracket \wedge var\ x =_u v)$
using *assms*
by (*pred-auto*, (*metis* (*full-types*) *vwb-lens-def wb-lens.get-put*)+)

lemma *one-point*:
assumes *mwb-lens x x* $\sharp$ *v*
shows $(\exists\ x \cdot P \wedge var\ x =_u v) = P\llbracket v/x\rrbracket$
using *assms*
by (*pred-auto*)

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

lemma *uvar-obtain-assign*:
assumes *vwb-lens 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 *vwb-lens x*
shows $(P = Q) \longleftrightarrow (\forall\ v.\ P\llbracket \ll v\gg/x\rrbracket = Q\llbracket \ll v\gg/x\rrbracket)$
using *assms*
by (*pred-auto*, *metis uvar-assign-exists*)

lemma *eq-split-substI*:
assumes *vwb-lens x* $\bigwedge v.\ P\llbracket \ll v\gg/x\rrbracket = Q\llbracket \ll v\gg/x\rrbracket$
shows $P = Q$
using *assms*(*1*) *assms*(*2*) *eq-split-subst* **by** *blast*

lemma *taut-split-subst*:
assumes *vwb-lens x*
shows $`P` \longleftrightarrow (\forall\ v.\ `P\llbracket \ll v\gg/x\rrbracket`)$

using *assms*
by (*pred-auto*, *metis uvar-assign-exists*)

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

lemma *subst-bool-split*:
assumes *vwb-lens 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-auto*)
finally show *?thesis* **.**
qed

lemma *taut-iff-eq*:
$`P \Leftrightarrow Q` \longleftrightarrow (P = Q)$
by *pred-auto*

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-auto*

lemma *exists-twice*: *mwb-lens x* $\Longrightarrow (\exists\ x \cdot \exists\ x \cdot P) = (\exists\ x \cdot P)$
by (*pred-auto*)

lemma *all-twice*: *mwb-lens x* $\Longrightarrow (\forall\ x \cdot \forall\ x \cdot P) = (\forall\ x \cdot P)$
by (*pred-auto*)

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

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

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-auto*)
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-auto*)
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-auto*, *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-auto*, *metis* (*no-types*, *lifting*) *lens.select-convs*(*2*))

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

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

lemma *ex-plus*:
$(\exists\ y;x \cdot P) = (\exists\ x \cdot \exists\ y \cdot P)$
by *pred-auto*

lemma *all-plus*:
$(\forall\ y;x \cdot P) = (\forall\ x \cdot \forall\ y \cdot P)$
by *pred-auto*

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

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

7.7 Cylindric algebra

lemma *C1*: $(\exists\ x \cdot false) = false$
by (*pred-auto*)

lemma *C2*: *wb-lens* $x \Longrightarrow$ ‘$P \Rightarrow (\exists\ x \cdot P)$‘
by (*pred-auto*, *metis wb-lens.get-put*)

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

lemma *C4a*: $x \approx_L y \Longrightarrow (\exists\ x \cdot \exists\ y \cdot P) = (\exists\ y \cdot \exists\ x \cdot P)$
by (*pred-auto*, *metis* (*no-types*, *lifting*) *lens.select-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-auto*

lemma *C6*:
assumes *wb-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-auto*, (*metis lens-indep-def*)+)

lemma *C7*:
assumes *weak-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-auto′*, *simp add*: *lens-indep-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-auto*

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

8 Alphabetised relations

theory *utp-rel*
imports
utp-pred
utp-lift
begin

default-sort *type*

8.1 Automatic Tactics

named-theorems *urel-defs*

We set up several automatic tactics that recast theorems on UTP predicates into equivalent HOL predicates, eliminating artefacts of the mechanisation as much as this is possible. Our approach is first to unfold all relevant definition of the UTP predicate model, then perform a transfer, and finally simplify by using lens and variable definitions, the split laws of alphabet records, and interpretation laws to convert record-based state spaces into products. The definition of the methods is facilitated by the Eisbach tool.

Without re-interpretation of lens types in state spaces (legacy).

method *rel-simp′* = (
(*unfold upred-defs urel-defs*)?,

(*transfer*),
(*simp add*: *fun-eq-iff relcomp-unfold OO-def*
lens-defs uvar-defs upred-defs alpha-splits Product-Type.split-beta)*?*,
(*clarsimp*)*?*)

Variations that adjoin *rel-simp'* with automatic tactics.

method *rel-auto'* = (*rel-simp'*, *auto?*)
method *rel-blast'* = (*rel-simp'*; *blast*)

With reinterpretation of lens types in state spaces (default).

method *rel-simp* = (
(*unfold upred-defs urel-defs*)*?*,
(*transfer*),
(*simp add*: *fun-eq-iff relcomp-unfold OO-def*
lens-defs uvar-defs upred-defs alpha-splits Product-Type.split-beta)*?*,
(*simp add*: *lens-interp-laws*)*?*,
(*clarsimp*)*?*)

Variations that adjoin *rel-simp* with automatic tactics.

method *rel-auto* = (*rel-simp*, *auto?*)
method *rel-blast* = (*rel-simp*; *blast*)

— TODO: Rename *rel-auto* into *rel-auto*.

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

definition *inα* :: ($'\alpha$, $'\alpha \times '\beta$) *uvar* **where**
inα = (| *lens-get* = *fst*, *lens-put* = λ (A, A') v. (v, A') |)

definition *outα* :: ($'\beta$, $'\alpha \times '\beta$) *uvar* **where**
outα = (| *lens-get* = *snd*, *lens-put* = λ (A, A') v. (A, v) |)

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

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

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

lemma *out-alpha-in-indep* [*simp*]:
outα ⋈ *in-var x in-var x* ⋈ *outα*
by (*simp-all add*: *in-var-def outα-def lens-indep-def fst-lens-def lens-comp-def*)

lemma *in-alpha-out-indep* [*simp*]:
inα ⋈ *out-var x out-var x* ⋈ *inα*
by (*simp-all add*: *in-var-def inα-def lens-indep-def fst-lens-def lens-comp-def*)

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

lemma *alpha-in-out*:
$\Sigma \approx_L$ *inα* $+_L$ *outα*
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*::$'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred*
((*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* (-$^-$ [*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_-)

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

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

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

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

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

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

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

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

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

syntax
  -svid-unit :: *svid* $\Rightarrow$ *svid-list* (-)

-svid-list :: *svid* ⇒ *svid-list* ⇒ *svid-list* (-,/ -)
-uexpr-unit :: ('*a*, 'α) *uexpr* ⇒ *uexpr-list* (- [*40*] *40*)
-uexpr-list :: ('*a*, 'α) *uexpr* ⇒ *uexpr-list* ⇒ *uexpr-list* (-,/ - [*40*,*40*] *40*)
-assignment :: *svid-list* ⇒ *uexprs* ⇒ 'α *hrelation* (**infixr** := *62*)
-mk-usubst :: *svid-list* ⇒ *uexprs* ⇒ 'α *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 := *v* <= *CONST assigns-r* (*CONST subst-upd* (*CONST id*) *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

definition *rassume* :: 'α *upred* ⇒ 'α *hrelation* ($-^\top$ [*999*] *999*) **where**
[*urel-defs*]: *rassume c* = (*II* $\triangleleft$ *c* $\triangleright_r$ *false*)

definition *rassert* :: 'α *upred* ⇒ 'α *hrelation* ($-_\bot$ [*999*] *999*) **where**
[*urel-defs*]: *rassert c* = (*II* $\triangleleft$ *c* $\triangleright_r$ *true*)

We describe some properties of relations

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

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

definition *uinj* :: ('*a*, '*b*) *relation* ⇒ *bool*
where *uinj R* ⟷ *II* ⊑ (*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* :: 'α *condition* ⇒ 'α *hrelation* ($\lceil - \rceil_t$)
where $\lceil b \rceil_t$ = ($\lceil b \rceil_<$ ∧ *II*)

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

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

definition *rel-var-res* :: 'α *hrelation* ⇒ ('*a*, 'α) *uvar* ⇒ 'α *hrelation* (**infix** $\restriction_\alpha$ *80*) **where**
P $\restriction_\alpha$ *x* = (∃ \$*x* • ∃ \$*x*´ • *P*)

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

8.2 Unrestriction Laws

lemma *unrest-iuvar* [*unrest*]: *mwb-lens x* ⟹ *out*α ♯ \$*x*
by (*simp add*: *out*α*-def*, *transfer*, *auto*)

lemma *unrest-ouvar* [*unrest*]: *mwb-lens* $x \Longrightarrow in\alpha \sharp \$x'$
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-auto*)

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

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

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

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

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

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

lemma *unrest-out-rel-var-res* [*unrest*]:
  vwb-lens $x \Longrightarrow \$x\acute{} \sharp (P \upharpoonright_\alpha x)$
  by (*simp add*: *rel-var-res-def unrest*)

8.3 Substitution laws

lemma *subst-seq-left* [*usubst*]:
  $out\alpha \sharp \sigma \Longrightarrow \sigma \dagger (P ;; Q) = ((\sigma \dagger P) ;; Q)$
  by (*rel-auto*, (*metis* (*no-types*, *lifting*) *Pair-inject surjective-pairing*)+)

lemma *subst-seq-right* [*usubst*]:
  $in\alpha \sharp \sigma \Longrightarrow \sigma \dagger (P ;; Q) = (P ;; (\sigma \dagger Q))$
  by (*rel-auto*, (*metis* (*no-types*, *lifting*) *Pair-inject surjective-pairing*)+)

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

lemma *subst-skip-r* [*usubst*]:
  $out\alpha \sharp \sigma \Longrightarrow \sigma \dagger II = \langle\lfloor\sigma\rfloor_s\rangle_a$
  by (*rel-auto*, (*metis* (*mono-tags*, *lifting*) *prod.sel*(*1*) *sndI surjective-pairing*)+)

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*]:
  $\sigma(\&out\alpha{:}x \mapsto_s v) = \sigma(\$x\acute{} \mapsto_s v)$
  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-drop-upd* [*usubst*]:
  fixes $x :: ('a, '\alpha)\ uvar$
  shows $\lfloor\sigma(\$x \mapsto_s v)\rfloor_s = \lfloor\sigma\rfloor_s(x \mapsto_s \lfloor v\rfloor_<)$
  by (*pred-auto*, *simp add*: *inα-def prod.case-eq-if*)

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-auto*, *auto simp add*: *unrest-usubst-def inα-def*)

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

8.4 Relation laws

Homogeneous relations form a quantale. This allows us to import a large number of laws from Struth and Armstrong's Kleene Algebra theory [1].

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

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

interpretation *upred-quantale*: *unital-quantale-plus*
where *times* = *seqr* **and** *one* = *skip-r* **and** *Sup* = *Sup* **and** *Inf* = *Inf* **and** *inf* = *inf* **and** *less-eq* = *less-eq* **and** *less* = *less*
and *sup* = *sup* **and** *bot* = *bot* **and** *top* = *top*
apply (*unfold-locales*)
apply (*rel-auto*)
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-auto*)
apply (*rel-auto*)
done

lemma *drop-pre-inv* [*simp*]: $[\![$ *out*α $\sharp$ *p* $]\!] \Longrightarrow \lceil\lfloor p\rfloor_<\rceil_< = p$
by (*pred-auto*, *auto simp add*: *out*α*-def lens-create-def fst-lens-def prod.case-eq-if*)

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

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

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

While loops with invariant decoration

definition *while-inv* :: $'\alpha$ *condition* $\Rightarrow$ $'\alpha$ *condition* $\Rightarrow$ $'\alpha$ *hrelation* $\Rightarrow$ $'\alpha$ *hrelation* (*while - invr - do - od*) **where**
while b invr p do S od = *while b do S od*

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

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

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-auto*

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-auto*

lemma *cond-unit-T* [*simp*]:$(P \triangleleft true \triangleright Q) = P$ **by** *rel-auto*

lemma *cond-unit-F* [*simp*]:$(P \triangleleft false \triangleright Q) = Q$ **by** *rel-auto*

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

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

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

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-auto*

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-auto*

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-auto*

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-auto*

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

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

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

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

lemma *cond-var-subst-left*:
assumes *vwb-lens 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 *vwb-lens 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-var-split*:
vwb-lens x $\Longrightarrow (P[\![true/x]\!] \triangleleft var\ x \triangleright P[\![false/x]\!]) = P$
by (*rel-auto*, (*metis* (*full-types*) *vwb-lens.put-eq*)+)

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-auto*

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-auto*

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-auto*

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

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

lemma *seqr-left-zero* [*simp*]:
$(\mathit{false} \;;; \; P) = \mathit{false}$
by *pred-auto*

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

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-blast*)

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

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

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

lemma *skip-var*:
fixes $x :: (\mathit{bool}, \, '\alpha) \; \mathit{uvar}$
shows $(\$x \wedge II) = (II \wedge \$x')$
by (*rel-auto*)

lemma *seqr-exists-left*:
$\mathit{mwb\text{-}lens} \; x \Longrightarrow ((\exists \; \$x \cdot P) \;;; \; Q) = (\exists \; \$x \cdot (P \;;; \; Q))$
by (*rel-auto*)

lemma *seqr-exists-right*:
$\mathit{mwb\text{-}lens} \; x \Longrightarrow (P \;;; \; (\exists \; \$x' \cdot Q)) = (\exists \; \$x' \cdot (P \;;; \; Q))$
by (*rel-auto*)

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

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

lemma *assigns-r-feasible*:
$(\langle \sigma \rangle_a \;;; \; \mathit{true}) = \mathit{true}$
by (*rel-auto*)

lemma *assign-subst* [*usubst*]:
$[\![\mathit{mwb\text{-}lens} \; x; \; \mathit{mwb\text{-}lens} \; y]\!] \Longrightarrow [\$x \mapsto_s \lceil u \rceil_<] \dagger (y := v) = (x, \, y := u, \, [x \mapsto_s u] \dagger v)$
by *rel-auto*

lemma *assigns-idem*: *mwb-lens* $x \Longrightarrow (x,x := u,v) = (x := v)$
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-auto*, *simp-all add*: *bij-is-inj bij-is-surj surj-f-inv-f*)

lemma *assign-pred-transfer*:
fixes $x :: ('a, '\alpha)\ uvar$
assumes $\$x \sharp b\ out\alpha \sharp b$
shows $(b \wedge x := v) = (x := v \wedge b^-)$
using *assms* **by** (*rel-blast*)

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

lemma *assign-test*: *mwb-lens* $x \Longrightarrow (x := \ll u\gg ;; x := \ll v\gg) = (x := \ll v\gg)$
by (*simp add*: *assigns-comp subst-upd-comp subst-lit usubst-upd-idem*)

lemma *assign-twice*: $[\![$ *vwb-lens* 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-auto*, *simp-all add*: *lens-indep-comm*)

lemma *assign-cond*:
fixes $x :: ('a, '\alpha)\ uvar$
assumes $out\alpha \sharp b$
shows $(x := e ;; (P \triangleleft b \triangleright Q)) = ((x := e ;; P) \triangleleft (b[\![\lceil e\rceil_</\$x]\!]) \triangleright (x := e ;; Q))$
by *rel-auto*

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

lemma *assign-r-alt-def*:
fixes $x :: ('a, '\alpha)\ uvar$
shows $x := v = II[\![\lceil v\rceil_</\$x]\!]$
by *rel-auto*

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

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

lemma *assigns-r-swap-uinj*:
$[\![$ *vwb-lens* x; *vwb-lens* y; $x \bowtie y\ ]\!] \Longrightarrow$ *uinj* $(x,y := \&y,\&x)$

using *assigns-r-uinj swap-usubst-inj* **by** *auto*

lemma *skip-r-unfold*:
vwb-lens $x \Longrightarrow II = (\$x\acute{} =_u \$x \wedge II\restriction_\alpha x)$
by (*rel-auto*, *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-auto*)

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

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

lemma *assign-unfold*:
vwb-lens $x \Longrightarrow (x := v) = (\$x\acute{} =_u \lceil v\rceil_< \wedge II\restriction_\alpha x)$
apply (*rel-auto*, *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-auto*

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

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

lemma *seqr-and-distl-uinj*:
uinj $R \Longrightarrow ((P \wedge Q) ;; R) = ((P ;; R) \wedge (Q ;; R))$
by (*rel-auto*)

lemma *seqr-unfold*:
$(P ;; Q) = (\exists\ v \cdot P[\![\guillemotleft v\guillemotright/\$\Sigma\acute{}]\!] \wedge Q[\![\guillemotleft v\guillemotright/\$\Sigma]\!])$
by *rel-auto*

lemma *seqr-middle*:
assumes *vwb-lens* x
shows $(P ;; Q) = (\exists\ v \cdot P[\![\guillemotleft v\guillemotright/\$x\acute{}]\!] ;; Q[\![\guillemotleft v\guillemotright/\$x]\!])$
using *assms*
apply (*rel-auto*)
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 *vwb-lens x*
  shows $(P \wedge (\$x\acute{} =_u \ll v\gg) \;;\; Q) = (P[\![\ll v\gg/\$x\acute{}]\!] \;;\; Q[\![\ll v\gg/\$x]\!])$
  using *assms*
  by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*)

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

lemma *seqr-insert-ident-left*:
  assumes *vwb-lens x* $\$x\acute{} \sharp P$ $\$x \sharp Q$
  shows $((\$x\acute{} =_u \$x \wedge P) \;;\; Q) = (P \;;\; Q)$
  using *assms*
  by (*rel-auto*, *meson vwb-lens-wb wb-lens-weak weak-lens.put-get*)

lemma *seqr-insert-ident-right*:
  assumes *vwb-lens x* $\$x\acute{} \sharp P$ $\$x \sharp Q$
  shows $(P \;;\; (\$x\acute{} =_u \$x \wedge Q)) = (P \;;\; Q)$
  using *assms*
  by (*rel-auto*, *metis* (*no-types*, *hide-lams*) *vwb-lens-def wb-lens-def weak-lens.put-get*)

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

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

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

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

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

theorem *precond-left-zero*:
  assumes $\mathit{out}\alpha \sharp p$ $p \neq \mathit{false}$
  shows $(\mathit{true} \;;\; p) = \mathit{true}$
  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

8.5 Converse laws

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

by (*rel-auto*)

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-blast*)

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

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

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

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

8.6 Assertions and assumptions

lemma *assume-twice*: $(b^{\top} ;; c^{\top}) = (b \wedge c)^{\top}$
by (*rel-auto*)

lemma *assert-twice*: $(b_{\perp} ;; c_{\perp}) = (b \wedge c)_{\perp}$
by (*rel-auto*)

8.7 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*: $(x{:}[\![P]\!] \vee x{:}[\![Q]\!]) = x{:}[\![P \vee Q]\!]$
by (*rel-auto*)

lemma *frame-conj*: $(x{:}[\![P]\!] \wedge x{:}[\![Q]\!]) = x{:}[\![P \wedge Q]\!]$
by (*rel-auto*)

lemma *frame-seq*:

$[\![$ *vwb-lens* x; $\$x' \sharp P$; $\$x \sharp Q$ $]\!] \Longrightarrow (x:[\![P]\!] ;; x:[\![Q]\!]) = x:[\![P ;; Q]\!]$
by (*rel-auto*, *metis vwb-lens-def wb-lens-weak weak-lens.put-get*)

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

While loop laws

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

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

8.8 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*:
mwb-lens x $\Longrightarrow$ *RID*(*x*)(*RID*(*x*)(*P*)) = *RID*(*x*)(*P*)
by *rel-auto*

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

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

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

lemma *RID-conj*:
vwb-lens x $\Longrightarrow$ *RID*(*x*)(*RID*(*x*)(*P*) $\wedge$ *RID*(*x*)(*Q*)) = (*RID*(*x*)(*P*) $\wedge$ *RID*(*x*)(*Q*))
by *rel-auto*

lemma *RID-assigns-r-diff*:
$\llbracket$ *vwb-lens x*; *x* $\sharp$ σ $\rrbracket$ $\Longrightarrow$ *RID*(*x*)($\langle\sigma\rangle_a$) = $\langle\sigma\rangle_a$
apply (*rel-auto*)
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*:
vwb-lens x $\Longrightarrow$ *RID*(*x*)(*x* := *v*) = *II*
apply (*rel-auto*)
using *vwb-lens.put-eq* **apply** *fastforce*
done

lemma *RID-seq-left*:
assumes *vwb-lens 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-auto*)
also from *assms* **have** ... = $(((\exists\ \$x \cdot \exists\ \$x' \cdot P)\ ;;\ (\exists\ \$x \cdot \exists\ \$x' \cdot Q)) \wedge \$x' =_u \$x)$
apply (*rel-auto*)
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-auto*, *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-auto*, *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-auto*
also have ... = $(RID(x)(P)\ ;;\ RID(x)(Q))$
by *rel-auto*
finally show *?thesis* **.**
qed

lemma *RID-seq-right*:
assumes *vwb-lens 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-auto*)
also from *assms* **have** ... = $(((\exists\ \$x \cdot \exists\ \$x' \cdot P)\ ;;\ (\exists\ \$x \cdot \exists\ \$x' \cdot Q)) \wedge \$x' =_u \$x)$
apply (*rel-auto*)
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-auto*, *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-auto*, *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-auto*
also have ... = $(RID(x)(P)\ ;;\ RID(x)(Q))$
by *rel-auto*
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*]:
vwb-lens $x \Longrightarrow x\ \sharp\sharp\ II$
by (*simp add*: *RID-skip-r unrest-relation-def*)

