Isabelle/UTP: Mechanised reasoning for the UTP

Simon Foster   Frank Zeyda

September 13, 2017

Contents

1 Parser Utilities 4

2 UTP variables 5
2.1 Initial syntax setup . . . 6
2.2 Variable foundations . . . 6
2.3 Variable lens properties . . . 7
2.4 Lens simplifications . . . 8
2.5 Syntax translations . . . 8

3 UTP expressions 10
3.1 Expression type . . . 10
3.2 Core expression constructs . . . 11
3.3 Type class instantiations . . . 12
3.4 Overloaded expression constructors . . . 16
3.5 Syntax translations . . . 17
3.6 Lifting set collectors . . . 20
3.7 Lifting limits . . . 20
3.8 Evaluation laws for expressions . . . 21
3.9 Misc laws . . . 22
3.10 Literalise tactics . . . 22

4 Unrestriction 23
4.1 Definitions and Core Syntax . . . 24
4.2 Unrestriction laws . . . 24

5 Substitution 26
5.1 Substitution definitions . . . 26
5.2 Syntax translations . . . 28
5.3 Substitution application laws . . . 29
5.4 Substitution laws . . . 31
5.5 Ordering substitutions . . . 33
5.6 Unrestriction laws . . . 33
5.7 Parallel Substitution Laws . . . 34

6 UTP Tactics 35
- 6.1 Theorem Attributes . . . 35
- 6.2 Generic Methods . . . 35
- 6.3 Transfer Tactics . . . 36
  - 6.3.1 Robust Transfer . . . 36
  - 6.3.2 Faster Transfer . . . 36
- 6.4 Interpretation . . . 37
- 6.5 User Tactics . . . 37

7 Alphabetised Predicates 39
- 7.1 Predicate type and syntax . . . 39
- 7.2 Predicate operators . . . 40
- 7.3 Unrestriction Laws . . . 45
- 7.4 Substitution Laws . . . 47

8 Predicate Calculus Laws 48
- 8.1 Propositional Logic . . . 49
- 8.2 Lattice laws . . . 51
- 8.3 Equality laws . . . 55
- 8.4 HOL Variable Quantifiers . . . 56
- 8.5 Case Splitting . . . 56
- 8.6 UTP Quantifiers . . . 57
- 8.7 Variable Restriction . . . 59
- 8.8 Conditional laws . . . 59
- 8.9 Additional Expression Laws . . . 61
- 8.10 Refinement By Observation . . . 62
- 8.11 Cylindric Algebra . . . 62

9 Fixed-points and Recursion 63
- 9.1 Fixed-point Laws . . . 63
- 9.2 Obtaining Unique Fixed-points . . . 63

10 UTP Events 65
- 10.1 Events . . . 65
- 10.2 Channels . . . 65
  - 10.2.1 Operators . . . 65

11 Alphabet Manipulation 66
- 11.1 Preliminaries . . . 66
- 11.2 Alphabet Extrusion . . . 66
- 11.3 Alphabet Restriction . . . 68
- 11.4 Alphabet Lens Laws . . . 69
- 11.5 Substitution Alphabet Extension . . . 70
- 11.6 Substitution Alphabet Restriction . . . 70

12 Lifting expressions 71
- 12.1 Lifting definitions . . . 71
- 12.2 Lifting Laws . . . 71
- 12.3 Substitution Laws . . . 72
- 12.4 Unrestriction laws . . . 72

13 Alphabetised Relations **72**
13.1 Relational Alphabets . . . 73
13.2 Relational Types and Operators . . . 74
13.3 Syntax Translations . . . 76
13.4 Relation Properties . . . 77
13.5 Unrestriction Laws . . . 78
13.6 Substitution laws . . . 79
13.7 Alphabet laws . . . 80
13.8 Relational unrestriction . . . 80
13.9 Relational alphabet extension . . . 83

14 Meta-level substitution **83**

15 UTP Deduction Tactic **85**

16 Relational Calculus Laws **87**
16.1 Conditional Laws . . . 87
16.2 Precondition and Postcondition Laws . . . 88
16.3 Sequential Composition Laws . . . 88
16.4 Iterated Sequential Composition Laws . . . 91
16.5 Quantale Laws . . . 91
16.6 Skip Laws . . . 92
16.7 Assignment Laws . . . 92
16.8 Converse Laws . . . 94
16.9 Assertion and Assumption Laws . . . 95
16.10 While Loop Laws . . . 96
16.11 Algebraic Properties . . . 97
16.11.1 Kleene Star . . . 98
16.11.2 Omega . . . 98
16.12 Relation Algebra Laws . . . 99
16.13 Kleene Algebra Laws . . . 99
16.14 Omega Algebra Laws . . . 100
16.15 Relational Hoare calculus . . . 100
16.16 Weakest precondition calculus . . . 101

17 UTP Theories **102**
17.1 Complete lattice of predicates . . . 102
17.2 Healthiness conditions . . . 103
17.3 Properties of healthiness conditions . . . 104
17.4 UTP theories hierarchy . . . 107
17.5 UTP theory hierarchy . . . 109
17.6 Theory of relations . . . 116
17.7 Theory links . . . 117

18 Concurrent Programming **118**
18.1 Variable Renamings . . . 118
18.2 Merge Predicates . . . 120
18.3 Separating Simulations . . . 120
18.4 Associative Merges . . . 122
18.5 Parallel Operators . . . 122

18.6 Unrestriction Laws . . . 123
18.7 Substitution laws . . . 123
18.8 Parallel-by-merge laws . . . 124
18.9 Example: Simple State-Space Division . . . 125

19 Relational operational semantics 126
19.1 Variable blocks . . . 127

20 Meta-theory for the Standard Core 131

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
  ../utils/utp-imports
  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** $\sqsubseteq$ı *50*) **and**
  Lattice.sup ($\bigsqcup$ı- [*90*] *90*) **and**
  Lattice.inf ($\bigsqcap$ı- [*90*] *90*) **and**
  Lattice.join (**infixl** $\sqcup$ı *65*) **and**
  Lattice.meet (**infixl** $\sqcap$ı *70*) **and**
  LFP (μı) **and**
  GFP (νı) **and**
  Set.member (*op* :) **and**
  Set.member ((-/ : -) [*51*, *51*] *50*) **and**
  disj (**infixr** | *30*)

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 [3, 4] 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* :: $('a \Longrightarrow {'\alpha}) \Rightarrow ('a \Longrightarrow {'\alpha} \times {'\beta})$ **where**
[*lens-defs*]: *in-var* $x = x ;_L fst_L$

definition *out-var* :: $('a \Longrightarrow {'\beta}) \Rightarrow ('a \Longrightarrow {'\alpha} \times {'\beta})$ **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* :: $({'\alpha} \Longrightarrow {'\alpha})$ (Σ) **where**
univ-alpha $\equiv 1_L$

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

definition *pr-var* :: $('a \Longrightarrow '\beta) \Rightarrow ('a \Longrightarrow '\beta)$ **where**
[*lens-defs*]: *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 *pr-var-mwb-lens* [*simp*]:
mwb-lens $x \Longrightarrow$ *mwb-lens* (*pr-var* x)
by (*simp add*: *pr-var-def*)

lemma *pr-var-vwb-lens* [*simp*]:
vwb-lens $x \Longrightarrow$ *vwb-lens* (*pr-var* x)
by (*simp add*: *pr-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 *pr-var-indeps* [*simp*]:
$x \bowtie y \Longrightarrow$ *pr-var* $x \bowtie y$
$x \bowtie y \Longrightarrow x \bowtie$ *pr-var* y
by (*simp-all add*: *pr-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*)

lemma *in-var-pr-var* [*simp*]:
in-var (*pr-var* x) = *in-var* x
by (*simp add*: *pr-var-def*)

lemma *out-var-pr-var* [*simp*]:
out-var (*pr-var* x) = *out-var* x
by (*simp add*: *pr-var-def*)

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** *svids* **and** *svar* **and** *svars* **and** *salpha*

These non-terminals correspond to the following syntactic entities. Non-terminal *svid* is an atomic variable identifier, and *svids* is a list of 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-unit :: *svid* ⇒ *svids* (-)
-svid-list :: *svid* ⇒ *svids* ⇒ *svids* (-,/ -)
-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 variable 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 Σ ↽ *CONST svar CONST id-lens*
-sinvar Σ ↽ *CONST ivar* 1_L
-soutvar Σ ↽ *CONST ovar* 1_L
-spvar (*-svid-dot x y*) ↽ *CONST svar* (*CONST lens-comp y x*)
-sinvar (*-svid-dot x y*) ↽ *CONST ivar* (*CONST lens-comp y x*)
-soutvar (*-svid-dot x y*) ↽ *CONST ovar* (*CONST lens-comp y x*)
-spvar x ⇌ *CONST svar x*
-sinvar x ⇌ *CONST ivar x*
-soutvar x ⇌ *CONST ovar x*

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

— Quotations
-ualpha-set A ⇀ *A*
-svar x ⇀ *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 [6], which allows us to reuse much of the existing library of HOL functions.

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

setup-lifting *type-definition-uexpr*

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

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

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 $\mathit{var} :: ('t \Longrightarrow '\alpha) \Rightarrow ('t,\ '\alpha)\ \mathit{uexpr}$ **is** *lens-get* **.**

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

lift-definition $\mathit{lit} :: 't \Rightarrow ('t,\ '\alpha)\ \mathit{uexpr}$ **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 $\mathit{uop} :: ('a \Rightarrow 'b) \Rightarrow ('a,\ '\alpha)\ \mathit{uexpr} \Rightarrow ('b,\ '\alpha)\ \mathit{uexpr}$
is $\lambda\ f\ e\ b.\ f\ (e\ b)$ **.**
lift-definition *bop* ::
$('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a,\ '\alpha)\ \mathit{uexpr} \Rightarrow ('b,\ '\alpha)\ \mathit{uexpr} \Rightarrow ('c,\ '\alpha)\ \mathit{uexpr}$
is $\lambda\ f\ u\ v\ b.\ f\ (u\ b)\ (v\ b)$ **.**
lift-definition *trop* ::
$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow ('a,\ '\alpha)\ \mathit{uexpr} \Rightarrow ('b,\ '\alpha)\ \mathit{uexpr} \Rightarrow ('c,\ '\alpha)\ \mathit{uexpr} \Rightarrow ('d,\ '\alpha)\ \mathit{uexpr}$
is $\lambda\ f\ u\ v\ w\ b.\ f\ (u\ b)\ (v\ b)\ (w\ b)$ **.**
lift-definition *qtop* ::
$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e) \Rightarrow$
$('a,\ '\alpha)\ \mathit{uexpr} \Rightarrow ('b,\ '\alpha)\ \mathit{uexpr} \Rightarrow ('c,\ '\alpha)\ \mathit{uexpr} \Rightarrow ('d,\ '\alpha)\ \mathit{uexpr} \Rightarrow$
$('e,\ '\alpha)\ \mathit{uexpr}$
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 $\mathit{ulambda} :: ('a \Rightarrow ('b,\ '\alpha)\ \mathit{uexpr}) \Rightarrow ('a \Rightarrow 'b,\ '\alpha)\ \mathit{uexpr}$

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* :: $('a,\ '\alpha)$ *uexpr* $\Rightarrow$ $('a,\ '\alpha)$ *uexpr* $\Rightarrow$ $(bool,\ '\alpha)$ *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* $\Rightarrow$ *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* :: $('a, 'b)\ uexpr \Rightarrow ('a, 'b)\ uexpr \Rightarrow ('a, 'b)\ uexpr$ **where**
divide-uexpr $u\ v = bop\ divide\ u\ v$
instance ..
end

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

instantiation *uexpr* :: (*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* $\Rightarrow$ $('a,\ 'b)$ *uexpr* $\Rightarrow$ *bool*
 is $\lambda\ P\ Q.\ (\forall\ A.\ P\ A \leq Q\ A)$ **.**

definition *less-uexpr* :: $('a, 'b)$ *uexpr* $\Rightarrow$ $('a, 'b)$ *uexpr* $\Rightarrow$ *bool*
where *less-uexpr* $P\ Q = (P \leq Q \wedge \neg\ Q \leq P)$
instance ..
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*: $[\![\mathit{numeral}\ x]\!]_e\ b = \mathit{numeral}\ x$
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*: $\mathit{numeral}\ x = \ll\mathit{numeral}\ x\gg$
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 :: $('\alpha::\mathit{trace}, 'a)$ *uexpr*
shows $a - 0 = a$
by (*simp add*: *minus-uexpr-def zero-uexpr-def*, *transfer*, *auto*)

lemma *uexpr-add-diff-cancel-left* [*simp*]:
fixes $a\ b$ :: $('\alpha::\mathit{trace}, 'a)$ *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 :: $'f \Rightarrow 'k \Rightarrow 'v$
— Function update, map update, list update...
uupd :: $'f \Rightarrow 'k \Rightarrow 'v \Rightarrow 'f$
— Domain of maps, lists...
udom :: $'f \Rightarrow 'a\ set$
— Range of maps, lists...
uran :: $'f \Rightarrow 'b\ set$
— Domain restriction
udomres :: $'a\ set \Rightarrow 'f \Rightarrow 'f$
— Range restriction
uranres :: $'f \Rightarrow 'b\ set \Rightarrow 'f$
— Collection cardinality
ucard :: $'f \Rightarrow nat$
— Collection summation
usums :: $'f \Rightarrow 'a$
— Construct a collection from a list of entries
uentries :: $'k\ set \Rightarrow ('k \Rightarrow 'v) \Rightarrow 'f$

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

definition *fun-apply* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$
where *fun-apply f x* = *f x*

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

definition *ffun-entries* :: $'k\ set \Rightarrow ('k \Rightarrow 'v) \Rightarrow ('k, 'v)\ ffun$ **where**
ffun-entries d f = *graph-ffun* $\{(k, f\ k) \mid k.\ k \in d\}$

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-augment* **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* **and** *usums pfun-sum* **and**
uentries pfun-entries **and** *uentries ffun-entries*

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 (*input*) *ulens-override x f g* $\equiv$ *lens-override f g x*

translations
0 <= CONST uempty — We have to do this so we don't see uempty. Is there a better way of printing?

