

Isabelle/UTP: Mechanised reasoning for the UTP

Simon Foster

Frank Zeyda

October 17, 2016

Contents

1	UTP variables	3
1.1	Deep UTP variables	6
1.2	Cardinalities	6
1.3	Injection functions	7
1.4	Deep variables	9
2	UTP expressions	12
2.1	Evaluation laws for expressions	20
2.2	Misc laws	20
3	Unrestriction	20
4	Substitution	22
4.1	Substitution definitions	23
4.2	Substitution laws	24
4.3	Unrestriction laws	27
5	Alphabet manipulation	27
5.1	Alphabet extension	28
5.2	Alphabet restriction	29
5.3	Alphabet lens laws	30
5.4	Alphabet coercion	30
5.5	Substitution alphabet extension	31
5.6	Substitution alphabet restriction	31
6	Lifting expressions	32
6.1	Lifting definitions	32
6.2	Lifting laws	32
6.3	Unrestriction laws	32
7	Alphabetised Predicates	32
7.1	Predicate syntax	33
7.2	Predicate operators	34
7.3	Proof support	37
7.4	Unrestriction Laws	37
7.5	Substitution Laws	38
7.6	Predicate Laws	40

7.7	Cylindric algebra	46
7.8	Quantifier lifting	47
8	Alphabetised relations	47
8.1	Unrestriction Laws	50
8.2	Substitution laws	51
8.3	Relation laws	52
8.4	Converse laws	58
8.5	Assertions and assumptions	59
8.6	Frame and antiframe	59
8.7	Relational unrestriction	61
8.8	Alphabet laws	63
8.9	Relation algebra laws	63
8.10	Relational alphabet extension	64
8.11	Program values	64
8.12	Relational Hoare calculus	64
8.13	Weakest precondition calculus	65
9	Relational operational semantics	66
10	UTP Theories	67
10.1	UTP theory hierarchy	69
11	Example UTP theory: Boyle's laws	71
11.1	Static invariant	71
11.2	Dynamic invariants	72
12	Designs	74
12.1	Definitions	74
12.2	Design laws	77
12.3	Design preconditions	83
12.4	H1: No observation is allowed before initiation	84
12.5	H2: A specification cannot require non-termination	86
12.6	H3: The design assumption is a precondition	91
12.7	H4: Feasibility	94
12.8	UTP theories	94
13	Concurrent programming	96
13.1	Design parallel composition	96
13.2	Parallel by merge	98
14	Reactive processes	102
14.1	Reactive lemmas	104
14.2	R1: Events cannot be undone	104
14.3	R2	106
14.4	R3	111
14.5	RH laws	112

15	Reactive designs	113
15.1	Commutativity properties	114
15.2	Reactive design composition	114
15.3	Healthiness conditions	121
15.4	Reactive design triples	123
15.5	Signature	132
15.6	Complete lattice	138

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  $\cdot$  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]

```

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

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

UTP variables carry two type parameters, $'a$ that corresponds to the variable's type and $'\alpha$ that corresponds to alphabet of which the variable is a type. There is thus a strong link between alphabets and variables in this model. Variables are characterized by two functions, *var-lookup*

and *var-update*, that respectively lookup and update the variable's value in some alphabetised state space. These functions can readily be extracted from an Isabelle record type.

type-synonym $(\text{'a}, \text{'}\alpha) \text{ uvar} = (\text{'a}, \text{'}\alpha) \text{ lens}$

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

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

translations $\text{VAR } x \Rightarrow \text{FLDLens } x$

abbreviation $\text{semi-uvar} \equiv \text{mwb-lens}$

abbreviation $\text{uvar} \equiv \text{vwb-lens}$

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

definition $\text{in-var} :: (\text{'a}, \text{'}\alpha) \text{ uvar} \Rightarrow (\text{'a}, \text{'}\alpha \times \text{'}\beta) \text{ uvar}$ **where**

$[\text{lens-defs}]: \text{in-var } x = x ;_L \text{fst}_L$

definition $\text{out-var} :: (\text{'a}, \text{'}\beta) \text{ uvar} \Rightarrow (\text{'a}, \text{'}\alpha \times \text{'}\beta) \text{ uvar}$ **where**

$[\text{lens-defs}]: \text{out-var } x = x ;_L \text{snd}_L$

definition $\text{pr-var} :: (\text{'a}, \text{'}\beta) \text{ uvar} \Rightarrow (\text{'a}, \text{'}\beta) \text{ uvar}$ **where**

$[\text{simp}]: \text{pr-var } x = x$

lemma $\text{in-var-semi-uvar} [\text{simp}]:$

$\text{semi-uvar } x \Longrightarrow \text{semi-uvar } (\text{in-var } x)$

by $(\text{simp add: comp-mwb-lens fst-vwb-lens in-var-def})$

lemma $\text{in-var-uvar} [\text{simp}]:$

$\text{uvar } x \Longrightarrow \text{uvar } (\text{in-var } x)$

by $(\text{simp add: comp-vwb-lens fst-vwb-lens in-var-def})$

lemma $\text{out-var-semi-uvar} [\text{simp}]:$

$\text{semi-uvar } x \Longrightarrow \text{semi-uvar } (\text{out-var } x)$

by $(\text{simp add: comp-mwb-lens out-var-def snd-vwb-lens})$

lemma $\text{out-var-uvar} [\text{simp}]:$

$\text{uvar } x \Longrightarrow \text{uvar } (\text{out-var } x)$

by $(\text{simp add: comp-vwb-lens out-var-def snd-vwb-lens})$

lemma $\text{in-out-indep} [\text{simp}]:$

$\text{in-var } x \bowtie \text{out-var } y$

by $(\text{simp add: lens-indep-def in-var-def out-var-def fst-lens-def snd-lens-def lens-comp-def})$

lemma $\text{out-in-indep} [\text{simp}]:$

$\text{out-var } x \bowtie \text{in-var } y$

by $(\text{simp add: lens-indep-def in-var-def out-var-def fst-lens-def snd-lens-def lens-comp-def})$

lemma $\text{in-var-indep} [\text{simp}]:$

$x \bowtie y \Longrightarrow \text{in-var } x \bowtie \text{in-var } y$

by $(\text{simp add: in-var-def out-var-def fst-vwb-lens lens-indep-left-comp})$

lemma $\text{out-var-indep} [\text{simp}]:$

$x \bowtie y \implies \text{out-var } x \bowtie \text{out-var } y$
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*]: $\text{lens-get } (\text{in-var } x) (A, A') = \text{lens-get } x A$
by (*simp add: in-var-def fst-lens-def lens-comp-def*)

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

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

lemma *var-update-out* [*simp*]: $\text{lens-put } (\text{out-var } x) (A, A') v = (A, \text{lens-put } x A' v)$
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 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* :: $('a, 'a) \text{ uvar } (\Sigma)$ **where**
univ-alpha $\equiv 1_L$

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

syntax

-salphaid :: $id \Rightarrow \text{salpha } (- [998] 998)$
-salphavar :: $svar \Rightarrow \text{salpha } (- [998] 998)$

-salphacomp :: $\text{salpha} \Rightarrow \text{salpha} \Rightarrow \text{salpha}$ (**infixr** ; 75)
-svid :: $id \Rightarrow \text{svid } (- [999] 999)$
-svid-alpha :: $\text{svid } (\Sigma)$
-svid-empty :: $\text{svid } (\emptyset)$
-svid-dot :: $\text{svid} \Rightarrow \text{svid} \Rightarrow \text{svid}$ (**-:-** [999,998] 999)
-spvar :: $\text{svid} \Rightarrow \text{svar}$ (**&-** [998] 998)
-sinvar :: $\text{svid} \Rightarrow \text{svar}$ (**\$-** [998] 998)
-soutvar :: $\text{svid} \Rightarrow \text{svar}$ (**\$-'** [998] 998)

consts

svar :: $'v \Rightarrow 'e$
ivar :: $'v \Rightarrow 'e$
ovar :: $'v \Rightarrow 'e$

ad hoc-overloading

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

translations

-salphaid $x \Rightarrow x$
-salphacomp $x y \Rightarrow x +_L y$
-salphavar $x \Rightarrow x$
-svid-alpha $== \Sigma$
-svid-empty $== 0_L$
-svid-dot $x y \Rightarrow y ;_L x$
-svid $x \Rightarrow x$
-sinvar $(\text{-svid-dot } x y) \leq \text{CONST } \text{ivar } (\text{CONST } \text{lens-comp } y x)$
-soutvar $(\text{-svid-dot } x y) \leq \text{CONST } \text{ovar } (\text{CONST } \text{lens-comp } y x)$

```

-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 \ll

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
 $\gg$ 

```

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* (\mathfrak{c})

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

type-synonym *uuniv* = *nat* *set*

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

fun *uuniv* :: *ucard* \Rightarrow *uuniv* *set* ($\mathcal{U}'(-)$) **where**

$$\begin{aligned}\mathcal{U}(\text{fin } n) &= \{\{x\} \mid x. x \leq n\} \mid \\ \mathcal{U}(\aleph_0) &= \{\{x\} \mid x. \text{True}\} \mid \\ \mathcal{U}(c) &= \text{UNIV}\end{aligned}$$

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

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

syntax

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

translations

UCARD('a) == CONST ucard-of (TYPE('a))

lemma *ucard-non-empty*:

$\mathcal{U}(x) \neq \{\}$
by (induct x, auto)

lemma *ucard-of-finite* [simp]:

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

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

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

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

uncountable (UNIV :: 'a set) \implies UCARD('a :: continuum) = c
apply (simp add: ucard-of-def)
using countable-finite **apply** blast

done

1.3 Injection functions

definition *uinject-finite* :: 'a::finite \Rightarrow uuniv **where**

uinject-finite $x = \{\text{to-nat-fin } x\}$

definition *uinject-aleph0* :: 'a::{countable, infinite} \Rightarrow uuniv **where**

uinject-aleph0 $x = \{\text{to-nat-bij } x\}$

definition *uinject-continuum* :: 'a::{continuum, infinite} \Rightarrow uuniv **where**

uinject-continuum $x = \text{to-nat-set-bij } x$

definition *uinject* :: 'a::continuum \Rightarrow uuniv **where**

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

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

lemma *uinject-finite*:

finite (*UNIV* :: '*a*::*continuum set*) \implies *uinject* = ($\lambda x :: 'a. \{to\text{-}nat\text{-}fin\ x\}$)
by (*rule ext*, *auto simp add: uinject-def*)

lemma *uinject-uncountable*:

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

lemma *card-finite-lemma*:

assumes *finite* (*UNIV* :: '*a set*)
shows $x < card\ (UNIV :: 'a\ set) \longleftrightarrow x \leq card\ (UNIV :: 'a\ set) - Suc\ 0$

proof –

have $card\ (UNIV :: 'a\ set) > 0$
by (*simp add: assms finite-UNIV-card-ge-0*)
thus *?thesis*
by *linarith*

qed

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

lemma *uinject-bij*:

bij-betw (*uinject* :: '*a*::*continuum* \Rightarrow *uuniv*) *UNIV* $\mathcal{U}(UCARD('a))$

proof (*cases finite* (*UNIV* :: '*a set*))

case *True* **thus** *?thesis*

apply (*auto simp add: uinject-def bij-betw-def inj-on-def image-def card-finite-lemma[THEN sym]*)

apply (*auto simp add: inj-eq to-nat-fin-inj to-nat-fin-bounded*)

using *to-nat-fin-ex* **apply** *blast*

done

next

case *False* **note** *infinite* = *this* **thus** *?thesis*

proof (*cases countable* (*UNIV* :: '*a set*))

case *True* **thus** *?thesis*

apply (*auto simp add: uinject-def bij-betw-def inj-on-def infinite image-def card-finite-lemma[THEN sym]*)

apply (*meson image-to-nat-on infinite surj-def*)

done

next

case *False* **note** *uncount* = *this* **thus** *?thesis*

apply (*simp add: uinject-uncountable*)

using *to-nat-set-bij* **apply** *blast*

done

qed

qed

lemma *uinject-card* [*simp*]: *uinject* ($x :: 'a :: continuum$) $\in \mathcal{U}(UCARD('a))$

by (*metis bij-betw-def rangeI uinject-bij*)

lemma *uinject-inv* [*simp*]:

uproject (*uinject* x) = x

by (*metis UNIV-I bij-betw-def inv-into-f-f uinject-bij uproject-def*)

lemma *uproject-inv* [*simp*]:

$x \in \mathcal{U}(UCARD('a::continuum)) \implies \text{uinject } ((\text{uproject} :: \text{nat set} \Rightarrow 'a) \ x) = x$
by (*metis bij-betw-inv-into-right uinject-bij uproject-def*)

1.4 Deep variables

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

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

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

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

setup-lifting *type-definition-vstore*

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

setup-lifting *type-definition-dvar*

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

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

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

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

term *fun-lens*

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

```
lift-definition dvar-get :: ('a::continuum) dvar  $\Rightarrow$  vstore  $\Rightarrow$  'a
is  $\lambda$  x s. (uproject :: univ  $\Rightarrow$  'a) (s(x)) .
```

```
lift-definition dvar-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 \implies vstore) **where**
dvar-lens x = (\lfloor lens-get = dvar-get x, lens-put = dvar-put x \rfloor)

```
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  $x :: 'a::\{\text{continuum}, \text{two}\}$  dvar and  $y :: 'b::\{\text{continuum}, \text{two}\}$  dvar
shows  $\text{dvar-lens } x \bowtie \text{dvar-lens } y \longleftrightarrow (\text{dvar-dname } x \neq \text{dvar-dname } y)$ 

```

proof –

```

obtain  $v1\ v2 :: 'b::\{\text{continuum}, \text{two}\}$  where  $v:v1 \neq v2$ 
using two-diff by auto

```

```

obtain  $u :: 'a::\{\text{continuum}, \text{two}\}$  and  $v :: 'b::\{\text{continuum}, \text{two}\}$ 
where  $uv: \text{uinject } u \neq \text{uinject } v$ 
by (metis (full-types) uinject-inv v)

```

show ?thesis

proof (simp add: dvar-lens-def lens-indep-def, transfer, auto simp add: fun-upd-twist)

fix $ya :: \text{dname}$

assume $a1: \text{ucard-of } (\text{TYPE}('b)::'b \text{ itself}) = \text{ucard-of } (\text{TYPE}('a)::'a \text{ itself})$

assume $\text{dname-card } ya = \text{ucard-of } (\text{TYPE}('a)::'a \text{ itself})$

assume $a2: \forall u\ v\ \sigma. (\forall x. \sigma\ x \in \mathcal{U}(\text{dname-card } x)) \longrightarrow \sigma(ya := \text{uinject } (u::'a)) = \sigma(ya := \text{uinject } (v::'b)) \wedge (\text{uproject } (\text{uinject } v)::'a) = \text{uproject } (\sigma\ ya) \wedge (\text{uproject } (\text{uinject } u)::'b) = \text{uproject } (\sigma\ ya)$

obtain $NN :: \text{vstore} \Rightarrow \text{dname} \Rightarrow \text{nat set}$ **where**

$\bigwedge v. \forall d. NN\ v\ d \in \mathcal{U}(\text{dname-card } d)$

by (metis (lifting) Abs-vstore-cases mem-Collect-eq)

then show False

using $a2\ a1$ **by** (metis uinject-card uproject-inv uv)

qed

qed

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

class *vst* =

fixes $\text{vstore-lens} :: \text{vstore} \Longrightarrow 'a\ (\mathcal{V})$

assumes $\text{vstore-vwb-lens } [\text{simp}]: \text{vwb-lens } \text{vstore-lens}$

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

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

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

adhoc-overloading

$\text{ivar } \text{in-dvar}$ **and** $\text{ovar } \text{out-dvar}$ **and** $\text{svar } \text{dvar-lift}$

lemma $\text{uvar-dvar}: \text{uvar } (x\uparrow)$

by (auto intro: comp-vwb-lens simp add: dvar-lift-def)

Deep variables with different names are independent

lemma *dvar-lift-indep-iff*:

fixes $x :: 'a::\{\text{continuum}, \text{two}\}$ *dvar* **and** $y :: 'b::\{\text{continuum}, \text{two}\}$ *dvar*

shows $x\uparrow \bowtie y\uparrow \longleftrightarrow \text{dvar-dname } x \neq \text{dvar-dname } y$

proof –

have $x\uparrow \bowtie y\uparrow \longleftrightarrow \text{dvar-lens } x \bowtie \text{dvar-lens } y$

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*]:
 $x \neq y \implies [x]_d \uparrow \bowtie [y]_d \uparrow$
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* \Rightarrow *svar* (%- [999] 999)
-sout-dvar :: *id* \Rightarrow *svar* (%-' [999] 999)

translations
-sin-dvar $x \Rightarrow$ *CONST in-dvar (CONST mk-dvar IDSTR(x))*
-sout-dvar $x \Rightarrow$ *CONST out-dvar (CONST mk-dvar IDSTR(x))*

definition *MkDVar* $x = [x]_d \uparrow$

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

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

Set up parsing and pretty printing for deep variables

syntax
-dvar :: *id* \Rightarrow *svid* (<->)
-dvar-ty :: *id* \Rightarrow *type* \Rightarrow *svid* (<-::->)
-dvard :: *id* \Rightarrow *logic* (<->_d)
-dvar-tyd :: *id* \Rightarrow *type* \Rightarrow *logic* (<-::->_d)

translations
-dvar $x \Rightarrow$ *CONST MkDVar IDSTR(x)*
-dvar-ty $x\ a \Rightarrow$ *-constrain (CONST MkDVar IDSTR(x)) (-uvar-ty a)*
-dvard $x \Rightarrow$ *CONST MkDVar IDSTR(x)*
-dvar-tyd $x\ a \Rightarrow$ *-constrain (CONST MkDVar IDSTR(x)) (-uvar-ty a)*

```

print-translation <<
  let fun MkDVar-tr' - [name] =
    Const (@{syntax-const -dvar}, dummyT) $
      Name-Utills.mk-id (HOLogic.dest-string (Name-Utills.deep-unmark-const name))
    | MkDVar-tr' - - = raise Match in
    [(@{const-syntax MkDVar}, MkDVar-tr')]
  end
  >>

end

```

2 UTP expressions

```

theory utp-expr
imports
  utp-var
  utp-dvar
begin

```

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

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

```

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

```

```

notation Rep-uexpr (⟦-⟧e)

```

```

lemma uexpr-eq-iff:
  e = f ⟷ (∀ b. ⟦e⟧e b = ⟦f⟧e b)
  using Rep-uexpr-inject[of e f, THEN sym] by (auto)

```

```

named-theorems ueval

```

```

setup-lifting type-definition-uexpr

```

Get the alphabet of an expression

```

definition alpha-of :: ('a, 'α) uexpr ⇒ ('α, 'α) lens (α'(-')) where
  alpha-of e = 1L

```

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

```

lift-definition var :: ('t, 'α) uvar ⇒ ('t, 'α) uexpr is lens-get .

```

```

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

```

```

definition dvar-exp :: 't::continuum dvar ⇒ ('t, 'α::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, 'α) *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, 'α) *uexpr* \Rightarrow ('b, 'α) *uexpr*
is $\lambda f e b. f (e b)$.

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

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

We also define a UTP expression version of function abstract

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

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

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

adhoc-overloading
ulit lit

syntax
 $\text{-uuvar} :: svar \Rightarrow logic$

translations
 $\text{-uuvar } x == CONST \text{ var } x$

syntax
 $\text{-uuvar} :: svar \Rightarrow logic (-)$

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*: $- u = uop \text{ uminus } u$
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:  $u * v = \text{bop } (op *) u v$ 
instance ..
end

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

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

instantiation uexpr :: (inverse, type) inverse
begin
  definition inverse-uexpr :: ( $'a, 'b$ )  $uexpr \Rightarrow ('a, 'b) uexpr$ 
  where inverse-uexpr  $u = \text{uop } \text{inverse } u$ 
instance ..
end

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

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

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

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

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

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  $\wedge \neg Q \leq P$ )
instance proof
  fix x y z :: ('a, 'b) uexpr
  show (x < y) = (x  $\leq$  y  $\wedge \neg y \leq x$ ) by (simp add: less-uexpr-def)
  show x  $\leq$  x by (transfer, auto)
  show x  $\leq$  y  $\implies$  y  $\leq$  z  $\implies$  x  $\leq$  z
    by (transfer, blast intro:order.trans)
  show x  $\leq$  y  $\implies$  y  $\leq$  x  $\implies$  x = y
    by (transfer, rule ext, simp add: eq-iff)
qed
end

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-uepr-def zero-uepr-def plus-uepr-def uminus-uepr-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-semiring*)
done

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

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

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

Set up automation for numerals

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

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

definition *eq-upred* :: (*'a, 'α*) *uepr* \Rightarrow (*'a, 'α*) *uepr* \Rightarrow (*bool, 'α*) *uepr*
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*
upd :: *'f* \Rightarrow *'k* \Rightarrow *'v* \Rightarrow *'f*
uom :: *'f* \Rightarrow *'a* *set*
uran :: *'f* \Rightarrow *'b* *set*
uomres :: *'a* *set* \Rightarrow *'f* \Rightarrow *'f*
uranres :: *'f* \Rightarrow *'b* *set* \Rightarrow *'f*
ucard :: *'f* \Rightarrow *nat*

definition *LNil* = *Nil*

definition *LZero* = *0*

adhoc-overloading

uempty *LZero* **and** *uempty* *LNil* **and**
uapply *fun-apply* **and** *uapply* *nth* **and** *uapply* *pfun-app* **and**
uapply *ffun-app* **and** *uapply* *cgf-apply* **and** *uapply* *tt-apply* **and**
upd *pfun-upd* **and** *upd* *ffun-upd* **and** *upd* *list-update* **and**
uom *Domain* **and** *uom* *fdom* **and** *uom* *fdom* **and** *uom* *seq-dom* **and**
uom *Range* **and** *uran* *pran* **and** *uran* *fran* **and** *uran* *set* **and**
uomres *pdom-res* **and** *uomres* *fdom-res* **and**
uranres *pran-res* **and** *uomres* *fran-res* **and**

ucard card and ucard pcard and ucard length

nonterminal *utuple-args and umaplet and umaplets*

syntax