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

lemma *seq-r-runrest* [*unrest*]:
assumes *vwb-lens* 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-auto*)

lemma *and-runrest* [*unrest*]: $[\![\; vwb\text{-}lens\ x;\ x \sharp\sharp P;\ x \sharp\sharp Q \;]\!] \Longrightarrow x \sharp\sharp (P \wedge Q)$
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*)

8.9 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-auto*

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

8.10 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-auto*

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-auto*)

8.11 Relational alphabet extension

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

lemma *rel-alpha-ext-alt-def*:
  assumes *vwb-lens* $y\ x +_L\ y \approx_L\ 1_L\ x \bowtie y$
  shows $P \oplus_R x = (P \oplus_p (x \times_L x) \wedge \$y' =_u \$y)$
  using *assms*
  apply (*rel-auto*, *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

8.12 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

8.13 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*]: $\llbracket\ \{\!|p|\!\}Q\{\!|r|\!\}_u;\ \{\!|p|\!\}Q\{\!|s|\!\}_u\ \rrbracket \Longrightarrow \{\!|p|\!\}Q\{\!|r \wedge s|\!\}_u$
  by *rel-auto*

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

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

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

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

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

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-auto*)
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-auto*)
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* $\Rightarrow$ *p*‘ ‘$(\neg b \wedge p) \Rightarrow$ *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

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

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

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-auto*

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

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

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

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

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-auto*

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

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

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

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

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

lemma *while-false-trel*:
  assumes $\sigma \dagger b = false$
  shows $(\sigma,\ while\ b\ do\ P\ od) \to_u (\sigma,\ II)$
  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* ($[\![\text{-}]\!]$)
where $[\![H]\!] \equiv \{P.\ P\ is\ H\}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

definition *WeakConjunctive* :: $'\alpha$ *Healthiness-condition* $\Rightarrow$ *bool* **where**

$WeakConjunctive(H) \longleftrightarrow (\forall\ P.\ \exists\ Q.\ H(P) = (P \wedge Q))$

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

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

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

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

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* × ′α)

overloading
  rel-hcond == *utp-hcond* :: (*REL* × ′α) *itself* ⇒ (′α × ′α) *Healthiness-condition*
  rel-unit == *utp-unit* :: (*REL* × ′α) *itself* ⇒ ′α *hrelation*
begin
  definition *rel-hcond* :: (*REL* × ′α) *itself* ⇒ (′α × ′α) *upred* ⇒ (′α × ′α) *upred* **where**
  rel-hcond T = *id*

  definition *rel-unit* :: (*REL* × ′α) *itself* ⇒ ′α *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*)

lemma *utp-partial-order*: *partial-order* (*utp-order T*)
  by (*unfold-locales*, *simp-all add*: *utp-order-def*)

lemma *mono-Monotone-utp-order*:
  mono f ⟹ *Monotone* (*utp-order T*) *f*
  apply (*auto simp add*: *isotone-def*)
  apply (*metis partial-order-def utp-partial-order*)
  apply (*simp add*: *utp-order-def*)
  apply (*metis monoD*)
done

end

11 Example UTP theory: Boyle's laws

In order to exemplify the use of Isabelle/UTP, we mechanise a simple theory representing Boyle's law. Boyle's law states that, for an ideal gas at fixed temperature, pressure p is inversely proportional to volume V, or more formally that for $k = p \cdot V$ is invariant, for constant k. 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*

declare *alpha-boyle.splits* [*alpha-splits*]

The two locale interpretations below are a technicality to improve automatic proof support via the predicate and relational tactics. This is to enable the (re-)interpretation of state spaces to remove any occurrences of lens types after the proof tactics *pred-simp* and *rel-simp*, or any of their derivatives have been applied. Eventually, it would be desirable to automate both interpretations as part of a custom outer command for defining alphabets.

interpretation *alpha-boyle-prd*: — Closed records are sufficient here.
  lens-interp λr::*alpha-boyle*. (*boyle-k* r, *boyle-p* r, *boyle-V* r)
apply (*unfold-locales*)
apply (*rule injI*)
apply (*clarsimp*)
done

interpretation *alpha-boyle-rel*: — Closed records are sufficient here.
  lens-interp λ(r::*alpha-boyle*, r'::*alpha-boyle*).
    (*boyle-k* r, *boyle-k* r', *boyle-p* r, *boyle-p* r', *boyle-V* r, *boyle-V* r')
apply (*unfold-locales*)
apply (*rule injI*)
apply (*clarsimp*)
done

For now we have to explicitly cast the fields to lenses using the VAR syntactic transformation function [3] – in the future this will be automated. We also have to add the definitional equations for these variables to the simplification set for predicates to enable automated proof through our tactics.

definition k :: *real* $\Longrightarrow$ *alpha-boyle* **where** k = *VAR boyle-k*
definition p :: *real* $\Longrightarrow$ *alpha-boyle* **where** p = *VAR boyle-p*
definition V :: *real* $\Longrightarrow$ *alpha-boyle* **where** V = *VAR boyle-V*

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

We also prove that our new lenses are well-behaved and independent of each other. A selection of these properties are shown below.

lemma *vwb-lens-k* [*simp*]: *vwb-lens* k
  by (*unfold-locales*, *simp-all add*: *k-def*)
lemma *boyle-indeps* [*simp*]:
  $k \bowtie p$ $p \bowtie k$ $k \bowtie V$ $V \bowtie k$ $p \bowtie V$ $V \bowtie p$
  by (*simp-all add*: *k-def p-def V-def lens-indep-def*)

11.1 Static invariant

We first create a simple UTP theory representing Boyle's laws on a single state, as a static invariant healthiness condition. We state Boyle's law using the function B, which recalculates the value of the constant k based on p and V.

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

We can then prove that B is both idempotent and monotone simply by application of the predicate tactic. Idempotence means that healthy predicates cannot be made more healthy. Together with idempotence, monotonicity ensures that image of the healthiness functions forms a complete lattice, which is useful to allow the representation of recursive and iterative constructions with the theory.

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

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

We also create some example observations; the first (φ_1) satisfies Boyle's law and the second doesn't (φ_2).

definition $\varphi_1 = ((\&p =_u \mathit{10}) \wedge (\&V =_u \mathit{5}) \wedge (\&k =_u \mathit{50}))$
definition $\varphi_2 = ((\&p =_u \mathit{10}) \wedge (\&V =_u \mathit{5}) \wedge (\&k =_u \mathit{100}))$

We first prove an obvious property: that these two predicates are different observations. We must show that there exists a valuation of one which is not of the other. This is achieved through application of *pred-tac*, followed by *sledgehammer* [2] which yields a *metis* proof.

lemma φ_1*-diff-*φ_2: $\varphi_1 \neq \varphi_2$
  by (*pred-auto*, *metis select-convs num.distinct*(*5*) *numeral-eq-iff semiring-norm*(*87*))

We prove that φ_1 satisfies Boyle's law by application of the predicate calculus tactic, *pred-tac*.

lemma *B-*φ_1: φ_1 *is* B
  by (*pred-auto*)

We prove that φ_2 does not satisfy Boyle's law by showing that applying B to it results in φ_1. We prove this using Isabelle's natural proof language, Isar.

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

11.2 Dynamic invariants

Next we build a relational theory that allows the pressure and volume to be changed, whilst still respecting Boyle's law. We create two dynamic invariants for this purpose.

definition $\mathit{D1}(P) = ((\$k =_u \$p{\cdot}\$V \Rightarrow \$k' =_u \$p'{\cdot}\$V') \wedge P)$
definition $\mathit{D2}(P) = (\$k' =_u \$k \wedge P)$

D1 states that if Boyle's law satisfied in the previous state, then it should be satisfied in the next state. We define this by conjunction of the formal specification of this property with the predicate. The annotations $\$p$ and $\$p'$ refer to relational variables p and p'. *D2* states that the constant k indeed remains constant throughout the evolution of the system, which is also specified as a conjunctive healthiness condition. As before we demonstrate that *D1* and *D2* are both idempotent and monotone.

lemma *D1-idempotent*: $\mathit{D1}(\mathit{D1}(P)) = \mathit{D1}(P)$ **by** *rel-auto*

lemma *D2-idempotent*: $D2(D2(P)) = D2(P)$ **by** *rel-auto*

lemma *D1-monotone*: $X \sqsubseteq Y \Longrightarrow D1(X) \sqsubseteq D1(Y)$ **by** *rel-auto*
lemma *D2-monotone*: $X \sqsubseteq Y \Longrightarrow D2(X) \sqsubseteq D2(Y)$ **by** *rel-auto*

Since these properties are relational, we discharge them using our relational calculus tactic *rel-tac*. Next we specify three operations that make up the signature of the theory.

definition *InitSys ip iV*
$= ((\ll ip \gg >_u 0 \wedge \ll iV \gg >_u 0)^\top \;;\; p, V, k := \ll ip \gg, \ll iV \gg, (\ll ip \gg \cdot \ll iV \gg))$

definition *ChPres dp*
$= ((\&p + \ll dp \gg >_u 0)^\top \;;\; p := \&p + \ll dp \gg \;;\; V := (\&k / \&p))$

definition *ChVol dV*
$= ((\&V + \ll dV \gg >_u 0)^\top \;;\; V := \&V + \ll dV \gg \;;\; p := (\&k / \&V))$

InitSys initialises the system with a given initial pressure (ip) and volume (iV). It assumes that both are greater than 0 using the assumption construct $c^\top$ which equates to II if c is true and *false* (i.e. errant) otherwise. It then creates a state assignment for p and V, uses the B healthiness condition to make it healthy (by calculating k), and finally turns the predicate into a postcondition using the $\lceil P \rceil_>$ function.

ChPres raises or lowers the pressure based on an input dp. It assumes that the resulting pressure change would not result in a zero or negative pressure, i.e. $p + dp > 0$. It assigns the updated value to p and recalculates V using the original value of k. *ChVol* is similar but updates the volume.

lemma *D1-InitSystem*: $D1$ $(InitSys\ ip\ iV) = InitSys\ ip\ iV$
by *rel-auto*

InitSys is *D1*, since it establishes the invariant for the system. However, it is not *D2* since it sets the global value of k and thus can change its value. We can however show that both *ChPres* and *ChVol* are healthy relations.

lemma *D1*: $D1$ $(ChPres\ dp) = ChPres\ dp$ **and** $D1$ $(ChVol\ dV) = ChVol\ dV$
by (*rel-auto*, *rel-auto*)

lemma *D2*: $D2$ $(ChPres\ dp) = ChPres\ dp$ **and** $D2$ $(ChVol\ dV) = ChVol\ dV$
by (*rel-auto*, *rel-auto*)

Finally we show a calculation a simple animation of Boyle's law, where the initial pressure and volume are set to 10 and 4, respectively, and then the pressure is lowered by 2.

lemma *ChPres-example*:
$(InitSys\ 10\ 4 \;;\; ChPres\ (-2)) = p, V, k := 8, 5, 40$
proof −
— *InitSys* yields an assignment to the three variables
have $InitSys\ 10\ 4 = p, V, k := 10, 4, 40$
by (*rel-auto*)
— This assignment becomes a substitution
hence $(InitSys\ 10\ 4 \;;\; ChPres\ (-2))$
$= (ChPres\ (-2))[\![10, 4, 40 / \$p, \$V, \$k]\!]$
by (*simp add*: *assigns-r-comp alpha*)
— Unfold definition of *ChPres*
also have ... $= ((\&p - 2 >_u 0)^\top[\![10, 4, 40 / \$p, \$V, \$k]\!]$
$;; p := \&p - 2 \;;\; V := \&k / \&p)$
by (*simp add*: *ChPres-def lit-num-simps usubst unrest*)

— Unfold definition of assumption
also have ... = $((p, V, k := 10, 4, 40 \lhd (8 :_u \mathit{real}) >_u 0 \rhd \mathit{false})$
$;; p := \&p - 2 ;; V := \&k / \&p)$
by (*simp add*: *rassume-def usubst alpha unrest*)
— $(0::'a) < (8::'a)$ is true; simplify conditional
also have ... = $(p, V, k := 10, 4, 40 ;; p := \&p - 2 ;; V := \&k / \&p)$
by *rel-auto*
— Application of both assignments
also have ... = $p, V, k := 8, 5, 40$
by *rel-auto*
finally show *?thesis* **.**
qed

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* = ok_v :: *bool*

declare *alpha-d.splits* [*alpha-splits*]

The two locale interpretations below are a technicality to improve automatic proof support via the predicate and relational tactics. This is to enable the (re-)interpretation of state spaces to remove any occurrences of lens types after the proof tactics *pred-simp* and *rel-simp*, or any of their derivatives have been applied. Eventually, it would be desirable to automate both interpretations as part of a custom outer command for defining alphabets.

interpretation *alpha-d*: *lens-interp* $\lambda r.\ (ok_v\ r,\ more\ r)$
apply (*unfold-locales*)
apply (*rule injI*)
apply (*clarsimp*)
done

interpretation *alpha-d-rel*:
lens-interp $\lambda(r, r').\ (ok_v\ r,\ ok_v\ r',\ more\ r,\ more\ r')$
apply (*unfold-locales*)
apply (*rule injI*)
apply (*clarsimp*)
done

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

definition $ok = VAR\ ok_v$

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

lemma *vwb-lens-ok* [*simp*]: *vwb-lens 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**
[*uvar-defs*]: *des-lens* $= (\!|$ *lens-get* $=$ *more*, *lens-put* $=$ *fld-put more-update* $|\!)$

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

translations
  -svid-alpha-d => Σ_D

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

lemma *ok-indep-des-lens* [*simp*]: $ok \bowtie$ *des-lens des-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-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* $(\lceil - \rceil_D)$
where $\lceil P \rceil_D \equiv P \oplus_p$ (*des-lens* $\times_L$ *des-lens*)

abbreviation *lift-pre-desr* :: $'\alpha$ *upred* $\Rightarrow ('\alpha, '\beta)$ *relation-d* $(\lceil - \rceil_{D<})$
where $\lceil p \rceil_{D<} \equiv \lceil \lceil p \rceil_< \rceil_D$

abbreviation *lift-post-desr* :: $'\beta$ *upred* $\Rightarrow ('\alpha, '\beta)$ *relation-d* $(\lceil - \rceil_{D>})$
where $\lceil p \rceil_{D>} \equiv \lceil \lceil p \rceil_> \rceil_D$

abbreviation *drop-desr* :: $('\alpha, '\beta)$ *relation-d* $\Rightarrow ('\alpha, '\beta)$ *relation* $(\lfloor - \rfloor_D)$
where $\lfloor P \rfloor_D \equiv P \restriction_p$ (*des-lens* $\times_L$ *des-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* σ = (*true* $\vdash_r$ *assigns-r* σ)

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

translations
  -assignmentd xs vs => CONST assigns-d (-mk-usubst (CONST id) xs vs)
  $x :=_D v$ *<= CONST assigns-d (CONST subst-upd (CONST id) (CONST svar x) v)*
  $x :=_D v$ *<= CONST assigns-d (CONST subst-upd (CONST id) x v)*
  $x,y :=_D u,v$ *<= CONST assigns-d (CONST subst-upd (CONST subst-upd (CONST id) (CONST svar x) u) (CONST svar y) v)*

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

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

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

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

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

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

translations
  $P^f \rightleftharpoons$ *CONST usubst (CONST subst-upd CONST id (CONST ovar CONST ok) false) P*
  $P^t \rightleftharpoons$ *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* :: $('\alpha, '\beta)$ *relation-d* $\Rightarrow$ $('\alpha, '\beta)$ *relation* ($pre_D'(\text{-}')$) **where**
$pre_D(P) = \lfloor \neg P[\![true,false/\$ok,\$ok']\!] \rfloor_D$

definition *post-design* :: $('\alpha, '\beta)$ *relation-d* $\Rightarrow$ $('\alpha, '\beta)$ *relation* ($post_D'(\text{-}')$) **where**
$post_D(P) = \lfloor P[\![true,true/\$ok,\$ok']\!] \rfloor_D$

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

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

declare *rdesign-def* [*upred-defs*]
declare *ndesign-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 :: ($'\alpha$, $'\beta$) *relation-d*
  assumes $\$ok \sharp P$ $\$ok' \sharp P$
  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*) :: (-, $'\alpha$ *alpha-d-scheme* $\times$ $'\beta$ *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*]:
  $a \bowtie x \Longrightarrow a \times_L b \bowtie$ *in-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*]:
  $b \bowtie x \Longrightarrow a \times_L b \bowtie$ *out-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-auto*, *auto simp add*: *outα-def des-lens-def prod-lens-def*)

thm *alpha-d.select-convs*

lemma *lift-dist-seq* [*simp*]:
$\lceil P \;;; Q \rceil_D = (\lceil P \rceil_D \;;; \lceil Q \rceil_D)$
by (*rel-auto*)

lemma *lift-des-skip-dr-unit-unrest*: $\$ok' \sharp P \Longrightarrow (P \;;; \lceil II \rceil_D) = P$
by (*rel-auto*)

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

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

lemma *design-false-pre*:
$(false \vdash P) = true$
by *rel-auto*

lemma *rdesign-false-pre*:
$(false \vdash_r P) = true$
by *rel-auto*

lemma *ndesign-false-pre*:
$(false \vdash_n P) = true$
by *rel-auto*

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-auto*
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-auto*)
finally show *?thesis* **.**
qed

theorem *rdesign-refinement*:
$(P1 \vdash_r Q1 \sqsubseteq P2 \vdash_r Q2) \longleftrightarrow (`P1 \Rightarrow P2` \wedge `P1 \wedge Q2 \Rightarrow Q1`)$
by *rel-auto*

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-auto*

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

lemma *ndesign-refine-intro*:
assumes ‘$p1 \Rightarrow p2$‘ ‘$\lceil p1 \rceil_< \wedge Q2 \Rightarrow Q1$‘
shows $p1 \vdash_n Q1 \sqsubseteq p2 \vdash_n Q2$
using *assms* **unfolding** *upred-defs*
by *pred-auto*

lemma *design-subst* [*usubst*]:
$[\![\$ok \sharp \sigma; \$ok' \sharp \sigma]\!] \Longrightarrow \sigma \dagger (P \vdash Q) = (\sigma \dagger P) \vdash (\sigma \dagger Q)$
by (*simp add*: *design-def usubst*)

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

theorem *design-npre*:
$(P \vdash Q)^f = (\neg \$ok \vee \neg P^f)$
by (*rel-auto*)

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*)

theorem *design-post*:
$(P \vdash Q)^t = ((\$ok \wedge P^t) \Rightarrow Q^t)$
by (*rel-auto*)

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

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

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-auto*)
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-auto*

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-auto*

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-auto*

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-auto*

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-auto*

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

lemma *design-bottom*:
  $\perp_D \sqsubseteq (P \vdash Q)$
  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-auto*

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-auto*

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 $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (\exists\ ok_0 \cdot ((P1 \vdash Q1)[\![\ll ok_0 \gg/\$ok']\!] ;; (P2 \vdash Q2)[\![\ll ok_0 \gg/\$ok]\!]))$
    by (*rule seqr-middle*, *simp*)
  also have ...
      $= (((P1 \vdash Q1)[\![false/\$ok']\!] ;; (P2 \vdash Q2)[\![false/\$ok]\!])$
        $\vee ((P1 \vdash Q1)[\![true/\$ok']\!] ;; (P2 \vdash Q2)[\![true/\$ok]\!]))$
    by (*simp add*: *true-alt-def false-alt-def*, *pred-auto*)
  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-auto*)
  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-auto*)
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-auto*)
finally show *?thesis* **.**
qed

lemma *design-export-ok*:
$P \vdash Q = (P \vdash (\$ok \wedge Q))$
by (*rel-auto*)

lemma *design-export-ok'*:
$P \vdash Q = (P \vdash (\$ok' \wedge Q))$
by (*rel-auto*)

theorem *design-composition*:
assumes
$\$ok' \sharp P1$ $\$ok \sharp P2$ $\$ok' \sharp Q1$ $\$ok \sharp Q2$
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 $(\$x \wedge P) = (P \wedge \$x')$
proof −
have $P = (\$x' =_u \$x \wedge P)$
by (*metis* (*no-types*, *lifting*) *RID-def assms conj-idem unrest-relation-def utp-pred.inf.left-commute*)
moreover have $(\$x' =_u \$x \wedge (\$x \wedge P)) = (\$x' =_u \$x \wedge (P \wedge \$x'))$
by (*rel-auto*)
ultimately show *?thesis*
by (*metis utp-pred.inf.assoc utp-pred.inf-left-commute*)
qed

theorem *design-composition-runrest*:
assumes
$\$ok' \sharp P1$ $\$ok \sharp P2$ $ok \sharp\sharp Q1$ $ok \sharp\sharp Q2$
shows $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (((\neg ((\neg P1) ;; true)) \wedge \neg (Q1^t ;; (\neg P2))) \vdash (Q1 ;; Q2))$
proof −
have $(\$ok \wedge \$ok' \wedge (Q1^t ;; Q2[\![true/\$ok]\!])) = (\$ok \wedge \$ok' \wedge (Q1 ;; Q2))$
proof −
have $(\$ok \wedge \$ok' \wedge (Q1 ;; Q2)) = (\$ok \wedge Q1 ;; Q2 \wedge \$ok')$
by (*metis* (*no-types*, *hide-lams*) *seqr-post-out seqr-pre-out utp-pred.inf.commute utp-rel.unrest-iuvar utp-rel.unrest-ouvar vwb-lens-ok vwb-lens-mwb*)
also have ... $= (Q1 \wedge \$ok' ;; \$ok \wedge Q2)$
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 vwb-lens-ok vwb-lens-mwb*)
finally show *?thesis*
by (*metis utp-pred.inf.left-commute utp-pred.inf-left-idem*)
qed
moreover have $(\neg (\neg P1 ;; true) \wedge \neg (Q1^t ;; \neg P2)) \vdash (Q1^t ;; Q2[\![true/\$ok]\!]) =$
$(\neg (\neg P1 ;; true) \wedge \neg (Q1^t ;; \neg P2)) \vdash (\$ok \wedge \$ok' \wedge (Q1^t ;; Q2[\![true/\$ok]\!]))$
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-auto*)

theorem *design-skip-idem* [*simp*]:
$(II_D ;; II_D) = II_D$
by (*rel-auto*)