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

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

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

translations
λ *x* $\cdot$ *p* == *CONST ulambda* (λ *x*. *p*)
x $:_u$ *'a* == *x* :: (*'a*, -) *uexpr*
-ulens-ovrd f g a => *CONST bop* (*CONST ulens-override a*) *f g*
-ulens-ovrd f g a <= *CONST bop* (λ*x y*. *CONST lens-override x1 y1 a*) *f g*

syntax — Tuples
-utuple :: (*'a*, *'α*) *uexpr* $\Rightarrow$ *utuple-args* $\Rightarrow$ (*'a* $*$ *'b*, *'α*) *uexpr* ((*1'*(-,/ -*'*)$_u$))
-utuple-arg :: (*'a*, *'α*) *uexpr* $\Rightarrow$ *utuple-args* (-)
-utuple-args :: (*'a*, *'α*) *uexpr* => *utuple-args* $\Rightarrow$ *utuple-args* (-,/ -)
-uunit :: (*'a*, *'α*) *uexpr* (*'*(*'*)$_u$)
-ufst :: (*'a* $\times$ *'b*, *'α*) *uexpr* $\Rightarrow$ (*'a*, *'α*) *uexpr* (π_1*'*(-*'*))
-usnd :: (*'a* $\times$ *'b*, *'α*) *uexpr* $\Rightarrow$ (*'b*, *'α*) *uexpr* (π_2*'*(-*'*))

translations
$()_u$ == $\ll()\gg$
$(x, y)_u$ == *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
-uundef :: *logic* ($\perp_u$)
-umap-empty :: *logic* ($[]_u$)
-uapply :: (*'a* $\Rightarrow$ *'b*, *'α*) *uexpr* $\Rightarrow$ *utuple-args* $\Rightarrow$ (*'b*, *'α*) *uexpr* (-*'*(-*'*)$_a$ [*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* $\Rightarrow$ *logic* ($\#_u$*'*(-*'*))
-uless :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infix** $<_u$ *50*)
-uleq :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infix** $\leq_u$ *50*)
-ugreat :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infix** $>_u$ *50*)
-ugeq :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infix** $\geq_u$ *50*)
-uceil :: *logic* $\Rightarrow$ *logic* ($\lceil$-$\rceil_u$)
-ufloor :: *logic* $\Rightarrow$ *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'(-, -'))
-uentries :: logic ⇒ logic ⇒ logic ($entr_u$'(-,-'))

translations
— Pretty printing for adhoc-overloaded constructs
$f(x)_a$ <= 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
$f(x,y,z,u)_a$ == CONST bop CONST uapply f $(x,y,z,u)_u$
$f(x,y,z)_a$ == CONST bop CONST uapply f $(x,y,z)_u$
$f(x,y)_a$ == CONST bop CONST uapply f $(x,y)_u$
$f(x)_a$ == 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≫
$\bot_u$ == ≪CONST undefined≫
$A \lhd_u f$ == CONST bop (CONST udomres) A f
$f \rhd_u A$ == CONST bop (CONST uranres) f A
$entr_u(d,f)$ == CONST bop CONST uentries d ≪f≫
-UMapUpd m (-UMaplets xy ms) == -UMapUpd (-UMapUpd m xy) ms
-UMapUpd m (-umaplet x y) == CONST trop CONST uupd m x y
-UMap ms == -UMapUpd $[]_u$ ms
-UMap (-UMaplets ms1 ms2) <= -UMapUpd (-UMap ms1) ms2
-UMaplets ms1 (-UMaplets ms2 ms3) <= -UMaplets (-UMaplets ms1 ms2) ms3

— 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** $\upharpoonleft_u$ 75)
$-uextract$ :: ('a set, 'α) uexpr ⇒ ('a list, 'α) uexpr ⇒ ('a list, 'α) uexpr (**infixl** $\upharpoonright_u$ 75)
$-uelems$ :: ('a list, 'α) uexpr ⇒ ('a set, 'α) uexpr ($elems_u$'(-'))
$-usorted$ :: ('a list, 'α) uexpr ⇒ (bool, 'α) uexpr ($sorted_u$'(-'))
$-udistinct$ :: ('a list, 'α) uexpr ⇒ (bool, 'α) uexpr ($distinct_u$'(-'))
$-uupto$ :: logic ⇒ logic ⇒ logic (⟨-..-⟩)
$-uupt$ :: logic ⇒ logic ⇒ logic (⟨-..<-⟩)
$-umap$ :: logic ⇒ logic ⇒ logic (map_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
$xs \upharpoonleft_u A$ == CONST bop CONST seq-filter xs A
$A \upharpoonright_u xs$ == CONST bop (op $\upharpoonright_l$) A xs
⟨n..k⟩ == CONST bop CONST upto n k
⟨n..<k⟩ == CONST bop CONST upt n k
map_u f xs == CONST bop CONST map f 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, 'α) uexpr ⇒ ('a set, 'α) uexpr ⇒ (bool, 'α) uexpr (**infix** $\subseteq_u$ 50)

translations
$finite_u(x)$ == CONST uop (CONST finite) x
$\{\}_u$ == ≪{}≫
$\{x, xs\}_u$ == CONST bop (CONST insert) x $\{xs\}_u$
$\{x\}_u$ == CONST bop (CONST insert) x ≪{}≫
$A \cup_u B$ == CONST bop (op ∪) A B
$A \cap_u B$ == CONST bop (op ∩) A B
$x \in_u A$ == CONST bop (op ∈) x A
$A \subset_u B$ == CONST bop (op ⊂) A B
$f \subset_u g$ <= CONST bop (op $\subset_p$) f g

$f \subset_u g$ <= $CONST\ bop\ (op \subset_f)\ f\ g$
$A \subseteq_u B$ == $CONST\ bop\ (op \subseteq)\ A\ B$
$f \subseteq_u g$ <= $CONST\ bop\ (op \subseteq_p)\ f\ g$
$f \subseteq_u g$ <= $CONST\ bop\ (op \subseteq_f)\ f\ g$

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

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

syntax — Sum types
-uinl :: *logic* ⇒ *logic* (inl_u'(-'))
-uinr :: *logic* ⇒ *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 :: $('a, '\alpha)\ uexpr \Rightarrow ('a, '\alpha)\ uexpr \Rightarrow ('a\ set, '\alpha)\ uexpr$ ((*1*{-..-}$_u$))
-uset-atLeastLessThan :: $('a, '\alpha)\ uexpr \Rightarrow ('a, '\alpha)\ uexpr \Rightarrow ('a\ set, '\alpha)\ uexpr$ ((*1*{-..<-}$_u$))
-uset-compr :: $pttrn \Rightarrow ('a\ set, '\alpha)\ uexpr \Rightarrow (bool, '\alpha)\ uexpr \Rightarrow ('b, '\alpha)\ uexpr \Rightarrow ('b\ set, '\alpha)\ uexpr$ ((*1*{- :/ - |/ - •/ -}$_u$))
-uset-compr-nset :: $pttrn \Rightarrow (bool, '\alpha)\ uexpr \Rightarrow ('b, '\alpha)\ uexpr \Rightarrow ('b\ set, '\alpha)\ uexpr$ ((*1*{- |/ - •/ -}$_u$))

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

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* :: $'a::order\text{-}topology \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b::t2\text{-}space$ **where**
$ulim\text{-}left = (\lambda\ p\ f.\ Lim\ (at\text{-}left\ p)\ f)$

definition *ulim-right* :: $'a::order\text{-}topology \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b::t2\text{-}space$ **where**
$ulim\text{-}right = (\lambda\ p\ f.\ Lim\ (at\text{-}right\ p)\ f)$

definition *ucont-on* :: $('a::topological\text{-}space \Rightarrow 'b::topological\text{-}space) \Rightarrow 'a\ set \Rightarrow bool$ **where**
$ucont\text{-}on = (\lambda\ f\ A.\ continuous\text{-}on\ A\ f)$

syntax
  $\text{-}ulim\text{-}left$ :: $id \Rightarrow logic \Rightarrow logic \Rightarrow logic$ (lim_u'(- → -$^-$')'(-'))
  $\text{-}ulim\text{-}right$ :: $id \Rightarrow logic \Rightarrow logic \Rightarrow logic$ (lim_u'(- → -$^+$')'(-'))
  $\text{-}ucont\text{-}on$ :: $logic \Rightarrow logic \Rightarrow logic$ (**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*]: $tail_u(\langle x\rangle\ \hat{}_u\ xs) = xs$
by (*transfer*, *simp*)

lemma *ufun-apply-lit* [*simp*]:
$\ll f\gg(\ll x\gg)_a = \ll f(x)\gg$
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 $0 = []_u$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$op + = op$ @ 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 [10, 9] 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 [3] and Oliveira's [9] 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* :: $('a \Longrightarrow '\alpha) \Rightarrow ('b, '\alpha)$ *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, 'α) *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*]: $[\![$ *mwb-lens* x; $x \bowtie y$ $]\!] \Longrightarrow y \sharp$ *var* x
by (*transfer*, *auto*)

lemma *unrest-iuvar* [*unrest*]: $[\![$ *mwb-lens* x; $x \bowtie y$ $]\!] \Longrightarrow \$y \sharp \$x$
by (*simp add*: *unrest-var*)

lemma *unrest-ouvar* [*unrest*]: $[\![$ *mwb-lens* x; $x \bowtie y$ $]\!] \Longrightarrow \$y\acute{} \sharp \$x\acute{}$
by (*simp add*: *unrest-var*)

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

lemma *unrest-iuvar-ouvar* [*unrest*]:
fixes x :: ('$a \Longrightarrow$ 'α)
assumes *mwb-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 :: ('$a \Longrightarrow$ 'α)
assumes *mwb-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** † *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 $subst\text{-}upd :: ('\alpha,'\beta)\ psubst \Rightarrow 'v \Rightarrow ('a,\ '\alpha)\ uexpr \Rightarrow ('\alpha,'\beta)\ psubst$

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

adhoc-overloading
  subst-upd subst-upd-uvar

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

lift-definition $usubst\text{-}lookup :: ('\alpha,'\beta)\ psubst \Rightarrow ('a \Longrightarrow '\beta) \Rightarrow ('a,\ '\alpha)\ uexpr\ (\langle\text{-}\rangle_s)$
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 $unrest\text{-}usubst :: ('a \Longrightarrow '\alpha) \Rightarrow '\alpha\ usubst \Rightarrow bool$
where $unrest\text{-}usubst\ x\ \sigma = (\forall\ \varrho\ v.\ \sigma\ (put_x\ \varrho\ v) = put_x\ (\sigma\ \varrho)\ v)$

adhoc-overloading
unrest unrest-usubst

Parallel substitutions allow us to divide the state space into three segments using two lens, A and B. They correspond to the part of the state that should be updated by the respective substitution. The two lenses should be independent. If any part of the state is not covered by either lenses then this area is left unchanged (framed).

definition *par-subst* :: $'\alpha$ *usubst* $\Rightarrow$ $('a \Longrightarrow '\alpha) \Rightarrow ('b \Longrightarrow '\alpha) \Rightarrow '\alpha$ *usubst* $\Rightarrow '\alpha$ *usubst* **where**
par-subst σ_1 *A B* $\sigma_2 = (\lambda\ s.\ (s \oplus_L (\sigma_1\ s)\ on\ A) \oplus_L (\sigma_2\ s)\ on\ B)$

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 (-)
  -par-subst :: logic ⇒ salpha ⇒ salpha ⇒ logic ⇒ logic (- [-|-]s - [100,0,0,101] 101)
```

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
  -par-subst σ1 A B σ2 == CONST par-subst σ1 A B σ2
```

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 $\sigma\ x = \sigma(x \mapsto_s \&x)$

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*)

lemma *subst-upd-id-lam* [*usubst*]: *subst-upd* $(\lambda\ x.\ x)$ $x\ v$ = *subst-upd id* $x\ v$
  by (*simp add*: *id-def*)

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*)

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

lemma *subst-upd-pr-var* [*usubst*]: $s(\&x \mapsto_s v) = s(x \mapsto_s v)$
  by (*simp add*: *pr-var-def*)

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: $put_y\ (put_x\ b_1\ (\llbracket\&y\rrbracket_e\ b_1))\ (\llbracket\&x\rrbracket_e\ b_1) = put_y\ (put_x\ b_2\ (\llbracket\&y\rrbracket_e\ b_2))\ (\llbracket\&x\rrbracket_e\ b_2)$
    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-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-pr-var-id* [*usubst*]:
  vwb-lens $x \Longrightarrow [x \mapsto_s var\ (pr\text{-}var\ x)] = id$
  apply (*simp add*: *subst-upd-uvar-def pr-var-def*)
  apply (*transfer*)
  apply (*rule ext*)
  apply (*auto*)
done

lemma *usubst-upd-comm-dash* [*usubst*]:
  fixes $x :: ('a \Longrightarrow '\alpha)$
  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 *subst-upd-lens-plus* [*usubst*]:
  subst-upd $\sigma\ (x +_L y)\ \ll(u,v)\gg = \sigma(y \mapsto_s \ll v\gg,\ x \mapsto_s \ll u\gg)$
  by (*simp add*: *lens-defs uexpr-defs subst-upd-uvar-def*, *transfer*, *auto*)

lemma *subst-upd-in-lens-plus* [*usubst*]:
  subst-upd $\sigma\ (ivar\ (x +_L y))\ \ll(u,v)\gg = \sigma(\$y \mapsto_s \ll v\gg,\ \$x \mapsto_s \ll u\gg)$
  by (*simp add*: *lens-defs uexpr-defs subst-upd-uvar-def*, *transfer*, *auto simp add*: *prod.case-eq-if*)

lemma *subst-upd-out-lens-plus* [*usubst*]:
  subst-upd $\sigma\ (ovar\ (x +_L y))\ \ll(u,v)\gg = \sigma(\$y\acute{} \mapsto_s \ll v\gg,\ \$x\acute{} \mapsto_s \ll u\gg)$
  by (*simp add*: *lens-defs uexpr-defs subst-upd-uvar-def*, *transfer*, *auto simp add*: *prod.case-eq-if*)

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-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-lens $x \Longrightarrow id -_s x = id$
by (*simp add*: *subst-del-def subst-upd-uvar-def pr-var-def*, *transfer*, *auto*)

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

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

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-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 :: $('a \Longrightarrow '\alpha)$
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 :: $('a \Longrightarrow '\alpha)$
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* :: $('a \Longrightarrow '\alpha) \Rightarrow ('b \Longrightarrow '\alpha) \Rightarrow$ *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*)

5.7 Parallel Substitution Laws

lemma *par-subst-id* [*usubst*]:
$[\![\textit{vwb-lens } A; \textit{vwb-lens } B]\!] \Longrightarrow id\ [A|B]_s\ id = id$
by (*simp add*: *par-subst-def lens-override-idem id-def*)

lemma *par-subst-left-empty* [*usubst*]:
$[\![\textit{vwb-lens } A]\!] \Longrightarrow \sigma\ [\&\emptyset|A]_s\ \varrho = id\ [\&\emptyset|A]_s\ \varrho$
by (*simp add*: *par-subst-def pr-var-def*)

lemma *par-subst-right-empty* [*usubst*]:
$[\![\textit{vwb-lens } A]\!] \Longrightarrow \sigma\ [A|\&\emptyset]_s\ \varrho = \sigma\ [A|\&\emptyset]_s\ id$
by (*simp add*: *par-subst-def pr-var-def*)

lemma *par-subst-comm*:
$[\![A \bowtie B]\!] \Longrightarrow \sigma\ [A|B]_s\ \varrho = \varrho\ [B|A]_s\ \sigma$
by (*simp add*: *par-subst-def lens-override-def lens-indep-comm*)

lemma *par-subst-upd-left-in* [*usubst*]:
$[\![\textit{vwb-lens } A; A \bowtie B; x \subseteq_L A]\!] \Longrightarrow \sigma(x \mapsto_s v)\ [A|B]_s\ \varrho = (\sigma\ [A|B]_s\ \varrho)(x \mapsto_s v)$
by (*simp add*: *par-subst-def subst-upd-uvar-def lens-override-put-right-in*)
(*simp add*: *lens-indep-comm lens-override-def sublens-pres-indep*)

lemma *par-subst-upd-left-out* [*usubst*]:
$[\![\textit{vwb-lens } A; x \bowtie A]\!] \Longrightarrow \sigma(x \mapsto_s v)\ [A|B]_s\ \varrho = (\sigma\ [A|B]_s\ \varrho)$
by (*simp add*: *par-subst-def subst-upd-uvar-def lens-override-put-right-out*)

lemma *par-subst-upd-right-in* [*usubst*]:
$[\![\textit{vwb-lens } B; A \bowtie B; x \subseteq_L B]\!] \Longrightarrow \sigma\ [A|B]_s\ \varrho(x \mapsto_s v) = (\sigma\ [A|B]_s\ \varrho)(x \mapsto_s v)$
using *lens-indep-sym par-subst-comm par-subst-upd-left-in* **by** *fastforce*

lemma *par-subst-upd-right-out* [*usubst*]:
$[\![\textit{vwb-lens } B; A \bowtie B; x \bowtie B]\!] \Longrightarrow \sigma\ [A|B]_s\ \varrho(x \mapsto_s v) = (\sigma\ [A|B]_s\ \varrho)$
by (*simp add*: *par-subst-comm par-subst-upd-left-out*)

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

In this theory we begin to create an Isabelle version of the alphabetised predicate calculus that is described in Chapter 1 of the UTP book [5].