```

-uc coerce    :: ('a, 'α) uexpr ⇒ type ⇒ ('a, 'α) uexpr (infix :u 50)
-unil        :: ('a list, 'α) uexpr (⟨⟩)
-ulist       :: args => ('a list, 'α) uexpr  ((⟨-⟩))
-uappend     :: ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (infixr ^u 80)
-ulast       :: ('a list, 'α) uexpr ⇒ ('a, 'α) uexpr (lastu'(-))
-ufront      :: ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (frontu'(-))
-uhead       :: ('a list, 'α) uexpr ⇒ ('a, 'α) uexpr (headu'(-))
-utail       :: ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (tailu'(-))
-utake       :: (nat, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (takeu'(-, -))
-udrop       :: (nat, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (dropu'(-, -))
-ucard       :: ('a list, 'α) uexpr ⇒ (nat, 'α) uexpr (#u'(-))
-ufilter     :: ('a list, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a list, 'α) uexpr (infixl |u 75)
-uextract    :: ('a set, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (infixl |u 75)
-uelems      :: ('a list, 'α) uexpr ⇒ ('a set, 'α) uexpr (elemsu'(-))
-usorted     :: ('a list, 'α) uexpr ⇒ (bool, 'α) uexpr (sortedu'(-))
-udistinct   :: ('a list, 'α) uexpr ⇒ (bool, 'α) uexpr (distinctu'(-))
-uless       :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (infix <u 50)
-upeq       :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (infix ≤u 50)
-ugreat      :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (infix >u 50)
-ugeq       :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (infix ≥u 50)
-umin        :: logic ⇒ logic ⇒ logic (minu'(-, -))
-umax        :: logic ⇒ logic ⇒ logic (maxu'(-, -))
-ugcd        :: logic ⇒ logic ⇒ logic (gcdu'(-, -))
-uempset     :: ('a set, 'α) uexpr ({u)
-uset        :: args => ('a set, 'α) uexpr ({(-)}u)
-uunion      :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr (infixl ∪u 65)
-uinter      :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr (infixl ∩u 70)
-umem        :: ('a, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (infix ∈u 50)
-usubset     :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (infix ⊆u 50)
-usubseteq   :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (infix ⊆u 50)
-utuple      :: ('a, 'α) uexpr ⇒ utuple-args ⇒ ('a * 'b, 'α) uexpr ((1'(-, -))u)
-utuple-arg  :: ('a, 'α) uexpr ⇒ utuple-args (-)
-utuple-args :: ('a, 'α) uexpr => utuple-args ⇒ utuple-args  (-, -)
-uunit       :: ('a, 'α) uexpr ((')u)
-ufst        :: ('a × 'b, 'α) uexpr ⇒ ('a, 'α) uexpr (π1'(-))
-usnd        :: ('a × 'b, 'α) uexpr ⇒ ('b, 'α) uexpr (π2'(-))
-uapply      :: ('a ⇒ 'b, 'α) uexpr ⇒ utuple-args ⇒ ('b, 'α) uexpr (-|u [999,0] 999)
-ulambda     :: pttm ⇒ logic ⇒ logic (λ · · - [0, 10] 10)
-udom        :: logic ⇒ logic (domu'(-))
-uran        :: logic ⇒ logic (ranu'(-))
-uinl        :: logic ⇒ logic (inlu'(-))
-uinr        :: logic ⇒ logic (inru'(-))
-umap-empty  :: logic (|u)
-umap-plus   :: logic ⇒ logic ⇒ logic (infixl ⊕u 85)
-umap-minus  :: logic ⇒ logic ⇒ logic (infixl ⊖u 85)
-udom-res    :: logic ⇒ logic ⇒ logic (infixl ≲u 85)
-uran-res    :: logic ⇒ logic ⇒ logic (infixl ≳u 85)
-umaplet     :: [logic, logic] => umaplet (- /|u / -)
              :: umaplet => umaplets  (-)
-UMaplets    :: [umaplet, umaplets] => umaplets (-, -)

```

$-UMapUpd \quad :: [logic, umaplets] \Rightarrow logic \ (-/'(-')_u \ [900,0] \ 900)$
 $-UMap \quad :: umaplets \Rightarrow logic \ ((1[-]_u))$

translations

$f(\downarrow v)_u \leq CONST \ uapply \ f \ v$
 $dom_u(f) \leq CONST \ udom \ f$
 $ran_u(f) \leq CONST \ uran \ f$
 $A \triangleleft_u f \leq CONST \ udomres \ A \ f$
 $f \triangleright_u A \leq CONST \ uranres \ f \ A$
 $\#_u(f) \leq CONST \ ucard \ f$
 $f(k \mapsto v)_u \leq CONST \ uupd \ f \ k \ v$

translations

$x :_u 'a == x :: ('a, -) \ uexpr$
 $\langle \rangle \quad == \ll \square \gg$
 $\langle x, xs \rangle \quad == CONST \ bop \ (op \ \#) \ x \ \langle xs \rangle$
 $\langle x \rangle \quad == CONST \ bop \ (op \ \#) \ x \ \ll \square \gg$
 $x \hat{ }_u y \quad == CONST \ bop \ (op \ @) \ x \ y$
 $last_u(xs) == CONST \ uop \ CONST \ last \ xs$
 $front_u(xs) == CONST \ uop \ CONST \ butlast \ xs$
 $head_u(xs) == CONST \ uop \ CONST \ hd \ xs$
 $tail_u(xs) == CONST \ uop \ CONST \ tl \ xs$
 $drop_u(n,xs) == CONST \ bop \ CONST \ drop \ n \ xs$
 $take_u(n,xs) == CONST \ bop \ CONST \ take \ n \ xs$
 $\#_u(xs) == CONST \ uop \ CONST \ ucard \ xs$
 $elems_u(xs) == CONST \ uop \ CONST \ set \ xs$
 $sorted_u(xs) == CONST \ uop \ CONST \ sorted \ xs$
 $distinct_u(xs) == CONST \ uop \ CONST \ distinct \ xs$
 $xs \downarrow_u A \quad == CONST \ bop \ CONST \ seq-filter \ xs \ A$
 $A \upharpoonright_u xs \quad == CONST \ bop \ (op \ \upharpoonright_i) \ A \ xs$
 $x <_u y \quad == CONST \ bop \ (op \ <) \ x \ y$
 $x \leq_u y \quad == CONST \ bop \ (op \ \leq) \ x \ y$
 $x >_u y \quad == y <_u x$
 $x \geq_u y \quad == y \leq_u x$
 $min_u(x, y) \quad == CONST \ bop \ (CONST \ min) \ x \ y$
 $max_u(x, y) \quad == CONST \ bop \ (CONST \ max) \ x \ y$
 $gcd_u(x, y) \quad == CONST \ bop \ (CONST \ gcd) \ x \ y$
 $\{\}_u \quad == \ll \{\} \gg$
 $\{x, xs\}_u == CONST \ bop \ (CONST \ insert) \ x \ \{xs\}_u$
 $\{x\}_u \quad == CONST \ bop \ (CONST \ insert) \ x \ \ll \{\} \gg$
 $A \cup_u B \quad == CONST \ bop \ (op \ \cup) \ A \ B$
 $A \cap_u B \quad == CONST \ bop \ (op \ \cap) \ A \ B$
 $f \oplus_u g \Rightarrow (f :: ((-, -) \ pfun, -) \ uexpr) + g$
 $f \ominus_u g \Rightarrow (f :: ((-, -) \ pfun, -) \ uexpr) - g$
 $x \in_u A \quad == CONST \ bop \ (op \ \in) \ x \ A$
 $A \subset_u B \quad == CONST \ bop \ (op \ <) \ A \ B$
 $A \subset_u B \quad \leq CONST \ bop \ (op \ \subset) \ A \ B$
 $f \subset_u g \quad \leq CONST \ bop \ (op \ \subset_p) \ f \ g$
 $f \subset_u g \quad \leq CONST \ bop \ (op \ \subset_f) \ f \ g$
 $A \subseteq_u B \quad == CONST \ bop \ (op \ \leq) \ A \ B$
 $A \subseteq_u B \quad \leq CONST \ bop \ (op \ \subseteq) \ A \ B$
 $f \subseteq_u g \quad \leq CONST \ bop \ (op \ \subseteq_p) \ f \ g$
 $f \subseteq_u g \quad \leq CONST \ bop \ (op \ \subseteq_f) \ f \ g$
 $()_u \quad == \ll () \gg$
 $(x, y)_u \quad == CONST \ bop \ (CONST \ Pair) \ x \ y$

$-utuple\ x\ (-utuple\ args\ y\ z) == -utuple\ x\ (-utuple\ arg\ (-utuple\ y\ z))$
 $\pi_1(x) == CONST\ uop\ CONST\ fst\ x$
 $\pi_2(x) == CONST\ uop\ CONST\ snd\ x$
 $f(\downarrow x)_u == CONST\ bop\ CONST\ uapply\ f\ x$
 $\lambda\ x \cdot p == CONST\ ulambda\ (\lambda\ x.\ p)$
 $dom_u(f) == CONST\ uop\ CONST\ udom\ f$
 $ran_u(f) == CONST\ uop\ CONST\ uran\ f$
 $inl_u(x) == CONST\ uop\ CONST\ Inl\ x$
 $inr_u(x) == CONST\ uop\ CONST\ Inr\ x$
 $\square_u == \ll CONST\ uempty \gg$
 $A \triangleleft_u f == CONST\ bop\ (CONST\ udomres)\ A\ f$
 $f \triangleright_u A == CONST\ bop\ (CONST\ uranres)\ f\ A$
 $-UMapUpd\ m\ (-UMaplets\ xy\ ms) == -UMapUpd\ (-UMapUpd\ m\ xy)\ ms$
 $-UMapUpd\ m\ (-umaplet\ x\ y) == CONST\ trop\ CONST\ uupd\ m\ x\ y$
 $-UMap\ ms == -UMapUpd\ \square_u\ ms$
 $-UMap\ (-UMaplets\ ms1\ ms2) <= -UMapUpd\ (-UMap\ ms1)\ ms2$
 $-UMaplets\ ms1\ (-UMaplets\ ms2\ ms3) <= -UMaplets\ (-UMaplets\ ms1\ ms2)\ ms3$
 $f(\downarrow x, y)_u == CONST\ bop\ CONST\ uapply\ f\ (x, y)_u$

Lifting set intervals

syntax

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

lift-definition $ZedSetCompr ::$

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

translations

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

Lifting limits

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

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

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

syntax

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

translations

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

lemmas $uexpr-defs =$

$alpha-of-def$
 $zero-uexpr-def$
 $one-uexpr-def$
 $plus-uexpr-def$

uminus-uepr-def
minus-uepr-def
times-uepr-def
inverse-uepr-def
divide-uepr-def
sgn-uepr-def
abs-uepr-def
mod-uepr-def
eq-upred-def
numeral-uepr-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*]: $\llbracket \langle x \rangle \rrbracket_e b = x$
by (*transfer*, *simp*)

lemma *var-ueval* [*ueval*]: $\llbracket \text{var } x \rrbracket_e b = \text{get}_x b$
by (*transfer*, *simp*)

lemma *uop-ueval* [*ueval*]: $\llbracket \text{uop } f \ x \rrbracket_e b = f (\llbracket x \rrbracket_e b)$
by (*transfer*, *simp*)

lemma *bop-ueval* [*ueval*]: $\llbracket \text{bop } f \ x \ y \rrbracket_e b = f (\llbracket x \rrbracket_e b) (\llbracket y \rrbracket_e b)$
by (*transfer*, *simp*)

lemma *trop-ueval* [*ueval*]: $\llbracket \text{trop } f \ x \ y \ z \rrbracket_e b = f (\llbracket x \rrbracket_e b) (\llbracket y \rrbracket_e b) (\llbracket z \rrbracket_e b)$
by (*transfer*, *simp*)

declare *uepr-defs* [*ueval*]

2.2 Misc laws

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

lemma *lit-num-simps*: $\langle 0 \rangle = 0 \ \langle 1 \rangle = 1 \ \langle \text{numeral } n \rangle = \text{numeral } n \ \langle - \ x \rangle = - \ \langle x \rangle$
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 $x \# p$, if altering the value of x has no effect on the valuation of p . This is a sufficient notion to prove many laws that would ordinarily rely on an fv function.

consts

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

syntax

$-unrest :: salpha \Rightarrow logic \Rightarrow logic \Rightarrow logic \text{ (infix } \# 20)$

translations

$-unrest\ x\ p == CONST\ unrest\ x\ p$

named-theorems $unrest$

method $unrest-tac = (simp\ add:\ unrest)?$

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

is $\lambda x\ e.\ \forall\ b\ v.\ e\ (put_x\ b\ v) = e\ b$.

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

$unrest-dvar-upred\ x\ P = unrest-upred\ (x\uparrow)\ P$

adhoc-overloading

$unrest\ unrest-upred$

lemma $unrest-var-comp\ [unrest]:$

$\llbracket x \# P; y \# P \rrbracket \Longrightarrow x; y \# P$

by $(transfer,\ simp\ add:\ lens-defs)$

lemma $unrest-lit\ [unrest]:\ x \# \llbracket v \rrbracket$

by $(transfer,\ simp)$

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

lemma $unrest-var\ [unrest]:\ \llbracket uvar\ x; x \bowtie y \rrbracket \Longrightarrow y \# var\ x$

by $(transfer,\ auto)$

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

by $(metis\ in-var-indep\ in-var-uvar\ unrest-var)$

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

by $(metis\ out-var-indep\ out-var-uvar\ unrest-var)$

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

fixes $x :: ('a, 'α) uvar$

assumes $uvar\ y$

shows $\$x \# \y'

by $(metis\ prod.collapse\ unrest-upred.rep-eq\ var.rep-eq\ var-lookup-out\ var-update-in)$

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

fixes $x :: ('a, 'α) uvar$

assumes $uvar\ y$

shows $\$x' \# \y

by $(metis\ prod.collapse\ unrest-upred.rep-eq\ var.rep-eq\ var-lookup-in\ var-update-out)$

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

```

by (transfer, simp)

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

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

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

lemma unrest-zero [unrest]:  $x \# 0$ 
  by (simp add: unrest-lit zero-uepr-def)

lemma unrest-one [unrest]:  $x \# 1$ 
  by (simp add: one-uepr-def unrest-lit)

lemma unrest-numeral [unrest]:  $x \# (\text{numeral } n)$ 
  by (simp add: numeral-uepr-simp unrest-lit)

lemma unrest-sgn [unrest]:  $x \# u \implies x \# \text{sgn } u$ 
  by (simp add: sgn-uepr-def unrest-uop)

lemma unrest-abs [unrest]:  $x \# u \implies x \# \text{abs } u$ 
  by (simp add: abs-uepr-def unrest-uop)

lemma unrest-plus [unrest]:  $\llbracket x \# u; x \# v \rrbracket \implies x \# u + v$ 
  by (simp add: plus-uepr-def unrest)

lemma unrest-uminus [unrest]:  $x \# u \implies x \# - u$ 
  by (simp add: uminus-uepr-def unrest)

lemma unrest-minus [unrest]:  $\llbracket x \# u; x \# v \rrbracket \implies x \# u - v$ 
  by (simp add: minus-uepr-def unrest)

lemma unrest-times [unrest]:  $\llbracket x \# u; x \# v \rrbracket \implies x \# u * v$ 
  by (simp add: times-uepr-def unrest)

lemma unrest-divide [unrest]:  $\llbracket x \# u; x \# v \rrbracket \implies x \# u / v$ 
  by (simp add: divide-uepr-def unrest)

lemma unrest-ulambda [unrest]:
   $\llbracket \text{uvar } v; \bigwedge x. v \# F \ x \rrbracket \implies v \# (\lambda x. F \ x)$ 
  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 (' α , ' β) *psubst* = ' α *alphabet* \Rightarrow ' β *alphabet*

type-synonym ' α *usubst* = ' α *alphabet* \Rightarrow ' α *alphabet*

lift-definition *subst* :: (' α , ' β) *psubst* \Rightarrow ('a, ' β) *uexpr* \Rightarrow ('a, ' α) *uexpr* **is**
 $\lambda \sigma e b. e (\sigma b)$.

ad hoc-overloading

usubst subst

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

consts *subst-upd* :: (' α , ' β) *psubst* \Rightarrow 'v \Rightarrow ('a, ' α) *uexpr* \Rightarrow (' α , ' β) *psubst*

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

definition *subst-upd-dvar* :: (' α , ' β ::*vst*) *psubst* \Rightarrow 'a::*continuum dvar* \Rightarrow ('a, ' α) *uexpr* \Rightarrow (' α , ' β) *psubst* **where**

subst-upd-dvar $\sigma x v = \text{subst-upd-uvar } \sigma (x \uparrow) v$

ad hoc-overloading

subst-upd subst-upd-uvar and subst-upd subst-upd-dvar

Lookup the expression associated with a variable in a substitution

lift-definition *usubst-lookup* :: (' α , ' β) *psubst* \Rightarrow ('a, ' β) *uvar* \Rightarrow ('a, ' α) *uexpr* ($\langle \cdot \rangle_s$)
is $\lambda \sigma x b. \text{get}_x (\sigma b)$.

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

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

ad hoc-overloading

unrest unrest-usubst

nonterminal *smaplet and smaplets*

syntax

-*smaplet* :: [*salpha*, 'a] \Rightarrow *smaplet* (- / \mapsto_s / -)
 :: *smaplet* \Rightarrow *smaplets* (-)
-*SMaplets* :: [*smaplet*, *smaplets*] \Rightarrow *smaplets* (-, / -)
-*SubstUpd* :: ['m *usubst*, *smaplets*] \Rightarrow 'm *usubst* (-/'(-) [900,0] 900)
-*Subst* :: *smaplets* \Rightarrow 'a \Rightarrow 'b ((1[-]))

translations

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

Deletion of a substitution maplet

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

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

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

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

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

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

apply (*auto*)
done

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

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

lemma *usubst-apply-unrest* [*usubst*]:
 $\llbracket \text{uvar } x; x \# \sigma \rrbracket \implies \langle \sigma \rangle_s x = \text{var } x$
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*]:
 $\text{uvar } x \implies \text{id} -_s x = \text{id}$
by (*simp add: subst-del-def subst-upd-uvar-def, transfer, auto*)

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

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

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

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

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

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

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

lemma *unrest-usubst-del* [*unrest*]: $\llbracket \text{uvar } x; x \# (\langle \sigma \rangle_s x); x \# \sigma -_s x \rrbracket \implies x \# (\sigma \dagger P)$
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* :: $('a, 'α) \text{ uvar} \Rightarrow ('b, 'α) \text{ uvar} \Rightarrow \text{bool}$ **where**
 $[no-atp]: \text{var-name-ord } x \ y = \text{True}$

syntax

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

translations

$\text{-var-name-ord } x \ y == \text{CONST var-name-ord } x \ y$

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*]: $\sigma \dagger \text{uop } f \ v = \text{uop } f \ (\sigma \dagger v)$

by (*transfer, simp*)

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

by (*transfer, simp*)

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

by (*transfer, simp*)

lemma *subst-plus* [*usubst*]: $\sigma \dagger (x + y) = \sigma \dagger x + \sigma \dagger y$

by (*simp add: plus-uepr-def subst-bop*)

lemma *subst-times* [*usubst*]: $\sigma \dagger (x * y) = \sigma \dagger x * \sigma \dagger y$

by (*simp add: times-uepr-def subst-bop*)

lemma *subst-minus* [*usubst*]: $\sigma \dagger (x - y) = \sigma \dagger x - \sigma \dagger y$

by (*simp add: minus-uepr-def subst-bop*)

lemma *subst-uminus* [*usubst*]: $\sigma \dagger (-x) = -(\sigma \dagger x)$

by (*simp add: uminus-uepr-def subst-uop*)

lemma *usubst-sgn* [*usubst*]: $\sigma \dagger \text{sgn } x = \text{sgn } (\sigma \dagger x)$

by (*simp add: sgn-uepr-def subst-uop*)

lemma *usubst-abs* [*usubst*]: $\sigma \dagger \text{abs } x = \text{abs } (\sigma \dagger x)$

by (*simp add: abs-uepr-def subst-uop*)

lemma *subst-zero* [*usubst*]: $\sigma \dagger 0 = 0$

by (*simp add: zero-uepr-def subst-lit*)

lemma *subst-one* [*usubst*]: $\sigma \dagger 1 = 1$

by (*simp add: one-uepr-def subst-lit*)

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

by (*simp add: eq-upred-def usubst*)

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

by (*transfer, simp*)

lemma *subst-upd-comp* [*usubst*]:

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

shows $\varrho(x \mapsto_s v) \circ \sigma = (\varrho \circ \sigma)(x \mapsto_s \sigma \dagger v)$

by (rule ext, simp add: uexpr-defs subst-upd-uvar-def, transfer, simp)

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

syntax

-*psubst* :: [logic, svars, uexprs] \Rightarrow logic
 -*subst* :: logic \Rightarrow uexprs \Rightarrow salphas \Rightarrow logic ((-'/-)) [999,0,0] 1000)
 -*uexprs* :: [logic, uexprs] \Rightarrow uexprs (-,/ -)
 :: logic \Rightarrow uexprs (-)
 -*svars* :: [svar, svars] \Rightarrow svars (-,/ -)
 :: svar \Rightarrow svars (-)
 -*salphas* :: [salpha, salphas] \Rightarrow salphas (-,/ -)
 :: salpha \Rightarrow salphas (-)

translations

-*subst* *P es vs* \Rightarrow CONST *subst* (-*psubst* (CONST *id*) *vs es*) *P*
 -*psubst* *m* (-*salphas* *x xs*) (-*uexprs* *v vs*) \Rightarrow -*psubst* (-*psubst* *m x v*) *xs vs*
 -*psubst* *m x v* \Rightarrow CONST *subst-upd* *m x v*
 $P[v/\$x] \leq$ CONST *usubst* (CONST *subst-upd* (CONST *id*) (CONST *ivar* *x*) *v*) *P*
 $P[v/\$x'] \leq$ CONST *usubst* (CONST *subst-upd* (CONST *id*) (CONST *ovar* *x*) *v*) *P*

lemma *subst-singleton*:

fixes *x* :: ('a, 'α) uvar
 assumes *x* $\#$ σ
 shows $\sigma(x \mapsto_s v) \upharpoonright P = (\sigma \upharpoonright P)[v/x]$
 using *assms*
 by (simp add: usubst, metis comp-apply id-apply subst-upd-uvar-def unrest-usubst-def)

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

4.3 Unrestriction laws

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

$\llbracket \text{semi-uvar } x; x \# v \rrbracket \Longrightarrow x \# P[v/x]$
 by (transfer, auto simp add: subst-upd-uvar-def unrest-upred-def)

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

semi-uvar *x* $\Longrightarrow x \# id$
 by (simp add: unrest-usubst-def)

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

$\llbracket x \bowtie y; x \# \sigma; x \# v \rrbracket \Longrightarrow x \# \sigma(y \mapsto_s v)$
 by (simp add: subst-upd-uvar-def unrest-usubst-def unrest-upred.rep-eq lens-indep-comm)

lemma *unrest-subst* [*unrest*]:

$\llbracket x \# P; x \# \sigma \rrbracket \Longrightarrow x \# (\sigma \upharpoonright P)$
 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* :: ($'a, 'b$) *uexpr* \Rightarrow ($'\beta, 'a$) *lens* \Rightarrow ($'a, 'a$) *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-tac*)

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

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

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

lemma *aext-numeral* [*alpha*]: *numeral* $n \oplus_p a = \text{numeral } n$
by (*pred-tac*)

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

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

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

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

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

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

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

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

by (*pred-tac*)

lemma *aext-var* [*alpha*]:

$var\ x \oplus_p a = var\ (x ;_L a)$

by (*pred-tac*)

lemma *aext-true* [*alpha*]: $true \oplus_p a = true$

by (*pred-tac*)

lemma *aext-false* [*alpha*]: $false \oplus_p a = false$

by (*pred-tac*)

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

by (*pred-tac*)

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

by (*pred-tac*)

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

by (*pred-tac*)

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

by (*pred-tac*)

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

by (*pred-tac*)

lemma *unrest-aext* [*unrest*]:

$\llbracket mwb\text{-}lens\ a;\ x \# p \rrbracket \Longrightarrow unrest\ (x ;_L a)\ (p \oplus_p a)$

by (*transfer*, *simp add: lens-comp-def*)

lemma *unrest-aext-indep* [*unrest*]:

$a \bowtie b \Longrightarrow b \# (p \oplus_p a)$

by *pred-tac*

5.2 Alphabet restriction

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

lift-definition *arestr* :: $(\alpha, \beta) \text{ uexpr} \Rightarrow (\beta, \alpha) \text{ lens} \Rightarrow (\alpha, \beta) \text{ uexpr} \rightarrow (\text{infixr } \downarrow_p \ 90)$

is $\lambda\ P\ x\ b.\ P\ (create_x\ b)$.

lemma *arestr-id* [*alpha*]: $P \downarrow_p 1_L = P$

by (*pred-tac*)

lemma *arestr-aext* [*simp*]: $mwb\text{-}lens\ a \Longrightarrow (P \oplus_p a) \downarrow_p a = P$

by (*pred-tac*)

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

lemma *aext-arestr* [*alpha*]:

assumes *mwb-lens a bij-lens* $(a +_L b)\ a \bowtie b\ b \# P$

shows $(P \downarrow_p a) \oplus_p a = P$

proof –

from *assms*(2) **have** $1_L \subseteq_L a +_L b$
by (*simp add: bij-lens-equiv-id lens-equiv-def*)
with *assms*(1,3,4) **show** ?thesis
apply (*auto simp add: alpha-of-def id-lens-def lens-plus-def sublens-def lens-comp-def prod.case-eq-if*)
apply (*pred-tac*)
apply (*metis lens-indep-comm mwb-lens-weak weak-lens.put-get*)
done
qed

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

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

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

lemma *arestr-numeral* [*alpha*]: *numeral* $n \downarrow_p a = \text{numeral } n$
by (*pred-tac*)

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

lemma *arestr-true* [*alpha*]: $\text{true} \downarrow_p a = \text{true}$
by (*pred-tac*)

lemma *arestr-false* [*alpha*]: $\text{false} \downarrow_p a = \text{false}$
by (*pred-tac*)

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

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

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

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

5.3 Alphabet lens laws

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

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

5.4 Alphabet coercion

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

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

syntax

$\text{-}\alpha\text{-coerce} :: \text{salpha} \Rightarrow \text{logic} \Rightarrow \text{logic } (!_{\alpha} \text{ - } \cdot \text{ - } [0, 10] \text{ } 10)$

translations

$\text{-}\alpha\text{-coerce } P \text{ } x == \text{CONST } \alpha\text{-coerce } P \text{ } x$

5.5 Substitution alphabet extension

definition $\text{subst-ext} :: ' \alpha \text{ usubst} \Rightarrow (' \alpha \Rightarrow ' \beta) \Rightarrow ' \beta \text{ usubst}$ (**infix** \oplus_s 65) **where**
 $[\text{upred-defs}]$: $\sigma \oplus_s x = (\lambda s. \text{put}_x s (\sigma (\text{get}_x s)))$

lemma $\text{id-subst-ext } [\text{usubst}, \alpha]$:

$\text{uvar } x \Rightarrow \text{id} \oplus_s x = \text{id}$

by pred-tac

lemma $\text{upd-subst-ext } [\alpha]$:

$\text{uvar } x \Rightarrow \sigma(y \mapsto_s v) \oplus_s x = (\sigma \oplus_s x)(\&x:y \mapsto_s v \oplus_p x)$

by pred-tac

lemma $\text{apply-subst-ext } [\alpha]$:

$\text{uvar } x \Rightarrow (\sigma \uparrow e) \oplus_p x = (\sigma \oplus_s x) \uparrow (e \oplus_p x)$

by (pred-tac)

lemma $\text{aext-upred-eq } [\alpha]$:

$((e =_u f) \oplus_p a) = ((e \oplus_p a) =_u (f \oplus_p a))$

by (pred-tac)

5.6 Substitution alphabet restriction

definition $\text{subst-res} :: ' \alpha \text{ usubst} \Rightarrow (' \beta \Rightarrow ' \alpha) \Rightarrow ' \beta \text{ usubst}$ (**infix** \upharpoonright_s 65) **where**
 $[\text{upred-defs}]$: $\sigma \upharpoonright_s x = (\lambda s. \text{get}_x (\sigma (\text{create}_x s)))$

lemma $\text{id-subst-res } [\alpha, \text{usubst}]$:

$\text{semi-uvar } x \Rightarrow \text{id} \upharpoonright_s x = \text{id}$

by pred-tac

lemma $\text{upd-subst-res } [\alpha]$:

$\text{uvar } x \Rightarrow \sigma(\&x:y \mapsto_s v) \upharpoonright_s x = (\sigma \upharpoonright_s x)(\&y \mapsto_s v \upharpoonright_p x)$

by (pred-tac)

lemma $\text{subst-ext-res } [\alpha, \text{usubst}]$:

$\text{uvar } x \Rightarrow (\sigma \oplus_s x) \upharpoonright_s x = \sigma$

by (pred-tac)

lemma $\text{unrest-subst-alpha-ext } [\text{unrest}]$:

$x \bowtie y \Rightarrow x \# (P \oplus_s y)$

by $(\text{pred-tac}, \text{auto simp add: unrest-usubst-def, metis lens-indep-def})$

end

6 Lifting expressions

```
theory utp-lift
  imports
    utp-alphabet
begin
```

6.1 Lifting definitions

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

abbreviation $\text{lift-pre} :: ('a, 'α) \text{uepr} \Rightarrow ('a, 'α \times 'β) \text{uepr} ([\cdot]_<)$
where $[P]_< \equiv P \oplus_p \text{fst}_L$

abbreviation $\text{drop-pre} :: ('a, 'α \times 'β) \text{uepr} \Rightarrow ('a, 'α) \text{uepr} ([\cdot]_<)$
where $[P]_< \equiv P \upharpoonright_p \text{fst}_L$

abbreviation $\text{lift-post} :: ('a, 'β) \text{uepr} \Rightarrow ('a, 'α \times 'β) \text{uepr} ([\cdot]_>)$
where $[P]_> \equiv P \oplus_p \text{snd}_L$

abbreviation $\text{drop-post} :: ('a, 'α \times 'β) \text{uepr} \Rightarrow ('a, 'β) \text{uepr} ([\cdot]_>)$
where $[P]_> \equiv P \upharpoonright_p \text{snd}_L$

6.2 Lifting laws

lemma $\text{lift-pre-var} [\text{simp}]$:
 $[\text{var } x]_< = \$x$
by (alpha-tac)

lemma $\text{lift-post-var} [\text{simp}]$:
 $[\text{var } x]_> = \$x'$
by (alpha-tac)

6.3 Unrestriction laws

lemma $\text{unrest-dash-var-pre} [\text{unrest}]$:
fixes $x :: ('a, 'α) \text{uvar}$
shows $\$x' \# [p]_<$
by (pred-tac)

end

7 Alphabetised Predicates

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

An alphabetised predicate is simply a boolean valued expression

type-synonym $'α \text{upred} = (\text{bool}, 'α) \text{uepr}$

translations

$(type) \text{ '}\alpha \text{ upred} \leq (type) (bool, \text{ '}\alpha) uexpr$

named-theorems *upred-defs*

7.1 Predicate syntax

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

no-notation

conj (**infixr** \wedge 35) **and**
disj (**infixr** \vee 30) **and**
Not (\neg - [40] 40)

consts

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

adhoc-overloading

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

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

syntax

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

translations

-uex $x \ P == \text{CONST } uex \ x \ P$
-uall $x \ P == \text{CONST } uall \ x \ P$
 $\exists \ x \cdot P == \text{CONST } ushEx \ (\lambda \ x. \ P)$
 $\exists \ x \in A \cdot P \Rightarrow \exists \ x \cdot \langle x \rangle \in_u A \wedge P$
 $\forall \ x \cdot P == \text{CONST } ushAll \ (\lambda \ x. \ P)$
 $\forall \ x \in A \cdot P \Rightarrow \forall \ x \cdot \langle x \rangle \in_u A \Rightarrow P$

$$\forall x \mid P \cdot Q \Rightarrow \forall x \cdot P \Rightarrow Q$$

7.2 Predicate operators

We chose to maximally reuse definitions and laws built into HOL. For this reason, when introducing the core operators we proceed by lifting operators from the polymorphic algebraic hierarchy of HOL. Thus the initial definitions take place in the context of type class instantiations. We first introduce our own class called *refine* that will add the refinement operator syntax to the HOL partial order class.

class *refine* = *order*

abbreviation *refineBy* :: 'a::*refine* \Rightarrow 'a \Rightarrow bool (infix \sqsubseteq 50) **where**
P \sqsubseteq *Q* \equiv *less-eq* *Q* *P*

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

no-notation *inf* (infixl \sqcap 70)

notation *inf* (infixl \sqcup 70)

no-notation *sup* (infixl \sqcup 65)

notation *sup* (infixl \sqcap 65)

no-notation *Inf* (\sqcap - [900] 900)

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

no-notation *Sup* (\sqcup - [900] 900)

notation *Sup* (\sqcap - [900] 900)

no-notation *bot* (\perp)

notation *bot* (\top)

no-notation *top* (\top)

notation *top* (\perp)

no-syntax

-*INF1* :: *pttrns* \Rightarrow 'b \Rightarrow 'b (($\exists \sqcap$ -./ -) [0, 10] 10)
-*INF* :: *pttrn* \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b (($\exists \sqcap$ - \in -./ -) [0, 0, 10] 10)
-*SUP1* :: *pttrns* \Rightarrow 'b \Rightarrow 'b (($\exists \sqcup$ -./ -) [0, 10] 10)
-*SUP* :: *pttrn* \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b (($\exists \sqcup$ - \in -./ -) [0, 0, 10] 10)

syntax

-*INF1* :: *pttrns* \Rightarrow 'b \Rightarrow 'b (($\exists \sqcup$ -./ -) [0, 10] 10)
-*INF* :: *pttrn* \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b (($\exists \sqcup$ - \in -./ -) [0, 0, 10] 10)
-*SUP1* :: *pttrns* \Rightarrow 'b \Rightarrow 'b (($\exists \sqcap$ -./ -) [0, 10] 10)
-*SUP* :: *pttrn* \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b (($\exists \sqcap$ - \in -./ -) [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* \Rightarrow ('a, 'b) *uexpr* \Rightarrow ('a, 'b) *uexpr*
is $\lambda P Q A. \text{sup } (P A) (Q A) .$

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

```

  is  $\lambda P Q A. \inf (P A) (Q A)$  .
instance
  by (intro-classes) (transfer, auto)+
end

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

```

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

```

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

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

```

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

```

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

```

```

lift-definition USUP :: ('a  $\Rightarrow$  'α upred)  $\Rightarrow$  ('a  $\Rightarrow$  ('b::complete-lattice, 'α) uexpr)  $\Rightarrow$  ('b, 'α) uexpr
is  $\lambda P F b. \text{Sup } \{\llbracket F x \rrbracket_e b \mid x. \llbracket P x \rrbracket_e b\}$  .

```

```

lift-definition UINF :: ('a  $\Rightarrow$  'α upred)  $\Rightarrow$  ('a  $\Rightarrow$  ('b::complete-lattice, 'α) uexpr)  $\Rightarrow$  ('b, 'α) uexpr
is  $\lambda P F b. \text{Inf } \{\llbracket F x \rrbracket_e b \mid x. \llbracket P x \rrbracket_e b\}$  .