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*:
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** (*rel-blast*)

theorem *rdesign-composition-wp*:
$((\lceil p1 \rceil_< \vdash_r Q1) ;; (\lceil p2 \rceil_< \vdash_r Q2)) = ((\lceil p1 \wedge Q1\ wp\ p2 \rceil_<) \vdash_r (Q1 ;; Q2))$
by *rel-blast*

theorem *ndesign-composition-wp*:
$((p1 \vdash_n Q1) ;; (p2 \vdash_n Q2)) = ((p1 \wedge Q1\ wp\ p2) \vdash_n (Q1 ;; Q2))$
by *rel-blast*

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

theorem *ndesign-wp* [*wp*]:
$(p \vdash_n Q)\ wp_D\ r = (p \wedge Q\ wp\ r)$
by (*simp add*: *ndesign-def rdesign-wp*)

theorem *wpd-seq-r*:
fixes $Q1\ Q2 :: '\alpha\ hrelation$
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-auto*)
done

theorem *wpnd-seq-r* [*wp*]:
fixes *Q1 Q2* :: *'α hrelation*
shows $(p1 \vdash_n Q1 ;; p2 \vdash_n Q2)\ wp_D\ r = (p1 \vdash_n Q1)\ wp_D\ ((p2 \vdash_n Q2)\ wp_D\ r)$
by (*simp add*: *ndesign-def wpd-seq-r*)

lemma *design-subst-ok-ok'*:
$(P[\![true/\$ok]\!] \vdash Q[\![true,true/\$ok,\$ok']\!]) = (P \vdash Q)$
proof −
have $(P \vdash Q) = ((\$ok \wedge P) \vdash (\$ok \wedge \$ok' \wedge Q))$
by (*pred-auto*)
also have ... $= ((\$ok \wedge P[\![true/\$ok]\!]) \vdash (\$ok \wedge (\$ok' \wedge Q[\![true/\$ok']\!])[\![true/\$ok]\!]))$
by (*metis conj-eq-out-var-subst conj-pos-var-subst upred-eq-true utp-pred.inf-commute vwb-lens-ok*)
also have ... $= ((\$ok \wedge P[\![true/\$ok]\!]) \vdash (\$ok \wedge \$ok' \wedge Q[\![true,true/\$ok,\$ok']\!]))$
by (*simp add*: *usubst*)
also have ... $= (P[\![true/\$ok]\!] \vdash Q[\![true,true/\$ok,\$ok']\!])$
by (*pred-auto*)
finally show *?thesis* **..**
qed

lemma *design-subst-ok'*:
$(P \vdash Q[\![true/\$ok']\!]) = (P \vdash Q)$
proof −
have $(P \vdash Q) = (P \vdash (\$ok' \wedge Q))$
by (*pred-auto*)
also have ... $= (P \vdash (\$ok' \wedge Q[\![true/\$ok']\!]))$
by (*metis conj-eq-out-var-subst upred-eq-true utp-pred.inf-commute vwb-lens-ok*)
also have ... $= (P \vdash Q[\![true/\$ok']\!])$
by (*pred-auto*)
finally show *?thesis* **..**
qed

theorem *design-left-unit-hom*:
fixes *P Q* :: *'α hrelation-d*
shows $(II_D ;; P \vdash_r Q) = (P \vdash_r Q)$
proof −
have $(II_D ;; P \vdash_r Q) = (true \vdash_r II ;; P \vdash_r Q)$
by (*simp add*: *skip-d-def*)
also have ... $= (true \wedge \neg (II ;; \neg P)) \vdash_r (II ;; Q)$
proof −
have $out\alpha \sharp true$
by *unrest-tac*
thus *?thesis*
using *rdesign-composition-cond* **by** *blast*
qed
also have ... $= (\neg (\neg P)) \vdash_r Q$
by *simp*
finally show *?thesis* **by** *simp*
qed

theorem *design-left-unit* [*simp*]:
$(II_D ;; P \vdash_r Q) = (P \vdash_r Q)$

by *rel-auto*

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-auto rel-auto*

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

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-auto*)

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

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-auto*)

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-auto*)

12.4 H1: No observation is allowed before initiation

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

lemma *H1-monotone*:
$P \sqsubseteq Q \Longrightarrow H1(P) \sqsubseteq H1(Q)$
by *pred-auto*

lemma *H1-below-top*:
$H1(P) \sqsubseteq \top_D$
by *pred-auto*

lemma *H1-design-skip*:
$H1(II) = II_D$
by *rel-auto*

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-auto*

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 (*simp add*: *upred-defs urel-defs*)
qed

theorem *H1-left-unit*:

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

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

theorem *H1-nok-left-zero*:
fixes P :: $'\alpha$ *hrelation-d*
assumes P *is* $H1$
shows $(\neg\ \$ok\ ;;\ P) = (\neg\ \$ok)$
proof −
have $(\neg\ \$ok\ ;;\ P) = ((\neg\ \$ok\ ;;\ true_h)\ ;;\ P)$
by (*simp add*: *precond-right-unit unrest*)
also have ... $= ((\neg\ \$ok)\ ;;\ true_h)$
by (*metis H1-left-zero assms seqr-assoc*)
also have ... $= (\neg\ \$ok)$
by (*simp add*: *precond-right-unit unrest*)
finally show *?thesis* **.**
qed

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

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

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-blast*

lemma *H1-USUP*:
assumes $A \neq \{\}$
shows $H1(\bigsqcap\ i \in A \cdot P(i)) = (\bigsqcap\ i \in A \cdot H1(P(i)))$

using *assms* **by** (*rel-auto*)

lemma *H1-Sup*:
assumes $A \neq \{\}$ $\forall$ $P \in A$. *P is H1*
shows ($\bigsqcap$ *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-auto*)

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-auto*
also have ... $= ((P \;;;\; (\neg \$ok \wedge \lceil II \rceil_D)) \vee (P \;;;\; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))))$
by *rel-auto*
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-auto*
also have ... $= (\exists \$ok' \cdot P \wedge \$ok' =_u \mathit{false})$
by *rel-auto*
also have ... $= P^f$
by (*metis C1 one-point out-var-uvar pr-var-def unrest-as-exists vwb-lens-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-auto*
also have ... $= (P^t \wedge \$ok')$
by *rel-auto*
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-auto*
  also have ... = ‘$(P^f \Rightarrow P^t)$‘
    by *pred-auto*
  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-auto*)

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 *rel-auto*

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 ($subst\text{-}tac$)
also have ... = $(\neg \$ok)$
by *pred-auto*
finally show *?thesis* **.**
qed

lemma *H2-true*: $H2(true) = true$
by (*rel-auto*)

lemma *H2-choice-closed*:
$[\![\ P\ is\ H2;\ Q\ is\ H2\ ]\!] \Longrightarrow P \sqcap Q\ is\ H2$
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-auto*)
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-auto*)

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-auto*
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-auto*
finally show *?thesis* **by** *simp*
qed

lemma *ok-pre*: $(\$ok \wedge \lceil pre_D(P) \rceil_D) = (\$ok \wedge (\neg P^f))$
apply (*pred-auto*)
done

lemma *ok-post*: $(\$ok \wedge \lceil post_D(P) \rceil_D) = (\$ok \wedge (P^t))$
apply (*pred-auto*)
done

theorem *H1-H2-eq-design*:
$H1\ (H2\ P) = (\neg\ P^f) \vdash P^t$
proof −
have $H1\ (H2\ P) = (\$ok \Rightarrow H2(P))$
by (*simp add*: *H1-def*)
also have ... $= (\$ok \Rightarrow (P^f \vee (P^t \wedge \$ok')))$
by (*metis H2-split*)
also have ... $= (\$ok \wedge (\neg\ P^f) \Rightarrow \$ok' \wedge \$ok \wedge P^t)$
by *rel-auto*
also have ... $= (\neg\ P^f) \vdash P^t$
by *rel-auto*
finally show *?thesis* **.**
qed

theorem *H1-H2-is-design*:
assumes *P is H1 P is H2*
shows $P = (\neg\ P^f) \vdash P^t$
using *assms* **by** (*metis H1-H2-eq-design Healthy-def*)

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-auto*
also have ... $= (\$ok \wedge (\neg\ P^f) \Rightarrow \$ok' \wedge \$ok \wedge P^t)$
by *pred-auto*
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-auto*
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-auto*)
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 $(\bigsqcup \ldots) = (\bigsqcup\ P \in A.\ \mathit{H1\text{-}H2}(P))$
by *auto*
also have $\ldots = (\bigsqcup\ P \in A \cdot \mathit{H1\text{-}H2}(P))$
by (*simp add*: *UINF-as-Inf-collect*)
also have $\ldots = (\bigsqcup\ P \in A \cdot (\neg\ P^f) \vdash P^t)$
by (*meson H1-H2-eq-design*)
also have $\ldots = (\bigsqcap\ P \in A \cdot \neg\ P^f) \vdash (\bigsqcup\ P \in A \cdot \neg\ P^f \Rightarrow P^t)$
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 $((p \vdash Q) ;; \mathit{II}_D) = (\neg\ (\neg\ p ;; true)) \vdash (Q^t ;; \mathit{II}[\![true/\$ok]\!])$
by (*simp add*: *skip-d-alt-def design-composition-subst unrest assms*)
also have $\ldots = p \vdash (Q^t ;; \mathit{II}[\![true/\$ok]\!])$
using *assms precond-equiv seqr-true-lemma* **by** *force*
also have $\ldots = p \vdash Q$
by (*rel-auto*)
finally show *?thesis*
by (*simp add*: *H3-def Healthy-def'*)
qed

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

also have ... $= (\neg (\neg P ;; true)) \vdash_r Q$
by *pred-auto*
finally have $P \vdash_r Q$ *is H3* $\longleftrightarrow P \vdash_r Q = (\neg (\neg P ;; true)) \vdash_r Q$
by (*metis H3-def Healthy-def'*)
also have ... $\longleftrightarrow P = (\neg (\neg P ;; true))$
by (*metis rdesign-pre*)
also have ... $\longleftrightarrow P = (P ;; true)$
by (*simp add*: *seqr-true-lemma*)
finally show *?thesis* **.**
qed

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

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

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-auto*
also have ... $= II_D$
by *rel-auto*
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 vwb-lens-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 *H3-ndesign*:
  $H3(p \vdash_n Q) = (p \vdash_n Q)$
  by (*simp add*: *H3-def ndesign-def unrest-pre-outα*)

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-auto*)
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-auto*)
finally show *?thesis* **.**
qed

lemma *H3-unrest-out-alpha* [*unrest*]: P *is H1-H3* $\Longrightarrow out\alpha \sharp pre_D(P)$
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-auto*

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

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-auto*)
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* × ′α *alphabet-d*) *itself* ⇒ ′α *hrelation-d* **where**
ndes-unit t = II_D

end

interpretation *des-utp-theory*: *utp-theory TYPE*(*DES* × ′α *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* × ′α *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* × ′α *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* × (′α *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* × ′α *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* ≡ $\mu_{utp\text{-}order\ DES}$ *F*

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

13 Concurrent programming

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

no-notation
Sublist.parallel (**infixl** ∥ *50*)

13.1 Design parallel composition

definition *design-par* :: ('α, 'β) *relation-d* ⇒ ('α, 'β) *relation-d* ⇒ ('α, 'β) *relation-d* (**infixr** ∥ *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-auto*)
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-auto*
also have ... = *true*
by *rel-auto*
finally show *?thesis* **.**
qed

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

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

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-auto+*)
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 ⋈ *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 vwb-des-lens vwb-lens-mwb*)
done

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

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

lemma *right-uvar* [*simp*]: *vwb-lens x* ⟹ *vwb-lens* (*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*; $i \neq j$⟧ ⟹ *ind-uvar i x* ⋈ *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 vwb-des-lens vwb-lens-mwb*)
done

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

lemma *out-ind-uvar-mwb-lens* [*simp*]:
  mwb-lens x $\Longrightarrow$ *mwb-lens* (*out-ind-uvar i x*)
  by (*simp add*: *out-ind-uvar-def*)

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

syntax
  -svarpre :: *svid* $\Rightarrow$ *svid* (*-*$_<$ [*999*] *999*)
  -svarleft :: *svid* $\Rightarrow$ *svid* (*0*−*-* [*999*] *999*)
  -svarright :: *svid* $\Rightarrow$ *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* = ($'\alpha \times {}'\alpha$ *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* $\Rightarrow$ $'\alpha$ *merge* $\Rightarrow$ $'\alpha$ *hrelation-d* $\Rightarrow$ $'\alpha$ *hrelation-d* (**infixr** $\|_-$ *85*)
where $P \parallel_M Q$ = ((((*P* ;; *U0*) $\|$ (*Q* ;; *U1*))) ;; *M*)

swap is a predicate that the swaps the left and right indices; it is used to specify commutativity of the parallel operator

definition $swap_m$ = (*0*−Σ,*1*−Σ $:=_D$ &*1*−Σ, &*0*−Σ)

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*
  by (*rel-auto*)

lemma *U1-H1-H2*: *U1 is H1-H2*
  by (*simp add*: *U1-def rdesign-is-H1-H2*)

lemma *U1-swap*: $(U1\ ;;\ swap_m) = U0$
by $(rel\text{-}auto)$

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 $(((P_1 \vdash_r P_2) \parallel (Q_1 \vdash_r Q_2))\ ;;\ swap_m) =$
$(\neg\ (\neg\ P_1 \lor \neg\ Q_1\ ;;\ true)) \vdash_r ((P_1 \Rightarrow P_2) \land (Q_1 \Rightarrow Q_2)\ ;;\ \langle[\&0{-}\Sigma \mapsto_s \&1{-}\Sigma,\ \&1{-}\Sigma \mapsto_s \&0{-}\Sigma]\rangle_a)$
by (*simp add*: *design-par-def* $swap_m$*-def assigns-d-def rdesign-composition*)
also have ... $= \ (\neg\ (\neg\ P_1 \lor \neg\ Q_1\ ;;\ true)) \vdash_r (((P_1 \Rightarrow P_2)\ ;;\ \langle[\&0{-}\Sigma \mapsto_s \&1{-}\Sigma,\ \&1{-}\Sigma \mapsto_s \&0{-}\Sigma]\rangle_a) \land ((Q_1 \Rightarrow Q_2)\ ;;\ \langle[\&0{-}\Sigma \mapsto_s \&1{-}\Sigma,\ \&1{-}\Sigma \mapsto_s \&0{-}\Sigma]\rangle_a))$
by $(rel\text{-}auto)$
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 assigns-d-def rdesign-composition*, *rel-auto*)
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 $(rel\text{-}auto)$
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 $(rel\text{-}auto)$
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

record *'t*::*ordered-cancel-monoid-diff alpha-rp'* =
$wait_v$ :: *bool*
tr_v :: *'t*

declare *alpha-rp'.splits* [*alpha-splits*]

The two locale interpretations below are a technicality to improve automatic proof support via the predicate and relational tactics. This is to enable the (re-)interpretation of state spaces to remove any occurrences of lens types after the proof tactics *pred-simp* and *rel-simp*, or any of their derivatives have been applied. Eventually, it would be desirable to automate both interpretations as part of a custom outer command for defining alphabets.

interpretation *alphabet-rp*:
lens-interp $\lambda(ok, r). (ok, wait_v\ r, tr_v\ r, more\ r)$
apply (*unfold-locales*)
apply (*rule injI*)
apply (*clarsimp*)
done

interpretation *alphabet-rp-rel*: *lens-interp* $\lambda(ok, ok', r, r')$.
$(ok, ok', wait_v\ r, wait_v\ r', tr_v\ r, tr_v\ r', more\ r, more\ r')$

apply (*unfold-locales*)
apply (*rule injI*)
apply (*clarsimp*)
done

type-synonym ($'t$, $'\alpha$) *alpha-rp-scheme* = ($'t$, $'\alpha$) *alpha-rp′-scheme alpha-d-scheme*

type-synonym ($'t$,$'\alpha$) *alphabet-rp* = ($'t$,$'\alpha$) *alpha-rp-scheme alphabet*
type-synonym ($'t$,$'\alpha$,$'\beta$) *relation-rp* = (($'t$,$'\alpha$) *alphabet-rp*, ($'t$,$'\beta$) *alphabet-rp*) *relation*
type-synonym ($'t$,$'\alpha$) *hrelation-rp* = (($'t$,$'\alpha$) *alphabet-rp*, ($'t$,$'\alpha$) *alphabet-rp*) *relation*
type-synonym ($'t$,$'\sigma$) *predicate-rp* = ($'t$,$'\sigma$) *alphabet-rp upred*

translations
  (*type*) ($'t$, $'\alpha$) *alphabet-rp* <= (*type*) ($'t$, $'\alpha$) *alpha-rp′-scheme alpha-d-ext*
  (*type*) ($'t$, $'\alpha$) *alphabet-rp* <= (*type*) ($'t$, $'\alpha$) *alpha-rp′-ext alpha-d-ext*

definition $wait_r$ = *VAR* $wait_v$
definition tr_r = *VAR* tr_v
definition Σ_r = *VAR more*

declare $wait_r$*-def* [*uvar-defs*]
declare tr_r*-def* [*uvar-defs*]
declare Σ_r*-def* [*uvar-defs*]

lemma $wait_r$*-vwb-lens* [*simp*]: *vwb-lens* $wait_r$
  by (*unfold-locales*, *simp-all add*: $wait_r$*-def*)

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

lemma *rea-vwb-lens* [*simp*]: *vwb-lens* Σ_r
  by (*unfold-locales*, *simp-all add*: Σ_r*-def*)

definition [*uvar-defs*]: *wait* = ($wait_r$ $;_L$ Σ_D)
definition [*uvar-defs*]: *tr* = (tr_r $;_L$ Σ_D)
definition [*uvar-defs*]: Σ_R = (Σ_r $;_L$ Σ_D)

lemma *wait-vwb-lens* [*simp*]: *vwb-lens wait*
  by (*simp add*: *wait-def*)

lemma *tr-vwb-lens* [*simp*]: *vwb-lens tr*
  by (*simp add*: *tr-def*)

lemma *rea-lens-vwb-lens* [*simp*]: *vwb-lens* Σ_R
  by (*simp add*: Σ_R*-def*)

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

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 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 *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-var-ords* [*usubst*]:
$\$tr \prec_v \$tr'$ $\$wait \prec_v \$wait'$
$\$ok \prec_v \$tr$ $\$ok' \prec_v \$tr'$ $\$ok \prec_v \$tr'$ $\$ok' \prec_v \$tr$
$\$ok \prec_v \$wait$ $\$ok' \prec_v \$wait'$ $\$ok \prec_v \$wait'$ $\$ok' \prec_v \$wait$
$\$tr \prec_v \$wait$ $\$tr' \prec_v \$wait'$ $\$tr \prec_v \$wait'$ $\$tr' \prec_v \$wait$
by (*simp-all add*: *var-name-ord-def*)

abbreviation *wait-f*::($'t$::*ordered-cancel-monoid-diff*, $'\alpha$, $'\beta$) *relation-rp* $\Rightarrow$ ($'t$, $'\alpha$, $'\beta$) *relation-rp*
where *wait-f* $R \equiv R[\![false/\$wait]\!]$

abbreviation *wait-t*::($'t$::*ordered-cancel-monoid-diff*, $'\alpha$, $'\beta$) *relation-rp* $\Rightarrow$ ($'t$, $'\alpha$, $'\beta$) *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*

abbreviation *lift-rea* :: - $\Rightarrow$ - ($\lceil\text{-}\rceil_R$) **where**
$\lceil P\rceil_R \equiv P \oplus_p (\Sigma_R \times_L \Sigma_R)$

abbreviation *drop-rea* :: ($'t$::*ordered-cancel-monoid-diff*, $'\alpha$, $'\beta$) *relation-rp* $\Rightarrow$ ($'\alpha$, $'\beta$) *relation* ($\lfloor\text{-}\rfloor_R$)
where
$\lfloor P\rfloor_R \equiv P \restriction_p (\Sigma_R \times_L \Sigma_R)$

abbreviation *rea-pre-lift* :: - $\Rightarrow$ - ($\lceil\text{-}\rceil_{R<}$) **where** $\lceil n\rceil_{R<} \equiv \lceil\lceil n\rceil_<\rceil_R$

