Isabelle/UTP: Mechanised reasoning for the UTP

Simon Foster Frank Zeyda

March 24, 2017

Contents

1 Parser Utilities **3**

2 UTP variables **4**

3 UTP expressions **7**
3.1 Evaluation laws for expressions 16
3.2 Misc laws 16
3.3 Literalise tactics 16

4 Unrestriction **17**

5 Substitution **19**
5.1 Substitution definitions 20
5.2 Substitution laws 21
5.3 Unrestriction laws 24

6 UTP Tactics **26**
6.1 Theorem Attributes 26
6.2 Generic Methods 26
6.3 Transfer Tactics 27
6.3.1 Robust Transfer 27
6.3.2 Faster Transfer 27
6.4 Interpretation 28
6.5 User Tactics 28

7 Alphabetised Predicates **30**
7.1 Predicate syntax 30
7.2 Predicate operators 31
7.3 Unrestriction Laws 35
7.4 Substitution Laws 37
7.5 Predicate Laws 38
7.6 Conditional laws 47
7.7 Cylindric algebra 48
7.8 Quantifier lifting 48

8 Alphabet manipulation **49**
8.1 Alphabet extension . . . 49
8.2 Alphabet restriction . . . 51
8.3 Alphabet lens laws . . . 52
8.4 Alphabet coercion . . . 52
8.5 Substitution alphabet extension . . . 52
8.6 Substitution alphabet restriction . . . 53

9 Lifting expressions **53**
9.1 Lifting definitions . . . 53
9.2 Lifting laws . . . 54
9.3 Unrestriction laws . . . 54

10 UTP Deduction Tactic **54**

11 Alphabetised relations **56**
11.1 Unrestriction Laws . . . 59
11.2 Substitution laws . . . 61
11.3 Relation laws . . . 62
11.4 Converse laws . . . 67
11.5 Assertions and assumptions . . . 69
11.6 Frame and antiframe . . . 69
11.7 Relational unrestriction . . . 71
11.8 Alphabet laws . . . 73
11.9 Relation algebra laws . . . 73
11.10 Relational alphabet extension . . . 73
11.11 Program values . . . 74
11.12 Relational Hoare calculus . . . 74
11.13 Weakest precondition calculus . . . 75

12 UTP Theories **76**
12.1 Complete lattice of predicates . . . 76
12.2 Healthiness conditions . . . 77
12.3 Properties of healthiness conditions . . . 77
12.4 UTP theories hierarchy . . . 80
12.5 UTP theory hierarchy . . . 81
12.6 Theory of relations . . . 88
12.7 Theory links . . . 89

13 Concurrent programming **90**

14 Relational operational semantics **94**
14.1 Variable blocks . . . 95

15 UTP Events **98**
15.1 Events . . . 98
15.2 Channels . . . 98
15.2.1 Operators . . . 99

16 Meta-theory for the Standard Core **99**

1 Parser Utilities

```
theory utp-parser-utils
imports
  Main
begin

syntax
  -id-string     :: id ⇒ string (IDSTR′(-′))

ML ⟪
signature UTP-PARSER-UTILS =
sig
  val mk-nib : int −> Ast.ast
  val mk-char : string −> Ast.ast
  val mk-string : string list −> Ast.ast
  val string-ast-tr : Ast.ast list −> Ast.ast
end;

structure Utp-Parser-Utils : UTP-PARSER-UTILS =
struct

val mk-nib =
  Ast.Constant o Lexicon.mark-const o
    fst o Term.dest-Const o HOLogic.mk-char;

fun mk-char s =
  if Symbol.is-ascii s then
    Ast.Appl [Ast.Constant @{const-syntax Char}, mk-nib (ord s div 16), mk-nib (ord s mod 16)]
  else error (Non−ASCII symbol:  ^ quote s);

fun mk-string [] = Ast.Constant @{const-syntax Nil}
  | mk-string (c :: cs) =
      Ast.Appl [Ast.Constant @{const-syntax List.Cons}, mk-char c, mk-string cs];

fun string-ast-tr [Ast.Variable str] =
      (case Lexicon.explode-str (str, Position.none) of
        [] =>
          Ast.Appl
            [Ast.Constant @{syntax-const -constrain},
              Ast.Constant @{const-syntax Nil}, Ast.Constant @{type-syntax string}]
      | ss => mk-string (map Symbol-Pos.symbol ss))
  | string-ast-tr [Ast.Appl [Ast.Constant @{syntax-const -constrain}, ast1, ast2]] =
      Ast.Appl [Ast.Constant @{syntax-const -constrain}, string-ast-tr [ast1], ast2]
  | string-ast-tr asts = raise Ast.AST (string-tr, asts);

end

signature NAME-UTILS =
sig
  val deep-unmark-const : term −> term
  val right-crop-by : int −> string −> string
  val last-char-str : string −> string
  val repeat-char : char −> int −> string
  val mk-id : string −> term
end;
```

```
structure Name-Utils : NAME-UTILS =
struct
  fun unmark-const-term (Const (name, typ)) =
    Const (Lexicon.unmark-const name, typ)
  | unmark-const-term term = term;

  val deep-unmark-const =
    (map-aterms unmark-const-term);

  fun right-crop-by n s =
    String.substring (s, 0, (String.size s) − n);

  fun last-char-str s =
    String.str (String.sub (s, (String.size s) − 1));

  fun repeat-char c n =
    if n > 0 then (String.str c) ^ (repeat-char c (n − 1)) else ;

  fun mk-id name = Free (name, dummyT);
end;
⟫
```

parse-translation ⟪

```
let
  fun id-string-tr [Free (full-name, -)] = HOLogic.mk-string full-name
    | id-string-tr [Const (full-name, -)] = HOLogic.mk-string full-name
    | id-string-tr - = raise Match;
in
  [(@{syntax-const -id-string}, K id-string-tr)]
end
⟫
```

end

2 UTP variables

theory *utp-var*
  imports
  Deriv
  ~~/src/HOL/Library/Prefix-Order
  ~~/src/HOL/Library/Char-ord
  ~~/src/Tools/Adhoc-Overloading
  ~~/src/HOL/Library/Monad-Syntax
  ~~/src/HOL/Library/Countable
  ~~/src/HOL/Eisbach/Eisbach
  ../contrib/Algebra/Complete-Lattice
  ../contrib/Algebra/Galois-Connection
  ../optics/Lenses
  ../utils/Profiling
  ../utils/TotalRecall
  ../utils/Library-extra/Pfun
  ../utils/Library-extra/Ffun
  ../utils/Library-extra/List-lexord-alt
  ../utils/Library-extra/Monoid-extra
  utp-parser-utils

begin

We will overload the square order relation with refinement and also the lattice operators so we will turn off these notations.

purge-notation
le (**infixl** $\sqsubseteq_1$ *50*) **and**
asup ($\bigsqcup_1$- [*90*] *90*) **and**
ainf ($\bigsqcap_1$- [*90*] *90*) **and**
join (**infixl** $\sqcup_1$ *65*) **and**
meet (**infixl** $\sqcap_1$ *70*)

We hide HOL's built-in relation type since we will replace it with our own

hide-type *rel*
type-synonym $'a$ *relation* = $('a \times 'a)$ *set*

declare *fst-vwb-lens* [*simp*]
declare *snd-vwb-lens* [*simp*]

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

This theory describes the foundational structure of UTP variables, upon which the rest of our model rests. We start by defining alphabets, which following [2, 3] 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 in this frame are simply modelled as lenses, where the view type $'a$ is the variable type, and the source type $'\alpha$ is the state-space type.

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

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

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

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

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

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

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

lemma *out-var-semi-uvar* [*simp*]:

mwb-lens x $\Longrightarrow$ *mwb-lens* (*out-var x*)
by (*simp add*: *comp-mwb-lens out-var-def*)

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

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

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

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

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

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

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

We also define some lookup abstraction simplifications.

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

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

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

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

Variables can also be used to effectively define sets of variables. Here we define the the universal alphabet (Σ) to be a variable with identity for both the lookup and update functions. Effectively this is just a function directly on the alphabet type.

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

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

syntax

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

consts
svar :: ′*v* ⇒ ′*e*
ivar :: ′*v* ⇒ ′*e*
ovar :: ′*v* ⇒ ′*e*

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

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

Syntactic function to construct a uvar type given a return type

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

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

3 UTP expressions

theory *utp-expr*
imports
utp-var
begin

purge-notation *BNF-Def.convol* (⟨(-,/ -)⟩)

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

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

typedef $('t, '\alpha)$ *uexpr* = *UNIV* :: $('\alpha$ *alphabet* $\Rightarrow 't)$ *set* **..**

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

lemma *uexpr-eq-iff*:
  $e = f \longleftrightarrow (\forall\ b.\ [\![e]\!]_e\ b = [\![f]\!]_e\ b)$
  using *Rep-uexpr-inject*[*of e f*, *THEN sym*] **by** (*auto*)

named-theorems *ueval* **and** *lit-simps*

setup-lifting *type-definition-uexpr*

Get the alphabet of an expression

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

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

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

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

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

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

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

We also define a UTP expression version of function abstract

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

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

consts

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

adhoc-overloading
ulit lit

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

translations
-uuvar x == *CONST var x*

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

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

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

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

end

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

Instantiating uminus also provides negation for predicates later

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

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

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

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

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

instantiation *uexpr* :: (*inverse*, *type*) *inverse*
begin
  definition *inverse-uexpr* :: ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr*
  where *inverse-uexpr u* = *uop inverse u*
instance ..
end

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

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

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

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 cancel-ab-semigroup-add-class.diff-diff-add*)+)

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

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

instantiation *uexpr* :: (*ord*, *type*) *ord*
begin
lift-definition *less-eq-uexpr* :: ($'a$, $'b$) *uexpr* $\Rightarrow$ ($'a$, $'b$) *uexpr* $\Rightarrow$ *bool*
is λ P Q. ($\forall$ A. P $A \leq Q$ A) **.**
definition *less-uexpr* :: ($'a$, $'b$) *uexpr* $\Rightarrow$ ($'a$, $'b$) *uexpr* $\Rightarrow$ *bool*
where *less-uexpr* P $Q = (P \leq Q \wedge \neg\ Q \leq P)$
instance ..
end

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

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

instance *uexpr* :: (*ordered-ab-group-add-abs*, *type*) *ordered-ab-group-add-abs*
apply (*intro-classes*)
apply (*simp add*: *abs-uexpr-def zero-uexpr-def plus-uexpr-def uminus-uexpr-def*, *transfer*, *simp add*:
abs-ge-self abs-le-iff abs-triangle-ineq)+
apply (*metis ab-group-add-class.ab-diff-conv-add-uminus abs-ge-minus-self abs-ge-self add-mono-thms-linordered-semiri*
done

lemma *uexpr-diff-zero* [*simp*]:
fixes a :: ($'\alpha$::*ordered-cancel-monoid-diff*, $'a$) *uexpr*
shows $a - 0 = a$
by (*simp add*: *minus-uexpr-def zero-uexpr-def*, *transfer*, *auto*)

lemma *uexpr-add-diff-cancel-left* [*simp*]:
fixes a b :: ($'\alpha$::*ordered-cancel-monoid-diff*, $'a$) *uexpr*
shows $(a + b) - a = b$
by (*simp add*: *minus-uexpr-def plus-uexpr-def*, *transfer*, *auto*)

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

instance *uexpr* :: (*ring-1*, *type*) *ring-1*
by (*intro-classes*) (*simp add*: *plus-uexpr-def uminus-uexpr-def minus-uexpr-def times-uexpr-def zero-uexpr-def*

one-uexpr-def, *transfer*, *simp add*: *fun-eq-iff*)+

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

Set up automation for numerals

lemma *numeral-uexpr-rep-eq*: $[\![$*numeral x*$]\!]_e$ *b* = *numeral x*
apply (*induct x*)
apply (*simp add*: *lit.rep-eq one-uexpr-def*)
apply (*simp add*: *bop.rep-eq numeral-Bit0 plus-uexpr-def*)
apply (*simp add*: *bop.rep-eq lit.rep-eq numeral-code*(*3*) *one-uexpr-def plus-uexpr-def*)
done

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

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

adhoc-overloading
  ueq eq-upred

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

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

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

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

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

syntax
  -ucoerce :: ('*a*, 'α) *uexpr* ⇒ *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** $\hat{}_u$ 80)
-ulast :: ('a list, 'α) uexpr ⇒ ('a, 'α) uexpr ($last_u$'(-'))
-ufront :: ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr ($front_u$'(-'))
-uhead :: ('a list, 'α) uexpr ⇒ ('a, 'α) uexpr ($head_u$'(-'))
-utail :: ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr ($tail_u$'(-'))
-utake :: (nat, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr ($take_u$'(-,/ -'))
-udrop :: (nat, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr ($drop_u$'(-,/ -'))
-ucard :: ('a list, 'α) uexpr ⇒ (nat, 'α) uexpr ($\#_u$'(-'))
-ufilter :: ('a list, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a list, 'α) uexpr (**infixl** $\upharpoonleft_u$ 75)
-uextract :: ('a set, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (**infixl** $\upharpoonright_u$ 75)
-uelems :: ('a list, 'α) uexpr ⇒ ('a set, 'α) uexpr ($elems_u$'(-'))
-usorted :: ('a list, 'α) uexpr ⇒ (bool, 'α) uexpr ($sorted_u$'(-'))
-udistinct :: ('a list, 'α) uexpr ⇒ (bool, 'α) uexpr ($distinct_u$'(-'))
-uless :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** $<_u$ 50)
-uleq :: ('a, 'α) uexpr ⇒ ('a, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** $\le_u$ 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** $\ge_u$ 50)
-umin :: logic ⇒ logic ⇒ logic (min_u'(-, -'))
-umax :: logic ⇒ logic ⇒ logic (max_u'(-, -'))
-ugcd :: logic ⇒ logic ⇒ logic (gcd_u'(-, -'))
-uceil :: logic ⇒ logic ($\lceil$-$\rceil_u$)
-ufloor :: logic ⇒ logic ($\lfloor$-$\rfloor_u$)
-ufinite :: logic ⇒ logic ($finite_u$'(-'))
-uempset :: ('a set, 'α) uexpr ($\{\}_u$)
-uset :: args => ('a set, 'α) uexpr ($\{(\text{-})\}_u$)
-uunion :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr (**infixl** $\cup_u$ 65)
-uinter :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr (**infixl** $\cap_u$ 70)
-umem :: ('a, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** $\in_u$ 50)
-usubset :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** $\subset_u$ 50)
-usubseteq :: ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** $\subseteq_u$ 50)
-utuple :: ('a, 'α) uexpr ⇒ utuple-args ⇒ ('a * 'b, 'α) uexpr ((1'(-,/ -')$_u$))
-utuple-arg :: ('a, 'α) uexpr ⇒ utuple-args (-)
-utuple-args :: ('a, 'α) uexpr => utuple-args ⇒ utuple-args (-,/ -)
-uunit :: ('a, 'α) uexpr ('(')$_u$)
-ufst :: ('a × 'b, 'α) uexpr ⇒ ('a, 'α) uexpr (π_1'(-'))
-usnd :: ('a × 'b, 'α) uexpr ⇒ ('b, 'α) uexpr (π_2'(-'))
-uapply :: ('a ⇒ 'b, 'α) uexpr ⇒ utuple-args ⇒ ('b, 'α) uexpr (-$(\!|$-$|\!)_u$ [999,0] 999)
-ulamba :: pttrn ⇒ logic ⇒ logic (λ - • - [0, 10] 10)
-udom :: logic ⇒ logic (dom_u'(-'))
-uran :: logic ⇒ logic (ran_u'(-'))
-uinl :: logic ⇒ logic (inl_u'(-'))
-uinr :: logic ⇒ logic (inr_u'(-'))
-umap-empty :: logic ($[]_u$)
-umap-plus :: logic ⇒ logic ⇒ logic (**infixl** $\oplus_u$ 85)
-umap-minus :: logic ⇒ logic ⇒ logic (**infixl** $\ominus_u$ 85)
-udom-res :: logic ⇒ logic ⇒ logic (**infixl** $\lhd_u$ 85)
-uran-res :: logic ⇒ logic ⇒ logic (**infixl** $\rhd_u$ 85)
-umaplet :: [logic, logic] => umaplet (- /↦/ -)
:: umaplet => umaplets (-)
-UMaplets :: [umaplet, umaplets] => umaplets (-,/ -)
-UMapUpd :: [logic, umaplets] => logic (-/'(-')$_u$ [900,0] 900)
-UMap :: umaplets => logic ((1[-]$_u$))

translations
$f(\!|v|\!)_u$ <= *CONST uapply f v*

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

translations
$x :_u {}'a == x :: ('a,\ \text{-})\ uexpr$
$\langle\rangle == \ll[]\gg$
$\langle x,\ xs\rangle == CONST\ bop\ (op\ \#)\ x\ \langle xs\rangle$
$\langle x\rangle == CONST\ bop\ (op\ \#)\ x\ \ll[]\gg$
$x\ \hat{}_u\ y == CONST\ bop\ (op\ @)\ x\ y$
$last_u(xs) == CONST\ uop\ CONST\ last\ xs$
$front_u(xs) == CONST\ uop\ CONST\ butlast\ xs$
$head_u(xs) == CONST\ uop\ CONST\ hd\ xs$
$tail_u(xs) == CONST\ uop\ CONST\ tl\ xs$
$drop_u(n,xs) == CONST\ bop\ CONST\ drop\ n\ xs$
$take_u(n,xs) == CONST\ bop\ CONST\ take\ n\ xs$
$\#_u(xs) == CONST\ uop\ CONST\ ucard\ xs$
$elems_u(xs) == CONST\ uop\ CONST\ set\ xs$
$sorted_u(xs) == CONST\ uop\ CONST\ sorted\ xs$
$distinct_u(xs) == CONST\ uop\ CONST\ distinct\ xs$
$xs \upharpoonleft_u A == CONST\ bop\ CONST\ seq\text{-}filter\ xs\ A$
$A \upharpoonright_u xs == CONST\ bop\ (op \upharpoonright_l)\ A\ xs$
$x <_u y == CONST\ bop\ (op <)\ x\ y$
$x \le_u y == CONST\ bop\ (op \le)\ x\ y$
$x >_u y == y <_u x$
$x \ge_u y == y \le_u x$
$min_u(x,\ y) == CONST\ bop\ (CONST\ min)\ x\ y$
$max_u(x,\ y) == CONST\ bop\ (CONST\ max)\ x\ y$
$gcd_u(x,\ y) == CONST\ bop\ (CONST\ gcd)\ x\ y$
$\lceil x\rceil_u == CONST\ uop\ CONST\ ceiling\ x$
$\lfloor x\rfloor_u == CONST\ uop\ CONST\ floor\ x$
$finite_u(x) == CONST\ uop\ (CONST\ finite)\ x$
$\{\}_u == \ll\{\}\gg$
$\{x,\ xs\}_u == CONST\ bop\ (CONST\ insert)\ x\ \{xs\}_u$
$\{x\}_u == CONST\ bop\ (CONST\ insert)\ x\ \ll\{\}\gg$
$A \cup_u B == CONST\ bop\ (op \cup)\ A\ B$
$A \cap_u B == CONST\ bop\ (op \cap)\ A\ B$
$f \oplus_u g => (f :: ((\text{-},\ \text{-})\ pfun,\ \text{-})\ uexpr) + g$
$f \ominus_u g => (f :: ((\text{-},\ \text{-})\ pfun,\ \text{-})\ uexpr) - g$
$x \in_u A == CONST\ bop\ (op \in)\ x\ A$
$A \subset_u B == CONST\ bop\ (op <)\ A\ B$
$A \subset_u B$ <= $CONST\ bop\ (op \subset)\ A\ B$
$f \subset_u g$ <= $CONST\ bop\ (op \subset_p)\ f\ g$
$f \subset_u g$ <= $CONST\ bop\ (op \subset_f)\ f\ g$
$A \subseteq_u B == CONST\ bop\ (op \le)\ A\ B$
$A \subseteq_u B$ <= $CONST\ bop\ (op \subseteq)\ A\ B$
$f \subseteq_u g$ <= $CONST\ bop\ (op \subseteq_p)\ f\ g$
$f \subseteq_u g$ <= $CONST\ bop\ (op \subseteq_f)\ f\ g$
$()_u == \ll()\gg$
$(x,\ y)_u == CONST\ bop\ (CONST\ Pair)\ x\ y$
$\text{-}utuple\ x\ (\text{-}utuple\text{-}args\ y\ z) == \text{-}utuple\ x\ (\text{-}utuple\text{-}arg\ (\text{-}utuple\ y\ z))$
$\pi_1(x) == CONST\ uop\ CONST\ fst\ x$

