Isabelle/UTP: Mechanised reasoning for the UTP

Simon Foster Frank Zeyda

May 26, 2017

Contents

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/HOL/Library/Product-Order
  ~~/src/Tools/Adhoc-Overloading
  ~~/src/HOL/Library/Monad-Syntax
  ~~/src/HOL/Library/Countable
  ~~/src/HOL/Library/Order-Continuity
  ~~/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
```

In this first UTP theory we set up variable, which are are built on lenses. A large part of this theory is setting up the parser for UTP variable syntax.

2.1 Initial syntax setup

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

purge-notation
Order.le (**infixl** $\sqsubseteq_1$ *50*) **and**
Lattice.sup ($\bigsqcup_{1}$- [*90*] *90*) **and**
Lattice.inf ($\bigsqcap_{1}$- [*90*] *90*) **and**
Lattice.join (**infixl** $\sqcup_1$ *65*) **and**
Lattice.meet (**infixl** $\sqcap_1$ *70*) **and**
LFP (μ_1) **and**
GFP (ν_1) **and**
Set.member (*op* :) **and**
Set.member ((-/ : -) [*51*, *51*] *50*)

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

hide-type *rel*
type-synonym $'a$ *relation* = ($'a$ × $'a$) *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*]

2.2 Variable foundations

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 $'\alpha$, though by convention usually a record type where each field corresponds to a variable. UTP variables in this frame are simply modelled as lenses $'a \Longrightarrow '\alpha$, where the view type $'a$ is the variable type, and the source type $'\alpha$ is the alphabet or state-space type.

We 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 by a tuple alphabet.

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

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

Variables can also be used to effectively define sets of variables. Here we define the the universal alphabet (Σ) to be the bijective lens 1_L. This characterises the whole of the source type, and thus is effectively the set of all alphabet variables.

abbreviation (*input*) *univ-alpha* :: ($'\alpha \Longrightarrow '\alpha$) ($\Sigma$) **where**
univ-alpha ≡ 1_L

The next construct is vacuous and simply exists to help the parser distinguish predicate variables from input and output variables.

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

2.3 Variable lens properties

We can now easily show that our UTP variable construction are various classes of well-behaved lens .

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*)

Moreover, we can show that input and output variables are independent, since they refer to different sections of the alphabet.

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*)

2.4 Lens simplifications

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*)

2.5 Syntax translations

In order to support nice syntax for variables, we here set up some translations. The first step is to introduce a collection of non-terminals.

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

These non-terminals correspond to the following syntactic entities. Non-terminal *svid* is an atomic variable identifier. *svar* is a decorated variable, such as an input or output variable, and *svars* is a list of decorated variables. *salpha* is an alphabet or set of variables. Such sets can be constructed only through lens composition due to typing restrictions. Next we introduce some syntax constructors.

syntax — Identifiers
-svid :: *id* ⇒ *svid* (- [*999*] *999*)
-svid-alpha :: *svid* (Σ)
-svid-empty :: *svid* (∅)
-svid-dot :: *svid* ⇒ *svid* ⇒ *svid* (-:- [*999*,*998*] *999*)

A variable identifier can either be a HOL identifier, the complete set of variables in the alphabet Σ, the empty set ∅, or a composite identifier separated by colons, which corresponds to a sort of qualification. The final option is effectively a lens composition.

syntax — Decorations
-spvar :: *svid* ⇒ *svar* (&- [*998*] *998*)
-sinvar :: *svid* ⇒ *svar* ($- [*998*] *998*)
-soutvar :: *svid* ⇒ *svar* ($-´ [*998*] *998*)

A cariable can be decorated with an ampersand, to indicate it is a predicate variable, with a dollar to indicate its an unprimed relational variable, or a dollar and "acute" symbol to indicate its a primed relational variable. Isabelle's parser is extensible so additional decorations can be and are added later.

syntax — Variable sets
-salphaid :: *id* ⇒ *salpha* (- [*998*] *998*)
-salphavar :: *svar* ⇒ *salpha* (- [*998*] *998*)
-salphacomp :: *salpha* ⇒ *salpha* ⇒ *salpha* (**infixr** ; *75*)
-svar-nil :: *svar* ⇒ *svars* (-)
-svar-cons :: *svar* ⇒ *svars* ⇒ *svars* (-,/ -)
-salphaset :: *svars* ⇒ *salpha* ({-})
-salphamk :: *logic* ⇒ *salpha*

The terminals of an alphabet are either HOL identifiers or UTP variable identifiers. We support two ways of constructing alphabets; by composition of smaller alphabets using a semi-colon or by a set-style construction $\{a, b, c\}$ with a list of UTP variables.

syntax — Quotations
-ualpha-set :: *svars* ⇒ *logic* ($\{\text{-}\}_\alpha$)
-svar :: *svar* ⇒ *logic* ($'(\text{-}')_v$)

For various reasons, the syntax constructors above all yield specific grammar categories and will not parser at the HOL top level (basically this is to do with us wanting to reuse the syntax for expressions). As a result we provide some quotation constructors above.

Next we need to construct the syntax translations rules. First we need a few polymorphic constants.

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

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

The functions above turn a representation of a variable (type $'v$), including its name and type, into some lens type $'e$. *svar* constructs a predicate variable, *ivar* and input variables, and *ovar* and output variable. The functions bridge between the model and encoding of the variable and its interpretation as a lens in order to integrate it into the general lens-based framework. Overriding these functions is then all we need to make use of any kind of variables in terms of interfacing it with the system. Although in core UTP variables are always modelled using record field, we can overload these constants to allow other kinds of variables, such as deep variables with explicit syntax and type information.

Finally, we set up the translations rules.

translations
— Identifiers
-svid x $\rightharpoonup$ *x*
-svid-alpha $\rightleftharpoons \Sigma$
-svid-empty $\rightleftharpoons 0_L$
-svid-dot x y $\rightharpoonup$ *y* $;_L$ *x*

— Decorations
-spvar $\Sigma \leftharpoondown$ *CONST svar CONST id-lens*
-sinvar $\Sigma \leftharpoondown$ *CONST ivar* 1_L
-soutvar $\Sigma \leftharpoondown$ *CONST ovar* 1_L
-sinvar (*-svid-dot x y*) $\leftharpoondown$ *CONST ivar* (*CONST lens-comp y x*)
-soutvar (*-svid-dot x y*) $\leftharpoondown$ *CONST ovar* (*CONST lens-comp y x*)
-spvar x $\rightleftharpoons$ *CONST svar x*
-sinvar x $\rightleftharpoons$ *CONST ivar x*
-soutvar x $\rightleftharpoons$ *CONST ovar x*

— Alphabets
-salphaid x $\rightharpoonup$ *x*
-salphacomp x y $\rightharpoonup$ *x* $+_L$ *y*
-salphavar x $\rightharpoonup$ *x*
-svar-nil x $\rightharpoonup$ *x*
-svar-cons x xs $\rightharpoonup$ *x* $+_L$ *xs*
-salphaset A $\rightharpoonup$ *A*
(*-svar-cons x* (*-salphamk y*)) $\leftharpoondown$ *-salphamk* (*x* $+_L$ *y*)
x $\leftharpoondown$ *-salphamk x*

— Quotations
-ualpha-set A $\rightharpoonup$ *A*
-svar x $\rightharpoonup$ *x*

The translation rules mainly convert syntax into lens constructions, using a mixture of lens operators and the bespoke variable definitions. Notably, a colon variable identifier qualification becomes a lens composition, and variable sets are constructed using len sum. The translation rules are carefully crafted to ensure both parsing and pretty printing.

Finally we create the following useful utility translation function that allows us to construct a UTP variable (lens) type given a return and alphabet type.

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

parse-translation ‹
let
  fun uvar-ty-tr [*ty*] = *Syntax.const* @{*type-syntax lens*} $ *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

3.1 Expression type

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 $'\alpha$ to the expression's type $'a$. This general model will allow us to unify all constructions under one type. The majority definitions in the file are given using the *lifting* package [5], which allows us to reuse much of the existing library of HOL functions.

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

setup-lifting *type-definition-uexpr*

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

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

The term $[\![e]\!]_e\ b$ effectively refers to the semantic interpretation of the expression under the state-space valuation (or variables binding) b. It can be used, in concert with the lifting package, to interpret UTP constructs to their HOL equivalents. We create some theorem sets to store such transfer theorems.

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

3.2 Core expression constructs

A variable expression corresponds to the lens *get* function associated with a variable. Specifically, given a lens the expression always returns that portion of the state-space referred to by the lens.

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

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

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

We define lifting for unary, binary, ternary, and quaternary expression constructs, that simply take a HOL function with correct number of arguments and apply it function to all possible results of the expressions.

lift-definition *uop* :: $('a \Rightarrow 'b) \Rightarrow ('a, 'a)$ *uexpr* $\Rightarrow ('b, 'a)$ *uexpr*
is $\lambda\ f\ e\ b.\ f\ (e\ b)$ **.**
lift-definition *bop* ::
$('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'a)$ *uexpr* $\Rightarrow ('b, 'a)$ *uexpr* $\Rightarrow ('c, 'a)$ *uexpr*
is $\lambda\ f\ u\ v\ b.\ f\ (u\ b)\ (v\ b)$ **.**
lift-definition *trop* ::
$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow ('a, 'a)$ *uexpr* $\Rightarrow ('b, 'a)$ *uexpr* $\Rightarrow ('c, 'a)$ *uexpr* $\Rightarrow ('d, 'a)$ *uexpr*
is $\lambda\ f\ u\ v\ w\ b.\ f\ (u\ b)\ (v\ b)\ (w\ b)$ **.**
lift-definition *qtop* ::
$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e) \Rightarrow$
$('a, 'a)$ *uexpr* $\Rightarrow ('b, 'a)$ *uexpr* $\Rightarrow ('c, 'a)$ *uexpr* $\Rightarrow ('d, 'a)$ *uexpr* $\Rightarrow$
$('e, 'a)$ *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 (λ) abstraction, that takes a function producing an expression and produces an expression producing a function.

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

UTP expression is equality is simply HOL equality lifted using the *bop* binary expression constructor.

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

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

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

adhoc-overloading
ulit lit **and**
ueq eq-upred

A literal is the expression $\ll v \gg$, where v is any HOL term. Actually, the literal construct is very versatile and also allows us to refer to HOL variables within UTP expressions, and has a variety of other uses. It can therefore also be considered as a kind of quotation mechanism.

We also set up syntax for UTP variable expressions.

syntax
-uuvar :: *svar* ⇒ *logic* (-)

translations
-uuvar x == *CONST var x*

Since we already have a parser for variables, we can directly reuse it and simply apply the *var* expression construct to lift the resulting variable to an expression.

3.3 Type class instantiations

Isabelle/HOL of course provides a large hierarchy of type classes that provide constructs such as numerals and the arithmetic operators. Fortunately we can directly make use of these for UTP expressions, and thus we now perform a long list of appropriate instantiations. We first lift the core arithemtic constants and operators using a mixture of literals, unary, and binary expression constructors.

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

It should be noted that instantiating the unary minus class, *uminus*, will also provide negation UTP 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
```