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*)
-uvar-res :: *logic* ⇒ *salpha* ⇒ *logic* (**infixl** $\upharpoonright_v$ *90*)

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*)
$\exists\ x \in A \cdot P$ => $\exists\ x \cdot \ll x \gg \in_u A \wedge P$
-ushAll x P == *CONST ushAll* (λ *x*. *P*)
$\forall\ x \in A \cdot P$ => $\forall\ x \cdot \ll x \gg \in_u A \Rightarrow P$
$\forall\ x \mid P \cdot Q$ => $\forall\ x \cdot P \Rightarrow Q$
$\forall\ x > y \cdot P$ => $\forall\ x \cdot \ll x \gg >_u y \Rightarrow P$
$\forall\ x < y \cdot P$ => $\forall\ x \cdot \ll x \gg <_u y \Rightarrow P$

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**
$P \sqsubseteq Q \equiv$ *less-eq Q P*

Since, on the whole, lattices in UTP are the opposite way up to the standard definitions in HOL, we syntactically invert the lattice operators. This is the one exception where we do steal HOL syntax, but I think it makes sense for UTP. 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* ($\bigsqcap$- [*900*] *900*)
notation *Inf* ($\bigsqcup$- [*900*] *900*)
purge-notation *Sup* ($\bigsqcup$- [*900*] *900*)
notation *Sup* ($\bigsqcap$- [*900*] *900*)

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

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

syntax
  -INF1 :: $pttrns \Rightarrow {'b} \Rightarrow {'b}$ (($3\bigsqcup$-./ -) [*0*, *10*] *10*)
  -INF :: $pttrn \Rightarrow {'a}\ set \Rightarrow {'b} \Rightarrow {'b}$ (($3\bigsqcup$-$\in$-./ -) [*0*, *0*, *10*] *10*)
  -SUP1 :: $pttrns \Rightarrow {'b} \Rightarrow {'b}$ (($3\bigsqcap$-./ -) [*0*, *10*] *10*)
  -SUP :: $pttrn \Rightarrow {'a}\ set \Rightarrow {'b} \Rightarrow {'b}$ (($3\bigsqcap$-$\in$-./ -) [*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* $\Rightarrow$ ($'a$, $'b$) *uexpr* $\Rightarrow$ ($'a$, $'b$) *uexpr*
  is $\lambda P\ Q\ A.$ *Lattices.sup* $(P\ A)\ (Q\ A)$ **.**
  lift-definition *inf-uexpr* :: ($'a$, $'b$) *uexpr* $\Rightarrow$ ($'a$, $'b$) *uexpr* $\Rightarrow$ ($'a$, $'b$) *uexpr*
  is $\lambda P\ Q\ A.$ *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

lemma *top-uexpr-rep-eq* [*simp*]:
  $[\![Orderings.bot]\!]_e\ b = False$
  by (*transfer*, *auto*)

lemma *bot-uexpr-rep-eq* [*simp*]:
  $[\![Orderings.top]\!]_e$ *b* = *True*
  by (*transfer*, *auto*)

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* $= (Lattices.inf :: 'α\ upred \Rightarrow 'α\ upred \Rightarrow 'α\ upred)$
definition *disj-upred* $= (Lattices.sup :: 'α\ upred \Rightarrow 'α\ upred \Rightarrow 'α\ upred)$
definition *not-upred* $= (uminus :: 'α\ upred \Rightarrow 'α\ upred)$
definition *diff-upred* $= (minus :: 'α\ upred \Rightarrow 'α\ upred \Rightarrow 'α\ upred)$

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

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

notation
  conj-upred (**infixr** $\wedge_p$ *35*) **and**
  disj-upred (**infixr** $\vee_p$ *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* $:: ('a \Rightarrow 'α\ upred) \Rightarrow ('a \Rightarrow ('b::complete\text{-}lattice, 'α)\ uexpr) \Rightarrow ('b, 'α)\ uexpr$
is $\lambda\ P\ F\ b.\ Sup\ \{[\![F\ x]\!]_e b \mid x.\ [\![P\ x]\!]_e b\}$ **.**

lift-definition *USUP* $:: ('a \Rightarrow 'α\ upred) \Rightarrow ('a \Rightarrow ('b::complete\text{-}lattice, 'α)\ uexpr) \Rightarrow ('b, 'α)\ uexpr$
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 $:: pttrn \Rightarrow logic \Rightarrow logic$ $(\bigwedge\ \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$
  -USup $:: pttrn \Rightarrow logic \Rightarrow logic$ $(\bigsqcup\ \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$
  -USup-mem $:: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic$ $(\bigwedge\ \text{-} \in \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$
  -USup-mem $:: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic$ $(\bigsqcup\ \text{-} \in \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$
  -USUP $:: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic$ $(\bigwedge\ \text{-} \mid \text{-}\ \cdot\ \text{-}\ [0, 0, 10]\ 10)$
  -USUP $:: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic$ $(\bigsqcup\ \text{-} \mid \text{-}\ \cdot\ \text{-}\ [0, 0, 10]\ 10)$
  -UInf $:: pttrn \Rightarrow logic \Rightarrow logic$ $(\bigvee\ \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$
  -UInf $:: pttrn \Rightarrow logic \Rightarrow logic$ $(\bigsqcap\ \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$
  -UInf-mem $:: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic$ $(\bigvee\ \text{-} \in \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$
  -UInf-mem $:: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic$ $(\bigsqcap\ \text{-} \in \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$
  -UINF $:: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic$ $(\bigvee\ \text{-} \mid \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$
  -UINF $:: pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic$ $(\bigsqcap\ \text{-} \mid \text{-}\ \cdot\ \text{-}\ [0, 10]\ 10)$

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* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* **is**
λ P Q A. P A $\longrightarrow$ Q A **.**

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

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

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

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

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

We define the following operator which is dual of existential quantification. It hides the valuation of variables other than x through existential quantification.

lift-definition *var-res* :: $'\alpha$ *upred* $\Rightarrow$ $('a \Longrightarrow '\alpha)$ $\Rightarrow$ $'\alpha$ *upred* **is**
λ P x b. $\exists$ b'. P $(b' \oplus_L\ b\ on\ x)$ **.**

translations
-uvar-res P a $\rightleftharpoons$ *CONST var-res P 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**
λ P A. $\forall$ A'. P A' **.**

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

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

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

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

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

syntax

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

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

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

We set up syntax for the conditional. This is effectively an infix version of if-then-else where the condition is in the middle.

abbreviation *cond* ::
(*'a*,*'α*) *uexpr* ⇒ *'α upred* ⇒ (*'a*,*'α*) *uexpr* ⇒ (*'a*,*'α*) *uexpr*
((*3*- ◃ - ▹/ -) [*52*,*0*,*53*] *52*)
where $P \triangleleft b \triangleright Q$ ≡ *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*]:
$[\![(\bigwedge i.\ x \sharp P(i)); (\bigwedge i.\ x \sharp Q(i))]\!] \Longrightarrow x \sharp (\bigsqcup\ i \mid P(i) \cdot Q(i))$
by (*pred-auto*)

lemma *unrest-UINF-mem* [*unrest*]:
$[\![(\bigwedge i.\ i \in A \Longrightarrow x \sharp P(i))]\!] \Longrightarrow x \sharp (\bigsqcap\ i{\in}A \cdot P(i))$
by (*pred-simp*, *metis*)

lemma *unrest-USUP-mem* [*unrest*]:
$[\![(\bigwedge i.\ i \in A \Longrightarrow x \sharp P(i))]\!] \Longrightarrow x \sharp (\bigsqcup\ i{\in}A \cdot P(i))$
by (*pred-simp*, *metis*)

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

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

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

The sublens proviso can be thought of as membership below.

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

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

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

lemma *unrest-all-in* [*unrest*]:
$[\![mwb\text{-}lens\ y; x \subseteq_L y]\!] \Longrightarrow x \sharp (\forall\ y \cdot P)$
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-var-res-diff* [*unrest*]:
assumes $x \bowtie y$
shows $y \sharp (P \upharpoonright_v x)$

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

lemma *unrest-var-res-in* [*unrest*]:
assumes *mwb-lens* x $y \subseteq_L x$ $y \sharp P$
shows $y \sharp (P \upharpoonright_v x)$
using *assms*
apply (*pred-auto*)
apply *fastforce*
apply (*metis* (*no-types*, *lifting*) *mwb-lens-weak weak-lens.put-get*)
done

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

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

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

7.4 Substitution Laws

Substitution is monotone

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

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

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

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

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

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

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

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

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

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

lemma *subst-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-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-same'* [*usubst*]:
mwb-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-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 *spec-refine*:
$Q \sqsubseteq (P \wedge R) \Longrightarrow (P \Rightarrow Q) \sqsubseteq R$
by (*rel-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*]: $(P \vee \neg\ P) = true$
by (*pred-auto*)

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

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

by (*pred-auto*)

lemma *p-imp-false* [*simp*]: $(P \Rightarrow \mathit{false}) = (\neg\ P)$
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 $P\ Q :: '\alpha\ \mathit{upred}$
shows $(P \sqcap Q) = (P \vee Q)$
by (*pred-auto*)

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

lemma *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*:
  $\llbracket \bigwedge x.\ P_1(x) = P_2(x); \bigwedge x.\ `P_1(x) \Rightarrow Q_1(x) =_u Q_2(x)` \rrbracket \Longrightarrow$
    $(\bigsqcap\ x \mid P_1(x) \cdot Q_1(x)) = (\bigsqcap\ x \mid P_2(x) \cdot Q_2(x))$
 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*]: $(\bigsqcap\ i\in insert\ x\ xs \cdot P(i)) = (P(x) \sqcap (\bigsqcap\ i\in xs \cdot P(i)))$
apply (*pred-simp*)
apply (*subst Sup-insert*[*THEN sym*])
apply (*rule-tac cong*[*of Sup Sup*])

apply (*auto*)
done

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

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

lemma *conj-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*)

lemma *UINF-refines'*:
assumes $\bigwedge i.\ P \sqsubseteq Q(i)$
shows $P \sqsubseteq (\bigsqcap\ i \cdot Q(i))$
using *assms*
apply (*rel-auto*) **using** *Sup-le-iff* **by** *fastforce*

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*)+)

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 $x :: ('a \Longrightarrow '\alpha)$
  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* $x \sharp v$
  shows $(\exists\ x \cdot P \wedge var\ x =_u v) = P[\![v/x]\!]$
  using *assms*
  by (*pred-auto*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.7 Variable Restriction

lemma *var-res-all*:
  $P \upharpoonright_v \&\Sigma = P$
  by (*rel-auto*)

lemma *var-res-twice*:
  mwb-lens $x \Longrightarrow P \upharpoonright_v x \upharpoonright_v x = P \upharpoonright_v x$
  by (*pred-auto*)

8.8 Conditional laws

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

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

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

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

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

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

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

lemma *cond-conj-not*: $((P \triangleleft b \triangleright Q) \wedge (\neg b)) = (Q \wedge (\neg b))$
  by (*rel-auto*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *conj-conds*:
$(P1 \triangleleft b \triangleright Q1 \wedge P2 \triangleleft b \triangleright Q2) = (P1 \wedge P2) \triangleleft b \triangleright (Q1 \wedge Q2)$
by *pred-auto*

lemma *disj-conds*:

$(P1 \triangleleft b \triangleright Q1 \vee P2 \triangleleft b \triangleright Q2) = (P1 \vee P2) \triangleleft b \triangleright (Q1 \vee Q2)$
by *pred-auto*

8.9 Additional Expression Laws

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

lemma *uzero-le-laws* [*simp*]:
$(0 :: ('a::\{linordered\text{-}semidom\}, '\alpha)\ uexpr) \leq_u numeral\ x = true$
$(1 :: ('a::\{linordered\text{-}semidom\}, '\alpha)\ uexpr) \leq_u numeral\ x = true$
$(0 :: ('a::\{linordered\text{-}semidom\}, '\alpha)\ uexpr) \leq_u 1 = true$
by (*pred-simp*)+

lemma *unumeral-le-1* [*simp*]:
assumes $(numeral\ i :: 'a::\{numeral,ord\}) \leq numeral\ j$
shows $(numeral\ i :: ('a, '\alpha)\ uexpr) \leq_u numeral\ j = true$
using *assms* **by** (*pred-auto*)

lemma *unumeral-le-2* [*simp*]:
assumes $(numeral\ i :: 'a::\{numeral,linorder\}) > numeral\ j$
shows $(numeral\ i :: ('a, '\alpha)\ uexpr) \leq_u numeral\ j = false$
using *assms* **by** (*pred-auto*)

lemma *uset-laws* [*simp*]:
$x \in_u \{\}_u = false$
$x \in_u \{m..n\}_u = (m \leq_u x \wedge x \leq_u n)$
by (*pred-auto*)+

lemma *pfun-entries-apply* [*simp*]:
$(entr_u(d,f) :: (('k, 'v)\ pfun, '\alpha)\ uexpr)(i)_a = ((\ll f \gg(i)_a) \triangleleft i \in_u d \triangleright \bot_u)$
by (*pred-auto*)

lemma *udom-uupdate-pfun* [*simp*]:
fixes $m :: (('k, 'v)\ pfun, '\alpha)\ uexpr$
shows $dom_u(m(k \mapsto v)_u) = \{k\}_u \cup_u dom_u(m)$
by (*rel-auto*)

lemma *uapply-uupdate-pfun* [*simp*]:
fixes $m :: (('k, 'v)\ pfun, '\alpha)\ uexpr$
shows $(m(k \mapsto v)_u)(i)_a = v \triangleleft i =_u k \triangleright m(i)_a$
by (*rel-auto*)

lemma *ulit-eq* [*simp*]: $x = y \Longrightarrow (\ll x \gg =_u \ll y \gg) = true$
by (*rel-auto*)

lemma *ulit-neq* [*simp*]: $x \neq y \Longrightarrow (\ll x \gg =_u \ll y \gg) = false$
by (*rel-auto*)

lemma *uset-mems* [*simp*]:
$x \in_u \{y\}_u = (x =_u y)$
$x \in_u A \cup_u B = (x \in_u A \vee x \in_u B)$
$x \in_u A \cap_u B = (x \in_u A \wedge x \in_u B)$
by (*rel-auto*)+

8.10 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

8.11 Cylindric Algebra

lemma *C1*: $(\exists\ x \cdot \mathit{false}) = \mathit{false}$
  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 $x :: ('a \Longrightarrow '\alpha)$
  shows $(\&x =_u \&x) = \mathit{true}$
  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 $((\exists\ x \cdot \&x =_u \&y \wedge P) \wedge (\exists\ x \cdot \&x =_u \&y \wedge \neg\ P)) = \mathit{false}$
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*: $(\mu\ X \cdot X) = \mathit{true}$
by (*simp add*: *antisym gfp-upperbound*)

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

lemma *nu-id*: $(\nu\ X \cdot X) = \mathit{false}$
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 [5].

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

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

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

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

lemma *chainE*:
  assumes *chain* $Y \bigwedge i. \llbracket\ Y\ 0 = false;\ Y\ (Suc\ i) \sqsubseteq Y\ i\ \rrbracket \Longrightarrow P$
  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* ::
  $('a\ upred \Rightarrow 'a\ upred) \Rightarrow 'a\ chain \Rightarrow bool$ **where**
$constr\ F\ E \longleftrightarrow chain\ E \wedge (\forall\ X\ n.\ ((F(X) \wedge E(n + 1)) = (F(X \wedge E(n)) \wedge E\ (n + 1))))$

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 $(\bigsqcap\ (range\ E) \wedge \mu\ F) = (\bigsqcap\ (range\ E) \wedge \nu\ F)$
proof −
  from *assms* **have** $\forall\ n.\ (E\ n \wedge \mu\ F) = (E\ n \wedge \nu\ F)$
  proof (*rule-tac allI*)
    fix n
    from *assms* **show** $(E\ n \wedge \mu\ F) = (E\ n \wedge \nu\ F)$
    proof (*induct n*)
      case *0* **thus** *?case* **by** (*simp add*: *constr-def*)
    next
      case (*Suc n*)
      note *hyp* = *this*
      thus *?case*
      proof −
        have $(E\ (n + 1) \wedge \mu\ F) = (E\ (n + 1) \wedge F\ (\mu\ F))$
          using *gfp-unfold*[*OF hyp*(*3*), *THEN sym*] **by** (*simp add*: *constr-def*)
        also from *hyp* **have** ... $= (E\ (n + 1) \wedge F\ (E\ n \wedge \mu\ F))$
          by (*metis conj-comm constr-def*)
        also from *hyp* **have** ... $= (E\ (n + 1) \wedge F\ (E\ n \wedge \nu\ F))$
          by *simp*
        also from *hyp* **have** ... $= (E\ (n + 1) \wedge \nu\ F)$
          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* $\bigsqcap\ (range\ E) = C$
  shows $(C \wedge \mu\ F) = (C \wedge \nu\ F)$
  using *assms*(*1*) *assms*(*2*) *assms*(*3*) *chain-pred-terminates* **by** *blast*

end

10 UTP Events

theory *utp-event*
imports *utp-pred*
begin

10.1 Events

Events of some type $'\vartheta$ are just the elements of that type.

type-synonym $'\vartheta$ *event* $= {}'\vartheta$

10.2 Channels

Typed channels are modelled as functions. Below, $'a$ determines the channel type and $'\vartheta$ the underlying event type. As with values, it is difficult to introduce channels as monomorphic types due to the fact that they can have arbitrary parametrisations in term of $'a$. Applying a channel to an element of its type yields an event, as we may expect. Though this is not formalised here, we may also sensibly assume that all channel- representing functions are injective. Note: is there benefit in formalising this here?

type-synonym $('a, {}'\vartheta)$ *chan* $= {}'a \Rightarrow {}'\vartheta$ *event*

A downside of the approach is that the event type $'\vartheta$ must be able to encode *all* events of a process model, and hence cannot be fixed upfront for a single channel or channel set. To do so, we actually require a notion of 'extensible' datatypes, in analogy to extensible record types. Another solution is to encode a notion of channel scoping that namely uses *sum* types to lift channel types into extensible ones, that is using channel-set specific scoping operators. This is a current work in progress.