$\pi_2(x)$ == *CONST uop CONST snd x*
$f(|x|)_u$ == *CONST bop CONST uapply f x*
$\lambda\ x \cdot p$ == *CONST ulambda* (λ *x. p*)
$dom_u(f)$ == *CONST uop CONST udom f*
$ran_u(f)$ == *CONST uop CONST uran f*
$inl_u(x)$ == *CONST uop CONST Inl x*
$inr_u(x)$ == *CONST uop CONST Inr x*
$[]_u$ == «*CONST uempty*»
$A \lhd_u f$ == *CONST bop* (*CONST udomres*) *A f*
$f \rhd_u A$ == *CONST bop* (*CONST uranres*) *f A*
-UMapUpd m (*-UMaplets xy ms*) == *-UMapUpd* (*-UMapUpd m xy*) *ms*
-UMapUpd m (*-umaplet x y*) == *CONST trop CONST uupd m x y*
-UMap ms == *-UMapUpd* $[]_u$ *ms*
-UMap (*-UMaplets ms1 ms2*) <= *-UMapUpd* (*-UMap ms1*) *ms2*
-UMaplets ms1 (*-UMaplets ms2 ms3*) <= *-UMaplets* (*-UMaplets ms1 ms2*) *ms3*
$f(|x,y|)_u$ == *CONST bop CONST uapply f* $(x,y)_u$

Lifting set intervals

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

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

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

Lifting limits

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

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

translations
$lim_u(x \to p^-)(e)$ == *CONST bop CONST ulim-left p* ($\lambda\ x \cdot e$)
$lim_u(x \to p^+)(e)$ == *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-uexpr-def
minus-uexpr-def

times-uexpr-def
inverse-uexpr-def
divide-uexpr-def
sgn-uexpr-def
abs-uexpr-def
mod-uexpr-def
eq-upred-def
numeral-uexpr-simp
ulim-left-def
ulim-right-def
ucont-on-def
LNil-def
LZero-def
plus-list-def

3.1 Evaluation laws for expressions

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

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

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

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

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

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

declare *uexpr-defs* [*ueval*]

3.2 Misc laws

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

3.3 Literalise tactics

The following tactic converts literal HOL expressions to UTP expressions and vice-versa via a collection of simplification rules. The two tactics are called "literalise", which converts UTP to expressions to HOL expressions – i.e. it pushes them into literals – and unliteralise that reverses this. We collect the equations in a theorem attribute called "lit_simps".

lemma *lit-num-simps* [*lit-simps*]: $\ll 0 \gg = 0$ $\ll 1 \gg = 1$ $\ll numeral\ n \gg = numeral\ n$ $\ll -\ x \gg = -\ \ll x \gg$
by (*simp-all add*: *ueval*, *transfer*, *simp*)

lemma *lit-arith-simps* [*lit-simps*]:
$\ll -\ x \gg = -\ \ll x \gg$
$\ll x + y \gg = \ll x \gg + \ll y \gg$ $\ll x - y \gg = \ll x \gg - \ll y \gg$
$\ll x * y \gg = \ll x \gg * \ll y \gg$ $\ll x\ /\ y \gg = \ll x \gg\ /\ \ll y \gg$

$\ll x\ div\ y \gg = \ll x \gg\ div\ \ll y \gg$
by (*simp add*: *uexpr-defs*, *transfer*, *simp*)+

lemma *lit-fun-simps* [*lit-simps*]:
$\ll i\ x\ y\ z\ u \gg = qtop\ i\ \ll x \gg\ \ll y \gg\ \ll z \gg\ \ll u \gg$
$\ll h\ x\ y\ z \gg = trop\ h\ \ll x \gg\ \ll y \gg\ \ll z \gg$
$\ll g\ x\ y \gg = bop\ g\ \ll x \gg\ \ll y \gg$
$\ll f\ x \gg = uop\ f\ \ll x \gg$
by (*transfer*, *simp*)+

In general unliteralising converts function applications to corresponding expression liftings. Since some operators, like + and *, have specific operators we also have to use $\alpha(?e) = 1_L$
$0 = \ll 0::?'a \gg$
$1 = \ll 1::?'a \gg$
$?u + ?v = bop\ op + ?u\ ?v$
$- ?u = uop\ uminus\ ?u$
$?u - ?v = bop\ op - ?u\ ?v$
$?u * ?v = bop\ op * ?u\ ?v$
$inverse\ ?u = uop\ inverse\ ?u$
$?u\ div\ ?v = bop\ op\ div\ ?u\ ?v$
$sgn\ ?u = uop\ sgn\ ?u$
$|?u| = uop\ abs\ ?u$
$?u\ mod\ ?v = bop\ op\ mod\ ?u\ ?v$
$(?x =_u ?y) = bop\ op = ?x\ ?y$
$numeral\ ?x = \ll numeral\ ?x \gg$
$ulim\text{-}left = (\lambda p.\ Lim\ (at\text{-}left\ p))$
$ulim\text{-}right = (\lambda p.\ Lim\ (at\text{-}right\ p))$
$ucont\text{-}on = (\lambda f\ A.\ continuous\text{-}on\ A\ f)$
$uempty = []$
$uempty = (0::?'a)$
$op + = op$ @ in reverse to correctly interpret these. Moreover, numerals must be handled separately by first simplifying them and then converting them into UTP expression numerals; hence the following two simplification rules.

lemma *lit-numeral-1*: $uop\ numeral\ x = Abs\text{-}uexpr\ (\lambda b.\ numeral\ ([\![x]\!]_e\ b))$
by (*simp add*: *uop-def*)

lemma *lit-numeral-2*: $Abs\text{-}uexpr\ (\lambda\ b.\ numeral\ v) = numeral\ v$
by (*metis lit.abs-eq lit-num-simps*(*3*))

method *literalise* = (*unfold lit-simps*[*THEN sym*])
method *unliteralise* = (*unfold lit-simps uexpr-defs*[*THEN sym*];
(*unfold lit-numeral-1* ; (*unfold ueval*); (*unfold lit-numeral-2*))?)+
end

4 Unrestriction

theory *utp-unrest*
imports *utp-expr*

begin

Unrestriction is an encoding of semantic freshness, that allows us to reason about the presence of variables in predicates without being concerned with abstract syntax trees. An expression p is unrestricted by variable x, written $x \sharp p$, if altering the value of x has no effect on the valuation of p. This is a sufficient notion to prove many laws that would ordinarily rely on an *fv* function.

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

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

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

named-theorems *unrest*

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

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

adhoc-overloading
  unrest unrest-upred

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

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

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

lemma *unrest-var* [*unrest*]: $\llbracket$ *vwb-lens x*; $x \bowtie y$ $\rrbracket \Longrightarrow y \sharp$ *var x*
  by (*transfer*, *auto*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5 Substitution

theory *utp-subst*
imports
utp-expr

utp-unrest
begin

5.1 Substitution definitions

We introduce a polymorphic constant that will be used to represent application of a substitution, and also a set of theorems to represent laws.

consts
$usubst :: 's \Rightarrow 'a \Rightarrow 'b$ (**infixr** † *80*)

named-theorems *usubst*

A substitution is simply a transformation on the alphabet; it shows how variables should be mapped to different values.

type-synonym $('\alpha,'\beta)\ psubst = '\alpha\ alphabet \Rightarrow '\beta\ alphabet$
type-synonym $'\alpha\ usubst = '\alpha\ alphabet \Rightarrow '\alpha\ alphabet$

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

adhoc-overloading
usubst subst

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

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

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

adhoc-overloading
subst-upd subst-upd-uvar

Lookup the expression associated with a variable in a substitution

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

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

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

adhoc-overloading
unrest unrest-usubst

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

syntax
```
-smaplet  :: [salpha, 'a] => smaplet             (- /↦s/ -)
          :: smaplet => smaplets               (-)
-SMaplets :: [smaplet, smaplets] => smaplets (-,/ -)
-SubstUpd :: ['m usubst, smaplets] => 'm usubst (-/'(-') [900,0] 900)
-Subst    :: smaplets => 'a ⇀ 'b             ((1[-]))
```

translations
```
-SubstUpd m (-SMaplets xy ms)      == -SubstUpd (-SubstUpd m xy) ms
```

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

Deletion of a substitution maplet

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

5.2 Substitution laws

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

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

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

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

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

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

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

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

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

done

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

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

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

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

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

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

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

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

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

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

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

lemma *usubst-ulambda* [*usubst*]: $\sigma \dagger (\lambda\ x \cdot P(x)) = (\lambda\ x \cdot \sigma \dagger P(x))$
  by (*transfer*, *simp*)

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

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

definition *var-name-ord* :: $('a, '\alpha)$ *uvar* $\Rightarrow$ $('b, '\alpha)$ *uvar* $\Rightarrow$ *bool* **where**

[*no-atp*]: *var-name-ord x y* = *True*

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

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

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

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

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

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

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

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

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

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

lemma *subst-mod* [*usubst*]: $\sigma \dagger (x \textit{ mod } y) = \sigma \dagger x \textit{ mod } \sigma \dagger y$
by (*simp add*: *mod-uexpr-def usubst*)

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

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

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

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

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

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

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

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

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

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

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

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

syntax
-psubst :: [*logic*, *svars*, *uexprs*] ⇒ *logic*
-subst :: *logic* ⇒ *uexprs* ⇒ *salphas* ⇒ *logic* ((-⟦-'/-⟧) [*999*,*0*,*0*] *1000*)
-uexprs :: [*logic*, *uexprs*] => *uexprs* (-,/ -)
:: *logic* => *uexprs* (-)
-svars :: [*svar*, *svars*] => *svars* (-,/ -)
:: *svar* => *svars* (-)
-salphas :: [*salpha*, *salphas*] => *salphas* (-,/ -)
:: *salpha* => *salphas* (-)

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

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

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

5.3 Unrestriction laws

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

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

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

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

6 UTP Tactics

theory *utp-tactics*
imports *Eisbach Lenses Interp utp-expr utp-unrest*
keywords *update-uexpr-rep-eq-thms* :: *thy-decl*
begin

In this theory, we define several automatic proof tactics that use transfer techniques to re-interpret proof goals about UTP predicates and relations in terms of pure HOL conjectures. The fundamental tactics to achieve this are *pred-simp* and *rel-simp*; a more detailed explanation of their behaviour is given below. The tactics can be given optional arguments to fine-tune their behaviour. By default, they use a weaker but faster form of transfer using rewriting; the option *robust*, however, forces them to use the slower but more powerful transfer of Isabelle's lifting package. A second option *no-interp* suppresses the re-interpretation of state spaces in order to eradicate record for tuple types prior to automatic proof.

In addition to *pred-simp* and *rel-simp*, we also provide the tactics *pred-auto* and *rel-auto*, as well as *pred-blast* and *rel-blast*; they, in essence, sequence the simplification tactics with the methods *auto* and *blast*, respectively.

6.1 Theorem Attributes

The following named attributes have to be introduced already here since our tactics must be able to see them. Note that we do not want to import the theories *utp-pred* and *utp-rel* here, so that both can potentially already make use of the tactics we define in this theory.

named-theorems *upred-defs upred definitional theorems*
named-theorems *urel-defs urel definitional theorems*

6.2 Generic Methods

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

Generic Predicate Tactics

method *gen-pred-tac* **methods** *transfer-tac interp-tac prove-tac* = (
  ((*unfold upred-defs*) [*1*])?;
  (*transfer-tac*),
  (*simp add*: *fun-eq-iff*
    lens-defs upred-defs alpha-splits Product-Type.split-beta)?,
  (*interp-tac*)?);
  (*prove-tac*)

Generic Relational Tactics

method *gen-rel-tac* **methods** *transfer-tac interp-tac prove-tac* = (
  ((*unfold upred-defs urel-defs*) [*1*])?;
  (*transfer-tac*),

(*simp add*: *fun-eq-iff relcomp-unfold OO-def*
  lens-defs upred-defs alpha-splits Product-Type.split-beta)?,
(*interp-tac*)?);
(*prove-tac*)

6.3 Transfer Tactics

Next, we define the component tactics used for transfer.

6.3.1 Robust Transfer

Robust transfer uses the transfer method of the lifting package.

method *slow-uexpr-transfer* = (*transfer*)

6.3.2 Faster Transfer

Fast transfer side-steps the use of the (*transfer*) method in favour of plain rewriting with the underlying *rep-eq-*... laws of lifted definitions. For moderately complex terms, surprisingly, the transfer step turned out to be a bottle-neck in some proofs; we observed that faster transfer resulted in a speed-up of approximately 30% when building the UTP theory heaps. On the downside, tactics using faster transfer do not always work but merely in about 95% of the cases. The approach typically works well when proving predicate equalities and refinements conjectures.

A known limitation is that the faster tactic, unlike lifting transfer, does not turn free variables into meta-quantified ones. This can, in some cases, interfere with the interpretation step and cause subsequent application of automatic proof tactics to fail. A fix is in progress [TODO].

Attribute Setup We first configure a dynamic attribute *uexpr-rep-eq-thms* to automatically collect all *rep-eq-* laws of lifted definitions on the *uexpr* type.

ML-file *uexpr-rep-eq.ML*

setup ⟪
  Global-Theory.add-thms-dynamic (@{*binding uexpr-rep-eq-thms*},
    uexpr-rep-eq.get-uexpr-rep-eq-thms o Context.theory-of)
⟫

We next configure a command **update-uexpr-rep-eq-thms** in order to update the content of the *uexpr-rep-eq-thms* attribute. Although the relevant theorems are collected automatically, for efficiency reasons, the user has to manually trigger the update process. The command must hence be executed whenever new lifted definitions for type *uexpr* are created. The updating mechanism uses **find-theorems** under the hood.

ML ⟪
  Outer-Syntax.command @{*command-keyword update-uexpr-rep-eq-thms*}
    reread and update content of the uexpr-rep-eq-thms attribute
    (*Scan.succeed* (*Toplevel.theory uexpr-rep-eq.read-uexpr-rep-eq-thms*));
⟫

update-uexpr-rep-eq-thms — Read *uexpr-rep-eq-thms* here.

Lastly, we require several named-theorem attributes to record the manual transfer laws and extra simplifications, so that the user can dynamically extend them in child theories.

named-theorems *uexpr-transfer-laws uexpr transfer laws*

declare *uexpr-eq-iff* [*uexpr-transfer-laws*]
named-theorems *uexpr-transfer-extra extra simplifications for uexpr transfer*

declare *unrest-upred.rep-eq* [*uexpr-transfer-extra*]
utp-expr.numeral-uexpr-rep-eq [*uexpr-transfer-extra*]
utp-expr.less-eq-uexpr.rep-eq [*uexpr-transfer-extra*]
Abs-uexpr-inverse [*simplified*, *uexpr-transfer-extra*]
Rep-uexpr-inverse [*uexpr-transfer-extra*]

Tactic Definition We have all ingredients now to define the fast transfer tactic as a single simplification step.

method *fast-uexpr-transfer* =
(*simp add*: *uexpr-transfer-laws uexpr-rep-eq-thms uexpr-transfer-extra*)

6.4 Interpretation

The interpretation of record state spaces as products is done using the laws provided by the utility theory *Interp*. Note that this step can be suppressed by using the *no-interp* option.

method *uexpr-interp-tac* = (*simp add*: *lens-interp-laws*)*?*

6.5 User Tactics

In this section, we finally set-up the six user tactics: *pred-simp*, *rel-simp*, *pred-auto*, *rel-auto*, *pred-blast* and *rel-blast*. For this, we first define the proof strategies that are to be applied *after* the transfer steps.

method *utp-simp-tac* = (*clarsimp*)*?*
method *utp-auto-tac* = ((*clarsimp*)*?*; *auto*)
method *utp-blast-tac* = ((*clarsimp*)*?*; *blast*)

The ML file below provides ML constructor functions for tactics that process arguments suitable and invoke the generic methods *gen-pred-tac* and *gen-rel-tac* with suitable arguments.

ML-file *utp-tactics.ML*

Finally, we execute the relevant outer commands for method setup. Sadly, this cannot be done at the level of Eisbach since the latter does not provide a convenient mechanism to process symbolic flags as arguments. It may be worth to put in a feature request with the developers of the Eisbach tool.

```
method-setup pred-simp = ‹‹
  (Scan.lift UTP-Tactics.scan-args) >>
  (fn args => fn ctx =>
    let val prove-tac = Basic-Tactics.utp-simp-tac in
      (UTP-Tactics.inst-gen-pred-tac args prove-tac ctx)
    end);
››

method-setup rel-simp = ‹‹
  (Scan.lift UTP-Tactics.scan-args) >>
    (fn args => fn ctx =>
      let val prove-tac = Basic-Tactics.utp-simp-tac in
        (UTP-Tactics.inst-gen-rel-tac args prove-tac ctx)
```

```
    end);
⟫

method-setup pred-auto = ⟪
  (Scan.lift UTP-Tactics.scan-args) >>
    (fn args => fn ctx =>
      let val prove-tac = Basic-Tactics.utp-auto-tac in
        (UTP-Tactics.inst-gen-pred-tac args prove-tac ctx)
      end);
⟫

method-setup rel-auto = ⟪
  (Scan.lift UTP-Tactics.scan-args) >>
    (fn args => fn ctx =>
      let val prove-tac = Basic-Tactics.utp-auto-tac in
        (UTP-Tactics.inst-gen-rel-tac args prove-tac ctx)
      end);
⟫

method-setup pred-blast = ⟪
  (Scan.lift UTP-Tactics.scan-args) >>
    (fn args => fn ctx =>
      let val prove-tac = Basic-Tactics.utp-blast-tac in
        (UTP-Tactics.inst-gen-pred-tac args prove-tac ctx)
      end);
⟫

method-setup rel-blast = ⟪
  (Scan.lift UTP-Tactics.scan-args) >>
    (fn args => fn ctx =>
      let val prove-tac = Basic-Tactics.utp-blast-tac in
        (UTP-Tactics.inst-gen-rel-tac args prove-tac ctx)
      end);
⟫
```

end

7 Alphabetised Predicates

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

An alphabetised predicate is a simply a boolean valued expression

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

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

7.1 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.

