Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming

Simon Foster[*], Frank Zeyda, Yakoub Nemouchi, Pedro Ribeiro, and Burkhart Wolff

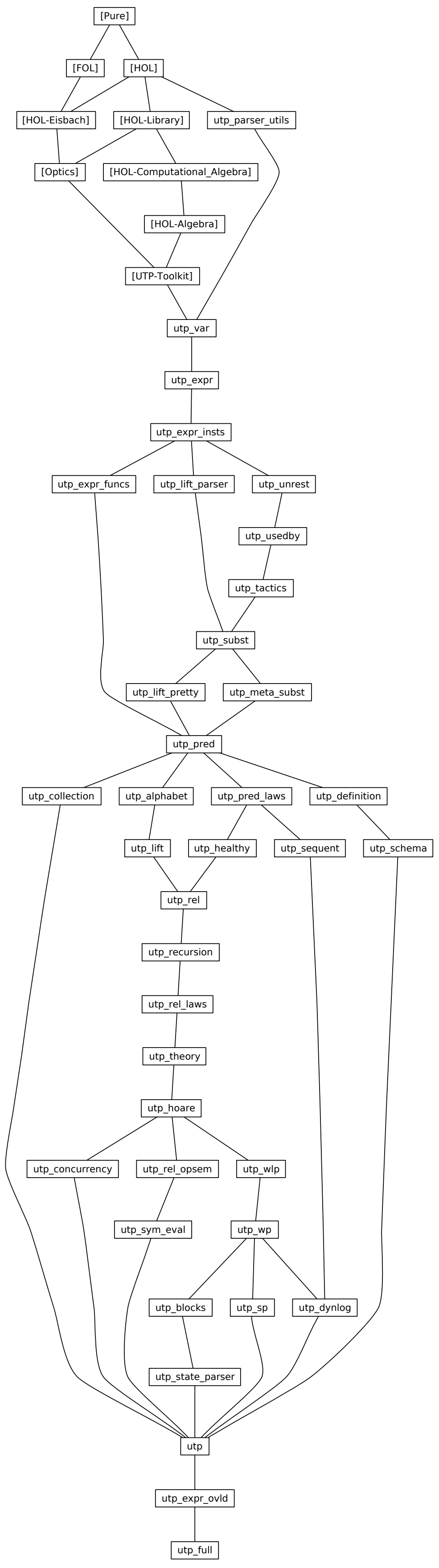
July 6, 2020

Abstract

Isabelle/UTP is a mechanised theory engineering toolkit based on Hoare and He's Unifying Theories of Programming (UTP). UTP enables the creation of denotational, algebraic, and operational semantics for different programming languages using an alphabetised relational calculus. We provide a semantic embedding of the alphabetised relational calculus in Isabelle/HOL, including new type definitions, relational constructors, automated proof tactics, and accompanying algebraic laws. Isabelle/UTP can be used to both capture laws of programming for different languages, and put these fundamental theorems to work in the creation of associated verification tools, using calculi like Hoare logics. This document describes the relational core of the UTP in Isabelle/HOL.

Contents

[*]Department of Computer Science, University of York. simon.foster@york.ac.uk


[Pure]
[FOL]
[HOL]
[HOL-Eisbach]
[HOL-Library]
utp_parser_utils
[Optics]
[HOL-Computational_Algebra]
[HOL-Algebra]
[UTP-Toolkit]
utp_var
utp_expr
utp_expr_insts
utp_expr_funcs
utp_lift_parser
utp_unrest
utp_usedby
utp_tactics
utp_subst
utp_lift_pretty
utp_meta_subst
utp_pred
utp_collection
utp_alphabet
utp_pred_laws
utp_definition
utp_lift
utp_healthy
utp_sequent
utp_schema
utp_rel
utp_recursion
utp_rel_laws
utp_theory
utp_hoare
utp_concurrency
utp_rel_opsem
utp_wlp
utp_sym_eval
utp_wp
utp_blocks
utp_sp
utp_dynlog
utp_state_parser
utp
utp_expr_ovld
utp_full

1 Introduction

This document contains the description of our mechanisation of Hoare and He's *Unifying Theories of Programming* [22, 7] (UTP) in Isabelle/HOL. UTP uses the "programs-as-predicates" approach, pioneered by Hehner [20, 18, 19], to encode denotational semantics and facilitate reasoning about programs. It uses the alphabetised relational calculus, which combines predicate calculus and relation algebra, to denote programs as relations between initial variables (x) and their subsequent values (x'). Isabelle/UTP[1] [16, 28, 15] semantically embeds this relational calculus into Isabelle/HOL, which enables application of the latter's proof facilities to program verification. For an introduction to UTP, we recommend two tutorials [6, 7], and also the UTP book [22].