Once we've set up all the core constructs for arithmetic, we can also instantiate the type classes for various algebras, including groups and rings. The proofs are done by definitional expansion, the *transfer* tactic, and then finally the theorems of the underlying HOL operators. This is mainly routine, so we don't comment further.

```
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*)+

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*)+

We can also define the order relation on expressions. Now, unlike the previous group and ring constructs, the order relations $op \leq$ and $op \leq$ return a *bool* type. This order is not therefore the lifted order which allows us to compare the valuation of two expressions, but rather the order on expressions themselves. Notably, this instantiation will later allow us to talk about predicate refinements and complete lattices.

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

UTP expressions whose return type is a partial ordered type, are also partially ordered as the following instantiation demonstrates.

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

We also lift the properties from certain ordered groups.

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

The following instantiation sets up numerals. This will allow us to have Isabelle number representations (i.e. 3,7,42,198 etc.) to UTP expressions directly.

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

The following two theorems also set up interpretation of numerals, meaning a UTP numeral can always be converted to a HOL numeral.

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*: $numeral\ x = \ll numeral\ x \gg$
by (*simp add*: *uexpr-eq-iff numeral-uexpr-rep-eq lit.rep-eq*)

We can also lift a few arithmetic properties from the class instantiations above using *transfer*.

lemma *uexpr-diff-zero* [*simp*]:
fixes *a* :: ('α::*trace*, '*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* :: ('α::*trace*, '*a*) *uexpr*
shows (*a* + *b*) − *a* = *b*
by (*simp add*: *minus-uexpr-def plus-uexpr-def*, *transfer*, *auto*)

3.4 Overloaded expression constructors

For convenience, we often want to utilise the same expression syntax for multiple constructs. This can be achieved using ad-hoc overloading. We create a number of polymorphic constants and then overload their definitions using appropriate implementations. In order for this to work, each collection must have its own unique type. Thus we do not use the HOL map type directly, but rather our own partial function type, for example.

consts
— Empty elements, for example empty set, nil list, 0...
uempty :: '*f*
— Function application, map application, list application...
uapply :: '*f* ⇒ '*k* ⇒ '*v*
— Function update, map update, list update...
uupd :: '*f* ⇒ '*k* ⇒ '*v* ⇒ '*f*
— Domain of maps, lists...
udom :: '*f* ⇒ '*a set*
— Range of maps, lists...
uran :: '*f* ⇒ '*b set*
— Domain restriction

udomres :: *'a set ⇒ 'f ⇒ 'f*
— Range restriction
uranres :: *'f ⇒ 'b set ⇒ 'f*
— Collection cardinality
ucard :: *'f ⇒ nat*
— Collection summation
usums :: *'f ⇒ 'a*

We need a function corresponding to function application in order to overload.

definition *fun-apply* :: *('a ⇒ 'b) ⇒ ('a ⇒ 'b)*
where *fun-apply f x = f x*

declare *fun-apply-def* [*simp*]

We then set up the overloading for a number of useful constructs for various collections.

adhoc-overloading
uempty 0 **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* **and**
usums list-sum **and** *usums Sum*

3.5 Syntax translations

The follows a large number of translations that lift HOL functions to UTP expressions using the various expression constructors defined above. Much of the time we try to keep the HOL syntax but add a "u" subscript.

abbreviation *ulens-override x f g ≡ lens-override f g x*

We add new non-terminals for UTP tuples and maplets.

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

syntax — Core expression constructs
-ucoerce :: *logic ⇒ type ⇒ logic* (**infix** $:_u$ *50*)
-ulambda :: *pttrn ⇒ logic ⇒ logic* (λ - $\cdot$ - [*0*, *10*] *10*)
-ulens-ovrd :: *logic ⇒ logic ⇒ svar ⇒ logic* (- $\oplus$ - *on* - [*85*, *0*, *86*] *86*)

translations
$\lambda\ x \cdot p$ == *CONST ulambda* ($\lambda\ x.\ p$)
$x :_u$ *'a* == *x* :: (*'a*, -) *uexpr*
-ulens-ovrd f g a == *CONST bop* (*CONST ulens-override a*) *f g*

syntax — Tuples
-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*'*(-*'*))

translations
$()_u$ == «()»
$(x, y)_u$ == *CONST bop* (*CONST Pair*) *x y*
-utuple x (*-utuple-args y z*) == *-utuple x* (*-utuple-arg* (*-utuple y z*))
$\pi_1(x)$ == *CONST uop CONST fst x*
$\pi_2(x)$ == *CONST uop CONST snd x*

syntax — Polymorphic constructs
-umap-empty :: *logic* ($[]_u$)
-uapply :: ('*a* ⇒ '*b*, 'α) *uexpr* ⇒ *utuple-args* ⇒ ('*b*, 'α) *uexpr* (-($\!|$-$|\!$)$_u$ [*999*,*0*] *999*)
-umaplet :: [*logic*, *logic*] => *umaplet* (- /$\mapsto$/ -)
:: *umaplet* => *umaplets* (-)
-UMaplets :: [*umaplet*, *umaplets*] => *umaplets* (-,/ -)
-UMapUpd :: [*logic*, *umaplets*] => *logic* (-/'(-')$_u$ [*900*,*0*] *900*)
-UMap :: *umaplets* => *logic* ((*1*[-]$_u$))
-ucard :: *logic* ⇒ *logic* (#$_u$'(-'))
-uless :: *logic* ⇒ *logic* ⇒ *logic* (**infix** $<_u$ *50*)
-uleq :: *logic* ⇒ *logic* ⇒ *logic* (**infix** $\leq_u$ *50*)
-ugreat :: *logic* ⇒ *logic* ⇒ *logic* (**infix** $>_u$ *50*)
-ugeq :: *logic* ⇒ *logic* ⇒ *logic* (**infix** $\geq_u$ *50*)
-uceil :: *logic* ⇒ *logic* ($\lceil$-$\rceil_u$)
-ufloor :: *logic* ⇒ *logic* ($\lfloor$-$\rfloor_u$)
-udom :: *logic* ⇒ *logic* (*dom*$_u$'(-'))
-uran :: *logic* ⇒ *logic* (*ran*$_u$'(-'))
-usum :: *logic* ⇒ *logic* (*sum*$_u$'(-'))
-udom-res :: *logic* ⇒ *logic* ⇒ *logic* (**infixl** $\lhd_u$ *85*)
-uran-res :: *logic* ⇒ *logic* ⇒ *logic* (**infixl** $\rhd_u$ *85*)
-umin :: *logic* ⇒ *logic* ⇒ *logic* (*min*$_u$'(-, -'))
-umax :: *logic* ⇒ *logic* ⇒ *logic* (*max*$_u$'(-, -'))
-ugcd :: *logic* ⇒ *logic* ⇒ *logic* (*gcd*$_u$'(-, -'))

translations
— Pretty printing for adhoc-overloaded constructs
$f(\!|x|\!)_u$ <= *CONST uapply f x*
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*

— Overloaded construct translations
$f(\!|x,y|\!)_u$ == *CONST bop CONST uapply f* $(x,y)_u$
$f(\!|x|\!)_u$ == *CONST bop CONST uapply f x*
$\#_u(xs)$ == *CONST uop CONST ucard xs*
sum$_u(A)$ == *CONST uop CONST usums A*
dom$_u(f)$ == *CONST uop CONST udom f*
ran$_u(f)$ == *CONST uop CONST uran f*
$[]_u$ == «*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) $\quad$ *<= -UMapUpd (-UMap ms1) ms2*
-UMaplets ms1 (-UMaplets ms2 ms3) <= -UMaplets (-UMaplets ms1 ms2) ms3

— Type-class polymorphic constructs
$x <_u y$ == *CONST bop (op <) x y*
$x \leq_u y$ == *CONST bop (op ≤) x y*
$x >_u y$ == $y <_u x$
$x \geq_u y$ == $y \leq_u x$
$min_u(x, y)$ == *CONST bop (CONST min) x y*
$max_u(x, y)$ == *CONST bop (CONST max) x y*
$gcd_u(x, y)$ == *CONST bop (CONST gcd) x y*
$\lceil x \rceil_u$ == *CONST uop CONST ceiling x*
$\lfloor x \rfloor_u$ == *CONST uop CONST floor x*

syntax — Lists / Sequences
-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* ($drop_u$'(-,/ -'))
-ufilter :: *('a list, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a list, 'α) uexpr* (**infixl** $\restriction_u$ *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* ($distinct_u$'(-'))

translations
⟨⟩ == «[]»
⟨*x, xs*⟩ == *CONST bop (op #) x* ⟨*xs*⟩
⟨*x*⟩ == *CONST bop (op #) x* «[]»
x $\hat{}_u$ *y* == *CONST bop (op @) x y*
$last_u(xs)$ == *CONST uop CONST last xs*
$front_u(xs)$ == *CONST uop CONST butlast xs*
$head_u(xs)$ == *CONST uop CONST hd xs*
$tail_u(xs)$ == *CONST uop CONST tl xs*
$drop_u(n,xs)$ == *CONST bop CONST drop n xs*
$take_u(n,xs)$ == *CONST bop CONST take n xs*
$elems_u(xs)$ == *CONST uop CONST set xs*
$sorted_u(xs)$ == *CONST uop CONST sorted xs*
$distinct_u(xs)$ == *CONST uop CONST distinct xs*
xs $\restriction_u$ *A* == *CONST bop CONST seq-filter xs A*
A $\upharpoonright_u$ *xs* == *CONST bop (op* $\upharpoonright_l$*) A xs*

syntax — Sets
-ufinite :: *logic ⇒ logic* ($finite_u$'(-'))
-uempset :: *('a set, 'α) uexpr* ($\{\}_u$)
-uset :: *args => ('a set, 'α) uexpr* ($\{(-)\}_u$)
-uunion :: *('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ ('a set, 'α) uexpr* (**infixl** $\cup_u$ *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 :: $('a\ set,\ 'a)\ uexpr \Rightarrow ('a\ set,\ 'a)\ uexpr \Rightarrow (bool,\ 'a)\ uexpr$ (**infix** $\subseteq_u$ *50*)

translations
$finite_u(x) == CONST\ uop\ (CONST\ finite)\ x$
$\{\}_u == \ll\{\}\gg$
$\{x,\ xs\}_u == CONST\ bop\ (CONST\ insert)\ x\ \{xs\}_u$
$\{x\}_u == CONST\ bop\ (CONST\ insert)\ x\ \ll\{\}\gg$
$A \cup_u B == CONST\ bop\ (op \cup)\ A\ B$
$A \cap_u B == CONST\ bop\ (op \cap)\ A\ B$
$x \in_u A == CONST\ bop\ (op \in)\ x\ A$
$A \subset_u B == CONST\ bop\ (op <)\ A\ B$
$A \subset_u B <= CONST\ bop\ (op \subset)\ A\ B$
$f \subset_u g <= CONST\ bop\ (op \subset_p)\ f\ g$
$f \subset_u g <= CONST\ bop\ (op \subset_f)\ f\ g$
$A \subseteq_u B == CONST\ bop\ (op \leq)\ A\ B$
$A \subseteq_u B <= CONST\ bop\ (op \subseteq)\ A\ B$
$f \subseteq_u g <= CONST\ bop\ (op \subseteq_p)\ f\ g$
$f \subseteq_u g <= CONST\ bop\ (op \subseteq_f)\ f\ g$

syntax — Partial functions
-umap-plus :: $logic \Rightarrow logic \Rightarrow logic$ (**infixl** $\oplus_u$ *85*)
-umap-minus :: $logic \Rightarrow logic \Rightarrow logic$ (**infixl** $\ominus_u$ *85*)

translations
$f \oplus_u g => (f :: ((-,\ -)\ pfun,\ -)\ uexpr) + g$
$f \ominus_u g => (f :: ((-,\ -)\ pfun,\ -)\ uexpr) - g$

syntax — Sum types
-uinl :: $logic \Rightarrow logic$ ($inl_u\ '(-')$)
-uinr :: $logic \Rightarrow logic$ ($inr_u\ '(-')$)

translations
$inl_u(x) == CONST\ uop\ CONST\ Inl\ x$
$inr_u(x) == CONST\ uop\ CONST\ Inr\ x$

3.6 Lifting set collectors

We provide syntax for various types of set collectors, including intervals and the Z-style set comprehension which is purpose built as a new lifted definition.

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

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

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

3.7 Lifting limits

We also lift the following functions on topological spaces for taking function limits, and describing continuity.

definition *ulim-left* :: *'a*::*order-topology* ⇒ (*'a* ⇒ *'b*) ⇒ *'b*::*t2-space* **where**
ulim-left = (λ *p f*. *Lim* (*at-left p*) *f*)

definition *ulim-right* :: *'a*::*order-topology* ⇒ (*'a* ⇒ *'b*) ⇒ *'b*::*t2-space* **where**
ulim-right = (λ *p f*. *Lim* (*at-right p*) *f*)

definition *ucont-on* :: (*'a*::*topological-space* ⇒ *'b*::*topological-space*) ⇒ *'a set* ⇒ *bool* **where**
ucont-on = (λ *f A*. *continuous-on A f*)

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

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

3.8 Evaluation laws for expressions

We now collect together all the definitional theorems for expression constructs, and use them to build an evaluation strategy for expressions that we will later use to construct proof tactics for UTP predicates.

lemmas *uexpr-defs* =
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
plus-list-def

The following laws show how to evaluate the core expressions constructs in terms of which the above definitions are defined. Thus, using these theorems together, we can convert any UTP expression into a pure HOL expression. All these theorems are marked as *ueval* theorems which can be used for evaluation.

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*)

We also add all the definitional expressions to the evaluation theorem set.

declare *uexpr-defs* [*ueval*]

3.9 Misc laws

We also prove a few useful algebraic and expansion laws for expressions.

lemma *uop-const* [*simp*]: $uop\ id\ u = u$
by (*transfer*, *simp*)

lemma *bop-const-1* [*simp*]: $bop\ (\lambda x\ y.\ y)\ u\ v = v$
by (*transfer*, *simp*)

lemma *bop-const-2* [*simp*]: $bop\ (\lambda x\ y.\ x)\ u\ v = u$
by (*transfer*, *simp*)

lemma *uinter-empty-1* [*simp*]: $x \cap_u \{\}_u = \{\}_u$
by (*transfer*, *simp*)

lemma *uinter-empty-2* [*simp*]: $\{\}_u \cap_u x = \{\}_u$
by (*transfer*, *simp*)

lemma *uunion-empty-1* [*simp*]: $\{\}_u \cup_u x = x$
by (*transfer*, *simp*)

lemma *uset-minus-empty* [*simp*]: $x - \{\}_u = x$
by (*simp add*: *uexpr-defs*, *transfer*, *simp*)

lemma *ulist-filter-empty* [*simp*]: $x \upharpoonright_u \{\}_u = \langle\rangle$
by (*transfer*, *simp*)

lemma *tail-cons* [*simp*]: $tail_u(\langle x\rangle\ ^\frown_u\ xs) = xs$
by (*transfer*, *simp*)

3.10 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 $uempty = []_u$

$1 = \ll 1 :: ?'a \gg$

$?u + ?v = bop\ op + ?u\ ?v$

$- ?u = uop\ uminus\ ?u$

$?u - ?v = bop\ op - ?u\ ?v$

$?u * ?v = bop\ op * ?u\ ?v$

$inverse\ ?u = uop\ inverse\ ?u$

$?u\ div\ ?v = bop\ op\ div\ ?u\ ?v$

$sgn\ ?u = uop\ sgn\ ?u$

$|?u| = uop\ abs\ ?u$

$?u\ mod\ ?v = bop\ op\ mod\ ?u\ ?v$

$(?x =_u ?y) = bop\ op = ?x\ ?y$

$numeral\ ?x = \ll numeral\ ?x \gg$

$ulim\text{-}left = (\lambda p.\ Lim\ (at\text{-}left\ p))$

$ulim\text{-}right = (\lambda p.\ Lim\ (at\text{-}right\ p))$

$ucont\text{-}on = (\lambda f\ A.\ continuous\text{-}on\ A\ f)$

$op + = op$ @ 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

4.1 Definitions and Core Syntax

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 lens 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.

Unrestriction was first defined in the work of Marcel Oliveira [7, 6] in his UTP mechanisation in *ProofPowerZ*. Our definition modifies his in that our variables are semantically characterised as lenses, and supported by the lens laws, rather than named syntactic entities. We effectively fuse the ideas from both Feliachi [2] and Oliveira's [6] mechanisations of the UTP, the former being also purely semantic in nature.

We first set up overloaded syntax for unrestriction, as several concepts will have this defined.

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*
-unrest (*-salphaset* (*-salphamk* ($x +_L y$))) *P* <= *-unrest* ($x +_L y$) *P*

Our syntax translations support both variables and variable sets such that we can write down predicates like $\&x \sharp P$ and also $\{\&x, \&y, \&z\} \sharp P$.

We set up a simple tactic for discharging unrestriction conjectures using a simplification set.

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

Unrestriction for expressions is defined as a lifted construct using the underlying lens operations. It states that lens x is unrestricted by expression e provided that, for any state-space binding b and variable valuation v, the value which the expression evaluates to is unaltered if we set x to v in b. In other words, we cannot effect the behaviour of e by changing x. Thus e does not observe the portion of state-space characterised by x. We add this definition to our overloaded constant.

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

adhoc-overloading
unrest unrest-uexpr

4.2 Unrestriction laws

We now prove unrestriction laws for the key constructs of our expression model. Many of these depend on lens properties and so variously employ the assumptions *mwb-lens* and *vwb-lens*, depending on the number of assumptions from the lenses theory is required.

Firstly, we prove a general property – if x and y are both unrestricted in P, then their composition is also unrestricted in P. One can interpret the composition here as a union – if the two sets of variables x and y are unrestricted, then so is their union.

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

No lens is restricted by a literal, since it returns the same value for any state binding.

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

If two lenses are equivalent, and thus they characterise the same state-space regions, then clearly unrestrictions over them are equivalent.

lemma *unrest-equiv*:
  fixes $P :: ('a, '\alpha)$ *uexpr*
  assumes *mwb-lens y x* $\approx_L$ *y x* $\sharp$ *P*
  shows $y \sharp P$
  by (*metis assms lens-equiv-def sublens-pres-mwb sublens-put-put unrest-uexpr.rep-eq*)

The following laws demonstrate the primary motivation for lens independence: a variable expression is unrestricted by another variable only when the two variables are independent. Lens independence thus effectively allows us to semantically characterise when two variables, or sets of variables, are different.

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

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

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

The following laws follow automatically from independence of input and output variables.

lemma *unrest-iuvar-ouvar* [*unrest*]:
  fixes $x :: ('a \Longrightarrow 'a)$
  assumes *vwb-lens y*
  shows $\$x \sharp \$y\acute{}$
  by (*metis prod.collapse unrest-uexpr.rep-eq var.rep-eq var-lookup-out var-update-in*)

lemma *unrest-ouvar-iuvar* [*unrest*]:
  fixes $x :: ('a \Longrightarrow 'a)$
  assumes *vwb-lens y*
  shows $\$x\acute{} \sharp \$y$
  by (*metis prod.collapse unrest-uexpr.rep-eq var.rep-eq var-lookup-in var-update-out*)

Unrestriction distributes through the various function lifting expression constructs; this allows us to prove unrestrictions for the majority of the expression language.

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*)

For convenience, we also prove unrestriction rules for the bespoke operators on equality, numbers, arithmetic etc.

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*)

For a λ-term we need to show that the characteristic function expression does not restrict v for any input value x.

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

Variable substitution, like unrestriction, will be characterised semantically using lenses and state-spaces. Effectively a substitution σ is simply a function on the state-space which can be applied to an expression e using the syntax $\sigma \dagger e$. 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** $\dagger$ *80*)

named-theorems *usubst*

A substitution is simply a transformation on the alphabet; it shows how variables should be mapped to different values. Most of the time these will be homogeneous functions but for flexibility we also allow some operations to be heterogeneous.

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

Application of a substitution simply applies the function σ to the state binding b before it is handed to e as an input. This effectively ensures all variables are updated in e.

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

Substitutions can be updated by associating variables with expressions. We thus create an additional polymorphic constant to represent updating the value of a variable to an expression in a substitution, where the variable is modelled by type $'v$. This again allows us to support different notions of variables, such as deep variables, later.

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

The following function takes a substitution form state-space $'\alpha$ to $'\beta$, a lens with source $'\beta$ and view "'a", and an expression over $'\alpha$ and returning a value of type "$'a$, and produces an updated substitution. It does this by constructing a substitution function that takes state binding b, and updates the state first by applying the original substitution σ, and then updating the part of the state associated with lens x with expression evaluated in the context of b. This effectively means that x is now associated with expression v. We add this definition to our overloaded constant.