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

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

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

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

nonterminal *idt-list*

syntax
  -idt-el :: *idt* ⇒ *idt-list* (-)
  -idt-list :: *idt* ⇒ *idt-list* ⇒ *idt-list* ((-,/ -) [*0*, *1*])
  -uex :: *salpha* ⇒ *logic* ⇒ *logic* (∃ - • - [*0*, *10*] *10*)
  -uall :: *salpha* ⇒ *logic* ⇒ *logic* (∀ - • - [*0*, *10*] *10*)
  -ushEx :: *idt-list* ⇒ *logic* ⇒ *logic* (∃ - • - [*0*, *10*] *10*)
  -ushAll :: *idt-list* ⇒ *logic* ⇒ *logic* (∀ - • - [*0*, *10*] *10*)
  -ushBEx :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∃ - ∈ - • - [*0*, *0*, *10*] *10*)
  -ushBAll :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∀ - ∈ - • - [*0*, *0*, *10*] *10*)
  -ushGAll :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∀ - | - • - [*0*, *0*, *10*] *10*)
  -ushGtAll :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∀ - > - • - [*0*, *0*, *10*] *10*)
  -ushLtAll :: *idt* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∀ - < - • - [*0*, *0*, *10*] *10*)

translations
  -uex x P == *CONST uex x P*
  -uall x P == *CONST uall x P*
  -ushEx (*-idt-el x*) *P* == *CONST ushEx* (λ *x*. *P*)
  -ushEx (*-idt-list x y*) *P* => *CONST ushEx* (λ *x*. (*-ushEx y P*))
  ∃ *x* ∈ *A* • *P* => ∃ *x* • «*x*» $\in_u$ *A* ∧ *P*
  -ushAll (*-idt-el x*) *P* == *CONST ushAll* (λ *x*. *P*)
  -ushAll (*-idt-list x y*) *P* => *CONST ushAll* (λ *x*. (*-ushAll y P*))
  ∀ *x* ∈ *A* • *P* => ∀ *x* • «*x*» $\in_u$ *A* ⇒ *P*
  ∀ *x* | *P* • *Q* => ∀ *x* • *P* ⇒ *Q*
  ∀ *x* > *y* • *P* => ∀ *x* • «*x*» $>_u$ *y* ⇒ *P*
  ∀ *x* < *y* • *P* => ∀ *x* • «*x*» $<_u$ *y* ⇒ *P*

7.2 Predicate operators

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

class *refine* = *order*

abbreviation *refineBy* :: *'a::refine* ⇒ *'a* ⇒ *bool* (**infix** ⊑ *50*) **where**
P ⊑ *Q* ≡ *less-eq Q P*

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

purge-notation *inf* (**infixl** ⊓ *70*)
notation *inf* (**infixl** ⊔ *70*)
purge-notation *sup* (**infixl** ⊔ *65*)
notation *sup* (**infixl** ⊓ *65*)

purge-notation *Inf* (⨅ - [*900*] *900*)
notation *Inf* (⨆ - [*900*] *900*)
purge-notation *Sup* (⨆ - [*900*] *900*)
notation *Sup* (⨅ - [*900*] *900*)

purge-notation *bot* (⊥)
notation *bot* (⊤)
purge-notation *top* (⊤)

notation *top* (⊥)

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

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

We trivially instantiate our refinement class

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

— Configure transfer law for refinement for the fast relational tactics.

theorem *upred-ref-iff* [*uexpr-transfer-laws*]:
$(P \sqsubseteq Q) = (\forall\, b.\ [\![Q]\!]_e\ b \longrightarrow [\![P]\!]_e\ b)$
apply (*transfer*)
apply (*clarsimp*)
done

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

instantiation *uexpr* :: (*lattice*, *type*) *lattice*
begin
  lift-definition *sup-uexpr* :: (*'a*, *'b*) *uexpr* ⇒ (*'a*, *'b*) *uexpr* ⇒ (*'a*, *'b*) *uexpr*
  is λ*P Q A*. *sup* (*P A*) (*Q A*) **.**
  lift-definition *inf-uexpr* :: (*'a*, *'b*) *uexpr* ⇒ (*'a*, *'b*) *uexpr* ⇒ (*'a*, *'b*) *uexpr*
  is λ*P Q A*. *inf* (*P A*) (*Q A*) **.**
instance
  by (*intro-classes*) (*transfer*, *auto*)+
end

instantiation *uexpr* :: (*bounded-lattice*, *type*) *bounded-lattice*
begin
  lift-definition *bot-uexpr* :: (*'a*, *'b*) *uexpr* **is** λ *A*. *bot* **.**
  lift-definition *top-uexpr* :: (*'a*, *'b*) *uexpr* **is** λ *A*. *top* **.**
instance
  by (*intro-classes*) (*transfer*, *auto*)+
end

instance *uexpr* :: (*distrib-lattice*, *type*) *distrib-lattice*
  by (*intro-classes*) (*transfer*, *rule ext*, *auto simp add*: *sup-inf-distrib1*)

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

instance *uexpr* :: (*boolean-algebra*, *type*) *boolean-algebra*
apply (*intro-classes*, *unfold uexpr-defs*; *transfer*, *rule ext*)
apply (*simp-all add*: *sup-inf-distrib1 diff-eq*)
done

instantiation *uexpr* :: (*complete-lattice*, *type*) *complete-lattice*
begin

lift-definition *Inf-uexpr* :: $('a, 'b)$ *uexpr set* $\Rightarrow$ $('a, 'b)$ *uexpr*
is λ *PS A. INF P:PS. P(A)* **.**
lift-definition *Sup-uexpr* :: $('a, 'b)$ *uexpr set* $\Rightarrow$ $('a, 'b)$ *uexpr*
is λ *PS A. SUP P:PS. P(A)* **.**
instance
by (*intro-classes*)
(*transfer*, *auto intro*: *INF-lower SUP-upper simp add*: *INF-greatest SUP-least*)+
end

syntax
-mu :: *idt* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (μ - $\cdot$ - [0, 10] 10)
-nu :: *idt* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (ν - $\cdot$ - [0, 10] 10)

translations
ν *X* $\cdot$ *P* == *CONST lfp* (λ *X. P*)
μ *X* $\cdot$ *P* == *CONST gfp* (λ *X. P*)

instance *uexpr* :: (*complete-distrib-lattice*, *type*) *complete-distrib-lattice*
apply (*intro-classes*)
apply (*transfer*, *rule ext*, *auto*)
using *sup-INF* **apply** *fastforce*
apply (*transfer*, *rule ext*, *auto*)
using *inf-SUP* **apply** *fastforce*
done

instance *uexpr* :: (*complete-boolean-algebra*, *type*) *complete-boolean-algebra* **..**

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

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

abbreviation *Conj-upred* :: $'\alpha$ *upred set* $\Rightarrow$ $'\alpha$ *upred* ($\bigwedge$- [900] 900) **where**
$\bigwedge A \equiv \bigsqcup A$

abbreviation *Disj-upred* :: $'\alpha$ *upred set* $\Rightarrow$ $'\alpha$ *upred* ($\bigvee$- [900] 900) **where**
$\bigvee A \equiv \bigsqcap A$

notation
conj-upred (**infixr** $\wedge_p$ 35) **and**
disj-upred (**infixr** $\vee_p$ 30)

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

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

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

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

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

We also define the other predicate operators

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

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

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

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

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

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

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

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

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

— Configuration for UTP tactics (see *utp-tactics*).

update-uexpr-rep-eq-thms — Reread *rep-eq* theorems.

declare *utp-pred.taut.rep-eq* [*upred-defs*]

adhoc-overloading

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

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

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

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

lemma *true-alt-def*: *true* = ≪*True*≫
by (*pred-auto*)

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

declare *true-alt-def*[*THEN sym*,*lit-simps*]
declare *false-alt-def*[*THEN sym*,*lit-simps*]

abbreviation *cond* ::
(*'a*,*'α*) *uexpr* ⇒ *'α upred* ⇒ (*'a*,*'α*) *uexpr* ⇒ (*'a*,*'α*) *uexpr*
((*3*- ◃ - ▹/ -) [*52*,*0*,*53*] *52*)
where *P* ◃ *b* ▹ *Q* ≡ *trop If b P Q*

7.3 Unrestriction Laws

lemma *unrest-true* [*unrest*]: *x* ♯ *true*
by (*pred-auto*)

lemma *unrest-false* [*unrest*]: *x* ♯ *false*
by (*pred-auto*)

lemma *unrest-conj* [*unrest*]: ⟦ *x* ♯ (*P* :: *'α upred*); *x* ♯ *Q* ⟧ ⟹ *x* ♯ *P* ∧ *Q*
by (*pred-auto*)

lemma *unrest-disj* [*unrest*]: ⟦ *x* ♯ (*P* :: *'α upred*); *x* ♯ *Q* ⟧ ⟹ *x* ♯ *P* ∨ *Q*

by (*pred-auto*)

lemma *unrest-USUP* [*unrest*]:
$\llbracket\ (\bigwedge\ i.\ x\ \sharp\ P(i));\ (\bigwedge\ i.\ x\ \sharp\ Q(i))\ \rrbracket \Longrightarrow x\ \sharp\ (\bigsqcap\ i\ |\ P(i)\ \cdot\ Q(i))$
by (*pred-auto*)

lemma *unrest-UINF* [*unrest*]:
$\llbracket\ (\bigwedge\ i.\ x\ \sharp\ P(i));\ (\bigwedge\ i.\ x\ \sharp\ Q(i))\ \rrbracket \Longrightarrow x\ \sharp\ (\bigsqcup\ i\ |\ P(i)\ \cdot\ Q(i))$
by (*pred-auto*)

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

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

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

The sublens proviso can be thought of as membership below.

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

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

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

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

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

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

lemma *unrest-shAll* [*unrest*]:

assumes $\bigwedge y.\ x \sharp P(y)$
shows $x \sharp (\forall\ y \cdot P(y))$
using *assms* **by** (*pred-auto*)

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

7.4 Substitution Laws

Substitution is monotone

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.5 Predicate Laws

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

interpretation *boolean-algebra diff-upred not-upred conj-upred op* $\leq$ *op* $<$
  disj-upred false-upred true-upred
  by (*unfold-locales*; *pred-auto*)

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

lemma *refBy-order*: $P \sqsubseteq Q$ = ‘$Q \Rightarrow P$‘
  by (*pred-auto*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *subsumption1*:
$`P \Rightarrow Q` \Longrightarrow (P \vee Q) = Q$
by (*pred-auto*)

lemma *subsumption2*:
$`Q \Rightarrow P` \Longrightarrow (P \vee Q) = P$
by (*pred-auto*)

lemma *neg-conj-cancel1*: $(\neg P \wedge (P \vee Q)) = (\neg P \wedge Q :: '\alpha\ upred)$
by (*pred-auto*)

lemma *neg-conj-cancel2*: $(\neg Q \wedge (P \vee Q)) = (\neg Q \wedge P :: '\alpha\ upred)$
by (*pred-auto*)

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

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

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

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

lemma *uinf-or*:
fixes $P\ Q :: '\alpha\ upred$
shows $(P \sqcap Q) = (P \vee Q)$
by (*pred-auto*)

lemma *usup-and*:
fixes $P\ Q :: '\alpha\ upred$
shows $(P \sqcup Q) = (P \wedge Q)$
by (*pred-auto*)

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

lemma *USUP-as-Sup*: $(\prod P \in \mathcal{P} \cdot P) = \prod \mathcal{P}$
apply (*simp add*: *upred-defs bop.rep-eq lit.rep-eq Sup-uexpr-def*)
apply (*pred-simp*)
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 (*pred-simp*)
apply (*simp add*: *Setcompr-eq-image*)
done

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

lemma *USUP-image-eq* [*simp*]: $USUP\ (\lambda i.\ \ll i\gg \in_u \ll f \text{ ` } A\gg)\ g = (\bigsqcap\ i{\in}A \cdot g(f(i)))$
by (*pred-simp*, *rule-tac cong*[*of Sup Sup*], *auto*)

lemma *UINF-image-eq* [*simp*]: $UINF\ (\lambda i.\ \ll i\gg \in_u \ll f \text{ ` } A\gg)\ g = (\bigsqcup\ i{\in}A \cdot g(f(i)))$
by (*pred-simp*, *rule-tac cong*[*of Inf Inf*], *auto*)

lemma *not-USUP*: $(\neg\ (\bigsqcap\ i{\in}A{\cdot}\ P(i))) = (\bigsqcup\ i{\in}A{\cdot}\ \neg\ P(i))$
by (*pred-auto*)

lemma *not-UINF*: $(\neg\ (\bigsqcup\ i{\in}A{\cdot}\ P(i))) = (\bigsqcap\ i{\in}A{\cdot}\ \neg\ P(i))$
by (*pred-auto*)

lemma *USUP-empty* [*simp*]: $(\bigsqcap\ i \in \{\} \cdot P(i)) = \mathit{false}$
by (*pred-auto*)

lemma *USUP-insert* [*simp*]: $(\bigsqcap\ i{\in}\mathit{insert}\ x\ xs \cdot P(i)) = (P(x) \sqcap (\bigsqcap\ i{\in}xs \cdot P(i)))$
apply (*pred-simp*)
apply (*subst Sup-insert*[*THEN sym*])
apply (*rule-tac cong*[*of Sup Sup*])
apply (*auto*)
done

lemma *UINF-empty* [*simp*]: $(\bigsqcup\ i \in \{\} \cdot P(i)) = true$
  by (*pred-auto*)

lemma *UINF-insert* [*simp*]: $(\bigsqcup\ i{\in}insert\ x\ xs \cdot P(i)) = (P(x) \sqcup (\bigsqcup\ i{\in}xs \cdot P(i)))$
  apply (*pred-simp*)
  apply (*subst Inf-insert*[*THEN sym*])
  apply (*rule-tac cong*[*of Inf Inf*])
  apply (*auto*)
done

lemma *conj-USUP-dist*:
  $(P \wedge (\bigsqcap\ Q{\in}S \cdot F(Q))) = (\bigsqcap\ Q{\in}S \cdot P \wedge F(Q))$
  by (*simp add*: *upred-defs bop.rep-eq lit.rep-eq*, *pred-auto*)

lemma *disj-USUP-dist*:
  $S \neq \{\} \Longrightarrow (P \vee (\bigsqcap\ Q{\in}S \cdot F(Q))) = (\bigsqcap\ Q{\in}S \cdot P \vee F(Q))$
  by (*simp add*: *upred-defs bop.rep-eq lit.rep-eq*, *pred-auto*)

lemma *conj-UINF-dist*:
  $S \neq \{\} \Longrightarrow (P \wedge (\bigsqcup\ Q{\in}S \cdot F(Q))) = (\bigsqcup\ Q{\in}S \cdot P \wedge F(Q))$
  by (*subst uexpr-eq-iff*, *auto simp add*: *conj-upred-def UINF.rep-eq inf-uexpr.rep-eq bop.rep-eq lit.rep-eq*)

lemma *UINF-conj-UINF*: $((\bigsqcup\ P \in A \cdot F(P)) \wedge (\bigsqcup\ P \in A \cdot G(P))) = (\bigsqcup\ P \in A \cdot F(P) \wedge G(P))$
  by (*simp add*: *upred-defs bop.rep-eq lit.rep-eq*, *pred-auto*)

lemma *UINF-cong*:
  assumes $\bigwedge P.\ P \in A \Longrightarrow F(P) = G(P)$
  shows $(\bigsqcap\ P{\in}A \cdot F(P)) = (\bigsqcap\ P{\in}A \cdot G(P))$
  by (*simp add*: *USUP-as-Sup-collect assms*)

lemma *USUP-cong*:
  assumes $\bigwedge P.\ P \in A \Longrightarrow F(P) = G(P)$
  shows $(\bigsqcup\ P{\in}A \cdot F(P)) = (\bigsqcup\ P{\in}A \cdot G(P))$
  by (*simp add*: *UINF-as-Inf-collect assms*)

lemma *UINF-subset-mono*: $A \subseteq B \Longrightarrow (\bigsqcap\ P{\in}B \cdot F(P)) \sqsubseteq (\bigsqcap\ P{\in}A \cdot F(P))$
  by (*simp add*: *SUP-subset-mono USUP-as-Sup-collect*)

lemma *USUP-subset-mono*: $A \subseteq B \Longrightarrow (\bigsqcup\ P{\in}A \cdot F(P)) \sqsubseteq (\bigsqcup\ P{\in}B \cdot F(P))$
  by (*simp add*: *INF-superset-mono UINF-as-Inf-collect*)

lemma *mu-id*: $(\mu\ X \cdot X) = true$
  by (*simp add*: *antisym gfp-upperbound*)

lemma *mu-const*: $(\mu\ X \cdot P) = P$
  by (*simp add*: *gfp-const*)

lemma *nu-id*: $(\nu\ X \cdot X) = false$
  by (*simp add*: *lfp-lowerbound utp-pred.bot.extremum-uniqueI*)

lemma *nu-const*: $(\nu\ X \cdot P) = P$
  by (*simp add*: *lfp-const*)

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

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

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

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

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

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

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

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

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

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

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

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

lemma *shEx-commute*: $(\exists\ x \cdot \exists\ y \cdot P\ x\ y) = (\exists\ y \cdot \exists\ x \cdot P\ x\ y)$
  by (*pred-auto*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *subst-bool-split*:
assumes *vwb-lens x*
shows $`P` = `(P[\![false/x]\!] \wedge P[\![true/x]\!])`$
proof −
from *assms* **have** $`P` = (\forall\ v.\ `P[\![\ll v\gg/x]\!]`)$
by (*subst taut-split-subst*[*of x*], *auto*)
also have ... $= (`P[\![\ll True\gg/x]\!]` \wedge `P[\![\ll False\gg/x]\!]`)$
by (*metis* (*mono-tags*, *lifting*))
also have ... $= `(P[\![false/x]\!] \wedge P[\![true/x]\!])`$
by (*pred-auto*)
finally show *?thesis* **.**
qed

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

lemma *subst-eq-replace*:
fixes $x :: ('a, '\alpha)$ *uvar*
shows $(p[\![u/x]\!] \wedge u =_u v) = (p[\![v/x]\!] \wedge u =_u v)$
by (*pred-auto*)

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

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

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

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

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

lemma *all-commute*:
assumes $x \bowtie y$
shows $(\forall\ x \cdot \forall\ y \cdot P) = (\forall\ y \cdot \forall\ x \cdot P)$
using *assms*

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

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

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

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

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

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

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

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

lemma *ex-mono*: $P \sqsubseteq Q \Longrightarrow (\exists\ x \cdot P) \sqsubseteq (\exists\ x \cdot Q)$
by (*pred-auto*)

lemma *ex-weakens*: *wb-lens* $x \Longrightarrow (\exists\ x \cdot P) \sqsubseteq P$
by (*pred-simp*, *metis wb-lens.get-put*)

lemma *all-mono*: $P \sqsubseteq Q \Longrightarrow (\forall\ x \cdot P) \sqsubseteq (\forall\ x \cdot Q)$
by (*pred-auto*)

lemma *all-strengthens*: *wb-lens* $x \Longrightarrow P \sqsubseteq (\forall\ x \cdot P)$
by (*pred-simp*, *metis wb-lens.get-put*)

lemma *ex-unrest*: $x \sharp P \Longrightarrow (\exists\ x \cdot P) = P$
by (*pred-auto*)

lemma *all-unrest*: $x \sharp P \Longrightarrow (\forall\ x \cdot P) = P$
by (*pred-auto*)

lemma *not-ex-not*: $\neg\ (\exists\ x \cdot \neg\ P) = (\forall\ x \cdot P)$
by (*pred-auto*)

lemma *not-all-not*: $\neg\ (\forall\ x \cdot \neg\ P) = (\exists\ x \cdot P)$
by (*pred-auto*)