```

```

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

```

```

syntax
-USup    :: idt  $\Rightarrow$  logic  $\Rightarrow$  logic          ( $\prod$  - · - [0, 10] 10)
-USup-mem :: idt  $\Rightarrow$  logic  $\Rightarrow$  logic  $\Rightarrow$  logic ( $\prod$  -  $\in$  - · - [0, 10] 10)
-USUP    :: idt  $\Rightarrow$  logic  $\Rightarrow$  logic  $\Rightarrow$  logic ( $\prod$  - | - · - [0, 0, 10] 10)
-UInf    :: idt  $\Rightarrow$  logic  $\Rightarrow$  logic          ( $\sqcup$  - · - [0, 10] 10)
-UInf-mem :: idt  $\Rightarrow$  logic  $\Rightarrow$  logic  $\Rightarrow$  logic ( $\sqcup$  -  $\in$  - · - [0, 10] 10)
-UINF    :: idt  $\Rightarrow$  logic  $\Rightarrow$  logic  $\Rightarrow$  logic ( $\sqcup$  - | - · - [0, 10] 10)

```

translations

$$\begin{aligned}
\Box x \mid P \cdot F &\Rightarrow \text{CONST USUP } (\lambda x. P) (\lambda x. F) \\
\Box x \cdot F &== \Box x \mid \text{true} \cdot F \\
\Box x \cdot F &== \Box x \mid \text{true} \cdot F \\
\Box x \in A \cdot F &\Rightarrow \Box x \mid \ll x \gg \in_u \ll A \gg \cdot F \\
\Box x \mid P \cdot F &\leq \text{CONST USUP } (\lambda x. P) (\lambda y. F) \\
\Box x \mid P \cdot F &\Rightarrow \text{CONST UINF } (\lambda x. P) (\lambda x. F) \\
\Box x \cdot F &== \Box x \mid \text{true} \cdot F \\
\Box x \in A \cdot F &\Rightarrow \Box x \mid \ll x \gg \in_u \ll A \gg \cdot F \\
\Box x \mid P \cdot F &\leq \text{CONST UINF } (\lambda x. P) (\lambda y. F)
\end{aligned}$$

We also define the other predicate operators

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

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

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

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

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

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

We have to add a u subscript to the closure operator as I don't want to override the syntax for HOL lists (we'll be using them later).

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

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

ad hoc-overloading

uttrue true-upred **and**
 $\text{ufalse false-upred}$ **and**
 unot not-upred **and**
 uconj conj-upred **and**
 udisj disj-upred **and**
 uimpl impl **and**
 wiff iff-upred **and**
 uex ex **and**
 uall all **and**
 ushEx shEx **and**
 ushAll shAll

syntax

$\text{-uneg} \quad :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \text{ (infixl } \neq_u 50)$
 $\text{-unmem} \quad :: ('a, 'a) \text{ uexpr} \Rightarrow ('a \text{ set}, 'a) \text{ uexpr} \Rightarrow (\text{bool}, 'a) \text{ uexpr} \text{ (infix } \notin_u 50)$

translations

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

7.3 Proof support

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

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

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

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

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

7.4 Unrestriction Laws

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

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

lemma *unrest-conj* [*unrest*]: $\ll x \# (P :: 'a \text{ upred}); x \# Q \gg \implies x \# P \wedge Q$
by (*pred-tac*)

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

lemma *unrest-USUP* [*unrest*]:
 $\ll (\bigwedge i. x \# P(i)); (\bigwedge i. x \# Q(i)) \gg \implies x \# (\prod i \mid P(i) \cdot Q(i))$
by (*simp add: USUP-def, pred-tac*)

lemma *unrest-UINF* [*unrest*]:
 $\ll (\bigwedge i. x \# P(i)); (\bigwedge i. x \# Q(i)) \gg \implies x \# (\bigsqcup i \mid P(i) \cdot Q(i))$
by (*simp add: UINF-def, pred-tac*)

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

lemma *unrest-iff* [*unrest*]: $\ll x \# P; x \# Q \gg \implies x \# P \Leftrightarrow Q$

by (*pred-tac*)

lemma *unrest-not* [*unrest*]: $x \# (P :: 'a \text{ upred}) \implies x \# (\neg P)$
 by (*pred-tac*)

The sublens proviso can be thought of as membership below.

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

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

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

lemma *unrest-all-in* [*unrest*]:
 $\llbracket \text{semi-uvar } y; x \subseteq_L y \rrbracket \implies x \# (\forall y \cdot P)$
 by *pred-tac*

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

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

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

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

7.5 Substitution Laws

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.6 Predicate Laws

Showing that predicates form a Boolean Algebra (under the predicate operators) gives us many useful laws.

interpretation *boolean-algebra* *diff-upred not-upred conj-upred op ≤ op < disj-upred false-upred true-upred*
by (*unfold-locales*, *pred-tac+*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

by (*pred-tac*) (*pred-tac*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *USUP-cong-eq*:

$$\llbracket \bigwedge x. P_1(x) = P_2(x); \bigwedge x. 'P_1(x) \Rightarrow Q_1(x) =_u Q_2(x)' \rrbracket \implies$$

$$(\bigcap x \mid P_1(x) \cdot Q_1(x)) = (\bigcap x \mid P_2(x) \cdot Q_2(x))$$

 by (*simp add: USUP-def, pred-tac, metis*)

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

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

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

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

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

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

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

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

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

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

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

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

lemma *conj-eq-out-var-subst*:
fixes $x :: ('a, 'α) \text{uvar}$
assumes *uvar* x
shows $(P \wedge \$x' =_u v) = (P \llbracket v/\$x' \rrbracket \wedge \$x' =_u v)$
using *assms*
by (*pred-tac*, (*metis vwb-lens-wb wb-lens.get-put*))+)

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

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

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

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

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

lemma *shEx-cong*: $\llbracket \bigwedge x. P \ x = Q \ x \rrbracket \implies \text{shEx } P = \text{shEx } Q$
by (*pred-tac*)

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

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

lemma *shAll-cong*: $\llbracket \bigwedge x. P\ x = Q\ x \rrbracket \implies \text{shAll}\ P = \text{shAll}\ Q$
by (*pred-tac*)

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

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

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

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

lemma *uvar-assign-exists*:
 $\text{uvar}\ x \implies \exists v. b = \text{put}_x\ b\ v$
by (*rule-tac* $x = \text{get}_x\ b$ **in** *exI*, *simp*)

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

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

lemma *eq-split-substI*:
assumes $\text{uvar}\ x \wedge v. P[\llbracket v \rrbracket/x] = Q[\llbracket v \rrbracket/x]$
shows $P = Q$
using *assms*(1) *assms*(2) *eq-split-subst* **by** *blast*

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

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

```

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

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

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

lemma exists-twice:  $\text{semi-uvar } x \implies (\exists x \cdot \exists x \cdot P) = (\exists x \cdot P)$ 
  by (pred-tac)

lemma all-twice:  $\text{semi-uvar } x \implies (\forall x \cdot \forall x \cdot P) = (\forall x \cdot P)$ 
  by (pred-tac)

lemma exists-sub:  $\llbracket \text{mwb-lens } y; x \subseteq_L y \rrbracket \implies (\exists x \cdot \exists y \cdot P) = (\exists y \cdot P)$ 
  by pred-tac

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

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

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

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

```

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

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

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

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

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

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

lemma *unrest-as-exists*:
 $vwb\text{-}lens\ x \implies (x \# P) \longleftrightarrow ((\exists x \cdot P) = P)$
by (*pred-tac*, *metis* *vwb-lens.put-eq*)

7.7 Cylindric algebra

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

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

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

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

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

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

lemma *C6*:
assumes $wb\text{-}lens\ x\ x \bowtie y\ x \bowtie z$
shows $(\&y =_u \&z) = (\exists x \cdot \&y =_u \&x \wedge \&x =_u \&z)$

using *assms*
by (*pred-tac*, (*metis lens-indep-def*)⁺)

lemma *C7*:

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

lemma *shEx-lift-conj-2* [*uquant-lift*]:

$(P \wedge (\exists x \cdot Q(x))) = (\exists x \cdot P \wedge Q(x))$
by *pred-tac*

end

8 Alphabetised relations

theory *utp-rel*

imports

utp-pred

utp-lift

begin

default-sort *type*

named-theorems *urel-defs*

consts

useq $:: 'a \Rightarrow 'b \Rightarrow 'c$ (**infixr** $:: 15$)

uskip $:: 'a$ (*II*)

definition *in α* $:: ('a, 'a \times 'b) \text{ wvar where}$

in α $= \llbracket \text{lens-get} = \text{fst}, \text{lens-put} = \lambda (A, A') v. (v, A') \rrbracket$

definition *out α* $:: ('b, 'a \times 'b) \text{ wvar where}$

out α $= \llbracket \text{lens-get} = \text{snd}, \text{lens-put} = \lambda (A, A') v. (A, v) \rrbracket$

declare *in α -def* [*urel-defs*]

declare *out α -def* [*urel-defs*]

lemma *var-in-alpha* [*simp*]: $x ;_L \text{in}\alpha = \text{ivar } x$

by (*simp add: fst-lens-def in α -def in-var-def*)

lemma *var-out-alpha* [*simp*]: $x ;_L \text{out}\alpha = \text{ovar } x$

by (*simp add: out α -def out-var-def snd-lens-def*)

lemma *out-alpha-in-indep* [*simp*]:

$out\alpha \bowtie in\text{-}var\ x\ in\text{-}var\ x \bowtie out\alpha$
by (*simp-all add: in-var-def out α -def lens-indep-def fst-lens-def lens-comp-def*)

lemma *in-alpha-out-indep* [*simp*]:
 $in\alpha \bowtie out\text{-}var\ x\ out\text{-}var\ x \bowtie in\alpha$
by (*simp-all add: in-var-def in α -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*:
 $\Sigma \approx_L in\alpha +_L out\alpha$
by (*metis fst-lens-def fst-snd-id-lens in α -def lens-equiv-reft out α -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*:: $'\alpha\ upred \Rightarrow '\alpha\ upred \Rightarrow '\alpha\ upred \Rightarrow '\alpha\ upred$
 $((\exists - \triangleleft - \triangleright / -) [14,0,15] 14)$
where $(P \triangleleft b \triangleright Q) \equiv (b \wedge P) \vee ((\neg b) \wedge Q)$

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

lift-definition *segr*:: $(('\alpha \times '\beta)\ upred) \Rightarrow ((''\beta \times '\gamma)\ upred) \Rightarrow (''\alpha \times '\gamma)\ upred$
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**
 $[urel\text{-}defs]: skip\text{-}ra\ v = (\$v' =_u \$v)$

syntax
 $\text{-}skip\text{-}ra :: salpha \Rightarrow logic\ (II.)$

translations
 $\text{-}skip\text{-}ra\ v == CONST\ skip\text{-}ra\ v$

abbreviation *usubst-rel-lift* :: $'\alpha\ usubst \Rightarrow (''\alpha \times '\beta)\ usubst\ ([_]_s)$ **where**
 $[\sigma]_s \equiv \sigma \oplus_s in\alpha$

abbreviation *usubst-rel-drop* :: $(''\alpha \times '\alpha)\ usubst \Rightarrow '\alpha\ usubst\ ([_]_s)$ **where**
 $[\sigma]_s \equiv \sigma \upharpoonright_s in\alpha$

definition *assigns-ra* :: $'\alpha\ usubst \Rightarrow (''\beta, '\alpha)\ lens \Rightarrow '\alpha\ hrelation\ (\langle _ \rangle_-)$ **where**
 $\langle \sigma \rangle_a = ([\sigma]_s \upharpoonright II_a)$

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**
 $skip\text{-}r = assigns\text{-}r\ id$

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

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

abbreviation *assign-2-r* ::

$(t1, 'a) \text{ uvar} \Rightarrow (t2, 'a) \text{ uvar} \Rightarrow (t1, 'a) \text{ uexpr} \Rightarrow (t2, 'a) \text{ uexpr} \Rightarrow 'a \text{ hrelation}$

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

nonterminal

svid-list and *uexpr-list*

syntax

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

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

-uexpr-unit :: $(a, 'a) \text{ uexpr} \Rightarrow \text{uexpr-list}\ (-\ [40]\ 40)$

-uexpr-list :: $(a, 'a) \text{ uexpr} \Rightarrow \text{uexpr-list} \Rightarrow \text{uexpr-list}\ (-, / -\ [40, 40]\ 40)$

-assignment :: *svid-list* \Rightarrow *uexprs* \Rightarrow $'a \text{ hrelation}$ (**infixr** := 62)

-mk-usubst :: *svid-list* \Rightarrow *uexprs* \Rightarrow $'a \text{ usubst}$

translations

-mk-usubst $\sigma\ (-\text{svid-unit}\ x)\ v == \sigma(\&x \mapsto_s v)$

-mk-usubst $\sigma\ (-\text{svid-list}\ x\ xs)\ (-\text{uexprs}\ v\ vs) == (-\text{mk-usubst}\ (\sigma(\&x \mapsto_s v))\ xs\ vs)$

-assignment $xs\ vs \Rightarrow \text{CONST assigns-r}\ (-\text{mk-usubst}\ (\text{CONST id})\ xs\ vs)$

$x := v <= \text{CONST assigns-r}\ (\text{CONST subst-upd}\ (\text{CONST id})\ (\text{CONST svar}\ x)\ v)$

$x := v <= \text{CONST assigns-r}\ (\text{CONST subst-upd}\ (\text{CONST id})\ x\ v)$

$x, y := u, v <= \text{CONST assigns-r}\ (\text{CONST subst-upd}\ (\text{CONST subst-upd}\ (\text{CONST id})\ (\text{CONST svar}\ x)\ u)\ (\text{CONST svar}\ y)\ v)$

ad hoc-overloading

useq seqr and

uskip skip-r

definition *rassume* :: $'a \text{ upred} \Rightarrow 'a \text{ hrelation}\ (-^\top\ [999]\ 999)$ **where**

[*urel-defs*]: *rassume* $c = (II \triangleleft c \triangleright_r \text{false})$

definition *rasassert* :: $'a \text{ upred} \Rightarrow 'a \text{ hrelation}\ (-_\perp\ [999]\ 999)$ **where**

[*urel-defs*]: *rasassert* $c = (II \triangleleft c \triangleright_r \text{true})$

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

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

We describe some properties of relations

definition *ufunctional* :: $(a, 'b) \text{ relation} \Rightarrow \text{bool}$

where *ufunctional* $R \longleftrightarrow (II \sqsubseteq (R^- ;; R))$

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

definition *uinj* :: $(a, 'b) \text{ relation} \Rightarrow \text{bool}$

where *uinj* $R \longleftrightarrow II \sqsubseteq (R ;; R^-)$

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

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

definition *lift-test* :: $'a \text{ condition} \Rightarrow 'a \text{ hrelation}\ ([_]_t)$

where $\lceil b \rceil_t = (\lceil b \rceil_{<} \wedge II)$

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

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

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

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

8.1 Unrestriction Laws

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

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

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

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

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

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

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

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

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

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

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

lemma *unrest-pre-in-var* [*unrest*]:

$x \# p1 \implies \$x \# \lceil p1 \rceil_<$
by (*transfer*, *simp*)

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

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

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

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

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

8.2 Substitution laws

lemma *subst-seq-left* [*usubst*]:
 $\text{out}\alpha \# \sigma \implies \sigma \dagger (P ;; Q) = ((\sigma \dagger P) ;; Q)$
by (*rel-tac*, *simp-all add: unrest-usubst-def, (metis (no-types, lifting) Pair-inject old.prod.case surjective-pairing)+*)

lemma *subst-seq-right* [*usubst*]:
 $\text{in}\alpha \# \sigma \implies \sigma \dagger (P ;; Q) = (P ;; (\sigma \dagger Q))$
by (*rel-tac*, *simp-all add: unrest-usubst-def, (metis (no-types, lifting) Pair-inject old.prod.case 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-tac*

lemma *subst-skip-r* [*usubst*]:
 $\text{out}\alpha \# \sigma \implies \sigma \dagger II = \langle \lfloor \sigma \rfloor_s \rangle_a$
by (*rel-tac*, *auto simp add: unrest-usubst-def, (metis (mono-tags, lifting) case-prod-conv prod.sel(1) sndI surjective-pairing)+*)

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

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

lemma *subst-lift-upd* [*usubst*]:
fixes $x :: ('a, 'a) \text{uvar}$
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 in α -def in-var-def*)

lemma *subst-drop-upd* [*usubst*]:

fixes $x :: ('a, 'α) \text{ uvar}$
shows $\lfloor \sigma(\$x \mapsto_s v) \rfloor_s = \lfloor \sigma \rfloor_s (x \mapsto_s \lfloor v \rfloor_<)$
by (*pred-tac*, *simp add: inα-def prod.case-eq-if*)

lemma *subst-lift-pre* [*usubst*]: $\lfloor \sigma \rfloor_s \uparrow \lfloor b \rfloor_< = \lfloor \sigma \uparrow b \rfloor_<$
by (*metis apply-subst-ext fst-lens-def fst-vwb-lens inα-def*)

lemma *unrest-usubst-lift-in* [*unrest*]:
 $x \# P \implies \$x \# \lfloor P \rfloor_s$
by (*pred-tac*, *auto simp add: unrest-usubst-def inα-def*)

lemma *unrest-usubst-lift-out* [*unrest*]:
fixes $x :: ('a, 'α) \text{ uvar}$
shows $\$x' \# \lfloor P \rfloor_s$
by (*pred-tac*, *auto simp add: unrest-usubst-def inα-def*)

8.3 Relation laws

Homogeneous relations form a quantale

abbreviation *truer* :: $'α \text{ hrelation}$ (*true_h*) **where**
truer $\equiv \text{true}$

abbreviation *false_r* :: $'α \text{ hrelation}$ (*false_h*) **where**
false_r $\equiv \text{false}$

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

lemma *drop-pre-inv* [*simp*]: $\llbracket \text{out}α \# p \rrbracket \implies \lceil \lfloor p \rfloor_< \rceil_< = p$
by (*pred-tac*, *auto simp add: outα-def lens-create-def fst-lens-def prod.case-eq-if*)

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

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

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

While loops with invariant decoration

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *cond-var-subst-left*:
assumes *uvar x*
shows $(P \triangleleft \$x \triangleright Q) = (P \llbracket \text{true}/\$x \rrbracket \triangleleft \$x \triangleright Q)$
using *assms* **by** (*metis cond-def conj-pos-var-subst*)

lemma *cond-var-subst-right*:
assumes *uvar x*
shows $(P \triangleleft \$x \triangleright Q) = (P \triangleleft \$x \triangleright Q \llbracket \text{false}/\$x \rrbracket)$
using *assms* **by** (*metis cond-def conj-neg-var-subst*)

lemma *cond-var-split*:
 $\text{uvar } x \Longrightarrow (P \llbracket \text{true}/x \rrbracket \triangleleft \text{var } x \triangleright P \llbracket \text{false}/x \rrbracket) = P$
by (*rel-tac*, (*metis (full-types) vwb-lens.put-eq*)+)

lemma *cond-seq-left-distr*:

$out\alpha \# b \implies ((P \triangleleft b \triangleright Q) ;; R) = ((P ;; R) \triangleleft b \triangleright (Q ;; R))$
by (*rel-tac*, *blast+*)

lemma *cond-seq-right-distr*:

$in\alpha \# b \implies (P ;; (Q \triangleleft b \triangleright R)) = ((P ;; Q) \triangleleft b \triangleright (P ;; R))$
by (*rel-tac*, *blast+*)

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

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

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

$(II ;; P) = P$
by *rel-tac*

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

$(P ;; II) = P$
by *rel-tac*

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

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

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

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

lemma *seqr-mono*:

$\llbracket P_1 \sqsubseteq P_2; Q_1 \sqsubseteq Q_2 \rrbracket \implies (P_1 ;; Q_1) \sqsubseteq (P_2 ;; Q_2)$
by (*rel-tac*, *blast*)

lemma *spec-refine*:

$Q \sqsubseteq (P \wedge R) \implies (P \Rightarrow Q) \sqsubseteq R$
by (*rel-tac*)

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

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

lemma *skip-var*:

fixes $x :: (bool, 'a) \text{ uvar}$
shows $(\$x \wedge II) = (II \wedge \$x')$
by (*rel-tac*)

lemma *seqr-exists-left*:

$semi\text{-uvar } x \implies ((\exists \$x \cdot P) ;; Q) = (\exists \$x \cdot (P ;; Q))$
by (*rel-tac*)

lemma *seqr-exists-right*:

$semi\text{-uvar } x \implies (P ;; (\exists \$x' \cdot Q)) = (\exists \$x' \cdot (P ;; Q))$
by (*rel-tac*)

lemma *assigns-subst* [*usubst*]:

$\lceil \sigma \rceil_s \dagger \langle \varrho \rangle_a = \langle \varrho \circ \sigma \rangle_a$
by (*rel-tac*)

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

by *rel-tac*

lemma *assigns-r-feasible*:

$(\langle \sigma \rangle_a ;; \text{true}) = \text{true}$
by (*rel-tac*)

lemma *assign-subst* [*usubst*]:

$\llbracket \text{semi-uvar } x; \text{semi-uvar } y \rrbracket \implies [\$x \mapsto_s \lceil u \rceil_<] \dagger (y := v) = (x, y := u, [x \mapsto_s u] \dagger v)$
by *rel-tac*

lemma *assigns-idem*: $\text{semi-uvar } x \implies (x, x := u, v) = (x := v)$

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*:

$\text{bij } f \implies \langle f \rangle_a^- = \langle \text{inv } f \rangle_a$
by (*rel-tac, simp-all add: bij-is-inj bij-is-surj surj-f-inv-f*)

lemma *assign-r-comp*: $\text{semi-uvar } x \implies (x := u ;; P) = P[\lceil u \rceil_</\$x]$

by (*simp add: assigns-r-comp usubst*)

lemma *assign-test*: $\text{semi-uvar } x \implies (x := \llbracket u \rrbracket ;; x := \llbracket v \rrbracket) = (x := \llbracket v \rrbracket)$

by (*simp add: assigns-comp subst-upd-comp subst-lit usubst-upd-idem*)

lemma *assign-twice*: $\llbracket \text{uvar } x; x \# f \rrbracket \implies (x := e ;; x := f) = (x := f)$

by (*simp add: assigns-comp usubst*)

lemma *assign-commute*:

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

lemma *assign-cond*:

fixes $x :: ('a, 'α) \text{uvar}$
assumes $\text{out } α \# b$
shows $(x := e ;; (P \triangleleft b \triangleright Q)) = ((x := e ;; P) \triangleleft (b[\lceil e \rceil_</\$x]) \triangleright (x := e ;; Q))$
by *rel-tac*

lemma *assign-rcond*:

fixes $x :: ('a, 'α) \text{uvar}$
shows $(x := e ;; (P \triangleleft b \triangleright_r Q)) = ((x := e ;; P) \triangleleft (b[\lceil e/x \rceil]) \triangleright_r (x := e ;; Q))$
by *rel-tac*

lemma *assign-r-alt-def*:

fixes $x :: ('a, 'α) \text{uvar}$
shows $x := v = II[\lceil v \rceil_</\$x]$
by *rel-tac*

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

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

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

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

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

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

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

lemma *assign-unfold*:
 $\text{uvar } x \implies (x := v) = (\$x' =_u [v]_< \wedge II|_\alpha x)$
apply (*rel-tac*, *auto simp add: comp-def*)
using *vwb-lens.put-eq* **by** *fastforce*

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

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

lemma *seqr-and-distr-ufunc*:
 $\text{ufunctional } P \implies (P ;; (Q \wedge R)) = ((P ;; Q) \wedge (P ;; R))$
by *rel-tac*

lemma *seqr-and-distl-uinj*:
 $\text{uinj } R \implies ((P \wedge Q) ;; R) = ((P ;; R) \wedge (Q ;; R))$
by (*rel-tac*, *metis*)

lemma *seqr-unfold*:
 $(P ;; Q) = (\exists v \cdot P[\llbracket v \rrbracket / \$ \Sigma] \wedge Q[\llbracket v \rrbracket / \$ \Sigma])$
by *rel-tac*

lemma *seqr-middle*:

assumes *uvar x*
shows $(P ;; Q) = (\exists v \cdot P[\llbracket v \rrbracket / \$x'] ;; Q[\llbracket v \rrbracket / \$x])$
using *assms*
apply (*rel-tac*)
apply (*rename-tac xa P Q a b y*)
apply (*rule-tac x=get_{xa} y in exI*)
apply (*rule-tac x=y in exI*)
apply (*simp*)

done

lemma *seqr-left-one-point*:

assumes *uvar x*
shows $(P \wedge (\$x' =_u \llbracket v \rrbracket) ;; Q) = (P[\llbracket v \rrbracket / \$x'] ;; Q[\llbracket v \rrbracket / \$x])$
using *assms*
by (*rel-tac, metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-right-one-point*:

assumes *uvar x*
shows $(P ;; (\$x =_u \llbracket v \rrbracket) \wedge Q) = (P[\llbracket v \rrbracket / \$x'] ;; Q[\llbracket v \rrbracket / \$x])$
using *assms*
by (*rel-tac, metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-insert-ident-left*:

assumes *uvar x* $\$x' \# P$ $\$x \# Q$
shows $((\$x' =_u \$x \wedge P) ;; Q) = (P ;; Q)$
using *assms*
by (*rel-tac, meson vwb-lens-wb wb-lens-weak weak-lens.put-get*)

lemma *seqr-insert-ident-right*:

assumes *uvar x* $\$x' \# P$ $\$x \# Q$
shows $(P ;; (\$x' =_u \$x \wedge Q)) = (P ;; Q)$
using *assms*
by (*rel-tac, metis (no-types, hide-lams) vwb-lens-def wb-lens-def weak-lens.put-get*)

lemma *seq-var-ident-lift*:

assumes *uvar x* $\$x' \# P$ $\$x \# Q$
shows $((\$x' =_u \$x \wedge P) ;; (\$x' =_u \$x) \wedge Q) = (\$x' =_u \$x \wedge (P ;; Q))$
using *assms* **apply** (*rel-tac*)
by (*metis (no-types, lifting) vwb-lens-wb wb-lens-weak weak-lens.put-get*)

theorem *precond-equiv*:

$P = (P ;; \text{true}) \longleftrightarrow (\text{out}\alpha \# P)$
by (*rel-tac*)

theorem *postcond-equiv*:

$P = (\text{true} ;; P) \longleftrightarrow (\text{in}\alpha \# P)$
by (*rel-tac*)

lemma *precond-right-unit*: $\text{out}\alpha \# p \implies (p ;; \text{true}) = p$

by (*metis precondition-equiv*)

lemma *postcond-left-unit*: $\text{in}\alpha \# p \implies (\text{true} ;; p) = p$

by (*metis postcond-equiv*)

```

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

```

8.4 Converse laws

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma not-convr [simp]:  $(\neg p)^- = (\neg p^-)$ 
  by (pred-tac)

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

theorem *seqr-pre-transfer*: $\text{in}\alpha \# q \implies ((P \wedge q) ;; R) = (P ;; (q^- \wedge R))$
by (*rel-tac*)

theorem *seqr-post-out*: $\text{in}\alpha \# r \implies (P ;; (Q \wedge r)) = ((P ;; Q) \wedge r)$
by (*rel-tac*, *blast+*)

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

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

lemma *seqr-pre-out*: $\text{out}\alpha \# p \implies ((p \wedge Q) ;; R) = (p \wedge (Q ;; R))$
by (*rel-tac*, *blast+*)

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

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

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

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

8.5 Assertions and assumptions

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

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

8.6 Frame and antiframe

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

definition *antiframe* :: $('a, 'a) \text{ lens} \Rightarrow 'a \text{ hrelation} \Rightarrow 'a \text{ hrelation}$ **where**
[*urel-defs*]: $\text{antiframe } x P = (H|_x \wedge P)$

syntax
-frame :: $\text{salph} \Rightarrow \text{logic} \Rightarrow \text{logic} \text{ } (-: \llbracket - \rrbracket [64, 0] 80)$
-antiframe :: $\text{salph} \Rightarrow \text{logic} \Rightarrow \text{logic} \text{ } (-: [-] [64, 0] 80)$

translations
-frame $x P == \text{CONST frame } x P$

-antiframe $x P == \text{CONST antiframe } x P$

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

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

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

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

While loop laws

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

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

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

theorem *while-unfold*:

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

8.7 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* :: $(\alpha, \alpha) \text{ uvar} \Rightarrow \alpha \text{ hrelation} \Rightarrow \alpha \text{ hrelation}$
where *RID* $x P = ((\exists \$x \cdot \exists \$x' \cdot P) \wedge \$x' =_u \$x)$

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

lemma *RID-idem*:

semi-uvar $x \Longrightarrow \text{RID}(x)(\text{RID}(x)(P)) = \text{RID}(x)(P)$
 by *rel-tac*

lemma *RID-mono*:

$P \sqsubseteq Q \Longrightarrow \text{RID}(x)(P) \sqsubseteq \text{RID}(x)(Q)$
 by *rel-tac*

lemma *RID-skip-r*:

uvar $x \Longrightarrow \text{RID}(x)(II) = II$
 apply *rel-tac*

using *vwb-lens.put-eq* **apply** *fastforce*
by *auto*

lemma *RID-disj*:

$\text{RID}(x)(P \vee Q) = (\text{RID}(x)(P) \vee \text{RID}(x)(Q))$
 by *rel-tac*

lemma *RID-conj*:

uvar $x \Longrightarrow \text{RID}(x)(\text{RID}(x)(P) \wedge \text{RID}(x)(Q)) = (\text{RID}(x)(P) \wedge \text{RID}(x)(Q))$
 by *rel-tac*

lemma *RID-assigns-r-diff*:

$\llbracket \text{uvar } x; x \# \sigma \rrbracket \Longrightarrow \text{RID}(x)(\langle \sigma \rangle_a) = \langle \sigma \rangle_a$
 apply (*rel-tac*)
 apply (*auto simp add: unrest-usubst-def*)
 apply (*metis vwb-lens.put-eq*)
 apply (*metis vwb-lens-wb wb-lens.get-put wb-lens-weak weak-lens.put-get*)
 done

lemma *RID-assign-r-same*:

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

lemma *RID-seq-left*:

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

lemma *RID-seq-right*:

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

also have ... = ($RID(x)(P) ;; RID(x)(Q)$)
 by *rel-tac*
 finally show *?thesis* .
 qed

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

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

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

lemma *assigns-r-runrest*:
 $\llbracket uvar\ x; x \# \sigma \rrbracket \Longrightarrow x \#\# \langle \sigma \rangle_a$
 by (*simp add: RID-assigns-r-diff unrest-relation-def*)

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

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

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

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

8.8 Alphabet laws

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

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

8.9 Relation algebra laws

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

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

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

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

theorem RA5: $(P \vee Q)^- = (P^- \vee Q^-)$
 by *rel-tac*

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

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

8.10 Relational alphabet extension

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

lemma *rel-alpha-ext-alt-def*:

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

done

8.11 Program values

abbreviation *prog-val* :: $'\alpha \text{ hrelation} \Rightarrow (' \alpha \text{ hrelation}, ' \alpha) \text{ uexpr} (\llbracket - \rrbracket_u)$
where $\llbracket P \rrbracket_u \equiv \langle P \rangle$

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

lemma *call-prog-val*: $call\ \llbracket P \rrbracket_u = P$
 by (*simp add: call-def urel-defs lit.rep-eq Rep-uexpr-inverse*)

end

8.12 Relational Hoare calculus

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

named-theorems *hoare*

definition *hoare-r* :: $'\alpha \text{ condition} \Rightarrow ' \alpha \text{ hrelation} \Rightarrow ' \alpha \text{ condition} \Rightarrow \text{bool} (\llbracket - \rrbracket - \llbracket - \rrbracket_u)$ **where**
 $\llbracket p \rrbracket Q \llbracket r \rrbracket_u = ((\llbracket p \rrbracket < \Rightarrow \llbracket r \rrbracket >) \sqsubseteq Q)$

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

lemma *hoare-r-conj* [*hoare*]: $\llbracket \llbracket p \rrbracket Q \llbracket r \rrbracket_u; \llbracket p \rrbracket Q \llbracket s \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket Q \llbracket r \wedge s \rrbracket_u$
 by *rel-tac*

lemma *hoare-r-conseq* [*hoare*]: $\llbracket 'p_1 \Rightarrow p_2'; \llbracket p_2 \rrbracket S \llbracket q_2 \rrbracket_u; 'q_2 \Rightarrow q_1' \rrbracket \Longrightarrow \llbracket p_1 \rrbracket S \llbracket q_1 \rrbracket_u$
 by *rel-tac*

lemma *assigns-hoare-r* [*hoare*]: $'p \Rightarrow \sigma \dagger q' \Longrightarrow \llbracket p \rrbracket \langle \sigma \rangle_a \llbracket q \rrbracket_u$

by *rel-tac*

lemma *skip-hoare-r* [*hoare*]: $\llbracket p \rrbracket II \llbracket p \rrbracket_u$
by *rel-tac*

lemma *seq-hoare-r* [*hoare*]: $\llbracket \llbracket p \rrbracket Q_1 \llbracket s \rrbracket_u ; \llbracket s \rrbracket Q_2 \llbracket r \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket Q_1 ;; Q_2 \llbracket r \rrbracket_u$
by *rel-tac*

lemma *cond-hoare-r* [*hoare*]: $\llbracket \llbracket b \wedge p \rrbracket S \llbracket q \rrbracket_u ; \llbracket \neg b \wedge p \rrbracket T \llbracket q \rrbracket_u \rrbracket \Longrightarrow \llbracket p \rrbracket S \triangleleft b \triangleright_r T \llbracket q \rrbracket_u$
by *rel-tac*

lemma *while-hoare-r* [*hoare*]:
assumes $\llbracket p \wedge b \rrbracket S \llbracket p \rrbracket_u$
shows $\llbracket p \rrbracket \text{while } b \text{ do } S \text{ od } \llbracket \neg b \wedge p \rrbracket_u$

proof –

from *assms* have $(\llbracket p \rrbracket_{<} \Rightarrow \llbracket p \rrbracket_{>}) \sqsubseteq (II \sqcap ((\llbracket b \rrbracket_{<} \wedge S) ;; (\llbracket p \rrbracket_{<} \Rightarrow \llbracket p \rrbracket_{>})))$
by (*simp add: hoare-r-def*) (*rel-tac*)

hence $p: (\llbracket p \rrbracket_{<} \Rightarrow \llbracket p \rrbracket_{>}) \sqsubseteq (\llbracket b \rrbracket_{<} \wedge S)^*_u$

by (*rule upred-quantale.star-inductl-one*[*rule-format*])

have $(\neg \llbracket b \rrbracket_{>} \wedge \llbracket p \rrbracket_{>}) \sqsubseteq ((\llbracket p \rrbracket_{<} \wedge (\llbracket p \rrbracket_{<} \Rightarrow \llbracket p \rrbracket_{>})) \wedge (\neg \llbracket b \rrbracket_{>}))$
by (*rel-tac*)

with p have $(\neg \llbracket b \rrbracket_{>} \wedge \llbracket p \rrbracket_{>}) \sqsubseteq ((\llbracket p \rrbracket_{<} \wedge (\llbracket b \rrbracket_{<} \wedge S)^*_u) \wedge (\neg \llbracket b \rrbracket_{>}))$
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 $\llbracket p \wedge b \rrbracket S \llbracket p \rrbracket_u$ ‘*pre* $\Rightarrow p$ ’ ‘ $(\neg b \wedge p) \Rightarrow \text{post}$ ’
shows $\llbracket \text{pre} \rrbracket \text{while } b \text{ invr } p \text{ do } S \text{ od } \llbracket \text{post} \rrbracket_u$
by (*metis assms hoare-r-conseq while-hoare-r while-inv-def*)

end

8.13 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* :: (‘ α , ‘ β) *relation* \Rightarrow ‘ β *condition* \Rightarrow ‘ α *condition* **where**
wp-upred *Q r* = $\lfloor \neg (Q ;; \neg \llbracket r \rrbracket_{<}) :: (\alpha, \beta) \text{ relation} \rfloor_{<}$

adhoc-overloading

uwp wp-upred

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

theorem *wp-assigns-r* [wp]:

$\langle \sigma \rangle_a \text{ wp } r = \sigma \uparrow r$

by *rel-tac*

theorem *wp-skip-r* [wp]:

$II \text{ wp } r = r$

by *rel-tac*

theorem *wp-true* [wp]:

$r \neq \text{true} \implies \text{true wp } r = \text{false}$

by *rel-tac*

theorem *wp-conj* [wp]:

$P \text{ wp } (q \wedge r) = (P \text{ wp } q \wedge P \text{ wp } r)$

by *rel-tac*

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

by *rel-tac*

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

by *rel-tac*

theorem *wp-hoare-link*:

$\llbracket p \rrbracket Q \llbracket r \rrbracket_u \longleftrightarrow (Q \text{ wp } r \sqsubseteq p)$

by *rel-tac*

end

9 Relational operational semantics

theory *utp-rel-opsem*

imports *utp-rel*

begin

fun *trel* :: $'\alpha \text{ usubst} \times '\alpha \text{ hrelation} \Rightarrow '\alpha \text{ usubst} \times '\alpha \text{ hrelation} \Rightarrow \text{bool}$ (**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*:

$\llbracket (\sigma, P) \rightarrow_u (\varrho, Q); (\varrho, Q) \rightarrow_u (\varphi, R) \rrbracket \implies (\sigma, P) \rightarrow_u (\varphi, R)$

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

assumes *uvar x*

shows $(\sigma, x := v) \rightarrow_u (\sigma(x \mapsto_s \sigma \uparrow v), II)$

by (*simp add: assigns-comp subst-upd-comp*)

lemma *seq-trel*:

```

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

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

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

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

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

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

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

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

declare trel.simps [simp del]

end

```

10 UTP Theories

```

theory utp-theory
imports utp-rel
begin

```

```

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

```

definition

```

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

```

```

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

```

unfolding *Healthy-def* **by** *auto*

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

abbreviation *Healthy-carrier* :: ' α *Healthiness-condition* \Rightarrow ' α *upred set* ($\llbracket - \rrbracket$)

where $\llbracket H \rrbracket \equiv \{P. P \text{ is } H\}$

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) $\longleftrightarrow \text{Idempotent}(H) \wedge \text{Monotonic}(H)$

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

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

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

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

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

lemma *Conjunctive-Idempotent*:
Conjunctive(H) $\implies \text{Idempotent}(H)$
by (*auto simp add: Conjunctive-def Idempotent-def*)

lemma *Conjunctive-Monotonic*:
Conjunctive(H) $\implies \text{Monotonic}(H)$
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*) $\longleftrightarrow (\exists F. \forall P. H(P) = (P \wedge F(P)) \wedge \text{Monotonic}(F))$

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 *WeakConjunctive-Healthy-Refinement*:

assumes *WeakConjunctive*(*HC*) **and** *P* *is* *HC*

shows $HC(P) \sqsubseteq P$

using *assms* **unfolding** *WeakConjunctive-def Healthy-def* **by** *simp*

lemma *WeakConjunctive-implies-WeakConjunctive*:

Conjunctive(*H*) \Longrightarrow *WeakConjunctive*(*H*)

unfolding *WeakConjunctive-def Conjunctive-def* **by** *pred-tac*

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

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

10.1 UTP theory hierarchy

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

consts

utp-hcond :: $('T \times '\alpha)$ *itself* \Rightarrow $(' \alpha \times ' \alpha)$ *Healthiness-condition* (\mathcal{H}_1)

utp-unit :: $('T \times '\alpha)$ *itself* \Rightarrow $' \alpha$ *hrelation* (\mathcal{II}_1)

definition *utp-order* :: $('T \times '\alpha)$ *itself* \Rightarrow $' \alpha$ *hrelation* *gorder* **where**

utp-order *T* = (\mid *carrier* = $\{P. P \text{ is } \mathcal{H}_T\}$, *eq* = (*op* =), *le* = *op* \sqsubseteq \mid)

locale *utp-theory* =

fixes *T* :: $('T \times '\alpha)$ *itself* (**structure**)

assumes *HCond-Idem*: $\mathcal{H}(\mathcal{H}(P)) = \mathcal{H}(P)$

begin

sublocale *partial-order* *utp-order* *T*

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

end

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

(structure)

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

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

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

sublocale *utp-theory-unital* \subseteq *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* \subseteq *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* \equiv *TYPE*(*REL* \times $'\alpha$)

overloading

rel-hcond == *utp-hcond* :: (*REL* \times $'\alpha$) *itself* \Rightarrow ($'\alpha \times '\alpha$) *Healthiness-condition*

rel-unit == *utp-unit* :: (*REL* \times $'\alpha$) *itself* \Rightarrow $'\alpha$ *hrelation*

begin

definition *rel-hcond* :: (*REL* \times $'\alpha$) *itself* \Rightarrow ($'\alpha \times '\alpha$) *upred* \Rightarrow ($'\alpha \times '\alpha$) *upred* **where**
 rel-hcond *T* = *id*

definition *rel-unit* :: (*REL* \times $'\alpha$) *itself* \Rightarrow $'\alpha$ *hrelation* **where**
 rel-unit *T* = *II*

end

interpretation *rel-theory*: *utp-theory-unital REL*

by (*unfold-locales, simp-all add: rel-hcond-def rel-unit-def Healthy-def*)

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* \implies *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 that 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
```

For now we have to explicitly cast the fields to lenses using the VAR syntactic transformation function – in 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  $\implies$  alpha-boyle where k = VAR boyle-k
definition p :: real  $\implies$  alpha-boyle where p = VAR boyle-p
definition V :: real  $\implies$  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 . The syntax differs a little from UTP; we try not to override HOL constants and so UTP predicate equality is subscripted. Moreover to distinguish variables denoting a predicate (like ϕ) from variables denoting UTP variables we have to prepend the latter with an ampersand.

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

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. 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-tac
```

```
lemma B-monotone:  $X \sqsubseteq Y \implies B(X) \sqsubseteq B(Y)$ 
by pred-tac
```

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

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

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

We first prove an obvious property: that these two predicates are different observations.

lemma $\varphi_1\text{-diff-}\varphi_2$: $\varphi_1 \neq \varphi_2$

by (*pred-tac*, *metis select-convs num.distinct(5) numeral-eq-iff semiring-norm(87)*)

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

lemma $B\text{-}\varphi_1$: φ_1 is B

by (*pred-tac*)

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

proof –

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

by (*simp add: \varphi_2-def*)

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

by (*simp add: B-def*)

also have $\dots = (\&p =_u 10 \wedge \&V =_u 5 \wedge \&k =_u \&p \cdot \&V)$

by *pred-tac*

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

by *pred-tac*

also have $\dots = \varphi_1$

by (*simp add: \varphi_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 $D1(P) = ((\&k =_u \&p \cdot \&V \Rightarrow \&k' =_u \&p' \cdot \&V') \wedge P)$

definition $D2(P) = (\&k' =_u \&k \wedge P)$

$D1$ 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 $D1\text{-idempotent}$: $D1(D1(P)) = D1(P)$ **by** *rel-tac*

lemma $D2\text{-idempotent}$: $D2(D2(P)) = D2(P)$ **by** *rel-tac*

lemma $D1\text{-monotone}$: $X \sqsubseteq Y \Longrightarrow D1(X) \sqsubseteq D1(Y)$ **by** *rel-tac*

lemma $D2\text{-monotone}$: $X \sqsubseteq Y \Longrightarrow D2(X) \sqsubseteq D2(Y)$ **by** *rel-tac*

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-tac*

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 the both *ChPres* and *ChVol* are both healthy relations.

lemma *D1*: $D1 (ChPres\ dp) = ChPres\ dp$ and $D1 (ChVol\ dV) = ChVol\ dV$

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

lemma *D2*: $D2 (ChPres\ dp) = ChPres\ dp$ and $D2 (ChVol\ dV) = ChVol\ dV$

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

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-tac*)

— This assignment becomes a substitution

hence $(InitSys\ 10\ 4 ;; ChPres\ (-2))$

$$= (ChPres\ (-2)) \llbracket 10, 4, 40 / \$p, \$V, \$k \rrbracket$$

by (*simp add: assigns-r-comp alpha*)

— Unfold definition of *ChPres*

also have $\dots = ((\&p - 2 >_u 0)^\top \llbracket 10, 4, 40 / \$p, \$V, \$k \rrbracket$

$$;; p := \&p - 2 ;; V := \&k / \&p)$$

by (*simp add: ChPres-def lit-num-simps usubst unrest*)

— Unfold definition of assumption

also have $\dots = ((p, V, k := 10, 4, 40 \triangleleft (8 :_u real) >_u 0 \triangleright false)$

$$;; p := \&p - 2 ;; V := \&k / \&p)$$

by (*simp add: rassume-def usubst alpha unrest*)

— $(0 :: 'a) < (8 :: 'a)$ is true; simplify conditional

also have $\dots = (p, V, k := 10, 4, 40 ;; p := \&p - 2 ;; V := \&k / \&p)$

by *rel-tac*

— Application of both assignments

also have $\dots = p, V, k := 8, 5, 40$

by *rel-tac*

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 = des-ok::bool

```

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

```

definition ok = VAR des-ok

```

```

declare ok-def [upred-defs]

```

```

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

```

```

lemma ok-ord [usubst]:
  $ok <_v $ok'
  by (simp add: var-name-ord-def)

```

```

type-synonym 'α alphabet-d = 'α alpha-d-scheme alphabet
type-synonym ('a, 'α) uvar-d = ('a, 'α alphabet-d) uvar
type-synonym ('α, 'β) relation-d = ('α alphabet-d, 'β alphabet-d) relation
type-synonym 'α hrelation-d = 'α alphabet-d hrelation

```

```

definition des-lens :: ('α, 'α alphabet-d) lens (ΣD) where
  des-lens = (| lens-get = more, lens-put = fld-put more-update |)

```

```

syntax
  -svid-alpha-d :: svid (ΣD)

```

```

translations
  -svid-alpha-d => ΣD

```

```

declare des-lens-def [upred-defs]

```

```

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

```

```

lemma ok-indep-des-lens [simp]: ok ⋈ des-lens des-lens ⋈ 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 $\lceil P \rceil_D \equiv P \oplus_p (\text{des-lens} \times_L \text{des-lens})$

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

where $\lfloor P \rfloor_D \equiv P \upharpoonright_p (\text{des-lens} \times_L \text{des-lens})$

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

definition *assigns-d* :: $'\alpha$ *usubst* \Rightarrow $'\alpha$ *hrelation-d* ($\langle _ \rangle_D$)

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

syntax

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

translations

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

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*) $\equiv ((P;;\text{true}) \Rightarrow P)$

syntax

-ok-f :: *logic* \Rightarrow *logic* ($\overset{f}{-}$ [1000] 1000)

-ok-t :: *logic* \Rightarrow *logic* ($\overset{t}{-}$ [1000] 1000)

-top-d :: *logic* (\top_D)

-bot-d :: *logic* (\perp_D)

translations

$P^f \Rightarrow \text{CONST usubst } (\text{CONST subst-upd } \text{CONST id } (\text{CONST ovar } \text{CONST ok}) \text{ false}) P$
 $P^t \Rightarrow \text{CONST usubst } (\text{CONST subst-upd } \text{CONST id } (\text{CONST ovar } \text{CONST ok}) \text{ true}) P$
 $\top_D \Rightarrow \text{CONST not-upred } (\text{CONST var } (\text{CONST ivar } \text{CONST ok}))$
 $\perp_D \Rightarrow \text{true}$

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

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

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

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*]: $\lfloor \lfloor P \rfloor_D \rfloor_D = P$
by (*simp add: arestr-aext prod-mwb-lens*)

lemma *lift-desr-inv*:

fixes $P :: (\alpha, \beta) \text{ relation-d}$
assumes $\$ok \# P \$ok' \# P$
shows $\lfloor \lfloor P \rfloor_D \rfloor_D = P$

proof –

have *bij-lens* ($\text{des-lens} \times_L \text{des-lens} +_L (\text{in-var ok} +_L \text{out-var ok}) :: (-, \alpha \text{ alpha-d-scheme} \times \beta \text{ alpha-d-scheme}) \text{ lens}$)
(is *bij-lens* ($?P$))

proof –

have $?P \approx_L (\text{ok} +_L \text{des-lens}) \times_L (\text{ok} +_L \text{des-lens})$ (**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]:
   $a \bowtie x \implies a \times_L b \bowtie \text{in-var } x$ 
  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]:
   $b \bowtie x \implies a \times_L b \bowtie \text{out-var } x$ 
  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]:  $\text{out}\alpha \# p \implies \text{out}\alpha \# [p]_D$ 
  by (pred-tac, auto simp add: out $\alpha$ -def des-lens-def prod-lens-def)

```

```

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

```

```

lemma lift-des-skip-dr-unit-unrest:  $\$ok' \# P \implies (P ;; [II]_D) = P$ 
  by (rel-tac, metis alpha-d.surjective alpha-d.update-convs(1))

```

```

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

```

```

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

```

```

lemma design-false-pre:
   $(\text{false} \vdash P) = \text{true}$ 
  by rel-tac

```

```

lemma rdesign-false-pre:
   $(\text{false} \vdash_r P) = \text{true}$ 
  by rel-tac

```

```

lemma ndesign-false-pre:
   $(\text{false} \vdash_n P) = \text{true}$ 
  by rel-tac

```

```

theorem design-refinement:
  assumes
     $\$ok \# P1 \ \$ok' \# P1 \ \$ok \# P2 \ \$ok' \# P2$ 
     $\$ok \# Q1 \ \$ok' \# Q1 \ \$ok \# Q2 \ \$ok' \# Q2$ 
  shows  $(P1 \vdash Q1 \sqsubseteq P2 \vdash Q2) \longleftrightarrow ('P1 \Rightarrow P2' \wedge 'P1 \wedge Q2 \Rightarrow Q1')$ 
proof -
  have  $(P1 \vdash Q1) \sqsubseteq (P2 \vdash Q2) \longleftrightarrow '(\$ok \wedge P2 \Rightarrow \$ok' \wedge Q2) \Rightarrow (\$ok \wedge P1 \Rightarrow \$ok' \wedge Q1)'$ 

```

```

  by pred-tac
also with assms have ... = '(P2 ⇒ $ok' ∧ Q2) ⇒ (P1 ⇒ $ok' ∧ Q1)'
  by (subst subst-bool-split[of in-var ok], simp-all, subst-tac)
also with assms have ... = '(¬ P2 ⇒ ¬ P1) ∧ ((P2 ⇒ Q2) ⇒ P1 ⇒ Q1)'
  by (subst subst-bool-split[of out-var ok], simp-all, subst-tac)
also have ... ⟷ '(P1 ⇒ P2)' ∧ 'P1 ∧ Q2 ⇒ Q1'
  by (pred-tac)
finally show ?thesis .
qed

```

theorem *rdesign-refinement*:

```

(P1 ⊢r Q1 ⊆ P2 ⊢r Q2) ⟷ ('P1 ⇒ P2' ∧ 'P1 ∧ Q2 ⇒ Q1')
apply (simp add: rdesign-def)
apply (subst design-refinement)
apply (simp-all add: unrest)
apply (pred-tac)
apply (metis alpha-d.select-convs(2))+
done

```

lemma *design-refine-intro*:

```

assumes 'P1 ⇒ P2' 'P1 ∧ Q2 ⇒ Q1'
shows P1 ⊢ Q1 ⊆ P2 ⊢ Q2
using assms unfolding upred-defs
by pred-tac

```

lemma *rdesign-refine-intro*:

```

assumes 'P1 ⇒ P2' 'P1 ∧ Q2 ⇒ Q1'
shows P1 ⊢r Q1 ⊆ P2 ⊢r Q2
using assms unfolding upred-defs
by pred-tac

```

lemma *ndesign-refine-intro*:

```

assumes 'p1 ⇒ p2' '[p1]_< ∧ Q2 ⇒ Q1'
shows p1 ⊢n Q1 ⊆ p2 ⊢n Q2
using assms unfolding upred-defs
by pred-tac

```

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

theorem *design-npre*:

```

(P ⊢ Q)f = (¬ $ok ∨ ¬ Pf)
by (rel-tac)

```

theorem *design-pre*:

```

¬ (P ⊢ Q)f = ($ok ∧ Pf)
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 ⊢ Q)t = (($ok ∧ Pt) ⇒ Qt)
by (rel-tac)

```

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

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

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

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

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

proof –

have $(\text{true} ;; (P \vdash Q)) = (\exists \text{ ok}_0 \cdot \text{true}[\llbracket \text{ok}_0 \rrbracket / \$\text{ok}' \rrbracket ;; (P \vdash Q)[\llbracket \text{ok}_0 \rrbracket / \$\text{ok} \rrbracket])$
by (*subst segr-middle*[*of ok*], *simp-all*)
also have $\dots = ((\text{true}[\llbracket \text{false} / \$\text{ok}' \rrbracket ;; (P \vdash Q)[\llbracket \text{false} / \$\text{ok} \rrbracket]) \vee (\text{true}[\llbracket \text{true} / \$\text{ok}' \rrbracket ;; (P \vdash Q)[\llbracket \text{true} / \$\text{ok} \rrbracket]))$
by (*simp add: disj-comm false-alt-def true-alt-def*)
also have $\dots = ((\text{true}[\llbracket \text{false} / \$\text{ok}' \rrbracket ;; \text{true}_h \rrbracket) \vee (\text{true} ;; ((P \vdash Q)[\llbracket \text{true} / \$\text{ok} \rrbracket]))$
by (*subst-tac, rel-tac*)
also have $\dots = \text{true}$
by (*subst-tac, simp add: precond-right-unit unrest*)
finally show *?thesis* .

qed

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

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

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

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

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

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

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

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

lemma *design-UNIF*:

$(\bigsqcup i \in A \cdot P(i) \vdash Q(i)) = (\prod i \in A \cdot P(i)) \vdash (\bigsqcup i \in A \cdot P(i) \Rightarrow Q(i))$
 by *rel-tac*

theorem *design-composition-subst*:

assumes

$\$ok' \# P1 \ \$ok \# P2$

shows $((P1 \vdash Q1) ;; (P2 \vdash Q2)) =$

$((\neg (\neg P1) ;; true) \wedge \neg (Q1 \llbracket true/\$ok' \rrbracket ;; \neg P2)) \vdash (Q1 \llbracket true/\$ok' \rrbracket ;; Q2 \llbracket true/\$ok \rrbracket))$

proof –

have $((P1 \vdash Q1) ;; (P2 \vdash Q2)) = (\exists ok_0 \cdot ((P1 \vdash Q1) \llbracket \llbracket ok_0 \rrbracket / \$ok' \rrbracket ;; (P2 \vdash Q2) \llbracket \llbracket ok_0 \rrbracket / \$ok \rrbracket))$
 by (*rule seqr-middle, simp*)

also have ...

$= (((P1 \vdash Q1) \llbracket false/\$ok' \rrbracket ;; (P2 \vdash Q2) \llbracket false/\$ok \rrbracket) \vee ((P1 \vdash Q1) \llbracket true/\$ok' \rrbracket ;; (P2 \vdash Q2) \llbracket true/\$ok \rrbracket))$

by (*simp add: true-alt-def false-alt-def, pred-tac*)

also from *assms*

have ... $= (((\$ok \wedge P1 \Rightarrow Q1 \llbracket true/\$ok' \rrbracket) ;; (P2 \Rightarrow \$ok' \wedge Q2 \llbracket true/\$ok \rrbracket)) \vee ((\neg (\$ok \wedge P1)) ;; true))$

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

also have ... $= ((\neg \$ok ;; true_h) \vee (\neg P1 ;; true) \vee (Q1 \llbracket true/\$ok' \rrbracket ;; \neg P2) \vee (\$ok' \wedge (Q1 \llbracket true/\$ok' \rrbracket ;; Q2 \llbracket true/\$ok \rrbracket)))$

by (*rel-tac*)

also have ... $= (((\neg (\neg P1) ;; true) \wedge \neg (Q1 \llbracket true/\$ok' \rrbracket ;; \neg P2)) \vdash (Q1 \llbracket true/\$ok' \rrbracket ;; Q2 \llbracket true/\$ok \rrbracket))$

by (*simp add: precondition-right-unit design-def unrest, rel-tac*)

finally show *?thesis* .

qed

lemma *design-export-ok*:

$P \vdash Q = (P \vdash (\$ok \wedge Q))$

by (*rel-tac*)

lemma *design-export-ok'*:

$P \vdash Q = (P \vdash (\$ok' \wedge Q))$

by (*rel-tac*)

theorem *design-composition*:

assumes

$\$ok' \# P1 \ \$ok \# P2 \ \$ok' \# Q1 \ \$ok \# Q2$

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 $x \# P$

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-tac*)

ultimately show *?thesis*

by (*metis utp-pred.inf.assoc utp-pred.inf-left-commute*)

qed

theorem *design-composition-runrest*:

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

theorem *rdesign-composition:*

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

lemma *skip-d-alt-def:* $II_D = true \vdash II$

by (*rel-tac*)

theorem *design-skip-idem* [*simp*]:

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

theorem *design-composition-cond:*

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

theorem *rdesign-composition-cond:*

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

theorem *design-composition-wp:*

fixes $Q1 \ Q2 :: 'a \text{ hrelation-}d$
assumes
 $ok \# p1 \ ok \# p2$
 $\$ok \# Q1 \ \$ok' \# Q1 \ \$ok \# Q2 \ \$ok' \# Q2$
shows $((\llbracket p1 \rrbracket_{<} \vdash Q1) ;; (\llbracket p2 \rrbracket_{<} \vdash Q2)) = ((\llbracket p1 \wedge Q1 \text{ wp } p2 \rrbracket_{<} \vdash (Q1 ;; Q2))$
using *assms*

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

theorem *rdesign-composition-wp*:

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

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

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

theorem *ndesign-composition-wp*:

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

shows $((p1 \vdash_n Q1) ;; (p2 \vdash_n Q2)) = ((p1 \wedge Q1\ wp\ p2) \vdash_n (Q1 ;; Q2))$

by (simp add: ndesign-def rdesign-composition-wp)

theorem *rdesign-wp [wp]*:

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

by rel-tac

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 $Q1\ Q2 :: 'a\ hrelation$

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

apply (simp add: wp)

apply (subst rdesign-composition-wp)

apply (simp only: wp)

apply (rel-tac)

done

theorem *wpnd-seq-r [wp]*:

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

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'*:

$(P \llbracket true/\$ok \rrbracket \vdash Q \llbracket true, true/\$ok, \$ok' \rrbracket) = (P \vdash Q)$

proof –

have $(P \vdash Q) = ((\$ok \wedge P) \vdash (\$ok \wedge \$ok' \wedge Q))$

by (pred-tac)

also have $\dots = ((\$ok \wedge P \llbracket true/\$ok \rrbracket) \vdash (\$ok \wedge (\$ok' \wedge Q \llbracket true/\$ok' \rrbracket) \llbracket true/\$ok \rrbracket))$

by (metis conj-eq-out-var-subst conj-pos-var-subst upred-eq-true utp-pred.inf-commute uvar-ok)

also have $\dots = ((\$ok \wedge P \llbracket true/\$ok \rrbracket) \vdash (\$ok \wedge \$ok' \wedge Q \llbracket true, true/\$ok, \$ok' \rrbracket))$

by (simp add: usubst)

also have $\dots = (P \llbracket true/\$ok \rrbracket \vdash Q \llbracket true, true/\$ok, \$ok' \rrbracket)$

by (pred-tac)

finally show ?thesis ..

qed

lemma *design-subst-ok'*:

$(P \vdash Q \llbracket true/\$ok' \rrbracket) = (P \vdash Q)$

proof –

have $(P \vdash Q) = (P \vdash (\$ok' \wedge Q))$

by (pred-tac)

also have $\dots = (P \vdash (\$ok' \wedge Q \llbracket true/\$ok' \rrbracket))$

by (metis conj-eq-out-var-subst upred-eq-true utp-pred.inf-commute uvar-ok)

also have ... = $(P \vdash Q \llbracket \text{true}/\text{\$ok} \rrbracket)$
 by (*pred-tac*)
 finally show ?thesis ..
 qed

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

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

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

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

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

lemma *assign-d-right-comp*:
 $((P \vdash_r Q) ;; \langle f \rangle_D) = ((\neg (\neg P ;; \text{true})) \vdash_r (Q ;; \langle f \rangle_a))$
 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*]:
 $\text{pre}_D(P \sqcap Q) = (\text{pre}_D(P) \wedge \text{pre}_D(Q))$
 by (*rel-tac*)

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

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

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

12.4 H1: No observation is allowed before initiation

lemma *H1-idem*:

$H1 (H1 P) = H1(P)$
by *pred-tac*

lemma *H1-monotone*:

$P \sqsubseteq Q \implies H1(P) \sqsubseteq H1(Q)$
by *pred-tac*

lemma *H1-below-top*:

$H1(P) \sqsubseteq \top_D$
by *pred-tac*

lemma *H1-design-skip*:

$H1(II) = II_D$
by *rel-tac*

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

theorem *H1-algebraic-intro*:

assumes
 $(true_h ;; R) = true_h$
 $(II_D ;; R) = R$
shows *R is H1*

proof –

have $R = (II_D ;; R)$ **by** (*simp add: assms(2)*)
also have $\dots = (H1(II) ;; R)$
 by (*simp add: H1-design-skip*)
also have $\dots = (\$ok \Rightarrow II) ;; R$
 by (*simp add: H1-def*)
also have $\dots = ((\neg \$ok ;; R) \vee R)$
 by (*simp add: impl-alt-def seqr-or-distl*)
also have $\dots = (((\neg \$ok ;; true_h) ;; R) \vee R)$
 by (*simp add: precond-right-unit unrest*)
also have $\dots = ((\neg \$ok ;; true_h) \vee R)$
 by (*metis assms(1) seqr-assoc*)
also have $\dots = (\$ok \Rightarrow R)$
 by (*simp add: impl-alt-def precond-right-unit unrest*)
finally show *?thesis* **by** (*metis H1-def Healthy-def'*)

qed

lemma *nok-not-false*:

$(\neg \$ok) \neq false$
by (*pred-tac, metis alpha-d.select-convs(1)*)

theorem *H1-left-zero*:

assumes *P is H1*
shows $(true ;; P) = true$

proof –

from *assms* **have** $(true ;; P) = (true ;; (\$ok \Rightarrow P))$
 by (*simp add: H1-def Healthy-def'*)

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

using *seqr-or-distr* by *blast*
 also from *assms* have ... = ($\text{true} \vee (\text{true} ;; P)$)
 by (*simp add: nok-not-false precondition-left-zero unrest*)
 finally show *?thesis*
 by (*rel-tac*)
 qed

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

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

theorem *H1-nok-left-zero*:
 fixes $P :: 'a \text{ hrelation-}d$
 assumes P is *H1*
 shows $(\neg \$ok ;; P) = (\neg \$ok)$
proof –
 have $(\neg \$ok ;; P) = ((\neg \$ok ;; \text{true}_h) ;; P)$
 by (*simp add: precondition-right-unit unrest*)
 also have ... = $(\neg \$ok) ;; \text{true}_h$
 by (*metis H1-left-zero assms seqr-assoc*)
 also have ... = $\neg \$ok$
 by (*simp add: precondition-right-unit unrest*)
 finally show *?thesis* .
 qed

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

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

lemma *H1-choice-closed*:
 $\llbracket P \text{ is } H1; Q \text{ is } H1 \rrbracket \Longrightarrow P \sqcap Q \text{ is } H1$
 by (*simp add: H1-def Healthy-def' disj-upred-def impl-alt-def semilattice-sup-class.sup-left-commute*)

lemma *H1-inf-closed*:

$\llbracket P \text{ is } H1; Q \text{ is } H1 \rrbracket \implies P \sqcup Q \text{ is } H1$
by (*rel-tac*, *blast+*)

lemma *H1-USUP*:

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

lemma *H1-Sup*:

assumes $A \neq \{\} \forall P \in A. P \text{ is } H1$
shows $(\bigsqcup A) \text{ is } H1$

proof –

from *assms*(2) **have** $H1 \text{ ‘ } A = A$
by (*auto simp add: Healthy-def rev-image-eqI*)
with *H1-USUP*[*of A id, OF assms*(1)] **show** *?thesis*
by (*simp add: USUP-as-Sup-image Healthy-def*)

qed

lemma *H1-UINF*:

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

lemma *H1-Inf*:

assumes $\forall P \in A. P \text{ is } H1$
shows $(\bigsqcup A) \text{ is } H1$

proof –

from *assms* **have** $H1 \text{ ‘ } A = A$
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 $\dots = (P ;; ((\$ok \Rightarrow \$ok \wedge \$ok') \wedge \lceil II \rceil_D))$
by *rel-tac*
also have $\dots = ((P ;; (\neg \$ok \wedge \lceil II \rceil_D)) \vee (P ;; (\$ok \wedge (\lceil II \rceil_D \wedge \$ok'))))$
by *rel-tac*
also have $\dots = (P^f \vee (P^t \wedge \$ok'))$

proof –

have $(P ;; (\neg \$ok \wedge \lceil II \rceil_D)) = P^f$

proof –

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

by *rel-tac*

also have $\dots = (\exists \$ok' \cdot P \wedge \$ok' =_u \text{false})$

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

also have $\dots = P^f$

by (*metis C1 one-point out-var-uvar pr-var-def unrest-as-exists uvar-ok vwb-lens-mwb*)

finally show *?thesis* .

```

qed
moreover have (P ;; ($ok ∧ ([II]D ∧ $ok')) = (Pt ∧ $ok')
proof -
  have (P ;; ($ok ∧ ([II]D ∧ $ok')) = (P ;; ($ok ∧ II))
    by (rel-tac, metis alpha-d.equality)
  also have ... = (Pt ∧ $ok')
    by (rel-tac, metis (full-types) alpha-d.surjective alpha-d.update-convs(1))+
  finally show ?thesis .
qed
ultimately show ?thesis
  by simp
qed
finally show ?thesis .
qed

lemma H2-split:
  shows H2(P) = (Pf ∨ (Pt ∧ $ok'))
  by (simp add: H2-def J-split)

theorem H2-equivalence:
  P is H2 ⟷ 'Pf ⇒ Pt'
proof -
  have 'P ⇔ (P ;; J)' ⟷ 'P ⇔ (Pf ∨ (Pt ∧ $ok'))'
    by (simp add: J-split)
  also from assms have ... ⟷ '(P ⇔ Pf ∨ Pt ∧ $ok')f ∧ (P ⇔ Pf ∨ Pt ∧ $ok')t'
    by (simp add: subst-bool-split)
  also from assms have ... = '(Pf ⇔ Pf) ∧ (Pt ⇔ Pf ∨ Pt)'
    by subst-tac
  also have ... = 'Pt ⇔ (Pf ∨ Pt)'
    by pred-tac
  also have ... = '(Pf ⇒ Pt)'
    by pred-tac
  finally show ?thesis using assms
    by (metis H2-def Healthy-def' taut-iff-eq)
qed

lemma H2-equiv:
  P is H2 ⟷ Pt ⊆ Pf
  using H2-equivalence refBy-order by blast

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

lemma H2-rdesign:
  H2(P ⊢r Q) = P ⊢r Q
  by (simp add: H2-design unrest rdesign-def)

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

theorem H2-idem:

```

$H2(H2(P)) = H2(P)$
 by (metis H2-def J-idem seqr-assoc)

theorem *H2-not-okay*: $H2(\neg \$ok) = (\neg \$ok)$
proof –
 have $H2(\neg \$ok) = ((\neg \$ok)^f \vee ((\neg \$ok)^t \wedge \$ok'))$
 by (simp add: H2-split)
 also have $\dots = (\neg \$ok \vee (\neg \$ok) \wedge \$ok')$
 by (subst-tac)
 also have $\dots = (\neg \$ok)$
 by pred-tac
 finally show ?thesis .
qed

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

lemma *H2-choice-closed*:
 $\llbracket P \text{ is } H2; Q \text{ is } H2 \rrbracket \implies P \sqcap Q \text{ is } H2$
 by (metis H2-def Healthy-def' disj-upred-def seqr-or-distl)

lemma *H2-inf-closed*:
 assumes $P \text{ is } H2 \quad Q \text{ is } H2$
 shows $P \sqcup Q \text{ is } H2$
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 $H2(\dots) = \dots$
 by (simp add: H2-split usubst, pred-tac)
 ultimately show ?thesis
 by (simp add: Healthy-def)
qed

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

theorem *H1-H2-commute*:
 $H1(H2 P) = H2(H1 P)$
proof –
 have $H2(H1 P) = ((\$ok \Rightarrow P) ;; J)$
 by (simp add: H1-def H2-def)
 also from assms have $\dots = ((\neg \$ok \vee P) ;; J)$
 by rel-tac
 also have $\dots = ((\neg \$ok ;; J) \vee (P ;; J))$
 using seqr-or-distl by blast
 also have $\dots = ((H2(\neg \$ok)) \vee H2(P))$
 by (simp add: H2-def)
 also have $\dots = ((\neg \$ok) \vee H2(P))$
 by (simp add: H2-not-okay)
 also have $\dots = H1(H2(P))$
 by rel-tac
 finally show ?thesis by simp
qed

lemma *ok-pre*: $(\$ok \wedge \lceil pre_D(P) \rceil_D) = (\$ok \wedge (\neg P^f))$
by (*pred-tac*)
 $(metis\ (mono-tags,\ lifting)\ \alpha-d.surjective\ \alpha-d.update-convs(1)\ \alpha-d.update-convs(2))+$

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

theorem *H1-H2-is-design*:

assumes *P is H1 P is H2*

shows $P = (\neg P^f) \vdash P^t$

proof –

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

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

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

by (*metis H2-split*)

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

by *pred-tac*

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

by *pred-tac*

also have $\dots = (\neg P^f) \vdash P^t$

by *pred-tac*

finally show *?thesis* .

qed

lemma *H1-H2-eq-design*:

H1 (H2 P) = (\neg P^f) \vdash P^t

apply (*subst H1-H2-is-design*)

apply (*simp-all add: Healthy-def H1-idem H2-idem H1-H2-commute*)

apply (*simp add: H2-split H1-def usubst*)

apply (*rel-tac*)

done

theorem *H1-H2-is-rdesign*:

assumes *P is H1 P is H2*

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

proof –

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

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

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

by (*metis H2-split*)

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

by *pred-tac*

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

by *pred-tac*

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

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

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

by *pred-tac*

also from *assms* **have** $\dots = pre_D(P) \vdash_r post_D(P)$

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

finally show *?thesis* .

qed

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

lemma *design-is-H1-H2*:

$\llbracket \$ok' \# P; \$ok' \# Q \rrbracket \implies (P \vdash Q) \text{ is } H1-H2$
by (*simp add: H1-design H2-design Healthy-def'*)

lemma *rdesign-is-H1-H2*:

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

lemma *seq-r-H1-H2-closed*:

assumes $P \text{ is } H1-H2\ Q \text{ is } H1-H2$
shows $(P ;; Q) \text{ is } H1-H2$

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

qed

lemma *assigns-d-comp-ext*:

fixes $P :: 'a\ hrelation-d$
assumes $P \text{ is } H1-H2$
shows $(\langle \sigma \rangle_D ;; P) = [\sigma \oplus_s \Sigma_D]_s \uparrow P$

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 $\dots = [\sigma]_s \uparrow pre_D(P) \vdash_r [\sigma]_s \uparrow post_D(P)$
by (*simp add: assign-d-left-comp*)
also have $\dots = [\sigma \oplus_s \Sigma_D]_s \uparrow (pre_D(P) \vdash_r post_D(P))$
by (*rel-tac*)
also have $\dots = [\sigma \oplus_s \Sigma_D]_s \uparrow P$
by (*metis H1-H2-commute H1-H2-is-rdesign H2-idem Healthy-def' assms*)
finally show *?thesis* .

qed

lemma *USUP-H1-H2-closed*:

assumes $A \neq \{\} \ \forall\ P \in A. P \text{ is } H1-H2$
shows $(\bigcap A) \text{ is } H1-H2$

proof –

from *assms* **have** $A: A = H1-H2\ 'A$
by (*auto simp add: Healthy-def rev-image-eqI*)
also have $(\bigcap \dots) = (\bigcap P \in A. H1-H2(P))$
by *auto*
also have $\dots = (\bigcap P \in A \cdot H1-H2(P))$
by (*simp add: USUP-as-Sup-collect*)
also have $\dots = (\bigcap P \in A \cdot (\neg P^f) \vdash P^t)$
by (*meson H1-H2-eq-design*)
also have $\dots = (\bigcup P \in A \cdot \neg P^f) \vdash (\bigcap P \in A \cdot P^t)$
by (*simp add: design-USUP assms*)
also have $\dots \text{ is } H1-H2$
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 (\sqcap_D - [900] 900) **where**
 $\sqcap_D A = (\text{if } (A = \{\}) \text{ then } \top_D \text{ else } \sqcap A)$

lemma *design-sup-H1-H2-closed*:
 assumes $\forall P \in A. P \text{ is } H1-H2$
 shows $(\sqcap_D A) \text{ is } H1-H2$
 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]: $\sqcap_D \{\} = \top_D$
 by (simp add: design-sup-def)

lemma *design-sup-non-empty* [simp]: $A \neq \{\} \Rightarrow \sqcap_D A = \sqcap A$
 by (simp add: design-sup-def)

lemma *UINF-H1-H2-closed*:
 assumes $\forall P \in A. P \text{ is } H1-H2$
 shows $(\sqcup A) \text{ is } H1-H2$
proof –
 from assms have $A: A = H1-H2 \text{ ' } A$
 by (auto simp add: Healthy-def rev-image-eqI)
 also have $(\sqcup \dots) = (\sqcup P \in A. H1-H2(P))$
 by auto
 also have $\dots = (\sqcup P \in A \cdot H1-H2(P))$
 by (simp add: UINF-as-Inf-collect)
 also have $\dots = (\sqcup P \in A \cdot (\neg P^f) \vdash P^t)$
 by (meson H1-H2-eq-design)
 also have $\dots = (\sqcap P \in A \cdot \neg P^f) \vdash (\sqcup P \in A \cdot \neg P^f \Rightarrow P^t)$
 by (simp add: design-UINF)
 also have $\dots \text{ is } H1-H2$
 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 (\sqcup_D - [900] 900) **where**
 $\sqcup_D A \equiv \sqcup A$

12.6 H3: The design assumption is a precondition

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

theorem *design-condition-is-H3*:
 assumes $\text{out}\alpha \nVdash p$
 shows $(p \vdash Q) \text{ is } H3$
proof –
 have $((p \vdash Q) ;; II_D) = (\neg(\neg p ;; \text{true})) \vdash (Q^t ;; II[\text{true}/\$ok])$
 by (simp add: skip-d-alt-def design-composition-subst unrest assms)
 also have $\dots = p \vdash (Q^t ;; II[\text{true}/\$ok])$
 using assms precond-equiv seqr-true-lemma by force

also have ... = $p \vdash Q$
 by (*rel-tac*, *metis* (*full-types*) *alpha-d.cases-scheme* *alpha-d.select-convs*(1) *alpha-d.update-convs*(1))
 finally show ?thesis
 by (*simp add: H3-def Healthy-def'*)
 qed

theorem *rdesign-H3-iff-pre*:

$P \vdash_r Q$ is $H3 \iff P = (P ;; true)$

proof –

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

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

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

by (*simp add: rdesign-composition*)

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

by *simp*

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

by *pred-tac*

finally have $P \vdash_r Q$ is $H3 \iff P \vdash_r Q = (\neg(\neg P ;; true)) \vdash_r Q$

by (*metis H3-def Healthy-def'*)

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

by (*metis rdesign-pre*)

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

by (*simp add: seqr-true-lemma*)

finally show ?thesis .

qed

theorem *design-H3-iff-pre*:

assumes $\$ok \# P \ \$ok' \# P \ \$ok \# Q \ \$ok' \# Q$

shows $P \vdash Q$ is $H3 \iff P = (P ;; true)$

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 $H3 \iff \lfloor P \rfloor_D = (\lfloor P \rfloor_D ;; true)$

using *rdesign-H3-iff-pre* by *blast*

ultimately show ?thesis

by (*metis assms drop-desr-inv lift-desr-inv lift-dist-seq aext-true*)

qed

theorem *H1-H3-commute*:

$H1 (H3 P) = H3 (H1 P)$

by *rel-tac*

lemma *skip-d-absorb-J-1*:

$(II_D ;; J) = II_D$

by (*metis H2-def H2-rdesign skip-d-def*)

lemma *skip-d-absorb-J-2*:

$(J ;; II_D) = II_D$

proof –

have $(J ;; II_D) = ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D ;; true \vdash II)$

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

also have ... = $(\exists ok_0 \cdot ((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D) \llbracket \llbracket ok_0 \rrbracket / \$ok' \rrbracket ;; (true \vdash II) \llbracket \llbracket ok_0 \rrbracket / \$ok \rrbracket$

by (*subst seqr-middle[of ok]*, *simp-all*)

also have ... = $((((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D) \llbracket false / \$ok' \rrbracket ;; (true \vdash II) \llbracket false / \$ok \rrbracket)$

$\vee (((\$ok \Rightarrow \$ok') \wedge \lceil II \rceil_D) \llbracket true / \$ok' \rrbracket ;; (true \vdash II) \llbracket true / \$ok \rrbracket)$

by (*simp add: disj-comm false-alt-def true-alt-def*)
 also have ... = $((\neg \$ok \wedge \lceil II \rceil_D ;; true) \vee (\lceil II \rceil_D ;; \$ok' \wedge \lceil II \rceil_D))$
 by *rel-tac*
 also have ... = II_D
 by *rel-tac*
 finally show ?thesis .
 qed

lemma *H2-H3-absorb*:
 $H2 (H3 P) = H3 P$
 by (*metis H2-def H3-def segr-assoc skip-d-absorb-J-1*)

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

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

theorem *H3-design-pre*:
 assumes $\$ok \nmid p \text{ out}\alpha \nmid p \ \$ok \nmid Q \ \$ok' \nmid Q$
 shows $H3(p \vdash Q) = p \vdash Q$
 using *assms*
 by (*metis Healthy-def' design-H3-iff-pre precondition-right-unit unrest-out α -var uvar-ok vwb-lens-mwb*)

theorem *H3-rdesign-pre*:
 assumes $\text{out}\alpha \nmid p$
 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-out α*)

theorem *H1-H3-is-design*:
 assumes $P \text{ is } H1 \ P \text{ is } H3$
 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 $P \text{ is } H1 \ P \text{ is } H3$
 shows $P = \text{pre}_D(P) \vdash_r \text{post}_D(P)$
 by (*metis H1-H2-is-rdesign H2-H3-absorb Healthy-def' assms*)

theorem *H1-H3-is-normal-design*:
 assumes $P \text{ is } H1 \ P \text{ is } H3$
 shows $P = \lfloor \text{pre}_D(P) \rfloor_{<} \vdash_n \text{post}_D(P)$
 by (*metis H1-H3-is-rdesign assms drop-pre-inv ndesign-def precondition-equiv rdesign-H3-iff-pre*)

abbreviation $H1-H3 \ p \equiv H1 (H3 \ p)$

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

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

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

lemma *H3-unrest-out-alpha-nok* [*unrest*]:

assumes *P is H1-H3*

shows $out\alpha \nmid P^f$

proof –

have $P = (\neg \ (P^f \ ; \ ; \ true)) \vdash \ P^t$

by (*metis H1-H3-eq-design Healthy-def assms*)

also have $out\alpha \nmid (...^f)$

by (*simp add: design-def usubst unrest, rel-tac*)

finally show *?thesis* .

qed

lemma *H3-unrest-out-alpha* [*unrest*]: $P \text{ is } H1-H3 \implies out\alpha \nmid pre_D(P)$

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 $P \ Q :: 'a \ hrelation-d$

assumes *P is H1-H3 Q is H1-H3*

shows $(P \ ; \ ; \ Q) \ wp_D \ r = P \ wp_D \ (Q \ wp_D \ r)$

by (*smt H1-H3-commute H1-H3-is-rdesign H1-idem Healthy-def' assms(1) assms(2) drop-pre-inv precond-equiv rdesign-H3-iff-pre wpd-seq-r*)

12.7 H4: Feasibility

theorem *H4-idem*:

$H4(H4(P)) = H4(P)$

by *pred-tac*

lemma *is-H4-alt-def*:

$P \text{ is } H4 \iff (P \ ; \ ; \ true) = true$

by (*rel-tac*)

lemma *H4-assigns-d*: $\langle \sigma \rangle_D \text{ is } H4$

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 $\dots = true$

by (*rel-tac*)

ultimately show *?thesis*

using *is-H4-alt-def* **by** *auto*

qed

12.8 UTP theories

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

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

abbreviation $DES \equiv \text{TYPE}(DES \times ' \alpha \text{ alphabet-d})$
abbreviation $NDES \equiv \text{TYPE}(NDES \times ' \alpha \text{ alphabet-d})$

overloading

$des\text{-}hcond == utp\text{-}hcond :: (DES \times ' \alpha \text{ alphabet-d}) \text{ itself} \Rightarrow (' \alpha \text{ alphabet-d} \times ' \alpha \text{ alphabet-d}) \text{ Healthiness-condition}$
 $des\text{-}unit == utp\text{-}unit :: (DES \times ' \alpha \text{ alphabet-d}) \text{ itself} \Rightarrow ' \alpha \text{ hrelation-d}$

$ndes\text{-}hcond == utp\text{-}hcond :: (NDES \times ' \alpha \text{ alphabet-d}) \text{ itself} \Rightarrow (' \alpha \text{ alphabet-d} \times ' \alpha \text{ alphabet-d}) \text{ Healthiness-condition}$
 $ndes\text{-}unit == utp\text{-}unit :: (NDES \times ' \alpha \text{ alphabet-d}) \text{ itself} \Rightarrow ' \alpha \text{ hrelation-d}$

begin

definition $des\text{-}hcond :: (DES \times ' \alpha \text{ alphabet-d}) \text{ itself} \Rightarrow (' \alpha \text{ alphabet-d} \times ' \alpha \text{ alphabet-d}) \text{ Healthiness-condition}$

where

$des\text{-}hcond \ t = H1\text{-}H2$

definition $des\text{-}unit :: (DES \times ' \alpha \text{ alphabet-d}) \text{ itself} \Rightarrow ' \alpha \text{ hrelation-d}$ **where**

$des\text{-}unit \ t = II_D$

definition $ndes\text{-}hcond :: (NDES \times ' \alpha \text{ alphabet-d}) \text{ itself} \Rightarrow (' \alpha \text{ alphabet-d} \times ' \alpha \text{ alphabet-d}) \text{ Healthiness-condition}$

where

$ndes\text{-}hcond \ t = H1\text{-}H3$

definition $ndes\text{-}unit :: (NDES \times ' \alpha \text{ alphabet-d}) \text{ itself} \Rightarrow ' \alpha \text{ hrelation-d}$ **where**

$ndes\text{-}unit \ t = II_D$

end

interpretation $des\text{-}utp\text{-}theory: utp\text{-}theory \ \text{TYPE}(DES \times ' \alpha \text{ alphabet-d})$

by ($simp \ add: H1\text{-}H2\text{-}commute \ H1\text{-}idem \ H2\text{-}idem \ des\text{-}hcond\text{-}def \ utp\text{-}theory\text{-}def$)

interpretation $ndes\text{-}utp\text{-}theory: utp\text{-}theory \ \text{TYPE}(NDES \times ' \alpha \text{ alphabet-d})$

by ($simp \ add: H1\text{-}H3\text{-}commute \ H1\text{-}idem \ H3\text{-}idem \ ndes\text{-}hcond\text{-}def \ utp\text{-}theory.intro$)

interpretation $des\text{-}left\text{-}unital: utp\text{-}theory\text{-}left\text{-}unital \ \text{TYPE}(DES \times ' \alpha \text{ alphabet-d})$

apply ($unfold\text{-}locales$)

apply ($simp\text{-}all \ add: des\text{-}hcond\text{-}def \ des\text{-}unit\text{-}def$)

apply ($simp \ add: rdesign\text{-}is\text{-}H1\text{-}H2 \ skip\text{-}d\text{-}def$)

apply ($metis \ H1\text{-}idem \ H1\text{-}left\text{-}unit \ Healthy\text{-}def'$)

done

interpretation $ndes\text{-}unital: utp\text{-}theory\text{-}unital \ \text{TYPE}(NDES \times (' \alpha \text{ alphabet-d}))$

apply ($unfold\text{-}locales, \ simp\text{-}all \ add: ndes\text{-}hcond\text{-}def \ ndes\text{-}unit\text{-}def$)

apply ($metis \ H1\text{-}rdesign \ H3\text{-}def \ Healthy\text{-}def' \ design\text{-}skip\text{-}idem \ skip\text{-}d\text{-}def$)

apply ($metis \ H1\text{-}idem \ H1\text{-}left\text{-}unit \ Healthy\text{-}def'$)

apply ($metis \ H1\text{-}H3\text{-}commute \ H3\text{-}def \ H3\text{-}idem \ Healthy\text{-}def'$)

done

interpretation $design\text{-}complete\text{-}lattice: utp\text{-}theory\text{-}lattice \ \text{TYPE}(DES \times ' \alpha \text{ alphabet-d})$

rewrites $carrier \ (utp\text{-}order \ DES) = \llbracket H1\text{-}H2 \rrbracket$

apply ($unfold\text{-}locales$)

apply ($simp\text{-}all \ add: des\text{-}hcond\text{-}def \ utp\text{-}order\text{-}def \ H1\text{-}idem \ H2\text{-}idem$)

apply ($rule\text{-}tac \ x = \sqcup_D \ A \ \text{in} \ exI$)

apply ($auto \ simp \ add: least\text{-}def \ Upper\text{-}def$)

```

using Inf-lower apply blast
apply (simp add: Ball-Collect UINF-H1-H2-closed)
apply (meson Ball-Collect Inf-greatest)
apply (rule-tac x= $\sqcap_D A$  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* :: $- \Rightarrow - (\mu_D)$ **where**

$\mu_D F \equiv \mu_{\text{utp-order } DES} F$

abbreviation *design-gfp* :: $- \Rightarrow - (\nu_D)$ **where**

$\nu_D F \equiv \nu_{\text{utp-order } DES} F$

end

13 Concurrent programming

theory *utp-concurrency*

imports *utp-designs*

begin

no-notation

Sublist.parallel (**infixl** \parallel 50)