10.2.1 Operators

The Z type of a channel corresponds to the entire carrier of the underlying HOL type of that channel. Strictly, the function is redundant but was added to mirror the mathematical account in [?]. (TODO: Ask Simon Foster for [?])

definition *chan-type* :: $('a, {}'\vartheta)$ *chan* $\Rightarrow {}'a$ *set* (δ_u) **where**
[*upred-defs*]: $\delta_u\ c = UNIV$

The next lifted function creates an expression that yields a channel event, from an expression on the channel type $'a$.

definition *chan-apply* ::
  $('a, {}'\vartheta)\ chan \Rightarrow ('a, {}'\alpha)\ uexpr \Rightarrow ('\vartheta\ event, {}'\alpha)\ uexpr\ ('(\text{-}\cdot\text{/-}')_u)$ **where**
[*upred-defs*]: $(c \cdot e)_u = \ll c \gg (e)_a$

lemma *unrest-chan-apply* [*unrest*]: $x \sharp e \Longrightarrow x \sharp (c \cdot e)_u$
  by (*rel-auto*)

lemma *usubst-chan-apply* [*usubst*]: $\sigma \dagger (c \cdot v)_u = (c \cdot \sigma \dagger v)_u$
  by (*rel-auto*)

end

11 Alphabet Manipulation

theory *utp-alphabet*
  imports
    utp-pred utp-event
begin

11.1 Preliminaries

Alphabets are simply types that characterise the state-space of an expression. Thus the Isabelle type system ensures that predicates cannot refer to variables not in the alphabet as this would be a type error. Often one would like to add or remove additional variables, for example if we wish to have a predicate which ranges only a smaller state-space, and then lift it into a predicate over a larger one. This is useful, for example, when dealing with relations which refer only to undashed variables (conditions) since we can use the type system to ensure well-formedness.

In this theory we will set up operators for extending and contracting and alphabet. We first set up a theorem attribute for alphabet laws and a tactic.

named-theorems *alpha*

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

11.2 Alphabet Extrusion

Alter an alphabet by application of a lens that demonstrates how the smaller alphabet (β) injects into the larger alphabet (α). This changes the type of the expression so it is parametrised over the large alphabet. We do this by using the lens *get* function to extract the smaller state binding, and then apply this to the expression.

We call this "extrusion" rather than "extension" because if the extension lens is bijective then it does not extend the alphabet. Nevertheless, it does have an effect because the type will be different which can be useful when converting predicates with equivalent alphabets.

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

update-uexpr-rep-eq-thms

Next we prove some of the key laws. Extending an alphabet twice is equivalent to extending by the composition of the two lenses.

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

The bijective 1_L lens identifies the source and view types. Thus an alphabet extension using this has no effect.

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

Literals do not depend on any variables, and thus applying an alphabet extension only alters the predicate's type, and not its valuation .

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-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-shAll* [*alpha*]: $(\forall\ x \cdot P(x)) \oplus_p a = (\forall\ x \cdot P(x) \oplus_p a)$
by (*pred-auto*)

lemma *aext-event* [*alpha*]: $(c{\cdot}v)_u \oplus_p a = (c{\cdot}v \oplus_p a)_u$
by (*pred-auto*)

Alphabet extension distributes through the function liftings.

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

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

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

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

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

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

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

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

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

Extending a variable expression over x is equivalent to composing x with the alphabet, thus effectively yielding a variable whose source is the large alphabet.

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*)

Alphabet extension is monotonic and continuous.

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

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

If a variable is unrestricted in a predicate, then the extended variable is unrestricted in the predicate with an alphabet extension.

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

If a given variable (or alphabet) b is independent of the extension lens a, that is, it is outside the original state-space of p, then it follows that once p is extended by a then b cannot be restricted.

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

11.3 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 loss-less.

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*)

11.4 Alphabet Lens Laws

lemma *alpha-in-var* [*alpha*]: $x ;_L fst_L =$ *in-var x*

by (*simp add*: *in-var-def*)

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

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

lemma *out-var-prod-lens* [*alpha*]:
$\mathit{wb\text{-}lens}\ X \Longrightarrow \mathit{out\text{-}var}\ x ;_L (X \times_L Y) = \mathit{out\text{-}var}\ (x ;_L Y)$
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

11.5 Substitution Alphabet Extension

This allows us to extend the alphabet of a substitution, in a similar way to expressions.

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

lemma *id-subst-ext* [*usubst*]:
$\mathit{wb\text{-}lens}\ x \Longrightarrow \mathit{id} \oplus_s x = \mathit{id}$
by *pred-auto*

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

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

11.6 Substitution Alphabet Restriction

This allows us to reduce the alphabet of a substitution, in a similar way to expressions.

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

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

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

lemma *subst-ext-res* [*usubst*]:
  $mwb\text{-}lens\ x \Longrightarrow (\sigma \oplus_s x) \upharpoonright_s x = \sigma$
  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

12 Lifting expressions

theory *utp-lift*
  imports
    utp-alphabet
begin

12.1 Lifting definitions

We define operators for converting an expression to and from a relational state space with the help of alphabet extrusion and restriction. In general throughout Isabelle/UTP we adopt the notation $\lceil P \rceil$ with some subscript to denote lifting an expression into a larger alphabet, and $\lfloor P \rfloor$ for dropping into a smaller alphabet.

The following two functions lift and drop an expression, respectively, whose alphabet is $'\alpha$, into a product alphabet $'\alpha \times {'\beta}$. This allows us to deal with expressions which refer only to undashed variables, and use the type-system to ensure this.

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

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

The following two functions lift and drop an expression, respectively, whose alphabet is $'\beta$, into a product alphabet $'\alpha \times {'\beta}$. This allows us to deal with expressions which refer only to dashed variables.

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

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

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

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

12.2 Lifting Laws

With the help of our alphabet laws, we can prove some intuitive laws about alphabet lifting. For example, lifting variables yields an unprimed or primed relational variable expression, respectively.

lemma *lift-pre-var* [*simp*]:

$\lceil var\ x \rceil_< = \$x$
by (*alpha-tac*)

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

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

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

12.3 Substitution Laws

lemma *pre-var-subst* [*usubst*]:
$\sigma(\$x \mapsto_s \ll v \gg) \dagger \lceil P \rceil_< = \sigma \dagger \lceil P[\![\ll v \gg/\&x]\!] \rceil_<$
by (*pred-simp*)

12.4 Unrestriction laws

Crucially, the lifting operators allow us to demonstrate unrestriction properties. For example, we can show that no primed variable is restricted in an expression over only the first element of the state-space product type.

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

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

13 Alphabetised Relations

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

An alphabetised relation is simply a predicate whose state-space is a product type. In this theory we construct the core operators of the relational calculus, and prove a libary of associated theorems, based on Chapters 2 and 5 of the UTP book [5].

13.1 Relational Alphabets

We set up convenient syntax to refer to the input and output parts of the alphabet, as is common in UTP. Since we are in a product space, these are simply the lenses fst_L and snd_L.

definition $in\alpha$ :: $('\alpha \Longrightarrow '\alpha \times '\beta)$ **where**
[*lens-defs*]: $in\alpha = fst_L$

definition $out\alpha$ :: $('\beta \Longrightarrow '\alpha \times '\beta)$ **where**
[*lens-defs*]: $out\alpha = snd_L$

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

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

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

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

lemma *drop-pre-inv* [*simp*]: $[\![\ out\alpha \sharp p\ ]\!] \Longrightarrow \lceil\lfloor p\rfloor_<\rceil_< = p$
by (*pred-simp*)

lemma *usubst-lookup-ivar-unrest* [*usubst*]:
$in\alpha \sharp \sigma \Longrightarrow \langle\sigma\rangle_s\ (ivar\ x) = \$x$
by (*rel-simp*, *metis fstI*)

lemma *usubst-lookup-ovar-unrest* [*usubst*]:
$out\alpha \sharp \sigma \Longrightarrow \langle\sigma\rangle_s\ (ovar\ x) = \$x\acute{}$
by (*rel-simp*, *metis sndI*)

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

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

The following two functions lift a predicate substitution to a relational one.

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

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

The alphabet of a relation then consists wholly of the input and output portions.

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

13.2 Relational Types and Operators

We create type synonyms for conditions (which are simply predicates) – i.e. relations without dashed variables –, alphabetised relations where the input and output alphabet can be different, and finally homogeneous relations.

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

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

We set up some overloaded constants for sequential composition and the identity in case we want to overload their definitions later.

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

We define a specialised version of the conditional where the condition can refer only to undashed variables, as is usually the case in programs, but not universally in UTP models. We implement this by lifting the condition predicate into the relational state-space with construction $\lceil b \rceil_<$.

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

Sequential composition is heterogeneous, and simply requires that the output alphabet of the first matches then input alphabet of the second. We define it by lifting HOL's built-in relational composition operator (*op* O). Since this returns a set, the definition states that the state binding b is an element of this set.

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

adhoc-overloading
useq seqr

We also set up a homogeneous sequential composition operator, and versions of *true* and *false* that are explicitly typed by a homogeneous alphabet.

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

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

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

We define the relational converse operator as an alphabet extrusion on the bijective lens $swap_L$ that swaps the elements of the product state-space.

abbreviation *conv-r* :: $('a, '\alpha \times '\beta)$ *uexpr* $\Rightarrow ('a, '\beta \times '\alpha)$ *uexpr* $(\text{-}^{-}\ [999]\ 999)$
where *conv-r* $e \equiv e \oplus_p swap_L$

Assignment is defined using substitutions, where latter defines what each variable should map to. The definition of the operator identifies the after state binding, b', with the substitution function applied to the before state binding b.

lift-definition *assigns-r* :: $'\alpha$ *usubst* $\Rightarrow$ $'\alpha$ *hrel* $(\langle\text{-}\rangle_a)$
 is $\lambda\ \sigma\ (b,\ b').\ b' = \sigma(b)$ **.**

Relational identity, or skip, is then simply an assignment with the identity substitution: it simply identifies all variables.

definition *skip-r* :: $'\alpha$ *hrel* **where**
[*urel-defs*]: *skip-r* = *assigns-r id*

adhoc-overloading
 uskip skip-r

We set up iterated sequential composition which iterates an indexed predicate over the elements of a list.

definition *seqr-iter* :: $'a$ *list* $\Rightarrow$ $('a \Rightarrow 'b\ hrel) \Rightarrow 'b$ *hrel* **where**
[*urel-defs*]: *seqr-iter xs P* = *foldr* $(\lambda\ i\ Q.\ P(i)\ ;;\ Q)$ *xs II*

A singleton assignment simply applies a singleton substitution function, and similarly for a double assignment.

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

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

We also define the alphabetised skip operator that identifies all input and output variables in the given alphabet lens. All other variables are unrestricted. We also set up syntax for it.

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

Similarly, we define the alphabetised assignment operator.

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

Assumptions ($c^\top$) and assertions ($c_\bot$) are encoded as conditionals. An assumption behaves like skip if the condition is true, and otherwise behaves like *false* (miracle). An assertion is the same, but yields *true*, which is an abort.

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

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

A test is like a precondition, except that it identifies to the postcondition, and is thus a refinement of *II*. It forms the basis for Kleene Algebra with Tests [7, 1] (KAT), which embeds a Boolean algebra into a Kleene algebra to represent conditions.

definition *lift-test* :: $'\alpha$ *cond* $\Rightarrow$ $'\alpha$ *hrel* $(\lceil\text{-}\rceil_t)$
where [*urel-defs*]: $\lceil b\rceil_t = (\lceil b\rceil_< \wedge II)$

We define two variants of while loops based on strongest and weakest fixed points. The former is *false* for an infinite loop, and the latter is *true*.

definition *while* :: $'\alpha$ *cond* $\Rightarrow$ $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* (*while*$^\top$ - *do* - *od*) **where**
[*urel-defs*]: *while*$^\top$ *b do P od* = $(\nu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)$

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

definition *while-bot* :: $'\alpha$ *cond* $\Rightarrow$ $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* (*while*$_\bot$ - *do* - *od*) **where**
[*urel-defs*]: *while*$_\bot$ *b do P od* = $(\mu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)$

While loops with invariant decoration (cf. [1]).

definition *while-inv* :: $'\alpha$ *cond* $\Rightarrow$ $'\alpha$ *cond* $\Rightarrow$ $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* (*while* - *invr* - *do* - *od*) **where**
[*urel-defs*]: *while b invr p do S od* = *while b do S od*

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** $\restriction_\alpha$ *80*) **where**
[*urel-defs*]: $P \restriction_\alpha x = (\exists\ \$x \cdot \exists\ \$x' \cdot P)$

We next describe frames and antiframes with the help of lenses. A frame states that P defines the behaviour of all variables not in a, and all those in a remain the same. An antiframe describes the converse: all variables in a are specified by P, and all other variables remain the same. For more information please see [8].

definition *frame* :: $('a \Longrightarrow '\alpha)$ $\Rightarrow$ $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* **where**
[*urel-defs*]: *frame a P* = $(\exists\ st \cdot P[\![\ll st\gg/\$\Sigma']\!] \wedge \$\Sigma' =_u \ll st\gg \oplus \$\Sigma\ on\ \&a)$

definition *antiframe* :: $('a \Longrightarrow '\alpha)$ $\Rightarrow$ $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* **where**
[*urel-defs*]: *antiframe a P* = $(\exists\ st \cdot P[\![\ll st\gg/\$\Sigma']\!] \wedge \$\Sigma' =_u \$\Sigma \oplus \ll st\gg\ on\ \&a)$

The nameset operator can be used to hide a portion of the after-state that lies outside the lens a. It can be useful to partition a relation's variables in order to conjoin it with another relation.

definition *nameset* :: $('a \Longrightarrow '\alpha)$ $\Rightarrow$ $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* **where**
[*urel-defs*]: *nameset a P* = $(P \restriction_v \{\$\Sigma,\$a'\})$

13.3 Syntax Translations

syntax
— Alternative traditional conditional syntax
-utp-if :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ((*if*$_u$ (-)/ *then* (-)/ *else* (-)) [*0*, *0*, *71*] *71*)
— Iterated sequential composition
-seqr-iter :: *pttrn* $\Rightarrow$ *'a list* $\Rightarrow$ $'\sigma$ *hrel* $\Rightarrow$ $'\sigma$ *hrel* ((*3*;; - : - $\cdot$/ -) [*0*, *0*, *10*] *10*)
— Single and multiple assignement
-assignment :: *svids* $\Rightarrow$ *uexprs* $\Rightarrow$ $'\alpha$ *hrel* ('(-') := '(-'))
-assignment :: *svids* $\Rightarrow$ *uexprs* $\Rightarrow$ $'\alpha$ *hrel* (**infixr** := *72*)
— Indexed assignment
-assignment-upd :: *svid* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infixr** [-] := *72*)
— Substitution constructor
-mk-usubst :: *svids* $\Rightarrow$ *uexprs* $\Rightarrow$ $'\alpha$ *usubst*
— Alphabetised skip
-skip-ra :: *salpha* $\Rightarrow$ *logic* (II_-)
— Frame
-frame :: *salpha* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (-:$[\![$-$]\!]$ [*64*,*0*] *80*)

— Antiframe
-antiframe :: *salpha* ⇒ *logic* ⇒ *logic* (-:[-] [*64*,*0*] *80*)
— Nameset
-nameset :: *salpha* ⇒ *logic* ⇒ *logic* (*ns* - · - [*0*,*999*] *999*)

translations
$\text{-}utp\text{-}if\ b\ P\ Q => P \lhd b \rhd_r Q$
$;;\ x : l \cdot P \rightleftharpoons (CONST\ seqr\text{-}iter)\ l\ (\lambda x.\ P)$
$\text{-}mk\text{-}usubst\ \sigma\ (\text{-}svid\text{-}unit\ x)\ v \rightleftharpoons \sigma(\&x \mapsto_s v)$
$\text{-}mk\text{-}usubst\ \sigma\ (\text{-}svid\text{-}list\ x\ xs)\ (\text{-}uexprs\ v\ vs) \rightleftharpoons (\text{-}mk\text{-}usubst\ (\sigma(\&x \mapsto_s v))\ xs\ vs)$
$\text{-}assignment\ xs\ vs => CONST\ assigns\text{-}r\ (\text{-}mk\text{-}usubst\ (CONST\ id)\ xs\ vs)$
$x := v <= CONST\ assigns\text{-}r\ (CONST\ subst\text{-}upd\ (CONST\ id)\ (CONST\ svar\ x)\ v)$
$x := v <= CONST\ assigns\text{-}r\ (CONST\ subst\text{-}upd\ (CONST\ id)\ x\ v)$
$x,y := u,v <= CONST\ assigns\text{-}r\ (CONST\ subst\text{-}upd\ (CONST\ subst\text{-}upd\ (CONST\ id)\ (CONST\ svar\ x)\ u)\ (CONST\ svar\ y)\ v)$
— Indexed assignment uses the overloaded collection update function *uupd*.
$x\ [k] := v => x := \&x(k \mapsto v)_u$
$\text{-}skip\text{-}ra\ v \rightleftharpoons CONST\ skip\text{-}ra\ v$
$\text{-}frame\ x\ P \rightleftharpoons CONST\ frame\ x\ P$
$\text{-}antiframe\ x\ P \rightleftharpoons CONST\ antiframe\ x\ P$
$\text{-}nameset\ x\ P \rightleftharpoons CONST\ nameset\ x\ P$

The following code sets up pretty-printing for homogeneous relational expressions. We cannot do this via the “translations” command as we only want the rule to apply when the input and output alphabet types are the same. The code has to deconstruct a $('a,\ '\alpha)$ *uexpr* type, determine that it is relational (product alphabet), and then checks if the types *alpha* and *beta* are the same. If they are, the type is printed as a *hexpr*. Otherwise, we have no match. We then set up a regular translation for the *hrel* type that uses this.

print-translation ⟪
```
let
fun tr' ctx [ a
        , Const (@{type-syntax prod},-) $ alpha $ beta ] =
  if (alpha = beta)
    then Syntax.const @{type-syntax hexpr} $ a $ alpha
    else raise Match;
in [(@{type-syntax uexpr},tr')]
end
```
⟫

translations
$(type)\ '\alpha\ hrel <= (type)\ (bool,\ '\alpha)\ hexpr$

13.4 Relation Properties

We describe some properties of relations, including functional and injective relations.