definition *subst-upd-uvar* :: $('\alpha,'\beta)$ *psubst* $\Rightarrow$ $('a \Longrightarrow '\beta) \Rightarrow ('a, '\alpha)$ *uexpr* $\Rightarrow ('\alpha,'\beta)$ *psubst* **where**
subst-upd-uvar $\sigma\ x\ v = (\lambda\ b.\ put_x\ (\sigma\ b)\ ([\![v]\!]_e b))$

adhoc-overloading
subst-upd subst-upd-uvar

The next function looks up the expression associated with a variable in a substitution by use of the *get* lens function.

lift-definition *usubst-lookup* :: $('\alpha,'\beta)$ *psubst* $\Rightarrow$ $('a \Longrightarrow '\beta) \Rightarrow ('a, '\alpha)$ *uexpr* $(\langle\text{-}\rangle_s)$

is $\lambda\ \sigma\ x\ b.\ get_x\ (\sigma\ b)$ **.**

Substitutions also exhibit a natural notion of unrestriction which states that σ does not restrict x if application of σ to an arbitrary state ρ will not effect the valuation of x. Put another way, it requires that *put* and the substitution commute.

definition *unrest-usubst* :: $('a \Longrightarrow '\alpha) \Rightarrow 'a\ usubst \Rightarrow bool$
where *unrest-usubst* $x\ \sigma = (\forall\ \varrho\ v.\ \sigma\ (put_x\ \varrho\ v) = put_x\ (\sigma\ \varrho)\ v)$

adhoc-overloading
unrest unrest-usubst

5.2 Syntax translations

We support two kinds of syntax for substitutions, one where we construct a substitution using a maplet-style syntax, with variables mapping to expressions. Such a constructed substitution can be applied to an expression. Alternatively, we support the more traditional notation, $P[\![v/x]\!]$, which also support multiple simultaneous substitutions. We have to use double square brackets as the single ones are already well used.

We set up non-terminals to represent a single substitution maplet, a sequence of maplets, a list of expressions, and a list of alphabets. The parser effectively uses *subst-upd* to construct substitutions from multiple variables.

nonterminal *smaplet* **and** *smaplets* **and** *uexprs* **and** *salphas*

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*[-]))
-psubst :: [*logic*, *svars*, *uexprs*] ⇒ *logic*
-subst :: *logic* ⇒ *uexprs* ⇒ *salphas* ⇒ *logic* ((-[![-'/-!]) [*990*,*0*,*0*] *991*)
-uexprs :: [*logic*, *uexprs*] => *uexprs* (-,/ -)
:: *logic* => *uexprs* (-)
-salphas :: [*salpha*, *salphas*] => *salphas* (-,/ -)
:: *salpha* => *salphas* (-)

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*
-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*

Thus we can write things like $\sigma(x \mapsto_s v)$ to update a variable x in σ with expression v, $[x \mapsto_s e,\ y \mapsto_s f]$ to construct a substitution with two variables, and finally $P[\![v/x]\!]$, the traditional syntax.

We can now express deletion of a substitution maplet.

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

5.3 Substitution application laws

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

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

Evaluation of a substitution expression involves application of the substitution to different variables. Thus we first prove laws for these cases. The simplest substitution, *id*, when applied to any variable x simply returns the variable expression, since *id* has no effect.

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

A substitution update naturally yields the given expression.

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*)

Substitution update is idempotent.

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*)

Substitution updates commute when the lenses are independent.

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*)

A substitution which swaps two independent variables is an injective function.

lemma *swap-usubst-inj*:
fixes x y :: ($'a \Longrightarrow '\alpha$)
assumes *vwb-lens* x *vwb-lens* y $x \bowtie y$
shows *inj* [$x \mapsto_s \&y, y \mapsto_s \&x$]
proof (*rule injI*)
fix b_1 :: $'\alpha$ **and** b_2 :: $'\alpha$
assume [$x \mapsto_s \&y, y \mapsto_s \&x$] b_1 = [$x \mapsto_s \&y, y \mapsto_s \&x$] b_2
hence *a*: put_y (put_x b_1 ($[\![\&y]\!]_e$ b_1)) ($[\![\&x]\!]_e$ b_1) = put_y (put_x b_2 ($[\![\&y]\!]_e$ b_2)) ($[\![\&x]\!]_e$ b_2)
by (*auto simp add*: *subst-upd-uvar-def*)
then have ($\forall$ a b c. put_x (put_y a b) c = put_y (put_x a c) b) $\wedge$
($\forall$ a b. get_x (put_y a b) = get_x a) $\wedge$ ($\forall$ a b. get_y (put_x a b) = get_y a)
by (*simp add*: *assms*(*3*) *lens-indep.lens-put-irr2 lens-indep-comm*)

then show $b_1 = b_2$
by (*metis a assms(1) assms(2) pr-var-def var.rep-eq vwb-lens.source-determination vwb-lens-def wb-lens-def weak-lens.put-get*)
qed

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

lemma *usubst-upd-comm-dash* [*usubst*]:
fixes $x :: ('a \Longrightarrow 'a)$
shows $\sigma(\$x\acute{} \mapsto_s v, \$x \mapsto_s u) = \sigma(\$x \mapsto_s u, \$x\acute{} \mapsto_s v)$
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*)

If a variable is unrestricted in a substitution then it's application has no effect.

lemma *usubst-apply-unrest* [*usubst*]:
$[\![\ vwb\text{-}lens\ x;\ x \sharp \sigma\ ]\!] \Longrightarrow \langle\sigma\rangle_s\ x = var\ x$
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*)

There follows various laws about deleting variables from a substitution.

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

lemma *subst-del-upd-same* [*usubst*]:
$mwb\text{-}lens\ x \Longrightarrow \sigma(x \mapsto_s v)\ -_s\ x = \sigma\ -_s\ x$
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*)

If a variable is unrestricted in an expression, then any substitution of that variable has no effect on the expression .

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*)

Any substitution is a monotonic function.

lemma *subst-mono*: *mono* (*subst* σ)
by (*simp add*: *less-eq-uexpr.rep-eq mono-def subst.rep-eq*)

5.4 Substitution laws

We now prove the key laws that show how a substitution should be performed for every expression operator, including the core function operators, literals, variables, and the arithmetic operators. They are all added to the *usubst* theorem attribute so that we can apply them using the substitution tactic.

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

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*]: $\sigma \dagger (x\ mod\ y) = \sigma \dagger x\ mod\ \sigma \dagger y$
by (*simp add*: *mod-uexpr-def usubst*)

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

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

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

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

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

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

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

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

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

This laws shows the effect of applying one substitution after another – we simply use function composition to compose them.

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

The next law is similar, but shows how such a substitution is to be applied to every updated variable additionally.

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

lemma *subst-singleton*:
fixes $x :: ('a \Longrightarrow 'a)$
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*

5.5 Ordering substitutions

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

definition *var-name-ord* :: $('a \Longrightarrow 'a) \Rightarrow ('b \Longrightarrow 'a) \Rightarrow bool$ **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*

A fact of the form $x \prec_v y$ has no logical information; it simply exists to define a total order on named lenses that is useful for normalisation. The following theorem is simply an instance of the commutativity law for substitutions. However, that law could not be a simplification law as it would cause the simplifier to loop. Assuming that the variable order is a total order then this theorem will not loop.

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*)

5.6 Unrestriction laws

These are the key unrestriction theorems for substitutions and expressions involving substitutions.

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-uexpr-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-uexpr.rep-eq lens-indep-comm*)

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

end

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-uexpr.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

7.1 Predicate type and syntax

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*

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. We similarly use polymorphic constants for the other predicate calculus operators.

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 :: $('a \Longrightarrow '\alpha) \Rightarrow 'p \Rightarrow 'p$
uall :: $('a \Longrightarrow '\alpha) \Rightarrow 'p \Rightarrow 'p$
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 in concert with the literal expression constructor $\ll x \gg$. Both varieties will be needed at various points. Syntactically they are distinguished by a boldface quantifier for the HOL versions (achieved by the "bold" escape in Isabelle).

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 :: *pttrn* ⇒ *logic* ⇒ *logic* (∃ - • - [0, 10] 10)
-ushAll :: *pttrn* ⇒ *logic* ⇒ *logic* (∀ - • - [0, 10] 10)
-ushBEx :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∃ - ∈ - • - [0, 0, 10] 10)
-ushBAll :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* (∀ - ∈ - • - [0, 0, 10] 10)
-ushGAll :: *pttrn* ⇒ *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*
-uex (*-salphaset* (*-salphamk* (*x* +$_L$ *y*))) *P* <= *-uex* (*x* +$_L$ *y*) *P*
-uall x P == *CONST uall x P*
-uall (*-salphaset* (*-salphamk* (*x* +$_L$ *y*))) *P* <= *-uall* (*x* +$_L$ *y*) *P*
-ushEx x P == *CONST ushEx* (λ *x*. *P*)
∃ *x* ∈ *A* • *P* => ∃ *x* • ≪*x*≫ ∈$_u$ *A* ∧ *P*
-ushAll x P == *CONST ushAll* (λ *x*. *P*)
∀ *x* ∈ *A* • *P* => ∀ *x* • ≪*x*≫ ∈$_u$ *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 hierarchy of HOL. Thus the initial definitions take place in the context of type class instantiations. We first introduce our own class called *refine* that will add the refinement operator syntax to the HOL partial order class.

class *refine* = *order*

abbreviation *refineBy* :: 'a::*refine* ⇒ '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. Indeed we make this inversion for all of the lattice operators.

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

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

purge-notation *bot* ($\bot$)
notation *bot* ($\top$)
purge-notation *top* ($\top$)
notation *top* ($\bot$)

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

syntax
-INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* (($3\bigsqcup$-./ -) [0, 10] 10)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (($3\bigsqcup$-∈-./ -) [0, 0, 10] 10)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* (($3\bigsqcap$-./ -) [0, 10] 10)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (($3\bigsqcap$-∈-./ -) [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 $\lambda P\ Q\ A.$ *Lattices.sup* (*P A*) (*Q A*) **.**
lift-definition *inf-uexpr* :: ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr*
is $\lambda P\ Q\ A.$ *Lattices.inf* (*P A*) (*Q A*) **.**
instance
by (*intro-classes*) (*transfer*, *auto*)+
end

instantiation *uexpr* :: (*bounded-lattice*, *type*) *bounded-lattice*
begin
lift-definition *bot-uexpr* :: ('*a*, '*b*) *uexpr* **is** $\lambda\ A.$ *Orderings.bot* **.**
lift-definition *top-uexpr* :: ('*a*, '*b*) *uexpr* **is** $\lambda\ A.$ *Orderings.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), a complete lattice, a completely distribute lattice, and a complete boolean algebra. This equip us with a very complete theory for basic logical propositions.

instance *uexpr* :: (*boolean-algebra*, *type*) *boolean-algebra*
apply (*intro-classes*, *unfold uexpr-defs*; *transfer*, *rule ext*)

apply (*simp-all add*: *sup-inf-distrib1 diff-eq*)
done

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

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* **..**

From the complete lattice, we can also define and give syntax for the fixed-point operators. Like the lattice operators, these are reversed in UTP.

syntax
  -mu :: *pttrn* ⇒ *logic* ⇒ *logic* (μ - ⋅ - [*0*, *10*] *10*)
  -nu :: *pttrn* ⇒ *logic* ⇒ *logic* (ν - ⋅ - [*0*, *10*] *10*)

notation *gfp* (μ)
notation *lfp* (ν)

translations
  ν *X* ⋅ *P* == *CONST lfp* (λ *X*. *P*)
  μ *X* ⋅ *P* == *CONST gfp* (λ *X*. *P*)

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

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

abbreviation *Conj-upred* :: '*α upred set* ⇒ '*α upred* (⋀- [*900*] *900*) **where**
⋀ *A* ≡ ⨅ *A*

abbreviation *Disj-upred* :: '*α upred set* ⇒ '*α upred* (⋁- [*900*] *900*) **where**
⋁ *A* ≡ ⨆ *A*

notation
  conj-upred (**infixr** ∧$_p$ *35*) **and**
  disj-upred (**infixr** ∨$_p$ *30*)

Perhaps slightly confusingly, the UTP infimum is the HOL supremum and vice-versa. This is because, again, in UTP the lattice is inverted due to the definition of refinement and a desire to have miracle at the top, and abort at the bottom.

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

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

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

syntax
-USup :: *pttrn* ⇒ *logic* ⇒ *logic* ($\bigwedge$ - ⋆ - [0, 10] 10)
-USup :: *pttrn* ⇒ *logic* ⇒ *logic* ($\bigsqcup$ - ⋆ - [0, 10] 10)
-USup-mem :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigwedge$ - ∈ - ⋆ - [0, 10] 10)
-USup-mem :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigsqcup$ - ∈ - ⋆ - [0, 10] 10)
-USUP :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigwedge$ - | - ⋆ - [0, 0, 10] 10)
-USUP :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigsqcup$ - | - ⋆ - [0, 0, 10] 10)
-UInf :: *pttrn* ⇒ *logic* ⇒ *logic* ($\bigvee$ - ⋆ - [0, 10] 10)
-UInf :: *pttrn* ⇒ *logic* ⇒ *logic* ($\bigsqcap$ - ⋆ - [0, 10] 10)
-UInf-mem :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigvee$ - ∈ - ⋆ - [0, 10] 10)
-UInf-mem :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigsqcap$ - ∈ - ⋆ - [0, 10] 10)
-UINF :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigvee$ - | - ⋆ - [0, 10] 10)
-UINF :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ($\bigsqcap$ - | - ⋆ - [0, 10] 10)

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

We also define the other predicate operators

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

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

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

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

lift-definition *all* :: $('a \Longrightarrow 'a) \Rightarrow 'a\ upred \Rightarrow 'a\ upred$ **is**

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

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

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

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

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

— 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* $(op \in)\ x\ A$)

declare *true-upred-def* [*upred-defs*]
declare *false-upred-def* [*upred-defs*]
declare *conj-upred-def* [*upred-defs*]
declare *disj-upred-def* [*upred-defs*]
declare *not-upred-def* [*upred-defs*]
declare *diff-upred-def* [*upred-defs*]
declare *subst-upd-uvar-def* [*upred-defs*]
declare *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 $P \lhd b \rhd Q \equiv$ *trop If b P Q*

7.3 Unrestriction Laws

lemma *unrest-allE*:
⟦ &Σ ♯ *P*; *P* = *true* ⟹ *Q*; *P* = *false* ⟹ *Q* ⟧ ⟹ *Q*
by (*pred-auto*)

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-UINF* [*unrest*]:
⟦ (⋀ *i*. *x* ♯ *P*(*i*)); (⋀ *i*. *x* ♯ *Q*(*i*)) ⟧ ⟹ *x* ♯ (⨅ *i* | *P*(*i*) · *Q*(*i*))
by (*pred-auto*)

lemma *unrest-USUP* [*unrest*]:
⟦ (⋀ *i*. *x* ♯ *P*(*i*)); (⋀ *i*. *x* ♯ *Q*(*i*)) ⟧ ⟹ *x* ♯ (⨆ *i* | *P*(*i*) · *Q*(*i*))
by (*pred-auto*)

lemma *unrest-UINF-mem* [*unrest*]:
⟦(⋀ *i*. *i* ∈ *A* ⟹ *x* ♯ *P*(*i*)) ⟧ ⟹ *x* ♯ (⨅ *i*∈*A* · *P*(*i*))
by (*pred-simp*, *metis*)

lemma *unrest-USUP-mem* [*unrest*]:
⟦(⋀ *i*. *i* ∈ *A* ⟹ *x* ♯ *P*(*i*)) ⟧ ⟹ *x* ♯ (⨆ *i*∈*A* · *P*(*i*))
by (*pred-simp*, *metis*)

lemma *unrest-impl* [*unrest*]: ⟦ *x* ♯ *P*; *x* ♯ *Q* ⟧ ⟹ *x* ♯ *P* ⇒ *Q*
by (*pred-auto*)

lemma *unrest-iff* [*unrest*]: ⟦ *x* ♯ *P*; *x* ♯ *Q* ⟧ ⟹ *x* ♯ *P* ⇔ *Q*
by (*pred-auto*)

lemma *unrest-not* [*unrest*]: *x* ♯ (*P* :: *'α upred*) ⟹ *x* ♯ (¬ *P*)
by (*pred-auto*)

The sublens proviso can be thought of as membership below.

lemma *unrest-ex-in* [*unrest*]:
⟦ *mwb-lens y*; *x* ⊆$_L$ *y* ⟧ ⟹ *x* ♯ (∃ *y* · *P*)
by (*pred-auto*)

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *subst-USUP* [*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*]:
$mwb\text{-}lens\ x \Longrightarrow \sigma(x \mapsto_s v) \dagger (\exists\ x \cdot P) = \sigma \dagger (\exists\ x \cdot P)$
by (*pred-auto*)

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-ex-unrest* [*usubst*]:
$x \sharp \sigma \Longrightarrow \sigma \dagger (\exists\ x \cdot P) = (\exists\ x \cdot \sigma \dagger P)$
by (*pred-auto*)

lemma *subst-all-same* [*usubst*]:
$mwb\text{-}lens\ x \Longrightarrow \sigma(x \mapsto_s v) \dagger (\forall\ x \cdot P) = \sigma \dagger (\forall\ x \cdot P)$
by (*simp add*: *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*)

end

8 Predicate Calculus Laws

theory *utp-pred-laws*
imports *utp-pred*
begin

8.1 Propositional Logic

Showing that predicates form a Boolean Algebra (under the predicate operators as opposed to the lattice 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 *taut-false* [*simp*]: *‘false‘* = *False*
by (*pred-auto*)

lemma *upred-eval-taut*:
‘P$[\![\ll b \gg / \&\Sigma]\!]$*‘* = $[\![P]\!]_e\, b$
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*: $P = R \Longrightarrow ((P::'\alpha\ upred) \vee Q) = (R \vee Q)$
by (*pred-auto*)

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

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

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

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

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

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

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

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

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

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

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

lemma *impl-mp1* [*simp*]: $(P \wedge (P \Rightarrow Q)) = (P \wedge Q)$
by (*pred-auto*)

lemma *impl-mp2* [*simp*]: $((P \Rightarrow Q) \wedge P) = (Q \wedge P)$
by (*pred-auto*)

lemma *impl-adjoin*: $((P \Rightarrow Q) \wedge R) = ((P \wedge R \Rightarrow Q \wedge R) \wedge R)$
by (*pred-auto*)

lemma *impl-refine-intro*:
$[\![\ Q_1 \sqsubseteq P_1;\ P_2 \sqsubseteq (P_1 \wedge Q_2)\ ]\!] \Longrightarrow (P_1 \Rightarrow P_2) \sqsubseteq (Q_1 \Rightarrow Q_2)$
by (*pred-auto*)

lemma *impl-disjI*: $[\![\ `P \Rightarrow R`;\ `Q \Rightarrow R`\ ]\!] \Longrightarrow `(P \vee Q) \Rightarrow R`$
by (*rel-auto*)

lemma *conditional-iff*:
$(P \Rightarrow Q) = (P \Rightarrow R) \longleftrightarrow `P \Rightarrow (Q \Leftrightarrow R)`$
by (*pred-auto*)

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