13.1 Design parallel composition

definition *design-par* :: $(' \alpha, ' \beta) \text{ relation-}d \Rightarrow (' \alpha, ' \beta) \text{ relation-}d \Rightarrow (' \alpha, ' \beta) \text{ relation-}d$ (**infixr** \parallel 85)

where

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

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** $P = P_1 \vdash_r P_2$

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 $((P_1 \vdash_r P_2) ;; II_D) \parallel ((Q_1 \vdash_r Q_2) ;; II_D) = (((P_1 \vdash_r P_2) \parallel (Q_1 \vdash_r Q_2)) ;; II_D)$

by (*simp add: design-par-def skip-d-def rdesign-composition, rel-tac*)

ultimately show *?thesis*

by *simp*

qed

lemma *design-par-H3-closure*:

assumes P is $H1-H3$ Q is $H1-H3$

shows $(P \parallel Q)$ is $H3$

using *assms*

by (*simp add: H3-unrest-out-alpha design-par-def precondition-right-unit rdesign-H3-iff-pre seqr-pre-out*)

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

proof –

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

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

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

by *rel-tac*

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

by *rel-tac*

finally show *?thesis* .

qed

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

by *rel-tac*

lemma *parallel-comm*: $P \parallel Q = Q \parallel P$

by *pred-tac*

lemma *parallel-idem*:

assumes P is $H1$ P is $H2$

shows $P \parallel P = P$

by (*metis H1-H2-is-rdesign assms conj-idem design-par-def*)

lemma *parallel-mono-1*:

assumes $P_1 \sqsubseteq P_2$ P_1 is $H1-H2$ P_2 is $H1-H2$

shows $P_1 \parallel Q \sqsubseteq P_2 \parallel Q$

proof –

have $\text{pre}_D(P_1) \vdash_r \text{post}_D(P_1) \sqsubseteq \text{pre}_D(P_2) \vdash_r \text{post}_D(P_2)$

by (*metis H1-H2-commute H1-H2-is-rdesign H1-idem Healthy-def' assms*)

hence $(\text{pre}_D(P_1) \vdash_r \text{post}_D(P_1)) \parallel Q \sqsubseteq (\text{pre}_D(P_2) \vdash_r \text{post}_D(P_2)) \parallel Q$

by (*auto simp add: rdesign-refinement design-par-def*) (*pred-tac+*)

thus *?thesis*

by (*metis H1-H2-commute H1-H2-is-rdesign H1-idem Healthy-def' assms*)

qed

lemma *parallel-mono-2*:

assumes $Q_1 \sqsubseteq Q_2$ Q_1 is $H1-H2$ Q_2 is $H1-H2$

shows $P \parallel Q_1 \sqsubseteq P \parallel Q_2$

by (*metis assms parallel-comm parallel-mono-1*)

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*:

$(P \triangleleft [b]_D \triangleright Q) \parallel R = ((P \parallel R) \triangleleft [b]_D \triangleright (Q \parallel R))$

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 *i*th element of the second part

definition *ind-uvar* $i\ x = x ;_L list_lens\ i ;_L snd_L ;_L des_lens$

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

definition *in-ind-uvar* $i\ x = in_var\ (ind_uvar\ i\ x)$

definition *out-ind-uvar* $i\ x = out_var\ (ind_uvar\ i\ x)$

definition *in-pre-uvar* $x = in_var\ (pre_uvar\ x)$

definition *out-pre-uvar* $x = out_var\ (pre_uvar\ x)$

definition *in-ind-uexpr* $i\ x = var\ (in_ind_uvar\ i\ x)$

definition *out-ind-uexpr* $i\ x = var\ (out_ind_uvar\ i\ x)$

definition *in-pre-uexpr* $x = var\ (in_pre_uvar\ x)$

definition *out-pre-uexpr* $x = var\ (out_pre_uvar\ x)$

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

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

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

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

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

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

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

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

lemma *left-uvar-indep-right-uvar* [*simp*]:

left-uvar $x \bowtie$ *right-uvar* y

apply (*simp* *add*: *left-uvar-def* *right-uvar-def* *lens-comp-assoc* [*THEN sym*])

apply (*metis* *in-out-indep* *in-var-def* *lens-indep-left-comp* *out-var-def* *out-var-indep* *uvar-des-lens* *vwb-lens-mwb*)

done

lemma *right-uvar-indep-left-uvar* [*simp*]:

right-uvar $x \bowtie$ *left-uvar* y

by (*simp* *add*: *lens-indep-sym*)

```

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

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

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

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

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

lemma out-ind-uvar-semi-uvar [simp]:
  semi-uvar x  $\implies$  semi-uvar (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 \Σ))

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

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** \parallel - 85)

where $P \parallel_M Q = (((P ;; U0) \parallel (Q ;; U1))) ;; M$

definition $\text{swap}_m = \text{true} \vdash_r (0-\Sigma, 1-\Sigma := \&1-\Sigma, \&0-\Sigma)$

declare One-nat-def [*simp del*]

declare $\text{swap}_m\text{-def}$ [*upred-defs*]

lemma $U0\text{-}H1\text{-}H2$: $U0$ is $H1\text{-}H2$
by (*simp add: U0-def rdesign-is-H1-H2*)

lemma $U0\text{-swap}$: $(U0 ;; \text{swap}_m) = U1$
apply (*simp add: U0-def swap_m-def rdesign-composition*)
apply (*subst seqr-and-distl-uj*)
using *assigns-r-swap-uj id-vwb-lens left-uvar right-uvar* **apply** *fastforce*
apply (*rel-tac*)
apply (*metis prod.collapse*)
done

lemma $U1\text{-}H1\text{-}H2$: $U1$ is $H1\text{-}H2$
by (*simp add: U1-def rdesign-is-H1-H2*)

lemma $U1\text{-swap}$: $(U1 ;; \text{swap}_m) = U0$
apply (*simp add: U1-def swap_m-def rdesign-composition*)
apply (*subst seqr-and-distl-uj*)
using *assigns-r-swap-uj id-vwb-lens left-uvar right-uvar* **apply** *fastforce*
apply (*rel-tac*)
apply (*metis prod.collapse*)
done

lemma $\text{swap-merge-par-distl}$:
assumes P is $H1\text{-}H2$ Q is $H1\text{-}H2$
shows $((P \parallel Q) ;; \text{swap}_m) = (P ;; \text{swap}_m) \parallel (Q ;; \text{swap}_m)$

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 $((P_1 \vdash_r P_2) \parallel (Q_1 \vdash_r Q_2)) ;; \text{swap}_m =$
 $(\neg (\neg P_1 \vee \neg Q_1 ;; \text{true})) \vdash_r ((P_1 \Rightarrow P_2) \wedge (Q_1 \Rightarrow Q_2) ;; \langle [\&0-\Sigma \mapsto_s \&1-\Sigma, \&1-\Sigma \mapsto_s \&0-\Sigma] \rangle_a)$
by (*simp add: design-par-def swap_m-def rdesign-composition*)
also have $\dots = (\neg (\neg P_1 \vee \neg Q_1 ;; \text{true})) \vdash_r (((P_1 \Rightarrow P_2) ;; \langle [\&0-\Sigma \mapsto_s \&1-\Sigma, \&1-\Sigma \mapsto_s \&0-\Sigma] \rangle_a) \wedge ((Q_1 \Rightarrow Q_2) ;; \langle [\&0-\Sigma \mapsto_s \&1-\Sigma, \&1-\Sigma \mapsto_s \&0-\Sigma] \rangle_a))$
apply (*subst seqr-and-distl-uj*)
using *assigns-r-swap-uj id-vwb-lens left-uvar right-uvar* **apply** *fastforce*
apply (*simp*)
done

also have $\dots = ((P_1 \vdash_r P_2) ;; \text{swap}_m) \parallel ((Q_1 \vdash_r Q_2) ;; \text{swap}_m)$
by (*simp add: design-par-def swap_m-def rdesign-composition, rel-tac*)

finally show *?thesis*
using $P Q$ **by** *blast*
qed

lemma *par-by-merge-left-zero*:

assumes M is $H1$

shows $true \parallel_M P = true$

proof –

have $true \parallel_M P = ((true ;; U0) \parallel (P ;; U1) ;; M)$ (**is** $- = ((?P \parallel ?Q) ;; ?M)$)

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

moreover have $?P = true$

by (*rel-tac, meson alpha-d.select-convs(1)*)

ultimately show $?thesis$

by (*metis H1-left-zero assms parallel-comm parallel-zero*)

qed

lemma *par-by-merge-right-zero*:

assumes M is $H1$

shows $P \parallel_M true = true$

proof –

have $P \parallel_M true = ((P ;; U0) \parallel (true ;; U1) ;; M)$ (**is** $- = ((?P \parallel ?Q) ;; ?M)$)

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

moreover have $?Q = true$

by (*rel-tac, meson alpha-d.select-convs(1)*)

ultimately show $?thesis$

by (*metis H1-left-zero assms parallel-comm parallel-zero*)

qed

lemma *par-by-merge-commute*:

assumes P is $H1-H2$ Q is $H1-H2$ $M = (swap_m ;; M)$

shows $P \parallel_M Q = Q \parallel_M P$

proof –

have $P \parallel_M Q = (((P ;; U0) \parallel (Q ;; U1)) ;; M)$

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

also have $... = (((P ;; U0) \parallel (Q ;; U1)) ;; swap_m) ;; M)$

by (*metis assms(3) seqr-assoc*)

also have $... = (((P ;; U0 ;; swap_m) \parallel (Q ;; U1 ;; swap_m)) ;; M)$

by (*simp add: U0-def U1-def assms(1) assms(2) rdesign-is-H1-H2 seq-r-H1-H2-closed seqr-assoc swap-merge-par-distl*)

also have $... = (((P ;; U1) \parallel (Q ;; U0)) ;; M)$

by (*simp add: U0-swap U1-swap*)

also have $... = Q \parallel_M P$

by (*simp add: par-by-merge-def parallel-comm*)

finally show $?thesis$.

qed

lemma *par-by-merge-mono-1*:

assumes $P_1 \sqsubseteq P_2$ P_1 is $H1-H2$ P_2 is $H1-H2$

shows $P_1 \parallel_M Q \sqsubseteq P_2 \parallel_M Q$

using *assms*

by (*auto intro:seqr-mono parallel-mono-1 seq-r-H1-H2-closed U0-H1-H2 U1-H1-H2 simp add: par-by-merge-def*)

lemma *par-by-merge-mono-2*:

assumes $Q_1 \sqsubseteq Q_2$ Q_1 is $H1-H2$ Q_2 is $H1-H2$

shows $(P \parallel_M Q_1) \sqsubseteq (P \parallel_M Q_2)$

using *assms*

by (*auto intro:seqr-mono parallel-mono-2 seq-r-H1-H2-closed U0-H1-H2 U1-H1-H2 simp add: par-by-merge-def*)

end

14 Reactive processes

theory *utp-reactive*

imports

utp-concurrency

utp-event

begin

record *'t::ordered-cancel-monoid-diff alpha-rp'* =

rp-wait :: *bool*

rp-tr :: *'t*

type-synonym *('t, 'α) alpha-rp-scheme* = *('t, 'α) alpha-rp'-scheme alpha-d-scheme*

type-synonym *('t, 'α) alphabet-rp* = *('t, 'α) alpha-rp-scheme alphabet*

type-synonym *('t, 'α, 'β) relation-rp* = *((('t, 'α) alphabet-rp, ('t, 'β) alphabet-rp) relation*

type-synonym *('t, 'α) hrelation-rp* = *((('t, 'α) alphabet-rp, ('t, 'α) alphabet-rp) relation*

type-synonym *('t, 'σ) predicate-rp* = *('t, 'σ) alphabet-rp upred*

definition *wait_r* = *VAR rp-wait*

definition *tr_r* = *VAR rp-tr*

definition *[upred-defs]: Σ_r* = *VAR more*

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

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

declare *Σ_r-def* *[upred-defs]*

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

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

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

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

lemma *rea-uvar* *[simp]: uvar Σ_r*

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

definition *wait* = *(wait_r ;_L Σ_D)*

definition *tr* = *(tr_r ;_L Σ_D)*

definition *[upred-defs]: Σ_R* = *(Σ_r ;_L Σ_D)*

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

by *(simp add: comp-vwb-lens wait-def)*

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

by *(simp add: comp-vwb-lens tr-def)*

lemma *rea-lens-uvar* *[simp]: uvar Σ_R*

by *(simp add: Σ_R-def comp-vwb-lens)*

lemma *rea-lens-under-des-lens: Σ_R ⊆_L Σ_D*

by *(simp add: Σ_R-def lens-comp-lb)*

lemma *rea-lens-indep-ok* *[simp]: Σ_R ⋈ ok ok ⋈ Σ_R*

using *ok-indep-des-lens(2) rea-lens-under-des-lens sublens-pres-indep* **apply** *blast*

using *lens-indep-sym ok-indep-des-lens(2) rea-lens-under-des-lens sublens-pres-indep* **apply** *blast*

done

declare *wait-def* [*upred-defs*]
declare *tr-def* [*upred-defs*]

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*, ' α ', ' β ') *relation-rp* \Rightarrow (*t*, ' α ', ' β ') *relation-rp*
where *wait-f* *R* $\equiv R[\text{false}/\$wait]$

abbreviation *wait-t*::(*t*::*ordered-cancel-monoid-diff*, ' α ', ' β ') *relation-rp* \Rightarrow (*t*, ' α ', ' β ') *relation-rp*
where *wait-t* *R* $\equiv R[\text{true}/\$wait]$

syntax

-wait-f :: *logic* \Rightarrow *logic* (*-_f* [1000] 1000)
-wait-t :: *logic* \Rightarrow *logic* (*-_t* [1000] 1000)

translations

$P_f \equiv \text{CONST usubst (CONST subst-upd CONST id (CONST ivar CONST wait) false) } P$
 $P_t \equiv \text{CONST usubst (CONST subst-upd CONST id (CONST ivar CONST wait) true) } P$

abbreviation *lift-rea* :: - \Rightarrow - ($[-]_R$) **where**
 $[P]_R \equiv P \oplus_p (\Sigma_R \times_L \Sigma_R)$

abbreviation *drop-rea* :: (*t*::*ordered-cancel-monoid-diff*, ' α ', ' β ') *relation-rp* \Rightarrow (' α ', ' β ') *relation* ($[-]_R$)
where

$$\lfloor P \rfloor_R \equiv P \upharpoonright_p (\Sigma_R \times_L \Sigma_R)$$

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

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

14.1 Reactive lemmas

lemma *unrest-ok-lift-rea* [*unrest*]:

$$\$ok \# \lceil P \rceil_R \$ok' \# \lceil P \rceil_R$$

by (*pred-tac*)⁺

lemma *unrest-wait-lift-rea* [*unrest*]:

$$\$wait \# \lceil P \rceil_R \$wait' \# \lceil P \rceil_R$$

by (*pred-tac*)⁺

lemma *unrest-tr-lift-rea* [*unrest*]:

$$\$tr \# \lceil P \rceil_R \$tr' \# \lceil P \rceil_R$$

by (*pred-tac*)⁺

lemma *tr-prefix-as-concat*: $(xs \leq_u ys) = (\exists zs \cdot ys =_u xs \hat{\ }_u \ll zs \gg)$

by (*rel-tac*, *simp* *add*: *less-eq-list-def* *prefixeq-def*)

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-tac*

lemma *R1-mono*: $P \sqsubseteq Q \implies R1(P) \sqsubseteq R1(Q)$

by *pred-tac*

lemma *R1-unrest* [*unrest*]: $\llbracket x \bowtie \text{in-var } tr; x \bowtie \text{out-var } tr; x \# P \rrbracket \implies x \# R1(P)$

by (*metis* *R1-def* *in-var-uvar* *lens-indep-sym* *out-var-uvar* *tr-uvar* *unrest-bop* *unrest-conj* *unrest-var*)

lemma *R1-false*: $R1(\text{false}) = \text{false}$

by *pred-tac*

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

by *pred-tac*

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

by *pred-tac*

lemma *R1-USUP*:

$$R1(\bigcap i \in A \cdot P(i)) = (\bigcap i \in A \cdot R1(P(i)))$$

by (*rel-tac*)

lemma *R1-UINF*:

assumes $A \neq \{\}$

shows $R1(\bigcup i \in A \cdot P(i)) = (\bigcup i \in A \cdot R1(P(i)))$

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

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

by *pred-tac*

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

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

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

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

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

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

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

lemma *R1-by-refinement*:
 $P \text{ is } R1 \iff ((\$tr \leq_u \$tr') \sqsubseteq P)$
by *rel-tac*

lemma *tr-le-trans*:
 $(\$tr \leq_u \$tr' ;; \$tr \leq_u \$tr') = (\$tr \leq_u \$tr')$
by (*rel-tac*, *metis alpha-d.select-convs(2) alpha-rp'.select-convs(2) eq-refl*)

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

lemma *R1-seqr-closure*:
assumes $P \text{ is } R1\ Q \text{ is } R1$
shows $(P ;; Q) \text{ is } R1$
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-tac*, *metis alpha-d.select-convs(2) alpha-rp'.select-convs(2) order-refl*)

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

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

lemma *R1-ok-true*: $(R1(P))\llbracket true/\$ok \rrbracket = R1(P\llbracket true/\$ok \rrbracket)$
by *pred-tac*

lemma *R1-ok-false*: $(R1(P))\llbracket false/\$ok \rrbracket = R1(P\llbracket false/\$ok \rrbracket)$
by *pred-tac*

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

lemma *R1-extend-conj-unrest*: $\llbracket \$tr \# Q; \$tr' \# Q \rrbracket \implies R1(P \wedge Q) = (R1(P) \wedge Q)$
 by *pred-tac*

lemma *R1-extend-conj-unrest'*: $\llbracket \$tr \# P; \$tr' \# P \rrbracket \implies R1(P \wedge Q) = (P \wedge R1(Q))$
 by *pred-tac*

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

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

14.3 R2

definition *R2a-def* [*upred-defs*]: $R2a(P) = (\bigcap s \cdot P[\llbracket \llbracket s \rrbracket, \llbracket s \rrbracket + (\$tr' - \$tr) / \$tr, \$tr' \rrbracket])$

definition *R2s-def* [*upred-defs*]: $R2s(P) = (P[\llbracket 0 / \$tr \rrbracket][\llbracket (\$tr' - \$tr) / \$tr' \rrbracket])$

definition *R2-def* [*upred-defs*]: $R2(P) = R1(R2s(P))$

definition *R2c-def* [*upred-defs*]: $R2c(P) = (R2s(P) \triangleleft R1(true) \triangleright P)$

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

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

lemma *R2a-equiv-R2s*: $P \text{ is } R2a \longleftrightarrow P \text{ is } R2s$
 by (*metis Healthy-def' R2a-R2s R2s-R2a*)

lemma *R2s-idem*: $R2s(R2s(P)) = R2s(P)$
 by (*pred-tac*)

lemma *R2s-unrest* [*unrest*]: $\llbracket uvar\ x; x \bowtie in\text{-}var\ tr; x \bowtie out\text{-}var\ tr; x \# P \rrbracket \implies x \# R2s(P)$
 by (*simp add: R2s-def unrest usubst lens-indep-sym*)

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

lemma *R2-mono*: $P \sqsubseteq Q \implies R2(P) \sqsubseteq R2(Q)$
 by (*pred-tac*)

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

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

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

lemma *R2s-USUP*:
 $R2s(\bigcap i \in A \cdot P(i)) = (\bigcap i \in A \cdot R2s(P(i)))$
 by (*simp add: R2s-def usubst*)

lemma *R2s-UINF*:

$R2s(\bigsqcup i \in A \cdot P(i)) = (\bigsqcup i \in A \cdot R2s(P(i)))$
by (*simp add: R2s-def usubst*)

lemma *R2-disj*: $R2(P \vee Q) = (R2(P) \vee R2(Q))$
by (*pred-tac*)

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

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

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

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

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

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

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

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

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

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

lemma *true-is-R2s*:
true is R2s
by (*simp add: Healthy-def R2s-true*)

lemma *R2s-lift-rea*: $R2s(\lceil P \rceil_R) = \lceil P \rceil_R$
by (*simp add: R2s-def usubst unrest*)

lemma *R2c-true*: $R2c(true) = true$
by *rel-tac*

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

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

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

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

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

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

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

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

lemma *R2c-tr'-ge-tr*: $R2c(\$tr' \geq_u \$tr) = (\$tr' \geq_u \$tr)$
 by (*rel-tac*)

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

lemma *R2c-skip-r*: $R2c(II) = II$

proof –

have $R2c(II) = R2c(\$tr' =_u \$tr \wedge II \upharpoonright_{\alpha} tr)$
 by (*subst skip-r-unfold*[*of tr*], *simp-all*)
 also have $\dots = (R2c(\$tr' =_u \$tr) \wedge II \upharpoonright_{\alpha} tr)$
 by (*simp add: R2c-and*, *simp add: R2c-def R2s-def usubst unrest cond-idem*)
 also have $\dots = (\$tr' =_u \$tr \wedge II \upharpoonright_{\alpha} tr)$
 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-tac*)

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

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-tac*)

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*: $R2(\$tr \leq_u \$tr') = (\$tr \leq_u \$tr')$
 by (*pred-tac*)

lemma *R2-form*:
 $R2(P) = (\exists tt \cdot P[0/\$tr][\ll tt \gg / \$tr'] \wedge \$tr' =_u \$tr + \ll tt \gg)$
 apply (*rel-tac*)

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 $(R2(P) ;; R2(Q)) =$
 $(\exists tt_1 \cdot \exists tt_2 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr'] ;; (Q[0/\$tr][\langle tt_2 \rangle / \$tr'])))$
 $\wedge (\$tr' =_u \$tr + \langle tt_1 \rangle + \langle tt_2 \rangle))$

proof –

have $(R2(P) ;; R2(Q)) = (\exists tr_0 \cdot (R2(P))[\langle tr_0 \rangle / \$tr'] ;; (R2(Q))[\langle tr_0 \rangle / \$tr'])$
by (*subst seqr-middle[of tr], simp-all*)

also have ... =

$(\exists tr_0 \cdot \exists tt_1 \cdot \exists tt_2 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr'] \wedge \langle tr_0 \rangle =_u \$tr + \langle tt_1 \rangle) ;;$
 $(Q[0/\$tr][\langle tt_2 \rangle / \$tr'] \wedge \$tr' =_u \langle tr_0 \rangle + \langle tt_2 \rangle)))$

by (*simp add: R2-form usubst unrest uquant-lift, rel-tac*)

also have ... =

$(\exists tr_0 \cdot \exists tt_1 \cdot \exists tt_2 \cdot ((\langle tr_0 \rangle =_u \$tr + \langle tt_1 \rangle \wedge P[0/\$tr][\langle tt_1 \rangle / \$tr']) ;;$
 $(Q[0/\$tr][\langle tt_2 \rangle / \$tr'] \wedge \$tr' =_u \langle tr_0 \rangle + \langle tt_2 \rangle)))$

by (*simp add: conj-comm*)

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot \exists tr_0 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr'] ;; (Q[0/\$tr][\langle tt_2 \rangle / \$tr'])))$
 $\wedge \langle tr_0 \rangle =_u \$tr + \langle tt_1 \rangle \wedge \$tr' =_u \langle tr_0 \rangle + \langle tt_2 \rangle)$

by *rel-tac*

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr'] ;; (Q[0/\$tr][\langle tt_2 \rangle / \$tr'])))$
 $\wedge (\exists tr_0 \cdot \langle tr_0 \rangle =_u \$tr + \langle tt_1 \rangle \wedge \$tr' =_u \langle tr_0 \rangle + \langle tt_2 \rangle))$

by *rel-tac*

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr'] ;; (Q[0/\$tr][\langle tt_2 \rangle / \$tr'])))$
 $\wedge (\$tr' =_u \$tr + \langle tt_1 \rangle + \langle tt_2 \rangle))$

by *rel-tac*

finally show *?thesis* .

qed

lemma *R2-seqr-distribute*:

fixes $P :: ('t :: \text{ordered-cancel-monoid-diff}, 'a, 'b) \text{ relation-rp}$ **and** $Q :: ('t, 'b, 'c) \text{ relation-rp}$
shows $R2(R2(P) ;; R2(Q)) = (R2(P) ;; R2(Q))$

proof –

have $R2(R2(P) ;; R2(Q)) =$

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[0/\$tr][\langle tt_1 \rangle / \$tr'] ;; Q[0/\$tr][\langle tt_2 \rangle / \$tr'])(\$tr' - \$tr) / \$tr')$
 $\wedge \$tr' - \$tr =_u \langle tt_1 \rangle + \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr)$

by (*simp add: R2-seqr-form, simp add: R2s-def usubst unrest, rel-tac*)

also have ... =

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[0/\$tr][\langle tt_1 \rangle / \$tr'] ;; Q[0/\$tr][\langle tt_2 \rangle / \$tr'])(\langle tt_1 \rangle + \langle tt_2 \rangle) / \$tr')$
 $\wedge \$tr' - \$tr =_u \langle tt_1 \rangle + \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr)$

by (*subst subst-eq-replace, simp*)

also have ... =

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[0/\$tr][\langle tt_1 \rangle / \$tr'] ;; Q[0/\$tr][\langle tt_2 \rangle / \$tr'])))$
 $\wedge \$tr' - \$tr =_u \langle tt_1 \rangle + \langle tt_2 \rangle) \wedge \$tr' \geq_u \$tr)$

by (*rel-tac*)

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot (P[0/\$tr][\langle tt_1 \rangle / \$tr'] ;; Q[0/\$tr][\langle tt_2 \rangle / \$tr'])))$
 $\wedge (\$tr' - \$tr =_u \langle tt_1 \rangle + \langle tt_2 \rangle \wedge \$tr' \geq_u \$tr)$

by *pred-tac*

also have ... =

$((\exists \text{ tt}_1 \cdot \exists \text{ tt}_2 \cdot (P[\![0/\$tr]\!][\![\langle\text{tt}_1\rangle/\$tr'\!]\!] \;; Q[\![0/\$tr]\!][\![\langle\text{tt}_2\rangle/\$tr'\!]\!])$
 $\wedge \$tr' =_u \$tr + \langle\text{tt}_1\rangle + \langle\text{tt}_2\rangle))$

proof –

have $\bigwedge \text{ tt}_1 \text{ tt}_2. (((\$tr' - \$tr =_u \langle\text{tt}_1\rangle + \langle\text{tt}_2\rangle) \wedge \$tr' \geq_u \$tr) :: ('t, 'α, 'γ) \text{ relation-rp})$
 $= (\$tr' =_u \$tr + \langle\text{tt}_1\rangle + \langle\text{tt}_2\rangle)$

apply (*rel-tac*)

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 $\dots = (R2(P) \;; R2(Q))$

by (*simp add: R2-seqr-form*)

finally show *?thesis* .

qed

lemma *R2-seqr-closure*:

assumes *P is R2 Q is R2*

shows $(P \;; Q) \text{ is } R2$

by (*metis Healthy-def' R2-seqr-distribute assms(1) assms(2)*)

lemma *R1-R2-commute*:

$R1(R2(P)) = R2(R1(P))$

by *pred-tac*

lemma *R2-R1-form*: $R2(R1(P)) = R1(R2s(P))$

by (*rel-tac*)

lemma *R2s-H1-commute*:

$R2s(H1(P)) = H1(R2s(P))$

by *rel-tac*

lemma *R2s-H2-commute*:

$R2s(H2(P)) = H2(R2s(P))$

by (*simp add: H2-split R2s-def usubst*)

lemma *R2-R1-seq-drop-left*:

$R2(R1(P) \;; R1(Q)) = R2(P \;; R1(Q))$

by *rel-tac*

lemma *R2c-idem*: $R2c(R2c(P)) = R2c(P)$

by (*rel-tac*)

lemma *R2c-H2-commute*: $R2c(H2(P)) = H2(R2c(P))$

by (*simp add: H2-split R2c-disj R2c-def R2s-def usubst, rel-tac*)

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-tac*

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 $R3\text{-def}$ [*upred-defs*]: $R3(P) = (II \triangleleft \$wait \triangleright P)$

definition $R3c\text{-def}$ [*upred-defs*]: $R3c(P) = (II_r \triangleleft \$wait \triangleright P)$

lemma $R3\text{-idem}$: $R3(R3(P)) = R3(P)$
by *rel-tac*

lemma $R3\text{-mono}$: $P \sqsubseteq Q \implies R3(P) \sqsubseteq R3(Q)$
by *rel-tac*

lemma $R3\text{-conj}$: $R3(P \wedge Q) = (R3(P) \wedge R3(Q))$
by *rel-tac*

lemma $R3\text{-disj}$: $R3(P \vee Q) = (R3(P) \vee R3(Q))$
by *rel-tac*

lemma $R3\text{-USUP}$:
assumes $A \neq \{\}$
shows $R3(\bigsqcap i \in A \cdot P(i)) = (\bigsqcap i \in A \cdot R3(P(i)))$
using *assms* by (*rel-tac*)

lemma $R3\text{-UINF}$:
assumes $A \neq \{\}$
shows $R3(\bigsqcup i \in A \cdot P(i)) = (\bigsqcup i \in A \cdot R3(P(i)))$
using *assms* by (*rel-tac*)

lemma $R3\text{-condr}$: $R3(P \triangleleft b \triangleright Q) = (R3(P) \triangleleft b \triangleright R3(Q))$
by *rel-tac*

lemma $R3\text{-skipr}$: $R3(II) = II$
by *rel-tac*

lemma $R3\text{-form}$: $R3(P) = ((\$wait \wedge II) \vee (\neg \$wait \wedge P))$
by *rel-tac*

lemma $R3\text{-semir-form}$:
 $(R3(P) ;; R3(Q)) = R3(P ;; R3(Q))$
by *rel-tac*

lemma $R3\text{-semir-closure}$:
assumes P is $R3$ Q is $R3$
shows $(P ;; Q)$ is $R3$
using *assms*
by (*metis Healthy-def' R3-semir-form*)

lemma $R3c\text{-semir-form}$:
 $(R3c(P) ;; R3c(R1(Q))) = R3c(P ;; R3c(R1(Q)))$
by *rel-tac*

lemma $R3c\text{-seq-closure}$:
assumes P is $R3c$ Q is $R3c$ Q is $R1$
shows $(P ;; Q)$ is $R3c$
by (*metis Healthy-def' R3c-semir-form assms*)

lemma *R3c-subst-wait*: $R3c(P) = R3c(P_f)$

by (*metis R3c-def cond-var-subst-right wait-uvar*)

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

by *rel-tac*

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

by *rel-tac*

lemma *R2-R3-commute*: $R2(R3(P)) = R3(R2(P))$

by (*rel-tac, (smt add.right-neutral alpha-d.surjective alpha-d.update-convs(2) alpha-rp'.surjective alpha-rp'.update-convs 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-tac, (smt add.right-neutral alpha-d.surjective alpha-d.update-convs(2) alpha-rp'.surjective alpha-rp'.update-convs 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-tac*

lemma *R3c-H2-commute*: $R3c(H2(P)) = H2(R3c(P))$

apply (*simp add: H2-split R3c-def usubst, rel-tac*)

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

done

lemma *R3c-idem*: $R3c(R3c(P)) = R3c(P)$

by *rel-tac*

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

by *rel-tac*

lemma *R3c-USUP*:

assumes $A \neq \{\}$

shows $R3c(\bigcap i \in A \cdot P(i)) = (\bigcap i \in A \cdot R3c(P(i)))$

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

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-tac*)

14.5 RH laws

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

lemma *RH-alt-def*:

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

by (*simp add: R1-idem R2-def RH-def*)

lemma *RH-alt-def'*:

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

by (simp add: R2-def RH-def)

lemma *RH-idem*:
 $RH(RH(P)) = RH(P)$
 by (metis R2-R3c-commute R2-def R2-idem R3c-idem RH-def)

lemma *RH-monotone*:
 $P \sqsubseteq Q \implies RH(P) \sqsubseteq RH(Q)$
 by rel-tac

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-tac)

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-tac)

lemma *RH-intro*:
 $\llbracket P \text{ is } R1; P \text{ is } R2; P \text{ is } R3c \rrbracket \implies P \text{ is } RH$
 by (simp add: Healthy-def' R2-def RH-def)

lemma *RH-seq-closure*:
 assumes $P \text{ is } RH \ Q \text{ is } RH$
 shows $(P ;; Q) \text{ is } RH$
proof (rule RH-intro)
 show $(P ;; Q) \text{ is } R1$
 by (metis Healthy-def' R1-seqr-closure R2-def RH-alt-def RH-def assms(1) assms(2))
 show $(P ;; Q) \text{ is } R2$
 by (metis Healthy-def' R2-def R2-idem R2-seqr-closure RH-def assms(1) assms(2))
 show $(P ;; Q) \text{ is } R3c$
 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-tac)

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 (*simp add: H2-split R1-def usubst, rel-tac*)

lemma *H2-R2s-comm*: $H2(R2s(P)) = R2s(H2(P))$
by (*simp add: H2-split R2s-def usubst, rel-tac*)

lemma *H2-R2-comm*: $H2(R2(P)) = R2(H2(P))$
by (*simp add: H2-R1-comm H2-R2s-comm R2-def*)

lemma *H2-R3-comm*: $H2(R3c(P)) = R3c(H2(P))$
by (*simp add: R3c-H2-commute*)

lemma *R3c-via-H1*: $R1(R3c(H1(P))) = R1(H1(R3(P)))$
by *rel-tac*

lemma *skip-rea-via-H1*: $\Pi_r = R1(H1(R3(\Pi)))$
by *rel-tac*

15.2 Reactive design composition

Pedro's proof for R1 design composition

lemma *R1-design-composition*:

fixes $P\ Q :: ('t :: \text{ordered-cancel-monoid-diff}, 'a, 'b) \text{ relation-rp}$

and $R\ S :: ('t, 'b, 'c) \text{ relation-rp}$

assumes $\$ok' \# P\ \$ok' \# Q\ \$ok \# R\ \$ok \# S$

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 $(R1(P \vdash Q) ;; R1(R \vdash S)) = (\exists\ ok_0 \cdot (R1(P \vdash Q)) \llbracket \llcorner ok_0 \gg / \$ok' \rrbracket ;; (R1(R \vdash S)) \llbracket \llcorner ok_0 \gg / \$ok \rrbracket)$

using *segr-middle uvar-ok by blast*

also from *assms* **have** $\dots = (\exists\ ok_0 \cdot R1((\$ok \wedge P) \Rightarrow (\llcorner ok_0 \gg \wedge Q)) ;; R1((\llcorner ok_0 \gg \wedge R) \Rightarrow (\$ok' \wedge S)))$

by (*simp add: design-def R1-def usubst unrest*)

also from *assms* **have** $\dots = ((R1((\$ok \wedge P) \Rightarrow (true \wedge Q)) ;; R1((true \wedge R) \Rightarrow (\$ok' \wedge S)))$

$\vee (R1((\$ok \wedge P) \Rightarrow (false \wedge Q)) ;; R1((false \wedge R) \Rightarrow (\$ok' \wedge S)))$

by (*simp add: false-alt-def true-alt-def*)

also from *assms* **have** $\dots = ((R1((\$ok \wedge P) \Rightarrow Q) ;; R1(R \Rightarrow (\$ok' \wedge S)))$

$\vee (R1(\neg (\$ok \wedge P)) ;; R1(true)))$

by *simp*

also from *assms* **have** $\dots = ((R1(\neg \$ok \vee \neg P \vee Q) ;; R1(\neg R \vee (\$ok' \wedge S)))$

$\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$

by (*simp add: impl-alt-def utp-pred.sup.assoc*)

also from *assms* **have** $\dots = (((R1(\neg \$ok \vee \neg P) \vee R1(Q)) ;; R1(\neg R \vee (\$ok' \wedge S)))$

$\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$

by (*simp add: R1-disj utp-pred.disj-assoc*)

also from *assms* **have** $\dots = ((R1(\neg \$ok \vee \neg P) ;; R1(\neg R \vee (\$ok' \wedge S)))$

$\vee (R1(Q) ;; R1(\neg R \vee (\$ok' \wedge S)))$

$\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$

by (*simp add: segr-or-distl utp-pred.sup.assoc*)

also from *assms* have ... = $((R1(Q) ;; R1(\neg R \vee (\$ok' \wedge S)))$
 $\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$
by *rel-tac*
also from *assms* have ... = $((R1(Q) ;; (R1(\neg R) \vee R1(S) \wedge \$ok'))$
 $\vee (R1(\neg \$ok \vee \neg P) ;; R1(true)))$
by (*simp add: R1-disj R1-extend-conj utp-pred.inf-commute*)
also have ... = $((R1(Q) ;; (R1(\neg R) \vee R1(S) \wedge \$ok'))$
 $\vee ((R1(\neg \$ok) :: ('t, 'alpha, 'beta) \text{ relation-rp}) ;; R1(true)))$
 $\vee (R1(\neg P) ;; R1(true)))$
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 $((R1(\neg \$ok) :: ('t, 'alpha, 'beta) \text{ relation-rp}) ;; R1(true)) =$
 $(R1(\neg \$ok) :: ('t, 'alpha, 'gamma) \text{ relation-rp})$
by (*rel-tac,metis alpha-d.select-convs(2) alpha-rp'.select-convs(2) order-refl*)
thus ?thesis
by *simp*
qed
also have ... = $((R1(Q) ;; (R1(\neg R) \vee (R1(S \wedge \$ok'))))$
 $\vee R1(\neg \$ok)$
 $\vee (R1(\neg P) ;; R1(true)))$
by (*simp add: R1-extend-conj*)
also have ... = $((R1(Q) ;; (R1(\neg R)))$
 $\vee (R1(Q) ;; (R1(S \wedge \$ok'))))$
 $\vee R1(\neg \$ok)$
 $\vee (R1(\neg P) ;; R1(true)))$
by (*simp add: seqr-or-distr utp-pred.sup.assoc*)
also have ... = $R1((R1(Q) ;; (R1(\neg R)))$
 $\vee (R1(Q) ;; (R1(S \wedge \$ok'))))$
 $\vee (\neg \$ok)$
 $\vee (R1(\neg P) ;; R1(true)))$
by (*simp add: R1-disj R1-seqr*)
also have ... = $R1((R1(Q) ;; (R1(\neg R)))$
 $\vee ((R1(Q) ;; R1(S)) \wedge \$ok')$
 $\vee (\neg \$ok)$
 $\vee (R1(\neg P) ;; R1(true)))$
by (*rel-tac*)
also have ... = $R1(\neg(\$ok \wedge \neg (R1(\neg P) ;; R1(true)) \wedge \neg (R1(Q) ;; (R1(\neg R))))$
 $\vee ((R1(Q) ;; R1(S)) \wedge \$ok'))$
by (*rel-tac*)
also have ... = $R1((\$ok \wedge \neg (R1(\neg P) ;; R1(true)) \wedge \neg (R1(Q) ;; (R1(\neg R))))$
 $\Rightarrow (\$ok' \wedge (R1(Q) ;; R1(S))))$
by (*simp add: impl-alt-def utp-pred.inf-commute*)
also have ... = $R1((\neg (R1(\neg P) ;; R1(true)) \wedge \neg (R1(Q) ;; R1(\neg R))) \vdash (R1(Q) ;; R1(S)))$
by (*simp add: design-def*)
finally show ?thesis .
qed

definition [*upred-defs*]: $R3c\text{-pre}(P) = (true \triangleleft \$wait \triangleright P)$

definition [*upred-defs*]: $R3c\text{-post}(P) = ([II]_D \triangleleft \$wait \triangleright P)$

lemma *R3c-pre-conj*: $R3c\text{-pre}(P \wedge Q) = (R3c\text{-pre}(P) \wedge R3c\text{-pre}(Q))$

by *rel-tac*

lemma *R3c-pre-seq*:
 $(true ;; Q) = true \implies R3c\text{-pre}(P ;; Q) = (R3c\text{-pre}(P) ;; Q)$
 by (*rel-tac*)

lemma *R2s-design*: $R2s(P \vdash Q) = (R2s(P) \vdash R2s(Q))$
 by (*simp add: R2s-def design-def usubst*)

lemma *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*:
 $R1(R3c(P \vdash Q)) = R1(R3c\text{-pre}(P) \vdash R3c\text{-post}(Q))$
 by (*rel-tac, simp-all add: alpha-d.equality*)

lemma *unrest-ok-R2s* [*unrest*]: $\$ok \# P \implies \$ok \# R2s(P)$
 by (*simp add: R2s-def unrest*)

lemma *unrest-ok'-R2s* [*unrest*]: $\$ok' \# P \implies \$ok' \# R2s(P)$
 by (*simp add: R2s-def unrest*)

lemma *unrest-ok-R2c* [*unrest*]: $\$ok \# P \implies \$ok \# R2c(P)$
 by (*simp add: R2c-def unrest*)

lemma *unrest-ok'-R2c* [*unrest*]: $\$ok' \# P \implies \$ok' \# R2c(P)$
 by (*simp add: R2c-def unrest*)

lemma *unrest-ok-R3c-pre* [*unrest*]: $\$ok \# P \implies \$ok \# R3c\text{-pre}(P)$
 by (*simp add: R3c-pre-def cond-def unrest*)

lemma *unrest-ok'-R3c-pre* [*unrest*]: $\$ok' \# P \implies \$ok' \# R3c\text{-pre}(P)$
 by (*simp add: R3c-pre-def cond-def unrest*)

lemma *unrest-ok-R3c-post* [*unrest*]: $\$ok \# P \implies \$ok \# R3c\text{-post}(P)$
 by (*simp add: R3c-post-def cond-def unrest*)

lemma *unrest-ok-R3c-post'* [*unrest*]: $\$ok' \# P \implies \$ok' \# R3c\text{-post}(P)$
 by (*simp add: R3c-post-def cond-def unrest*)

lemma *R3c-R1-design-composition*:
 assumes $\$ok' \# P \ \$ok' \# Q \ \$ok \# R \ \$ok \# S$
 shows $(R3c(R1(P \vdash Q)) ;; R3c(R1(R \vdash S))) =$
 $R3c(R1((\neg (R1(\neg P) ;; R1(true)) \wedge \neg ((R1(Q) \wedge \neg \$wait') ;; R1(\neg R))))$
 $\vdash (R1(Q) ;; ([II]_D \triangleleft \$wait \triangleright R1(S))))$

proof –
 have 1: $(\neg (R1(\neg R3c\text{-pre } P) ;; R1\ true)) = (R3c\text{-pre } (\neg (R1(\neg P) ;; R1\ true)))$
 by (*rel-tac*)
 have 2: $(\neg (R1(R3c\text{-post } Q) ;; R1(\neg R3c\text{-pre } R))) = R3c\text{-pre}(\neg (R1\ Q \wedge \neg \$wait' ;; R1(\neg R)))$
 by (*rel-tac*)
 have 3: $(R1(R3c\text{-post } Q) ;; R1(R3c\text{-post } S)) = R3c\text{-post } (R1\ Q ;; ([II]_D \triangleleft \$wait \triangleright R1\ S))$
 by (*rel-tac*)
 show ?thesis
 apply (*simp add: R3c-semir-form R1-R3c-commute[THEN sym] R1-R3c-design unrest*)
 apply (*subst R1-design-composition*)

apply (*simp-all add: unrest assms R3c-pre-conj 1 2 3*)
done
qed

lemma *R1-des-lift-skip*: $R1(\lceil II \rceil_D) = \lceil II \rceil_D$
by (*rel-tac*)

lemma *R2s-subst-wait-true* [*usubst*]:
 $(R2s(P))\llbracket true/\$wait \rrbracket = R2s(P\llbracket true/\$wait \rrbracket)$
by (*simp add: R2s-def usubst unrest*)

lemma *R2s-subst-wait'-true* [*usubst*]:
 $(R2s(P))\llbracket true/\$wait' \rrbracket = R2s(P\llbracket true/\$wait' \rrbracket)$
by (*simp add: R2s-def usubst unrest*)

lemma *R2-subst-wait-true* [*usubst*]:
 $(R2(P))\llbracket true/\$wait \rrbracket = R2(P\llbracket true/\$wait \rrbracket)$
by (*simp add: R2-def R1-def R2s-def usubst unrest*)

lemma *R2-subst-wait'-true* [*usubst*]:
 $(R2(P))\llbracket true/\$wait' \rrbracket = R2(P\llbracket true/\$wait' \rrbracket)$
by (*simp add: R2-def R1-def R2s-def usubst unrest*)

lemma *R2-subst-wait-false* [*usubst*]:
 $(R2(P))\llbracket false/\$wait \rrbracket = R2(P\llbracket false/\$wait \rrbracket)$
by (*simp add: R2-def R1-def R2s-def usubst unrest*)

lemma *R2-subst-wait'-false* [*usubst*]:
 $(R2(P))\llbracket false/\$wait' \rrbracket = R2(P\llbracket false/\$wait' \rrbracket)$
by (*simp add: R2-def R1-def R2s-def usubst unrest*)

lemma *R2-des-lift-skip*:
 $R2(\lceil II \rceil_D) = \lceil II \rceil_D$
by (*rel-tac, metis 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-tac*)

lemma *R2-design-composition*:
assumes $\$ok' \# P \ \$ok' \# Q \ \$ok \# R \ \$ok \# S$
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 $\$ok' \# P \ \$ok' \# Q \ \$ok \# R \ \$ok \# S$
shows $(RH(P \vdash Q) ;; RH(R \vdash S)) =$
 $RH((\neg (R1 (\neg R2s P) ;; R1 true) \wedge \neg (R1 (R2s Q) \wedge \neg \$wait' ;; R1 (\neg R2s R))) \vdash (R1 (R2s Q) ;; (\lceil II \rceil_D \triangleleft \$wait \triangleright R1 (R2s S))))$

proof –
have 1: $R2c (R1 (\neg R2s P) ;; R1 true) = (R1 (\neg R2s P) ;; R1 true)$

proof –

have 1: $(R1 \ (\neg \ R2s \ P) \ ;\ ;\ R1 \ true) = (R1(R2 \ (\neg \ P) \ ;\ ;\ R2 \ true))$
 by *(rel-tac)*

have $R2c(R1(R2 \ (\neg \ P) \ ;\ ;\ R2 \ true)) = R2c(R1(R2 \ (\neg \ P) \ ;\ ;\ R2 \ true))$
 using *R2c-not* by *blast*

also have $\dots = R2(R2 \ (\neg \ P) \ ;\ ;\ R2 \ true)$
 by *(metis R1-R2c-commute R1-R2c-is-R2)*

also have $\dots = (R2 \ (\neg \ P) \ ;\ ;\ R2 \ true)$
 by *(simp add: R2-seqr-distribute)*

also have $\dots = (R1 \ (\neg \ R2s \ P) \ ;\ ;\ R1 \ true)$
 by *(simp add: R2-def R2s-not R2s-true)*

finally show *?thesis*
 by *(simp add: 1)*

qed

have 2: $R2c \ (R1 \ (R2s \ Q) \wedge \neg \ \$wait' \ ;\ ;\ R1 \ (\neg \ R2s \ R)) = (R1 \ (R2s \ Q) \wedge \neg \ \$wait' \ ;\ ;\ R1 \ (\neg \ R2s \ R))$

proof –

have $(R1 \ (R2s \ Q) \wedge \neg \ \$wait' \ ;\ ;\ R1 \ (\neg \ R2s \ R)) = R1 \ (R2 \ (Q \wedge \neg \ \$wait') \ ;\ ;\ R2 \ (\neg \ R))$
 by *(rel-tac)*

hence $R2c \ (R1 \ (R2s \ Q) \wedge \neg \ \$wait' \ ;\ ;\ R1 \ (\neg \ R2s \ R)) = (R2 \ (Q \wedge \neg \ \$wait') \ ;\ ;\ R2 \ (\neg \ R))$
 by *(metis R1-R2c-commute R1-R2c-is-R2 R2-seqr-distribute)*

also have $\dots = (R1 \ (R2s \ Q) \wedge \neg \ \$wait' \ ;\ ;\ R1 \ (\neg \ R2s \ R))$
 by *rel-tac*

finally show *?thesis* .

qed

have 3: $R2c((R1 \ (R2s \ Q) \ ;\ ;\ (\lceil II \rceil_D \triangleleft \ \$wait \triangleright R1 \ (R2s \ S)))) = (R1 \ (R2s \ Q) \ ;\ ;\ (\lceil II \rceil_D \triangleleft \ \$wait \triangleright R1 \ (R2s \ S)))$

proof –

have $R2c(((R1 \ (R2s \ Q))\llbracket true/\$wait' \rrbracket \ ;\ ;\ (\lceil II \rceil_D \triangleleft \ \$wait \triangleright R1 \ (R2s \ S))\llbracket true/\$wait \rrbracket))$
 $= ((R1 \ (R2s \ Q))\llbracket true/\$wait' \rrbracket \ ;\ ;\ (\lceil II \rceil_D \triangleleft \ \$wait \triangleright R1 \ (R2s \ S))\llbracket true/\$wait \rrbracket)$

proof –

have $R2c(((R1 \ (R2s \ Q))\llbracket true/\$wait' \rrbracket \ ;\ ;\ (\lceil II \rceil_D \triangleleft \ \$wait \triangleright R1 \ (R2s \ S))\llbracket true/\$wait \rrbracket)) =$
 $R2c(R1 \ (R2s \ (Q\llbracket true/\$wait' \rrbracket)) \ ;\ ;\ \lceil II \rceil_D \llbracket true/\$wait \rrbracket)$

by *(simp add: usubst cond-unit-T R1-def R2s-def, rel-tac)*

also have $\dots = R2c(R2 \ (Q\llbracket true/\$wait' \rrbracket) \ ;\ ;\ R2 \ (\lceil II \rceil_D \llbracket true/\$wait \rrbracket))$

by *(metis R2-def R2-des-lift-skip R2-subst-wait-true)*

also have $\dots = (R2 \ (Q\llbracket true/\$wait' \rrbracket) \ ;\ ;\ R2 \ (\lceil II \rceil_D \llbracket true/\$wait \rrbracket))$

using *R2c-seq* by *blast*

also have $\dots = ((R1 \ (R2s \ Q))\llbracket true/\$wait' \rrbracket \ ;\ ;\ (\lceil II \rceil_D \triangleleft \ \$wait \triangleright R1 \ (R2s \ S))\llbracket true/\$wait \rrbracket)$

apply *(simp add: usubst cond-unit-T R2-des-lift-skip)*

apply *(metis R2-def R2-des-lift-skip R2-subst-wait'-true R2-subst-wait-true)*

done

finally show *?thesis* .

qed

moreover have $R2c(((R1 \ (R2s \ Q))\llbracket false/\$wait' \rrbracket \ ;\ ;\ (\lceil II \rceil_D \triangleleft \ \$wait \triangleright R1 \ (R2s \ S))\llbracket false/\$wait \rrbracket))$
 $= ((R1 \ (R2s \ Q))\llbracket false/\$wait' \rrbracket \ ;\ ;\ (\lceil II \rceil_D \triangleleft \ \$wait \triangleright R1 \ (R2s \ S))\llbracket false/\$wait \rrbracket)$

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 ... = $R3c(R1((\neg(R1(\neg R2s P) ;; R1 true) \wedge \neg(R1(R2s Q) \wedge \neg \$wait' ;; R1(\neg R2s R)))) \vdash$
 $(R1(R2s Q) ;; ([II]_D \triangleleft \$wait \triangleright R1(R2s S))))$
 by (simp add: R3c-R1-design-composition assms unrest)
 also have ... = $R3c(R1(R2c((\neg(R1(\neg R2s P) ;; R1 true) \wedge \neg(R1(R2s Q) \wedge \neg \$wait' ;; R1(\neg R2s R)))) \vdash$
 $(R1(R2s Q) ;; ([II]_D \triangleleft \$wait \triangleright R1(R2s S))))$
 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-tac)

lemma *RH-design-export-R2s*: $RH(P \vdash Q) = RH(P \vdash R2s(Q))$
 by (rel-tac)

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 $RH(P_1 \vdash Q_1) \subseteq RH(P_2 \vdash Q_2)$
 by (simp add: RH-monotone assms(1) assms(2) design-refine-intro)

Marcel's proof for reactive design composition

lemma *reactive-design-composition*:

assumes $out\alpha \# p_1$ 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-tac, metis alpha-d.select-convs(1) alpha-d.select-convs(2) order-refl)
 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-tac, metis alpha-d.select-convs(1) alpha-d.select-convs(2) order-refl)
 ultimately show ?thesis
 by (smt assms(1) precondition-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-tac)
 have $2: ((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr') ;; (\$wait \wedge \neg \$ok \wedge \$tr \leq_u \$tr')) = false$
 by rel-tac
 have $3: ((\$ok' \wedge Q_1 \wedge \$tr \leq_u \$tr') ;; (\neg \$wait \wedge \neg \$ok \wedge \$tr \leq_u \$tr')) = false$
 by rel-tac
 have $4: R2(\neg P_2) = R1(\neg P_2)$
 by (metis Healthy-def' R1-negate-R1 R2-def R2s-not assms(3))
 show ?thesis
 by (simp add: 1 2 3 4 seqr-or-distr)

qed

also have ... = $RH((\neg \$ok) \vee (\neg p_1)$
 $\vee ((\$ok' \wedge Q_1) ;; (\$wait \wedge \$ok' \wedge II))$
 $\vee ((\$ok' \wedge Q_1) ;; (\neg \$wait \wedge R1(\neg P_2)))$
 $\vee ((\$ok' \wedge Q_1) ;; (\neg \$wait \wedge \$ok' \wedge R2(Q_2))))$
 by (rel-tac)

also have ... = $RH((\neg \$ok) \vee (\neg p_1)$
 $\vee (\$ok' \wedge \$wait' \wedge Q_1)$
 $\vee ((\$ok' \wedge Q_1) ;; (\neg \$wait \wedge R1(\neg P_2)))$
 $\vee ((\$ok' \wedge Q_1) ;; (\neg \$wait \wedge \$ok' \wedge R1(Q_2))))$

proof –

have $((\$ok' \wedge Q_1) ;; (\$wait \wedge \$ok' \wedge H)) = (\$ok' \wedge \$wait' \wedge Q_1)$
by *(rel-tac)*
moreover have $R2(Q_2) = R1(Q_2)$
by *(metis Healthy-def' R2-def assms(5))*
ultimately show *?thesis* **by** *simp*
qed

also have $\dots = RH((\neg \$ok) \vee (\neg p_1) \vee (\$ok' \wedge \$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; (R1(\neg P_2))) \vee ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; (\$ok' \wedge R1(Q_2))))$
by *rel-tac*

also have $\dots = RH((\neg \$ok) \vee (\neg p_1) \vee ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; R1(\neg P_2)) \vee (\$ok' \wedge ((\$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; R1(Q_2))))$
by *rel-tac*

also have $\dots = RH(\neg (\$ok \wedge p_1 \wedge \neg ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; R1(\neg P_2))) \vee (\$ok' \wedge ((\$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; R1(Q_2))))$
by *rel-tac*

also have $\dots = ?rhs$
proof –
have $(\neg (\$ok \wedge p_1 \wedge \neg ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; R1(\neg P_2))) \vee (\$ok' \wedge ((\$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; R1(Q_2))))$
 $= ((\$ok \wedge (p_1 \wedge \neg ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; R1(\neg P_2)))) \Rightarrow (\$ok' \wedge ((\$wait' \wedge Q_1) \vee ((\$ok' \wedge \neg \$wait' \wedge Q_1) ;; R1(Q_2))))$
by *pred-tac*
thus *?thesis*
by *(simp add: design-def)*
qed

finally show *?thesis* .
qed

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-tac*

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-tac)

lemma *CSP1-R1-commute*:
 $CSP1(R1(P)) = R1(CSP1(P))$
 by (rel-tac)

lemma *CSP1-R2c-commute*:
 $CSP1(R2c(P)) = R2c(CSP1(P))$
 by (rel-tac)

lemma *CSP1-R3c-commute*:
 $CSP1(R3c(P)) = R3c(CSP1(P))$
 by (rel-tac)

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-tac

lemma *CSP1-R3c*: $CSP1(R3(P)) = R3c(CSP1(P))$
 by rel-tac

lemma *CSP1-reactive-design*: $CSP1(RH(P \vdash Q)) = RH(P \vdash Q)$
 by rel-tac

lemma *CSP2-reactive-design*:
 assumes $\$ok' \nmid P \ \$ok' \nmid Q$
 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-tac

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

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 $\dots = CSP1(H2(R1(R2s(R3c(P)))))$
 by (simp add: CSP2-def)
 also have $\dots = CSP1(R1(H2(R2s(R3c(P)))))$
 by (simp add: R1-H2-commute)
 also have $\dots = R1(H1(R1(H2(R2s(R3c(P)))))$
 by (simp add: CSP1-R1-H1 R1-idem)
 also have $\dots = R1(H1(H2(R2s(R3c(R1(P)))))$
 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 ... = $RH(H1-H2(R1(P)))$
by (*metis R1-R2-commute R1-idem R2-R3c-commute R2-def RH-def*)
also have ... = $RH(H1-H2(P))$
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 $0: (\neg (H1-H2(P))^f) = (\$ok \wedge \neg P^f)$
by (*simp add: H1-def H2-split, pred-tac*)
have $1: (H1-H2(P))^t = (\$ok \Rightarrow (P^f \vee P^t))$
by (*simp add: H1-def H2-split, pred-tac*)
have $(\neg (H1-H2(P))^f) \vdash (H1-H2(P))^t = ((\neg P^f) \vdash P^t)$
by (*simp add: 0 1, pred-tac*)
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 $\$ok' \# P \ \$ok' \# Q$
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* :: $- \Rightarrow - \Rightarrow -$ (*infix* \diamond 65) **where**
[upred-defs]: $P \diamond Q = (P \triangleleft \$wait' \triangleright Q)$

lemma *wait'-cond-unrest* [*unrest*]:

$\llbracket out-var \ wait \triangleright x; x \# P; x \# Q \rrbracket \Longrightarrow x \# (P \diamond Q)$
by (*simp add: wait'-cond-def unrest*)

lemma *wait'-cond-subst* [*usubst*]:

$\$wait' \# \sigma \Longrightarrow \sigma \dagger (P \diamond Q) = (\sigma \dagger P) \diamond (\sigma \dagger Q)$
by (*simp add: wait'-cond-def usubst unrest*)

lemma *wait'-cond-left-false*: $false \diamond P = (\neg \$wait' \wedge P)$
by (*rel-tac*)

lemma *wait'-cond-seq*: $((P \diamond Q) ;; R) = ((P ;; \$wait \wedge R) \vee (Q ;; \neg \$wait \wedge R))$

by (simp add: wait'-cond-def cond-def segr-or-distl, rel-tac)

lemma wait'-cond-true: $(P \diamond Q \wedge \$wait') = (P \wedge \$wait')$
 by (rel-tac)

lemma wait'-cond-false: $(P \diamond Q \wedge (\neg \$wait')) = (Q \wedge (\neg \$wait'))$
 by (rel-tac)

lemma wait'-cond-idem: $P \diamond P = P$
 by (rel-tac)

lemma wait'-cond-conj-exchange:
 $((P \diamond Q) \wedge (R \diamond S)) = (P \wedge R) \diamond (Q \wedge S)$
 by rel-tac

lemma subst-wait'-cond-true [usubst]: $(P \diamond Q)[\$true/\$wait'] = P[\$true/\$wait']$
 by rel-tac

lemma subst-wait'-cond-false [usubst]: $(P \diamond Q)[\$false/\$wait'] = Q[\$false/\$wait']$
 by rel-tac

lemma subst-wait'-left-subst: $(P[\$true/\$wait'] \diamond Q) = (P \diamond Q)$
 by (metis wait'-cond-def cond-def conj-comm conj-eq-out-var-subst upred-eq-true wait-uvar)

lemma subst-wait'-right-subst: $(P \diamond Q[\$false/\$wait']) = (P \diamond Q)$
 by (metis cond-def conj-eq-out-var-subst upred-eq-false utp-pred.inf commute wait'-cond-def wait-uvar)

lemma wait'-cond-split: $P[\$true/\$wait'] \diamond P[\$false/\$wait'] = P$
 by (simp add: wait'-cond-def cond-var-split)

lemma R1-wait'-cond: $R1(P \diamond Q) = R1(P) \diamond R1(Q)$
 by rel-tac

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:

$RH(P \vdash (R1(R2c(Q)) \vee R) \diamond S) = RH(P \vdash (Q \vee R) \diamond S)$

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 $\$ok' \# P \ \$ok' \# Q_1 \ \$ok' \# Q_2 \ \$ok \# R \ \$ok \# S_1 \ \$ok \# S_2$

$\$wait' \# Q_2 \ \$wait \# S_1 \ \$wait \# S_2$

shows $(RH(P \vdash Q_1 \diamond Q_2) ;; RH(R \vdash S_1 \diamond S_2)) =$

$RH((\neg (R1 (\neg R2s P) ;; R1 true) \wedge \neg (R1 (R2s Q_2) \wedge \neg \$wait' ;; R1 (\neg R2s R))) \vdash$
 $((Q_1 \vee (R1 (R2s Q_2) ;; R1 (R2s S_1))) \diamond ((R1 (R2s Q_2) ;; R1 (R2s S_2))))$

proof –

have 1: $(\neg (R1 (R2s (Q_1 \diamond Q_2)) \wedge \neg \$wait' ;; R1 (\neg R2s R))) =$

$(\neg (R1 (R2s Q_2) \wedge \neg \$wait' ;; R1 (\neg R2s R)))$

by (metis (no-types, hide-lams) R1-extend-conj R2s-conj R2s-not R2s-wait' wait'-cond-false)

have 2: $(R1 (R2s (Q_1 \diamond Q_2)) ;; ([II]_D \triangleleft \$wait \triangleright R1 (R2s (S_1 \diamond S_2)))) =$

$((R1 (R2s Q_1) \vee (R1 (R2s Q_2) ;; R1 (R2s S_1))) \diamond (R1 (R2s Q_2) ;; R1 (R2s S_2)))$

proof –

have $(R1 (R2s Q_1) ;; \$wait \wedge ([II]_D \triangleleft \$wait \triangleright R1 (R2s S_1) \diamond R1 (R2s S_2)))$

$= (R1 (R2s Q_1) \wedge \$wait')$

proof –

have $(R1 (R2s Q_1) ;; \$wait \wedge ([II]_D \triangleleft \$wait \triangleright R1 (R2s S_1) \diamond R1 (R2s S_2)))$

$= (R1 (R2s Q_1) ;; \$wait \wedge [II]_D)$

by (rel-tac)

also have $\dots = ((R1 (R2s Q_1) ;; [II]_D) \wedge \$wait')$

by (rel-tac)

also from assms(2) have $\dots = ((R1 (R2s Q_1)) \wedge \$wait')$

by (simp add: lift-des-skip-dr-unit-unrest unrest)

finally show ?thesis .

qed

moreover have $(R1 (R2s Q_2) ;; \neg \$wait \wedge ([II]_D \triangleleft \$wait \triangleright R1 (R2s S_1) \diamond R1 (R2s S_2)))$

$= ((R1 (R2s Q_2)) ;; (R1 (R2s S_1) \diamond R1 (R2s S_2)))$

proof –

have $(R1 (R2s Q_2) ;; \neg \$wait \wedge ([II]_D \triangleleft \$wait \triangleright R1 (R2s S_1) \diamond R1 (R2s S_2)))$

$= (R1 (R2s Q_2) ;; \neg \$wait \wedge (R1 (R2s S_1) \diamond R1 (R2s S_2)))$

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 $\dots = ((R1 (R2s Q_2)) \llbracket false/\$wait' \rrbracket ;; (R1 (R2s S_1) \diamond R1 (R2s S_2)) \llbracket false/\$wait \rrbracket)$

by (metis false-alt-def seqr-right-one-point upred-eq-false wait-uvar)

also have $\dots = ((R1 (R2s Q_2)) ;; (R1 (R2s S_1) \diamond R1 (R2s S_2)))$

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*]: $\$ok \# pre_R P$
 by (*simp add: pre_R-def unrest usubst*)

lemma *ok-peri-unrest* [*unrest*]: $\$ok \# peri_R P$
 by (*simp add: peri_R-def unrest usubst*)

lemma *ok-post-unrest* [*unrest*]: $\$ok \# post_R P$
 by (*simp add: post_R-def unrest usubst*)

lemma *ok'-pre-unrest* [*unrest*]: $\$ok' \# pre_R P$
 by (*simp add: pre_R-def unrest usubst*)

lemma *ok'-peri-unrest* [*unrest*]: $\$ok' \# peri_R P$
 by (*simp add: peri_R-def unrest usubst*)

lemma *ok'-post-unrest* [*unrest*]: $\$ok' \# post_R P$
 by (*simp add: post_R-def unrest usubst*)

lemma *wait-pre-unrest* [*unrest*]: $\$wait \# pre_R P$
 by (*simp add: pre_R-def unrest usubst*)

lemma *wait-peri-unrest* [*unrest*]: $\$wait \# peri_R P$
 by (*simp add: peri_R-def unrest usubst*)

lemma *wait-post-unrest* [*unrest*]: $\$wait \# post_R P$
 by (*simp add: post_R-def unrest usubst*)

lemma *wait'-peri-unrest* [*unrest*]: $\$wait' \# peri_R P$
 by (*simp add: peri_R-def unrest usubst*)

lemma *wait'-post-unrest* [*unrest*]: $\$wait' \# post_R P$
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 *pre_s-design*: $pre_s \dagger (P \vdash Q \diamond R) = pre_s \dagger (P \Rightarrow Q)$
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 *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 *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*: $peri_R(RH(P \vdash Q \diamond R)) = R1(R2c(pre_s \dagger (P \Rightarrow Q)))$
by (*simp add: RH-R2c-def usubst peri_R-def R3c-def pre_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 $RH((\neg P^f_f) \vdash P^t_f \llbracket true/\$wait' \rrbracket \diamond P^t_f \llbracket false/\$wait' \rrbracket) = P$
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 $P = RH((\neg P^f_f) \vdash P^t_f \llbracket true/\$wait' \rrbracket \diamond P^t_f \llbracket false/\$wait' \rrbracket)$
by (*simp add: CSP-reactive-tri-design-lemma assms*)
also have $\dots = RH(pre_R(P) \vdash peri_R(P) \diamond post_R(P))$
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 $\dots = R1(R3c(R2c(true \vdash II)))$

by (metis *R2c-R3c-commute RH-R2c-def*)

also have $\dots = R1(R3c(true \vdash II))$

by (simp add: *design-def impl-alt-def R2c-disj R2c-not R2c-ok R2c-and R2c-skip-r R2c-ok*)

also have $\dots = R1(II_r \triangleleft \$wait \triangleright true \vdash II)$

by (metis *R3c-def*)

also have $\dots = II_r$

by (rel-tac)

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*: $RH(P \vdash_f Q \vdash_f) = RH(P \vdash Q)$

by (metis *RH-subst-wait wait-false-design*)

lemma *RH-design-subst-wait-pre*: $RH(P \vdash_f Q) = RH(P \vdash Q)$

by (subst *RH-design-subst-wait[THEN sym]*, simp add: *usubst RH-design-subst-wait*)

lemma *RH-design-subst-wait-post*: $RH(P \vdash Q \vdash_f) = RH(P \vdash Q)$

by (subst *RH-design-subst-wait[THEN sym]*, simp add: *usubst RH-design-subst-wait*)

lemma *RH-peri-subst-false-wait*: $RH(P \vdash Q \vdash_f \diamond R) = RH(P \vdash Q \diamond R)$

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-uvar*)

done

lemma *RH-post-subst-false-wait*: $RH(P \vdash Q \diamond R \vdash_f) = RH(P \vdash Q \diamond R)$

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-uvar*)

done

lemma *skip-rea-reactive-tri-design*:

$II_r = RH(true \vdash false \diamond \lceil II \rceil_D)$ (is ?lhs = ?rhs)

proof –

have $?rhs = RH(true \vdash (\neg \$wait' \wedge \lceil II \rceil_D))$

by (simp add: *wait'-cond-def cond-def*)

have $\dots = RH(true \vdash (\neg \$wait \wedge \lceil II \rceil_D))$ (is $RH(true \vdash ?Q1) = RH(true \vdash ?Q2)$)

proof –

have $?Q1 = ?Q2$

by (rel-tac)