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

definition *uinj* :: $('a,\ 'b)\ rel \Rightarrow bool$
where [*urel-defs*]: $uinj\ R \longleftrightarrow II \sqsubseteq R ;;\ R^-$

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

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

13.5 Unrestriction Laws

lemma *unrest-iuvar* [*unrest*]: $out\alpha \sharp \$x$
 by (*metis fst-snd-lens-indep lift-pre-var outα-def unrest-aext-indep*)

lemma *unrest-ouvar* [*unrest*]: $in\alpha \sharp \$x'$
 by (*metis inα-def lift-post-var snd-fst-lens-indep unrest-aext-indep*)

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 $\$x' \sharp Q$
 shows $\$x' \sharp P \ ;; \ Q$
 using *assms* **by** (*rel-auto*)

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

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

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

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

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

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

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

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

lemma *unrest-convr-inα* [*unrest*]:
 $out\alpha \sharp p \Longrightarrow in\alpha \sharp p^-$
 by (*transfer*, *auto simp add*: *lens-defs*)

lemma *unrest-in-rel-var-res* [*unrest*]:
 $\mathit{vwb\text{-}lens}\ x \Longrightarrow \$x \sharp (P \upharpoonright_\alpha x)$

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*)

13.6 Substitution laws

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

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

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

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

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

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

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

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

lemma *subst-pre-skip* [*usubst*]: $\lceil\sigma\rceil_s \dagger II = \langle\sigma\rangle_a$
by (*rel-auto*)

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

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

lemma *subst-lift-upd* [*alpha*]:
  fixes $x :: ('a \Longrightarrow '\alpha)$
  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*: *pr-var-def fst-lens-def inα-def in-var-def*)

lemma *subst-drop-upd* [*alpha*]:
  fixes $x :: ('a \Longrightarrow '\alpha)$
  shows $\lfloor\sigma(\$x \mapsto_s v)\rfloor_s = \lfloor\sigma\rfloor_s(x \mapsto_s \lfloor v\rfloor_<)$
  by *pred-simp*

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

lemma *unrest-usubst-lift-in* [*unrest*]:
  $x \sharp P \Longrightarrow \$x \sharp \lceil P\rceil_s$
  by *pred-simp*

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

13.7 Alphabet laws

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

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

13.8 Relational unrestriction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

13.9 Relational alphabet extension

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

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

end

14 Meta-level substitution

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

Meta substitution substituted a HOL variable in a UTP expression for another UTP expression. It is analogous to UTP substitution, but acts on functions.

lift-definition *msubst* :: $('b \Rightarrow ('a, '\alpha)$ *uexpr*$) \Rightarrow ('b, '\alpha)$ *uexpr* $\Rightarrow ('a, '\alpha)$ *uexpr*
is λ *F v b*. *F* (*v b*) *b* **.**

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

syntax
  -msubst :: *logic* $\Rightarrow$ *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ((-⟦-→-⟧) [*990*,*0*,*0*] *991*)

translations
  -msubst P x v == *CONST msubst* (λ *x*. *P*) *v*

lemma *msubst-true* [*usubst*]: *true*⟦*x*→*v*⟧ = *true*
  by (*pred-auto*)

lemma *msubst-false* [*usubst*]: *false*⟦*x*→*v*⟧ = *false*
  by (*pred-auto*)

lemma *msubst-lit* [*usubst*]: «*x*»⟦*x*→*v*⟧ = *v*
  by (*pred-auto*)

lemma *msubst-lit-2-1* [*usubst*]: «*x*»⟦(*x*,*y*)→$(u,v)_u$⟧ = *u*
  by (*pred-auto*)

lemma *msubst-lit-2-2* [*usubst*]: «*y*»⟦(*x*,*y*)→$(u,v)_u$⟧ = *v*
  by (*pred-auto*)

lemma *msubst-lit'* [*usubst*]: «*y*»⟦*x*→*v*⟧ = «*y*»
  by (*pred-auto*)

lemma *msubst-lit'-2* [*usubst*]: «*z*»⟦(*x*,*y*)→*v*⟧ = «*z*»
  by (*pred-auto*)

lemma *msubst-uop* [*usubst*]: $(uop\ f\ (v\ x))[\![x \to u]\!] = uop\ f\ ((v\ x)[\![x \to u]\!])$
by (*rel-auto*)

lemma *msubst-uop-2* [*usubst*]: $(uop\ f\ (v\ x\ y))[\![(x,y) \to u]\!] = uop\ f\ ((v\ x\ y)[\![(x,y) \to u]\!])$
by (*pred-simp*, *pred-simp*)

lemma *msubst-bop* [*usubst*]: $(bop\ f\ (v\ x)\ (w\ x))[\![x \to u]\!] = bop\ f\ ((v\ x)[\![x \to u]\!])\ ((w\ x)[\![x \to u]\!])$
by (*rel-auto*)

lemma *msubst-bop-2* [*usubst*]: $(bop\ f\ (v\ x\ y)\ (w\ x\ y))[\![(x,y) \to u]\!] = bop\ f\ ((v\ x\ y)[\![(x,y) \to u]\!])\ ((w\ x\ y)[\![(x,y) \to u]\!])$
by (*pred-simp*, *pred-simp*)

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

lemma *msubst-not-2* [*usubst*]: $(\neg\ P\ x\ y)[\![(x,y) \to v]\!] = (\neg\ ((P\ x\ y)[\![(x,y) \to v]\!]))$
by (*pred-auto*)+

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

lemma *msubst-disj-2* [*usubst*]: $(P\ x\ y \vee Q\ x\ y)[\![(x,y) \to v]\!] = ((P\ x\ y)[\![(x,y) \to v]\!] \vee (Q\ x\ y)[\![(x,y) \to v]\!])$
by (*pred-auto*)+

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

lemma *msubst-conj-2* [*usubst*]: $(P\ x\ y \wedge Q\ x\ y)[\![(x,y) \to v]\!] = ((P\ x\ y)[\![(x,y) \to v]\!] \wedge (Q\ x\ y)[\![(x,y) \to v]\!])$
by (*pred-auto*)+

lemma *msubst-implies* [*usubst*]:
$(P\ x \Rightarrow Q\ x)[\![x \to v]\!] = ((P\ x)[\![x \to v]\!] \Rightarrow (Q\ x)[\![x \to v]\!])$
by (*pred-auto*)

lemma *msubst-implies-2* [*usubst*]:
$(P\ x\ y \Rightarrow Q\ x\ y)[\![(x,y) \to v]\!] = ((P\ x\ y)[\![(x,y) \to v]\!] \Rightarrow (Q\ x\ y)[\![(x,y) \to v]\!])$
by (*pred-auto*)+

lemma *msubst-shAll* [*usubst*]:
$(\forall\ x \cdot P\ x\ y)[\![y \to v]\!] = (\forall\ x \cdot (P\ x\ y)[\![y \to v]\!])$
by (*pred-auto*)

lemma *msubst-shAll-2* [*usubst*]:
$(\forall\ x \cdot P\ x\ y\ z)[\![(y,z) \to v]\!] = (\forall\ x \cdot (P\ x\ y\ z)[\![(y,z) \to v]\!])$
by (*pred-auto*)+

lemma *msubst-event* [*usubst*]:
$(c \cdot v\ x)_u[\![x \to u]\!] = (c \cdot (v\ x)[\![x \to u]\!])_u$
by (*pred-simp*)

lemma *msubst-event-2* [*usubst*]:
$(c \cdot v\ x\ y)_u[\![(x,y) \to u]\!] = (c \cdot (v\ x\ y)[\![(x,y) \to u]\!])_u$
by (*pred-simp*)+

lemma *msubst-var* [*usubst*]:
$(utp\text{-}expr.var\ x)[\![y{\rightarrow}u]\!] = utp\text{-}expr.var\ x$
by (*pred-simp*)

lemma *msubst-var-2* [*usubst*]:
$(utp\text{-}expr.var\ x)[\![(y,z){\rightarrow}u]\!] = utp\text{-}expr.var\ x$
by (*pred-simp*)+

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

15 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

16 Relational Calculus Laws

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

16.1 Conditional Laws

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

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

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

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

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

16.2 Precondition and Postcondition Laws

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

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

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

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

theorem *precond-left-zero*:
assumes $out\alpha \sharp p$ $p \neq false$
shows $(true \;;\; p) = true$
using *assms* **by** (*rel-auto*)

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

16.3 Sequential Composition Laws

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*

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *seqr-left-one-point-true*:
assumes *vwb-lens x*
shows $((P \wedge \$x\acute{}) \mathbin{;;} Q) = (P[\![true/\$x\acute{}]\!] \mathbin{;;} Q[\![true/\$x]\!])$

by (*metis assms seqr-left-one-point true-alt-def upred-eq-true*)

lemma *seqr-left-one-point-false*:
assumes *vwb-lens x*
shows $((P \wedge \neg \$x\acute{}) ;; Q) = (P⟦false/\$x\acute{}⟧ ;; Q⟦false/\$x⟧)$
by (*metis assms false-alt-def seqr-left-one-point upred-eq-false*)

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

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

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

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

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

lemma *seqr-bool-split*:
assumes *vwb-lens x*
shows $P ;; Q = (P⟦true/\$x\acute{}⟧ ;; Q⟦true/\$x⟧ \vee P⟦false/\$x\acute{}⟧ ;; Q⟦false/\$x⟧)$
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 $(P \triangleleft \$x\acute{} \triangleright Q) ;; R = (P⟦true/\$x\acute{}⟧ ;; R⟦true/\$x⟧ \vee Q⟦false/\$x\acute{}⟧ ;; R⟦false/\$x⟧)$
proof −
have $(P \triangleleft \$x\acute{} \triangleright Q) ;; R = ((\$x\acute{} \wedge P) ;; R \vee (\neg \$x\acute{} \wedge Q) ;; R)$
by (*simp add*: *cond-def seqr-or-distl*)
also have ... $= ((P \wedge \$x\acute{}) ;; R \vee (Q \wedge \neg \$x\acute{}) ;; R)$
by (*rel-auto*)
also have ... $= (P⟦true/\$x\acute{}⟧ ;; R⟦true/\$x⟧ \vee Q⟦false/\$x\acute{}⟧ ;; R⟦false/\$x⟧)$
by (*simp add*: *seqr-left-one-point-true seqr-left-one-point-false assms*)
finally show *?thesis* **.**
qed

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

by (*rel-auto*)

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

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

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

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

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

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

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

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

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

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

16.4 Iterated Sequential Composition Laws

lemma *iter-seqr-nil* [*simp*]: $(;;\ i : [] \cdot P(i)) = II$
by (*simp add*: *seqr-iter-def*)

lemma *iter-seqr-cons* [*simp*]: $(;;\ i : (x \# xs) \cdot P(i)) = P(x) \;;\; (;;\ i : xs \cdot P(i))$
by (*simp add*: *seqr-iter-def*)

16.5 Quantale Laws

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

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

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

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

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

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

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

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

16.6 Skip Laws

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 :: $(bool \Longrightarrow {'}\alpha)$
shows $(\$x \wedge II) = (II \wedge \$x')$
by (*rel-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*:
$\llbracket$ *vwb-lens* y; $x +_L y \approx_L 1_L$; $x \bowtie y$ $\rrbracket \Longrightarrow II\restriction_\alpha x = II_y$
apply (*rel-auto*)
apply (*metis* (*no-types*, *lifting*) *lens-indep-def*)
apply (*metis vwb-lens.put-eq*)
done

16.7 Assignment Laws

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

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

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

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

lemma *assigns-idem*: $\mathit{mwb\text{-}lens}\ x \Longrightarrow (x,x := u,v) = (x := v)$
  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*:
  $\mathit{bij}\ f \Longrightarrow \langle f\rangle_a{}^- = \langle \mathit{inv}\ f\rangle_a$
  by (*rel-auto*, *simp-all add*: *bij-is-inj bij-is-surj surj-f-inv-f*)

lemma *assign-pred-transfer*:
  fixes $x :: ('a \Longrightarrow '\alpha)$
  assumes $\$x \sharp b$ $\mathit{out}\alpha \sharp b$
  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 alpha*)

lemma *assign-test*: $\mathit{mwb\text{-}lens}\ x \Longrightarrow (x := \ll u\gg \;;; x := \ll v\gg) = (x := \ll v\gg)$
  by (*simp add*: *assigns-comp usubst*)

lemma *assign-twice*: $[\![\ \mathit{mwb\text{-}lens}\ x;\ x \sharp f\ ]\!] \Longrightarrow (x := e \;;; x := f) = (x := f)$
  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 $x :: ('a \Longrightarrow '\alpha)$
  assumes $\mathit{out}\alpha \sharp b$
  shows $(x := e \;;; (P \triangleleft b \triangleright Q)) = ((x := e \;;; P) \triangleleft (b[\![\lceil e\rceil_</\$x]\!]) \triangleright (x := e \;;; Q))$
  by (*rel-auto*)

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

lemma *assign-r-alt-def*:
  fixes $x :: ('a \Longrightarrow '\alpha)$

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 (*simp add*: *assigns-r-uinj swap-usubst-inj subst-upd-pr-var*)

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*

16.8 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 $x :: ('a \Longrightarrow '\alpha)$
shows $(\$x)^{-} = \$x'$
by *pred-auto*

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

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

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

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

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

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

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

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

by (*rel-auto*)

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

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

16.9 Assertion and Assumption Laws

declare *sublens-def* [*lens-defs del*]

lemma *assume-false*: $[false]^\top = false$
by (*rel-auto*)

lemma *assume-true*: $[true]^\top = II$
by (*rel-auto*)

lemma *assume-seq*: $[b]^\top ;; [c]^\top = [b \wedge c]^\top$
by (*rel-auto*)

lemma *assert-false*: $\{false\}_\bot = true$
by (*rel-auto*)

lemma *assert-true*: $\{true\}_\bot = II$
by (*rel-auto*)

lemma *assert-seq*: $\{b\}_\bot ;; \{c\}_\bot = \{b \wedge c\}_\bot$
by (*rel-auto*)