lemma *p-or-not-p* [*simp*]: $(P \vee \neg P) = \mathit{true}$
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*]: $(\neg ((P::'\alpha\ \mathit{upred}) \wedge Q)) = ((\neg P) \vee (\neg Q))$
by (*pred-auto*)

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

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

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

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

lemma *true-not-false* [*simp*]: $\mathit{true} \neq \mathit{false}\ \mathit{false} \neq \mathit{true}$
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 *true-iff* [*simp*]: $(P \Leftrightarrow \mathit{true}) = P$
by (*pred-auto*)

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

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

8.2 Lattice laws

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

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

lemma *UINF-alt-def*:
$(\bigsqcap\ i \mid A(i) \cdot P(i)) = (\bigsqcap\ i \cdot A(i) \wedge P(i))$
by (*rel-auto*)

lemma *USUP-true* [*simp*]: $(\bigsqcup\ P \mid F(P) \cdot true) = true$
by (*pred-auto*)

lemma *UINF-mem-UNIV* [*simp*]: $(\bigsqcap\ x{\in}UNIV \cdot P(x)) = (\bigsqcap\ x \cdot P(x))$
by (*pred-auto*)

lemma *USUP-mem-UNIV* [*simp*]: $(\bigsqcup\ x{\in}UNIV \cdot P(x)) = (\bigsqcup\ x \cdot P(x))$
by (*pred-auto*)

lemma *USUP-false* [*simp*]: $(\bigsqcup\ i \cdot false) = false$
by (*pred-simp*)

lemma *UINF-true* [*simp*]: $(\bigsqcap\ i \cdot true) = true$
by (*pred-simp*)

lemma *UINF-mem-true* [*simp*]: $A \neq \{\} \Longrightarrow (\bigsqcap\ i{\in}A \cdot true) = true$
by (*pred-auto*)

lemma *UINF-false* [*simp*]: $(\bigsqcap\ i \mid P(i) \cdot false) = false$
by (*pred-auto*)

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

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

lemma *UINF-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 *UINF-as-Sup-collect'*: $(\bigsqcap P \cdot f(P)) = (\bigsqcap P.\ f(P))$
apply (*simp add*: *upred-defs bop.rep-eq lit.rep-eq Sup-uexpr-def*)
apply (*pred-simp*)
apply (*simp add*: *full-SetCompr-eq*)
done

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

lemma *USUP-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 *USUP-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 *USUP-as-Inf-collect'*: $(\bigsqcup P \cdot f(P)) = (\bigsqcup P.\ f(P))$
apply (*simp add*: *upred-defs bop.rep-eq lit.rep-eq Sup-uexpr-def*)
apply (*pred-simp*)
apply (*simp add*: *full-SetCompr-eq*)
done

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

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

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

lemma *subst-continuous* [*usubst*]: $\sigma \dagger (\bigsqcap\ A) = (\bigsqcap\ \{\sigma \dagger P \mid P.\ P \in A\})$
by (*simp add*: *UINF-as-Sup*[*THEN sym*] *usubst setcompr-eq-image*)

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

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

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

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

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

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

lemma *conj-UINF-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-UINF-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-USUP-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 USUP.rep-eq inf-uexpr.rep-eq bop.rep-eq lit.rep-eq*)

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

lemma *UINF-all-cong*:
assumes $\bigwedge P.\ F(P) = G(P)$
shows $(\bigsqcap\ P \cdot F(P)) = (\bigsqcap\ P \cdot G(P))$
by (*simp add*: *UINF-as-Sup-collect assms*)

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*: *UINF-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*: *USUP-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 UINF-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 USUP-as-Inf-collect*)

lemma *UINF-impl*: $(\bigsqcap\ P{\in}A \cdot F(P) \Rightarrow G(P)) = ((\bigsqcup\ P{\in}A \cdot F(P)) \Rightarrow (\bigsqcap\ P{\in}A \cdot G(P)))$
by (*pred-auto*)

lemma *UINF-all-nats* [*simp*]:
fixes $P :: nat \Rightarrow {'\alpha}\ upred$
shows $(\bigsqcap\ n \cdot \bigsqcap\ i{\in}\{0..n\} \cdot P(i)) = (\bigsqcap\ i{\in}\{0..\} \cdot P(i))$
by (*pred-auto*)

8.3 Equality laws

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 $x :: ({'a} \Longrightarrow {'\alpha})$
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 $x :: ({'a} \Longrightarrow {'\alpha})$
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 *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 *upred-true-eq* [*simp*]: $(true =_u p) = p$
by (*pred-auto*)

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

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

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

8.4 HOL Variable Quantifiers

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*: $[\![\bigwedge x.\ P\ x = Q\ x\ ]\!] \Longrightarrow shEx\ P = shEx\ Q$
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*: $[\![\bigwedge x.\ P\ x = Q\ x\ ]\!] \Longrightarrow shAll\ P = shAll\ Q$
by (*pred-auto*)

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*)

8.5 Case Splitting

lemma *eq-split-subst*:
assumes *vwb-lens x*
shows $(P = Q) \longleftrightarrow (\forall\ v.\ P[\![\ll v\gg/x]\!] = Q[\![\ll v\gg/x]\!])$

using *assms*
by (*pred-auto*, *metis vwb-lens-wb wb-lens.source-stability*)

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

lemma *taut-split-subst*:
assumes *vwb-lens x*
shows ‘P‘ $\longleftrightarrow$ ($\forall$ *v*. ‘$P[\![\ll v \gg / x]\!]$‘)
using *assms*
by (*pred-auto*, *metis vwb-lens-wb wb-lens.source-stability*)

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

lemma *bool-eq-splitI*:
assumes *vwb-lens x* $P[\![true/x]\!] = Q[\![true/x]\!]$ $P[\![false/x]\!] = Q[\![false/x]\!]$
shows $P = Q$
by (*metis* (*full-types*) *assms eq-split-subst false-alt-def true-alt-def*)

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

8.6 UTP Quantifiers

lemma *one-point*:
assumes *mwb-lens x x* ♯ *v*
shows ($\exists$ *x* · $P \wedge var\ x =_u v$) = $P[\![v/x]\!]$
using *assms*
by (*pred-auto*)

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

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

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

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

8.7 Conditional laws

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *spec-cond-dist*: $(P \Rightarrow (Q \lhd b \rhd R)) = ((P \Rightarrow Q) \lhd b \rhd (P \Rightarrow R))$
by (*pred-auto*)

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

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

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

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

lemma *cond-assign-subst*:
vwb-lens x $\Longrightarrow (P \lhd utp\text{-}expr.var\ x =_u v \rhd Q) = (P[\![v/x]\!] \lhd utp\text{-}expr.var\ x =_u v \rhd Q)$
apply (*rel-simp*) **using** *vwb-lens.put-eq* **by** *force*

8.8 Refinement By Observation

Function to obtain the set of observations of a predicate

definition *obs-upred* :: $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *set* ($[\![\text{-}]\!]_o$)
where [*upred-defs*]: $[\![P]\!]_o = \{b.\ [\![P]\!]_e b\}$

lemma *obs-upred-refine-iff*:
$P \sqsubseteq Q \longleftrightarrow [\![Q]\!]_o \subseteq [\![P]\!]_o$
by (*pred-auto*)

A refinement can be demonstrated by considering only the observations of the predicates which

are relevant, i.e. not unrestricted, for them. In other words, if the alphabet can be split into two disjoint segments, x and y, and neither predicate refers to y then only x need be considered when checking for observations.

lemma *refine-by-obs*:
assumes $x \bowtie y$ *bij-lens* $(x +_L y)$ $y \sharp P$ $y \sharp Q$ $\{v.\ `P[\![\ll v \gg / x]\!]`\} \subseteq \{v.\ `Q[\![\ll v \gg / x]\!]`\}$
shows $Q \sqsubseteq P$
using *assms*(3−5)
apply (*simp add*: *obs-upred-refine-iff subset-eq*)
apply (*pred-simp*)
apply (*rename-tac b*)
apply (*drule-tac* $x{=}get_x b$ **in** *spec*)
apply (*auto simp add*: *assms*)
apply (*metis assms*(1) *assms*(2) *bij-lens.axioms*(2) *bij-lens-axioms-def lens-override-def lens-override-plus*)+
done

8.9 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 $x :: ('a \Longrightarrow 'a)$
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*)

end

9 Fixed-points and Recursion

theory *utp-recursion*

imports *utp-pred-laws*
begin

9.1 Fixed-point Laws

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 (*meson lfp-lowerbound utp-pred-laws.bot.extremum-unique*)

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

lemma *mu-refine-intro*:
assumes $(C \Rightarrow S) \sqsubseteq F(C \Rightarrow S)$ '$C \Rightarrow (\mu\ F \Leftrightarrow \nu\ F)$'
shows $(C \Rightarrow S) \sqsubseteq \mu\ F$
proof −
from *assms* **have** $(C \Rightarrow S) \sqsubseteq \nu\ F$
by (*simp add*: *lfp-lowerbound*)
with *assms* **show** *?thesis*
by (*pred-auto*)
qed

9.2 Obtaining Unique Fixed-points

Obtaining termination proofs via approximation chains. Theorems and proofs adapted from chapter 2, page 63 of the UTP book [4].

type-synonym *'a chain* = *nat* ⇒ *'a upred*

definition *chain* :: *'a chain* ⇒ *bool* **where**
chain $Y = ((Y\ 0 = false) \wedge (\forall\ i.\ Y\ (Suc\ i) \sqsubseteq Y\ i))$

lemma *chain0* [*simp*]: *chain* $Y \Longrightarrow Y\ 0 = false$
by (*simp add*:*chain-def*)

lemma *chainI*:
assumes $Y\ 0 = false \bigwedge i.\ Y\ (Suc\ i) \sqsubseteq Y\ i$
shows *chain* Y
using *assms* **by** (*auto simp add*: *chain-def*)

lemma *chainE*:
assumes *chain* $Y \bigwedge i.\ [\![\ Y\ 0 = false;\ Y\ (Suc\ i) \sqsubseteq Y\ i\ ]\!] \Longrightarrow P$
shows P
using *assms* **by** (*simp add*: *chain-def*)

lemma *L274*:
assumes $\forall\ n.\ ((E\ n \wedge_p X) = (E\ n \wedge Y))$
shows $(\bigsqcap (range\ E) \wedge X) = (\bigsqcap (range\ E) \wedge Y)$
using *assms* **by** (*pred-auto*)

Constructive chains

definition *constr* ::
  (*'a upred* ⇒ *'a upred*) ⇒ *'a chain* ⇒ *bool* **where**
constr F E ⟷ *chain E* ∧ (∀ *X n*. ((*F*(*X*) ∧ *E*(*n* + *1*)) = (*F*(*X* ∧ *E*(*n*)) ∧ *E* (*n* + *1*))))

This lemma gives a way of showing that there is a unique fixed-point when the predicate function can be built using a constructive function F over an approximation chain E

lemma *chain-pred-terminates*:
  assumes *constr F E mono F*
  shows (⨅ (*range E*) ∧ *μ F*) = (⨅ (*range E*) ∧ *ν F*)
proof −
  from *assms* **have** ∀ *n*. (*E n* ∧ *μ F*) = (*E n* ∧ *ν F*)
  proof (*rule-tac allI*)
    fix *n*
    from *assms* **show** (*E n* ∧ *μ F*) = (*E n* ∧ *ν F*)
    proof (*induct n*)
      case *0* **thus** *?case* **by** (*simp add*: *constr-def*)
    next
      case (*Suc n*)
      note *hyp* = *this*
      thus *?case*
      proof −
        have (*E* (*n* + *1*) ∧ *μ F*) = (*E* (*n* + *1*) ∧ *F* (*μ F*))
          using *gfp-unfold*[*OF hyp*(*3*), *THEN sym*] **by** (*simp add*: *constr-def*)
        also from *hyp* **have** ... = (*E* (*n* + *1*) ∧ *F* (*E n* ∧ *μ F*))
          by (*metis conj-comm constr-def*)
        also from *hyp* **have** ... = (*E* (*n* + *1*) ∧ *F* (*E n* ∧ *ν F*))
          by *simp*
        also from *hyp* **have** ... = (*E* (*n* + *1*) ∧ *ν F*)
          by (*metis* (*no-types*, *lifting*) *conj-comm constr-def lfp-unfold*)
        ultimately show *?thesis*
          by *simp*
      qed
    qed
  qed
  thus *?thesis*
    by (*auto intro*: *L274*)
qed

theorem *constr-fp-uniq*:
  assumes *constr F E mono F* ⨅ (*range E*) = *C*
  shows (*C* ∧ *μ F*) = (*C* ∧ *ν F*)
  using *assms*(*1*) *assms*(*2*) *assms*(*3*) *chain-pred-terminates* **by** *blast*

end

10 Alphabet manipulation

theory *utp-alphabet*
  imports
    utp-pred
begin

named-theorems *alpha*

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

10.1 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*, 'β) *uexpr* $\Rightarrow$ ('β, 'α) *lens* $\Rightarrow$ ('*a*, 'α) *uexpr* (**infixr** $\oplus_p$ *95*)
is λ *P x b*. *P* (get_x *b*) **.**

update-uexpr-rep-eq-thms

lemma *aext-twice*: $(P \oplus_p a) \oplus_p b = P \oplus_p (a ;_L b)$
by (*pred-auto*)

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*]:
$(x \mathbin{/} y) \oplus_p a = (x \oplus_p a) \mathbin{/} (y \oplus_p a)$
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 *aext-mono*: $P \sqsubseteq Q \Longrightarrow P \oplus_p a \sqsubseteq Q \oplus_p a$
by (*pred-auto*)

lemma *aext-cont* [*alpha*]: *vwb-lens a* $\Longrightarrow (\bigsqcap A) \oplus_p a = (\bigsqcap P{\in}A.\ \ P \oplus_p a)$
by (*pred-simp*)

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

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

10.2 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 $\lambda\ P\ x\ b.\ P\ (create_x\ b)$ **.**

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*: *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*)

10.3 Alphabet lens laws

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

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

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

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

10.4 Alphabet coercion

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

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

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

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

10.5 Substitution alphabet extension

definition *subst-ext* :: $'a\ usubst \Rightarrow ('a \Longrightarrow 'b) \Rightarrow 'b\ usubst$ (**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*]:
$wb\text{-}lens\ x \Longrightarrow id \oplus_s x = id$
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*)

10.6 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* [*usubst*]:
mwb-lens $x \Longrightarrow id \restriction_s x = id$
by *pred-auto*

lemma *upd-subst-res* [*alpha*]:
mwb-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* [*usubst*]:
mwb-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

11 Lifting expressions

theory *utp-lift*
imports
utp-alphabet
begin

11.1 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* ($\lceil$-$\rceil_<$)
where $\lceil P \rceil_< \equiv P \oplus_p fst_L$

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

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

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

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

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

11.2 Lifting laws

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

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

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

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

11.3 Unrestriction laws

lemma *unrest-dash-var-pre* [*unrest*]:
  fixes $x :: ('a \Longrightarrow 'a)$
  shows $\$x\acute{} \sharp \lceil p \rceil_<$
  by (*pred-auto*)

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

12 Alphabetised relations

theory *utp-rel*
imports
  utp-pred-laws
  utp-recursion
  utp-lift
  utp-tactics
begin

default-sort *type*

consts
  useq :: $'a \Rightarrow 'b \Rightarrow 'c$ (**infixr** ;; *71*)
  uskip :: $'a$ (*II*)

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

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

declare *in*α*-def* [*urel-defs*]
declare *out*α*-def* [*urel-defs*]

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

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

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

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

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

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

lemma *alpha-in-out*:
$\Sigma \approx_L$ *in*$\alpha +_L$ *out*α
by (*metis fst-lens-def fst-snd-id-lens in*α*-def lens-equiv-refl out*α*-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*- $\triangleleft$ - $\triangleright_r$/ -) [*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* (-$^-$ [*999*] *999*)
is $\lambda\ e\ (b1,\ b2).\ e\ (b2,\ b1)$.

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

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*α

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

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 $\lambda\ \sigma\ (A,\ A').\ A' = \sigma(A)$.

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

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

abbreviation *assign-2-r* ::
$('t1 \Longrightarrow 'α) \Rightarrow ('t2 \Longrightarrow 'α) \Rightarrow ('t1, 'α)$ *uexpr* $\Rightarrow ('t2, 'α)$ *uexpr* $\Rightarrow 'α$ *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 :: $('a, 'α)$ *uexpr* $\Rightarrow$ *uexpr-list* (- [*40*] *40*)
-uexpr-list :: $('a, 'α)$ *uexpr* $\Rightarrow$ *uexpr-list* $\Rightarrow$ *uexpr-list* (-,/ - [*70*,*70*] *70*)
-assignment :: *svid-list* $\Rightarrow$ *uexprs* $\Rightarrow 'α$ *hrel* (**infixr** := *72*)
-assignment-upd :: *svid* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infixr** [-] := *72*)
-mk-usubst :: *svid-list* $\Rightarrow$ *uexprs* $\Rightarrow 'α$ *usubst*

translations
-mk-usubst σ (*-svid-unit x*) *v* == $\sigma(\&x \mapsto_s v)$
-mk-usubst σ (*-svid-list x xs*) (*-uexprs v vs*) == (*-mk-usubst* $(\sigma(\&x \mapsto_s v))$ *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*)
x [*k*] := *v* => *x* := $\&x(k \mapsto v)_u$

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

Homogeneous sequential composition

abbreviation *seqh* :: $'α$ *hrel* $\Rightarrow 'α$ *hrel* $\Rightarrow 'α$ *hrel* (**infixr** $;;_h$ *71*) **where**
seqh P Q $\equiv$ (*P* ;; *Q*)

definition *rassume* :: $'α$ *upred* $\Rightarrow 'α$ *hrel* ($-^\top$ [*999*] *999*) **where**
[*urel-defs*]: *rassume c* = *II* $\lhd$ *c* $\rhd_r$ *false*

definition *rassert* :: $'α$ *upred* $\Rightarrow 'α$ *hrel* ($-_\perp$ [*999*] *999*) **where**
[*urel-defs*]: *rassert c* = *II* $\lhd$ *c* $\rhd_r$ *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* $\Rightarrow$ ($'a \Longrightarrow '\alpha$) $\Rightarrow$ $'\alpha$ *hrel* (**infix** $\upharpoonright_\alpha$ *80*) **where**
$P \upharpoonright_\alpha x = (\exists\ \$x \cdot \exists\ \$x\acute{} \cdot P)$

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

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

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

12.1 Unrestriction Laws

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

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

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

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

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

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

lemma *unrest-out*α*-var* [*unrest*]:
$[\![$ *mwb-lens* x; *out*α $\sharp$ (P :: ($'a$, ($'\alpha \times '\beta$)) *uexpr*) $]\!] \Longrightarrow \$x\acute{} \sharp P$
by (*rel-auto*)

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

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

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

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

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

12.2 Substitution laws

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

lemma *subst-seq-right* [*usubst*]:
inα $\sharp$ $\sigma \Longrightarrow \sigma \dagger (P ;; Q) = P ;; (\sigma \dagger Q)$
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 $x :: (bool \Longrightarrow 'a)$