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

14.1 Reactive lemmas

lemma *unrest-ok-lift-rea* [*unrest*]:
$\$ok \sharp \lceil P \rceil_R$ $\$ok' \sharp \lceil P \rceil_R$
by (*pred-auto*)+

lemma *unrest-wait-lift-rea* [*unrest*]:
$\$wait \sharp \lceil P \rceil_R$ $\$wait' \sharp \lceil P \rceil_R$
by (*pred-auto*)+

lemma *unrest-tr-lift-rea* [*unrest*]:
$\$tr \sharp \lceil P \rceil_R$ $\$tr' \sharp \lceil P \rceil_R$
by (*pred-auto*)+

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

14.2 R1: Events cannot be undone

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

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

lemma *R1-mono*: $P \sqsubseteq Q \Longrightarrow R1(P) \sqsubseteq R1(Q)$
by *pred-auto*

lemma *R1-unrest* [*unrest*]: $\llbracket\ x \bowtie in\text{-}var\ tr;\ x \bowtie out\text{-}var\ tr;\ x \sharp P\ \rrbracket \Longrightarrow x \sharp R1(P)$
by (*metis R1-def in-var-uvar lens-indep-sym out-var-uvar tr-vwb-lens unrest-bop unrest-conj unrest-var*)

lemma *R1-false*: $R1(false) = false$
by *pred-auto*

lemma *R1-conj*: $R1(P \wedge Q) = (R1(P) \wedge R1(Q))$
by *pred-auto*

lemma *R1-disj*: $R1(P \vee Q) = (R1(P) \vee R1(Q))$
by *pred-auto*

lemma *R1-USUP*:
$R1(\bigsqcap\ i \in A \cdot P(i)) = (\bigsqcap\ i \in A \cdot R1(P(i)))$
by (*rel-auto*)

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-auto*)

lemma *R1-extend-conj*: $R1(P \wedge Q) = (R1(P) \wedge Q)$
by *pred-auto*

lemma *R1-extend-conj'*: $R1(P \wedge Q) = (P \wedge R1(Q))$
by *pred-auto*

lemma *R1-cond*: $R1(P \triangleleft b \triangleright Q) = (R1(P) \triangleleft b \triangleright R1(Q))$
by *rel-auto*

lemma *R1-negate-R1*: $R1(\neg\ R1(P)) = R1(\neg\ P)$
  by *pred-auto*

lemma *R1-wait-true*: $(R1\ P)_t = R1(P)_t$
  by *pred-auto*

lemma *R1-wait-false*: $(R1\ P)\ _f = R1(P)\ _f$
  by *pred-auto*

lemma *R1-skip*: $R1(II) = II$
  by *rel-auto*

lemma *R1-skip-rea*: $R1(II_r) = II_r$
  by *rel-auto*

lemma *R1-by-refinement*:
  $P\ is\ R1 \longleftrightarrow ((\$tr \le_u \$tr') \sqsubseteq P)$
  by *rel-blast*

lemma *tr-le-trans*:
  $(\$tr \le_u \$tr'\ ;;\ \$tr \le_u \$tr') = (\$tr \le_u \$tr')$
  by (*rel-auto*)

lemma *R1-seqr*:
  $R1(R1(P)\ ;;\ R1(Q)) = (R1(P)\ ;;\ R1(Q))$
  by (*rel-auto*)

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-true-comp*: $(R1(true)\ ;;\ R1(true)) = R1(true)$
  by (*rel-auto*)

lemma *R1-ok'-true*: $(R1(P))^t = R1(P^t)$
  by *pred-auto*

lemma *R1-ok'-false*: $(R1(P))^f = R1(P^f)$
  by *pred-auto*

lemma *R1-ok-true*: $(R1(P))[\![true/\$ok]\!] = R1(P[\![true/\$ok]\!])$
  by *pred-auto*

lemma *R1-ok-false*: $(R1(P))[\![false/\$ok]\!] = R1(P[\![false/\$ok]\!])$
  by *pred-auto*

lemma *seqr-R1-true-right*: $((P\ ;;\ R1(true)) \vee P) = (P\ ;;\ (\$tr \le_u \$tr'))$
  by *rel-auto*

lemma *R1-extend-conj-unrest*: $[\![\ \$tr \sharp\ Q;\ \$tr' \sharp\ Q\ ]\!] \Longrightarrow R1(P \wedge Q) = (R1(P) \wedge Q)$
  by *pred-auto*

lemma *R1-extend-conj-unrest'*: $[\![\$tr \sharp P; \$tr' \sharp P]\!] \Longrightarrow R1(P \wedge Q) = (P \wedge R1(Q))$
by *pred-auto*

lemma *R1-tr'-eq-tr*: $R1(\$tr' =_u \$tr) = (\$tr' =_u \$tr)$
by (*rel-auto*)

lemma *R1-H2-commute*: $R1(H2(P)) = H2(R1(P))$
by (*simp add*: *H2-split R1-def usubst*, *rel-auto*)

14.3 R2

definition *R2a-def* [*upred-defs*]: $R2a\ (P) = (\bigsqcap\ s \cdot P[\![\ll s\gg,\ll s\gg+(\$tr'-\$tr)/\$tr,\$tr']\!])$
definition *R2s-def* [*upred-defs*]: $R2s\ (P) = (P[\![0/\$tr]\!][\![(\$tr'-\$tr)/\$tr']\!])$
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-auto*

lemma *R2s-R2a*: $R2s(R2a(P)) = R2a(P)$
by *rel-auto*

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-auto*)

lemma *R2s-unrest* [*unrest*]: $[\![$ *vwb-lens* x; $x \bowtie$ *in-var* tr; $x \bowtie$ *out-var* tr; $x \sharp P]\!] \Longrightarrow x \sharp R2s(P)$
by (*simp add*: *R2s-def unrest usubst lens-indep-sym*)

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

lemma *R2-mono*: $P \sqsubseteq Q \Longrightarrow R2(P) \sqsubseteq R2(Q)$
by (*pred-auto*)

lemma *R2s-conj*: $R2s(P \wedge Q) = (R2s(P) \wedge R2s(Q))$
by (*pred-auto*)

lemma *R2-conj*: $R2(P \wedge Q) = (R2(P) \wedge R2(Q))$
by (*pred-auto*)

lemma *R2s-disj*: $R2s(P \vee Q) = (R2s(P) \vee R2s(Q))$
by *pred-auto*

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-auto*)

lemma $R2s$-not: $R2s(\neg\ P) = (\neg\ R2s(P))$
  by *pred-auto*

lemma $R2s$-$condr$: $R2s(P \triangleleft b \triangleright Q) = (R2s(P) \triangleleft R2s(b) \triangleright R2s(Q))$
  by *rel-auto*

lemma $R2$-$condr$: $R2(P \triangleleft b \triangleright Q) = (R2(P) \triangleleft R2(b) \triangleright R2(Q))$
  by *rel-auto*

lemma $R2$-$condr'$: $R2(P \triangleleft b \triangleright Q) = (R2(P) \triangleleft R2s(b) \triangleright R2(Q))$
  by *rel-auto*

lemma $R2s$-ok: $R2s(\$ok) = \$ok$
  by *rel-auto*

lemma $R2s$-ok': $R2s(\$ok\acute{}) = \$ok\acute{}$
  by *rel-auto*

lemma $R2s$-$wait$: $R2s(\$wait) = \$wait$
  by *rel-auto*

lemma $R2s$-$wait'$: $R2s(\$wait\acute{}) = \$wait\acute{}$
  by *rel-auto*

lemma $R2s$-$true$: $R2s(true) = true$
  by *pred-auto*

lemma $R2s$-$false$: $R2s(false) = false$
  by *pred-auto*

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 $R2c$-$true$: $R2c(true) = true$
  by *rel-auto*

lemma $R2c$-$false$: $R2c(false) = false$
  by *rel-auto*

lemma $R2c$-and: $R2c(P \wedge Q) = (R2c(P) \wedge R2c(Q))$
  by (*rel-auto*)

lemma $R2c$-$disj$: $R2c(P \vee Q) = (R2c(P) \vee R2c(Q))$
  by (*rel-auto*)

lemma $R2c$-not: $R2c(\neg\ P) = (\neg\ R2c(P))$
  by (*rel-auto*)

lemma $R2c$-ok: $R2c(\$ok) = (\$ok)$
  by (*rel-auto*)

lemma *R2c-ok'*: $R2c(\$ok\acute{}) = (\$ok\acute{})$
  by (*rel-auto*)

lemma *R2c-wait*: $R2c(\$wait) = \$wait$
  by (*rel-auto*)

lemma *R2c-tr'-minus-tr*: $R2c(\$tr\acute{} =_u \$tr) = (\$tr\acute{} =_u \$tr)$
  apply (*rel-auto*) **using** *minus-zero-eq* **by** *blast*

lemma *R2c-tr'-ge-tr*: $R2c(\$tr\acute{} \geq_u \$tr) = (\$tr\acute{} \geq_u \$tr)$
  by (*rel-auto*)

lemma *R2c-condr*: $R2c(P \triangleleft b \triangleright Q) = (R2c(P) \triangleleft R2c(b) \triangleright R2c(Q))$
  by (*rel-auto*)

lemma *R2c-skip-r*: $R2c(II) = II$
proof −
  have $R2c(II) = R2c(\$tr\acute{} =_u \$tr \wedge II\restriction_\alpha tr)$
    by (*subst skip-r-unfold*[*of tr*], *simp-all*)
  also have ... $= (R2c(\$tr\acute{} =_u \$tr) \wedge II\restriction_\alpha tr)$
    by (*simp add*: *R2c-and*, *simp add*: *R2c-def R2s-def usubst unrest cond-idem*)
  also have ... $= (\$tr\acute{} =_u \$tr \wedge II\restriction_\alpha tr)$
    by (*simp add*: *R2c-tr'-minus-tr*)
  finally show *?thesis*
    by (*subst skip-r-unfold*[*of tr*], *simp-all*)
qed

lemma *R1-R2c-commute*: $R1(R2c(P)) = R2c(R1(P))$
  by (*rel-auto*)

lemma *R1-R2c-is-R2*: $R1(R2c(P)) = R2(P)$
  by (*rel-auto*)

lemma *R2c-skip-rea*: $R2c\ II_r = II_r$
  by (*simp add*: *skip-rea-def R2c-and R2c-disj R2c-skip-r R2c-not R2c-ok R2c-tr'-ge-tr*)

lemma *R1-R2s-R2c*: $R1(R2s(P)) = R1(R2c(P))$
  by (*rel-auto*)

lemma *R2-skip-rea*: $R2(II_r) = II_r$
  by (*metis R1-R2c-is-R2 R1-skip-rea R2c-skip-rea*)

lemma *R2-tr-prefix*: $R2(\$tr \leq_u \$tr\acute{}) = (\$tr \leq_u \$tr\acute{})$
  by (*pred-auto*)

lemma *R2-form*:
  $R2(P) = (\exists\ tt \cdot P[\![0/\$tr]\!][\![\ll tt\gg/\$tr\acute{}]\!] \wedge \$tr\acute{} =_u \$tr + \ll tt\gg)$
  apply (*rel-auto*)
  apply (*metis cancel-monoid-add-class.add-diff-cancel-left' ordered-cancel-monoid-diff-class.le-iff-add*)
  using *ordered-cancel-monoid-diff-class.le-iff-add* **apply** *blast*
done

lemma *R2-seqr-form*:
  shows $(R2(P)\ ;;\ R2(Q)) =$

$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!]) ;; (Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!]))$
$\wedge (\$tr´ =_u \$tr + «tt_1» + «tt_2»))$

proof −
have $(R2(P) ;; R2(Q)) = (\exists\ tr_0 \cdot (R2(P))[\![«tr_0»/\$tr´]\!] ;; (R2(Q))[\![«tr_0»/\$tr]\!])$
by (*subst seqr-middle*[*of tr*], *simp-all*)
also have ... =
$(\exists\ tr_0 \cdot \exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!] \wedge «tr_0» =_u \$tr + «tt_1») ;;$
$(Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!] \wedge \$tr´ =_u «tr_0» + «tt_2»)))$
by (*simp add*: *R2-form usubst unrest uquant-lift*, *rel-blast*)
also have ... =
$(\exists\ tr_0 \cdot \exists\ tt_1 \cdot \exists\ tt_2 \cdot ((«tr_0» =_u \$tr + «tt_1» \wedge P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!]) ;;$
$(Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!] \wedge \$tr´ =_u «tr_0» + «tt_2»)))$
by (*simp add*: *conj-comm*)
also have ... =
$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot \exists\ tr_0 \cdot ((P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!]) ;; (Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!]))$
$\wedge «tr_0» =_u \$tr + «tt_1» \wedge \$tr´ =_u «tr_0» + «tt_2»)$
by *rel-blast*
also have ... =
$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!]) ;; (Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!]))$
$\wedge (\exists\ tr_0 \cdot «tr_0» =_u \$tr + «tt_1» \wedge \$tr´ =_u «tr_0» + «tt_2»))$
by *rel-auto*
also have ... =
$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot ((P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!]) ;; (Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!]))$
$\wedge (\$tr´ =_u \$tr + «tt_1» + «tt_2»))$
by *rel-auto*
finally show *?thesis* **.**
qed

lemma *R2-seqr-distribute*:
fixes P :: ($'t$::*ordered-cancel-monoid-diff*,$'\alpha$,$'\beta$) *relation-rp* **and** Q :: ($'t$,$'\beta$,$'\gamma$) *relation-rp*
shows $R2(R2(P) ;; R2(Q)) = (R2(P) ;; R2(Q))$
proof −
have $R2(R2(P) ;; R2(Q)) =$
$((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!] ;; Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!])[\![(\$tr´ - \$tr)/\$tr´]\!]$
$\wedge \$tr´ - \$tr =_u «tt_1» + «tt_2») \wedge \$tr´ \geq_u \$tr)$
by (*simp add*: *R2-seqr-form*, *simp add*: *R2s-def usubst unrest*, *rel-auto*)
also have ... =
$((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!] ;; Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!])[\![(«tt_1» + «tt_2»)/\$tr´]\!]$
$\wedge \$tr´ - \$tr =_u «tt_1» + «tt_2») \wedge \$tr´ \geq_u \$tr)$
by (*subst subst-eq-replace*, *simp*)
also have ... =
$((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!] ;; Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!])$
$\wedge \$tr´ - \$tr =_u «tt_1» + «tt_2») \wedge \$tr´ \geq_u \$tr)$
by (*rel-auto*)
also have ... =
$(\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!] ;; Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!])$
$\wedge (\$tr´ - \$tr =_u «tt_1» + «tt_2» \wedge \$tr´ \geq_u \$tr))$
by *pred-auto*
also have ... =
$((\exists\ tt_1 \cdot \exists\ tt_2 \cdot (P[\![0/\$tr]\!][\![«tt_1»/\$tr´]\!] ;; Q[\![0/\$tr]\!][\![«tt_2»/\$tr´]\!])$
$\wedge \$tr´ =_u \$tr + «tt_1» + «tt_2»))$
proof −
have $\bigwedge\ tt_1\ tt_2.\ (((\$tr´ - \$tr =_u «tt_1» + «tt_2») \wedge \$tr´ \geq_u \$tr)$ :: ($'t$,$'\alpha$,$'\gamma$) *relation-rp*)
$= (\$tr´ =_u \$tr + «tt_1» + «tt_2»)$
apply (*rel-auto*)

apply (*metis add.assoc cancel-monoid-add-class.add-diff-cancel-left' ordered-cancel-monoid-diff-class.le-iff-add*)
apply (*simp add*: *add.assoc*)
using *add.assoc ordered-cancel-monoid-diff-class.le-iff-add* **by** *blast*
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-auto*

lemma *R2-R1-form*: *R2*(*R1*(*P*)) = *R1*(*R2s*(*P*))
by (*rel-auto*)

lemma *R2s-H1-commute*:
R2s(*H1*(*P*)) = *H1*(*R2s*(*P*))
by *rel-auto*

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-auto*

lemma *R2c-idem*: *R2c*(*R2c*(*P*)) = *R2c*(*P*)
by (*rel-auto*)

lemma *R2c-H2-commute*: *R2c*(*H2*(*P*)) = *H2*(*R2c*(*P*))
by (*simp add*: *H2-split R2c-disj R2c-def R2s-def usubst*, *rel-auto*)

lemma *R2c-seq*: *R2c*(*R2*(*P*) ;; *R2*(*Q*)) = (*R2*(*P*) ;; *R2*(*Q*))
by (*metis* (*no-types*, *lifting*) *R1-R2c-commute R1-R2c-is-R2 R2-seqr-distribute R2c-idem*)

lemma *R2-R2c-def*: *R2*(*P*) = *R1*(*R2c*(*P*))
by *rel-auto*

lemma *R2c-R1-seq*: *R2c*(*R1*(*R2c*(*P*)) ;; *R1*(*R2c*(*Q*))) = (*R1*(*R2c*(*P*)) ;; *R1*(*R2c*(*Q*)))
using *R2c-seq*[*of P Q*] **by** (*simp add*: *R2-R2c-def*)

14.4 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-auto*

lemma *R3-mono*: $P \sqsubseteq Q \Longrightarrow R3(P) \sqsubseteq R3(Q)$
by *rel-auto*

lemma *R3-conj*: $R3(P \wedge Q) = (R3(P) \wedge R3(Q))$
by *rel-auto*

lemma *R3-disj*: $R3(P \vee Q) = (R3(P) \vee R3(Q))$
by *rel-auto*

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-auto*)

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-auto*)

lemma *R3-condr*: $R3(P \triangleleft b \triangleright Q) = (R3(P) \triangleleft b \triangleright R3(Q))$
by *rel-auto*

lemma *R3-skipr*: $R3(II) = II$
by *rel-auto*

lemma *R3-form*: $R3(P) = ((\$wait \wedge II) \vee (\neg\ \$wait \wedge P))$
by *rel-auto*

lemma *wait-R3*:
$(\$wait \wedge R3(P)) = (II \wedge \$wait')$
by (*rel-auto*)

lemma *nwait-R3*:
$(\neg\$wait \wedge R3(P)) = (\neg\$wait \wedge P)$
by (*rel-auto*)

lemma *R3-semir-form*:
$(R3(P)\ ;;\ R3(Q)) = R3(P\ ;;\ R3(Q))$
by *rel-auto*

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-simp*, *safe*, *auto intro*: *order-trans*)

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-vwb-lens*)

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

lemma *R1-R3c-commute*: $R1(R3c(P)) = R3c(R1(P))$
by *rel-auto*

lemma *R2-R3-commute*: $R2(R3(P)) = R3(R2(P))$
by (*rel-auto*, (*smt add.right-neutral alpha-d.surjective alpha-d.update-convs*(*2*) *alpha-rp′.surjective alpha-rp′.update-convs*(*2*) *cancel-monoid-add-class.add-diff-cancel-left′ ordered-cancel-monoid-diff-class.le-iff-add*)+)

lemma *R2-R3c-commute*: $R2(R3c(P)) = R3c(R2(P))$
by (*rel-auto*, (*smt add.right-neutral alpha-d.surjective alpha-d.update-convs*(*2*) *alpha-rp′.surjective alpha-rp′.update-convs*(*2*) *cancel-monoid-add-class.add-diff-cancel-left′ ordered-cancel-monoid-diff-class.le-iff-add*)+)

lemma *R2c-R3c-commute*: $R2c(R3c(P)) = R3c(R2c(P))$
by (*simp add*: *R3c-def R2c-condr R2c-wait R2c-skip-rea*)

lemma *R1-H1-R3c-commute*:
$R1(H1(R3c(P))) = R3c(R1(H1(P)))$
by *rel-auto*

lemma *R3c-H2-commute*: $R3c(H2(P)) = H2(R3c(P))$
by (*simp add*: *H2-split R3c-def usubst*, *rel-auto*)

lemma *R3c-idem*: $R3c(R3c(P)) = R3c(P)$
by *rel-auto*

lemma *R3c-conj*: $R3c(P \wedge Q) = (R3c(P) \wedge R3c(Q))$
by (*rel-auto*)

lemma *R3c-disj*: $R3c(P \vee Q) = (R3c(P) \vee R3c(Q))$
by *rel-auto*

lemma *R3c-USUP*:
assumes $A \neq \{\}$
shows $R3c(\bigsqcap\ i \in A \cdot P(i)) = (\bigsqcap\ i \in A \cdot R3c(P(i)))$
using *assms* **by** (*rel-auto*)

lemma *R3c-UINF*:
assumes $A \neq \{\}$
shows $R3c(\bigsqcup\ i \in A \cdot P(i)) = (\bigsqcup\ i \in A \cdot R3c(P(i)))$
using *assms* **by** (*rel-auto*)

14.5 RH laws

definition *RH-def* [*upred-defs*]: $RH(P) = R1(R2s(R3c(P)))$

notation *RH* ($\mathbf{R}'(-')$)