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

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

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

lemma *antiframe-skip* [*simp*]:
vwb-lens $x \Longrightarrow x:[II] = II$
by (*rel-auto*)

lemma *antiframe-assign-in*:
$\llbracket$ *vwb-lens* a; $x \subseteq_L a$ $\rrbracket \Longrightarrow a:[x := v] = x := v$
by (*rel-auto*, *simp-all add*: *lens-get-put-quasi-commute lens-put-of-quotient*)

lemma *nameset-skip*: *vwb-lens* $x \Longrightarrow (ns\ x \cdot II) = II_x$
by (*rel-auto*, *meson vwb-lens-wb wb-lens.get-put*)

lemma *nameset-skip-ra*: *vwb-lens* $x \Longrightarrow (ns\ x \cdot II_x) = II_x$
by (*rel-auto*)

declare *sublens-def* [*lens-defs*]

16.10 While Loop Laws

theorem *while-unfold*:
  $while\ b\ do\ P\ od = ((P\ ;;\ while\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$
proof −
  have m:$mono\ (\lambda X.\ (P\ ;;\ X) \triangleleft b \triangleright_r II)$
    by (*auto intro*: *monoI seqr-mono cond-mono*)
  have $(while\ b\ do\ P\ od) = (\nu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)$
    by (*simp add*: *while-def*)
  also have ... $= ((P\ ;;\ (\nu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
    by (*subst lfp-unfold*, *simp-all add*: *m*)
  also have ... $= ((P\ ;;\ while\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$
    by (*simp add*: *while-def*)
  finally show *?thesis* **.**
qed

theorem *while-false*: $while\ false\ do\ P\ od = II$
  by (*subst while-unfold*, *simp add*: *aext-false*)

theorem *while-true*: $while\ true\ do\ P\ od = false$
  apply (*simp add*: *while-def alpha*)
  apply (*rule antisym*)
  apply (*simp-all*)
  apply (*rule lfp-lowerbound*)
  apply (*simp*)
done

theorem *while-bot-unfold*:
  $while_\bot\ b\ do\ P\ od = ((P\ ;;\ while_\bot\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$
proof −
  have m:$mono\ (\lambda X.\ (P\ ;;\ X) \triangleleft b \triangleright_r II)$
    by (*auto intro*: *monoI seqr-mono cond-mono*)
  have $(while_\bot\ b\ do\ P\ od) = (\mu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)$
    by (*simp add*: *while-bot-def*)
  also have ... $= ((P\ ;;\ (\mu\ X \cdot (P\ ;;\ X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
    by (*subst gfp-unfold*, *simp-all add*: *m*)
  also have ... $= ((P\ ;;\ while_\bot\ b\ do\ P\ od) \triangleleft b \triangleright_r II)$
    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

16.11 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 derived from semirings

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

translations
  P ^ *i* <= *CONST power.power II op* ;; *P i*
  P ^ *i* <= (*CONST power.power II op* ;; *P*) *i*

Set up transfer tactic for powers

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

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

update-uexpr-rep-eq-thms

lemma *Sup-power-expand*:
  fixes *P* :: *nat* ⇒ *'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*: $(\bigsqcap i\in\{0..Suc\ n\}.\ P \mathbin{\hat{}} i) = (\bigsqcap i\in\{0..n\}.\ P \mathbin{\hat{}} i) \sqcap P \mathbin{\hat{}} Suc\ n$
proof −
  have $(\bigsqcap i\in\{0..Suc\ n\}.\ P \mathbin{\hat{}} i) = (\bigsqcap i\in insert\ (Suc\ n)\ \{0..n\}.\ P \mathbin{\hat{}} i)$
    by (*simp add*: *atLeast0-atMost-Suc*)
  also have ... $= P \mathbin{\hat{}} Suc\ n \sqcap (\bigsqcap i\in\{0..n\}.\ P \mathbin{\hat{}} i)$
    by (*simp*)
  finally show *?thesis*

by (*simp add*: *Lattices.sup-commute*)
qed

The following two proofs are adapted from the AFP entry Kleene Algebra. See also [2, 1].

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

lemma *SUP-atLeastAtMost-first*:
fixes $P :: nat \Rightarrow {'a}{::}complete\text{-}lattice$
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*)

lemma *upower-seqr-iter*: $P \mathbin{\hat{}} n = (;;\ Q : replicate\ n\ P \cdot Q)$
by (*induct n*, *simp-all*)

16.11.1 Kleene Star

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

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

update-uexpr-rep-eq-thms

16.11.2 Omega

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

16.12 Relation Algebra Laws

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

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

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

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

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

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

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

16.13 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 $Q \sqsubseteq R$ $Q \sqsubseteq P \;;\; Q$
  shows $Q \sqsubseteq P^{\star} \;;\; R$
proof –
  have $P^{\star} \;;\; R = (\bigsqcap\ i.\ P\ \hat{}\ i \;;\; R)$
    by (*simp add*: *ustar-def UINF-as-Sup-collect′ seq-SUP-distr*)
  also have $Q \sqsubseteq \ldots$
    by (*simp add*: *SUP-least assms upower-inductl*)
  finally show *?thesis* **.**
qed

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

lemma *ustar-refines-nu*: $(\nu\ X \cdot P \;;\; X \sqcap II) \sqsubseteq P^{\star}$
  by (*metis* (*no-types*, *lifting*) *lfp-greatest semilattice-sup-class.le-sup-iff*
    semilattice-sup-class.sup-idem upred-semiring.mult-2-right
    upred-semiring.one-add-one ustar-inductl)

lemma *ustar-as-nu*: $P^{\star} = (\nu\ X \cdot P \;;\; X \sqcap II)$

proof (*rule antisym*)
  show $(\nu\ X \cdot P\ ;;\ X \sqcap II) \sqsubseteq P^\star$
    by (*simp add*: *ustar-refines-nu*)
  show $P^\star \sqsubseteq (\nu\ X \cdot P\ ;;\ X \sqcap II)$
    by (*metis lfp-lowerbound upred-semiring.add-commute ustar-unfoldl*)
qed

16.14 Omega Algebra Laws

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

end

16.15 Relational Hoare calculus

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

named-theorems *hoare*

method *hoare-auto* = ((*simp add*: *assigns-r-comp usubst unrest*)?, — Eliminate assignments where possible
  auto intro!: *hoare simp add*: *usubst unrest* — Apply Hoare logic laws
)

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

lemma *while-hoare-r* [*hoare*]:
  assumes $\{\!|p \wedge b|\!\}S\{\!|p|\!\}_u$
  shows $\{\!|p|\!\}$*while* b *do* S *od*$\{\!|\neg b \wedge p|\!\}_u$
  using *assms*
  by (*simp add*: *while-def hoare-r-def*, *rule-tac lfp-lowerbound*) (*rel-auto*)

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

16.16 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*]

lemma *wp-true* [*wp*]: *p wp true* = *true*
  by (*rel-simp*)

theorem *wp-assigns-r* [*wp*]:
  $\langle\sigma\rangle_a$ *wp* $r = \sigma \dagger r$
  by *rel-auto*

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

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

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

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

theorem *wp-cond* [*wp*]: $(P \triangleleft b \triangleright_r Q)$ *wp* $r = ((b \Rightarrow P$ *wp* $r) \wedge ((\neg\ b) \Rightarrow Q$ *wp* $r))$
  by *rel-auto*

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

by *rel-auto*

If two programs have the same weakest precondition for any postcondition then the programs are the same.

theorem *wp-eq-intro*: $\llbracket \bigwedge r.\ P\ wp\ r = Q\ wp\ r \rrbracket \Longrightarrow P = Q$
by (*rel-auto robust*, *fastforce+*)
end

17 UTP Theories

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

Closure laws for theories

named-theorems *closure*

17.1 Complete lattice of predicates

definition *upred-lattice* :: ($'\alpha$ *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* :: $'\alpha$ *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*)

17.2 Healthiness conditions

type-synonym $'\alpha$ *health* = $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred*

definition
Healthy::$'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *health* $\Rightarrow$ *bool* (**infix** *is 30*)
where P *is* $H \equiv (H\ P = P)$

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

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

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

abbreviation *Healthy-carrier* :: $'\alpha$ *health* $\Rightarrow$ $'\alpha$ *upred set* ($[\![\text{-}]\!]_H$)
where $[\![H]\!]_H \equiv \{P.\ P \textit{ is } H\}$

lemma *Healthy-carrier-image*:
$A \subseteq [\![\mathcal{H}]\!]_H \Longrightarrow \mathcal{H}\ `\ A = A$
by (*auto simp add*: *image-def*, (*metis Healthy-if mem-Collect-eq subsetCE*)+)

lemma *Healthy-carrier-Collect*: $A \subseteq [\![H]\!]_H \Longrightarrow A = \{H(P) \mid P.\ P \in A\}$
by (*simp add*: *Healthy-carrier-image Setcompr-eq-image*)

lemma *Healthy-func*:
$[\![\ F \in [\![\mathcal{H}_1]\!]_H \to [\![\mathcal{H}_2]\!]_H;\ P \textit{ is } \mathcal{H}_1\ ]\!] \Longrightarrow \mathcal{H}_2(F(P)) = F(P)$
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*:
$[\![\ P \in F\ `\ A;\ \bigwedge x.\ F\ x \textit{ is } H\ ]\!] \Longrightarrow P \textit{ is } H$
by *blast*

lemma *Healthy-SUPREMUM*:
$A \subseteq [\![H]\!]_H \Longrightarrow \textit{SUPREMUM}\ A\ H = \bigsqcap\ A$
by (*drule Healthy-carrier-image*, *presburger*)

lemma *Healthy-INFIMUM*:
$A \subseteq [\![H]\!]_H \Longrightarrow \textit{INFIMUM}\ A\ H = \bigsqcup\ A$
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*: $[\![A \subseteq [\![H]\!]_H; P \in A]\!] \Longrightarrow P \text{ is } H$
  by *blast*

17.3 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*]: $Continuous\ H = (\forall A. A \neq \{\} \longrightarrow H (\bigsqcap A) = \bigsqcap (H \text{ ` } A))$

lemma *Healthy-Idempotent* [*closure*]:
  $Idempotent\ H \Longrightarrow H(P) \text{ is } H$
  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*: $Idempotent\ f \Longrightarrow f \text{ ` } f \text{ ` } A = f \text{ ` } A$
  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 $HC(P \triangleleft b \triangleright Q) = (HC(P) \triangleleft b \triangleright HC(Q))$
  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*; $\bigwedge$ *i*. *P*(*i*) *is H* $]\!] \Longrightarrow$ ($\bigsqcap$ *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* $\Longrightarrow$ *sup-continuous F*
  by (*simp add*: *Continuous-def sup-continuous-def*)

lemma *Healthy-fixed-points* [*simp*]: *fps* $\mathcal{P}$ *H* = $[\![H]\!]_H$
  by (*simp add*: *fps-def upred-lattice-def Healthy-def*)

lemma *USUP-healthy*: $A \subseteq [\![H]\!]_H \Longrightarrow (\bigsqcup\ P{\in}A \cdot F(P)) = (\bigsqcup\ P{\in}A \cdot F(H(P)))$
  by (*rule USUP-cong*, *simp add*: *Healthy-subset-member*)

lemma *UINF-healthy*: $A \subseteq [\![H]\!]_H \Longrightarrow (\bigsqcap\ P{\in}A \cdot F(P)) = (\bigsqcap\ P{\in}A \cdot F(H(P)))$
  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*)

17.4 UTP theories hierarchy

typedef ($'\mathcal{T}$, $'\alpha$) *uthy* = *UNIV* :: *unit set*
  by *auto*

We create a unitary parametric type to represent UTP theories. These are merely tags and contain no data other than to help the type-system resolve polymorphic definitions. The two parameters denote the name of the UTP theory – as a unique type – and the minimal alphabet that the UTP theory requires. We will then use Isabelle's ad-hoc overloading mechanism to associate theory constructs, like healthiness conditions and units, with each of these types. This will allow the type system to retrieve definitions based on a particular theory context.

definition *uthy* :: ($'a$, $'b$) *uthy* **where**
uthy = *Abs-uthy* ()

lemma *uthy-eq* [*intro*]:
  fixes *x y* :: ($'a$, $'b$) *uthy*
  shows *x* = *y*
  by (*cases x*, *cases y*, *simp*)

syntax
  -UTHY :: *type* ⇒ *type* ⇒ *logic* (*UTHY*'(-, -'))

translations
  UTHY($'T$, $'\alpha$) == *CONST uthy* :: ($'T$, $'\alpha$) *uthy*

We set up polymorphic constants to denote the healthiness conditions associated with a UTP theory. Unfortunately we can currently only characterise UTP theories of homogeneous relations; this is due to restrictions in the instantiation of Isabelle's polymorphic constants which apparently cannot specialise types in this way.

consts
  utp-hcond :: $('\mathcal{T}, '\alpha)$ *uthy* $\Rightarrow$ $('\alpha \times '\alpha)$ *health* $(\mathcal{H}_1)$

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

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

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

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

end

17.5 UTP theory hierarchy

We next define a hierarchy of locales that characterise different classes of UTP theory. Minimally we require that a UTP theory's healthiness condition is idempotent.

locale *utp-theory* =
  fixes $\mathcal{T}$ :: $('\mathcal{T},\ '\alpha)$ *uthy* (**structure**)
  assumes *HCond-Idem*: $\mathcal{H}(\mathcal{H}(P)) = \mathcal{H}(P)$
begin

  lemma *uthy-simp*:
    uthy = $\mathcal{T}$
    by *blast*

A UTP theory fixes $\mathcal{T}$, the structural element denoting the UTP theory. All constants associated with UTP theories can then be resolved by the type system.

  lemma *HCond-Idempotent* [*closure*,*intro*]: *Idempotent* $\mathcal{H}$
    by (*simp add*: *Idempotent-def HCond-Idem*)

  sublocale *partial-order uthy-order* $\mathcal{T}$
    by (*unfold-locales*, *simp-all add*: *utp-order-def*)
end

Theory summation is commutative provided the healthiness conditions commute.

lemma *uthy-plus-comm*:
  assumes $\mathcal{H}_{T_1} \circ \mathcal{H}_{T_2} = \mathcal{H}_{T_2} \circ \mathcal{H}_{T_1}$
  shows $T_1 +_T T_2 \approx_T T_2 +_T T_1$
proof −
  have T_1 = *uthy* T_2 = *uthy*
    by *blast+*
  thus *?thesis*
    using *assms* **by** (*simp add*: *uth-eq-def prod-hcond-def*)
qed

lemma *uthy-plus-assoc*: $T_1 +_T (T_2 +_T T_3) \approx_T (T_1 +_T T_2) +_T T_3$
  by (*simp add*: *uth-eq-def prod-hcond-def comp-def*)

lemma *uthy-plus-idem*: *utp-theory* $T \Longrightarrow T +_T T \approx_T T$

by (*simp add*: *uth-eq-def prod-hcond-def Healthy-def utp-theory.HCond-Idem utp-theory.uthy-simp*)

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

The healthiness conditions of a UTP theory lattice form a complete lattice, and allows us to make use of complete lattice results from HOL-Algebra, such as the Knaster-Tarski theorem. We can also retrieve lattice operators as below.