The Isabelle/UTP core mechanises most of definitions and theorems from chapters 1, 2, 4, and 7 of [22], and some material contained in chapters 5 and 10. This essentially amounts to alphabetised predicate calculus, its core laws, the UTP theory infrastructure, and also parallel-by-merge [22, chapter 5], which adds concurrency primitives. The Isabelle/UTP core does not contain the theory of designs [6] and CSP [7], which are both represented in their own theory developments.

A large part of the mechanisation, however, is foundations that enable these core UTP theories. In particular, Isabelle/UTP builds on our implementation of lenses [16, 14], which gives a formal semantics to state spaces and variables. This, in turn, builds on a previous version of Isabelle/UTP [9, 10], which provided a shallow embedding of UTP by using Isabelle record types to represent alphabets. We follow this approach and, additionally, use the lens laws [11, 16] to characterise well-behaved variables. We also add meta-logical infrastructure for dealing with free variables and substitution. All this, we believe, adds an additional layer rigour to the UTP.

The alphabets-as-types approach does impose a number of theoretical limitations. For example, alphabets can only be extended when an injection into a larger state-space type can be exhibited. It is therefore not possible to arbitrarily augment an alphabet with additional variables, but new types must be created to do this. This is largely because as in previous work [9, 10], we actually encode state spaces rather than alphabets, the latter being implicit. Namely, a relation is typed by the state space type that it manipulates, and the alphabet is represented by collection of lenses into this state space. This aspect of our mechanisation is actually much closer to the relational program model in Back's refinement calculus [3].

The pay-off is that the Isabelle/HOL type checker can be directly applied to relational constructions, which makes proof much more automated and efficient. Moreover, our use of lenses mitigates the limitations by providing meta-logical style operators, such as equality on variables, and alphabet membership [16]. Isabelle/UTP can therefore directly harness proof automation from Isabelle/HOL, which allows its use in building efficient verification tools [13, 12]. For a detailed discussion of semantic embedding approaches, please see [28].

In addition to formalising variables, we also make a number of generalisations to UTP laws. Notably, our lens-based representation of state leads us to adopt Back's approach to both assignment and local variables [3]. Assignment becomes a point-free operator that acts on state-space update functions, which provides a rich set of algebraic theorems. Local variables are represented using stacks, unlike in the UTP book where they utilise alphabet extension.

[1] Isabelle/UTP website: https://www.cs.york.ac.uk/circus/isabelle-utp/

We give a summary of the main contributions within the Isabelle/UTP core, which can all be seen in the table of contents.

1. Formalisation of variables and state-spaces using lenses [16];
2. an expression model, together with lifted operators from HOL;
3. the meta-logical operators of unrestriction, used-by, substitution, alphabet extrusion, and alphabet restriction;
4. the alphabetised predicate calculus and associated algebraic laws;
5. the alphabetised relational calculus and associated algebraic laws;
6. proof tactics for the above based on interpretation [23];
7. a formalisation of UTP theories using locales [4] and building on HOL-Algebra [5];
8. Hoare logic [21] and dynamic logic [17];
9. weakest precondition and strongest postcondition calculi [8];
10. concurrent programming with parallel-by-merge;
11. relational operational semantics.

2 UTP Variables

```
theory utp-var
  imports
  UTP−Toolkit.utp-toolkit
  utp-parser-utils
begin
```

In this first UTP theory we set up variables, which are are built on lenses [11, 16]. 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 ⊑ı 50) and
  Lattice.sup (⨆ı- [90] 90) and
  Lattice.inf (⨅ı- [90] 90) and
  Lattice.join (infixl ⊔ı 65) and
  Lattice.meet (infixl ⊓ı 70) and
  Set.member (op :) and
  Set.member ((-/ : -) [51, 51] 50) and
  disj (infixr | 30) and
  conj (infixr & 35)