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 $x :: (- \Longrightarrow 'a)$
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 $x :: (- \Longrightarrow 'a)$
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*]:
$\sigma \dagger (P \lhd b \rhd Q) = (\sigma \dagger P \lhd \sigma \dagger b \rhd \sigma \dagger Q)$
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 in*α*-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*: *out*α*-def out-var-def snd-lens-def*)

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

lemma *subst-drop-upd* [*usubst*]:
fixes $x :: ('a \Longrightarrow 'b)$
shows $\lfloor \sigma(\$x \mapsto_s v) \rfloor_s = \lfloor \sigma \rfloor_s(x \mapsto_s \lfloor v \rfloor_<)$
by (*pred-simp*, *simp add*: *in*α*-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 in*α*-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 in*α*-def*)

lemma *unrest-usubst-lift-out* [*unrest*]:
fixes $x :: ('a \Longrightarrow 'b)$
shows $\$x' \sharp \lceil P \rceil_s$
by (*pred-simp*, *auto simp add*: *unrest-usubst-def in*α*-def*)

12.3 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* :: $'\alpha$ *hrel* (*true*$_h$) **where**
truer ≡ *true*

abbreviation *falser* :: $'\alpha$ *hrel* (*false*$_h$) **where**
falser ≡ *false*

lemma *drop-pre-inv* [*simp*]: ⟦ *out*α ♯ *p* ⟧ ⟹ $\lceil\lfloor p\rfloor_<\rceil_<$ = *p*
by (*pred-simp*, *auto simp add*: *out*α*-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* :: $'\alpha$ *cond* ⇒ $'\alpha$ *hrel* ⇒ $'\alpha$ *hrel* (*while*$^\top$ - *do* - *od*) **where**
while$^\top$ *b do P od* = (*ν X* · (*P* ;; *X*) ◁ *b* ▷$_r$ *II*)

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

definition *while-bot* :: $'\alpha$ *cond* ⇒ $'\alpha$ *hrel* ⇒ $'\alpha$ *hrel* (*while*$_\bot$ - *do* - *od*) **where**
while$_\bot$ *b do P od* = (*μ X* · (*P* ;; *X*) ◁ *b* ▷$_r$ *II*)

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

While loops with invariant decoration

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

lemma *comp-cond-left-distr*:
((*P* ◁ *b* ▷$_r$ *Q*) ;; *R*) = ((*P* ;; *R*) ◁ *b* ▷$_r$ (*Q* ;; *R*))
by (*rel-auto*)

lemma *cond-seq-left-distr*:
outα ♯ *b* ⟹ ((*P* ◁ *b* ▷ *Q*) ;; *R*) = ((*P* ;; *R*) ◁ *b* ▷ (*Q* ;; *R*))
by (*rel-auto*)

lemma *cond-seq-right-distr*:
inα ♯ *b* ⟹ (*P* ;; (*Q* ◁ *b* ▷ *R*)) = ((*P* ;; *Q*) ◁ *b* ▷ (*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*]:
false ;; *P* = *false*
by *pred-auto*

lemma *seqr-right-zero* [*simp*]:
P ;; *false* = *false*

by *pred-auto*

Quantale laws for relations

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

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

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

lemma *seq-UINF-distl'*: $P \mathbin{;;} (\bigsqcap Q \cdot F(Q)) = (\bigsqcap Q \cdot P \mathbin{;;} F(Q))$
by (*metis UINF-mem-UNIV seq-UINF-distl*)

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

lemma *seq-UINF-distr'*: $(\bigsqcap P \cdot F(P)) \mathbin{;;} Q = (\bigsqcap P \cdot F(P) \mathbin{;;} Q)$
by (*metis UINF-mem-UNIV seq-UINF-distr*)

lemma *seq-SUP-distl*: $P \mathbin{;;} (\bigsqcap i{\in}A.\ Q(i)) = (\bigsqcap i{\in}A.\ P \mathbin{;;} Q(i))$
by (*metis image-image seq-Sup-distl*)

lemma *seq-SUP-distr*: $(\bigsqcap i{\in}A.\ P(i)) \mathbin{;;} Q = (\bigsqcap i{\in}A.\ P(i) \mathbin{;;} Q)$
by (*simp add*: *seq-Sup-distr*)

lemma *impl-seqr-mono*: $[\![\ \text{`}P \Rightarrow Q\text{`};\ \text{`}R \Rightarrow S\text{`}\ ]\!] \Longrightarrow \text{`}(P \mathbin{;;} R) \Rightarrow (Q \mathbin{;;} S)\text{`}$
by (*pred-blast*)

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

lemma *seqr-monotonic*:
$[\![\ mono\ P;\ mono\ Q\ ]\!] \Longrightarrow mono\ (\lambda\ X.\ P\ X \mathbin{;;} Q\ X)$
by (*simp add*: *mono-def*, *rel-blast*)

lemma *cond-mono*:
$[\![\ P_1 \sqsubseteq P_2;\ Q_1 \sqsubseteq Q_2\ ]\!] \Longrightarrow (P_1 \lhd b \rhd Q_1) \sqsubseteq (P_2 \lhd b \rhd Q_2)$
by (*rel-auto*)

lemma *cond-monotonic*:
$[\![\ mono\ P;\ mono\ Q\ ]\!] \Longrightarrow mono\ (\lambda\ X.\ P\ X \lhd b \rhd Q\ X)$
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-conj-not*: $((P \lhd b \rhd Q) \wedge (\neg\ b)) = (Q \wedge (\neg\ b))$
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 $x :: (bool \Longrightarrow 'a)$
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*]:
$[\![\ mwb\text{-}lens\ x;\ mwb\text{-}lens\ y\ ]\!] \Longrightarrow [\$x \mapsto_s \lceil u \rceil_<] \dagger (y := v) = (x,\ y := u,\ [x \mapsto_s u] \dagger v)$
by (*rel-auto*)

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

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

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

lemma *assign-pred-transfer*:
fixes $x :: ('a \Longrightarrow 'a)$
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* $\Longrightarrow (x := «u»\ ;;\ x := «v») = (x := «v»)$
by (*simp add*: *assigns-comp subst-upd-comp subst-lit usubst-upd-idem*)

lemma *assign-twice*: $[\![\ mwb\text{-}lens\ x;\ x \sharp f\ ]\!] \Longrightarrow (x := e\ ;;\ x := f) = (x := f)$

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 $x :: ('a \Longrightarrow 'a)$
assumes $out\alpha \sharp b$
shows $(x := e ;; (P \lhd b \rhd Q)) = ((x := e ;; P) \lhd (b[\![\lceil e \rceil_< / \$x]\!]) \rhd (x := e ;; Q))$
by (*rel-auto*)

lemma *assign-rcond*:
fixes $x :: ('a \Longrightarrow 'a)$
shows $(x := e ;; (P \lhd b \rhd_r Q)) = ((x := e ;; P) \lhd (b[\![e/x]\!]) \rhd_r (x := e ;; Q))$
by (*rel-auto*)

lemma *assign-r-alt-def*:
fixes $x :: ('a \Longrightarrow 'a)$
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 $\Longrightarrow II = (\$x' =_u \$x \wedge II \restriction_\alpha x)$
by (*rel-simp*, *metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens.get-put*)

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

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

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

lemma *assign-unfold*:
vwb-lens x $\Longrightarrow (x := v) = (\$x' =_u \lceil v \rceil_< \wedge II \restriction_\alpha x)$

apply (*rel-auto*, *auto simp add*: *comp-def*)
using *vwb-lens.put-eq* **by** *fastforce*

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

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

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

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

lemma *seqr-unfold*:
$(P ;; Q) = (\exists\ v \cdot P[\![\ll v \gg / \$\Sigma\,']\!] \land Q[\![\ll v \gg / \$\Sigma]\!])$
by (*rel-auto*)

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

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

lemma *seqr-left-one-point-true*:
assumes *vwb-lens x*
shows $((P \land \$x\,') ;; Q) = (P[\![true/\$x\,']\!] ;; Q[\![true/\$x]\!])$
by (*metis assms seqr-left-one-point true-alt-def upred-eq-true*)

lemma *seqr-left-one-point-false*:
assumes *vwb-lens x*
shows $((P \land \neg \$x\,') ;; Q) = (P[\![false/\$x\,']\!] ;; Q[\![false/\$x]\!])$
by (*metis assms false-alt-def seqr-left-one-point upred-eq-false*)

lemma *seqr-right-one-point-true*:
assumes *vwb-lens x*
shows $(P \mathbin{;;} (\$x \wedge Q)) = (P[\![true/\$x']\!] \mathbin{;;} Q[\![true/\$x]\!])$
by (*metis assms seqr-right-one-point true-alt-def upred-eq-true*)

lemma *seqr-right-one-point-false*:
assumes *vwb-lens x*
shows $(P \mathbin{;;} (\neg \$x \wedge Q)) = (P[\![false/\$x']\!] \mathbin{;;} Q[\![false/\$x]\!])$
by (*metis assms false-alt-def seqr-right-one-point upred-eq-false*)

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

lemma *seqr-bool-split*:
assumes *vwb-lens x*
shows $P \mathbin{;;} Q = (P[\![true/\$x']\!] \mathbin{;;} Q[\![true/\$x]\!] \vee P[\![false/\$x']\!] \mathbin{;;} Q[\![false/\$x]\!])$
using *assms*
by (*subst seqr-middle*[*of x*], *simp-all add*: *true-alt-def false-alt-def*)

lemma *cond-inter-var-split*:
assumes *vwb-lens x*
shows $(P \lhd \$x' \rhd Q) \mathbin{;;} R = (P[\![true/\$x']\!] \mathbin{;;} R[\![true/\$x]\!] \vee Q[\![false/\$x']\!] \mathbin{;;} R[\![false/\$x]\!])$
proof −
have $(P \lhd \$x' \rhd Q) \mathbin{;;} R = ((\$x' \wedge P) \mathbin{;;} R \vee (\neg \$x' \wedge Q) \mathbin{;;} R)$
by (*simp add*: *cond-def seqr-or-distl*)
also have $... = ((P \wedge \$x') \mathbin{;;} R \vee (Q \wedge \neg \$x') \mathbin{;;} R)$
by (*rel-auto*)
also have $... = (P[\![true/\$x']\!] \mathbin{;;} R[\![true/\$x]\!] \vee Q[\![false/\$x']\!] \mathbin{;;} R[\![false/\$x]\!])$
by (*simp add*: *seqr-left-one-point-true seqr-left-one-point-false assms*)
finally show *?thesis* **.**
qed

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

theorem *postcond-equiv*:
$P = (true \mathbin{;;} P) \longleftrightarrow (in\alpha \sharp P)$
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* $p \neq false$
shows $(true ;; p) = true$
using *assms*
apply (*simp add*: *out*α*-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*)

12.4 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 $x :: ('a \Longrightarrow 'a)$
shows $(\$x)^- = \$x\,'$
by *pred-auto*

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

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

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

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

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

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

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

by (*pred-auto*)

lemma *seqr-convr* [*simp*]: $(p \mathbin{;;} q)^- = (q^- \mathbin{;;} p^-)$
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*: $in\alpha \sharp q \Longrightarrow ((P \wedge q) \mathbin{;;} R) = (P \mathbin{;;} (q^- \wedge R))$
by (*rel-auto*)

theorem *seqr-pre-transfer'*:
$((P \wedge \lceil q \rceil_>) \mathbin{;;} R) = (P \mathbin{;;} (\lceil q \rceil_< \wedge R))$
by (*rel-auto*)

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

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

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

lemma *seqr-pre-out*: $out\alpha \sharp p \Longrightarrow ((p \wedge Q) \mathbin{;;} R) = (p \wedge (Q \mathbin{;;} R))$
by (*rel-blast*)

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

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

lemma *seqr-to-conj*: $[\![out\alpha \sharp P; in\alpha \sharp Q]\!] \Longrightarrow (P \mathbin{;;} Q) = (P \wedge Q)$
by (*metis postcond-left-unit seqr-pre-out utp-pred-laws.inf-top.right-neutral*)

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

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

12.5 Assertions and assumptions

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

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

12.6 Frame and antiframe

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

definition *antiframe* :: ('*a*, 'α) *lens* $\Rightarrow$ 'α *hrel* $\Rightarrow$ 'α *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´ ♯ P; $x ♯ 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 = $((P \;;;\; \textit{while } b \textit{ do } P \textit{ od}) \lhd b \rhd_r II)$
proof −
have *m*:*mono* $(\lambda X.\ (P \;;;\; X) \lhd b \rhd_r II)$
by (*auto intro*: *monoI seqr-mono cond-mono*)
have (*while b do P od*) = $(\nu\ X \cdot (P \;;;\; X) \lhd b \rhd_r II)$
by (*simp add*: *while-def*)
also have ... = $((P \;;;\; (\nu\ X \cdot (P \;;;\; X) \lhd b \rhd_r II)) \lhd b \rhd_r II)$
by (*subst lfp-unfold*, *simp-all add*: *m*)
also have ... = $((P \;;;\; \textit{while } b \textit{ do } P \textit{ od}) \lhd b \rhd_r II)$
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*:
$while_\bot\ b\ do\ P\ od = ((P\ ;;\ while_\bot\ b\ do\ P\ od) \lhd b \rhd_r II)$
proof −
have *m*:*mono* $(\lambda X.\ (P\ ;;\ X) \lhd b \rhd_r II)$
by (*auto intro*: *monoI seqr-mono cond-mono*)
have $(while_\bot\ b\ do\ P\ od) = (\mu\ X \cdot (P\ ;;\ X) \lhd b \rhd_r II)$
by (*simp add*: *while-bot-def*)
also have ... $= ((P\ ;;\ (\mu\ X \cdot (P\ ;;\ X) \lhd b \rhd_r II)) \lhd b \rhd_r II)$
by (*subst gfp-unfold*, *simp-all add*: *m*)
also have ... $= ((P\ ;;\ while_\bot\ b\ do\ P\ od) \lhd b \rhd_r II)$
by (*simp add*: *while-bot-def*)
finally show *?thesis* **.**
qed

theorem *while-bot-false*: $while_\bot\ false\ do\ P\ od = II$
by (*simp add*: *while-bot-def mu-const alpha*)

theorem *while-bot-true*: $while_\bot\ true\ do\ P\ od = (\mu\ X \cdot P\ ;;\ X)$
by (*simp add*: *while-bot-def alpha*)

An infinite loop with a feasible body corresponds to a program error (non-termination).

theorem *while-infinite*: $P\ ;;\ true_h = true \Longrightarrow while_\bot\ true\ do\ P\ od = true$
apply (*simp add*: *while-bot-true*)
apply (*rule antisym*)
apply (*simp*)
apply (*rule gfp-upperbound*)
apply (*simp*)
done

12.7 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 $RID :: ('a \Longrightarrow '\alpha) \Rightarrow '\alpha\ hrel \Rightarrow '\alpha\ hrel$
where $RID\ x\ P = ((\exists\ \$x \cdot \exists\ \$x' \cdot P) \wedge \$x' =_u \$x)$

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

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

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

lemma *RID-skip-r*:
vwb-lens x $\Longrightarrow RID(x)(II) = II$

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

lemma *RID-disj*:
RID(*x*)(*P* ∨ *Q*) = (*RID*(*x*)(*P*) ∨ *RID*(*x*)(*Q*))
by (*rel-auto*)

lemma *RID-conj*:
vwb-lens x ⟹ *RID*(*x*)(*RID*(*x*)(*P*) ∧ *RID*(*x*)(*Q*)) = (*RID*(*x*)(*P*) ∧ *RID*(*x*)(*Q*))
by (*rel-auto*)

lemma *RID-assigns-r-diff*:
⟦ *vwb-lens x*; *x* ♯ *σ* ⟧ ⟹ *RID*(*x*)(⟨*σ*⟩$_a$) = ⟨*σ*⟩$_a$
apply (*rel-auto*)
apply (*metis vwb-lens.put-eq*)
apply (*metis vwb-lens-wb wb-lens.get-put wb-lens-weak weak-lens.put-get*)
done

lemma *RID-assign-r-same*:
vwb-lens x ⟹ *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*) = ((∃ $*x* • ∃ $*x*′ • ((∃ $*x* • ∃ $*x*′ • *P*) ∧ $*x*′ =$_u$ $*x*) ;; *Q*) ∧ $*x*′ =$_u$ $*x*)
by (*simp add*: *RID-def usubst*)
also from *assms* **have** ... = ((((∃ $*x* • ∃ $*x*′ • *P*) ∧ (∃ $*x* • $*x*′ =$_u$ $*x*)) ;; (∃ $*x*′ • *Q*)) ∧ $*x*′ =$_u$ $*x*)
by (*rel-auto*)
also from *assms* **have** ... = (((∃ $*x* • ∃ $*x*′ • *P*) ;; (∃ $*x* • ∃ $*x*′ • *Q*)) ∧ $*x*′ =$_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** ... = ((((∃ $*x* • ∃ $*x*′ • *P*) ∧ $*x*′ =$_u$ $*x*) ;; (∃ $*x* • ∃ $*x*′ • *Q*)) ∧ $*x*′ =$_u$ $*x*)
by (*rel-simp*, *metis* (*full-types*) *mwb-lens.put-put vwb-lens-def wb-lens-weak weak-lens.put-get*)
also have ... = ((((∃ $*x* • ∃ $*x*′ • *P*) ∧ $*x*′ =$_u$ $*x*) ;; ((∃ $*x* • ∃ $*x*′ • *Q*) ∧ $*x*′ =$_u$ $*x*)) ∧ $*x*′ =$_u$ $*x*)
by (*rel-simp*, *fastforce*)
also have ... = ((((∃ $*x* • ∃ $*x*′ • *P*) ∧ $*x*′ =$_u$ $*x*) ;; ((∃ $*x* • ∃ $*x*′ • *Q*) ∧ $*x*′ =$_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*)) = ((∃ $*x* • ∃ $*x*′ • *P* ;; ((∃ $*x* • ∃ $*x*′ • *Q*) ∧ $*x*′ =$_u$ $*x*)) ∧ $*x*′

$=_u$ \$x)
  by (*simp add*: *RID-def usubst*)
 also from *assms* **have** ... = (((∃ \$x • P) ;; (∃ \$x • ∃ \$x´ • Q) ∧ (∃ \$x´ • \$x´ $=_u$ \$x)) ∧ \$x´ $=_u$ \$x)
  by (*rel-auto*)
 also from *assms* **have** ... = (((∃ \$x • ∃ \$x´ • P) ;; (∃ \$x • ∃ \$x´ • Q)) ∧ \$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** ... = ((((∃ \$x • ∃ \$x´ • P) ∧ \$x´ $=_u$ \$x) ;; (∃ \$x • ∃ \$x´ • Q)) ∧ \$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 ... = ((((∃ \$x • ∃ \$x´ • P) ∧ \$x´ $=_u$ \$x) ;; ((∃ \$x • ∃ \$x´ • Q) ∧ \$x´ $=_u$ \$x)) ∧ \$x´ $=_u$ \$x)
  by (*rel-simp*, *fastforce*)
 also have ... = ((((∃ \$x • ∃ \$x´ • P) ∧ \$x´ $=_u$ \$x) ;; ((∃ \$x • ∃ \$x´ • Q) ∧ \$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* ⟹ '*α*) ⇒ '*α hrel* ⇒ *bool* (**infix** ♯♯ *20*)
where (*x* ♯♯ *P*) ⟷ (*P* = *RID*(*x*)(*P*))

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

lemma *skip-r-runrest* [*unrest*]:
 vwb-lens x ⟹ *x* ♯♯ *II*
 by (*simp add*: *RID-skip-r unrest-relation-def*)

lemma *assigns-r-runrest*:
 ⟦ *vwb-lens x*; *x* ♯ *σ* ⟧ ⟹ *x* ♯♯ ⟨*σ*⟩$_a$
 by (*simp add*: *RID-assigns-r-diff unrest-relation-def*)

lemma *seq-r-runrest* [*unrest*]:
 assumes *vwb-lens x x* ♯♯ *P x* ♯♯ *Q*
 shows *x* ♯♯ (*P* ;; *Q*)
 by (*metis RID-seq-left assms unrest-relation-def*)

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

lemma *and-runrest* [*unrest*]: ⟦ *vwb-lens x*; *x* ♯♯ *P*; *x* ♯♯ *Q* ⟧ ⟹ *x* ♯♯ (*P* ∧ *Q*)
 by (*metis RID-conj unrest-relation-def*)

lemma *or-runrest* [*unrest*]: ⟦ *x* ♯♯ *P*; *x* ♯♯ *Q* ⟧ ⟹ *x* ♯♯ (*P* ∨ *Q*)
 by (*simp add*: *RID-disj unrest-relation-def*)

12.8 Alphabet laws

lemma *aext-cond* [*alpha*]:
 (*P* ◁ *b* ▷ *Q*) ⊕$_p$ *a* = ((*P* ⊕$_p$ *a*) ◁ (*b* ⊕$_p$ *a*) ▷ (*Q* ⊕$_p$ *a*))
 by (*rel-auto*)

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

wb-lens a $\Longrightarrow$ $((P \mathbin{;;} Q) \oplus_p (a \times_L a)) = ((P \oplus_p (a \times_L a)) \mathbin{;;} (Q \oplus_p (a \times_L a)))$
by (*rel-simp*, *metis wb-lens-weak weak-lens.put-get*)

12.9 Algebraic properties

interpretation *upred-semiring*: *semiring-1*
where *times* = *seqr* **and** *one* = *skip-r* **and** *zero* = $false_h$ **and** *plus* = *Lattices.sup*
by (*unfold-locales*, (*rel-auto*)+)

We introduce the power syntax dervied from semirings

abbreviation *upower* :: $'\alpha$ *hrel* $\Rightarrow$ *nat* $\Rightarrow$ $'\alpha$ *hrel* (**infixr** ^ *80*) **where**
upower P n $\equiv$ *upred-semiring.power P n*

translations
$P \hat{} i <= CONST\ power.power\ II\ op \mathbin{;;} P\ i$
$P \hat{} i <= (CONST\ power.power\ II\ op \mathbin{;;} P)\ i$

Set up transfer tactic for powers

lemma *upower-rep-eq* [*uexpr-transfer-laws*]:
$[\![P \hat{} i]\!]_e b = (b \in (\{p.\ [\![P]\!]_e\ p\}\ \hat{\wedge}\ i))$
proof (*induct i arbitrary*: *P b*)
case *0*
then show *?case*
by (*simp*, *rel-auto*, *simp add*: *Id-fstsnd-eq*)
next
case (*Suc i*)
show *?case*
by (*simp add*: *Suc seqr.rep-eq relpow-commute*)
qed

lemma *upower-rep-eq-alt* [*uexpr-transfer-laws*]:
$[\![power.power\ \langle id \rangle_a\ op \mathbin{;;} P\ i]\!]_e\ b = (b \in (\{p.\ [\![P]\!]_e\ p\}\ \hat{\wedge}\ i))$
by (*metis skip-r-def upower-rep-eq*)

lemma *Sup-power-expand*:
fixes *P* :: *nat* $\Rightarrow$ $'a$::*complete-lattice*
shows $P(0) \sqcap (\bigsqcap i.\ P(i+1)) = (\bigsqcap i.\ P(i))$
proof −
have *UNIV* = *insert* (*0*::*nat*) {*1*..}
by *auto*
moreover have $(\bigsqcap i.\ P(i)) = \bigsqcap\ (P\ `\ UNIV)$
by (*blast*)
moreover have $\bigsqcap\ (P\ `\ insert\ 0\ \{1..\}) = P(0) \sqcap SUPREMUM\ \{1..\}\ P$
by (*simp*)
moreover have *SUPREMUM* {*1*..} *P* = $(\bigsqcap i.\ P(i+1))$
by (*simp add*: *atLeast-Suc-greaterThan*)
ultimately show *?thesis*
by (*simp only*:)
qed

lemma *Sup-upto-Suc*: $(\bigsqcap i{\in}\{0..Suc\ n\}.\ P \hat{} i) = (\bigsqcap i{\in}\{0..n\}.\ P \hat{} i) \sqcap P \hat{} Suc\ n$
proof −
have $(\bigsqcap i{\in}\{0..Suc\ n\}.\ P \hat{} i) = (\bigsqcap i{\in}insert\ (Suc\ n)\ \{0..n\}.\ P \hat{} i)$
by (*simp add*: *atLeast0-atMost-Suc*)
also have ... = $P \hat{} Suc\ n \sqcap (\bigsqcap i{\in}\{0..n\}.\ P \hat{} i)$

by (*simp*)
finally show *?thesis*
by (*simp add*: *Lattices.sup-commute*)
qed

The following two proofs are adapted from the AFP entry Kleene Algebra (Armstrong, Struth, Weber)

lemma *upower-inductl*: $Q \sqsubseteq (P \mathbin{;;} Q \sqcap R) \Longrightarrow Q \sqsubseteq P \mathbin{\hat{}} n \mathbin{;;} R$
proof (*induct n*)
case *0*
then show *?case* **by** (*auto*)
next
case (*Suc n*)
then show *?case*
by (*auto*, *metis* (*no-types*, *hide-lams*) *dual-order.trans order-refl seqr-assoc seqr-mono*)
qed

lemma *upower-inductr*:
assumes $Q \sqsubseteq (R \sqcap Q \mathbin{;;} P)$