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

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

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

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

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

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

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

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

syntax
 -tmu :: *logic* $\Rightarrow$ *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\boldsymbol{\mu}$ı - $\cdot$ - [*0*, *10*] *10*)
 -tnu :: *logic* $\Rightarrow$ *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\boldsymbol{\nu}$ı - $\cdot$ - [*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 $\{P.\ (P\ is\ \mathcal{H}) \wedge F\ P \sqsubseteq P\} = \{P.\ (P\ is\ \mathcal{H}) \wedge F\ (\mathcal{H}\ P) \sqsubseteq P\}$
   by (*auto simp add*: *Healthy-def*)

thus *?thesis*
by (*simp add*: *LFP-def*)
qed

lemma *GFP-healthy-comp*: $\boldsymbol{\nu}\ F = \boldsymbol{\nu}\ (F \circ \mathcal{H})$
proof –
have $\{P.\ (P\ \mathit{is}\ \mathcal{H}) \wedge P \sqsubseteq F\ P\} = \{P.\ (P\ \mathit{is}\ \mathcal{H}) \wedge P \sqsubseteq F\ (\mathcal{H}\ P)\}$
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*: $P\ \mathit{is}\ \mathcal{H} \Longrightarrow P \sqsubseteq \top$
using *weak.top-higher* **by** *auto*

lemma *utp-bottom*: $P\ \mathit{is}\ \mathcal{H} \Longrightarrow \bot \sqsubseteq P$
using *weak.bottom-lower* **by** *auto*

end

lemma *upred-top*: $\top_{\mathcal{P}} = \mathit{false}$
using *ball-UNIV greatest-def* **by** *fastforce*

lemma *upred-bottom*: $\bot_{\mathcal{P}} = \mathit{true}$
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* $\subseteq$ *utp-theory-lattice*
proof –

We can then use the Knaster-Tarski theorem to obtain a complete lattice, and thus provide all the usual properties.

interpret *weak-complete-lattice fpl* $\mathcal{P}$ $\mathcal{H}$
by (*rule Knaster-Tarski*, *auto simp add*: *upred-lattice.weak.weak-complete-lattice-axioms*)

have *complete-lattice* (*fpl* $\mathcal{P}$ $\mathcal{H}$)
by (*unfold-locales*, *simp add*: *fps-def sup-exists*, (*blast intro*: *sup-exists inf-exists*)+)

hence *complete-lattice* (*uthy-order* $\mathcal{T}$)
by (*simp add*: *utp-order-def*, *simp add*: *upred-lattice-def*)

thus *utp-theory-lattice* $\mathcal{T}$
by (*simp add*: *utp-theory-axioms utp-theory-lattice-def*)
qed

context *utp-theory-mono*
begin

In a monotone theory, the top and bottom can always be obtained by applying the healthiness condition to the predicate top and bottom, respectively.

lemma *healthy-top*: $\top = \mathcal{H}(\mathit{false})$
proof –
  have $\top = \top_{\mathit{fpl}\ \mathcal{P}\ \mathcal{H}}$
    by (*simp add*: *utp-order-fpl*)
  also have ... $= \mathcal{H}\ \top_{\mathcal{P}}$
    using *Knaster-Tarski-idem-extremes*(*1*)[*of* $\mathcal{P}$ $\mathcal{H}$]
    by (*simp add*: *HCond-Idempotent HCond-Mono*)
  also have ... $= \mathcal{H}$ *false*
    by (*simp add*: *upred-top*)
  finally show *?thesis* **.**
qed

lemma *healthy-bottom*: $\bot = \mathcal{H}(\mathit{true})$
proof –
  have $\bot = \bot_{\mathit{fpl}\ \mathcal{P}\ \mathcal{H}}$
    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*: $\mathit{Mono}_{\mathit{uthy\text{-}order}\ \mathcal{T}}\ \mathcal{H}$
    by (*simp add*: *HCond-Mono isotone-utp-orderI*)
  have *3*: $\mathit{Idem}_{\mathit{uthy\text{-}order}\ \mathcal{T}}\ \mathcal{H}$
    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* $\subseteq$ *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*: $\{P.\ (P \textit{ is } \mathcal{H}) \land F\ P \sqsubseteq P\} \neq \{\}$
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*: $\bigwedge P. F(\mathcal{H}(P)) = \mathcal{H}(F(\mathcal{H}(P)))$
by (*metis HCond-Idem Healthy-def assms*(*2*) *funcset-mem mem-Collect-eq*)
show *?thesis*
proof (*rule Sup-least*, *auto*)
fix P
assume *a*: $F\ (\mathcal{H}\ P) \sqsubseteq P$
hence *F*: $(F\ (\mathcal{H}\ P)) \sqsubseteq (\mathcal{H}\ P)$
by (*metis 1 HCond-Mono mono-def*)
show $\bigsqcap\{P. (P \textit{ is } \mathcal{H}) \wedge F\ P \sqsubseteq P\} \sqsubseteq P$
proof (*rule Sup-upper2*[*of* $F\ (\mathcal{H}\ P)$])
show $F\ (\mathcal{H}\ P) \in \{P. (P \textit{ is } \mathcal{H}) \wedge F\ P \sqsubseteq P\}$
proof (*auto*)
show $F\ (\mathcal{H}\ P) \textit{ is } \mathcal{H}$
by (*metis 1 Healthy-def*)
show $F\ (F\ (\mathcal{H}\ P)) \sqsubseteq F\ (\mathcal{H}\ P)$
using *F mono-def assms*(*1*) **by** *blast*
qed
show $F\ (\mathcal{H}\ P) \sqsubseteq P$
by (*simp add*: *a*)
qed
qed
qed

with *ne* **show** *?thesis*
by (*simp add*: *LFP-def gfp-def*, *subst healthy-inf-cont*, *auto simp add*: *lfp-def*)
qed
from *ne* **show** $(\mu\ X \cdot F\ (\mathcal{H}\ X)) \sqsubseteq \boldsymbol{\mu}\ F$
apply (*simp add*: *LFP-def gfp-def*, *subst healthy-inf-cont*, *auto simp add*: *lfp-def*)
apply (*rule Sup-least*)
apply (*auto simp add*: *Healthy-def Sup-upper*)
done
qed

end

In another direction, we can also characterise UTP theories that are relational. Minimally this requires that the healthiness condition is closed under sequential composition.

locale *utp-theory-rel* =
utp-theory +
assumes *Healthy-Sequence* [*closure*]: $[\![\ P \textit{ is } \mathcal{H};\ Q \textit{ is } \mathcal{H}\ ]\!] \Longrightarrow (P\ ;;\ Q) \textit{ is } \mathcal{H}$

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

lemma *seq-cont-Sup-distl*:
assumes $P \textit{ is } \mathcal{H}\ A \subseteq [\![\mathcal{H}]\!]_H\ A \neq \{\}$
shows $P\ ;;\ (\bigsqcap\ A) = \bigsqcap\ \{P\ ;;\ Q \mid Q.\ Q \in A\ \}$
proof –
have $\{P\ ;;\ Q \mid Q.\ Q \in A\ \} \subseteq [\![\mathcal{H}]\!]_H$
using *Healthy-Sequence assms*(*1*) *assms*(*2*) **by** (*auto*)
thus *?thesis*
by (*simp add*: *healthy-inf-cont seq-Sup-distl setcompr-eq-image assms*)

qed

lemma *seq-cont-Sup-distr*:
assumes Q *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* $(\text{-}\star_1\ [999]\ 999)$ **where**
utp-star $\mathcal{T}$ $P = (\boldsymbol{\nu}_{\mathcal{T}}\ (\lambda\ X.\ (P\ ;;\ X) \sqcap_{\mathcal{T}} \mathcal{II}_{\mathcal{T}}))$

definition *utp-omega* $(\text{-}\boldsymbol{\omega}_1\ [999]\ 999)$ **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* $\subseteq$ *utp-theory-left-unital*
by (*simp add*: *Healthy-Unit Unit-Left Healthy-Sequence utp-theory-rel-def utp-theory-axioms utp-theory-rel-axioms-def utp-theory-left-unital-axioms-def utp-theory-left-unital-def*)

sublocale *utp-theory-unital* $\subseteq$ *utp-theory-right-unital*
by (*simp add*: *Healthy-Unit Unit-Right Healthy-Sequence utp-theory-rel-def utp-theory-axioms utp-theory-rel-axioms-def utp-theory-right-unital-axioms-def utp-theory-right-unital-def*)

17.6 Theory of relations

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

typedecl *REL*
abbreviation *REL* $\equiv$ *UTHY*(*REL*, $'\alpha$)

We declare the type *REL* to be the tag for this theory. We need know nothing about this type (other than it's non-empty), since it is merely a name. We also create the corresponding constant to refer to the theory. Then we can use it to instantiate the relevant polymorphic constants.

overloading
rel-hcond == *utp-hcond* :: (*REL*, $'\alpha$) *uthy* $\Rightarrow$ ($'\alpha \times {'\alpha}$) *health*
rel-unit == *utp-unit* :: (*REL*, $'\alpha$) *uthy* $\Rightarrow$ $'\alpha$ *hrel*
begin

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

definition *rel-hcond* :: (*REL*, $'\alpha$) *uthy* $\Rightarrow$ ($'\alpha \times {'\alpha}$) *upred* $\Rightarrow$ ($'\alpha \times {'\alpha}$) *upred* **where**
rel-hcond T = *id*

The unit of the theory is simply the relational unit.

definition *rel-unit* :: (*REL*, $'\alpha$) *uthy* $\Rightarrow$ $'\alpha$ *hrel* **where**
rel-unit T = *II*
end

Finally we can show that relations are a monotone and unital theory using a locale interpretation, which requires that we prove all the relevant properties. It's convenient to rewrite some of the theorems so that the provisos are more UTP like; e.g. that the carrier is the set of healthy predicates.

interpretation *rel-theory*: *utp-theory-mono-unital REL*
rewrites *carrier* (*uthy-order REL*) = $[\![id]\!]_H$
by (*unfold-locales*, *simp-all add*: *rel-hcond-def rel-unit-def Healthy-def*)

We can then, for instance, determine what the top and bottom of our new theory is.

lemma *REL-top*: $\top_{REL}$ = *false*
by (*simp add*: *rel-theory.healthy-top*, *simp add*: *rel-hcond-def*)

lemma *REL-bottom*: $\bot_{REL} = true$
by (*simp add*: *rel-theory.healthy-bottom*, *simp add*: *rel-hcond-def*)

A number of theorems have been exported, such at the fixed point unfolding laws.

thm *rel-theory.GFP-unfold*

17.7 Theory links

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

definition *mk-conn* ($- \Leftarrow\langle -,- \rangle\Rightarrow -$ [*90,0,0,91*] *91*) **where**
$H1 \Leftarrow\langle \mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow H2 \equiv (\!|\ orderA = utp\text{-}order\ H1,\ orderB = utp\text{-}order\ H2,\ lower = \mathcal{H}_2,\ upper = \mathcal{H}_1\ |\!)$

abbreviation *mk-conn'* ($- \leftarrow\langle -,- \rangle\rightarrow -$ [*90,0,0,91*] *91*) **where**
$T1 \leftarrow\langle \mathcal{H}_1,\mathcal{H}_2\rangle\rightarrow T2 \equiv \mathcal{H}_{T1} \Leftarrow\langle \mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow \mathcal{H}_{T2}$

lemma *mk-conn-orderA* [*simp*]: $\mathcal{X}_{H1 \Leftarrow\langle \mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow H2} = utp\text{-}order\ H1$
by (*simp add:mk-conn-def*)

lemma *mk-conn-orderB* [*simp*]: $\mathcal{Y}_{H1 \Leftarrow\langle \mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow H2} = utp\text{-}order\ H2$
by (*simp add:mk-conn-def*)

lemma *mk-conn-lower* [*simp*]: $\pi_{*H1 \Leftarrow\langle \mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow H2} = \mathcal{H}_1$
by (*simp add*: *mk-conn-def*)

lemma *mk-conn-upper* [*simp*]: $\pi^*{}_{H1 \Leftarrow\langle \mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow H2} = \mathcal{H}_2$
by (*simp add*: *mk-conn-def*)

lemma *galois-comp*: $(H_2 \Leftarrow\langle \mathcal{H}_3,\mathcal{H}_4\rangle\Rightarrow H_3) \circ_g (H_1 \Leftarrow\langle \mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow H_2) = H_1 \Leftarrow\langle \mathcal{H}_1\circ\mathcal{H}_3,\mathcal{H}_4\circ\mathcal{H}_2\rangle\Rightarrow H_3$
by (*simp add*: *comp-galcon-def mk-conn-def*)

Example Galois connection / retract: Existential quantification

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

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

lemma *ex-closed-unrest*:
vwb-lens $x \Longrightarrow [\![ex\ x]\!]_H = \{P.\ x \sharp P\}$
by (*simp add*: *Healthy-def unrest-as-exists*)

Any theory can be composed with an existential quantification to produce a Galois connection

theorem *ex-retract*:
assumes *vwb-lens* x *Idempotent* H $ex\ x \circ H = H \circ ex\ x$
shows *retract* $((ex\ x \circ H) \Leftarrow\langle ex\ x,\ H\rangle\Rightarrow H)$
proof (*unfold-locales*, *simp-all*)
show $H \in [\![ex\ x \circ H]\!]_H \rightarrow [\![H]\!]_H$
using *Healthy-Idempotent assms* **by** *blast*
from *assms*(*1*) *assms*(*3*)[*THEN sym*] **show** $ex\ x \in [\![H]\!]_H \rightarrow [\![ex\ x \circ H]\!]_H$
by (*simp add*: *Pi-iff Healthy-def fun-eq-iff exists-twice*)
fix $P\ Q$
assume P *is* $(ex\ x \circ H)$ Q *is* H

thus $(H\ P \sqsubseteq Q) = (P \sqsubseteq (\exists\ x \cdot Q))$
by (*metis* (*no-types*, *lifting*) *Healthy-Idempotent Healthy-if assms comp-apply dual-order.trans ex-weakens utp-pred-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

18 Concurrent Programming

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

In this theory we describe the UTP scheme for concurrency, *parallel-by-merge*, which provides a general parallel operator parametrised by a "merge predicate" that explains how to merge the after states of the composed predicates. It can thus be applied to many languages and concurrency schemes, with this theory providing a number of generic laws. The operator is explained in more detail in Chapter 7 of the UTP book [5].

18.1 Variable Renamings

In parallel-by-merge constructions, a merge predicate defines the behaviour following execution of of parallel processes, $P \parallel Q$, 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. The initial state-space before execution is $'\alpha$, the final state-space after the first parallel process is $'\beta_0$, and the final state-space for the second is $'\beta_1$. These three functions lift variables on these three state-spaces, respectively.

alphabet $('\alpha, '\beta_0, '\beta_1)$ *mrg* =
mrg-prior :: $'\alpha$
mrg-left :: $'\beta_0$
mrg-right :: $'\beta_1$

definition *pre-uvar* :: $('a \Longrightarrow '\alpha) \Rightarrow ('a \Longrightarrow ('\alpha, '\beta_0, '\beta_1)$ *mrg*$)$ **where**
[*upred-defs*]: *pre-uvar* $x = x\ ;_L$ *mrg-prior*

definition *left-uvar* :: $('a \Longrightarrow '\beta_0) \Rightarrow ('a \Longrightarrow ('\alpha, '\beta_0, '\beta_1)$ *mrg*$)$ **where**
[*upred-defs*]: *left-uvar* $x = x\ ;_L$ *mrg-left*

definition *right-uvar* :: $('a \Longrightarrow '\beta_1) \Rightarrow ('a \Longrightarrow ('\alpha, '\beta_0, '\beta_1)$ *mrg*$)$ **where**
[*upred-defs*]: *right-uvar* $x = x\ ;_L$ *mrg-right*

We set up syntax for the three variable classes using a subscript $<$, 0-x, and 1-x, respectively.

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

We proved behavedness closure properties about the lenses.

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*)

We prove various independence laws about the variable classes.

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

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

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

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

lemma *right-uvar-indep-right-uvar* [*simp*]:

$x \bowtie y \Longrightarrow$ *right-uvar* $x \bowtie$ *right-uvar* y
by (*simp add*: *right-uvar-def*)

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

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

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

18.2 Merge Predicates

A merge predicate is 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* = $(('\alpha,\ '\alpha,\ '\alpha)\ mrg,\ '\alpha)\ rel$

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

definition $skip_m$ :: $'\alpha$ *merge* **where**
[*upred-defs*]: $skip_m = (\$\Sigma\acute{} =_u \$\Sigma_<)$

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

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

A symmetric merge is one for which swapping the order of the merged concurrent predicates has no effect. We represent this by the following healthiness condition that states that $swap_m$ is a left-unit.

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

18.3 Separating Simulations

U0 and U1 are relations modify the variables of the input state-space such that they become indexed with 0 and 1, respectively.

definition *U0* :: $('\beta_0,\ ('\alpha,\ '\beta_0,\ '\beta_1)\ mrg)\ rel$ **where**
[*upred-defs*]: $U0 = (\$0{-}\Sigma\acute{} =_u \$\Sigma)$

definition *U1* :: $('\beta_1,\ ('\alpha,\ '\beta_0,\ '\beta_1)\ mrg)\ rel$ **where**
[*upred-defs*]: $U1 = (\$1{-}\Sigma\acute{} =_u \$\Sigma)$

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

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

As shown below, separating simulations can also be expressed using the following two alphabet extrusions

definition $U0\alpha$ **where** [*upred-defs*]: $U0\alpha = (1_L \times_L \textit{mrg-left})$

definition $U1\alpha$ **where** [*upred-defs*]: $U1\alpha = (1_L \times_L \textit{mrg-right})$

We then create the following intuitive syntax for separating simulations.