7.6 Conditional laws

lemma *cond-def*:
$(P \triangleleft b \triangleright Q) = ((b \wedge P) \vee ((\neg\ b) \wedge Q))$
by (*pred-auto*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *cond-conj*: $P \triangleleft b \wedge c \triangleright Q = (P \triangleleft c \triangleright Q) \triangleleft b \triangleright Q$
by (*pred-auto*)

lemma *cond-USUP-dist*: $(\bigsqcup\ P{\in}S \cdot F(P)) \triangleleft b \triangleright (\bigsqcup\ P{\in}S \cdot G(P)) = (\bigsqcup\ P{\in}S \cdot F(P) \triangleleft b \triangleright G(P))$
by (*pred-auto*)

lemma *cond-UINF-dist*: $(\bigsqcap\ P{\in}S \cdot F(P)) \triangleleft b \triangleright (\bigsqcap\ P{\in}S \cdot G(P)) = (\bigsqcap\ P{\in}S \cdot F(P) \triangleleft b \triangleright G(P))$
by (*pred-auto*)

lemma *cond-var-subst-left*:
assumes *vwb-lens x*
shows $(P[\![true/x]\!] \triangleleft var\ x \triangleright Q) = (P \triangleleft var\ x \triangleright Q)$
using *assms* **by** (*pred-auto*, *metis* (*full-types*) *vwb-lens-wb wb-lens.get-put*)

lemma *cond-var-subst-right*:
assumes *vwb-lens x*
shows $(P \triangleleft var\ x \triangleright Q[\![false/x]\!]) = (P \triangleleft var\ x \triangleright Q)$
using *assms* **by** (*pred-auto*, *metis* (*full-types*) *vwb-lens.put-eq*)

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

7.7 Cylindric algebra

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

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

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

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

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

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

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

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

7.8 Quantifier lifting

named-theorems *uquant-lift*

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

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

end

8 Alphabet manipulation

theory *utp-alphabet*
imports
utp-pred
begin

named-theorems *alpha*

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

8.1 Alphabet extension

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

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

update-uexpr-rep-eq-thms

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *aext-ulambda* [*alpha*]: $((\lambda\ x \cdot P(x)) \oplus_p\ a) = (\lambda\ x \cdot P(x) \oplus_p\ a)$
by (*pred-auto*)

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

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

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

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

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

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

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

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

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

8.2 Alphabet restriction

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

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

update-uexpr-rep-eq-thms

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.3 Alphabet lens laws

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

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

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

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

8.4 Alphabet coercion

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

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

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

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

8.5 Substitution alphabet extension

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

lemma *id-subst-ext* [*usubst*,*alpha*]:

$wb\text{-}lens\ x \Longrightarrow id \oplus_s x = id$
by *pred-auto*

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

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

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

8.6 Substitution alphabet restriction

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

lemma *id-subst-res* [*alpha*,*usubst*]:
$mwb\text{-}lens\ x \Longrightarrow id \restriction_s x = id$
by *pred-auto*

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

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

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

9 Lifting expressions

theory *utp-lift*
imports
utp-alphabet
begin

9.1 Lifting definitions

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

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

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

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

abbreviation *drop-post* :: $('a, '\alpha \times '\beta)$ *uexpr* $\Rightarrow$ $('a, '\beta)$ *uexpr* $(\lfloor\text{-}\rfloor_>)$
where $\lfloor P\rfloor_> \equiv P \upharpoonright_p snd_L$

abbreviation *lift-cond-pre* $(\lceil\text{-}\rceil_\leftarrow)$ **where** $\lceil P\rceil_\leftarrow \equiv P \oplus_p (1_L \times_L 0_L)$
abbreviation *lift-cond-post* $(\lceil\text{-}\rceil_\rightarrow)$ **where** $\lceil P\rceil_\rightarrow \equiv P \oplus_p (0_L \times_L 1_L)$

abbreviation *drop-cond-pre* $(\lfloor\text{-}\rfloor_\leftarrow)$ **where** $\lfloor P\rfloor_\leftarrow \equiv P \upharpoonright_p (1_L \times_L 0_L)$
abbreviation *drop-cond-post* $(\lfloor\text{-}\rfloor_\rightarrow)$ **where** $\lfloor P\rfloor_\rightarrow \equiv P \upharpoonright_p (0_L \times_L 1_L)$

9.2 Lifting laws

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

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

lemma *lift-cond-pre-var* [*simp*]:
$\lceil \$x\rceil_\leftarrow = \$x$
by (*pred-auto*)

lemma *lift-cond-post-var* [*simp*]:
$\lceil \$x'\rceil_\rightarrow = \$x'$
by (*pred-auto*)

9.3 Unrestriction laws

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

lemma *unrest-dash-var-cond-pre* [*unrest*]:
fixes x :: $('a, '\alpha)$ *uvar*
shows $\$x' \sharp \lceil P\rceil_\leftarrow$
by (*pred-auto*)
end

10 UTP Deduction Tactic

theory *utp-deduct*
imports *utp-pred*
begin

named-theorems *uintro*
named-theorems *uelim*
named-theorems *udest*

lemma *utrueI* [*uintro*]: $[\![true]\!]_e b$
by (*pred-auto*)

lemma *uopI* [*uintro*]: $f\ ([\![x]\!]_e b) \Longrightarrow [\![uop\ f\ x]\!]_e b$

by (*pred-auto*)

lemma *bopI* [*uintro*]: $f\ ([\![x]\!]_e b)\ ([\![y]\!]_e b) \Longrightarrow [\![bop\ f\ x\ y]\!]_e b$
by (*pred-auto*)

lemma *tropI* [*uintro*]: $f\ ([\![x]\!]_e b)\ ([\![y]\!]_e b)\ ([\![z]\!]_e b) \Longrightarrow [\![trop\ f\ x\ y\ z]\!]_e b$
by (*pred-auto*)

lemma *uconjI* [*uintro*]: $[\![\ [\![p]\!]_e b;\ [\![q]\!]_e b\ ]\!] \Longrightarrow [\![p \wedge q]\!]_e b$
by (*pred-auto*)

lemma *uconjE* [*uelim*]: $[\![\ [\![p \wedge q]\!]_e b;\ [\![\ [\![p]\!]_e b\ ;\ [\![q]\!]_e b\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*pred-auto*)

lemma *uimpI* [*uintro*]: $[\![\ [\![p]\!]_e b \Longrightarrow [\![q]\!]_e b\ ]\!] \Longrightarrow [\![p \Rightarrow q]\!]_e b$
by (*pred-auto*)

lemma *uimpE* [*elim*]: $[\![\ [\![p \Rightarrow q]\!]_e b;\ ([\![p]\!]_e b \Longrightarrow [\![q]\!]_e b) \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*pred-auto*)

lemma *ushAllI* [*uintro*]: $[\![\ \bigwedge x.\ [\![p(x)]\!]_e b\ ]\!] \Longrightarrow [\![\forall\ x \cdot p(x)]\!]_e b$
by *pred-auto*

lemma *ushExI* [*uintro*]: $[\![\ [\![p(x)]\!]_e b\ ]\!] \Longrightarrow [\![\exists\ x \cdot p(x)]\!]_e b$
by *pred-auto*

lemma *udeduct-tautI* [*uintro*]: $[\![\ \bigwedge b.\ [\![p]\!]_e b\ ]\!] \Longrightarrow `p`$
using *taut.rep-eq* **by** *blast*

lemma *udeduct-refineI* [*uintro*]: $[\![\ \bigwedge b.\ [\![q]\!]_e b \Longrightarrow [\![p]\!]_e b\ ]\!] \Longrightarrow p \sqsubseteq q$
by *pred-auto*

lemma *udeduct-eqI* [*uintro*]: $[\![\ \bigwedge b.\ [\![p]\!]_e b \Longrightarrow [\![q]\!]_e b;\ \bigwedge b.\ [\![q]\!]_e b \Longrightarrow [\![p]\!]_e b\ ]\!] \Longrightarrow p = q$
by (*pred-auto*)

Some of the following lemmas help backward reasoning with bindings

lemma *conj-implies*: $[\![\ [\![P \wedge Q]\!]_e\ b\ ]\!] \Longrightarrow [\![P]\!]_e\ b \wedge [\![Q]\!]_e\ b$
by *pred-auto*

lemma *conj-implies2*: $[\![\ [\![P]\!]_e\ b \wedge [\![Q]\!]_e\ b\ ]\!] \Longrightarrow [\![P \wedge Q]\!]_e\ b$
by *pred-auto*

lemma *disj-eq*: $[\![\ [\![P]\!]_e\ b \vee [\![Q]\!]_e\ b\ ]\!] \Longrightarrow [\![P \vee Q]\!]_e\ b$
by *pred-auto*

lemma *disj-eq2*: $[\![\ [\![P \vee Q]\!]_e\ b\ ]\!] \Longrightarrow [\![P]\!]_e\ b \vee [\![Q]\!]_e\ b$
by *pred-auto*

lemma *conj-eq-subst*: $([\![P \wedge Q]\!]_e\ b \wedge [\![P]\!]_e\ b = [\![R]\!]_e\ b) = ([\![R \wedge Q]\!]_e\ b \wedge [\![P]\!]_e\ b = [\![R]\!]_e\ b)$
by *pred-auto*

lemma *conj-imp-subst*: $([\![P \wedge Q]\!]_e\ b \wedge ([\![Q]\!]_e\ b \longrightarrow ([\![P]\!]_e\ b = [\![R]\!]_e\ b))) = ([\![R \wedge Q]\!]_e\ b \wedge ([\![Q]\!]_e\ b \longrightarrow ([\![P]\!]_e\ b = [\![R]\!]_e\ b)))$
by *pred-auto*

lemma *disj-imp-subst*: $(\llbracket Q \wedge (P \vee S)\rrbracket_e\ b \wedge (\llbracket Q\rrbracket_e\ b \longrightarrow (\llbracket P\rrbracket_e\ b = \llbracket R\rrbracket_e\ b))) = (\llbracket Q \wedge (R \vee S)\rrbracket_e\ b \wedge (\llbracket Q\rrbracket_e\ b \longrightarrow (\llbracket P\rrbracket_e\ b = \llbracket R\rrbracket_e\ b)))$
by *pred-auto*

Simplifications on value equality

lemma *uexpr-eq*: $(\llbracket e_0\rrbracket_e\ b = \llbracket e_1\rrbracket_e\ b) = \llbracket e_0 =_u e_1\rrbracket_e\ b$
by *pred-auto*

lemma *uexpr-trans*: $(\llbracket P \wedge e_0 =_u e_1\rrbracket_e\ b \wedge (\llbracket P\rrbracket_e\ b \longrightarrow \llbracket e_1 =_u e_2\rrbracket_e\ b)) = (\llbracket P \wedge e_0 =_u e_2\rrbracket_e\ b \wedge (\llbracket P\rrbracket_e\ b \longrightarrow \llbracket e_1 =_u e_2\rrbracket_e\ b))$
by (*pred-auto*)

lemma *uexpr-trans2*: $(\llbracket P \wedge Q \wedge e_0 =_u e_1\rrbracket_e\ b \wedge (\llbracket Q\rrbracket_e\ b \longrightarrow \llbracket e_1 =_u e_2\rrbracket_e\ b)) = (\llbracket P \wedge Q \wedge e_0 =_u e_2\rrbracket_e\ b \wedge (\llbracket P\rrbracket_e\ b \longrightarrow \llbracket e_1 =_u e_2\rrbracket_e\ b))$
by (*pred-auto*)

lemma *uequality*: $\llbracket\ (\llbracket R\rrbracket_e\ b = \llbracket Q\rrbracket_e\ b)\ \rrbracket \Longrightarrow \llbracket P \wedge R\rrbracket_e\ b = \llbracket P \wedge Q\rrbracket_e\ b$
by *pred-auto*

lemma *ueqe1*:$\llbracket\ \llbracket P\rrbracket_e\ b \Longrightarrow (\llbracket R\rrbracket_e\ b = \llbracket Q\rrbracket_e\ b)\ \rrbracket \Longrightarrow (\llbracket P \wedge R\rrbracket_e\ b \Longrightarrow \llbracket P \wedge Q\rrbracket_e\ b)$
by *pred-auto*

lemma *ueqe2*: $(\llbracket P\rrbracket_e\ b \Longrightarrow (\llbracket Q\rrbracket_e\ b = \llbracket R\rrbracket_e\ b) \wedge \llbracket Q \wedge P\rrbracket_e\ b = \llbracket R \wedge P\rrbracket_e\ b)$
$\Longrightarrow$
$(\llbracket P\rrbracket_e\ b \Longrightarrow (\llbracket Q\rrbracket_e\ b = \llbracket R\rrbracket_e\ b))$
by *pred-auto*

lemma *ueqe3*: $\llbracket\ \llbracket P\rrbracket_e\ b \Longrightarrow (\llbracket Q\rrbracket_e\ b = \llbracket R\rrbracket_e\ b)\ \rrbracket \Longrightarrow (\llbracket R \wedge P\rrbracket_e\ b = \llbracket Q \wedge P\rrbracket_e\ b)$
by *pred-auto*

The following allows simplifying the equality if $P \Rightarrow Q = R$

lemma *ueqe3-imp*: $(\bigwedge\ b.\ \llbracket P\rrbracket_e\ b \Longrightarrow (\llbracket Q\rrbracket_e\ b = \llbracket R\rrbracket_e\ b)) \Longrightarrow ((R \wedge P) = (Q \wedge P))$
by *pred-auto*

lemma *ueqe3-imp3*: $(\bigwedge\ b.\ \llbracket P\rrbracket_e\ b \Longrightarrow (\llbracket Q\rrbracket_e\ b = \llbracket R\rrbracket_e\ b)) \Longrightarrow ((P \wedge Q) = (P \wedge R))$
by *pred-auto*

lemma *ueqe3-imp2*: $\llbracket\ (\bigwedge\ b.\ \llbracket P0 \wedge P1\rrbracket_e\ b \Longrightarrow \llbracket Q\rrbracket_e\ b \Longrightarrow \llbracket R\rrbracket_e\ b = \llbracket S\rrbracket_e\ b)\ \rrbracket \Longrightarrow ((P0 \wedge P1 \wedge (Q \Rightarrow R)) = (P0 \wedge P1 \wedge (Q \Rightarrow S)))$
by *pred-auto*

The following can introduce the binding notation into predicates

lemma *conj-bind-dist*: $\llbracket P \wedge Q\rrbracket_e\ b = (\llbracket P\rrbracket_e\ b \wedge \llbracket Q\rrbracket_e\ b)$
by *pred-auto*

lemma *disj-bind-dist*: $\llbracket P \vee Q\rrbracket_e\ b = (\llbracket P\rrbracket_e\ b \vee \llbracket Q\rrbracket_e\ b)$
by *pred-auto*

lemma *imp-bind-dist*: $\llbracket P \Rightarrow Q\rrbracket_e\ b = (\llbracket P\rrbracket_e\ b \longrightarrow \llbracket Q\rrbracket_e\ b)$
by *pred-auto*
end

11 Alphabetised relations

theory *utp-rel*

imports
utp-pred
utp-lift
utp-tactics
begin

default-sort *type*

consts
useq :: $'a \Rightarrow 'b \Rightarrow 'c$ (**infixr** ;; *51*)
uskip :: $'a$ (*II*)

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

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

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

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

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

type-synonym $'\alpha$ *cond* $= '\alpha$ *upred*
type-synonym $('\alpha, '\beta)$ *rel* $= ('\alpha \times '\beta)$ *upred*
type-synonym $'\alpha$ *hrel* $= ('\alpha \times '\alpha)$ *upred*

translations
(*type*) $('\alpha, '\beta)$ *rel* <= (*type*) $('\alpha \times '\beta)$ *upred*

abbreviation *rcond*::$('\alpha, '\beta)$ *rel* $\Rightarrow '\alpha$ *cond* $\Rightarrow ('\alpha, '\beta)$ *rel* $\Rightarrow ('\alpha, '\beta)$ *rel*
$((3\text{-} \triangleleft\ \text{-} \triangleright_r/\ \text{-})\ [52,0,53]\ 52)$
where $(P \triangleleft b \triangleright_r Q) \equiv (P \triangleleft \lceil b \rceil_< \triangleright Q)$

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

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

is λ *e* (*b1*, *b2*). *e* (*b2*, *b1*) **.**

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

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

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

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

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

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

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

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

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

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

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

syntax
-svid-unit :: *svid* $\Rightarrow$ *svid-list* (-)
-svid-list :: *svid* $\Rightarrow$ *svid-list* $\Rightarrow$ *svid-list* (-,/ -)
-uexpr-unit :: ($'a$, $'\alpha$) *uexpr* $\Rightarrow$ *uexpr-list* (- [*40*] *40*)
-uexpr-list :: ($'a$, $'\alpha$) *uexpr* $\Rightarrow$ *uexpr-list* $\Rightarrow$ *uexpr-list* (-,/ - [*70*,*70*] *70*)
-assignment :: *svid-list* $\Rightarrow$ *uexprs* $\Rightarrow$ $'\alpha$ *hrel* (**infixr** := *62*)
-mk-usubst :: *svid-list* $\Rightarrow$ *uexprs* $\Rightarrow$ $'\alpha$ *usubst*

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

adhoc-overloading
useq seqr **and**

uskip skip-r

Homogeneous sequential composition

abbreviation *seqh* :: $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* (**infixr** $;;_h$ *51*) **where**
seqh P Q $\equiv$ (*P* ;; *Q*)

definition *rassume* :: $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *hrel* ($\text{-}^\top$ [*999*] *999*) **where**
[*urel-defs*]: *rassume c* = *II* $\triangleleft$ *c* $\triangleright_r$ *false*

definition *rassert* :: $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *hrel* ($\text{-}_\perp$ [*999*] *999*) **where**
[*urel-defs*]: *rassert c* = *II* $\triangleleft$ *c* $\triangleright_r$ *true*

We describe some properties of relations

definition *ufunctional* :: ($'a$, $'b$) *rel* $\Rightarrow$ *bool*
where *ufunctional R* $\longleftrightarrow$ *II* $\sqsubseteq$ R^- ;; *R*

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

definition *uinj* :: ($'a$, $'b$) *rel* $\Rightarrow$ *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* :: $'\alpha$ *cond* $\Rightarrow$ $'\alpha$ *hrel* ($\lceil\text{-}\rceil_t$)
where $\lceil b \rceil_t$ = ($\lceil b \rceil_<$ $\wedge$ *II*)

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

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

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

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

— Configuration for UTP tactics (see *utp-tactics*).

update-uexpr-rep-eq-thms — Reread *rep-eq* theorems.