shows $Q \sqsubseteq R \mathbin{;;} (P \mathbin{\hat{}} n)$
using *assms* **proof** (*induct n*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
have $R \mathbin{;;} P \mathbin{\hat{}} Suc\ n = (R \mathbin{;;} P \mathbin{\hat{}} n) \mathbin{;;} P$
by (*metis seqr-assoc upred-semiring.power-Suc2*)
also have $Q \mathbin{;;} P \sqsubseteq ...$
using *Suc.hyps assms seqr-mono* **by** *auto*
also have $Q \sqsubseteq ...$
using *assms* **by** *auto*
finally show *?case* **.**
qed

lemma *SUP-atLeastAtMost-first*:
fixes $P :: nat \Rightarrow 'a$::*complete-lattice*
assumes $m \leq n$
shows $(\bigsqcap i{\in}\{m..n\}.\ P(i)) = P(m) \sqcap (\bigsqcap i{\in}\{Suc\ m..n\}.\ P(i))$
by (*metis SUP-insert assms atLeastAtMost-insertL*)

Kleene star

definition *ustar* :: $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* (-* [*999*] *999*) **where**
$P^\star = (\bigsqcap i{\in}\{0..\} \cdot P \mathbin{\hat{}} i)$

lemma *ustar-rep-eq* [*uexpr-transfer-laws*]:
$[\![P^\star]\!]_e\, b = (b \in (\{p.\ [\![P]\!]_e\ p\}^*))$
by (*simp add*: *ustar-def*, *rel-auto*, *simp-all add*: *relpow-imp-rtrancl rtrancl-imp-relpow*)

Omega

definition *uomega* :: $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* (-$^\omega$ [*999*] *999*) **where**
$P^\omega = (\mu\ X \cdot P \mathbin{;;} X)$

12.10 Relation algebra laws

theorem *RA1*: $(P \mathbin{;;} (Q \mathbin{;;} R)) = ((P \mathbin{;;} Q) \mathbin{;;} R)$

using *seqr-assoc* **by** *auto*

theorem *RA2*: $(P \mathbin{;;} II) = P$ $(II \mathbin{;;} P) = P$
by *simp-all*

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

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

theorem *RA5*: $(P \vee Q)^- = (P^- \vee Q^-)$
by (*rel-auto*)

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

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

12.11 Kleene algebra laws

theorem *ustar-unfoldl*: $P^\star \sqsubseteq II \sqcap P;;P^\star$
by (*rel-simp*, *simp add*: *rtrancl-into-trancl2 trancl-into-rtrancl*)

theorem *ustar-inductl*:
assumes $Q \sqsubseteq (R \sqcap P \mathbin{;;} Q)$
shows $Q \sqsubseteq P^\star \mathbin{;;} R$
proof −
have $P^\star \mathbin{;;} R = (\bigsqcap i.\ P \mathbin{\hat{}} i \mathbin{;;} R)$
by (*simp add*: *ustar-def UINF-as-Sup-collect′ seq-SUP-distr*)
also have $Q \sqsubseteq ...$
by (*metis* (*no-types*, *lifting*) *SUP-least assms semilattice-sup-class.sup-commute upower-inductl*)
finally show *?thesis* **.**
qed

theorem *ustar-inductr*:
assumes $Q \sqsubseteq (R \sqcap Q \mathbin{;;} P)$
shows $Q \sqsubseteq R \mathbin{;;} P^\star$
proof −
have $R \mathbin{;;} P^\star = (\bigsqcap i.\ R \mathbin{;;} P \mathbin{\hat{}} i)$
by (*simp add*: *ustar-def UINF-as-Sup-collect′ seq-SUP-distl*)
also have $Q \sqsubseteq ...$
by (*meson SUP-least assms upower-inductr*)
finally show *?thesis* **.**
qed

12.12 Omega algbra

lemma *uomega-induct*:
$P \mathbin{;;} P^\omega \sqsubseteq P^\omega$
by (*simp add*: *uomega-def*, *metis eq-refl gfp-unfold monoI seqr-mono*)

12.13 Relational alphabet extension

lift-definition *rel-alpha-ext* :: $'\beta$ *hrel* $\Rightarrow$ $('\beta \Longrightarrow '\alpha) \Rightarrow$ $'\alpha$ *hrel* (**infix** $\oplus_R$ *65*)

is $\lambda\ P\ x\ (b1,\ b2).\ P\ (get_x\ b1,\ get_x\ b2) \wedge (\forall\ b.\ b1 \oplus_L b\ on\ x = b2 \oplus_L b\ on\ x)$ **.**

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

12.14 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 $\lambda\ P\ b.\ P\ (fst\ b)\ b$ **.**

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

13 Meta-level substitution

theory *utp-meta-subst*
imports *utp-rel*
begin

definition *msubst* :: ($'a \Rightarrow '\alpha$ *upred*) $\Rightarrow$ ($'a$, $'\alpha$) *uexpr* $\Rightarrow$ $'\alpha$ *upred* **where**
[*upred-defs*]: *msubst F v* = $(\bigsqcap\ x \mid \ll x \gg =_u v \cdot F(x))$

syntax
-msubst :: *logic* $\Rightarrow$ *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ((-$[\![$-$\rightarrow$-$]\!]$) [*990*,*0*,*0*] *991*)

translations
-msubst P x v == *CONST msubst* $(\lambda\ x.\ P)\ v$

lemma *msubst-true* [*usubst*]: $true[\![x \rightarrow v]\!] = true$
by (*pred-auto*)

lemma *msubst-false* [*usubst*]: $false[\![x \rightarrow v]\!] = false$
by (*pred-auto*)

lemma *msubst-lit* [*usubst*]: $\ll x \gg [\![x \rightarrow v]\!] = v$
by (*pred-auto*)

lemma *msubst-not* [*usubst*]: $(\neg\ P(x))[\![x \rightarrow v]\!] = (\neg\ ((P\ x)[\![x \rightarrow v]\!]))$
by (*pred-auto*)

lemma *msubst-disj* [*usubst*]: $(P(x) \vee Q(x))[\![x \rightarrow v]\!] = ((P(x))[\![x \rightarrow v]\!] \vee (Q(x))[\![x \rightarrow v]\!])$
by (*pred-auto*)

lemma *msubst-conj* [*usubst*]: $(P(x) \wedge Q(x))[\![x \rightarrow v]\!] = ((P(x))[\![x \rightarrow v]\!] \wedge (Q(x))[\![x \rightarrow v]\!])$
by (*pred-auto*)

lemma *msubst-seq* [*usubst*]: $(P(x)\ ;;\ Q(x))[\![x \rightarrow \ll v \gg]\!] = ((P(x))[\![x \rightarrow \ll v \gg]\!]\ ;;\ (Q(x))[\![x \rightarrow \ll v \gg]\!])$
by (*rel-auto*)

lemma *msubst-unrest* [*unrest*]: $[\![\bigwedge v.\ x \sharp P(v);\ x \sharp k\ ]\!] \Longrightarrow x \sharp P(v)[\![v \rightarrow k]\!]$
by (*pred-auto*)

end

14 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*: $([\![Q \wedge (P \vee S)]\!]_e\ b \wedge ([\![Q]\!]_e\ b \longrightarrow ([\![P]\!]_e\ b = [\![R]\!]_e\ b))) = ([\![Q \wedge (R \vee S)]\!]_e\ b \wedge ([\![Q]\!]_e\ b \longrightarrow ([\![P]\!]_e\ b = [\![R]\!]_e\ b)))$
by *pred-auto*

Simplifications on value equality

lemma *uexpr-eq*: $([\![e_0]\!]_e\ b = [\![e_1]\!]_e\ b) = [\![e_0 =_u e_1]\!]_e\ b$
by *pred-auto*

lemma *uexpr-trans*: $([\![P \wedge e_0 =_u e_1]\!]_e\ b \wedge ([\![P]\!]_e\ b \longrightarrow [\![e_1 =_u e_2]\!]_e\ b)) = ([\![P \wedge e_0 =_u e_2]\!]_e\ b \wedge ([\![P]\!]_e\ b \longrightarrow [\![e_1 =_u e_2]\!]_e\ b))$
by (*pred-auto*)

lemma *uexpr-trans2*: $([\![P \wedge Q \wedge e_0 =_u e_1]\!]_e\ b \wedge ([\![Q]\!]_e\ b \longrightarrow [\![e_1 =_u e_2]\!]_e\ b)) = ([\![P \wedge Q \wedge e_0 =_u e_2]\!]_e\ b \wedge ([\![P]\!]_e\ b \longrightarrow [\![e_1 =_u e_2]\!]_e\ b))$
by (*pred-auto*)

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

lemma *ueqe1*:$[\![\ [\![P]\!]_e\ b \Longrightarrow ([\![R]\!]_e\ b = [\![Q]\!]_e\ b)\ ]\!] \Longrightarrow ([\![P \wedge R]\!]_e\ b \Longrightarrow [\![P \wedge Q]\!]_e\ b)$
by *pred-auto*

lemma *ueqe2*: $([\![P]\!]_e\ b \Longrightarrow ([\![Q]\!]_e\ b = [\![R]\!]_e\ b) \wedge [\![Q \wedge P]\!]_e\ b = [\![R \wedge P]\!]_e\ b)$
$\Longrightarrow$
$([\![P]\!]_e\ b \Longrightarrow ([\![Q]\!]_e\ b = [\![R]\!]_e\ b))$
by *pred-auto*

lemma *ueqe3*: $[\![\ [\![P]\!]_e\ b \Longrightarrow ([\![Q]\!]_e\ b = [\![R]\!]_e\ b)\ ]\!] \Longrightarrow ([\![R \wedge P]\!]_e\ b = [\![Q \wedge P]\!]_e\ b)$
by *pred-auto*

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

lemma *ueqe3-imp*: $(\bigwedge b.\ [\![P]\!]_e\ b \Longrightarrow ([\![Q]\!]_e\ b = [\![R]\!]_e\ b)) \Longrightarrow ((R \wedge P) = (Q \wedge P))$

by *pred-auto*

lemma *ueqe3-imp3*: $(\bigwedge b.\ [\![P]\!]_e\ b \Longrightarrow ([\![Q]\!]_e\ b = [\![R]\!]_e\ b)) \Longrightarrow ((P \wedge Q) = (P \wedge R))$
by *pred-auto*

lemma *ueqe3-imp2*: $[\![\ (\bigwedge b.\ [\![P0 \wedge P1]\!]_e\ b \Longrightarrow [\![Q]\!]_e\ b \Longrightarrow [\![R]\!]_e\ b = [\![S]\!]_e\ b)\ ]\!] \Longrightarrow ((P0 \wedge P1 \wedge (Q \Rightarrow R)) = (P0 \wedge P1 \wedge (Q \Rightarrow S)))$
by *pred-auto*

The following can introduce the binding notation into predicates

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

lemma *disj-bind-dist*: $[\![P \vee Q]\!]_e\ b = ([\![P]\!]_e\ b \vee [\![Q]\!]_e\ b)$
by *pred-auto*

lemma *imp-bind-dist*: $[\![P \Rightarrow Q]\!]_e\ b = ([\![P]\!]_e\ b \longrightarrow [\![Q]\!]_e\ b)$
by *pred-auto*
end

14.1 Relational Hoare calculus

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

named-theorems *hoare*

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

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

lemma *hoare-r-conj* [*hoare*]: $[\![\ \{\!|p|\!\}\,Q\,\{\!|r|\!\}_u;\ \{\!|p|\!\}\,Q\,\{\!|s|\!\}_u\ ]\!] \Longrightarrow \{\!|p|\!\}\,Q\,\{\!|r \wedge s|\!\}_u$
by *rel-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*]: $[\![\ \{\!|b \wedge p|\!\}\,S\,\{\!|q|\!\}_u\ ;\ \{\!|\neg b \wedge p|\!\}\,T\,\{\!|q|\!\}_u\ ]\!] \Longrightarrow \{\!|p|\!\}\,S \lhd b \rhd_r T\,\{\!|q|\!\}_u$
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

14.2 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* = σ † *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* $\lhd$ *b* $\rhd_r$ *Q*) *wp r* = ((*b* $\Rightarrow$ *P wp r*) $\wedge$ ((¬ *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

15 UTP Theories

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

Closure laws for theories

named-theorems *closure*

15.1 Complete lattice of predicates

definition *upred-lattice* :: ('α *upred*) *gorder* ($\mathcal{P}$) **where**
upred-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 :: 'α *upred set*
show $\exists s.$ *is-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-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*)

15.2 Healthiness conditions

type-synonym 'α *health* = 'α *upred* $\Rightarrow$ 'α *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* ($[\![$-$]\!]_H$)
where $[\![H]\!]_H \equiv \{P.\ P \text{ is } H\}$

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-func*:
$[\![\ F \in [\![\mathcal{H}_1]\!]_H \to [\![\mathcal{H}_2]\!]_H;\ P \text{ is } \mathcal{H}_1\ ]\!] \Longrightarrow \mathcal{H}_2(F(P)) = F(P)$
using *Healthy-if* **by** *blast*

lemma *Healthy-apply-closed*:
assumes $F \in [\![H]\!]_H \to [\![H]\!]_H$ *P is H*
shows *F*(*P*) *is H*
using *assms*(*1*) *assms*(*2*) **by** *auto*

lemma *Healthy-set-image-member*:
$[\![\ P \in F$ ‘ $A; \bigwedge x.\ F\ x \text{ is } H\ ]\!] \Longrightarrow P \text{ is } H$
by *blast*

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-nu* [*closure*]:
assumes *mono F* $F \in [\![id]\!]_H \to [\![H]\!]_H$
shows ν *F is H*
by (*metis* (*mono-tags*) *Healthy-def Healthy-func assms eq-id-iff lfp-unfold*)

lemma *Healthy-mu* [*closure*]:
assumes *mono F* $F \in [\![id]\!]_H \to [\![H]\!]_H$
shows μ *F is H*
by (*metis* (*mono-tags*) *Healthy-def Healthy-func assms eq-id-iff gfp-unfold*)

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*

15.3 Properties of healthiness conditions

definition *Idempotent* :: $'\alpha\ health \Rightarrow bool$ **where**
$Idempotent(H) \longleftrightarrow (\forall\ P.\ H(H(P)) = H(P))$

abbreviation *Monotonic* :: $'\alpha\ health \Rightarrow bool$ **where**
$Monotonic(H) \equiv mono\ H$

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 *Healthy-range*: $Idempotent\ H \Longrightarrow range\ H = [\![H]\!]_H$
by (*auto simp add*: *image-def Healthy-if Healthy-Idempotent*, *metis Healthy-if*)

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*: *monoI*)

lemma *Monotonic-comp* [*intro*]:
$[\![Monotonic\ f;\ Monotonic\ g]\!] \Longrightarrow Monotonic\ (f \circ g)$

by (*simp add*: *mono-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 mono-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-laws.inf.assoc utp-pred-laws.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-laws.inf.assoc utp-pred-laws.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-laws.inf-sup-distrib2* **by** *fastforce*

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

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

lemma *WeakConjunctive-Refinement*:
assumes *WeakConjunctive*(HC)
shows $P \sqsubseteq HC(P)$
using *assms* **unfolding** *WeakConjunctive-def* **by** (*metis utp-pred-laws.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 *mono-def* [*upred-defs*]

lemma *Disjunctuous-Monotonic*: *Disjunctuous H* $\Longrightarrow$ *Monotonic H*
by (*metis Disjunctuous-def mono-def semilattice-sup-class.le-iff-sup*)

lemma *ContinuousD* [*dest*]: $[\![$ *Continuous H*; *A* $\neq$ {} $]\!] \Longrightarrow$ *H* ($\bigsqcap$ *A*) = ($\bigsqcap$ *P*$\in$*A*. *H*(*P*))
by (*simp add*: *Continuous-def*)

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* [*closure*]: *Continuous H* $\Longrightarrow$ *Monotonic H*
by (*simp add*: *Continuous-Disjunctous Disjunctuous-Monotonic*)

lemma *Continuous-comp* [*intro*]:
$[\![$ *Continuous f*; *Continuous g* $]\!] \Longrightarrow$ *Continuous* (*f* $\circ$ *g*)
by (*simp add*: *Continuous-def*)

Closure laws derived from continuity

lemma *Sup-Continuous-closed* [*closure*]:
$[\![$ *Continuous H*; $\bigwedge$ *i*. *i* $\in$ *A* $\Longrightarrow$ *P*(*i*) *is H*; *A* $\neq$ {} $]\!] \Longrightarrow$ ($\bigsqcap$ *i*$\in$*A*. *P*(*i*)) *is H*
by (*drule ContinuousD*[*of H P ' A*], *simp add*: *UINF-mem-UNIV*[*THEN sym*] *UINF-as-Sup*[*THEN sym*])
(*metis* (*no-types*, *lifting*) *Healthy-def' SUP-cong image-image*)

lemma *UINF-mem-Continuous-closed* [*closure*]:
$[\![$ *Continuous H*; $\bigwedge$ *i*. *i* $\in$ *A* $\Longrightarrow$ *P*(*i*) *is H*; *A* $\neq$ {} $]\!] \Longrightarrow$ ($\bigsqcap$ *i*$\in$*A* $\cdot$ *P*(*i*)) *is H*
by (*simp add*: *Sup-Continuous-closed UINF-as-Sup-collect*)

lemma *UINF-mem-Continuous-closed-pair* [*closure*]:
assumes *Continuous H* $\bigwedge$ *i j*. (*i*, *j*) $\in$ *A* $\Longrightarrow$ *P i j is H A* $\neq$ {}
shows ($\bigsqcap$ (*i*,*j*)$\in$*A* $\cdot$ *P i j*) *is H*
proof −
have ($\bigsqcap$ (*i*,*j*)$\in$*A* $\cdot$ *P i j*) = ($\bigsqcap$ *x*$\in$*A* $\cdot$ *P* (*fst x*) (*snd x*))
by (*rel-auto*)
also have ... *is H*
by (*metis* (*mono-tags*) *UINF-mem-Continuous-closed assms*(*1*) *assms*(*2*) *assms*(*3*) *prod.collapse*)
finally show *?thesis* **.**
qed

lemma *UINF-mem-Continuous-closed-triple* [*closure*]:
assumes *Continuous H* $\bigwedge$ *i j k*. (*i*, *j*, *k*) $\in$ *A* $\Longrightarrow$ *P i j k is H A* $\neq$ {}
shows ($\bigsqcap$ (*i*,*j*,*k*)$\in$*A* $\cdot$ *P i j k*) *is H*
proof −
have ($\bigsqcap$ (*i*,*j*,*k*)$\in$*A* $\cdot$ *P i j k*) = ($\bigsqcap$ *x*$\in$*A* $\cdot$ *P* (*fst x*) (*fst* (*snd x*)) (*snd* (*snd x*)))
by (*rel-auto*)
also have ... *is H*
by (*metis* (*mono-tags*) *UINF-mem-Continuous-closed assms*(*1*) *assms*(*2*) *assms*(*3*) *prod.collapse*)
finally show *?thesis* **.**
qed

lemma *UINF-Continuous-closed* [*closure*]:

⟦ *Continuous H*; ⋀ *i*. *P*(*i*) *is H* ⟧ ⟹ (⨅ *i* ∙ *P*(*i*)) *is H*
using *UINF-mem-Continuous-closed*[*of H UNIV P*]
by (*simp add*: *UINF-mem-UNIV*)

All continuous functions are also Scott-continuous

lemma *sup-continuous-Continuous* [*closure*]: *Continuous F* ⟹ *sup-continuous F*
by (*simp add*: *Continuous-def sup-continuous-def*)

lemma *Healthy-fixed-points* [*simp*]: *fps* 𝒫 *H* = ⟦*H*⟧$_H$
by (*simp add*: *fps-def upred-lattice-def Healthy-def*)

lemma *USUP-healthy*: *A* ⊆ ⟦*H*⟧$_H$ ⟹ (⨆ *P*∈*A* ∙ *F*(*P*)) = (⨆ *P*∈*A* ∙ *F*(*H*(*P*)))
by (*rule USUP-cong*, *simp add*: *Healthy-subset-member*)

lemma *UINF-healthy*: *A* ⊆ ⟦*H*⟧$_H$ ⟹ (⨅ *P*∈*A* ∙ *F*(*P*)) = (⨅ *P*∈*A* ∙ *F*(*H*(*P*)))
by (*rule UINF-cong*, *simp add*: *Healthy-subset-member*)

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 mono-def*)

15.4 UTP theories hierarchy

typedef ('𝒯, '*α*) *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*, '*α*) == *CONST uthy* :: ('*T*, '*α*) *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 :: ('𝒯, '*α*) *uthy* ⇒ ('*α* × '*α*) *health* (ℋ1)

definition *utp-order* :: $('\alpha \times '\alpha)$ *health* $\Rightarrow$ $'\alpha$ *hrel gorder* **where**
utp-order $H = (\!|$ *carrier* $= \{P.\ P$ *is* $H\}$, *eq* $= (op =)$, *le* $= op \sqsubseteq |\!)$

abbreviation *uthy-order* $T \equiv$ *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) $= op \sqsubseteq$
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*: *mono-def isotone-def utp-weak-partial-order*)

lemma *Mono-utp-orderI*:
$[\![\bigwedge P\ Q.\ [\![P \sqsubseteq Q; P$ *is* $H; Q$ *is* $H]\!] \Longrightarrow F(P) \sqsubseteq F(Q)]\!] \Longrightarrow$ *Mono*$_{\textit{utp-order } H}$ 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* :: $('T_1, 'α)$ *uthy* ⇒ $('T_2, 'α)$ *uthy* ⇒ $('T_1 \times 'T_2, 'α)$ *uthy* (**infixl** $+_T$ *65*) **where**
uthy-plus T_1 T_2 = *uthy*

overloading
prod-hcond == *utp-hcond* :: $('T_1 \times 'T_2, 'α)$ *uthy* ⇒ $('α \times 'α)$ *health*
begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition *prod-hcond* :: $('T_1 \times 'T_2, 'α)$ *uthy* ⇒ $('α \times 'α)$ *upred* ⇒ $('α \times 'α)$ *upred* **where**
prod-hcond $T = \mathcal{H}_{UTHY('T_1, 'α)} \circ \mathcal{H}_{UTHY('T_2, 'α)}$

end

15.5 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 $\mathcal{T}$ :: $('\mathcal{T}, 'α)$ *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** $\mathcal{T}$ :: ('$\mathcal{T}$, 'α) *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* ($\top_1$)
where *utp-top* $\mathcal{T} \equiv$ *top* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-bottom* ($\bot_1$)
where *utp-bottom* $\mathcal{T} \equiv$ *bottom* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-join* (**infixl** $\sqcup_1$ *65*) **where**
utp-join $\mathcal{T} \equiv$ *join* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-meet* (**infixl** $\sqcap_1$ *70*) **where**
utp-meet $\mathcal{T} \equiv$ *meet* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-sup* ($\bigsqcup_1$- [*90*] *90*) **where**
utp-sup $\mathcal{T} \equiv$ *Lattice.sup* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-inf* ($\bigsqcap_1$- [*90*] *90*) **where**
utp-inf $\mathcal{T} \equiv$ *Lattice.inf* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-gfp* ($\boldsymbol{\nu}_1$) **where**
utp-gfp $\mathcal{T} \equiv$ *GFP* (*uthy-order* $\mathcal{T}$)

abbreviation *utp-lfp* ($\boldsymbol{\mu}_1$) **where**
utp-lfp $\mathcal{T} \equiv$ *LFP* (*uthy-order* $\mathcal{T}$)

syntax
-tmu :: *logic* ⇒ *pttrn* ⇒ *logic* ⇒ *logic* ($\boldsymbol{\mu}_1$ - ⋅ - [*0*, *10*] *10*)
-tnu :: *logic* ⇒ *pttrn* ⇒ *logic* ⇒ *logic* ($\boldsymbol{\nu}_1$ - ⋅ - [*0*, *10*] *10*)

notation *gfp* (μ)
notation *lfp* (ν)

translations
$\boldsymbol{\nu}_T\ X \cdot P$ == *CONST utp-lfp* T (λ X. P)
$\boldsymbol{\mu}_T\ X \cdot P$ == *CONST utp-gfp* T (λ X. P)

lemma *upred-lattice-inf*:
Lattice.inf $\mathcal{P}$ $A = \bigsqcap$ 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 *is* $\mathcal{H}$) ∧ F $P \sqsubseteq P$} = {P. (P *is* $\mathcal{H}$) ∧ 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 $\{P.\ (P\ is\ \mathcal{H}) \wedge P \sqsubseteq F\ P\} = \{P.\ (P\ is\ \mathcal{H}) \wedge P \sqsubseteq F\ (\mathcal{H}\ P)\}$
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*: $\top_{\mathcal{P}} = false$
using *ball-UNIV greatest-def* **by** *fastforce*

lemma *upred-bottom*: $\bot_{\mathcal{P}} = true$
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*: $\top = \mathcal{H}(false)$
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*: $\bot = \mathcal{H}(true)$
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 HCond-Cont 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 $\bigsqcap A = (\textit{if } (A = \{\}) \textit{ then } \top \textit{ else } (\bigsqcap A))$
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* [*closure*]:
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*: $\{P.\ (P \textit{ is } \mathcal{H}) \wedge F\ P \sqsubseteq P\} \neq \{\}$
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 $\bigsqcap \{P.\ (P \textit{ is } \mathcal{H}) \wedge F(P) \sqsubseteq P\} \sqsubseteq \bigsqcap \{P.\ F(\mathcal{H}(P)) \sqsubseteq P\}$
proof −