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-auto*

lemma *RH-disj*: $RH(P \vee Q) = (RH(P) \vee RH(Q))$
by (*simp add*: *RH-def R3c-disj R2s-disj R1-disj*)

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

lemma *RH-UINF*:
assumes $A \neq \{\}$
shows $RH(\bigsqcup\ i \in A \cdot P(i)) = (\bigsqcup\ i \in A \cdot RH(P(i)))$
using *assms* **by** (*rel-auto*)

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*: $RH(P) = R1(R2c(R3c(P)))$
by (*rel-auto*)

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′*)

lemma *RH-subst-wait*: $RH(P\ _f) = RH(P)$
by (*metis R3c-subst-wait RH-alt-def′*)

end

15 Reactive designs

theory *utp-rea-designs*
imports *utp-reactive*
begin

15.1 Commutativity properties

lemma *H2-R1-comm*: $H2(R1(P)) = R1(H2(P))$
by (*rel-auto*)

lemma *H2-R2s-comm*: $H2(R2s(P)) = R2s(H2(P))$
by (*rel-auto*)

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-auto*

lemma *skip-rea-via-H1*: $II_r = R1(H1(R3(II)))$
by *rel-auto*

15.2 Reactive design composition

Pedro's proof for R1 design composition

lemma *R1-design-composition*:
fixes $P\ Q$ :: ($'t$::*ordered-cancel-monoid-diff*,$'\alpha$,$'\beta$) *relation-rp*
and $R\ S$ :: ($'t$,$'\beta$,$'\gamma$) *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 vwb-lens-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-blast*
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) $\wedge$ $\$ok\'$))
$\vee$ (($R1$($\neg$ $\$ok$) :: ($'t$,$'\alpha$,$'\beta$) *relation-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 (($R1$($\neg$ $\$ok$) :: ($'t$,$'\alpha$,$'\beta$) *relation-rp*) ;; $R1$(*true*)) =
($R1$($\neg$ $\$ok$) :: ($'t$,$'\alpha$,$'\gamma$) *relation-rp*)
by (*rel-auto*)
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-blast*)
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-blast*)
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-post*$(P) = (\lceil II \rceil_D \triangleleft \$wait \triangleright P)$

lemma *R3c-pre-conj*: *R3c-pre*$(P \wedge Q) = ($*R3c-pre*$(P) \wedge$ *R3c-pre*$(Q))$
by *rel-auto*

lemma *R3c-pre-seq*:
$(true \;;\; Q) = true \Longrightarrow$ *R3c-pre*$(P \;;\; Q) = ($*R3c-pre*$(P) \;;\; Q)$
by (*rel-auto*)

lemma *R2s-design*: *R2s*$(P \vdash Q) = ($*R2s*$(P) \vdash$ *R2s*$(Q))$
by (*simp add*: *R2s-def design-def usubst*)

lemma *R2c-design*: *R2c*$(P \vdash Q) = ($*R2c*$(P) \vdash$ *R2c*$(Q))$
by (*simp add*: *design-def impl-alt-def R2c-disj R2c-not R2c-ok R2c-and R2c-ok'*)

lemma *R1-R3c-design*:
R1(*R3c*$(P \vdash Q)$) = *R1*(*R3c-pre*$(P) \vdash$ *R3c-post*(Q))
by (*rel-auto*)

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-R2c* [*unrest*]: $\$ok \sharp P \Longrightarrow \$ok \sharp$ *R2c*(P)
by (*simp add*: *R2c-def unrest*)

lemma *unrest-ok'-R2c* [*unrest*]: $\$ok' \sharp P \Longrightarrow \$ok' \sharp$ *R2c*(P)
by (*simp add*: *R2c-def unrest*)

lemma *unrest-ok-R3c-pre* [*unrest*]: $\$ok \sharp P \Longrightarrow \$ok \sharp$ *R3c-pre*(P)
by (*simp add*: *R3c-pre-def cond-def unrest*)

lemma *unrest-ok'-R3c-pre* [*unrest*]: $\$ok' \sharp P \Longrightarrow \$ok' \sharp$ *R3c-pre*(P)
by (*simp add*: *R3c-pre-def cond-def unrest*)

lemma *unrest-ok-R3c-post* [*unrest*]: $\$ok \sharp P \Longrightarrow \$ok \sharp$ *R3c-post*(P)
by (*simp add*: *R3c-post-def cond-def unrest*)

lemma *unrest-ok-R3c-post'* [*unrest*]: $\$ok' \sharp P \Longrightarrow \$ok' \sharp$ *R3c-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-pre* $P)$;; *R1* *true*)) = (*R3c-pre* $(\neg$ (*R1* $(\neg P)$;; *R1* *true*)))
by (*rel-auto*)
have *2*:$(\neg$ (*R1* (*R3c-post* Q) ;; *R1* $(\neg$ *R3c-pre* R))) = *R3c-pre*$(\neg$ (*R1* $Q \wedge \neg \$wait'$;; *R1* $(\neg R)$))
by (*rel-auto*)
have *3*:(*R1* (*R3c-post* Q) ;; *R1* (*R3c-post* S)) = *R3c-post* (*R1* Q ;; $(\lceil II \rceil_D \triangleleft \$wait \triangleright$ *R1* S))
by (*rel-auto*)

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 *R1-des-lift-skip*: $R1(\lceil II\rceil_D) = \lceil II\rceil_D$
by (*rel-auto*)

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-auto*, *metis alpha-rp′.cases-scheme alpha-rp′.select-convs*(*2*) *alpha-rp′.update-convs*(*2*) *minus-zero-eq*)

lemma *R2c-R2s-absorb*: $R2c(R2s(P)) = R2s(P)$
by (*rel-auto*)

lemma *R2-design-composition*:
assumes $\$ok' \sharp P$ $\$ok' \sharp Q$ $\$ok \sharp R$ $\$ok \sharp S$
shows $(R2(P \vdash Q) ;; R2(R \vdash S)) =$
$R2((\neg (R1\ (\neg\ R2c\ P)\ ;;\ R1\ true) \wedge \neg (R1\ (R2c\ Q)\ ;;\ R1\ (\neg\ R2c\ R))) \vdash (R1\ (R2c\ Q)\ ;;\ R1\ (R2c\ S)))$
apply (*simp add*: *R2-R2c-def R2c-design R1-design-composition assms unrest R2c-not R2c-and R2c-disj R1-R2c-commute*[*THEN sym*] *R2c-idem R2c-R1-seq*)
apply (*metis* (*no-types*, *lifting*) *R2c-R1-seq R2c-not R2c-true*)
done

lemma *RH-design-composition*:
assumes $\$ok' \sharp P$ $\$ok' \sharp Q$ $\$ok \sharp R$ $\$ok \sharp S$
shows $(RH(P \vdash Q) ;; RH(R \vdash S)) =$
$RH((\neg (R1\ (\neg\ R2s\ P)\ ;;\ R1\ true) \wedge \neg (R1\ (R2s\ Q) \wedge \neg\ \$wait'\ ;;\ R1\ (\neg\ R2s\ R))) \vdash$

$(R1\ (R2s\ Q) \mathbin{;;} (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))))$

proof −
have *1*: $R2c\ (R1\ (\neg\ R2s\ P) \mathbin{;;} R1\ true) = (R1\ (\neg\ R2s\ P) \mathbin{;;} R1\ true)$
proof −
have *1*:$(R1\ (\neg\ R2s\ P) \mathbin{;;} R1\ true) = (R1(R2\ (\neg\ P) \mathbin{;;} R2\ true))$
by (*rel-auto*)
have $R2c(R1(R2\ (\neg\ P) \mathbin{;;} R2\ true)) = R2c(R1(R2\ (\neg\ P) \mathbin{;;} R2\ true))$
using *R2c-not* **by** *blast*
also have ... $= R2(R2\ (\neg\ P) \mathbin{;;} R2\ true)$
by (*metis R1-R2c-commute R1-R2c-is-R2*)
also have ... $= (R2\ (\neg\ P) \mathbin{;;} R2\ true)$
by (*simp add*: *R2-seqr-distribute*)
also have ... $= (R1\ (\neg\ R2s\ P) \mathbin{;;} R1\ true)$
by (*simp add*: *R2-def R2s-not R2s-true*)
finally show *?thesis*
by (*simp add*: *1*)
qed

have *2*:$R2c\ (R1\ (R2s\ Q) \wedge \neg\ \$wait' \mathbin{;;} R1\ (\neg\ R2s\ R)) = (R1\ (R2s\ Q) \wedge \neg\ \$wait' \mathbin{;;} R1\ (\neg\ R2s\ R))$
proof −
have $(R1\ (R2s\ Q) \wedge \neg\ \$wait' \mathbin{;;} R1\ (\neg\ R2s\ R)) = R1\ (R2\ (Q \wedge \neg\ \$wait') \mathbin{;;} R2\ (\neg\ R))$
by (*rel-auto*)
hence $R2c\ (R1\ (R2s\ Q) \wedge \neg\ \$wait' \mathbin{;;} R1\ (\neg\ R2s\ R)) = (R2\ (Q \wedge \neg\ \$wait') \mathbin{;;} R2\ (\neg\ R))$
by (*metis R1-R2c-commute R1-R2c-is-R2 R2-seqr-distribute*)
also have ... $= (R1\ (R2s\ Q) \wedge \neg\ \$wait' \mathbin{;;} R1\ (\neg\ R2s\ R))$
by *rel-auto*
finally show *?thesis* **.**
qed

have *3*:$R2c((R1\ (R2s\ Q) \mathbin{;;} (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S)))) = (R1\ (R2s\ Q) \mathbin{;;} (\lceil II \rceil_D \triangleleft \$wait \triangleright R1$
$(R2s\ S)))$
proof −
have $R2c(((R1\ (R2s\ Q))[\![true/\$wait']\!] \mathbin{;;} (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![true/\$wait]\!]))$
$= ((R1\ (R2s\ Q))[\![true/\$wait']\!] \mathbin{;;} (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![true/\$wait]\!])$
proof −
have $R2c(((R1\ (R2s\ Q))[\![true/\$wait']\!] \mathbin{;;} (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![true/\$wait]\!])) =$
$R2c(R1\ (R2s\ (Q[\![true/\$wait']\!])) \mathbin{;;} \lceil II \rceil_D[\![true/\$wait]\!])$
by (*simp add*: *usubst cond-unit-T R1-def R2s-def*)
also have ... $= R2c(R2(Q[\![true/\$wait']\!]) \mathbin{;;} R2(\lceil II \rceil_D[\![true/\$wait]\!]))$
by (*metis R2-def R2-des-lift-skip R2-subst-wait-true*)
also have ... $= (R2(Q[\![true/\$wait']\!]) \mathbin{;;} R2(\lceil II \rceil_D[\![true/\$wait]\!]))$
using *R2c-seq* **by** *blast*
also have ... $= ((R1\ (R2s\ Q))[\![true/\$wait']\!] \mathbin{;;} (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![true/\$wait]\!])$
apply (*simp add*: *usubst 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 $R2c(((R1\ (R2s\ Q))[\![false/\$wait']\!] \mathbin{;;} (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![false/\$wait]\!]))$
$= ((R1\ (R2s\ Q))[\![false/\$wait']\!] \mathbin{;;} (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))[\![false/\$wait]\!])$
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* (*no-types*, *hide-lams*) *R1-R2s-R2c R1-R3c-commute R2c-R3c-commute 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 ... $= R3c(R1(R2c((\neg\ (R1\ (\neg\ R2s\ P)$;; $R1\ true) \wedge \neg\ (R1\ (R2s\ Q) \wedge \neg\ \$wait'$;; $R1\ (\neg\ R2s\ R))) \vdash$
$(R1\ (R2s\ Q)$;; $(\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S))))))$
by (*simp add*: *R2c-design R2c-and R2c-not 1 2 3*)
finally show *?thesis*
by (*simp add*: *R1-R2s-R2c R1-R3c-commute R2c-R3c-commute RH-R2c-def*)
qed

lemma *RH-design-export-R1*: $RH(P \vdash Q) = RH(P \vdash R1(Q))$
by (*rel-auto*)

lemma *RH-design-export-R2s*: $RH(P \vdash Q) = RH(P \vdash R2s(Q))$
by (*rel-auto*)

lemma *RH-design-export-R2*: $RH(P \vdash Q) = RH(P \vdash R2(Q))$
by (*metis R2-def RH-design-export-R1 RH-design-export-R2s*)

lemma *RH-design-pre-neg-R1*: $RH((\neg\ R1\ P) \vdash Q) = RH((\neg\ P) \vdash Q)$
by (*metis* (*no-types*, *lifting*) *R1-R2c-commute R1-R3c-commute R1-def R1-disj RH-R2c-def design-def impl-alt-def not-conj-deMorgans utp-pred.double-compl utp-pred.inf.orderE utp-pred.inf-le2*)

lemma *RH-design-pre-R2s*: $RH((R2s\ P) \vdash Q) = RH(P \vdash Q)$
by (*metis* (*no-types*, *lifting*) *R1-R2c-is-R2 R1-R2s-R2c R2-R3c-commute R2s-design R2s-idem RH-alt-def'*)

lemma *RH-design-pre-R2c*: $RH((R2c\ P) \vdash Q) = RH(P \vdash Q)$
by (*metis* (*no-types*, *lifting*) *R2c-design R2c-idem RH-absorbs-R2c*)

lemma *RH-design-pre-neg-R1-R2c*: $RH((\neg\ R1\ (R2c\ P)) \vdash Q) = RH((\neg\ P) \vdash Q)$
by (*simp add*: *RH-design-pre-neg-R1*, *metis R2c-not RH-design-pre-R2c*)

lemma *RH-design-refine-intro*:
assumes ‘$P_1 \Rightarrow P_2$‘ ‘$P_1 \wedge Q_2 \Rightarrow Q_1$‘
shows $RH(P_1 \vdash Q_1) \sqsubseteq RH(P_2 \vdash Q_2)$
by (*simp add*: *RH-monotone assms*(*1*) *assms*(*2*) *design-refine-intro*)

Marcel’s proof for reactive design composition

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

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
$(RH(p_1 \vdash Q_1)$;; $RH(P_2 \vdash Q_2)) =$
$RH((p_1 \wedge \neg\ ((\$ok' \wedge \neg\ \$wait' \wedge Q_1)$;; $R1\ (\neg\ P_2)))$
$\vdash (((\$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg\ \$wait' \wedge Q_1)$;; $R1(Q_2)))))$ (**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* (*no-types, hide-lams*) *R1-R2-commute R1-idem R2-R3c-commute R2-def R2-seqr-distribute R3c-semir-form RH-alt-def′ calculation comp-apply*)
also have ... $= RH\ (R1\ ((\neg \$ok \vee R2s\ (\neg p_1)) \vee \$ok' \wedge R2s\ Q_1)\ ;;\ RH(P_2 \vdash Q_2))$
by (*simp add*: *design-def R2-R1-form impl-alt-def R2s-not R2s-ok R2s-disj R2s-conj R2s-ok′*)
also have ... $= RH(((\neg \$ok \wedge \$tr \leq_u \$tr')\ ;;\ RH(P_2 \vdash Q_2))$
$\vee ((\neg R2s(p_1) \wedge \$tr \leq_u \$tr')\ ;;\ RH(P_2 \vdash Q_2))$
$\vee ((\$ok' \wedge R2s(Q_1) \wedge \$tr \leq_u \$tr')\ ;;\ RH(P_2 \vdash Q_2)))$
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 ... $= RH(((\neg \$ok \wedge \$tr \leq_u \$tr')\ ;;\ RH(P_2 \vdash Q_2))$
$\vee ((\neg p_1 \wedge \$tr \leq_u \$tr')\ ;;\ RH(P_2 \vdash Q_2))$
$\vee ((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr')\ ;;\ RH(P_2 \vdash Q_2)))$
by (*metis Healthy-def′ assms*(*2*) *assms*(*4*))

also have ... $= RH((\neg \$ok \wedge \$tr \leq_u \$tr')$
$\vee (\neg p_1 \wedge \$tr \leq_u \$tr')$
$\vee ((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr')\ ;;\ RH(P_2 \vdash Q_2)))$
proof −
have $((\neg \$ok \wedge \$tr \leq_u \$tr')\ ;;\ RH(P_2 \vdash Q_2)) = (\neg \$ok \wedge \$tr \leq_u \$tr')$
by (*rel-auto*)
moreover have $(((\neg p_1\ ;;\ true) \wedge \$tr \leq_u \$tr')\ ;;\ RH(P_2 \vdash Q_2)) = ((\neg p_1\ ;;\ true) \wedge \$tr \leq_u \$tr')$
by (*rel-auto*)
ultimately show *?thesis*
by (*smt assms*(*1*) *precond-right-unit unrest-not*)
qed

also have ... $= RH((\neg \$ok \wedge \$tr \leq_u \$tr')$
$\vee (\neg p_1 \wedge \$tr \leq_u \$tr')$
$\vee ((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr')\ ;;\ (\$wait \wedge \$ok' \wedge II))$
$\vee ((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr')\ ;;\ (\neg \$wait \wedge R1(\neg P_2) \wedge \$tr \leq_u \$tr'))$
$\vee ((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr')\ ;;\ (\neg \$wait \wedge \$ok' \wedge R2(Q_2) \wedge \$tr \leq_u \$tr')))$
proof −
have *1*:$RH(P_2 \vdash Q_2) = ((\$wait \wedge \neg \$ok \wedge \$tr \leq_u \$tr')$
$\vee (\$wait \wedge \$ok' \wedge II)$
$\vee (\neg \$wait \wedge \neg \$ok \wedge \$tr \leq_u \$tr')$
$\vee (\neg \$wait \wedge R2(\neg P_2) \wedge \$tr \leq_u \$tr')$
$\vee (\neg \$wait \wedge \$ok' \wedge R2(Q_2) \wedge \$tr \leq_u \$tr'))$
by (*simp add*: *RH-alt-def′ R2-condr′ R2s-wait R2-skip-rea R3c-def usubst*, *rel-auto*)
have *2*:$((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr')\ ;;\ (\$wait \wedge \neg \$ok \wedge \$tr \leq_u \$tr')) = false$
by *rel-auto*
have *3*:$((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr')\ ;;\ (\neg \$wait \wedge \neg \$ok \wedge \$tr \leq_u \$tr')) = false$
by *rel-auto*
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-blast*)

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-auto*)
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-auto'*

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-auto'*

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-auto'*

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-auto*
thus *?thesis*
by (*simp add: design-def*)
qed

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

15.3 Healthiness conditions

definition [*upred-defs*]: $CSP1(P) = (P \vee (\neg \$ok \wedge \$tr \leq_u \$tr'))$

CSP2 is just H2 since the type system will automatically have J identifying the reactive variables as required.