11.1 Unrestriction Laws

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

lemma *unrest-ouvar* [*unrest*]: *in*α $\sharp$ \$*x*´
  by (*simp add*: *in*α*-def*, *transfer*, *auto*)

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

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

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

lemma *unrest-inα-var* [*unrest*]:
$[\![$ *mwb-lens* x; *in*α $\sharp$ (P :: ('a, ('α $\times$ 'β)) *uexpr*) $]\!] \Longrightarrow \$x \sharp P$
by (*rel-auto*)

lemma *unrest-outα-var* [*unrest*]:
$[\![$ *mwb-lens* x; *out*α $\sharp$ (P :: ('a, ('α $\times$ 'β)) *uexpr*) $]\!] \Longrightarrow \$x´ \sharp P$
by (*rel-auto*)

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

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

lemma *unrest-pre-outα* [*unrest*]: *out*α $\sharp \lceil b \rceil_<$
by (*transfer*, *auto simp add*: *outα-def*)

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

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

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

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

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

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

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

11.2 Substitution laws

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

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

The following laws support substitution in heterogeneous relations for polymorphically types literal expressions. These cannot be supported more generically due to limitations in HOL's type system. The laws are presented in a slightly strange way so as to be as general as possible.

lemma *bool-seqr-laws* [*usubst*]:
fixes $x :: (bool \Longrightarrow {}'\alpha)$
shows
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x \mapsto_s true) \dagger (P ;; Q) = \sigma \dagger (P[\![true/\$x]\!] ;; Q)$
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x \mapsto_s false) \dagger (P ;; Q) = \sigma \dagger (P[\![false/\$x]\!] ;; Q)$
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x' \mapsto_s true) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[\![true/\$x']\!])$
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x' \mapsto_s false) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[\![false/\$x']\!])$
by (*rel-auto*)+

lemma *zero-one-seqr-laws* [*usubst*]:
fixes $x :: (\text{-} \Longrightarrow {}'\alpha)$
shows
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x \mapsto_s 0) \dagger (P ;; Q) = \sigma \dagger (P[\![0/\$x]\!] ;; Q)$
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x \mapsto_s 1) \dagger (P ;; Q) = \sigma \dagger (P[\![1/\$x]\!] ;; Q)$
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x' \mapsto_s 0) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[\![0/\$x']\!])$
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x' \mapsto_s 1) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[\![1/\$x']\!])$
by (*rel-auto*)+

lemma *numeral-seqr-laws* [*usubst*]:
fixes $x :: (\text{-} \Longrightarrow {}'\alpha)$
shows
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x \mapsto_s numeral\ n) \dagger (P ;; Q) = \sigma \dagger (P[\![numeral\ n/\$x]\!] ;; Q)$
$\bigwedge P\ Q\ \sigma.\ \sigma(\$x' \mapsto_s numeral\ n) \dagger (P ;; Q) = \sigma \dagger (P ;; Q[\![numeral\ n/\$x']\!])$
by (*rel-auto*)+

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

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

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

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

lemma *subst-lift-upd* [*usubst*]:
fixes $x :: ('a, {}'\alpha)\ uvar$

shows $\lceil \sigma(x \mapsto_s v) \rceil_s = \lceil \sigma \rceil_s(\$x \mapsto_s \lceil v \rceil_<)$
by (*simp add*: *alpha usubst*, *simp add*: *fst-lens-def inα-def in-var-def*)

lemma *subst-drop-upd* [*usubst*]:
fixes x :: ('*a*, 'α) *uvar*
shows $\lfloor \sigma(\$x \mapsto_s v) \rfloor_s = \lfloor \sigma \rfloor_s(x \mapsto_s \lfloor v \rfloor_<)$
by (*pred-simp*, *simp add*: *inα-def prod.case-eq-if*)

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

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

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

11.3 Relation laws

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

abbreviation *truer* :: 'α *hrel* ($true_h$) **where**
truer ≡ *true*

abbreviation *falser* :: 'α *hrel* ($false_h$) **where**
falser ≡ *false*

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

We define two variants of while loops based on strongest and weakest fixed points. Only the latter properly accounts for infinite behaviours.

definition *while* :: 'α *cond* ⇒ 'α *hrel* ⇒ 'α *hrel* ($while^\top$ - *do* - *od*) **where**
$while^\top\ b\ do\ P\ od = (\nu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)$

abbreviation *while-top* :: 'α *cond* ⇒ 'α *hrel* ⇒ 'α *hrel* (*while* - *do* - *od*) **where**
while b do P od ≡ $while^\top$ *b do P od*

definition *while-bot* :: 'α *cond* ⇒ 'α *hrel* ⇒ 'α *hrel* ($while_\bot$ - *do* - *od*) **where**
$while_\bot\ b\ do\ P\ od = (\mu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)$

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

While loops with invariant decoration

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

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

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

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

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

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

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

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

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

Quantale laws for relations

lemma *seq-Sup-distl*: $P \;;\; (\bigsqcap\ A) = (\bigsqcap\ Q{\in}A.\ P \;;\; Q)$
  by (*transfer*, *auto*)

lemma *seq-Sup-distr*: $(\bigsqcap\ A) \;;\; Q = (\bigsqcap\ P{\in}A.\ P \;;\; Q)$
  by (*transfer*, *auto*)

lemma *seq-UINF-distl*: $P \;;\; (\bigsqcap\ Q{\in}A \cdot F(Q)) = (\bigsqcap\ Q{\in}A \cdot P \;;\; F(Q))$
  by (*simp add*: *USUP-as-Sup-collect seq-Sup-distl*)

lemma *seq-UINF-distr*: $(\bigsqcap\ P{\in}A \cdot F(P)) \;;\; Q = (\bigsqcap\ P{\in}A \cdot F(P) \;;\; Q)$
  by (*simp add*: *USUP-as-Sup-collect seq-Sup-distr*)

lemma *impl-seqr-mono*: $\llbracket$ ‘$P \Rightarrow Q$‘; ‘$R \Rightarrow S$‘ $\rrbracket \Longrightarrow$ ‘$(P \;;\; R) \Rightarrow (Q \;;\; S)$‘
  by (*pred-blast*)

lemma *seqr-mono*:
  $\llbracket P_1 \sqsubseteq P_2;\ Q_1 \sqsubseteq Q_2 \rrbracket \Longrightarrow (P_1 \;;\; Q_1) \sqsubseteq (P_2 \;;\; Q_2)$
  by (*rel-blast*)

lemma *seqr-monotonic*:
  $\llbracket\ \mathit{mono}\ P;\ \mathit{mono}\ Q\ \rrbracket \Longrightarrow \mathit{mono}\ (\lambda\ X.\ P\ X \;;\; Q\ X)$
  by (*simp add*: *mono-def*, *rel-blast*)

lemma *cond-mono*:
  $\llbracket P_1 \sqsubseteq P_2;\ Q_1 \sqsubseteq Q_2 \rrbracket \Longrightarrow (P_1 \triangleleft b \triangleright Q_1) \sqsubseteq (P_2 \triangleleft b \triangleright Q_2)$
  by (*rel-auto*)

lemma *cond-monotonic*:
  $\llbracket$ *mono P*; *mono Q* $\rrbracket \Longrightarrow$ *mono* $(\lambda\ X.\ P\ X \triangleleft b \triangleright Q\ X)$
  by (*simp add*: *mono-def*, *rel-blast*)

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

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

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

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

lemma *seqr-exists-left*:
  $((\exists\ \$x \cdot P)\ ;;\ Q) = (\exists\ \$x \cdot (P\ ;;\ Q))$
  by (*rel-auto*)

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

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

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

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

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

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

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

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

lemma *assign-pred-transfer*:
  fixes x :: $('a,\ '\alpha)$ *uvar*
  assumes $\$x \sharp b$ $out\alpha \sharp b$

shows $(b \wedge x := v) = (x := v \wedge b^{-})$
using *assms* **by** (*rel-blast*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *seqr-middle*:
  assumes *vwb-lens x*
  shows $(P ;; Q) = (\exists\ v \cdot P[\![\ll v\gg/\$x\acute{}]\!] ;; Q[\![\ll v\gg/\$x]\!])$
  using *assms*
  apply (*rel-auto robust*)
  apply (*rename-tac xa P Q a b y*)
  apply (*rule-tac* $x{=}get_{xa}\ y$ **in** *exI*)
  apply (*rule-tac* $x{=}y$ **in** *exI*)
  apply (*simp*)
done

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

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

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

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

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

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

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

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

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

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

theorem *feasibile-iff-true-right-zero*:
$P ;; true = true \longleftrightarrow `\exists\ out\alpha \cdot P`$
by (*rel-auto*)

11.4 Converse laws

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

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

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

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

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

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

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

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

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

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

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

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

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

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

theorem *seqr-pre-transfer'*:
  $((P \wedge \lceil q \rceil_{>})\ ;;\ R) = (P\ ;;\ (\lceil q \rceil_{<} \wedge R))$
  by (*rel-auto*)

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

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

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

lemma *seqr-pre-out*: $out\alpha \sharp p \Longrightarrow ((p \wedge Q) ;; R) = (p \wedge (Q ;; R))$
by (*rel-blast*)

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

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

lemma *seqr-to-conj*: $[\![out\alpha \sharp P; in\alpha \sharp Q]\!] \Longrightarrow (P ;; Q) = (P \wedge Q)$
by (*metis postcond-left-unit seqr-pre-out utp-pred.inf-top.right-neutral*)

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

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

11.5 Assertions and assumptions

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

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

11.6 Frame and antiframe

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

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

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

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

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

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

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

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

While loop laws

theorem *while-unfold*:
while b do P od $= ((P \; ;; \; \textit{while } b \textit{ do } P \textit{ od}) \triangleleft b \triangleright_r II)$
proof −
have *m*:*mono* $(\lambda X. \; (P \; ;; \; X) \triangleleft b \triangleright_r II)$
by (*auto intro*: *monoI seqr-mono cond-mono*)
have (*while b do P od*) $= (\nu \; X \cdot (P \; ;; \; X) \triangleleft b \triangleright_r II)$
by (*simp add*: *while-def*)
also have ... $= ((P \; ;; \; (\nu \; X \cdot (P \; ;; \; X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
by (*subst lfp-unfold*, *simp-all add*: *m*)
also have ... $= ((P \; ;; \; \textit{while } b \textit{ do } P \textit{ od}) \triangleleft b \triangleright_r II)$
by (*simp add*: *while-def*)
finally show *?thesis* **.**
qed

theorem *while-false*: *while false do P od* $= II$
by (*subst while-unfold*, *simp add*: *aext-false*)

theorem *while-true*: *while true do P od = false*
apply (*simp add*: *while-def alpha*)
apply (*rule antisym*)
apply (*simp-all*)
apply (*rule lfp-lowerbound*)
apply (*simp*)
done

theorem *while-bot-unfold*:
$\textit{while}_\bot \; b \textit{ do } P \textit{ od} = ((P \; ;; \; \textit{while}_\bot \; b \textit{ do } P \textit{ od}) \triangleleft b \triangleright_r II)$
proof −
have *m*:*mono* $(\lambda X. \; (P \; ;; \; X) \triangleleft b \triangleright_r II)$
by (*auto intro*: *monoI seqr-mono cond-mono*)
have $(\textit{while}_\bot \; b \textit{ do } P \textit{ od}) = (\mu \; X \cdot (P \; ;; \; X) \triangleleft b \triangleright_r II)$
by (*simp add*: *while-bot-def*)
also have ... $= ((P \; ;; \; (\mu \; X \cdot (P \; ;; \; X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
by (*subst gfp-unfold*, *simp-all add*: *m*)
also have ... $= ((P \; ;; \; \textit{while}_\bot \; b \textit{ do } P \textit{ od}) \triangleleft b \triangleright_r II)$
by (*simp add*: *while-bot-def*)
finally show *?thesis* **.**
qed

theorem *while-bot-false*: $\textit{while}_\bot$ *false do P od* $= II$
by (*simp add*: *while-bot-def mu-const alpha*)

theorem *while-bot-true*: $\textit{while}_\bot$ *true do P od* $= (\mu \; X \cdot P \; ;; \; X)$
by (*simp add*: *while-bot-def alpha*)

An infinite loop with a feasible body corresponds to a program error (non-termination).

theorem *while-infinite*: $P \; ;; \; \textit{true}_h = \textit{true} \Longrightarrow \textit{while}_\bot \textit{ true do } P \textit{ od} = \textit{true}$
apply (*simp add*: *while-bot-true*)
apply (*rule antisym*)

apply (*simp*)
apply (*rule gfp-upperbound*)
apply (*simp*)
done

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

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

lemma *RID-idem*:
$mwb\text{-}lens\ x \Longrightarrow RID(x)(RID(x)(P)) = RID(x)(P)$
by (*rel-auto*)

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

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

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

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

lemma *RID-assigns-r-diff*:
$[\![\ vwb\text{-}lens\ x;\ x \sharp \sigma\ ]\!] \Longrightarrow RID(x)(\langle\sigma\rangle_a) = \langle\sigma\rangle_a$
apply (*rel-auto*)
apply (*metis vwb-lens.put-eq*)
apply (*metis vwb-lens-wb wb-lens.get-put wb-lens-weak weak-lens.put-get*)
done

lemma *RID-assign-r-same*:
$vwb\text{-}lens\ x \Longrightarrow RID(x)(x := v) = II$
apply (*rel-auto*)
using *vwb-lens.put-eq* **apply** *fastforce*
done

lemma *RID-seq-left*:
assumes *vwb-lens x*
shows $RID(x)(RID(x)(P) ;; Q) = (RID(x)(P) ;; RID(x)(Q))$
proof −
have $RID(x)(RID(x)(P) ;; Q) = ((\exists\ \$x \cdot \exists\ \$x' \cdot ((\exists\ \$x \cdot \exists\ \$x' \cdot P) \wedge \$x' =_u \$x) ;; Q) \wedge \$x' =_u \$x)$

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

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

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

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

lemma *skip-r-runrest* [*unrest*]:
vwb-lens x $\Longrightarrow$ $x \ \sharp\sharp \ II$

by (*simp add*: *RID-skip-r unrest-relation-def*)

lemma *assigns-r-runrest*:
$\llbracket$ *vwb-lens* x; $x \sharp \sigma$ $\rrbracket \Longrightarrow x \sharp\sharp \langle\sigma\rangle_a$
by (*simp add*: *RID-assigns-r-diff unrest-relation-def*)

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

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

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

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

11.8 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-auto*)

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

11.9 Relation algebra laws

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

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

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

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

theorem *RA5*: $(P \vee Q)^- = (P^- \vee Q^-)$
by (*rel-auto*)

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

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

11.10 Relational alphabet extension

lift-definition *rel-alpha-ext* :: $'\beta$ *hrel* $\Rightarrow ('\beta \Longrightarrow '\alpha) \Rightarrow '\alpha$ *hrel* (**infix** $\oplus_R$ *65*)

is λ *P x* (*b1*, *b2*). *P* (get_x *b1*, get_x *b2*) $\wedge$ ($\forall$ *b*. *b1* $\oplus_L$ *b on x* = *b2* $\oplus_L$ *b on x*) **.**

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

11.11 Program values

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

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

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

11.12 Relational Hoare calculus

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

named-theorems *hoare*

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

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

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

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

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

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

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

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

lemma *while-hoare-r* [*hoare*]:
  assumes $\{\!|p \wedge b|\!\}S\{\!|p|\!\}_u$

shows $\{\!|p|\!\}$ *while b do S od*$\{\!|\neg b \wedge p|\!\}_u$
using *assms*
by (*simp add*: *while-def hoare-r-def*, *rule-tac lfp-lowerbound*) (*rel-auto*)

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

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

adhoc-overloading
uwp wp-upred

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

theorem *wp-assigns-r* [*wp*]:
$\langle\sigma\rangle_a$ *wp r* = $\sigma \dagger r$
by *rel-auto*

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

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

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

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

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

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

by *rel-auto*

If two programs have the same weakest precondition for any postcondition then the programs are the same.

theorem *wp-eq-intro*: $[\![\bigwedge r.\ P\ wp\ r = Q\ wp\ r]\!] \Longrightarrow P = Q$
by (*rel-auto robust*, *fastforce+*)
end

12 UTP Theories

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

Closure laws for theories

named-theorems *closure*

12.1 Complete lattice of predicates

definition *upred-lattice* :: $('\alpha\ upred)\ gorder\ (\mathcal{P})$ **where**
$upred\text{-}lattice = (\!|\ carrier = UNIV,\ eq = (op =),\ le = op \sqsubseteq\ |\!)$

$\mathcal{P}$ is the complete lattice of alphabetised predicates. All other theories will be defined relative to it.

interpretation *upred-lattice*: *complete-lattice* $\mathcal{P}$
proof (*unfold-locales*, *simp-all add*: *upred-lattice-def*)
fix $A :: '\alpha\ upred\ set$
show $\exists s.\ is\text{-}lub\ (\!|carrier = UNIV,\ eq = op =,\ le = op \sqsubseteq|\!)\ s\ A$
apply (*rule-tac* $x = \bigsqcup\ A$ **in** *exI*)
apply (*rule least-UpperI*)
apply (*auto intro*: *Inf-greatest simp add*: *Inf-lower Upper-def*)
done
show $\exists i.\ is\text{-}glb\ (\!|carrier = UNIV,\ eq = op =,\ le = op \sqsubseteq|\!)\ i\ A$
apply (*rule-tac* $x = \bigsqcap\ A$ **in** *exI*)
apply (*rule greatest-LowerI*)
apply (*auto intro*: *Sup-least simp add*: *Sup-upper Lower-def*)
done
qed

lemma *upred-weak-complete-lattice* [*simp*]: *weak-complete-lattice* $\mathcal{P}$
by (*simp add*: *upred-lattice.weak.weak-complete-lattice-axioms*)

lemma *upred-lattice-eq* [*simp*]:
$op\ .=_{\mathcal{P}}\ = op =$
by (*simp add*: *upred-lattice-def*)

lemma *upred-lattice-le* [*simp*]:
$le\ \mathcal{P}\ P\ Q = (P \sqsubseteq Q)$
by (*simp add*: *upred-lattice-def*)

lemma *upred-lattice-carrier* [*simp*]:
$carrier\ \mathcal{P} = UNIV$
by (*simp add*: *upred-lattice-def*)

12.2 Healthiness conditions

type-synonym $'\alpha$ *health* = $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred*

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

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

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

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

abbreviation *Healthy-carrier* :: $'\alpha$ *health* $\Rightarrow$ $'\alpha$ *upred set* ($[\![\text{-}]\!]_H$)
where $[\![H]\!]_H \equiv \{P.\ P \textit{ is } H\}$

lemma *Healthy-carrier-image*:
$A \subseteq [\![\mathcal{H}]\!]_H \Longrightarrow \mathcal{H}$ ` $A = A$
by (*auto simp add*: *image-def*, (*metis Healthy-if mem-Collect-eq subsetCE*)+)

lemma *Healthy-carrier-Collect*: $A \subseteq [\![H]\!]_H \Longrightarrow A = \{H(P) \mid P.\ P \in A\}$
by (*simp add*: *Healthy-carrier-image Setcompr-eq-image*)

lemma *Healthy-SUPREMUM*:
$A \subseteq [\![H]\!]_H \Longrightarrow$ *SUPREMUM* $A\ H = \bigsqcap\ A$
by (*drule Healthy-carrier-image*, *presburger*)

lemma *Healthy-INFIMUM*:
$A \subseteq [\![H]\!]_H \Longrightarrow$ *INFIMUM* $A\ H = \bigsqcup\ A$
by (*drule Healthy-carrier-image*, *presburger*)

lemma *Healthy-subset-member*: $[\![\ A \subseteq [\![H]\!]_H;\ P \in A\ ]\!] \Longrightarrow H(P) = P$
by (*meson Ball-Collect Healthy-if*)

lemma *is-Healthy-subset-member*: $[\![\ A \subseteq [\![H]\!]_H;\ P \in A\ ]\!] \Longrightarrow P$ *is* H
by *blast*

12.3 Properties of healthiness conditions

definition *Idempotent* :: $'\alpha$ *health* $\Rightarrow$ *bool* **where**
Idempotent$(H) \longleftrightarrow (\forall\ P.\ H(H(P)) = H(P))$

definition *Monotonic* :: $'\alpha$ *health* $\Rightarrow$ *bool* **where**
Monotonic$(H) \longleftrightarrow (\forall\ P\ Q.\ Q \sqsubseteq P \longrightarrow (H(Q) \sqsubseteq H(P)))$

definition *IMH* :: $'\alpha$ *health* $\Rightarrow$ *bool* **where**
IMH$(H) \longleftrightarrow$ *Idempotent*$(H) \wedge$ *Monotonic*(H)

definition *Antitone* :: $'\alpha$ *health* $\Rightarrow$ *bool* **where**
Antitone$(H) \longleftrightarrow (\forall\ P\ Q.\ Q \sqsubseteq P \longrightarrow (H(P) \sqsubseteq H(Q)))$