```

    thus ?thesis by simp
qed
also have ... = RH (true ⊢ [II]D)
  by (rel-tac)
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 ∧ $ΣR' =u $ΣR)
  by (rel-tac)

lemma skip-rea-reactive-tri-design':
  IIr = RH(true ⊢ false ◊ ($tr' =u $tr ∧ $ΣR' =u $ΣR)) (is ?lhs = ?rhs)
proof -
  have ?rhs = RH (true ⊢ (¬ $wait' ∧ $tr' =u $tr ∧ $ΣR' =u $ΣR))
    by (simp add: wait'-cond-def cond-def)
  also have ... = RH (true ⊢ ($wait' =u $wait ∧ $tr' =u $tr ∧ $ΣR' =u $ΣR)) (is RH (true ⊢ ?Q1)
= RH (true ⊢ ?Q2))
  proof -
    have ?Q1f = ?Q2f
      by (rel-tac)
    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 (¬ preR P) = (¬ preR (R1(P)))
  by (simp add: preR-def R1-def usubst)

lemma R1-peri: R1 (periR P) = periR (R1(P))
  by (simp add: periR-def R1-def usubst)

lemma R1-post: R1 (postR P) = postR (R1(P))
  by (simp add: postR-def R1-def usubst)

lemma R2s-pre:
  R2s (preR P) = preR (R2s P)
  by (simp add: preR-def R2s-def usubst)

lemma R2s-peri: R2s (periR P) = periR (R2s P)
  by (simp add: periR-def R2s-def usubst)

lemma R2s-post: R2s (postR P) = postR (R2s P)
  by (simp add: postR-def R2s-def usubst)

lemma RH-pre-RH-design:
  $ok' ‡ P ⇒ RH(preR(RH(P ⊢ Q)) ⊢ R) = RH(P ⊢ 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 uvar-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 $\dots = RH(P \vdash (\$ok \wedge P^t \Rightarrow Q^t))$
by (*subst RH-design-subst-wait-post[THEN sym, of - (\\$ok \wedge P^t \Rightarrow Q^t)], simp add: usubst*)
also have $\dots = RH(P \vdash (P^t \Rightarrow Q^t))$
by (*rel-tac*)
also have $\dots = RH(P \vdash (P \Rightarrow Q))$
by (*subst design-subst-ok'[THEN sym, of - P \Rightarrow Q], simp add: usubst*)
also have $\dots = RH(P \vdash Q)$
by (*rel-tac*)
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 uvar-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-uvar*)
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-uvar*)
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*: $P \text{ is CSP} \implies R1(R2s P) = P$
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-tac*) **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-tac, simp-all add: alpha-d.equality alpha-rp'.equality*)

lemma *wait-pre-lemma*:

assumes $\$wait' \# P$
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-tac*)

also have $\dots = (P \llbracket false/\$wait' \rrbracket ;; (\neg pre_R Q) \llbracket false/\$wait \rrbracket)$
by (*metis false-alt-def seqr-left-one-point wait-uvar*)

also have $\dots = (P ;; \neg pre_R Q)$
by (*simp add: usubst unrest assms*)

finally show *?thesis* .

qed

lemma *rea-left-unit-lemma*:

assumes $\$ok \# P \ \$wait \# P$
shows $((\$tr' =_u \$tr \wedge \$ \Sigma_R' =_u \$ \Sigma_R) ;; P) = P$

proof –

have $P = (II ;; P)$
by *simp*

also have $\dots = ((\$ok' =_u \$ok \wedge \$wait' =_u \$wait \wedge \$tr' =_u \$tr \wedge \$ \Sigma_R' =_u \$ \Sigma_R) ;; P)$
by (*metis skip-r-rea*)

also from *assms* **have** $\dots = ((\$tr' =_u \$tr \wedge \$ \Sigma_R' =_u \$ \Sigma_R) ;; P)$
by (*simp add: seqr-insert-ident-left assms unrest*)

finally show *?thesis* ..

qed

lemma *rea-right-unit-lemma*:

assumes $\$ok' \# P \ \$wait' \# P$
shows $(P ;; (\$tr' =_u \$tr \wedge \$ \Sigma_R' =_u \$ \Sigma_R)) = P$

proof –

have $P = (P ;; II)$
by *simp*

also have $\dots = (P ;; (\$ok' =_u \$ok \wedge \$wait' =_u \$wait \wedge \$tr' =_u \$tr \wedge \$ \Sigma_R' =_u \$ \Sigma_R))$
by (*metis skip-r-rea*)

also from *assms* **have** $\dots = (P ;; (\$tr' =_u \$tr \wedge \$ \Sigma_R' =_u \$ \Sigma_R))$
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 Σ_R' =_u Σ_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 α $\#$ 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 Σ_R' =_u Σ_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-tac)

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-tac)

15.5 Signature

definition [urel-defs]: Miracle = RH(true \vdash false \diamond false)

definition [urel-defs]: Chaos = RH(false \vdash true \diamond true)

definition assigns-rea :: 'α usubst \Rightarrow ('t::ordered-cancel-monoid-diff, 'α) hrelation-rp ($\langle \cdot \rangle_R$) **where**
 assigns-rea σ = RH(true \vdash false \diamond (\$tr' =_u \$tr \wedge [$\langle \sigma \rangle_a$]_R))

definition reactive-sup :: - set \Rightarrow - (\sqcap_R) **where**
 $\sqcap_R A$ = (if (A = {}) then Miracle else $\sqcap A$)

definition reactive-inf :: - set \Rightarrow - (\sqcup_R) **where**
 $\sqcup_R A$ = (if (A = {}) then Chaos else $\sqcup A$)

definition *rea-design-par* :: $- \Rightarrow - \Rightarrow -$ (**infixr** \parallel_R 85) **where**
 $P \parallel_R Q = RH((pre_R(P) \wedge pre_R(Q)) \vdash (P^t_f \wedge Q^t_f))$

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 $\dots \sqsubseteq RH(true \vdash false)$
by (*rule RH-monotone, rel-tac*)
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 $\dots \sqsubseteq RH(pre_R(P) \vdash peri_R(P) \diamond post_R(P))$
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 $\dots = RH(true \vdash false \diamond false)$
by (*simp add: RH-tri-design-composition R1-false R2s-true R2s-false R2c-true R1-true-comp unrest usubst*)
also have $\dots = Miracle$
by (*simp add: Miracle-def*)
finally show *?thesis* .

qed

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 $\dots = RH((\neg R1 \ true \wedge \neg (R1 \ true \wedge \neg \$wait' ;; R1 (\neg R2c (pre_R P)))) \vdash$
 $(true \vee (R1 \ true ;; R1 (R2c (peri_R P)))) \diamond (R1 \ true ;; R1 (R2c (post_R P))))$
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 $\dots = RH((\neg \$ok \vee R1 \ true \vee (R1 \ true \wedge \neg \$wait' ;; R1 (\neg R2c (pre_R P)))) \vee$

$\$ok' \wedge (true \vee (R1 \text{ true} ;; R1 (R2c (peri_R P)))) \diamond (R1 \text{ true} ;; R1 (R2c (post_R P))))$
 by (simp add: design-def impl-alt-def)
 also have ... = $RH(R1((\neg \$ok \vee R1 \text{ true} \vee (R1 \text{ true} \wedge \neg \$wait' ;; R1 (\neg R2c (pre_R P)))) \vee$
 $\$ok' \wedge (true \vee (R1 \text{ true} ;; R1 (R2c (peri_R P)))) \diamond (R1 \text{ true} ;; R1 (R2c (post_R P))))$
 by (simp add: R1-R2c-commute R1-R3c-commute R1-idem RH-R2c-def)
 also have ... = $RH(R1((\neg \$ok \vee true \vee (R1 \text{ true} \wedge \neg \$wait' ;; R1 (\neg R2c (pre_R P)))) \vee$
 $\$ok' \wedge (true \vee (R1 \text{ true} ;; R1 (R2c (peri_R P)))) \diamond (R1 \text{ true} ;; R1 (R2c (post_R P))))$
 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-tac)
 thus ?thesis by simp
 qed
 finally show ?thesis .
 qed

lemma USUP-CSP-closed:

assumes $A \neq \{\}$ $\forall P \in A. P \text{ is CSP}$
 shows $(\sqcap A) \text{ is CSP}$
 proof –
 from assms have $A: A = CSP \cdot A$
 by (auto simp add: Healthy-def rev-image-eqI)
 also have $(\sqcap ...) = (\sqcap P \in A. CSP(P))$
 by auto
 also have ... = $(\sqcap P \in A \cdot CSP(P))$
 by (simp add: USUP-as-Sup-collect)
 also have ... = $(\sqcap P \in A \cdot RH((\neg P^f_f) \vdash P^t_f))$
 by (metis (no-types) CSP-RH-design-form)
 also have ... = $RH(\sqcap P \in A \cdot (\neg P^f_f) \vdash P^t_f)$
 by (simp add: RH-USUP assms(1))
 also have ... = $RH((\sqcup P \in A \cdot \neg P^f_f) \vdash (\sqcap P \in A \cdot P^t_f))$
 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 $A \neq \{\}$ $\forall P \in A. P \text{ is CSP}$

shows $(\sqcup A)$ is CSP
proof –
 from *assms* have $A: A = \text{CSP} \cdot A$
 by (*auto simp add: Healthy-def rev-image-eqI*)
 also have $(\sqcup \dots) = (\sqcup P \in A. \text{CSP}(P))$
 by *auto*
 also have $\dots = (\sqcup P \in A \cdot \text{CSP}(P))$
 by (*simp add: UINF-as-Inf-collect*)
 also have $\dots = (\sqcup P \in A \cdot \text{RH}((\neg P^f_f) \vdash P^t_f))$
 by (*simp add: CSP-RH-design-form*)
 also have $\dots = \text{RH}(\sqcup P \in A \cdot (\neg P^f_f) \vdash P^t_f)$
 by (*simp add: RH-UINF assms(1)*)
 also have $\dots = \text{RH}((\sqcap P \in A \cdot \neg P^f_f) \vdash (\sqcup P \in A \cdot \neg P^f_f \Rightarrow P^t_f))$
 by (*simp add: design-UINF*)
 also have $\dots = \text{CSP}(\dots)$
 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 $(\sqcap_R A)$ 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 $\sqcap_R A \sqsubseteq P$
 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 $P \sqsubseteq \sqcap_R A$
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 $\forall P \in A. P \text{ is CSP}$
shows $(\bigsqcup_R A) \text{ is CSP}$
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 $\forall Q \in A. Q \text{ is CSP } P \in A$
shows $P \sqsubseteq \bigsqcup_R A$
using *assms*
by (*auto simp add: reactive-inf-def Inf-lower*)

lemma *CSP-inf-lower-bound*:
assumes $\forall P \in A. P \text{ is CSP } \forall P \in A. P \sqsubseteq Q \text{ } Q \text{ is CSP}$
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-tac*)

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-tac*)

lemma *assigns-rea-comp-lemma*:
assumes $\$ok \nmid P \ \$wait \nmid P$
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 $\dots = (\langle \sigma \oplus_s \Sigma_R \rangle_a ;; P)$
 by (*simp add: assigns-lift-rea-unfold assigns-lift-des-unfold, rel-tac*)
 also have $\dots = (\lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger P)$
 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-tac*)

using *minus-zero-eq* **apply** *blast*

done

lemma *Healthy-if*: $P \text{ is } H \implies (H P = P)$

unfolding *Healthy-def* **by** *auto*

lemma *assigns-rea-comp*:

assumes $\$ok \# P \$ok \# Q_1 \$ok \# Q_2 \$wait \# P \$wait \# Q_1 \$wait \# Q_2$

$Q_1 \text{ is } R1 \ Q_2 \text{ is } R1 \ P \text{ is } R2s \ Q_1 \text{ is } R2s \ Q_2 \text{ is } R2s$

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 $\dots = RH ((\neg ((\$tr' =_u \$tr \wedge \lceil \langle \sigma \rangle_a \rceil_R) \wedge \neg \$wait' ;;$
 $R1 (\neg P))) \vdash \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_1 \diamond \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_2)$

by (*simp add: RH-tri-design-composition unrest assms R2s-true R1-false R1-R2s-frame Healthy-if assigns-rea-comp-lemma*)

also have $\dots = RH ((\neg ((\$tr' =_u \$tr \wedge \lceil \langle \sigma \rangle_a \rceil_R) \wedge \$wait' =_u \ll False \gg ;;$
 $R1 (\neg P))) \vdash \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_1 \diamond \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_2)$

by (*simp add: false-alt-def[THEN sym]*)

also have $\dots = RH ((\neg ((\$tr' =_u \$tr \wedge \lceil \langle \sigma \rangle_a \rceil_R) \ll false/\$wait' \gg ;;$
 $(R1 (\neg P)) \ll false/\$wait' \gg)) \vdash \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_1 \diamond \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_2)$

by (*simp add: seqr-left-one-point false-alt-def*)

also have $\dots = RH ((\neg ((\$tr' =_u \$tr \wedge \lceil \langle \sigma \rangle_a \rceil_R) ;; (R1 (\neg P)))) \vdash \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_1 \diamond \lceil \sigma \oplus_s \Sigma_R \rceil_s$
 $\dagger Q_2)$

by (*simp add: R1-def usubst unrest assms*)

also have $\dots = RH ((\neg \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger R1 (\neg P)) \vdash \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_1 \diamond \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_2)$

by (*simp add: assigns-rea-comp-lemma assms unrest*)

also have $\dots = RH ((\neg R1 (\neg \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger P)) \vdash \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_1 \diamond \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_2)$

by (*simp add: R1-def usubst unrest*)

also have $\dots = RH ((\lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger P) \vdash \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_1 \diamond \lceil \sigma \oplus_s \Sigma_R \rceil_s \dagger Q_2)$

by (*simp add: RH-design-R1-neg-precond*)

finally show *?thesis* .

qed

lemma *RH-design-par*:

assumes

$\$ok' \# P_1 \$wait \# P_1 \$ok' \# P_2 \$wait \# P_2$

$\$ok' \# Q_1 \$wait \# Q_1 \$ok' \# Q_2 \$wait \# Q_2$

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 $RH(P_1 \vdash Q_1) \parallel_R RH(P_2 \vdash Q_2) =$
 $RH ((\neg R1 (R2c (\neg P_1 \ll true/\$ok \gg)) \wedge \neg R1 (R2c (\neg P_2 \ll true/\$ok \gg))) \vdash$
 $(R1 (R2s (\$ok \wedge P_1 \Rightarrow Q_1)) \wedge R1 (R2s (\$ok \wedge P_2 \Rightarrow Q_2))))$

by (*simp add: rea-design-par-def rea-pre-RH-design RH-postcondition, simp add: usubst assms*)

also have $\dots =$

$RH ((P_1 \ll true/\$ok \gg \wedge P_2 \ll true/\$ok \gg) \vdash$
 $(R1 (R2s (\$ok \wedge P_1 \Rightarrow Q_1)) \wedge R1 (R2s (\$ok \wedge P_2 \Rightarrow Q_2))))$

by (*metis (no-types, hide-lams) R2c-and R2c-not RH-design-pre-R2c RH-design-pre-neg-conj-R1 double-negation*)

also have $\dots = RH ((P_1 \wedge P_2) \vdash (R1 (R2s (\$ok \wedge P_1 \Rightarrow Q_1)) \wedge R1 (R2s (\$ok \wedge P_2 \Rightarrow Q_2))))$

by (metis conj-pos-var-subst design-def subst-conj uvar-ok)
 also have ... = RH ((P₁ ∧ P₂) ⊢ (R1 (R2s ((\$ok ∧ P₁ ⇒ Q₁) ∧ (\$ok ∧ P₂ ⇒ Q₂))))))
 by (simp add: R1-conj R2s-conj)
 also have ... = RH ((P₁ ∧ P₂) ⊢ ((\$ok ∧ P₁ ⇒ Q₁) ∧ (\$ok ∧ P₂ ⇒ Q₂)))
 by (metis (mono-tags, lifting) RH-design-export-R1 RH-design-export-R2s)
 also have ... = RH ((P₁ ∧ P₂) ⊢ (Q₁ ∧ Q₂))
 by (rel-tac)
 finally show ?thesis .
 qed

lemma RH-tri-design-par:

assumes

\$ok' # P₁ \$wait # P₁ \$ok' # P₂ \$wait # P₂
 \$ok' # Q₁ \$wait # Q₁ \$ok' # Q₂ \$wait # Q₂
 \$ok' # R₁ \$wait # R₁ \$ok' # R₂ \$wait # R₂

shows RH(P₁ ⊢ Q₁ ◊ R₁) ||_R RH(P₂ ⊢ Q₂ ◊ R₂) = RH((P₁ ∧ P₂) ⊢ (Q₁ ∧ Q₂) ◊ (R₁ ∧ R₂))
 by (simp add: RH-design-par assms unrest wait'-cond-conj-exchange)

lemma RH-design-par-comm:

P ||_R Q = Q ||_R P

by (simp add: rea-design-par-def utp-pred.inf-commute)

lemma RH-design-par-zero:

assumes P is CSP

shows Chaos ||_R P = Chaos

proof –

have Chaos ||_R P = RH (false ⊢ true ◊ true) ||_R RH (pre_R(P) ⊢ peri_R(P) ◊ post_R(P))

by (simp add: Chaos-def CSP-reactive-tri-design assms)

also have ... = RH (false ⊢ peri_R P ◊ 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

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: $\top_{\text{utp-order RDES}} = \text{Miracle}$

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*: $\perp_{\text{utp-order } RDES} = \text{Chaos}$
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*($RDES \times ('t::\text{ordered-cancel-monoid-diff}, 'a) \text{ alphabet-rp}$)
rewrites *carrier* (*utp-order RDES*) = $\llbracket \text{CSP} \rrbracket$
and $\top_{\text{utp-order } RDES} = \text{Miracle}$
and $\perp_{\text{utp-order } RDES} = \text{Chaos}$
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* :: $- \Rightarrow - (\mu_R)$ **where**
 $\mu_R F \equiv \mu_{\text{utp-order } RDES} F$

abbreviation *rdes-gfp* :: $- \Rightarrow - (\nu_R)$ **where**
 $\nu_R F \equiv \nu_{\text{utp-order } RDES} F$

lemma *rdes-lfp-copy*: $\llbracket \text{mono } F; F \in \llbracket \text{CSP} \rrbracket \rightarrow \llbracket \text{CSP} \rrbracket \rrbracket \Longrightarrow \mu_R F = F (\mu_R F)$
by (*metis hrd-lattice.LFP-unfold mono-Monotone-utp-order*)

lemma *rdes-gfp-copy*: $\llbracket \text{mono } F; F \in \llbracket \text{CSP} \rrbracket \rightarrow \llbracket \text{CSP} \rrbracket \rrbracket \Longrightarrow \nu_R F = F (\nu_R F)$
by (*metis hrd-lattice.GFP-unfold mono-Monotone-utp-order*)

end