definition [*upred-defs*]: $CSP2(P) = H2(P)$

abbreviation $CSP(P) \equiv CSP1(CSP2(RH(P)))$

lemma *CSP1-idem*:
$CSP1(CSP1(P)) = CSP1(P)$
by *pred-auto*

lemma *CSP2-idem*:
  $CSP2(CSP2(P)) = CSP2(P)$
  by (*simp add*: *CSP2-def H2-idem*)

lemma *CSP1-CSP2-commute*:
  $CSP1(CSP2(P)) = CSP2(CSP1(P))$
  by (*simp add*: *CSP1-def CSP2-def H2-split usubst*, *rel-auto*)

lemma *CSP1-R1-commute*:
  $CSP1(R1(P)) = R1(CSP1(P))$
  by (*rel-auto*)

lemma *CSP1-R2c-commute*:
  $CSP1(R2c(P)) = R2c(CSP1(P))$
  by (*rel-auto*)

lemma *CSP1-R3c-commute*:
  $CSP1(R3c(P)) = R3c(CSP1(P))$
  by (*rel-auto*)

lemma *CSP-idem*: $CSP(CSP(P)) = CSP(P)$
 by (*metis* (*no-types*, *hide-lams*) *CSP1-CSP2-commute CSP1-R1-commute CSP1-R2c-commute CSP1-R3c-commute CSP1-idem CSP2-def CSP2-idem R1-H2-commute R2c-H2-commute R3c-H2-commute RH-R2c-def RH-idem*)

lemma *CSP1-via-H1*: $R1(H1(P)) = R1(CSP1(P))$
  by *rel-auto*

lemma *CSP1-R3c*: $CSP1(R3(P)) = R3c(CSP1(P))$
  by *rel-auto*

lemma *CSP1-reactive-design*: $CSP1(RH(P \vdash Q)) = RH(P \vdash Q)$
  by *rel-auto*

lemma *CSP2-reactive-design*:
  assumes $\$ok' \sharp P$ $\$ok' \sharp Q$
  shows $CSP2(RH(P \vdash Q)) = RH(P \vdash Q)$
  using *assms*
  by (*simp add*: *CSP2-def H2-R1-comm H2-R2-comm H2-R3-comm H2-design RH-def H2-R2s-comm*)

lemma *CSP1-R1-H1*:
  $R1(H1(P)) = CSP1(R1(P))$
  by *rel-auto*

lemma *wait-false-design*:
  $(P \vdash Q)\ _f = ((P\ _f) \vdash (Q\ _f))$
  by (*rel-auto*)

lemma *CSP-RH-design-form*:
  $CSP(P) = RH((\neg P^f{}_f) \vdash P^t{}_f)$
proof −
  have $CSP(P) = CSP1(CSP2(R1(R2s(R3c(P)))))$
    by (*metis Healthy-def' RH-def assms*)
  also have ... $= CSP1(H2(R1(R2s(R3c(P)))))$
    by (*simp add*: *CSP2-def*)
  also have ... $= CSP1(R1(H2(R2s(R3c(P)))))$

by (*simp add*: *R1-H2-commute*)
also have ... = $R1(H1(R1(H2(R2s(R3c(P))))))$
by (*simp add*: *CSP1-R1-H1 R1-idem*)
also have ... = $R1(H1(H2(R2s(R3c(R1(P))))))$
by (*metis* (*no-types*, *hide-lams*) *CSP1-R1-H1 R1-H2-commute R1-R2-commute R1-idem R2-R3c-commute R2-def*)
also have ... = $R1(R2s(H1(H2(R3c(R1(P))))))$
by (*simp add*: *R2s-H1-commute R2s-H2-commute*)
also have ... = $R1(R2s(H1(R3c(H2(R1(P))))))$
by (*simp add*: *R3c-H2-commute*)
also have ... = $R2(R1(H1(R3c(H2(R1(P))))))$
by (*metis R1-R2-commute R1-idem R2-def*)
also have ... = $R2(R3c(R1(H1(H2(R1(P))))))$
by (*simp add*: *R1-H1-R3c-commute*)
also have ... = $RH(H1\text{-}H2(R1(P)))$
by (*metis R1-R2-commute R1-idem R2-R3c-commute R2-def RH-def*)
also have ... = $RH(H1\text{-}H2(P))$
by (*metis* (*no-types*, *hide-lams*) *CSP1-R1-H1 R1-H2-commute R1-R2-commute R1-R3c-commute R1-idem RH-alt-def*)
also have ... = $RH((\neg\ P^f) \vdash P^t)$
proof −
have *0*:$(\neg\ (H1\text{-}H2(P))^f) = (\$ok \wedge \neg\ P^f)$
by (*simp add*: *H1-def H2-split*, *pred-auto*)
have *1*:$(H1\text{-}H2(P))^t = (\$ok \Rightarrow (P^f \vee P^t))$
by (*simp add*: *H1-def H2-split*, *pred-auto*)
have $(\neg\ (H1\text{-}H2(P))^f) \vdash (H1\text{-}H2(P))^t = ((\neg\ P^f) \vdash P^t)$
by (*simp add*: *0 1*, *pred-auto*)
thus *?thesis*
by (*metis H1-H2-commute H1-H2-is-design H1-idem H2-idem Healthy-def'*)
qed
also have ... = $RH((\neg\ {P^f}_f) \vdash {P^t}_f)$
by (*metis* (*no-types*, *lifting*) *RH-subst-wait subst-not wait-false-design*)
finally show *?thesis* **.**
qed

lemma *CSP-reactive-design*:
assumes *P is CSP*
shows $RH((\neg\ {P^f}_f) \vdash {P^t}_f) = P$
by (*metis CSP-RH-design-form Healthy-def' assms*)

lemma *CSP-RH-design*:
assumes $\$ok' \sharp P$ $\$ok' \sharp Q$
shows $CSP(RH(P \vdash Q)) = RH(P \vdash Q)$
by (*metis CSP1-reactive-design CSP2-reactive-design RH-idem assms*(*1*) *assms*(*2*))

15.4 Reactive design triples

definition *wait'-cond* :: - ⇒ - ⇒ - (**infix** ⋄ *65*) **where**
[*upred-defs*]: $P \diamond Q = (P \triangleleft \$wait' \triangleright Q)$

lemma *wait'-cond-unrest* [*unrest*]:
$[\![$ *out-var wait* $\bowtie x$; $x \sharp P$; $x \sharp Q$ $]\!] \Longrightarrow x \sharp (P \diamond Q)$
by (*simp add*: *wait'-cond-def unrest*)

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*: $false \diamond P = (\neg\ \$wait\acute{}\ \wedge\ P)$
by (*rel-auto*)

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-blast*)

lemma *wait′-cond-true*: $(P \diamond Q \wedge \$wait\acute{}) = (P \wedge \$wait\acute{})$
by (*rel-auto*)

lemma *wait′-cond-false*: $(P \diamond Q \wedge (\neg\$wait\acute{})) = (Q \wedge (\neg\$wait\acute{}))$
by (*rel-auto*)

lemma *wait′-cond-idem*: $P \diamond P = P$
by (*rel-auto*)

lemma *wait′-cond-conj-exchange*:
$((P \diamond Q) \wedge (R \diamond S)) = (P \wedge R) \diamond (Q \wedge S)$
by *rel-auto*

lemma *subst-wait′-cond-true* [*usubst*]: $(P \diamond Q)[\![true/\$wait\acute{}]\!] = P[\![true/\$wait\acute{}]\!]$
by *rel-auto*

lemma *subst-wait′-cond-false* [*usubst*]: $(P \diamond Q)[\![false/\$wait\acute{}]\!] = Q[\![false/\$wait\acute{}]\!]$
by *rel-auto*

lemma *subst-wait′-left-subst*: $(P[\![true/\$wait\acute{}]\!] \diamond Q) = (P \diamond Q)$
by (*metis wait′-cond-def cond-def conj-comm conj-eq-out-var-subst upred-eq-true wait-vwb-lens*)

lemma *subst-wait′-right-subst*: $(P \diamond Q[\![false/\$wait\acute{}]\!]) = (P \diamond Q)$
by (*metis cond-def conj-eq-out-var-subst upred-eq-false utp-pred.inf.commute wait′-cond-def wait-vwb-lens*)

lemma *wait′-cond-split*: $P[\![true/\$wait\acute{}]\!] \diamond P[\![false/\$wait\acute{}]\!] = P$
by (*simp add*: *wait′-cond-def cond-var-split*)

lemma *R1-wait′-cond*: $R1(P \diamond Q) = R1(P) \diamond R1(Q)$
by *rel-auto*

lemma *R2s-wait′-cond*: $R2s(P \diamond Q) = R2s(P) \diamond R2s(Q)$
by (*simp add*: *wait′-cond-def R2s-def R2s-def usubst*)

lemma *R2-wait′-cond*: $R2(P \diamond Q) = R2(P) \diamond R2(Q)$
by (*simp add*: *R2-def R2s-wait′-cond R1-wait′-cond*)

lemma *RH-design-peri-R1*: $RH(P \vdash R1(Q) \diamond R) = RH(P \vdash Q \diamond R)$
by (*metis* (*no-types*, *lifting*) *R1-idem R1-wait′-cond RH-design-export-R1*)

lemma *RH-design-post-R1*: $RH(P \vdash Q \diamond R1(R)) = RH(P \vdash Q \diamond R)$
by (*metis R1-wait′-cond RH-design-export-R1 RH-design-peri-R1*)

lemma *RH-design-peri-R2s*: $RH(P \vdash R2s(Q) \diamond R) = RH(P \vdash Q \diamond R)$
by (*metis* (*no-types*, *lifting*) *R2s-idem R2s-wait′-cond RH-design-export-R2s*)

lemma *RH-design-post-R2s*: $RH(P \vdash Q \diamond R2s(R)) = RH(P \vdash Q \diamond R)$

by (*metis* (*no-types*, *lifting*) *R2s-idem R2s-wait′-cond RH-design-export-R2s*)

lemma *RH-design-peri-R2c*: $RH(P \vdash R2c(Q) \diamond R) = RH(P \vdash Q \diamond R)$
by (*metis* (*no-types*, *lifting*) *R1-R2c-is-R2 R2-wait′-cond R2c-idem RH-design-export-R2*)

lemma *RH-design-post-R2c*: $RH(P \vdash Q \diamond R2c(R)) = RH(P \vdash Q \diamond R)$
by (*metis* (*no-types*, *lifting*) *R1-R2c-is-R2 R2-wait′-cond R2c-idem RH-design-export-R2*)

lemma *RH-design-lemma1*:
$RH(P \vdash (R1(R2c(Q)) \lor R) \diamond S) = RH(P \vdash (Q \lor R) \diamond S)$
by (*simp add*: *design-def impl-alt-def wait′-cond-def RH-R2c-def R2c-R3c-commute R1-R3c-commute R1-disj R2c-disj R2c-and R1-cond R2c-condr R1-R2c-commute R2c-idem R1-extend-conj′ R1-idem*)

lemma *RH-tri-design-composition*:
assumes $\$ok\acute{} \sharp P$ $\$ok\acute{} \sharp Q_1$ $\$ok\acute{} \sharp Q_2$ $\$ok \sharp R$ $\$ok \sharp S_1$ $\$ok \sharp S_2$
$\$wait\acute{} \sharp Q_2$ $\$wait \sharp S_1$ $\$wait \sharp S_2$
shows $(RH(P \vdash Q_1 \diamond Q_2) ;; RH(R \vdash S_1 \diamond S_2)) =$
$RH((\neg (R1\ (\neg\ R2s\ P) ;; R1\ true) \land \neg (R1\ (R2s\ Q_2) \land \neg\ \$wait\acute{} ;; R1\ (\neg\ R2s\ R))) \vdash$
$((Q_1 \lor (R1\ (R2s\ Q_2) ;; R1\ (R2s\ S_1))) \diamond ((R1\ (R2s\ Q_2) ;; R1\ (R2s\ S_2)))))$
proof −
have *1*:$(\neg (R1\ (R2s\ (Q_1 \diamond Q_2)) \land \neg\ \$wait\acute{} ;; R1\ (\neg\ R2s\ R))) =$
$(\neg (R1\ (R2s\ Q_2) \land \neg\ \$wait\acute{} ;; R1\ (\neg\ R2s\ R)))$
by (*metis* (*no-types*, *hide-lams*) *R1-extend-conj R2s-conj R2s-not R2s-wait′ wait′-cond-false*)
have *2*: $(R1\ (R2s\ (Q_1 \diamond Q_2)) ;; (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ (S_1 \diamond S_2)))) =$
$((R1\ (R2s\ Q_1) \lor (R1\ (R2s\ Q_2) ;; R1\ (R2s\ S_1))) \diamond (R1\ (R2s\ Q_2) ;; R1\ (R2s\ S_2)))$
proof −
have $(R1\ (R2s\ Q_1) ;; \$wait \land (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S_1) \diamond R1\ (R2s\ S_2)))$
$= (R1\ (R2s\ Q_1) \land \$wait\acute{})$
proof −
have $(R1\ (R2s\ Q_1) ;; \$wait \land (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S_1) \diamond R1\ (R2s\ S_2)))$
$= (R1\ (R2s\ Q_1) ;; \$wait \land \lceil II \rceil_D)$
by (*rel-auto*)
also have ... $= ((R1\ (R2s\ Q_1) ;; \lceil II \rceil_D) \land \$wait\acute{})$
by (*rel-auto*)
also from *assms*(*2*) **have** ... $= ((R1\ (R2s\ Q_1)) \land \$wait\acute{})$
by (*simp add*: *lift-des-skip-dr-unit-unrest unrest*)
finally show *?thesis* **.**
qed

moreover have $(R1\ (R2s\ Q_2) ;; \neg\ \$wait \land (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S_1) \diamond R1\ (R2s\ S_2)))$
$= ((R1\ (R2s\ Q_2)) ;; (R1\ (R2s\ S_1) \diamond R1\ (R2s\ S_2)))$
proof −
have $(R1\ (R2s\ Q_2) ;; \neg\ \$wait \land (\lceil II \rceil_D \triangleleft \$wait \triangleright R1\ (R2s\ S_1) \diamond R1\ (R2s\ S_2)))$
$= (R1\ (R2s\ Q_2) ;; \neg\ \$wait \land (R1\ (R2s\ S_1) \diamond R1\ (R2s\ S_2)))$
by (*metis* (*no-types*, *lifting*) *cond-def conj-disj-not-abs utp-pred.double-compl utp-pred.inf.left-idem utp-pred.sup-assoc utp-pred.sup-inf-absorb*)

also have ... $= ((R1\ (R2s\ Q_2))[\![false/\$wait\acute{}]\!] ;; (R1\ (R2s\ S_1) \diamond R1\ (R2s\ S_2))[\![false/\$wait]\!])$
by (*metis false-alt-def seqr-right-one-point upred-eq-false wait-vwb-lens*)

also have ... $= ((R1\ (R2s\ Q_2)) ;; (R1\ (R2s\ S_1) \diamond R1\ (R2s\ S_2)))$
by (*simp add*: *wait′-cond-def usubst unrest assms*)

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

moreover
have $((R1\ (R2s\ Q_1) \wedge \$wait') \vee ((R1\ (R2s\ Q_2))\ ;;\ (R1\ (R2s\ S_1) \diamond R1\ (R2s\ S_2))))$
$= (R1\ (R2s\ Q_1) \vee (R1\ (R2s\ Q_2)\ ;;\ R1\ (R2s\ S_1))) \diamond ((R1\ (R2s\ Q_2)\ ;;\ R1\ (R2s\ S_2)))$
by (*simp add*: *wait′-cond-def cond-seq-right-distr cond-and-T-integrate unrest*)

ultimately show *?thesis*
by (*simp add*: *R2s-wait′-cond R1-wait′-cond wait′-cond-seq*)
qed

show *?thesis*
apply (*subst RH-design-composition*)
apply (*simp-all add*: *assms*)
apply (*simp add*: *assms wait′-cond-def unrest*)
apply (*simp add*: *assms wait′-cond-def unrest*)
apply (*simp add*: *1 2*)
apply (*simp add*: *R1-R2s-R2c RH-design-lemma1*)
done
qed

Syntax for pre-, post-, and periconditions

abbreviation $pre_s \equiv [\$ok \mapsto_s true, \$ok' \mapsto_s false, \$wait \mapsto_s false]$
abbreviation $peri_s \equiv [\$ok \mapsto_s true, \$ok' \mapsto_s true, \$wait \mapsto_s false, \$wait' \mapsto_s true]$
abbreviation $post_s \equiv [\$ok \mapsto_s true, \$ok' \mapsto_s true, \$wait \mapsto_s false, \$wait' \mapsto_s false]$

abbreviation $npre_R(P) \equiv pre_s \dagger P$

definition [*upred-defs*]: $pre_R(P) = (\neg (npre_R(P)))$
definition [*upred-defs*]: $peri_R(P) = (peri_s \dagger P)$
definition [*upred-defs*]: $post_R(P) = (post_s \dagger P)$

lemma *ok-pre-unrest* [*unrest*]: $\$ok \sharp pre_R\ P$
by (*simp add*: pre_R*-def unrest usubst*)

lemma *ok-peri-unrest* [*unrest*]: $\$ok \sharp peri_R\ P$
by (*simp add*: $peri_R$*-def unrest usubst*)

lemma *ok-post-unrest* [*unrest*]: $\$ok \sharp post_R\ P$
by (*simp add*: $post_R$*-def unrest usubst*)

lemma *ok′-pre-unrest* [*unrest*]: $\$ok' \sharp pre_R\ P$
by (*simp add*: pre_R*-def unrest usubst*)

lemma *ok′-peri-unrest* [*unrest*]: $\$ok' \sharp peri_R\ P$
by (*simp add*: $peri_R$*-def unrest usubst*)

lemma *ok′-post-unrest* [*unrest*]: $\$ok' \sharp post_R\ P$
by (*simp add*: $post_R$*-def unrest usubst*)

lemma *wait-pre-unrest* [*unrest*]: $\$wait \sharp pre_R\ P$
by (*simp add*: pre_R*-def unrest usubst*)

lemma *wait-peri-unrest* [*unrest*]: $\$wait \sharp peri_R\ P$
by (*simp add*: $peri_R$*-def unrest usubst*)

lemma *wait-post-unrest* [*unrest*]: $\$wait \sharp post_R\ P$
by (*simp add*: $post_R$*-def unrest usubst*)

lemma *wait'-peri-unrest* [*unrest*]: $\$wait' \sharp peri_R\ P$
by (*simp add*: $peri_R$*-def unrest usubst*)

lemma *wait'-post-unrest* [*unrest*]: $\$wait' \sharp post_R\ P$
by (*simp add*: $post_R$*-def unrest usubst*)

lemma pre_s*-design*: $pre_s \dagger (P \vdash Q) = (\neg\ pre_s \dagger P)$
by (*simp add*: *design-def* pre_R*-def usubst*)

lemma $peri_s$*-design*: $peri_s \dagger (P \vdash Q \diamond R) = peri_s \dagger (P \Rightarrow Q)$
by (*simp add*: *design-def usubst wait'-cond-def*)

lemma $post_s$*-design*: $post_s \dagger (P \vdash Q \diamond R) = post_s \dagger (P \Rightarrow R)$
by (*simp add*: *design-def usubst wait'-cond-def*)

lemma pre_s*-R1* [*usubst*]: $pre_s \dagger R1(P) = R1(pre_s \dagger P)$
by (*simp add*: *R1-def usubst*)

lemma pre_s*-R2c* [*usubst*]: $pre_s \dagger R2c(P) = R2c(pre_s \dagger P)$
by (*simp add*: *R2c-def R2s-def usubst*)

lemma $peri_s$*-R1* [*usubst*]: $peri_s \dagger R1(P) = R1(peri_s \dagger P)$
by (*simp add*: *R1-def usubst*)

lemma $peri_s$*-R2c* [*usubst*]: $peri_s \dagger R2c(P) = R2c(peri_s \dagger P)$
by (*simp add*: *R2c-def R2s-def usubst*)

lemma $post_s$*-R1* [*usubst*]: $post_s \dagger R1(P) = R1(post_s \dagger P)$
by (*simp add*: *R1-def usubst*)

lemma $post_s$*-R2c* [*usubst*]: $post_s \dagger R2c(P) = R2c(post_s \dagger P)$
by (*simp add*: *R2c-def R2s-def usubst*)

lemma *rea-pre-RH-design*: $pre_R(RH(P \vdash Q)) = (\neg\ R1(R2c(pre_s \dagger (\neg\ P))))$
by (*simp add*: *RH-R2c-def usubst R3c-def* pre_R*-def* pre_s*-design*)

lemma *rea-peri-RH-design*: $peri_R(RH(P \vdash Q \diamond R)) = R1(R2c(peri_s \dagger (P \Rightarrow Q)))$
by (*simp add*:*RH-R2c-def usubst* $peri_R$*-def R3c-def* $peri_s$*-design*)

lemma *rea-post-RH-design*: $post_R(RH(P \vdash Q \diamond R)) = R1(R2c(post_s \dagger (P \Rightarrow R)))$
by (*simp add*:*RH-R2c-def usubst* $post_R$*-def R3c-def* $post_s$*-design*)

lemma *CSP-reactive-tri-design-lemma*:
assumes *P is CSP*
shows $RH((\neg\ P^f{}_f) \vdash P^t{}_f[\![true/\$wait']\!] \diamond P^t{}_f[\![false/\$wait']\!]) = P$
by (*simp add*: *CSP-reactive-design assms wait'-cond-split*)

lemma *CSP-reactive-tri-design*:
assumes *P is CSP*
shows $RH(pre_R(P) \vdash peri_R(P) \diamond post_R(P)) = P$
proof −
have $P = RH((\neg\ P^f{}_f) \vdash P^t{}_f[\![true/\$wait']\!] \diamond P^t{}_f[\![false/\$wait']\!])$

by (*simp add*: *CSP-reactive-tri-design-lemma assms*)
also have ... = $RH(pre_R(P) \vdash peri_R(P) \diamond post_R(P))$
apply (*simp add*: *usubst*)
apply (*subst design-subst-ok-ok′*[*THEN sym*])
apply (*simp add*: pre_R*-def* $peri_R$*-def* $post_R$*-def usubst unrest*)
done
finally show *?thesis* **..**
qed

lemma *skip-rea-reactive-design*:
$II_r = RH(true \vdash II)$
proof −
have $RH(true \vdash II) = R1(R2c(R3c(true \vdash II)))$
by (*metis RH-R2c-def*)
also have ... = $R1(R3c(R2c(true \vdash II)))$
by (*metis R2c-R3c-commute RH-R2c-def*)
also have ... = $R1(R3c(true \vdash II))$
by (*simp add*: *design-def impl-alt-def R2c-disj R2c-not R2c-ok R2c-and R2c-skip-r R2c-ok′*)
also have ... = $R1(II_r \triangleleft \$wait \triangleright true \vdash II)$
by (*metis R3c-def*)
also have ... = II_r
by (*rel-auto*)
finally show *?thesis* **..**
qed

lemma *skip-rea-reactive-design′*:
$II_r = RH(true \vdash \lceil II \rceil_D)$
by (*metis aext-true rdesign-def skip-d-alt-def skip-d-def skip-rea-reactive-design*)

lemma *RH-design-subst-wait*: $RH(P\ _f \vdash Q\ _f) = RH(P \vdash Q)$
by (*metis RH-subst-wait wait-false-design*)

lemma *RH-design-subst-wait-pre*: $RH(P\ _f \vdash Q) = RH(P \vdash Q)$
by (*subst RH-design-subst-wait*[*THEN sym*], *simp add*: *usubst RH-design-subst-wait*)