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

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

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

definition *Disjunctuous* :: $'\alpha$ *health* $\Rightarrow$ *bool* **where**
[*upred-defs*]: $Disjunctuous\ H = (\forall\ P\ Q.\ H(P \sqcap Q) = (H(P) \sqcap H(Q)))$

definition *Continuous* :: $'\alpha$ *health* $\Rightarrow$ *bool* **where**
[*upred-defs*]: $Continuous\ H = (\forall\ A.\ A \neq \{\} \longrightarrow H\ (\bigsqcap\ A) = \bigsqcap\ (H\ `\ A))$

lemma *Healthy-Idempotent* [*closure*]:
$Idempotent\ H \Longrightarrow H(P)\ is\ H$
by (*simp add*: *Healthy-def Idempotent-def*)

lemma *Idempotent-id* [*simp*]: *Idempotent id*
by (*simp add*: *Idempotent-def*)

lemma *Idempotent-comp* [*intro*]:
$[\![\ Idempotent\ f;\ Idempotent\ g;\ f \circ g = g \circ f\ ]\!] \Longrightarrow Idempotent\ (f \circ g)$
by (*auto simp add*: *Idempotent-def comp-def*, *metis*)

lemma *Idempotent-image*: $Idempotent\ f \Longrightarrow f\ `\ f\ `\ A = f\ `\ A$
by (*metis* (*mono-tags*, *lifting*) *Idempotent-def image-cong image-image*)

lemma *Monotonic-id* [*simp*]: *Monotonic id*
by (*simp add*: *Monotonic-def*)

lemma *Monotonic-comp* [*intro*]:
$[\![\ Monotonic\ f;\ Monotonic\ g\ ]\!] \Longrightarrow Monotonic\ (f \circ g)$
by (*auto simp add*: *Monotonic-def*)

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

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

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

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

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

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

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

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

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

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

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

lemma *Disjunctuous-Monotonic*: *Disjunctuous H* $\Longrightarrow$ *Monotonic H*
by (*metis Disjunctuous-def Monotonic-def semilattice-sup-class.le-iff-sup*)

lemma *Continuous-Disjunctous*: *Continuous H* $\Longrightarrow$ *Disjunctuous H*
apply (*auto simp add*: *Continuous-def Disjunctuous-def*)
apply (*rename-tac P Q*)
apply (*drule-tac* $x=\{P,Q\}$ **in** *spec*)
apply (*simp*)
done

lemma *Continuous-Monotonic*: *Continuous H* $\Longrightarrow$ *Monotonic H*
by (*simp add*: *Continuous-Disjunctous Disjunctuous-Monotonic*)

lemma *Continuous-comp* [*intro*]:
$\llbracket$ *Continuous f*; *Continuous g* $\rrbracket \Longrightarrow$ *Continuous* $(f \circ g)$
by (*simp add*: *Continuous-def*)

lemma *Healthy-fixed-points* [*simp*]: *fps* $\mathcal{P}$ $H = \llbracket H \rrbracket_H$
by (*simp add*: *fps-def upred-lattice-def Healthy-def*)

lemma *upred-lattice-Idempotent* [*simp*]: $Idem_{\mathcal{P}}$ H = *Idempotent H*
using *upred-lattice.weak-partial-order-axioms* **by** (*auto simp add*: *idempotent-def Idempotent-def*)

lemma *upred-lattice-Monotonic* [*simp*]: $Mono_{\mathcal{P}}\ H = Monotonic\ H$
  using *upred-lattice.weak-partial-order-axioms* **by** (*auto simp add*: *isotone-def Monotonic-def*)

12.4 UTP theories hierarchy

typedef $('\mathcal{T}, '\alpha)$ *uthy* = *UNIV* :: *unit set*
  by *auto*

We create a unitary parametric type to represent UTP theories. These are merely tags and contain no data other than to help the type-system resolve polymorphic definitions. The two parameters denote the name of the UTP theory – as a unique type – and the minimal alphabet that the UTP theory requires. We will then use Isabelle's ad-hoc overloading mechanism to associate theory constructs, like healthiness conditions and units, with each of these types. This will allow the type system to retrieve definitions based on a particular theory context.

definition *uthy* :: $('a, 'b)$ *uthy* **where**
uthy = *Abs-uthy* ()

lemma *uthy-eq* [*intro*]:
  fixes $x\ y$ :: $('a, 'b)$ *uthy*
  shows $x = y$
  by (*cases* x, *cases* y, *simp*)

syntax
  -UTHY :: *type* ⇒ *type* ⇒ *logic* (*UTHY*'(-, -'))

translations
  UTHY$('T, '\alpha)$ == *CONST uthy* :: $('T, '\alpha)$ *uthy*

We set up polymorphic constants to denote the healthiness conditions associated with a UTP theory. Unfortunately we can currently only characterise UTP theories of homogeneous relations; this is due to restrictions in the instantiation of Isabelle's polymorphic constants which apparently cannot specialise types in this way.

consts
  utp-hcond :: $('\mathcal{T}, '\alpha)$ *uthy* ⇒ $('\alpha \times '\alpha)$ *health* ($\mathcal{H}$ı)

definition *utp-order* :: $('\alpha \times '\alpha)$ *health* ⇒ $'\alpha$ *hrel gorder* **where**
utp-order H = (| *carrier* = $\{P.\ P\ is\ H\}$, *eq* = (*op* =), *le* = *op* ⊑ |)

abbreviation *uthy-order T* ≡ *utp-order* $\mathcal{H}_T$

Constant *utp-order* obtains the order structure associated with a UTP theory. Its carrier is the set of healthy predicates, equality is HOL equality, and the order is refinement.

lemma *utp-order-carrier* [*simp*]:
  carrier (*utp-order H*) = $[\![H]\!]_H$
  by (*simp add*: *utp-order-def*)

lemma *utp-order-eq* [*simp*]:
  eq (*utp-order T*) = *op* =
  by (*simp add*: *utp-order-def*)

lemma *utp-order-le* [*simp*]:
  le (*utp-order T*) = *op* ⊑
  by (*simp add*: *utp-order-def*)

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

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

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

lemma *isotone-utp-orderI*: *Monotonic* $H \Longrightarrow$ *isotone* (*utp-order* X) (*utp-order* Y) H
by (*auto simp add*: *Monotonic-def isotone-def utp-weak-partial-order*)

lemma *Mono-utp-orderI*:
$[\![\bigwedge P\ Q.\ [\![P \sqsubseteq Q;\ P \text{ is } H;\ Q \text{ is } H]\!] \Longrightarrow F(P) \sqsubseteq F(Q)]\!] \Longrightarrow Mono_{utp\text{-}order\ H}\ F$
by (*auto simp add*: *isotone-def utp-weak-partial-order*)

The UTP order can equivalently be characterised as the fixed point lattice, *fpl*.

lemma *utp-order-fpl*: *utp-order* $H = fpl\ \mathcal{P}\ H$
by (*auto simp add*: *utp-order-def upred-lattice-def fps-def Healthy-def*)

definition *uth-eq* :: $('T_1, '\alpha)$ *uthy* $\Rightarrow$ $('T_2, '\alpha)$ *uthy* $\Rightarrow$ *bool* (**infix** $\approx_T$ *50*) **where**
$T_1 \approx_T T_2 \longleftrightarrow [\![\mathcal{H}_{T_1}]\!]_H = [\![\mathcal{H}_{T_2}]\!]_H$

lemma *uth-eq-refl*: $T \approx_T T$
by (*simp add*: *uth-eq-def*)

lemma *uth-eq-sym*: $T_1 \approx_T T_2 \longleftrightarrow T_2 \approx_T T_1$
by (*auto simp add*: *uth-eq-def*)

lemma *uth-eq-trans*: $[\![T_1 \approx_T T_2;\ T_2 \approx_T T_3]\!] \Longrightarrow T_1 \approx_T T_3$
by (*auto simp add*: *uth-eq-def*)

definition *uthy-plus* :: $('T_1, '\alpha)$ *uthy* $\Rightarrow$ $('T_2, '\alpha)$ *uthy* $\Rightarrow$ $('T_1 \times 'T_2, '\alpha)$ *uthy* (**infixl** $+_T$ *65*) **where**
uthy-plus $T_1\ T_2$ = *uthy*

overloading
prod-hcond == *utp-hcond* :: $('T_1 \times 'T_2, '\alpha)$ *uthy* $\Rightarrow$ $('\alpha \times '\alpha)$ *health*
begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition *prod-hcond* :: $('T_1 \times 'T_2, '\alpha)$ *uthy* $\Rightarrow$ $('\alpha \times '\alpha)$ *upred* $\Rightarrow$ $('\alpha \times '\alpha)$ *upred* **where**
prod-hcond $T = \mathcal{H}_{UTHY('T_1,\ '\alpha)} \circ \mathcal{H}_{UTHY('T_2,\ '\alpha)}$

end

12.5 UTP theory hierarchy

We next define a hierarchy of locales that characterise different classes of UTP theory. Minimally we require that a UTP theory's healthiness condition is idempotent.

locale *utp-theory* =
  fixes $\mathcal{T}$ :: ($'\mathcal{T}$, $'\alpha$) *uthy* (**structure**)
  assumes *HCond-Idem*: $\mathcal{H}(\mathcal{H}(P)) = \mathcal{H}(P)$
begin

  lemma *uthy-simp*:
    uthy = $\mathcal{T}$
    by *blast*

A UTP theory fixes $\mathcal{T}$, the structural element denoting the UTP theory. All constants associated with UTP theories can then be resolved by the type system.

  lemma *HCond-Idempotent* [*closure*,*intro*]: *Idempotent* $\mathcal{H}$
    by (*simp add*: *Idempotent-def HCond-Idem*)

  sublocale *partial-order uthy-order* $\mathcal{T}$
    by (*unfold-locales*, *simp-all add*: *utp-order-def*)
end

Theory summation is commutative provided the healthiness conditions commute.

lemma *uthy-plus-comm*:
  assumes $\mathcal{H}_{T_1} \circ \mathcal{H}_{T_2} = \mathcal{H}_{T_2} \circ \mathcal{H}_{T_1}$
  shows $T_1 +_T T_2 \approx_T T_2 +_T T_1$
proof −
  have T_1 = *uthy* T_2 = *uthy*
    by *blast+*
  thus *?thesis*
    using *assms* **by** (*simp add*: *uth-eq-def prod-hcond-def*)
qed

lemma *uthy-plus-assoc*: $T_1 +_T (T_2 +_T T_3) \approx_T (T_1 +_T T_2) +_T T_3$
  by (*simp add*: *uth-eq-def prod-hcond-def comp-def*)

lemma *uthy-plus-idem*: *utp-theory* $T \Longrightarrow T +_T T \approx_T T$
  by (*simp add*: *uth-eq-def prod-hcond-def Healthy-def utp-theory.HCond-Idem utp-theory.uthy-simp*)

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

The healthiness conditions of a UTP theory lattice form a complete lattice, and allows us to make use of complete lattice results from HOL-Algebra, such as the Knaster-Tarski theorem. We can also retrieve lattice operators as below.

abbreviation *utp-top* ($\top$ı)
where *utp-top* $\mathcal{T}$ ≡ *atop* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-bottom* ($\bot$ı)
where *utp-bottom* $\mathcal{T}$ ≡ *abottom* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-join* (**infixl** $\sqcup$ı *65*) **where**
utp-join $\mathcal{T}$ ≡ *join* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-meet* (**infixl** $\sqcap$ı *70*) **where**
utp-meet $\mathcal{T}$ ≡ *meet* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-sup* ($\bigsqcup$ı- [*90*] *90*) **where**
utp-sup $\mathcal{T}$ ≡ *asup* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-inf* ($\bigsqcap$1- [*90*] *90*) **where**
utp-inf $\mathcal{T} \equiv$ *ainf* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-gfp* ($\boldsymbol{\nu}$1) **where**
utp-gfp $\mathcal{T} \equiv \nu_{\textit{uthy-order}}\ \mathcal{T}$

abbreviation *utp-lfp* ($\boldsymbol{\mu}$1) **where**
utp-lfp $\mathcal{T} \equiv \mu_{\textit{uthy-order}}\ \mathcal{T}$

lemma *upred-lattice-inf*:
ainf $\mathcal{P}$ $A = \bigsqcap\ A$
by (*metis Sup-least Sup-upper UNIV-I antisym-conv subsetI upred-lattice.weak.inf-greatest upred-lattice.weak.inf-lower upred-lattice-carrier upred-lattice-le*)

We can then derive a number of properties about these operators, as below.

context *utp-theory-lattice*
begin

lemma *LFP-healthy-comp*: $\boldsymbol{\mu}\ F = \boldsymbol{\mu}\ (F \circ \mathcal{H})$
proof −
have $\{P.\ (P\ \textit{is}\ \mathcal{H}) \wedge F\ P \sqsubseteq P\} = \{P.\ (P\ \textit{is}\ \mathcal{H}) \wedge F\ (\mathcal{H}\ P) \sqsubseteq P\}$
by (*auto simp add*: *Healthy-def*)
thus *?thesis*
by (*simp add*: *LFP-def*)
qed

lemma *GFP-healthy-comp*: $\boldsymbol{\nu}\ F = \boldsymbol{\nu}\ (F \circ \mathcal{H})$
proof −
have $\{P.\ (P\ \textit{is}\ \mathcal{H}) \wedge P \sqsubseteq F\ P\} = \{P.\ (P\ \textit{is}\ \mathcal{H}) \wedge P \sqsubseteq F\ (\mathcal{H}\ P)\}$
by (*auto simp add*: *Healthy-def*)
thus *?thesis*
by (*simp add*: *GFP-def*)
qed

lemma *top-healthy* [*closure*]: $\top$ *is* $\mathcal{H}$
using *weak.top-closed* **by** *auto*

lemma *bottom-healthy* [*closure*]: $\bot$ *is* $\mathcal{H}$
using *weak.bottom-closed* **by** *auto*

lemma *utp-top*: P *is* $\mathcal{H} \Longrightarrow P \sqsubseteq \top$
using *weak.top-higher* **by** *auto*

lemma *utp-bottom*: P *is* $\mathcal{H} \Longrightarrow \bot \sqsubseteq P$
using *weak.bottom-lower* **by** *auto*

end

lemma *upred-top*: $\top_{\mathcal{P}} = \textit{false}$
using *ball-UNIV greatest-def* **by** *fastforce*

lemma *upred-bottom*: $\bot_{\mathcal{P}} = \textit{true}$
by *fastforce*

One way of obtaining a complete lattice is showing that the healthiness conditions are monotone, which the below locale characterises.

locale *utp-theory-mono* = *utp-theory* +
 assumes *HCond-Mono* [*closure*,*intro*]: *Monotonic* $\mathcal{H}$

sublocale *utp-theory-mono* ⊆ *utp-theory-lattice*
proof −

We can then use the Knaster-Tarski theorem to obtain a complete lattice, and thus provide all the usual properties.

 interpret *weak-complete-lattice fpl* $\mathcal{P}$ $\mathcal{H}$
  by (*rule Knaster-Tarski*, *auto simp add*: *upred-lattice.weak.weak-complete-lattice-axioms*)

 have *complete-lattice* (*fpl* $\mathcal{P}$ $\mathcal{H}$)
  by (*unfold-locales*, *simp add*: *fps-def sup-exists*, (*blast intro*: *sup-exists inf-exists*)+)

 hence *complete-lattice* (*uthy-order* $\mathcal{T}$)
  by (*simp add*: *utp-order-def*, *simp add*: *upred-lattice-def*)

 thus *utp-theory-lattice* $\mathcal{T}$
  by (*simp add*: *utp-theory-axioms utp-theory-lattice-def*)
qed

context *utp-theory-mono*
begin

In a monotone theory, the top and bottom can always be obtained by applying the healthiness condition to the predicate top and bottom, respectively.

 lemma *healthy-top*: $\top = \mathcal{H}(\mathit{false})$
 proof −
  have $\top = \top_{fpl\ \mathcal{P}\ \mathcal{H}}$
   by (*simp add*: *utp-order-fpl*)
  also have ... = $\mathcal{H}\ \top_{\mathcal{P}}$
   using *Knaster-Tarski-idem-extremes*(*1*)[*of* $\mathcal{P}$ $\mathcal{H}$]
   by (*simp add*: *HCond-Idempotent HCond-Mono*)
  also have ... = $\mathcal{H}$ *false*
   by (*simp add*: *upred-top*)
  finally show *?thesis* **.**
 qed

 lemma *healthy-bottom*: $\bot = \mathcal{H}(\mathit{true})$
 proof −
  have $\bot = \bot_{fpl\ \mathcal{P}\ \mathcal{H}}$
   by (*simp add*: *utp-order-fpl*)
  also have ... = $\mathcal{H}\ \bot_{\mathcal{P}}$
   using *Knaster-Tarski-idem-extremes*(*2*)[*of* $\mathcal{P}$ $\mathcal{H}$]
   by (*simp add*: *HCond-Idempotent HCond-Mono*)
  also have ... = $\mathcal{H}$ *true*
   by (*simp add*: *upred-bottom*)
  finally show *?thesis* **.**
 qed

lemma *healthy-inf*:
 assumes $A \subseteq [\![\mathcal{H}]\!]_H$
 shows $\bigsqcap\ A = \mathcal{H}\ (\bigsqcap\ A)$
proof −
 have *1*: *weak-complete-lattice* (*uthy-order* $\mathcal{T}$)

by (*simp add*: *weak.weak-complete-lattice-axioms*)
have *2*: $Mono_{uthy\text{-}order}$ $\mathcal{T}$ $\mathcal{H}$
by (*simp add*: *HCond-Mono isotone-utp-orderI*)
have *3*: $Idem_{uthy\text{-}order}$ $\mathcal{T}$ $\mathcal{H}$
by (*simp add*: *HCond-Idem idempotent-def*)
show *?thesis*
using *Knaster-Tarski-idem-inf-eq*[*OF upred-weak-complete-lattice*, *of* $\mathcal{H}$]
by (*simp*, *metis HCond-Idempotent HCond-Mono assms partial-object.simps*(*3*) *upred-lattice-def upred-lattice-inf utp-order-def*)
qed

end

locale *utp-theory-continuous* = *utp-theory* +
assumes *HCond-Cont* [*closure*,*intro*]: *Continuous* $\mathcal{H}$

sublocale *utp-theory-continuous* ⊆ *utp-theory-mono*
proof
show *Monotonic* $\mathcal{H}$
by (*simp add*: *Continuous-Monotonic HCond-Cont*)
qed

context *utp-theory-continuous*
begin

lemma *healthy-inf-cont*:
assumes $A \subseteq [\![\mathcal{H}]\!]_H$ $A \neq \{\}$
shows $\bigsqcap$ $A = \bigsqcap$ A
proof −
have $\bigsqcap$ $A = \bigsqcap$ $(\mathcal{H}`A)$
using *Continuous-def assms*(*1*) *assms*(*2*) *healthy-inf* **by** *auto*
also have ... = $\bigsqcap$ A
by (*unfold Healthy-carrier-image*[*OF assms*(*1*)], *simp*)
finally show *?thesis* **.**
qed

lemma *healthy-inf-def*:
assumes $A \subseteq [\![\mathcal{H}]\!]_H$
shows $\bigsqcap$ A = (*if* ($A = \{\}$) *then* $\top$ *else* ($\bigsqcap$ A))
using *assms healthy-inf-cont weak.weak-inf-empty* **by** *auto*

lemma *healthy-meet-cont*:
assumes P *is* $\mathcal{H}$ Q *is* $\mathcal{H}$
shows $P \sqcap Q = P \sqcap Q$
using *healthy-inf-cont*[*of* $\{P, Q\}$] *assms*
by (*simp add*: *Healthy-if meet-def*)

lemma *meet-is-healthy*:
assumes P *is* $\mathcal{H}$ Q *is* $\mathcal{H}$
shows $P \sqcap Q$ *is* $\mathcal{H}$
by (*metis Continuous-Disjunctous Disjunctuous-def HCond-Cont Healthy-def′ assms*(*1*) *assms*(*2*))

lemma *meet-bottom* [*simp*]:
assumes P *is* $\mathcal{H}$
shows $P \sqcap \bot = \bot$

by (*simp add*: *assms semilattice-sup-class.sup-absorb2 utp-bottom*)

lemma *meet-top* [*simp*]:
assumes P *is* $\mathcal{H}$
shows $P \sqcap \top = P$
by (*simp add*: *assms semilattice-sup-class.sup-absorb1 utp-top*)