```
    have 1: ⋀ P. F(ℋ(P)) = ℋ(F(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 (ℋ P) ⊑ P
      hence F: (F (ℋ P)) ⊑ (ℋ P)
        by (metis 1 HCond-Mono mono-def)
      show ⨅{P. (P is ℋ) ∧ F P ⊑ P} ⊑ P
      proof (rule Sup-upper2[of F (ℋ P)])
        show F (ℋ P) ∈ {P. (P is ℋ) ∧ F P ⊑ P}
        proof (auto)
          show F (ℋ P) is ℋ
            by (metis 1 Healthy-def)
          show F (F (ℋ P)) ⊑ F (ℋ P)
            using F mono-def assms(1) by blast
        qed
        show F (ℋ P) ⊑ P
          by (simp add: a)
      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 (μ X · F (ℋ X)) ⊑ μ 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 is ℋ; Q is ℋ ⟧ ⟹ (P ;; Q) is ℋ

locale utp-theory-cont-rel = utp-theory-continuous + utp-theory-rel
begin

  lemma seq-cont-Sup-distl:
    assumes P is ℋ A ⊆ ⟦ℋ⟧_H A ≠ {}
    shows P ;; (⨅ A) = ⨅ {P ;; Q | Q. Q ∈ A }
  proof −
    have {P ;; Q | Q. Q ∈ A } ⊆ ⟦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 *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*]: $\mathcal{II}$ *is* $\mathcal{H}$
and *Left-Unit*: P *is* $\mathcal{H} \Longrightarrow (\mathcal{II} ;; P) = P$

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

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

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 $\mathcal{T}$ $P = (\boldsymbol{\mu}_{\mathcal{T}}\ (\lambda\ X.\ (P ;; X)))$

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* ⊆ *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* ⊆ *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*)

15.6 Theory of relations

We can exemplify the creation of a UTP theory with the theory of relations, a trivial theory.

typedecl *REL*
abbreviation *REL* ≡ *UTHY*(*REL*, *'α*)

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*, *'α*) *uthy* ⇒ (*'α* × *'α*) *health*
rel-unit == *utp-unit* :: (*REL*, *'α*) *uthy* ⇒ *'α hrel*
begin

The healthiness condition of a relation is simply identity, since every alphabetised relation is healthy.

definition *rel-hcond* :: (*REL*, *'α*) *uthy* ⇒ (*'α* × *'α*) *upred* ⇒ (*'α* × *'α*) *upred* **where**
rel-hcond T = *id*

The unit of the theory is simply the relational unit.

definition *rel-unit* :: (*REL*, *'α*) *uthy* ⇒ *'α 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*

15.7 Theory links

We can also describe links between theories, such a Galois connections and retractions, using the following notation.

definition *mk-conn* (- ⇐⟨-,-⟩⇒ - [*90*,*0*,*0*,*91*] *91*) **where**
H1 ⇐⟨$\mathcal{H}_1$,$\mathcal{H}_2$⟩⇒ *H2* ≡ (| *orderA* = *utp-order H1*, *orderB* = *utp-order H2*, *lower* = $\mathcal{H}_2$, *upper* = $\mathcal{H}_1$ |)

abbreviation *mk-conn'* (- ←⟨-,-⟩→ - [*90*,*0*,*0*,*91*] *91*) **where**
T1 ←⟨$\mathcal{H}_1$,$\mathcal{H}_2$⟩→ *T2* ≡ $\mathcal{H}_{T1}$ ⇐⟨$\mathcal{H}_1$,$\mathcal{H}_2$⟩⇒ $\mathcal{H}_{T2}$

lemma *mk-conn-orderA* [*simp*]: $\mathcal{X}_{H1}$ ⇐⟨$\mathcal{H}_1$,$\mathcal{H}_2$⟩⇒ *H2* = *utp-order H1*
by (*simp add*:*mk-conn-def*)

lemma *mk-conn-orderB* [*simp*]: $\mathcal{Y}_{H1}$ ⇐⟨$\mathcal{H}_1$,$\mathcal{H}_2$⟩⇒ *H2* = *utp-order H2*
by (*simp add*:*mk-conn-def*)

lemma *mk-conn-lower* [*simp*]: π_* *H1* ⇐⟨$\mathcal{H}_1$,$\mathcal{H}_2$⟩⇒ *H2* = $\mathcal{H}_1$
by (*simp add*: *mk-conn-def*)

lemma *mk-conn-upper* [*simp*]: π^* *H1* ⇐⟨$\mathcal{H}_1$,$\mathcal{H}_2$⟩⇒ *H2* = $\mathcal{H}_2$
by (*simp add*: *mk-conn-def*)

lemma *galois-comp*: (H_2 ⇐⟨$\mathcal{H}_3$,$\mathcal{H}_4$⟩⇒ H_3) ∘$_g$ (H_1 ⇐⟨$\mathcal{H}_1$,$\mathcal{H}_2$⟩⇒ H_2) = H_1 ⇐⟨$\mathcal{H}_1$∘$\mathcal{H}_3$,$\mathcal{H}_4$∘$\mathcal{H}_2$⟩⇒ H_3
by (*simp add*: *comp-galcon-def mk-conn-def*)

Example Galois connection / retract: Existential quantification

lemma *Idempotent-ex*: *mwb-lens x* ⟹ *Idempotent* (*ex x*)
by (*simp add*: *Idempotent-def exists-twice*)

lemma *Monotonic-ex*: *mwb-lens x* ⟹ *Monotonic* (*ex x*)
by (*simp add*: *mono-def ex-mono*)

lemma *ex-closed-unrest*:
vwb-lens x ⟹ ⟦*ex x*⟧$_H$ = {*P*. *x* ♯ *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* ∘ *H* = *H* ∘ *ex x*
shows *retract* ((*ex x* ∘ *H*) ⇐⟨*ex x*, *H*⟩⇒ *H*)
proof (*unfold-locales*, *simp-all*)
show *H* ∈ ⟦*ex x* ∘ *H*⟧$_H$ → ⟦*H*⟧$_H$
using *Healthy-Idempotent assms* **by** *blast*
from *assms*(*1*) *assms*(*3*)[*THEN sym*] **show** *ex x* ∈ ⟦*H*⟧$_H$ → ⟦*ex x* ∘ *H*⟧$_H$
by (*simp add*: *Pi-iff Healthy-def fun-eq-iff exists-twice*)
fix *P Q*
assume *P is* (*ex x* ∘ *H*) *Q is H*
thus (*H P* ⊑ *Q*) = (*P* ⊑ (∃ *x* • *Q*))
by (*metis* (*no-types*, *lifting*) *Healthy-Idempotent Healthy-if assms comp-apply dual-order.trans ex-weakens utp-pred-laws.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

16 Concurrent programming

theory *utp-concurrency*
imports
utp-rel
utp-tactics
utp-theory
begin

16.1 Variable renamings

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*]: $left\text{-}uvar\ x = x\ ;_L\ fst_L\ ;_L\ snd_L$

definition [*upred-defs*]: $right\text{-}uvar\ x = x\ ;_L\ snd_L\ ;_L\ snd_L$

definition [*upred-defs*]: $pre\text{-}uvar\ x = x\ ;_L\ fst_L$

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

lemma *left-uvar-indep-pre-uvar* [*simp*]:
left-uvar x $\bowtie$ *pre-uvar y*
by (*simp add*: *left-uvar-def pre-uvar-def*)

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

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

lemma *right-uvar-indep-pre-uvar* [*simp*]:
right-uvar x $\bowtie$ *pre-uvar y*
by (*simp add*: *right-uvar-def pre-uvar-def*)

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

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

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

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

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

lemma *right-uvar* [*simp*]: *vwb-lens x* ⟹ *vwb-lens* (*right-uvar x*)
by (*simp add*: *right-uvar-def*)

lemma *pre-uvar* [*simp*]: *vwb-lens x* ⟹ *vwb-lens* (*pre-uvar x*)
by (*simp add*: *pre-uvar-def*)

lemma *left-uvar-mwb* [*simp*]: *mwb-lens x* ⟹ *mwb-lens* (*left-uvar x*)
by (*simp add*: *left-uvar-def*)

lemma *right-uvar-mwb* [*simp*]: *mwb-lens x* ⟹ *mwb-lens* (*right-uvar x*)
by (*simp add*: *right-uvar-def*)

lemma *pre-uvar-mwb* [*simp*]: *mwb-lens x* ⟹ *mwb-lens* (*pre-uvar x*)
by (*simp add*: *pre-uvar-def*)

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

translations
-svarpre x == *CONST pre-uvar x*
-svarleft x == *CONST left-uvar x*
-svarright x == *CONST right-uvar x*
-svarpre Σ <= *CONST pre-uvar* 1_L
-svarleft Σ <= *CONST left-uvar* 1_L
-svarright Σ <= *CONST right-uvar* 1_L

16.2 Merge predicates

A merge is then a relation whose input has three parts: the prior variables, the output variables of the left predicate, and the output of the right predicate.

type-synonym $'\alpha$ *merge* = ($'\alpha$ × ($'\alpha$ × $'\alpha$), $'\alpha$) *rel*

skip is the merge predicate which ignores the output of both parallel predicates

definition $skip_m$ :: $'\alpha$ *merge* **where**

[*upred-defs*]: $skip_m = (\$\Sigma' =_u \$\Sigma_<)$

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 $swap_m :: ('\alpha \times '\beta \times '\beta, '\alpha \times '\beta \times '\beta)$ *rel* **where**
[*upred-defs*]: $swap_m = (0{-}\Sigma,1{-}\Sigma := \&1{-}\Sigma,\&0{-}\Sigma)$

The following healthiness condition on merges is used to represent commutativity

abbreviation *SymMerge* :: $'\alpha$ *merge* $\Rightarrow$ $'\alpha$ *merge* **where**
$SymMerge(M) \equiv (swap_m ;; M)$

16.3 Separating simulations

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 $U0\alpha$ **where** [*upred-defs*]: $U0\alpha = (1_L \times_L$ *out-var* $fst_L)$

definition $U1\alpha$ **where** [*upred-defs*]: $U1\alpha = (1_L \times_L$ *out-var* $snd_L)$

abbreviation *U0-alpha-lift* ($\lceil\text{-}\rceil_0$) **where** $\lceil P \rceil_0 \equiv P \oplus_p U0\alpha$

abbreviation *U1-alpha-lift* ($\lceil\text{-}\rceil_1$) **where** $\lceil P \rceil_1 \equiv P \oplus_p U1\alpha$