lemma *RH-design-subst-wait-post*: $RH(P \vdash Q\ _f) = RH(P \vdash Q)$
by (*subst RH-design-subst-wait*[*THEN sym*], *simp add*: *usubst RH-design-subst-wait*)

lemma *RH-peri-subst-false-wait*: $RH(P \vdash Q\ _f \diamond R) = RH(P \vdash Q \diamond R)$
apply (*subst RH-design-subst-wait-post*[*THEN sym*])
apply (*simp add*: *usubst unrest*)
apply (*metis RH-design-subst-wait RH-design-subst-wait-pre out-in-indep out-var-uvar unrest-false unrest-usubst-id unrest-usubst-upd vwb-lens.axioms*(*2*) *wait′-cond-subst wait-vwb-lens*)
done

lemma *RH-post-subst-false-wait*: $RH(P \vdash Q \diamond R\ _f) = RH(P \vdash Q \diamond R)$
apply (*subst RH-design-subst-wait-post*[*THEN sym*])
apply (*simp add*: *usubst unrest*)
apply (*metis RH-design-subst-wait RH-design-subst-wait-pre out-in-indep out-var-uvar unrest-false unrest-usubst-id unrest-usubst-upd vwb-lens.axioms*(*2*) *wait′-cond-subst wait-vwb-lens*)
done

lemma *skip-rea-reactive-tri-design*:
$II_r = RH(true \vdash false \diamond \lceil II \rceil_D)$ (**is** *?lhs* = *?rhs*)
proof −

have *?rhs* = *RH* (*true* ⊢ (¬ $wait´ ∧ ⌈*II*⌉$_D$))
  by (*simp add*: *wait′-cond-def cond-def*)
have ... = *RH* (*true* ⊢ (¬ $wait ∧ ⌈*II*⌉$_D$)) (**is** *RH* (*true* ⊢ *?Q1*) = *RH* (*true* ⊢ *?Q2*))
proof −
  have *?Q1* = *?Q2*
    by (*rel-auto*)
  thus *?thesis* **by** *simp*
qed
also have ... = *RH* (*true* ⊢ ⌈*II*⌉$_D$)
  by (*rel-auto*)
finally show *?thesis*
  by (*simp add*: *skip-rea-reactive-design′ wait′-cond-def cond-def*)
qed

lemma *skip-d-lift-rea*:
⌈*II*⌉$_D$ = ($wait´ $=_u$ $wait ∧ $tr´ $=_u$ $tr ∧ $Σ$_R$´ $=_u$ $Σ$_R$)
by (*rel-auto*)

lemma *skip-rea-reactive-tri-design′*:
II$_r$ = *RH*(*true* ⊢ *false* ⋄ ($tr´ $=_u$ $tr ∧ $Σ$_R$´ $=_u$ $Σ$_R$)) (**is** *?lhs* = *?rhs*)
proof −
have *?rhs* = *RH* (*true* ⊢ (¬ $wait´ ∧ $tr´ $=_u$ $tr ∧ $Σ$_R$´ $=_u$ $Σ$_R$))
  by (*simp add*: *wait′-cond-def cond-def*)
also have ... = *RH* (*true* ⊢ ($wait´ $=_u$ $wait ∧ $tr´ $=_u$ $tr ∧ $Σ$_R$´ $=_u$ $Σ$_R$)) (**is** *RH* (*true* ⊢ *?Q1*) = *RH* (*true* ⊢ *?Q2*))
proof −
  have *?Q1* $_f$ = *?Q2* $_f$
    by (*rel-auto*)
  thus *?thesis*
    by (*metis RH-design-subst-wait*)
qed
also have ... = *RH* (*true* ⊢ ⌈*II*⌉$_D$)
  by (*metis skip-d-lift-rea*)
finally show *?thesis*
  by (*simp add*: *skip-rea-reactive-design′*)
qed

lemma *R1-neg-pre*: *R1* (¬ *pre*$_R$ *P*) = (¬ *pre*$_R$ (*R1*(*P*)))
by (*simp add*: *pre*$_R$*-def R1-def usubst*)

lemma *R1-peri*: *R1* (*peri*$_R$ *P*) = *peri*$_R$ (*R1*(*P*))
by (*simp add*: *peri*$_R$*-def R1-def usubst*)

lemma *R1-post*: *R1* (*post*$_R$ *P*) = *post*$_R$ (*R1*(*P*))
by (*simp add*: *post*$_R$*-def R1-def usubst*)

lemma *R2s-pre*:
R2s (*pre*$_R$ *P*) = *pre*$_R$ (*R2s P*)
by (*simp add*: *pre*$_R$*-def R2s-def usubst*)

lemma *R2s-peri*: *R2s* (*peri*$_R$ *P*) = *peri*$_R$ (*R2s P*)
by (*simp add*: *peri*$_R$*-def R2s-def usubst*)

lemma *R2s-post*: *R2s* (*post*$_R$ *P*) = *post*$_R$ (*R2s P*)
by (*simp add*: *post*$_R$*-def R2s-def usubst*)

lemma *RH-pre-RH-design*:
  $\$ok' \sharp P \Longrightarrow RH(pre_R(RH(P \vdash Q)) \vdash R) = RH(P \vdash R)$
  apply (*simp add*: *rea-pre-RH-design RH-design-pre-neg-R1-R2c usubst*)
  apply (*subst subst-to-singleton*)
  apply (*simp add*: *unrest*)
  apply (*simp add*: *RH-design-subst-wait-pre*)
  apply (*simp add*: *usubst*)
  apply (*metis conj-pos-var-subst design-def vwb-lens-ok*)
done

lemma *RH-postcondition*: $(RH(P \vdash Q))^t{}_f = R1(R2s(\$ok \wedge P^t{}_f \Rightarrow Q^t{}_f))$
  by (*simp add*: *RH-def R1-def R3c-def usubst R2s-def design-def*)

lemma *RH-postcondition-RH*: $RH(P \vdash (RH(P \vdash Q))^t{}_f) = RH(P \vdash Q)$
proof −
  have $RH(P \vdash (RH(P \vdash Q))^t{}_f) = RH\ (P \vdash (\$ok \wedge P^t{}_f \Rightarrow Q^t{}_f))$
    by (*simp add*: *RH-postcondition RH-design-export-R1*[*THEN sym*] *RH-design-export-R2s*[*THEN sym*])
  also have ... $= RH\ (P \vdash (\$ok \wedge P^t \Rightarrow Q^t))$
    by (*subst RH-design-subst-wait-post*[*THEN sym*, *of* - $(\$ok \wedge P^t \Rightarrow Q^t)$], *simp add*: *usubst*)
  also have ... $= RH\ (P \vdash (P^t \Rightarrow Q^t))$
    by (*rel-auto*)
  also have ... $= RH\ (P \vdash (P \Rightarrow Q))$
    by (*subst design-subst-ok'*[*THEN sym*, *of* - $P \Rightarrow Q$], *simp add*: *usubst*)
  also have ... $= RH\ (P \vdash Q)$
    by (*rel-auto*)
  finally show *?thesis* **.**
qed

lemma *peri$_R$-alt-def*: $peri_R(P) = (P^t{}_f)[\![true/\$ok]\!][\![true/\$wait']\!]$
  by (*simp add*: *peri$_R$-def usubst*)

lemma *post$_R$-alt-def*: $post_R(P) = (P^t{}_f)[\![true/\$ok]\!][\![false/\$wait']\!]$
  by (*simp add*: *post$_R$-def usubst*)

lemma *design-export-ok-true*: $P \vdash Q[\![true/\$ok]\!] = P \vdash Q$
  by (*metis conj-pos-var-subst design-export-ok vwb-lens-ok*)

lemma *design-export-peri-ok-true*: $P \vdash Q[\![true/\$ok]\!] \diamond R = P \vdash Q \diamond R$
  apply (*subst design-export-ok-true*[*THEN sym*])
  apply (*simp add*: *usubst unrest*)
  apply (*metis design-export-ok-true out-in-indep out-var-uvar unrest-true unrest-usubst-id unrest-usubst-upd vwb-lens-mwb wait'-cond-subst wait-vwb-lens*)
done

lemma *design-export-post-ok-true*: $P \vdash Q \diamond R[\![true/\$ok]\!] = P \vdash Q \diamond R$
  apply (*subst design-export-ok-true*[*THEN sym*])
  apply (*simp add*: *usubst unrest*)
  apply (*metis design-export-ok-true out-in-indep out-var-uvar unrest-true unrest-usubst-id unrest-usubst-upd vwb-lens-mwb wait'-cond-subst wait-vwb-lens*)
done

lemma *RH-peri-RH-design*:
  $RH(P \vdash peri_R(RH(P \vdash Q \diamond R)) \diamond S) = RH(P \vdash Q \diamond S)$

apply (*simp add*: *peri$_R$-alt-def subst-wait′-left-subst design-export-peri-ok-true RH-postcondition*)
apply (*simp add*: *rea-peri-RH-design RH-design-peri-R1 RH-design-peri-R2s*)
oops

lemma *CSP-R1-R2s*: P *is* $CSP \Longrightarrow R1\ (R2s\ P) = P$
by (*metis* (*no-types*) *CSP-reactive-design R1-R2c-is-R2 R1-R2s-R2c R2-idem RH-alt-def′*)

lemma *R1-R2s-tr-diff-conj*: $(R1\ (R2s\ (\$tr\' =_u \$tr \wedge P))) = (\$tr\' =_u \$tr \wedge R2s(P))$
apply (*rel-auto*) **using** *minus-zero-eq* **by** *blast*

lemma *R2s-state′-eq-state*: $R2s\ (\$\Sigma_R\' =_u \$\Sigma_R) = (\$\Sigma_R\' =_u \$\Sigma_R)$
by (*simp add*: *R2s-def usubst*)

lemma *skip-r-rea*: $II = (\$ok\' =_u \$ok \wedge \$wait\' =_u \$wait \wedge \$tr\' =_u \$tr \wedge \$\Sigma_R\' =_u \$\Sigma_R)$
by (*rel-auto*)

lemma *wait-pre-lemma*:
assumes $\$wait\' \sharp P$
shows $(P \wedge \neg \$wait\' ;; \neg pre_R\ Q) = (P ;; \neg pre_R\ Q)$
proof −
have $(P \wedge \neg \$wait\' ;; \neg pre_R\ Q) = (P \wedge \$wait\' =_u false ;; \neg pre_R\ Q)$
by (*rel-auto*)
also have ... $= (P[\![false/\$wait\']\!] ;; (\neg pre_R\ Q)[\![false/\$wait]\!])$
by (*metis false-alt-def seqr-left-one-point wait-vwb-lens*)
also have ... $= (P ;; \neg pre_R\ Q)$
by (*simp add*: *usubst unrest assms*)
finally show *?thesis* **.**
qed

lemma *rea-left-unit-lemma*:
assumes $\$ok \sharp P$ $\$wait \sharp P$
shows $((\$tr\' =_u \$tr \wedge \$\Sigma_R\' =_u \$\Sigma_R) ;; P) = P$
proof −
have $P = (II ;; P)$
by *simp*
also have ... $= ((\$ok\' =_u \$ok \wedge \$wait\' =_u \$wait \wedge \$tr\' =_u \$tr \wedge \$\Sigma_R\' =_u \$\Sigma_R) ;; P)$
by (*metis skip-r-rea*)
also from *assms* **have** ... $= ((\$tr\' =_u \$tr \wedge \$\Sigma_R\' =_u \$\Sigma_R) ;; P)$
by (*simp add*: *seqr-insert-ident-left assms unrest*)
finally show *?thesis* **..**
qed

lemma *rea-right-unit-lemma*:
assumes $\$ok\' \sharp P$ $\$wait\' \sharp P$
shows $(P ;; (\$tr\' =_u \$tr \wedge \$\Sigma_R\' =_u \$\Sigma_R)) = P$
proof −
have $P = (P ;; II)$
by *simp*
also have ... $= (P ;; (\$ok\' =_u \$ok \wedge \$wait\' =_u \$wait \wedge \$tr\' =_u \$tr \wedge \$\Sigma_R\' =_u \$\Sigma_R))$
by (*metis skip-r-rea*)
also from *assms* **have** ... $= (P ;; (\$tr\' =_u \$tr \wedge \$\Sigma_R\' =_u \$\Sigma_R))$
by (*simp add*: *seqr-insert-ident-right assms unrest*)
finally show *?thesis* **..**
qed

lemma *skip-rea-left-unit*:
assumes P *is CSP*
shows $(II_r ;; P) = P$
proof −
have $(II_r ;; P) = (II_r ;; RH\ (pre_R\ P \vdash peri_R\ P \diamond post_R\ P))$
by (*metis CSP-reactive-tri-design assms*)
also have ... $= (RH(true \vdash false \diamond (\$tr' =_u \$tr \wedge \$\Sigma_R' =_u \$\Sigma_R)) ;; RH\ (pre_R\ P \vdash peri_R\ P \diamond post_R\ P))$
by (*metis skip-rea-reactive-tri-design'*)
also have ... $= RH\ (pre_R\ P \vdash peri_R\ P \diamond post_R\ P)$
apply (*subst RH-tri-design-composition*)
apply (*simp-all add*: *unrest R2s-true R1-false R1-neg-pre R1-peri R1-post R2s-pre R2s-peri R2s-post CSP-R1-R2s R1-R2s-tr-diff-conj assms*)
apply (*simp add*: *R2s-conj R2s-state'-eq-state wait-pre-lemma rea-left-unit-lemma unrest*)
done
also have ... $= P$
by (*metis CSP-reactive-tri-design assms*)
finally show *?thesis* **.**
qed

lemma *skip-rea-left-semi-unit*:
assumes P *is CSP* $out\alpha \sharp pre_R\ P$
shows $(P ;; II_r) = RH\ ((\neg (\neg pre_R\ P ;; R1\ true)) \vdash peri_R\ P \diamond post_R\ P)$
proof −
have $(P ;; II_r) = (RH\ (pre_R\ P \vdash peri_R\ P \diamond post_R\ P) ;; II_r)$
by (*metis CSP-reactive-tri-design assms*)
also have ... $= (RH\ (pre_R\ P \vdash peri_R\ P \diamond post_R\ P) ;; RH(true \vdash false \diamond (\$tr' =_u \$tr \wedge \$\Sigma_R' =_u \$\Sigma_R)))$
by (*metis skip-rea-reactive-tri-design'*)
also have ... $= RH\ ((\neg (\neg pre_R\ P ;; R1\ true)) \vdash peri_R\ P \diamond post_R\ P)$
apply (*subst RH-tri-design-composition*)
apply (*simp-all add*: *unrest R2s-true R1-false R2s-false R1-neg-pre R1-peri R1-post R2s-pre R2s-peri R2s-post CSP-R1-R2s R1-R2s-tr-diff-conj assms*)
apply (*simp add*: *R2s-conj R2s-state'-eq-state wait-pre-lemma rea-right-unit-lemma unrest*)
done
finally show *?thesis* **.**
qed

lemma *HR-design-wait-false*: $RH(P_f \vdash Q_f) = RH(P \vdash Q)$
by (*metis R3c-subst-wait RH-R2c-def wait-false-design*)

lemma *RH-design-R1-neg-precond*: $RH((\neg R1(\neg P)) \vdash Q) = RH(P \vdash Q)$
by (*rel-auto*)

lemma *RH-design-pre-neg-conj-R1*: $RH((\neg R1\ P \wedge \neg R1\ Q) \vdash R) = RH((\neg P \wedge \neg Q) \vdash R)$
by (*rel-auto*)

15.5 Signature

definition [*urel-defs*]: $Miracle = RH(true \vdash false \diamond false)$

definition [*urel-defs*]: $Chaos = RH(false \vdash true \diamond true)$

definition [*urel-defs*]: $Term = RH(true \vdash true \diamond true)$

definition *assigns-rea* :: $'\alpha$ *usubst* $\Rightarrow$ (*'t::ordered-cancel-monoid-diff*, $'\alpha$) *hrelation-rp* $(\langle \text{-} \rangle_R)$ **where**

assigns-rea σ = $RH(true \vdash false \diamond (\$tr' =_u \$tr \wedge \lceil \langle \sigma \rangle_a \rceil_R))$

definition *reactive-sup* :: - *set* ⇒ - ($\bigsqcap_R$) **where**
$\bigsqcap_R A = (if\ (A = \{\})\ then\ Miracle\ else\ \bigsqcap\ A)$

definition *reactive-inf* :: - *set* ⇒ - ($\bigsqcup_R$) **where**
$\bigsqcup_R A = (if\ (A = \{\})\ then\ Chaos\ else\ \bigsqcup\ A)$

definition *rea-design-par* :: - ⇒ - ⇒ - (**infixr** $\|_R$ *85*) **where**
$P \|_R Q = RH((pre_R(P) \wedge pre_R(Q)) \vdash (P^t{}_f \wedge Q^t{}_f))$

lemma *Miracle-greatest*:
assumes *P is CSP*
shows $P \sqsubseteq Miracle$
proof −
have $P = RH\ (pre_R(P) \vdash peri_R(P) \diamond post_R(P))$
by (*metis CSP-reactive-tri-design assms*)
also have ... $\sqsubseteq RH(true \vdash false)$
by (*rule RH-monotone*, *rel-auto*)
also have $RH(true \vdash false) = RH(true \vdash false \diamond false)$
by (*simp add*: *wait′-cond-def cond-def*)
finally show *?thesis*
by (*simp add*: *Miracle-def*)
qed

lemma *Chaos-least*:
assumes *P is CSP*
shows $Chaos \sqsubseteq P$
proof −
have $Chaos = RH(true)$
by (*simp add*: *Chaos-def design-def*)
also have ... $\sqsubseteq RH(pre_R(P) \vdash peri_R(P) \diamond post_R(P))$
by (*simp add*: *RH-monotone*)
also have $RH(pre_R(P) \vdash peri_R(P) \diamond post_R(P)) = P$
by (*metis CSP-reactive-tri-design assms*)
finally show *?thesis* **.**
qed

lemma *Miracle-left-zero*:
assumes *P is CSP*
shows (*Miracle* ;; *P*) = *Miracle*
proof −
have $(Miracle\ ;;\ P) = (RH(true \vdash false \diamond false)\ ;;\ RH\ (pre_R(P) \vdash peri_R(P) \diamond post_R(P)))$
by (*metis CSP-reactive-tri-design Miracle-def assms*)
also have ... $= RH(true \vdash false \diamond false)$
by (*simp add*: *RH-tri-design-composition R1-false R2s-true R2s-false R2c-true R1-true-comp unrest usubst*)
also have ... = *Miracle*
by (*simp add*: *Miracle-def*)
finally show *?thesis* **.**
qed

thm *CSP-reactive-design*

lemma *Chaos-def′*: $Chaos = RH(false \vdash true)$

by (*simp add*: *Chaos-def design-false-pre*)

lemma *Chaos-left-zero*:
assumes P *is CSP*
shows $(Chaos\ ;;\ P) = Chaos$
proof −
have $(Chaos\ ;;\ P) = (RH(false \vdash true \diamond true)\ ;;\ RH\ (pre_R(P) \vdash peri_R(P) \diamond post_R(P)))$
by (*metis CSP-reactive-tri-design Chaos-def assms*)
also have ... $= RH\ ((\neg\ R1\ true \wedge \neg\ (R1\ true \wedge \neg\ \$wait'\ ;;\ R1\ (\neg\ R2c\ (pre_R\ P)))) \vdash$
$(true \vee (R1\ true\ ;;\ R1\ (R2c\ (peri_R\ P)))) \diamond (R1\ true\ ;;\ R1\ (R2c\ (post_R\ P))))$
by (*simp add*: *RH-tri-design-composition R2s-true R1-true-comp R2s-false unrest*, *metis* (*no-types*) *R1-R2s-R2c R1-negate-R1*)
also have ... $= RH\ ((\neg\ \$ok \vee R1\ true \vee (R1\ true \wedge \neg\ \$wait'\ ;;\ R1\ (\neg\ R2c\ (pre_R\ P)))) \vee$
$\$ok' \wedge (true \vee (R1\ true\ ;;\ R1\ (R2c\ (peri_R\ P)))) \diamond (R1\ true\ ;;\ R1\ (R2c\ (post_R\ P))))$
by (*simp add*: *design-def impl-alt-def*)
also have ... $= RH(R1((\neg\ \$ok \vee R1\ true \vee (R1\ true \wedge \neg\ \$wait'\ ;;\ R1\ (\neg\ R2c\ (pre_R\ P)))) \vee$
$\$ok' \wedge (true \vee (R1\ true\ ;;\ R1\ (R2c\ (peri_R\ P)))) \diamond (R1\ true\ ;;\ R1\ (R2c\ (post_R\ P)))))$
by (*simp add*: *R1-R2c-commute R1-R3c-commute R1-idem RH-R2c-def*)
also have ... $= RH(R1((\neg\ \$ok \vee true \vee (R1\ true \wedge \neg\ \$wait'\ ;;\ R1\ (\neg\ R2c\ (pre_R\ P)))) \vee$
$\$ok' \wedge (true \vee (R1\ true\ ;;\ R1\ (R2c\ (peri_R\ P)))) \diamond (R1\ true\ ;;\ R1\ (R2c\ (post_R\ P)))))$
by (*metis* (*no-types*, *hide-lams*) *R1-disj R1-idem*)
also have ... $= RH(true)$
by (*simp add*: *R1-R2c-commute R1-R3c-commute R1-idem RH-R2c-def*)
also have ... $= Chaos$
by (*simp add*: *Chaos-def design-def*)
finally show *?thesis* **.**
qed