The UTP theory lfp operator can be rewritten to the alphabetised predicate lfp when in a continuous context.

theorem *utp-lfp-def*:
assumes *Monotonic* F $F \in [\![\mathcal{H}]\!]_H \to [\![\mathcal{H}]\!]_H$
shows $\boldsymbol{\mu}\ F = (\mu\ X \cdot F(\mathcal{H}(X)))$
proof (*rule antisym*)
have *ne*: $\{P.\ (P$ *is* $\mathcal{H}) \wedge F\ P \sqsubseteq P\} \neq \{\}$
proof −
have $F\ \top \sqsubseteq \top$
using *assms*(*2*) *utp-top weak.top-closed* **by** *force*
thus *?thesis*
by (*auto*, *rule-tac* $x{=}\top$ **in** *exI*, *auto simp add*: *top-healthy*)
qed
show $\boldsymbol{\mu}\ F \sqsubseteq (\mu\ X \cdot F\ (\mathcal{H}\ X))$
proof −
have $\bigsqcap\{P.\ (P$ *is* $\mathcal{H}) \wedge F(P) \sqsubseteq P\} \sqsubseteq \bigsqcap\{P.\ F(\mathcal{H}(P)) \sqsubseteq P\}$
proof −
have *1*: $\bigwedge\ P.\ F(\mathcal{H}(P)) = \mathcal{H}(F(\mathcal{H}(P)))$
by (*metis HCond-Idem Healthy-def assms*(*2*) *funcset-mem mem-Collect-eq*)
show *?thesis*
proof (*rule Sup-least*, *auto*)
fix P
assume *a*: $F\ (\mathcal{H}\ P) \sqsubseteq P$
hence F: $(F\ (\mathcal{H}\ P)) \sqsubseteq (\mathcal{H}\ P)$
by (*metis 1 HCond-Mono Monotonic-def*)
show $\bigsqcap\{P.\ (P$ *is* $\mathcal{H}) \wedge F\ P \sqsubseteq P\} \sqsubseteq P$
proof (*rule Sup-upper2*[*of* $F\ (\mathcal{H}\ P)$])
show $F\ (\mathcal{H}\ P) \in \{P.\ (P$ *is* $\mathcal{H}) \wedge F\ P \sqsubseteq P\}$
proof (*auto*)
show $F\ (\mathcal{H}\ P)$ *is* $\mathcal{H}$
by (*metis 1 Healthy-def*)
show $F\ (F\ (\mathcal{H}\ P)) \sqsubseteq F\ (\mathcal{H}\ P)$
using F *Monotonic-def assms*(*1*) **by** *blast*
qed
show $F\ (\mathcal{H}\ P) \sqsubseteq P$
by (*simp add*: *a*)
qed
qed
qed

with *ne* **show** *?thesis*
by (*simp add*: *LFP-def gfp-def*, *subst healthy-inf-cont*, *auto simp add*: *lfp-def*)
qed
from *ne* **show** $(\mu\ X \cdot F\ (\mathcal{H}\ X)) \sqsubseteq \boldsymbol{\mu}\ F$
apply (*simp add*: *LFP-def gfp-def*, *subst healthy-inf-cont*, *auto simp add*: *lfp-def*)
apply (*rule Sup-least*)
apply (*auto simp add*: *Healthy-def Sup-upper*)
done

qed

end

In another direction, we can also characterise UTP theories that are relational. Minimally this requires that the healthiness condition is closed under sequential composition.

locale *utp-theory-rel* =
  utp-theory +
  assumes *Healthy-Sequence* [*closure*]: $[\![P \text{ is } \mathcal{H}; Q \text{ is } \mathcal{H}]\!] \Longrightarrow (P ;; Q) \text{ is } \mathcal{H}$

locale *utp-theory-cont-rel* = *utp-theory-continuous* + *utp-theory-rel*
begin

  lemma *seq-cont-Sup-distl*:
    assumes $P \text{ is } \mathcal{H}\ A \subseteq [\![\mathcal{H}]\!]_H\ A \neq \{\}$
    shows $P ;; (\bigsqcap A) = \bigsqcap \{P ;; Q \mid Q.\ Q \in A\}$
  proof −
    have $\{P ;; Q \mid Q.\ Q \in A\} \subseteq [\![\mathcal{H}]\!]_H$
      using *Healthy-Sequence assms(1) assms(2)* **by** (*auto*)
    thus *?thesis*
      by (*simp add*: *healthy-inf-cont seq-Sup-distl setcompr-eq-image assms*)
  qed

  lemma *seq-cont-Sup-distr*:
    assumes $Q \text{ is } \mathcal{H}\ A \subseteq [\![\mathcal{H}]\!]_H\ A \neq \{\}$
    shows $(\bigsqcap A) ;; Q = \bigsqcap \{P ;; Q \mid P.\ P \in A\}$
  proof −
    have $\{P ;; Q \mid P.\ P \in A\} \subseteq [\![\mathcal{H}]\!]_H$
      using *Healthy-Sequence assms(1) assms(2)* **by** (*auto*)
    thus *?thesis*
      by (*simp add*: *healthy-inf-cont seq-Sup-distr setcompr-eq-image assms*)
  qed

end

There also exist UTP theories with units, and the following operator is a theory specific operator for them.

consts
  utp-unit :: $('\mathcal{T}, '\alpha)$ *uthy* $\Rightarrow$ $'\alpha$ *hrel* $(\mathcal{II}_1)$

Not all theories have both a left and a right unit (e.g. H1-H2 designs) and so we split up the locale into two cases.

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

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

locale *utp-theory-unital* =
  utp-theory-rel +
  assumes *Healthy-Unit* [*closure*]: $\mathcal{II} \text{ is } \mathcal{H}$

and *Unit-Left*: P *is* $\mathcal{H} \Longrightarrow (\mathcal{II}\ ;;\ P) = P$
and *Unit-Right*: P *is* $\mathcal{H} \Longrightarrow (P\ ;;\ \mathcal{II}) = P$

locale *utp-theory-mono-unital* = *utp-theory-mono* + *utp-theory-unital*

definition *utp-star* (-$\star$1 [*999*] *999*) **where**
utp-star $\mathcal{T}$ $P = (\boldsymbol{\nu}_{\mathcal{T}}\ (\lambda\ X.\ (P\ ;;\ X) \sqcap_{\mathcal{T}} \mathcal{II}_{\mathcal{T}}))$

definition *utp-omega* (-$\boldsymbol{\omega}$1 [*999*] *999*) **where**
utp-omega $\mathcal{T}$ $P = (\mu_{\mathcal{T}}\ (\lambda\ X.\ (P\ ;;\ X)))$

locale *utp-pre-left-quantale* = *utp-theory-continuous* + *utp-theory-left-unital*
begin

lemma *star-healthy* [*closure*]: $P\star$ *is* $\mathcal{H}$
by (*metis mem-Collect-eq utp-order-carrier utp-star-def weak.GFP-closed*)

lemma *star-unfold*: P *is* $\mathcal{H} \Longrightarrow P\star = (P;;P\star) \sqcap \mathcal{II}$
apply (*simp add*: *utp-star-def healthy-meet-cont*)
apply (*subst GFP-unfold*)
apply (*rule Mono-utp-orderI*)
apply (*simp add*: *healthy-meet-cont closure semilattice-sup-class.le-supI1 seqr-mono*)
apply (*auto intro*: *funcsetI*)
apply (*simp add*: *Healthy-Left-Unit Healthy-Sequence healthy-meet-cont meet-is-healthy*)
using *Healthy-Left-Unit Healthy-Sequence healthy-meet-cont weak.GFP-closed* **apply** *auto*
done

end

sublocale *utp-theory-unital* $\subseteq$ *utp-theory-left-unital*
by (*simp add*: *Healthy-Unit Unit-Left Healthy-Sequence utp-theory-rel-def utp-theory-axioms utp-theory-rel-axioms-def 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 Healthy-Sequence utp-theory-rel-def utp-theory-axioms utp-theory-rel-axioms-def utp-theory-right-unital-axioms-def utp-theory-right-unital-def*)

12.6 Theory of relations

We can exemplify the creation of a UTP theory with the theory of relations, a trivial theory.

typedecl *REL*
abbreviation *REL* $\equiv$ *UTHY*(*REL*, $'\alpha$)

We declare the type *REL* to be the tag for this theory. We need know nothing about this type (other than it's non-empty), since it is merely a name. We also create the corresponding constant to refer to the theory. Then we can use it to instantiate the relevant polymorphic constants.

overloading
rel-hcond == *utp-hcond* :: (*REL*, $'\alpha$) *uthy* $\Rightarrow$ ($'\alpha \times {'\alpha}$) *health*
rel-unit == *utp-unit* :: (*REL*, $'\alpha$) *uthy* $\Rightarrow$ $'\alpha$ *hrel*
begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition *rel-hcond* :: $(REL, '\alpha)$ *uthy* $\Rightarrow ('\alpha \times '\alpha)$ *upred* $\Rightarrow ('\alpha \times '\alpha)$ *upred* **where**
rel-hcond $T = id$

The unit of the theory is simply the relational unit.

definition *rel-unit* :: $(REL, '\alpha)$ *uthy* $\Rightarrow '\alpha$ *hrel* **where**
rel-unit $T = II$
end

Finally we can show that relations are a monotone and unital theory using a locale interpretation, which requires that we prove all the relevant properties. It's convenient to rewrite some of the theorems so that the provisos are more UTP like; e.g. that the carrier is the set of healthy predicates.

interpretation *rel-theory*: *utp-theory-mono-unital REL*
rewrites *carrier* (*uthy-order REL*) $= [\![id]\!]_H$
by (*unfold-locales*, *simp-all add*: *rel-hcond-def rel-unit-def Healthy-def*)

We can then, for instance, determine what the top and bottom of our new theory is.

lemma *REL-top*: $\top_{REL} = false$
by (*simp add*: *rel-theory.healthy-top*, *simp add*: *rel-hcond-def*)

lemma *REL-bottom*: $\bot_{REL} = true$
by (*simp add*: *rel-theory.healthy-bottom*, *simp add*: *rel-hcond-def*)

A number of theorems have been exported, such at the fixed point unfolding laws.

thm *rel-theory.GFP-unfold*

12.7 Theory links

We can also describe links between theories, such a Galois connections and retractions, using the following notation.

definition *mk-conn* (- $\Leftarrow\langle$-,-$\rangle\Rightarrow$ - [*90*,*0*,*0*,*91*] *91*) **where**
$H1 \Leftarrow\langle\mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow H2 \equiv (\!|\ orderA = utp\text{-}order\ H1,\ orderB = utp\text{-}order\ H2,\ lower = \mathcal{H}_2,\ upper = \mathcal{H}_1\ |\!)$

abbreviation *mk-conn'* (- $\leftarrow\langle$-,-$\rangle\rightarrow$ - [*90*,*0*,*0*,*91*] *91*) **where**
$T1 \leftarrow\langle\mathcal{H}_1,\mathcal{H}_2\rangle\rightarrow T2 \equiv \mathcal{H}_{T1} \Leftarrow\langle\mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow \mathcal{H}_{T2}$

lemma *mk-conn-orderA* [*simp*]: $\mathcal{X}_{H1\ \Leftarrow\langle\mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow\ H2} = utp\text{-}order\ H1$
by (*simp add*:*mk-conn-def*)

lemma *mk-conn-orderB* [*simp*]: $\mathcal{Y}_{H1\ \Leftarrow\langle\mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow\ H2} = utp\text{-}order\ H2$
by (*simp add*:*mk-conn-def*)

lemma *mk-conn-lower* [*simp*]: $\pi_{*H1\ \Leftarrow\langle\mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow\ H2} = \mathcal{H}_1$
by (*simp add*: *mk-conn-def*)

lemma *mk-conn-upper* [*simp*]: $\pi^*{}_{H1\ \Leftarrow\langle\mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow\ H2} = \mathcal{H}_2$
by (*simp add*: *mk-conn-def*)

lemma *galois-comp*: $(H_2 \Leftarrow\langle\mathcal{H}_3,\mathcal{H}_4\rangle\Rightarrow H_3) \circ_g (H_1 \Leftarrow\langle\mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow H_2) = H_1 \Leftarrow\langle\mathcal{H}_1\circ\mathcal{H}_3,\mathcal{H}_4\circ\mathcal{H}_2\rangle\Rightarrow H_3$
by (*simp add*: *comp-galcon-def mk-conn-def*)

Example Galois connection / retract: Existential quantification

lemma *Idempotent-ex*: *mwb-lens* $x \Longrightarrow$ *Idempotent* (*ex* x)

by (*simp add*: *Idempotent-def exists-twice*)

lemma *Monotonic-ex*: *mwb-lens x* $\Longrightarrow$ *Monotonic* (*ex x*)
by (*simp add*: *Monotonic-def ex-mono*)

lemma *ex-closed-unrest*:
vwb-lens x $\Longrightarrow$ $[\![ex\ x]\!]_H = \{P.\ x \sharp P\}$
by (*simp add*: *Healthy-def unrest-as-exists*)

Any theory can be composed with an existential quantification to produce a Galois connection

theorem *ex-retract*:
assumes *vwb-lens x Idempotent H* $ex\ x \circ H = H \circ ex\ x$
shows *retract* $((ex\ x \circ H) \Leftarrow\langle ex\ x,\ H\rangle\Rightarrow H)$
proof (*unfold-locales*, *simp-all*)
show $H \in [\![ex\ x \circ H]\!]_H \to [\![H]\!]_H$
using *Healthy-Idempotent assms* **by** *blast*
from *assms*(*1*) *assms*(*3*)[*THEN sym*] **show** $ex\ x \in [\![H]\!]_H \to [\![ex\ x \circ H]\!]_H$
by (*simp add*: *Pi-iff Healthy-def fun-eq-iff exists-twice*)
fix *P Q*
assume *P is* $(ex\ x \circ H)$ *Q is H*
thus $(H\ P \sqsubseteq Q) = (P \sqsubseteq (\exists\ x \cdot Q))$
by (*metis* (*no-types*, *lifting*) *Healthy-Idempotent Healthy-if assms comp-apply dual-order.trans ex-weakens utp-pred.ex-mono vwb-lens-wb*)
next
fix *P*
assume *P is* $(ex\ x \circ H)$
thus $(\exists\ x \cdot H\ P) \sqsubseteq P$
by (*simp add*: *Healthy-def*)
qed

corollary *ex-retract-id*:
assumes *vwb-lens x*
shows *retract* $(ex\ x \Leftarrow\langle ex\ x,\ id\rangle\Rightarrow id)$
using *assms ex-retract*[**where** *H*=*id*] **by** (*auto*)
end

13 Concurrent programming

theory *utp-concurrency*
imports *utp-rel utp-tactics*
begin

In parallel-by-merge constructions, a merge predicate defines the behaviour following execution of of parallel processes, P —— Q, as a relation that merges the output of P and Q. In order to achieve this we need to separate the variable values output from P and Q, and in addition the variable values before execution. The following three constructs do these separations.

definition [*upred-defs*]: *left-uvar* $x = x ;_L fst_L ;_L snd_L$

definition [*upred-defs*]: *right-uvar* $x = x ;_L snd_L ;_L snd_L$

definition [*upred-defs*]: *pre-uvar* $x = x ;_L fst_L$

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 (*simp add*: *alpha-in-var alpha-out-var*)
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*]: *vwb-lens* $x \Longrightarrow$ *vwb-lens* (*left-uvar* x)
by (*simp add*: *left-uvar-def*)

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

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* = $('\alpha \times ('\alpha \times '\alpha), '\alpha)$ *rel*

U0 and U1 are relations that index all input variables x to 0-x and 1-x, respectively.

definition [*upred-defs*]: *U0* = ($\$0{-}\Sigma' =_u \$\Sigma$)

definition [*upred-defs*]: *U1* = ($\$1{-}\Sigma' =_u \$\Sigma$)

As shown below, separating simulations can also be expressed using the following two alphabet extrusions

definition *U0*α **where** [*upred-defs*]: *U0*α = ($1_L \times_L$ *out-var* fst_L)

definition *U1*α **where** [*upred-defs*]: *U1*α = ($1_L \times_L$ *out-var* snd_L)

abbreviation *U0-alpha-lift* ($\lceil$-$\rceil_0$) **where** $\lceil P \rceil_0 \equiv P \oplus_p$ *U0*α

abbreviation *U1-alpha-lift* ($\lceil$-$\rceil_1$) **where** $\lceil P \rceil_1 \equiv P \oplus_p$ *U1*α

We implement the following useful abbreviation for separating of two parallel processes and copying of the before variables, all to act as input to the merge predicate.

abbreviation *par-sep* (**infixl** $\|_s$ *85*) **where**
$P \|_s Q \equiv (P \;;;\; U0) \wedge (Q \;;;\; U1) \wedge \$\Sigma_<' =_u \$\Sigma$

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* (- $\|_-$ - [*85*,*0*,*86*] *85*)
where [*upred-defs*]: $P \|_M Q = (P \|_s Q \;;;\; M)$

nil is the merge predicate which ignores the output of both parallel predicates

definition [*upred-defs*]: $nil_m = (\$\Sigma' =_u \$\Sigma_<)$

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

— TODO: There is an ambiguity below due to list assignment and tuples.

definition [*upred-defs*]: $swap_m = (0{-}\Sigma, 1{-}\Sigma := \&1{-}\Sigma, \&0{-}\Sigma)$