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** $\lceil P \rceil_1 \equiv P \oplus_p U1\alpha$

$\lceil P \rceil_0$ is predicate P where all variables are indexed by 0, and $\lceil P \rceil_1$ is where all variables are indexed by 1. We can thus equivalently express separating simulations using alphabet extrusion.

lemma *U0-as-alpha*: $(P ;; U0) = \lceil P \rceil_0$
  by (*rel-auto*)

lemma *U1-as-alpha*: $(P ;; U1) = \lceil P \rceil_1$
  by (*rel-auto*)

lemma *U0α-vwb-lens* [*simp*]: *vwb-lens* $U0\alpha$
  by (*simp add*: *U0α-def id-vwb-lens prod-vwb-lens*)

lemma *U1α-vwb-lens* [*simp*]: *vwb-lens* $U1\alpha$
  by (*simp add*: *U1α-def id-vwb-lens prod-vwb-lens*)

lemma *U0α-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: *U0α-def right-uvar-def out-var-def prod-as-plus lens-comp-assoc*[*THEN sym*])

lemma *U1α-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*]:
  $\sigma(\$0{-}x\acute{} \mapsto_s true) \dagger \lceil P \rceil_0 = \sigma \dagger \lceil P[\![true/\$x\acute{}]\!] \rceil_0$
  $\sigma(\$0{-}x\acute{} \mapsto_s false) \dagger \lceil P \rceil_0 = \sigma \dagger \lceil P[\![false/\$x\acute{}]\!] \rceil_0$
  by (*pred-auto+*)

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

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

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

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

lemma *U1-skip* [*alpha*]: $\lceil II \rceil_1 = (\$1{-}\Sigma\acute{} =_u \$\Sigma)$
  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*]: $(in\text{-}var\ x) ;_L U0\alpha = in\text{-}var\ x$
by (*simp add*: *U0α-def alpha-in-var in-var-prod-lens pre-uvar-def*)

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

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

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

18.4 Associative Merges

Associativity of a merge means that if we construct a three way merge from a two way merge and then rotate the three inputs of the merge to the left, then we get exactly the same three way merge back.

We first construct the operator that constructs the three way merge by effectively wiring up the two way merge in an appropriate way.

definition *ThreeWayMerge* :: $'\alpha\ merge \Rightarrow (('\alpha,\ '\alpha,\ ('\alpha,\ '\alpha,\ '\alpha)\ mrg)\ mrg,\ '\alpha)\ rel$ ($\mathbf{M}3'(\text{-}')$) **where**
[*upred-defs*]: $ThreeWayMerge\ M = ((\$0{-}\Sigma' =_u \$0{-}\Sigma \wedge \$1{-}\Sigma' =_u \$1{-}0{-}\Sigma \wedge \$\Sigma_<' =_u \$\Sigma_<) \;;\; M \;;\; U0 \wedge \$1{-}\Sigma' =_u \$1{-}1{-}\Sigma \wedge \$\Sigma_<' =_u \$\Sigma_<) \;;\; M$

The next definition rotates the inputs to a three way merge to the left one place.

abbreviation $rotate_m$ **where** $rotate_m \equiv (0{-}\Sigma,1{-}0{-}\Sigma,1{-}1{-}\Sigma) := (\&1{-}0{-}\Sigma,\&1{-}1{-}\Sigma,\&0{-}\Sigma)$

Finally, a merge is associative if rotating the inputs does not effect the output.

definition *AssocMerge* :: $'\alpha\ merge \Rightarrow bool$ **where**
[*upred-defs*]: $AssocMerge\ M = (rotate_m \;;\; \mathbf{M}3(M) = \mathbf{M}3(M))$

18.5 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* (**infixr** $\|_s$ *85*) **where**
$P \|_s Q \equiv (P \;;\; U0) \wedge (Q \;;\; U1) \wedge \$\Sigma_<' =_u \$\Sigma$

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

definition *par-by-merge* ($\text{-} \|_{\text{-}} \text{-}$ [*85,0,86*] *85*)
where [*upred-defs*]: $P \|_M Q = (P \|_s Q \;;\; M)$

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

lemma *shEx-pbm-left*: $((\exists\ x \cdot P\ x) \|_M Q) = (\exists\ x \cdot (P\ x \|_M Q))$

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*)

18.6 Unrestriction Laws

lemma *unrest-in-par-by-merge* [*unrest*]:
$[\![\ \$x \sharp P;\ \$x_< \sharp M;\ \$x \sharp Q\ ]\!] \Longrightarrow \$x \sharp P \parallel_M Q$
by (*rel-auto*, *fastforce+*)

lemma *unrest-out-par-by-merge* [*unrest*]:
$[\![\ \$x' \sharp M\ ]\!] \Longrightarrow \$x' \sharp P \parallel_M Q$
by (*rel-auto*)

18.7 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 coercions. 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 $x :: (\text{-} \Longrightarrow {}'\alpha)$
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 $x :: (\text{-} \Longrightarrow {}'\alpha)$
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 $x :: (\text{-} \Longrightarrow {}'\alpha)$
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 :: (- $\Longrightarrow$ $'\alpha$)
shows
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s\ numeral\ n) \dagger (P \parallel_M Q) = \sigma \dagger ((P[\![numeral\ n/\$x]\!]) \parallel_{M[\![numeral\ n/\$x_<]\!]} (Q[\![numeral\ n/\$x]\!]))$
$\bigwedge P\ Q\ M\ \sigma.\ \sigma(\$x\acute{} \mapsto_s\ numeral\ n) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M[\![numeral\ n/\$x\acute{}]\!]} Q)$
by (*rel-auto*)+

18.8 Parallel-by-merge laws

lemma *par-by-merge-false* [*simp*]:
$P \parallel_{false} Q = false$
by (*rel-auto*)

lemma *par-by-merge-left-false* [*simp*]:
$false \parallel_M Q = false$
by (*rel-auto*)

lemma *par-by-merge-right-false* [*simp*]:
$P \parallel_M false = false$
by (*rel-auto*)

lemma *par-by-merge-seq-add*: $(P \parallel_M Q)\ ;;\ R = (P \parallel_{M\ ;;\ R} Q)$
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 $P \parallel_{skip_m} Q = II$
using *assms* **by** (*rel-auto*)

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

lemma *par-sep-swap*: $P \parallel_s Q\ ;;\ swap_m = Q \parallel_s P$
by (*rel-auto*)

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

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

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

lemma *par-by-merge-mono-1*:
  assumes $P_1 \sqsubseteq P_2$
  shows $P_1 \parallel_M Q \sqsubseteq P_2 \parallel_M Q$
  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*)

theorem *par-by-merge-assoc*:
  assumes *M is SymMerge AssocMerge M*
  shows $(P \parallel_M Q) \parallel_M R = P \parallel_M (Q \parallel_M R)$
proof −
  have $(P \parallel_M Q) \parallel_M R = ((P \;;;\; U0) \wedge (Q \;;;\; U0 \;;;\; U1) \wedge (R \;;;\; U1 \;;;\; U1) \wedge \$\Sigma_<{}' =_u \$\Sigma) \;;;\; \mathbf{M}3(M)$
    by (*rel-blast*)
  also have ... $= ((P \;;;\; U0) \wedge (Q \;;;\; U0 \;;;\; U1) \wedge (R \;;;\; U1 \;;;\; U1) \wedge \$\Sigma_<{}' =_u \$\Sigma) \;;;\; rotate_m \;;;\; \mathbf{M}3(M)$
    using *AssocMerge-def assms*(*2*) **by** *force*
  also have ... $= ((Q \;;;\; U0) \wedge (R \;;;\; U0 \;;;\; U1) \wedge (P \;;;\; U1 \;;;\; U1) \wedge \$\Sigma_<{}' =_u \$\Sigma) \;;;\; \mathbf{M}3(M)$
    by (*rel-blast*)
  also have ... $= (Q \parallel_M R) \parallel_M P$
    by (*rel-blast*)
  also have ... $= P \parallel_M (Q \parallel_M R)$
    by (*simp add*: *assms*(*1*) *par-by-merge-commute*)
  finally show *?thesis* **.**
qed

theorem *par-by-merge-choice-left*:
  $(P \sqcap Q) \parallel_M R = (P \parallel_M R) \sqcap (Q \parallel_M R)$
  by (*rel-auto*)

theorem *par-by-merge-choice-right*:
  $P \parallel_M (Q \sqcap R) = (P \parallel_M Q) \sqcap (P \parallel_M R)$
  by (*rel-auto*)

theorem *par-by-merge-USUP-mem-left*:
  $(\bigsqcap\ i{\in}I \cdot P(i)) \parallel_M Q = (\bigsqcap\ i{\in}I \cdot P(i) \parallel_M Q)$
  by (*rel-auto*)

theorem *par-by-merge-USUP-ind-left*:
  $(\bigsqcap\ i \cdot P(i)) \parallel_M Q = (\bigsqcap\ i \cdot P(i) \parallel_M Q)$
  by (*rel-auto*)

theorem *par-by-merge-USUP-mem-right*:
  $P \parallel_M (\bigsqcap\ i{\in}I \cdot Q(i)) = (\bigsqcap\ i{\in}I \cdot P \parallel_M Q(i))$
  by (*rel-auto*)

theorem *par-by-merge-USUP-ind-right*:
  $P \parallel_M (\bigsqcap\ i \cdot Q(i)) = (\bigsqcap\ i \cdot P \parallel_M Q(i))$
  by (*rel-auto*)

18.9 Example: Simple State-Space Division

The following merge predicate divides the state space using a pair of independent lenses.

definition *StateMerge* :: $('a \Longrightarrow '\alpha) \Rightarrow ('b \Longrightarrow '\alpha) \Rightarrow '\alpha$ *merge* $(M[\text{-}|\text{-}]_\sigma)$ **where**
[*upred-defs*]: $M[a|b]_\sigma = (\$\Sigma\acute{} =_u (\$\Sigma_< \oplus \$0{-}\Sigma \textit{ on } \&a) \oplus \$1{-}\Sigma \textit{ on } \&b)$

lemma *swap-StateMerge*: $a \bowtie b \Longrightarrow (swap_m \;;;\; M[a|b]_\sigma) = M[b|a]_\sigma$
  by (*rel-auto*, *simp-all add*: *lens-indep-comm*)

abbreviation *StateParallel* :: $'\alpha$ *hrel* $\Rightarrow ('a \Longrightarrow '\alpha) \Rightarrow ('b \Longrightarrow '\alpha) \Rightarrow '\alpha$ *hrel* $\Rightarrow '\alpha$ *hrel* (- $|\text{-}|\text{-}|_\sigma$ - [*85*,*0*,*0*,*86*] *86*)
where $P\ |a|b|_\sigma\ Q \equiv P \parallel_{M[a|b]_\sigma} Q$

lemma *StateParallel-commute*: $a \bowtie b \Longrightarrow P\ |a|b|_\sigma\ Q = Q\ |b|a|_\sigma\ P$
  by (*metis par-by-merge-commute-swap swap-StateMerge*)

lemma *StateParallel-form*:
  $P\ |a|b|_\sigma\ Q = (\exists\ (st_0,\ st_1) \cdot P[\![\ll st_0\gg/\$\Sigma\acute{}]\!] \wedge Q[\![\ll st_1\gg/\$\Sigma\acute{}]\!] \wedge \$\Sigma\acute{} =_u (\$\Sigma \oplus \ll st_0\gg \textit{ on } \&a) \oplus \ll st_1\gg \textit{ on } \&b)$
  by (*rel-auto*)

lemma *StateParallel-skip*:
  assumes *vwb-lens a vwb-lens b* $a \bowtie b$
  shows $II\ |a|b|_\sigma\ P = b{:}[P]$
  using *assms* **by** (*rel-auto*)

end

19 Relational operational semantics

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

This theory uses the laws of relational calculus to create a basic operational semantics. It is based on Chapter 10 of the UTP book [5].

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 usubst*)

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

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

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

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

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

19.1 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 $::\ ('\mathcal{T},\ '\alpha)\ uthy \Rightarrow '\beta \Longrightarrow '\alpha\ (\mathbf{v}_1)$

pvar-assigns :: $('\mathcal{T}, '\alpha)$ *uthy* ⇒ $'\beta$ *usubst* ⇒ $'\alpha$ *hrel* (⟨-⟩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* ⇒ *svid* (**v**1)
-thy-asgn :: $('\mathcal{T}, '\alpha)$ *uthy* ⇒ *svids* ⇒ *uexprs* ⇒ *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* ⇒ $('a \Longrightarrow '\alpha)$ **where**
[*upred-defs*]: *top-var x* = (*list-lens 0* $;_L$ *x*)

The remainder of the local variable stack (the tail)

definition *rest-var* :: $('a::two, '\alpha)$ *lvar* ⇒ $('a\ list \Longrightarrow '\alpha)$ **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* :: $('\mathcal{T}, '\alpha)$ *uthy* ⇒ $('a, '\beta)$ *lvar* ⇒ $'\alpha$ *hrel* **where**
[*urel-defs*]: *var-begin T x* = *x* $::=_{T}$ ⟨«*undefined*»⟩ $\hat{}_u$ &*x*

definition *var-end* :: $('\mathcal{T}, '\alpha)$ *uthy* ⇒ $('a, '\beta)$ *lvar* ⇒ $'\alpha$ *hrel* **where**
[*urel-defs*]: *var-end T x* = (*x* $::=_{T}$ $tail_u$(&*x*))

var-begin takes as parameters a UTP theory and a local variable, and uses the theory assignment operator to push and undefined value onto the variable stack. *var-end* removes the top most variable from the stack in a similar way.

definition *var-vlet* :: $('\mathcal{T}, '\alpha)$ *uthy* $\Rightarrow$ $('a, '\alpha)$ *lvar* $\Rightarrow$ $'\alpha$ *hrel* **where**
[*urel-defs*]: *var-vlet* $T\ x = ((\$x \neq_u \langle\rangle) \wedge \mathcal{II}_T)$

Next we set up the typical UTP variable block syntax, though with a suitable subscript index to represent the UTP theory parameter.

syntax
  -var-begin :: *logic* ⇒ *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 $x \bowtie y$
shows $(var\ x\ ;;\ end\ y) = (end\ y\ ;;\ var\ x)$
by (*simp add*: *var-begin-def var-end-def shEx-lift-seq-1 shEx-lift-seq-2*
Healthy-pvar-assigns pvar-assigns-comp
assms usubst unrest lens-indep-sym, *simp add*: *assms usubst-upd-comm*)

The beginning and end of variable blocks from different variables commute.

lemma *var-block-vacuous*:
fixes x :: $('a::two, '\beta)$ *lvar*
assumes *vwb-lens* x
shows $(var\ x \cdot \mathcal{II}) = \mathcal{II}$
by (*simp add*: *Left-Unit assms var-end-healthy var-open-close*)

A variable block with a skip inside results in a skip.

end

Example instantiation for the theory of relations

overloading
rel-pvar == *pvar* :: $(REL, '\alpha)$ *uthy* $\Rightarrow$ $'\alpha \Longrightarrow '\alpha$
rel-pvar-assigns == *pvar-assigns* :: $(REL, '\alpha)$ *uthy* $\Rightarrow$ $'\alpha$ *usubst* $\Rightarrow$ $'\alpha$ *hrel*
begin
definition *rel-pvar* :: $(REL, '\alpha)$ *uthy* $\Rightarrow$ $'\alpha \Longrightarrow '\alpha$ **where**
[*upred-defs*]: *rel-pvar* $T = 1_L$
definition *rel-pvar-assigns* :: $(REL, '\alpha)$ *uthy* $\Rightarrow$ $'\alpha$ *usubst* $\Rightarrow$ $'\alpha$ *hrel* **where**
[*upred-defs*]: *rel-pvar-assigns* $T\ \sigma = \langle\sigma\rangle_a$
end

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

20 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-rel-laws
  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. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In *RAMiCS 2012*, volume 7560 of *LNCS*. Springer, September 2012.

[3] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.

[4] A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus: a process specification and verification environment. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, 2012.

[5] T. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[6] 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.

[7] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997.

[8] C. Morgan. *Programming from Specifications.* Prentice-Hall, London, UK, 1990.

[9] M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In *UTP 2006*, volume 4010 of *LNCS*, pages 123–140. Springer, 2007.

[10] 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.