lemma *RH-design-choice*:
$(RH(P \vdash Q_1 \diamond Q_2) \sqcap RH(R \vdash S_1 \diamond S_2)) = RH((P \wedge R) \vdash ((Q_1 \vee S_1) \diamond (Q_2 \vee S_2)))$
proof −
have $(RH(P \vdash Q_1 \diamond Q_2) \sqcap RH(R \vdash S_1 \diamond S_2)) = RH((P \vdash Q_1 \diamond Q_2) \sqcap (R \vdash S_1 \diamond S_2))$
by (*simp add*: *disj-upred-def*[*THEN sym*] *RH-disj*[*THEN sym*])
also have ... $= RH\ ((P \wedge R) \vdash (Q_1 \diamond Q_2 \vee S_1 \diamond S_2))$
by (*simp add*: *design-choice*)
also have ... $= RH\ ((P \wedge R) \vdash ((Q_1 \vee S_1) \diamond (Q_2 \vee S_2)))$
proof −
have $(Q_1 \diamond Q_2 \vee S_1 \diamond S_2) = ((Q_1 \vee S_1) \diamond (Q_2 \vee S_2))$
by (*rel-auto*)
thus *?thesis* **by** *simp*
qed
finally show *?thesis* **.**
qed

lemma *USUP-CSP-closed*:
assumes $A \neq \{\}\ \forall\ P \in A.\ P$ *is CSP*
shows $(\bigsqcap\ A)$ *is CSP*
proof −
from *assms* **have** A: $A = CSP\ `\ A$
by (*auto simp add*: *Healthy-def rev-image-eqI*)
also have $(\bigsqcap\ ...) = (\bigsqcap\ P \in A.\ CSP(P))$
by *auto*
also have ... $= (\bigsqcap\ P \in A \cdot CSP(P))$
by (*simp add*: *USUP-as-Sup-collect*)
also have ... $= (\bigsqcap\ P \in A \cdot RH((\neg\ P^f{}_f) \vdash P^t{}_f))$

by (*metis* (*no-types*) *CSP-RH-design-form*)
also have ... = $RH(\bigsqcap\ P \in A \cdot (\neg\ P^f{}_f) \vdash P^t{}_f)$
by (*simp add*: *RH-USUP assms*(*1*))
also have ... = $RH((\bigsqcup\ P \in A \cdot \neg\ P^f{}_f) \vdash (\bigsqcap\ P \in A \cdot P^t{}_f))$
by (*simp add*: *design-USUP assms*)
also have ... = *CSP*(...)
by (*simp add*: *CSP-RH-design unrest*)
finally show *?thesis*
by (*simp add*: *Healthy-def CSP-idem*)
qed

lemma *UINF-CSP-closed*:
assumes $A \neq \{\}$ $\forall\ P \in A.\ P$ *is CSP*
shows $(\bigsqcup\ A)$ *is CSP*
proof −
from *assms* **have** *A*: $A = CSP$ ‘ A
by (*auto simp add*: *Healthy-def rev-image-eqI*)
also have $(\bigsqcup\ ...) = (\bigsqcup\ P \in A.\ CSP(P))$
by *auto*
also have ... = $(\bigsqcup\ P \in A \cdot CSP(P))$
by (*simp add*: *UINF-as-Inf-collect*)
also have ... = $(\bigsqcup\ P \in A \cdot RH((\neg\ P^f{}_f) \vdash P^t{}_f))$
by (*simp add*: *CSP-RH-design-form*)
also have ... = $RH(\bigsqcup\ P \in A \cdot (\neg\ P^f{}_f) \vdash P^t{}_f)$
by (*simp add*: *RH-UINF assms*(*1*))
also have ... = $RH\ ((\bigsqcap\ P \in A \cdot \neg\ P^f{}_f) \vdash (\bigsqcup\ P \in A \cdot \neg\ P^f{}_f \Rightarrow P^t{}_f))$
by (*simp add*: *design-UINF*)
also have ... = *CSP*(...)
by (*simp add*: *CSP-RH-design unrest*)
finally show *?thesis*
by (*simp add*: *Healthy-def CSP-idem*)
qed

lemma *CSP-sup-closed*:
assumes $\forall\ P \in A.\ P$ *is CSP*
shows $(\bigsqcap_R\ A)$ *is CSP*
proof (*cases* $A = \{\}$)
case *True*
moreover have *Miracle is CSP*
by (*simp add*: *Miracle-def Healthy-def CSP-RH-design unrest*)
ultimately show *?thesis*
by (*simp add*: *reactive-sup-def*)
next
case *False*
with *USUP-CSP-closed assms* **show** *?thesis*
by (*auto simp add*: *reactive-sup-def*)
qed

lemma *CSP-sup-below*:
assumes $\forall\ Q \in A.\ Q$ *is CSP* $P \in A$
shows $\bigsqcap_R\ A \sqsubseteq P$
using *assms*
by (*auto simp add*: *reactive-sup-def Sup-upper*)

lemma *CSP-sup-upper-bound*:

assumes $\forall\ Q \in A.\ Q$ *is CSP* $\forall\ Q \in A.\ P \sqsubseteq Q$ P *is CSP*
shows $P \sqsubseteq \bigsqcap_R A$
proof (*cases* $A = \{\}$)
case *True*
thus *?thesis*
by (*simp add*: *reactive-sup-def Miracle-greatest assms*)
next
case *False*
thus *?thesis*
by (*simp add*: *reactive-sup-def cSup-least assms*)
qed

lemma *CSP-inf-closed*:
assumes $\forall\ P \in A.\ P$ *is CSP*
shows $(\bigsqcup_R A)$ *is CSP*
proof (*cases* $A = \{\}$)
case *True*
moreover have *Chaos is CSP*
by (*simp add*: *Chaos-def Healthy-def CSP-RH-design unrest*)
ultimately show *?thesis*
by (*simp add*: *reactive-inf-def*)
next
case *False*
with *UINF-CSP-closed assms* **show** *?thesis*
by (*auto simp add*: *reactive-inf-def*)
qed

lemma *CSP-inf-above*:
assumes $\forall\ Q \in A.\ Q$ *is CSP* $P \in A$
shows $P \sqsubseteq \bigsqcup_R A$
using *assms*
by (*auto simp add*: *reactive-inf-def Inf-lower*)

lemma *CSP-inf-lower-bound*:
assumes $\forall\ P \in A.\ P$ *is CSP* $\forall\ P \in A.\ P \sqsubseteq Q$ Q *is CSP*
shows $\bigsqcup_R A \sqsubseteq Q$
proof (*cases* $A = \{\}$)
case *True*
thus *?thesis*
by (*simp add*: *reactive-inf-def Chaos-least assms*)
next
case *False*
thus *?thesis*
by (*simp add*: *reactive-inf-def cInf-greatest assms*)
qed

lemma *assigns-lift-rea-unfold*:
$(\$wait' =_u \$wait \wedge \$tr' =_u \$tr \wedge \lceil\langle\sigma\rangle_a\rceil_R) = \lceil\langle\sigma \oplus_s \Sigma_r\rangle_a\rceil_D$
by (*rel-auto*)

lemma *assigns-lift-des-unfold*:
$(\$ok' =_u \$ok \wedge \lceil\langle\sigma\rangle_a\rceil_D) = \langle\sigma \oplus_s \Sigma_D\rangle_a$
by (*rel-auto*)

lemma *assigns-rea-comp-lemma*:

assumes $\$ok \sharp P$ $\$wait \sharp P$
shows $((\$tr' =_u \$tr \wedge \lceil\langle\sigma\rangle_a\rceil_R) ;; P) = (\lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger P)$
proof –
have $((\$tr' =_u \$tr \wedge \lceil\langle\sigma\rangle_a\rceil_R) ;; P) =$
$((\$ok' =_u \$ok \wedge \$wait' =_u \$wait \wedge \$tr' =_u \$tr \wedge \lceil\langle\sigma\rangle_a\rceil_R) ;; P)$
by (*simp add*: *seqr-insert-ident-left unrest assms*)
also have ... $= (\langle\sigma \oplus_s \Sigma_R\rangle_a ;; P)$
by (*simp add*: *assigns-lift-rea-unfold assigns-lift-des-unfold*, *rel-auto*)
also have ... $= (\lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger P)$
by (*simp add*: *assigns-r-comp*)
finally show *?thesis* **.**
qed

lemma *R1-R2s-frame*:
R1 (*R2s* $(\$tr' =_u \$tr \wedge \lceil P\rceil_R)) = (\$tr' =_u \$tr \wedge \lceil P\rceil_R)$
apply (*rel-auto*)
using *minus-zero-eq* **apply** *blast*
done

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

lemma *assigns-rea-comp*:
assumes $\$ok \sharp P$ $\$ok \sharp Q_1$ $\$ok \sharp Q_2$ $\$wait \sharp P$ $\$wait \sharp Q_1$ $\$wait \sharp Q_2$
Q_1 *is R1* Q_2 *is R1* P *is R2s* Q_1 *is R2s* Q_2 *is R2s*
shows $(\langle\sigma\rangle_R ;; RH(P \vdash Q_1 \diamond Q_2)) = RH(\lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger P \vdash \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_1 \diamond \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_2)$
proof –
have $(\langle\sigma\rangle_R ;; RH(P \vdash Q_1 \diamond Q_2)) =$
$(RH\ (true \vdash false \diamond (\$tr' =_u \$tr \wedge \lceil\langle\sigma\rangle_a\rceil_R)) ;; RH\ (P \vdash Q_1 \diamond Q_2))$
by (*simp add*: *assigns-rea-def*)
also have ... $= RH\ ((\neg\ ((\$tr' =_u \$tr \wedge \lceil\langle\sigma\rangle_a\rceil_R) \wedge \neg\ \$wait' ;;$
$R1\ (\neg\ P))) \vdash \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_1 \diamond \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_2)$
by (*simp add*: *RH-tri-design-composition unrest assms R2s-true R1-false R1-R2s-frame Healthy-if assigns-rea-comp-lemma*)
also have ... $= RH\ ((\neg\ ((\$tr' =_u \$tr \wedge \lceil\langle\sigma\rangle_a\rceil_R) \wedge \$wait' =_u \ll False\gg ;;$
$R1\ (\neg\ P))) \vdash \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_1 \diamond \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_2)$
by (*simp add*: *false-alt-def*[*THEN sym*])
also have ... $= RH\ ((\neg\ ((\$tr' =_u \$tr \wedge \lceil\langle\sigma\rangle_a\rceil_R)[\![false/\$wait']\!] ;;$
$(R1\ (\neg\ P))[\![false/\$wait]\!])) \vdash \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_1 \diamond \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_2)$
by (*simp add*: *seqr-left-one-point false-alt-def*)
also have ... $= RH\ ((\neg\ ((\$tr' =_u \$tr \wedge \lceil\langle\sigma\rangle_a\rceil_R) ;; (R1\ (\neg\ P)))) \vdash \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_1 \diamond \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_2)$
by (*simp add*: *R1-def usubst unrest assms*)
also have ... $= RH\ ((\neg\ \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger R1\ (\neg\ P)) \vdash \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_1 \diamond \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_2)$
by (*simp add*: *assigns-rea-comp-lemma assms unrest*)
also have ... $= RH\ ((\neg\ R1\ (\neg\ \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger P)) \vdash \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_1 \diamond \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_2)$
by (*simp add*: *R1-def usubst unrest*)
also have ... $= RH\ ((\lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger P) \vdash \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_1 \diamond \lceil\sigma \oplus_s \Sigma_R\rceil_s \dagger Q_2)$
by (*simp add*: *RH-design-R1-neg-precond*)
finally show *?thesis* **.**
qed

lemma *RH-design-par*:
assumes
$\$ok' \sharp P_1$ $\$wait \sharp P_1$ $\$ok' \sharp P_2$ $\$wait \sharp P_2$

$\$ok' \sharp Q_1$ $\$wait \sharp Q_1$ $\$ok' \sharp Q_2$ $\$wait \sharp Q_2$
shows $RH(P_1 \vdash Q_1) \parallel_R RH(P_2 \vdash Q_2) = RH((P_1 \wedge P_2) \vdash (Q_1 \wedge Q_2))$
proof −
have $RH(P_1 \vdash Q_1) \parallel_R RH(P_2 \vdash Q_2) =$
$RH\ ((\neg\ R1\ (R2c\ (\neg\ P_1[\![true/\$ok]\!])) \wedge \neg\ R1\ (R2c\ (\neg\ P_2[\![true/\$ok]\!]))) \vdash$
$(R1\ (R2s\ (\$ok \wedge P_1 \Rightarrow Q_1)) \wedge R1\ (R2s\ (\$ok \wedge P_2 \Rightarrow Q_2))))$
by (*simp add*: *rea-design-par-def rea-pre-RH-design RH-postcondition*, *simp add*: *usubst assms*)
also have ... =
$RH\ ((P_1[\![true/\$ok]\!] \wedge P_2[\![true/\$ok]\!]) \vdash$
$(R1\ (R2s\ (\$ok \wedge P_1 \Rightarrow Q_1)) \wedge R1\ (R2s\ (\$ok \wedge P_2 \Rightarrow Q_2))))$
by (*metis* (*no-types*, *hide-lams*) *R2c-and R2c-not RH-design-pre-R2c RH-design-pre-neg-conj-R1 double-negation*)
also have ... = $RH\ ((P_1 \wedge P_2) \vdash (R1\ (R2s\ (\$ok \wedge P_1 \Rightarrow Q_1)) \wedge R1\ (R2s\ (\$ok \wedge P_2 \Rightarrow Q_2))))$
by (*metis conj-pos-var-subst design-def subst-conj vwb-lens-ok*)
also have ... = $RH\ ((P_1 \wedge P_2) \vdash (R1\ (R2s\ ((\$ok \wedge P_1 \Rightarrow Q_1) \wedge (\$ok \wedge P_2 \Rightarrow Q_2)))))$
by (*simp add*: *R1-conj R2s-conj*)
also have ... = $RH\ ((P_1 \wedge P_2) \vdash ((\$ok \wedge P_1 \Rightarrow Q_1) \wedge (\$ok \wedge P_2 \Rightarrow Q_2)))$
by (*metis* (*mono-tags*, *lifting*) *RH-design-export-R1 RH-design-export-R2s*)
also have ... = $RH\ ((P_1 \wedge P_2) \vdash (Q_1 \wedge Q_2))$
by (*rel-auto*)
finally show *?thesis* **.**
qed

lemma *RH-tri-design-par*:
assumes
$\$ok' \sharp P_1$ $\$wait \sharp P_1$ $\$ok' \sharp P_2$ $\$wait \sharp P_2$
$\$ok' \sharp Q_1$ $\$wait \sharp Q_1$ $\$ok' \sharp Q_2$ $\$wait \sharp Q_2$
$\$ok' \sharp R_1$ $\$wait \sharp R_1$ $\$ok' \sharp R_2$ $\$wait \sharp R_2$
shows $RH(P_1 \vdash Q_1 \diamond R_1) \parallel_R RH(P_2 \vdash Q_2 \diamond R_2) = RH((P_1 \wedge P_2) \vdash (Q_1 \wedge Q_2) \diamond (R_1 \wedge R_2))$
by (*simp add*: *RH-design-par assms unrest wait′-cond-conj-exchange*)

lemma *RH-design-par-comm*:
$P \parallel_R Q = Q \parallel_R P$
by (*simp add*: *rea-design-par-def utp-pred.inf-commute*)

lemma *RH-design-par-zero*:
assumes *P is CSP*
shows $Chaos \parallel_R P = Chaos$
proof −
have $Chaos \parallel_R P = RH\ (false \vdash true \diamond true) \parallel_R RH\ (pre_R(P) \vdash peri_R(P) \diamond post_R(P))$
by (*simp add*: *Chaos-def CSP-reactive-tri-design assms*)
also have ... = $RH\ (false \vdash peri_R\ P \diamond post_R\ P)$
by (*simp add*: *RH-tri-design-par unrest*)
also have ... = *Chaos*
by (*simp add*: *Chaos-def design-false-pre*)
finally show *?thesis* **.**
qed

lemma *RH-design-par-unit*:
assumes *P is CSP*
shows $Term \parallel_R P = P$
proof −
have $Term \parallel_R P = RH\ (true \vdash true \diamond true) \parallel_R RH\ (pre_R(P) \vdash peri_R(P) \diamond post_R(P))$
by (*simp add*: *Term-def CSP-reactive-tri-design assms*)
also have ... = $RH\ (pre_R\ P \vdash peri_R\ P \diamond post_R\ P)$

by (*simp add*: *RH-tri-design-par unrest*)
also have ... = *P*
by (*simp add*: *CSP-reactive-tri-design assms*)
finally show *?thesis* **.**
qed

15.6 Complete lattice

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

abbreviation *RDES* ≡ *TYPE*(*RDES* × (*'t*::*ordered-cancel-monoid-diff*,*'α*) *alphabet-rp*)

overloading
rdes-hcond == *utp-hcond* :: (*RDES* × (*'t*::*ordered-cancel-monoid-diff*,*'α*) *alphabet-rp*) *itself* ⇒ ((*'t*,*'α*) *alphabet-rp* × (*'t*,*'α*) *alphabet-rp*) *Healthiness-condition*
begin
definition *rdes-hcond* :: (*RDES* × (*'t*::*ordered-cancel-monoid-diff*,*'α*) *alphabet-rp*) *itself* ⇒ ((*'t*,*'α*) *alphabet-rp* × (*'t*,*'α*) *alphabet-rp*) *Healthiness-condition* **where**
[*upred-defs*]: *rdes-hcond T* = *CSP*
end

interpretation *rdes-theory*: *utp-theory TYPE*(*RDES* × (*'t*::*ordered-cancel-monoid-diff*,*'α*) *alphabet-rp*)
by (*unfold-locales*, *simp-all add*: *rdes-hcond-def CSP-idem*)

lemma *Miracle-is-top*: $\top_{\textit{utp-order RDES}}$ = *Miracle*
apply (*auto intro*!:*some-equality simp add*: *atop-def some-equality greatest-def utp-order-def rdes-hcond-def*)
apply (*metis CSP-sup-closed emptyE reactive-sup-def*)
using *Miracle-greatest* **apply** *blast*
apply (*metis CSP-sup-closed dual-order.antisym equals0D reactive-sup-def Miracle-greatest*)
done

lemma *Chaos-is-bot*: $\bot_{\textit{utp-order RDES}}$ = *Chaos*
apply (*auto intro*!:*some-equality simp add*: *abottom-def some-equality least-def utp-order-def rdes-hcond-def*)
apply (*metis CSP-inf-closed emptyE reactive-inf-def*)
using *Chaos-least* **apply** *blast*
apply (*metis Chaos-least CSP-inf-closed dual-order.antisym equals0D reactive-inf-def*)
done

interpretation *hrd-lattice*: *utp-theory-lattice TYPE*(*RDES* × (*'t*::*ordered-cancel-monoid-diff*,*'α*) *alphabet-rp*)
rewrites *carrier* (*utp-order RDES*) = ⟦*CSP*⟧
and $\top_{\textit{utp-order RDES}}$ = *Miracle*
and $\bot_{\textit{utp-order RDES}}$ = *Chaos*
apply (*unfold-locales*)
apply (*simp-all add*: *Miracle-is-top Chaos-is-bot*)
apply (*simp-all add*: *utp-order-def rdes-hcond-def*)
apply (*rename-tac A*)
apply (*rule-tac x*=$\bigsqcup_R$ *A* **in** *exI*, *auto intro*:*CSP-inf-above CSP-inf-lower-bound CSP-inf-closed simp add*: *least-def Upper-def CSP-inf-above*)
apply (*rename-tac A*)
apply (*rule-tac x*=$\bigsqcap_R$ *A* **in** *exI*, *auto intro*:*CSP-sup-below CSP-sup-upper-bound CSP-sup-closed simp add*: *greatest-def Lower-def CSP-inf-above*)
done

abbreviation *rdes-lfp* :: - ⇒ - (μ_R) **where**
μ_R *F* ≡ $\mu_{\textit{utp-order RDES}}$ *F*

abbreviation *rdes-gfp* :: - $\Rightarrow$ - (ν_R) **where**
$\nu_R\ F \equiv \nu_{utp\text{-}order\ RDES}\ F$

lemma *rdes-lfp-copy*: $[\![$ *mono* F; $F \in [\![CSP]\!] \to [\![CSP]\!]$ $]\!] \Longrightarrow \mu_R\ F = F\ (\mu_R\ F)$
  by (*metis hrd-lattice.LFP-unfold mono-Monotone-utp-order*)

lemma *rdes-gfp-copy*: $[\![$ *mono* F; $F \in [\![CSP]\!] \to [\![CSP]\!]$ $]\!] \Longrightarrow \nu_R\ F = F\ (\nu_R\ F)$
  by (*metis hrd-lattice.GFP-unfold mono-Monotone-utp-order*)

end

References

[1] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2015.

[2] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *FroCoS*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.

[3] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.

[4] A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus: a process specification and verification environment. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, 2012.