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

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

We can equivalently express separating simulations using alphabet extrusion

lemma *U0-as-alpha*: $(P \;;;\; U0) = \lceil P \rceil_0$
by (*rel-auto*)

lemma *U1-as-alpha*: $(P \;;;\; U1) = \lceil P \rceil_1$
by (*rel-auto*)

lemma *U0α-vwb-lens* [*simp*]: *vwb-lens U0α*
by (*simp add*: *U0α-def id-vwb-lens prod-vwb-lens*)

lemma *U1α-vwb-lens* [*simp*]: *vwb-lens U1α*
by (*simp add*: *U1α-def id-vwb-lens prod-vwb-lens*)

lemma *U0-alpha-out-var* [*alpha*]: $\lceil \$x' \rceil_0 = \$0{-}x'$
by (*rel-auto*)

lemma *U1-alpha-out-var* [*alpha*]: $\lceil \$x' \rceil_1 = \$1{-}x'$
by (*rel-auto*)

lemma *U0α-comp-in-var* [*alpha*]: $(in\text{-}var\ x) ;_L U0\alpha = in\text{-}var\ x$
by (*simp add*: *U0α-def alpha-in-var in-var-prod-lens pre-uvar-def*)

lemma *U0α-comp-out-var* [*alpha*]: $(out\text{-}var\ x) ;_L U0\alpha = out\text{-}var\ (left\text{-}uvar\ x)$
by (*simp add*: *U0α-def alpha-out-var id-wb-lens left-uvar-def out-var-prod-lens*)

lemma *U1α-comp-in-var* [*alpha*]: $(in\text{-}var\ x) ;_L U1\alpha = in\text{-}var\ x$
by (*simp add*: *U1α-def alpha-in-var in-var-prod-lens pre-uvar-def*)

lemma *U1α-comp-out-var* [*alpha*]: $(out\text{-}var\ x) ;_L U1\alpha = out\text{-}var\ (right\text{-}uvar\ x)$
by (*simp add*: *U1α-def alpha-out-var id-wb-lens right-uvar-def out-var-prod-lens*)

lemma *U0-seq-subst*: $(P \;;;\; U0)[\![\ll v \gg / \$0{-}x']\!] = (P[\![\ll v \gg / \$x']\!] \;;;\; U0)$
by (*rel-auto*)

lemma *U1-seq-subst*: $(P \;;;\; U1)[\![\ll v \gg / \$1{-}x']\!] = (P[\![\ll v \gg / \$x']\!] \;;;\; U1)$
by (*rel-auto*)

lemma *par-by-merge-false* [*simp*]:
$P \parallel_{false} Q = false$
by (*rel-auto*)

lemma *par-by-merge-left-false* [*simp*]:
false $\|_M$ Q = *false*
by (*rel-auto*)

lemma *par-by-merge-right-false* [*simp*]:
P $\|_M$ *false* = *false*
by (*rel-auto*)

lemma *par-by-merge-commute*:
assumes ($swap_m$;; M) = M
shows $P \|_M Q = Q \|_M P$
proof −
have $P \|_M Q = (((P$;; $U0) \wedge (Q$;; $U1) \wedge \$\Sigma_<{}' =_u \$\Sigma)$;; $M)$
by (*simp add*: *par-by-merge-def*)
also have ... = $((((P$;; $U0) \wedge (Q$;; $U1) \wedge \$\Sigma_<{}' =_u \$\Sigma)$;; $swap_m)$;; $M)$
by (*metis assms seqr-assoc*)
also have ... = $(((P$;; $U0$;; $swap_m) \wedge (Q$;; $U1$;; $swap_m) \wedge \$\Sigma_<{}' =_u \$\Sigma)$;; $M)$
by (*rel-auto*)
also have ... = $(((P$;; $U1) \wedge (Q$;; $U0) \wedge \$\Sigma_<{}' =_u \$\Sigma)$;; $M)$
by (*simp add*: *U0-swap U1-swap*)
also have ... = $Q \|_M P$
by (*simp add*: *par-by-merge-def utp-pred.inf.left-commute*)
finally show *?thesis* **.**
qed

lemma *shEx-pbm-left*: $((\exists\ x \cdot P\ x) \|_M Q) = (\exists\ x \cdot (P\ x \|_M Q))$
by (*rel-auto*)

lemma *shEx-pbm-right*: $(P \|_M (\exists\ x \cdot Q\ x)) = (\exists\ x \cdot (P \|_M Q\ x))$
by (*rel-auto*)

lemma *par-by-merge-mono-1*:
assumes $P_1 \sqsubseteq P_2$
shows $P_1 \|_M Q \sqsubseteq P_2 \|_M Q$
using *assms* **by** (*rel-auto*)

lemma *par-by-merge-mono-2*:
assumes $Q_1 \sqsubseteq Q_2$
shows $(P \|_M Q_1) \sqsubseteq (P \|_M Q_2)$
using *assms* **by** (*rel-blast*)

lemma *zero-one-pbm-laws* [*usubst*]:
fixes x :: (- $\Longrightarrow$ $'\alpha$)
shows
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s 0) \dagger (P \|_M Q) = \sigma \dagger ((P[\![0/\$x]\!]) \|_{M[\![0/\$x_<]\!]} (Q[\![0/\$x]\!]))$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s 1) \dagger (P \|_M Q) = \sigma \dagger ((P[\![1/\$x]\!]) \|_{M[\![1/\$x_<]\!]} (Q[\![1/\$x]\!]))$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x' \mapsto_s 0) \dagger (P \|_M Q) = \sigma \dagger (P \|_{M[\![0/\$x']\!]} Q)$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x' \mapsto_s 1) \dagger (P \|_M Q) = \sigma \dagger (P \|_{M[\![1/\$x']\!]} Q)$
by (*rel-auto*)+

lemma *numeral-pbm-laws* [*usubst*]:
fixes x :: (- $\Longrightarrow$ $'\alpha$)
shows
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s numeral\ n) \dagger (P \|_M Q) = \sigma \dagger ((P[\![numeral\ n/\$x]\!]) \|_{M[\![numeral\ n/\$x_<]\!]}$
$(Q[\![numeral\ n/\$x]\!]))$

$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x\acute{}\ \mapsto_s\ \mathit{numeral}\ n) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M[\![\mathit{numeral}\ n/\$x\acute{}]\!]} Q)$
by (*rel-auto*)+
end

14 Relational operational semantics

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

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

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

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

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

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

lemma *seq-trel*:
assumes $(\sigma, P) \to_u (\varrho, Q)$
shows $(\sigma, P ;; R) \to_u (\varrho, Q ;; R)$
by (*metis* (*no-types*, *lifting*) *assms order-refl seqr-assoc seqr-mono trel.simps*)

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

lemma *nondet-left-trel*:
$(\sigma, P \sqcap Q) \to_u (\sigma, P)$
by (*metis* (*no-types*, *hide-lams*) *disj-comm disj-upred-def semilattice-sup-class.sup.absorb-iff1 semilattice-sup-class.sup.l*
seqr-or-distr trel.simps)

lemma *nondet-right-trel*:
$(\sigma, P \sqcap Q) \to_u (\sigma, Q)$
by (*simp add*: *seqr-mono*)

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

lemma *rcond-false-trel*:
assumes $\sigma \dagger b = \mathit{false}$

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

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

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

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

14.1 Variable blocks

theory *utp-local*
imports *utp-theory*
begin

Local variables are represented as lenses whose view type is a list of values. A variable therefore effectively records the stack of values that variable has had, if any. This allows us to denote variable scopes using assignments that push and pop this stack to add or delete a particular local variable.

type-synonym $('a,\ '\alpha)\ lvar = ('a\ list,\ '\alpha)\ uvar$

Different UTP theories have different assignment operators; consequently in order to generically characterise variable blocks we need to abstractly characterise assignments. We first create two polymorphic constants that characterise the underlying program state model of a UTP theory.

consts
pvar $:: ('\mathcal{T},\ '\alpha)\ uthy \Rightarrow '\beta \Longrightarrow '\alpha$ ($\mathbf{v}$1)
pvar-assigns $:: ('\mathcal{T},\ '\alpha)\ uthy \Rightarrow '\beta\ usubst \Rightarrow '\alpha\ hrel$ ($\langle$-$\rangle$1)

pvar is a lens from the program state, $'\beta$, to the overall global state $'\alpha$, which also contains none user-space information, such as observational variables. *pvar-assigns* takes as parameter a UTP theory and returns an assignment operator which maps a substitution over the program state to a homogeneous relation on the global state. We now set up some syntax translations for these operators.

syntax
-svid-pvar $:: ('\mathcal{T},\ '\alpha)\ uthy \Rightarrow svid$ ($\mathbf{v}$1)
-thy-asgn $:: ('\mathcal{T},\ '\alpha)\ uthy \Rightarrow svid\text{-}list \Rightarrow uexprs \Rightarrow logic$ (**infixr** ::=1 *55*)

translations
-svid-pvar T => *CONST pvar T*
-thy-asgn T xs vs => *CONST pvar-assigns T* (*-mk-usubst* (*CONST id*) *xs vs*)

Next, we define constants to represent the top most variable on the local variable stack, and the remainder after this. We define these in terms of the list lens, and so for each another lens is produced.

definition *top-var* $:: ('a{::}two,\ '\alpha)\ lvar \Rightarrow ('a,\ '\alpha)\ uvar$ **where**

[*upred-defs*]: *top-var* x = (*list-lens 0* ;$_L$ x)

The remainder of the local variable stack (the tail)

definition *rest-var* :: ('*a*::*two*, 'α) *lvar* $\Rightarrow$ ('*a list*, 'α) *uvar* **where**
[*upred-defs*]: *rest-var* x = (*tl-lens* ;$_L$ x)

We can show that the top variable is a mainly well-behaved lense, and that the top most variable lens is independent of the rest of the stack.

lemma *top-mwb-lens* [*simp*]: *mwb-lens* $x \Longrightarrow$ *mwb-lens* (*top-var* x)
by (*simp add*: *list-mwb-lens top-var-def*)

lemma *top-rest-var-indep* [*simp*]:
mwb-lens $x \Longrightarrow$ *top-var* $x \bowtie$ *rest-var* x
by (*simp add*: *lens-indep-left-comp rest-var-def top-var-def*)

lemma *top-var-pres-indep* [*simp*]:
$x \bowtie y \Longrightarrow$ *top-var* $x \bowtie y$
by (*simp add*: *lens-indep-left-ext top-var-def*)

syntax
-top-var :: *svid* $\Rightarrow$ *svid* (@- [*999*] *999*)
-rest-var :: *svid* $\Rightarrow$ *svid* (↓- [*999*] *999*)

translations
-top-var x == *CONST top-var* x
-rest-var x == *CONST rest-var* x

With operators to represent local variables, assignments, and stack manipulation defined, we can go about defining variable blocks themselves.

definition *var-begin* :: ('$\mathcal{T}$, 'α) *uthy* $\Rightarrow$ ('*a*, 'β) *lvar* $\Rightarrow$ 'α *hrel* **where**
[*urel-defs*]: *var-begin* T x = x ::=$_T$ ⟨«*undefined*»⟩ ^$_u$ &x

definition *var-end* :: ('$\mathcal{T}$, 'α) *uthy* $\Rightarrow$ ('*a*, 'β) *lvar* $\Rightarrow$ 'α *hrel* **where**
[*urel-defs*]: *var-end* T x = (x ::=$_T$ $tail_u$(&x))

var-begin takes as parameters a UTP theory and a local variable, and uses the theory assignment operator to push and undefined value onto the variable stack. *var-end* removes the top most variable from the stack in a similar way.

definition *var-vlet* :: ('$\mathcal{T}$, 'α) *uthy* $\Rightarrow$ ('*a*, 'α) *lvar* $\Rightarrow$ 'α *hrel* **where**
[*urel-defs*]: *var-vlet* T x = ((\$$x$ $\neq_u$ ⟨⟩) $\wedge$ $\mathcal{II}_T$)

Next we set up the typical UTP variable block syntax, though with a suitable subscript index to represent the UTP theory parameter.

syntax
-var-begin :: *logic* $\Rightarrow$ *svid* $\Rightarrow$ *logic* (*var*ı - [*100*] *100*)
-var-begin-asn :: *logic* $\Rightarrow$ *svid* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (*var*ı - := -)
-var-end :: *logic* $\Rightarrow$ *svid* $\Rightarrow$ *logic* (*end*ı - [*100*] *100*)
-var-vlet :: *logic* $\Rightarrow$ *svid* $\Rightarrow$ *logic* (*vlet*ı - [*100*] *100*)
-var-scope :: *logic* $\Rightarrow$ *svid* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (*var*ı - • - [*0*,*10*] *10*)
-var-scope-ty :: *logic* $\Rightarrow$ *svid* $\Rightarrow$ *type* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (*var*ı - :: - • - [*0*,*0*,*10*] *10*)
-var-scope-ty-assign :: *logic* $\Rightarrow$ *svid* $\Rightarrow$ *type* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (*var*ı - :: - := - • - [*0*,*0*,*0*,*10*] *10*)

translations
-var-begin T x == *CONST var-begin* T x

-*var-begin-asn* T x e => var_T x ;; @x ::=$_T$ e
-*var-end* T x == *CONST var-end* T x
-*var-vlet* T x == *CONST var-vlet* T x
var_T x · P => var_T x ;; ((λ x. P) (*CONST top-var* x)) ;; end_T x
var_T x · P => var_T x ;; ((λ x. P) (*CONST top-var* x)) ;; end_T x

In order to substantiate standard variable block laws, we need some underlying laws about assignments, which is the purpose of the following locales.

locale *utp-prog-var* = *utp-theory* $\mathcal{T}$ **for** $\mathcal{T}$ :: ($'\mathcal{T}$, $'\alpha$) *uthy* (**structure**) +
fixes $\mathcal{VT}$ :: $'\beta$ *itself*
assumes *pvar-uvar*: *vwb-lens* (**v** :: $'\beta \Longrightarrow '\alpha$)
and *Healthy-pvar-assigns* [*closure*]: $\langle\sigma$ :: $'\beta$ *usubst*$\rangle$ *is* $\mathcal{H}$
and *pvar-assigns-comp*: ($\langle\sigma\rangle$;; $\langle\varrho\rangle$) = $\langle\varrho \circ \sigma\rangle$

We require that (1) the user-space variable is a very well-behaved lens, (2) that the assignment operator is healthy, and (3) that composing two assignments is equivalent to composing their substitutions. The next locale extends this with a left unit.

locale *utp-local-var* = *utp-prog-var* $\mathcal{T}$ V + *utp-theory-left-unital* $\mathcal{T}$ **for** $\mathcal{T}$ :: ($'\mathcal{T}$, $'\alpha$) *uthy* (**structure**) **and** V :: $'\beta$ *itself* +
assumes *pvar-assign-unit*: $\langle id$ :: $'\beta$ *usubst*$\rangle$ = $\mathcal{II}$
begin

If a left unit exists then an assignment with an identity substitution should yield the identity relation, as the above assumption requires. With these laws available, we can prove the main laws of variable blocks.

lemma *var-begin-healthy* [*closure*]:
fixes x :: ($'a$, $'\beta$) *lvar*
shows *var* x *is* $\mathcal{H}$
by (*simp add*: *var-begin-def Healthy-pvar-assigns*)

lemma *var-end-healthy* [*closure*]:
fixes x :: ($'a$, $'\beta$) *lvar*
shows *end* x *is* $\mathcal{H}$
by (*simp add*: *var-end-def Healthy-pvar-assigns*)

The beginning and end of a variable block are both healthy theory elements.

lemma *var-open-close*:
fixes x :: ($'a$, $'\beta$) *lvar*
assumes *vwb-lens* x
shows (*var* x ;; *end* x) = $\mathcal{II}$
by (*simp add*: *var-begin-def var-end-def shEx-lift-seq-1 Healthy-pvar-assigns pvar-assigns-comp pvar-assign-unit usubst assms*)

Opening and then immediately closing a variable blocks yields a skip.

lemma *var-open-close-commute*:
fixes x :: ($'a$, $'\beta$) *lvar* **and** y :: ($'b$, $'\beta$) *lvar*
assumes *vwb-lens* x *vwb-lens* y $x \bowtie y$
shows (*var* x ;; *end* y) = (*end* y ;; *var* x)
by (*simp add*: *var-begin-def var-end-def shEx-lift-seq-1 shEx-lift-seq-2*
Healthy-pvar-assigns pvar-assigns-comp
assms usubst unrest lens-indep-sym, *simp add*: *assms usubst-upd-comm*)

The beginning and end of variable blocks from different variables commute.

lemma *var-block-vacuous*:

fixes x :: $('a{::}two,\ '\beta)$ *lvar*
assumes *vwb-lens* x
shows $(var\ x \cdot \mathcal{II}) = \mathcal{II}$
by (*simp add*: *Left-Unit assms var-end-healthy var-open-close*)

A variable block with a skip inside results in a skip.

end

Example instantiation for the theory of relations

overloading
rel-pvar == *pvar* :: $(REL,\ '\alpha)$ *uthy* $\Rightarrow\ '\alpha \Longrightarrow\ '\alpha$
rel-pvar-assigns == *pvar-assigns* :: $(REL,\ '\alpha)$ *uthy* $\Rightarrow\ '\alpha$ *usubst* $\Rightarrow\ '\alpha$ *hrel*
begin
definition *rel-pvar* :: $(REL,\ '\alpha)$ *uthy* $\Rightarrow\ '\alpha \Longrightarrow\ '\alpha$ **where**
[*upred-defs*]: *rel-pvar* $T = 1_L$
definition *rel-pvar-assigns* :: $(REL,\ '\alpha)$ *uthy* $\Rightarrow\ '\alpha$ *usubst* $\Rightarrow\ '\alpha$ *hrel* **where**
[*upred-defs*]: *rel-pvar-assigns* $T\ \sigma = \langle\sigma\rangle_a$
end

interpretation *rel-local-var*: *utp-local-var* $UTHY(REL,\ '\alpha)\ TYPE('\alpha)$
proof −
interpret *vw*: *vwb-lens pvar* REL :: $'\alpha \Longrightarrow\ '\alpha$
by (*simp add*: *rel-pvar-def id-vwb-lens*)
show *utp-local-var* $TYPE('\alpha)\ UTHY(REL,\ '\alpha)$
proof
show $\bigwedge\sigma{::}'\alpha \Rightarrow\ '\alpha.\ \langle\sigma\rangle_{REL}$ *is* $\mathcal{H}_{REL}$
by (*simp add*: *rel-pvar-assigns-def rel-hcond-def Healthy-def*)
show $\bigwedge(\sigma{::}'\alpha \Rightarrow\ '\alpha)\ \varrho.\ \langle\sigma\rangle_{UTHY(REL,\ '\alpha)}\ ;;\ \langle\varrho\rangle_{REL} = \langle\varrho \circ \sigma\rangle_{REL}$
by (*simp add*: *rel-pvar-assigns-def assigns-comp*)
show $\langle id{::}'\alpha \Rightarrow\ '\alpha\rangle_{UTHY(REL,\ '\alpha)} = \mathcal{II}_{REL}$
by (*simp add*: *rel-pvar-assigns-def rel-unit-def skip-r-def*)
qed
qed
end

15 UTP Events

theory *utp-event*
imports *utp-pred*
begin

15.1 Events

Events of some type $'\vartheta$ are just the elements of that type.

type-synonym $'\vartheta$ *event* $=\ '\vartheta$

15.2 Channels

Typed channels are modelled as functions. Below, $'a$ determines the channel type and $'\vartheta$ the underlying event type. As with values, it is difficult to introduce channels as monomorphic types due to the fact that they can have arbitrary parametrisations in term of $'a$. Applying a channel to an element of its type yields an event, as we may expect. Though this is not formalised here, we may also sensibly assume that all channel- representing functions are injective. Note: is there benefit in formalising this here?

type-synonym $('a, '\vartheta)$ *chan* $=$ $'a \Rightarrow '\vartheta$ *event*

A downside of the approach is that the event type $'\vartheta$ must be able to encode *all* events of a process model, and hence cannot be fixed upfront for a single channel or channel set. To do so, we actually require a notion of 'extensible' datatypes, in analogy to extensible record types. Another solution is to encode a notion of channel scoping that namely uses *sum* types to lift channel types into extensible ones, that is using channel-set specific scoping operators. This is a current work in progress.

15.2.1 Operators

The Z type of a channel corresponds to the entire carrier of the underlying HOL type of that channel. Strictly, the function is redundant but was added to mirror the mathematical account in [?]. (TODO: Ask Simon Foster for [?])

definition *chan-type* :: $('a, '\vartheta)$ *chan* $\Rightarrow$ $'a$ *set* (δ_u) **where**
$\delta_u\ c = UNIV$

The next lifted function creates an expression that yields a channel event, from an expression on the channel type $'a$.

definition *chan-apply* ::
$('a, '\vartheta)$ *chan* $\Rightarrow$ $('a, '\alpha)$ *uexpr* $\Rightarrow$ $('\vartheta$ *event*, $'\alpha)$ *uexpr* $('(\text{-}\cdot/\text{-}')_u)$ **where**
[*upred-defs*]: $(c{\cdot}e)_u = \ll c \gg (\!| e |\!)_u$
end

16 Meta-theory for the Standard Core

theory *utp*
imports
utp-var
utp-expr
utp-unrest
utp-subst
utp-alphabet
utp-lift
utp-pred
utp-deduct
utp-rel
utp-tactics
utp-hoare
utp-wp
utp-theory
utp-concurrency
utp-rel-opsem
utp-local
utp-event
begin end

References

[1] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2015.

[2] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.

[3] A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus: a process specification and verification environment. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, 2012.