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 $U0\alpha$*-vwb-lens* [*simp*]: *vwb-lens* $U0\alpha$
by (*simp add*: $U0\alpha$*-def id-vwb-lens prod-vwb-lens*)

lemma $U1\alpha$*-vwb-lens* [*simp*]: *vwb-lens* $U1\alpha$
by (*simp add*: $U1\alpha$*-def id-vwb-lens prod-vwb-lens*)

lemma $U0\alpha$*-indep-right-uvar* [*simp*]: *vwb-lens* $x \Longrightarrow U0\alpha \bowtie$ *out-var* (*right-uvar* x)
by (*force intro*: *plus-pres-lens-indep fst-snd-lens-indep lens-indep-left-comp*
simp add: $U0\alpha$*-def right-uvar-def out-var-def prod-as-plus lens-comp-assoc*[*THEN sym*])

lemma $U1\alpha$*-indep-left-uvar* [*simp*]: *vwb-lens* $x \Longrightarrow U1\alpha \bowtie$ *out-var* (*left-uvar* x)

by (*force intro*: *plus-pres-lens-indep fst-snd-lens-indep lens-indep-left-comp*
simp add: *U1α-def left-uvar-def out-var-def prod-as-plus lens-comp-assoc*[*THEN sym*])

lemma *U0-alpha-lift-bool-subst* [*usubst*]:
$\sigma(\$0\text{-}x\,' \mapsto_s \mathit{true}) \dagger \lceil P \rceil_0 = \sigma \dagger \lceil P[\![\mathit{true}/\$x\,']\!] \rceil_0$
$\sigma(\$0\text{-}x\,' \mapsto_s \mathit{false}) \dagger \lceil P \rceil_0 = \sigma \dagger \lceil P[\![\mathit{false}/\$x\,']\!] \rceil_0$
by (*pred-auto+*)

lemma *U1-alpha-lift-bool-subst* [*usubst*]:
$\sigma(\$1\text{-}x\,' \mapsto_s \mathit{true}) \dagger \lceil P \rceil_1 = \sigma \dagger \lceil P[\![\mathit{true}/\$x\,']\!] \rceil_1$
$\sigma(\$1\text{-}x\,' \mapsto_s \mathit{false}) \dagger \lceil P \rceil_1 = \sigma \dagger \lceil P[\![\mathit{false}/\$x\,']\!] \rceil_1$
by (*pred-auto+*)

lemma *U0-alpha-out-var* [*alpha*]: $\lceil \$x\,' \rceil_0 = \$0\text{-}x\,'$
by (*rel-auto*)

lemma *U1-alpha-out-var* [*alpha*]: $\lceil \$x\,' \rceil_1 = \$1\text{-}x\,'$
by (*rel-auto*)

lemma *U0-skip* [*alpha*]: $\lceil \mathit{II} \rceil_0 = (\$0\text{-}\Sigma\,' =_u \$\Sigma)$
by (*rel-auto*)

lemma *U1-skip* [*alpha*]: $\lceil \mathit{II} \rceil_1 = (\$1\text{-}\Sigma\,' =_u \$\Sigma)$
by (*rel-auto*)

lemma *U0-seqr* [*alpha*]: $\lceil P \;;;\; Q \rceil_0 = P \;;;\; \lceil Q \rceil_0$
by (*rel-auto*)

lemma *U1-seqr* [*alpha*]: $\lceil P \;;;\; Q \rceil_1 = P \;;;\; \lceil Q \rceil_1$
by (*rel-auto*)

lemma *U0α-comp-in-var* [*alpha*]: $(\mathit{in\text{-}var}\ x) ;_L U0\alpha = \mathit{in\text{-}var}\ x$
by (*simp add*: *U0α-def alpha-in-var in-var-prod-lens pre-uvar-def*)

lemma *U0α-comp-out-var* [*alpha*]: $(\mathit{out\text{-}var}\ x) ;_L U0\alpha = \mathit{out\text{-}var}\ (\mathit{left\text{-}uvar}\ x)$
by (*simp add*: *U0α-def alpha-out-var id-wb-lens left-uvar-def out-var-prod-lens*)

lemma *U1α-comp-in-var* [*alpha*]: $(\mathit{in\text{-}var}\ x) ;_L U1\alpha = \mathit{in\text{-}var}\ x$
by (*simp add*: *U1α-def alpha-in-var in-var-prod-lens pre-uvar-def*)

lemma *U1α-comp-out-var* [*alpha*]: $(\mathit{out\text{-}var}\ x) ;_L U1\alpha = \mathit{out\text{-}var}\ (\mathit{right\text{-}uvar}\ x)$
by (*simp add*: *U1α-def alpha-out-var id-wb-lens right-uvar-def out-var-prod-lens*)

16.4 Parallel operators

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**
$P \parallel_s Q \equiv (P \;;;\; U0) \wedge (Q \;;;\; U1) \wedge \$\Sigma_<\,' =_u \$\Sigma$

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*]: $P \parallel_M Q = (P \parallel_s Q ;; M)$

lemma *par-by-merge-alt-def*: $P \parallel_M Q = (\lceil P \rceil_0 \wedge \lceil Q \rceil_1 \wedge \$\Sigma_< ' =_u \$\Sigma) ;; M$
by (*simp add*: *par-by-merge-def U0-as-alpha U1-as-alpha*)

lemma *shEx-pbm-left*: $((\exists\ x \cdot P\ x) \parallel_M Q) = (\exists\ x \cdot (P\ x \parallel_M Q))$
by (*rel-auto*)

lemma *shEx-pbm-right*: $(P \parallel_M (\exists\ x \cdot Q\ x)) = (\exists\ x \cdot (P \parallel_M Q\ x))$
by (*rel-auto*)

16.5 Substitution laws

Substitution is a little tricky because when we push the expression through the composition operator the alphabet of the expression must also change. Consequently for now we only support literal substitution, though this could be generalised with suitable alphabet coercsions. We need quite a number of variants to support this which are below.

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 *lit-pbm-subst* [*usubst*]:
fixes $x :: (- \Longrightarrow 'a)$
shows
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s \ll v \gg) \dagger (P \parallel_M Q) = \sigma \dagger ((P[\![\ll v \gg/\$x]\!]) \parallel_{M[\![\ll v \gg/\$x_<]\!]} (Q[\![\ll v \gg/\$x]\!]))$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x' \mapsto_s \ll v \gg) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M[\![\ll v \gg/\$x']\!]} Q)$
by (*rel-auto*)+

lemma *bool-pbm-subst* [*usubst*]:
fixes $x :: (- \Longrightarrow 'a)$
shows
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s \mathit{false}) \dagger (P \parallel_M Q) = \sigma \dagger ((P[\![\mathit{false}/\$x]\!]) \parallel_{M[\![\mathit{false}/\$x_<]\!]} (Q[\![\mathit{false}/\$x]\!]))$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s \mathit{true}) \dagger (P \parallel_M Q) = \sigma \dagger ((P[\![\mathit{true}/\$x]\!]) \parallel_{M[\![\mathit{true}/\$x_<]\!]} (Q[\![\mathit{true}/\$x]\!]))$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x' \mapsto_s \mathit{false}) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M[\![\mathit{false}/\$x']\!]} Q)$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x' \mapsto_s \mathit{true}) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M[\![\mathit{true}/\$x']\!]} Q)$
by (*rel-auto*)+

lemma *zero-one-pbm-subst* [*usubst*]:
fixes $x :: (- \Longrightarrow 'a)$
shows
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s 0) \dagger (P \parallel_M Q) = \sigma \dagger ((P[\![0/\$x]\!]) \parallel_{M[\![0/\$x_<]\!]} (Q[\![0/\$x]\!]))$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s 1) \dagger (P \parallel_M Q) = \sigma \dagger ((P[\![1/\$x]\!]) \parallel_{M[\![1/\$x_<]\!]} (Q[\![1/\$x]\!]))$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x' \mapsto_s 0) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M[\![0/\$x']\!]} Q)$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x' \mapsto_s 1) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M[\![1/\$x']\!]} Q)$
by (*rel-auto*)+

lemma *numeral-pbm-subst* [*usubst*]:
fixes $x :: (- \Longrightarrow 'a)$
shows
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s \mathit{numeral}\ n) \dagger (P \parallel_M Q) = \sigma \dagger ((P[\![\mathit{numeral}\ n/\$x]\!]) \parallel_{M[\![\mathit{numeral}\ n/\$x_<]\!]}$

$(Q[\![numeral\ n/\$x]\!]))$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x\acute{}\mapsto_s numeral\ n)\ \dagger\ (P\ \|_M\ Q) = \sigma\ \dagger\ (P\ \|_{M[\![numeral\ n/\$x\acute{}]\!]}\ Q)$
by (*rel-auto*)+

16.6 Parallel-by-merge laws

lemma *par-by-merge-false* [*simp*]:
$P\ \|_{false}\ Q = false$
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-seq-add*: $(P\ \|_M\ Q)\ ;;\ R = (P\ \|_{M\ ;;\ R}\ Q)$
by (*simp add*: *par-by-merge-def seqr-assoc*)

A skip parallel-by-merge yields a skip whenever the parallel predicates are both feasible.

lemma *par-by-merge-skip*:
assumes $P\ ;;\ true = true\ Q\ ;;\ true = true$
shows $P\ \|_{skip_m}\ Q = II$
using *assms* **by** (*rel-auto*)

lemma *skip-merge-swap*: $swap_m\ ;;\ skip_m = skip_m$
by (*rel-auto*)

Parallel-by-merge commutes when the merge predicate is unchanged by swap

lemma *par-by-merge-commute-swap*:
shows $P\ \|_M\ Q = Q\ \|_{swap_m\ ;;\ M}\ P$
proof −
have $Q\ \|_{swap_m\ ;;\ M}\ P = ((((Q\ ;;\ U0) \wedge (P\ ;;\ U1) \wedge \$\Sigma_<\acute{}\ =_u\ \$\Sigma)\ ;;\ swap_m)\ ;;\ M)$
by (*simp add*: *par-by-merge-def seqr-assoc*)
also have $... = (((Q\ ;;\ U0\ ;;\ swap_m) \wedge (P\ ;;\ U1\ ;;\ swap_m) \wedge \$\Sigma_<\acute{}\ =_u\ \$\Sigma)\ ;;\ M)$
by (*rel-auto*)
also have $... = (((Q\ ;;\ U1) \wedge (P\ ;;\ U0) \wedge \$\Sigma_<\acute{}\ =_u\ \$\Sigma)\ ;;\ M)$
by (*simp add*: *U0-swap U1-swap*)
also have $... = P\ \|_M\ Q$
by (*simp add*: *par-by-merge-def utp-pred-laws.inf.left-commute*)
finally show *?thesis* **..**
qed

lemma *par-by-merge-commute*:
assumes *M is SymMerge*
shows $P\ \|_M\ Q = Q\ \|_M\ P$
by (*metis Healthy-if assms par-by-merge-commute-swap*)

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 $(P \parallel_M Q_1) \sqsubseteq (P \parallel_M Q_2)$
using *assms* **by** (*rel-blast*)

end

17 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** $\rightarrow_u$ *85*) **where**
$(\sigma, P) \rightarrow_u (\varrho, Q) \longleftrightarrow (\langle\sigma\rangle_a \;;; P) \sqsubseteq (\langle\varrho\rangle_a \;;; Q)$

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

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

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

lemma *assign-trel*:
$(\sigma, x := v) \rightarrow_u (\sigma(x \mapsto_s \sigma \dagger v), II)$
by (*simp add*: *assigns-comp subst-upd-comp*)

lemma *seq-trel*:
assumes $(\sigma, P) \rightarrow_u (\varrho, Q)$
shows $(\sigma, P \;;; R) \rightarrow_u (\varrho, Q \;;; R)$
by (*metis* (*no-types*, *lifting*) *assms order-refl seqr-assoc seqr-mono trel.simps*)

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

lemma *nondet-left-trel*:
$(\sigma, P \sqcap Q) \rightarrow_u (\sigma, P)$
by (*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*:
$(\sigma, P \sqcap Q) \rightarrow_u (\sigma, Q)$
by (*simp add*: *seqr-mono*)

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

lemma *rcond-false-trel*:

assumes $\sigma \dagger b = false$
shows $(\sigma, P \lhd b \rhd_r Q) \rightarrow_u (\sigma, Q)$
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

17.1 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 $('a, '\alpha)\ lvar = ('a\ list \Longrightarrow '\alpha)$

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 :: $('\mathcal{T}, '\alpha)\ uthy \Rightarrow '\beta \Longrightarrow '\alpha$ (**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 :: $('\mathcal{T}, '\alpha)\ uthy \Rightarrow svid$ (**v**$_1$)
-thy-asgn :: $('\mathcal{T}, '\alpha)\ uthy \Rightarrow svid\text{-}list \Rightarrow uexprs \Rightarrow logic$ (**infixr** ::=$_1$ *72*)

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* :: $('a::two, '\alpha)\ lvar \Rightarrow ('a \Longrightarrow '\alpha)$ **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* ⇒ (*'a list* ⟹ *'α*) **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* ⟹ *mwb-lens* (*top-var x*)
by (*simp add*: *list-mwb-lens top-var-def*)

lemma *top-rest-var-indep* [*simp*]:
mwb-lens x ⟹ *top-var x* ⋈ *rest-var x*
by (*simp add*: *lens-indep-left-comp rest-var-def top-var-def*)

lemma *top-var-pres-indep* [*simp*]:
x ⋈ *y* ⟹ *top-var x* ⋈ *y*
by (*simp add*: *lens-indep-left-ext top-var-def*)

syntax
-top-var :: *svid* ⇒ *svid* (@- [*999*] *999*)
-rest-var :: *svid* ⇒ *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* :: (*'T*, *'α*) *uthy* ⇒ (*'a*, *'β*) *lvar* ⇒ *'α hrel* **where**
[*urel-defs*]: *var-begin T x* = *x* ::=$_T$ ⟨*≪undefined≫*⟩ ^$_u$ &*x*

definition *var-end* :: (*'T*, *'α*) *uthy* ⇒ (*'a*, *'β*) *lvar* ⇒ *'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* :: (*'T*, *'α*) *uthy* ⇒ (*'a*, *'α*) *lvar* ⇒ *'α hrel* **where**
[*urel-defs*]: *var-vlet T x* = (($*x* ≠$_u$ ⟨⟩) ∧ $\mathcal{II}_T$)

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* ⇒ *svid* ⇒ *logic* (*var*ı - [*100*] *100*)
-var-begin-asn :: *logic* ⇒ *svid* ⇒ *logic* ⇒ *logic* (*var*ı - := -)
-var-end :: *logic* ⇒ *svid* ⇒ *logic* (*end*ı - [*100*] *100*)
-var-vlet :: *logic* ⇒ *svid* ⇒ *logic* (*vlet*ı - [*100*] *100*)
-var-scope :: *logic* ⇒ *svid* ⇒ *logic* ⇒ *logic* (*var*ı - • - [*0*,*10*] *10*)
-var-scope-ty :: *logic* ⇒ *svid* ⇒ *type* ⇒ *logic* ⇒ *logic* (*var*ı - :: - • - [*0*,*0*,*10*] *10*)
-var-scope-ty-assign :: *logic* ⇒ *svid* ⇒ *type* ⇒ *logic* ⇒ *logic* ⇒ *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* ⋈ *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 $x :: ('a::two, 'β)$ *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* :: $(REL, 'α)$ *uthy* ⇒ $'α \Longrightarrow 'α$
rel-pvar-assigns == *pvar-assigns* :: $(REL, 'α)$ *uthy* ⇒ $'α$ *usubst* ⇒ $'α$ *hrel*
begin
definition *rel-pvar* :: $(REL, 'α)$ *uthy* ⇒ $'α \Longrightarrow 'α$ **where**
[*upred-defs*]: *rel-pvar* $T = 1_L$
definition *rel-pvar-assigns* :: $(REL, 'α)$ *uthy* ⇒ $'α$ *usubst* ⇒ $'α$ *hrel* **where**
[*upred-defs*]: *rel-pvar-assigns* $T\ \sigma = \langle\sigma\rangle_a$
end

interpretation *rel-local-var*: *utp-local-var* $UTHY(REL, 'α)\ TYPE('α)$
proof −
interpret *vw*: *vwb-lens pvar REL* :: $'α \Longrightarrow 'α$
by (*simp add*: *rel-pvar-def id-vwb-lens*)
show *utp-local-var* $TYPE('α)\ UTHY(REL, 'α)$
proof
show $\bigwedge σ::'α ⇒ 'α.\ \langle σ\rangle_{REL}$ *is* $\mathcal{H}_{REL}$
by (*simp add*: *rel-pvar-assigns-def rel-hcond-def Healthy-def*)
show $\bigwedge(σ::'α ⇒ 'α)\ ϱ.\ \langle σ\rangle_{UTHY(REL, 'α)} ;; \langle ϱ\rangle_{REL} = \langle ϱ ∘ σ\rangle_{REL}$
by (*simp add*: *rel-pvar-assigns-def assigns-comp*)
show $\langle id::'α ⇒ 'α\rangle_{UTHY(REL, 'α)} = \mathcal{II}_{REL}$
by (*simp add*: *rel-pvar-assigns-def rel-unit-def skip-r-def*)
qed
qed
end

18 UTP Events

theory *utp-event*
imports *utp-pred*
begin

18.1 Events

Events of some type $'ϑ$ are just the elements of that type.

type-synonym $'ϑ$ *event* = $'ϑ$

18.2 Channels

Typed channels are modelled as functions. Below, $'a$ determines the channel type and $'ϑ$ 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 $('a, 'v)$ *chan* $= 'a \Rightarrow 'v$ *event*

A downside of the approach is that the event type $'v$ 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.

18.2.1 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* :: $('a, 'v)$ *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* ::
$('a, 'v)$ *chan* $\Rightarrow$ $('a, 'a)$ *uexpr* $\Rightarrow$ $('v$ *event*, $'a)$ *uexpr* $('(\cdot\!\cdot/\text{-}')_u)$ **where**
[*upred-defs*]: $(c \cdot e)_u = \ll c \gg (\!| e |\!)_u$
end

19 Meta-theory for the Standard Core

theory *utp*
imports
utp-var
utp-expr
utp-unrest
utp-subst
utp-meta-subst
utp-alphabet
utp-lift
utp-pred
utp-pred-laws
utp-recursion
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.

[4] T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[5] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *CPP*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.

[6] M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In *UTP 2006*, volume 4010 of *LNCS*, pages 123–140. Springer, 2007.

[7] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2006. YCST-2006-02.