declare fst-vwb-lens [simp]
declare snd-vwb-lens [simp]
declare comp-vwb-lens [simp]
```

declare *lens-inv-bij* [*simp*]
declare *id-bij-lens* [*simp*]
declare *lens-indep-left-ext* [*simp*]
declare *lens-indep-right-ext* [*simp*]
declare *lens-comp-quotient* [*simp*]
declare *plus-lens-distr* [*THEN sym*, *simp*]
declare *lens-comp-assoc* [*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 [9, 10] 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* $x = x \;;_L \mathit{fst}_L$

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

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-weak-lens* [*simp*]:
  weak-lens $x \Longrightarrow$ *weak-lens* (*in-var* x)
  by (*simp add*: *comp-weak-lens in-var-def*)

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-weak-lens* [*simp*]:
  weak-lens $x \Longrightarrow$ *weak-lens* (*pr-var* x)
  by (*simp add*: *pr-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-weak-lens* [*simp*]:
weak-lens x $\Longrightarrow$ *weak-lens* (*out-var x*)
by (*simp add*: *comp-weak-lens out-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*)

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

lemma *pr-var-lens-plus* [*simp*]:
pr-var ($x +_L y$) = ($x +_L y$)
by (*simp add*: *pr-var-def*)

lemma *pr-var-lens-comp-1* [*simp*]:
pr-var x $;_L$ y = *pr-var* (x $;_L$ y)
by (*simp add*: *pr-var-def*)

lemma *pr-var-lens-comp-2* [*simp*]:
(x $;_L$ *pr-var* y) = *pr-var* (x $;_L$ y)
by (*simp-all add*: *pr-var-def*)

lemma *pr-var-len-quotient-1* [*simp*]:
pr-var x $/_L$ y = *pr-var* (x $/_L$ y)
by (*simp add*: *pr-var-def*)

lemma *pr-var-len-quotient-2* [*simp*]:
x $/_L$ *pr-var* y = *pr-var* (x $/_L$ y)
by (*simp add*: *pr-var-def*)

lemma *in-var-plus* [*simp*]: *in-var* ($x +_L y$) = *in-var* x $+_L$ *in-var* y
by (*simp add*: *in-var-def*)

lemma *out-var-plus* [*simp*]: *out-var* ($x +_L y$) = *out-var* x $+_L$ *out-var* y
by (*simp add*: *out-var-def*)

Similar properties follow for sublens

lemma *in-var-sublens* [*simp*]:
$y \subseteq_L x \Longrightarrow$ *in-var* y $\subseteq_L$ *in-var* x
by (*metis* (*no-types*, *hide-lams*) *in-var-def lens-comp-assoc sublens-def*)

lemma *out-var-sublens* [*simp*]:
$y \subseteq_L x \Longrightarrow$ *out-var* y $\subseteq_L$ *out-var* x
by (*metis* (*no-types*, *hide-lams*) *out-var-def lens-comp-assoc sublens-def*)

lemma *pr-var-sublens-l* [*simp*]: $a \subseteq_L b \Longrightarrow$ *pr-var* (a) $\subseteq_L b$
by (*simp add*: *pr-var-def*)

lemma *pr-var-sublens-r* [*simp*]: $a \subseteq_L b \Longrightarrow a \subseteq_L$ *pr-var* (b)
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*)

lemma *get-lens-plus* [*simp*]: $get_{x\ +_L\ y}\ s = (get_x\ s,\ get_y\ s)$
  by (*simp add*: *lens-defs*)

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-position* ⇒ *svid* (- [*999*] *999*)
  -svlongid :: *longid-position* ⇒ *svid* (- [*999*] *999*)
  -svid-unit :: *svid* ⇒ *svids* (-)
  -svid-list :: *svid* ⇒ *svids* ⇒ *svids* (-,/ -)
  -svid-alpha :: *svid* (**v**)
  -svid-dot :: *svid* ⇒ *svid* ⇒ *svid* (-:- [*999*,*998*] *998*)
  -svid-res :: *svid* ⇒ *svid* ⇒ *svid* (-↾- [*999*,*998*] *998*)
  -mk-svid-list :: *svids* ⇒ *logic* — Helper function for summing a list of identifiers
  -svid-view :: *logic* ⇒ *svid* ($\mathcal{V}$[-]) — View of a symmetric lens
  -svid-coview :: *logic* ⇒ *svid* ($\mathcal{C}$[-]) — Coview of a symmetric lens

A variable identifier can either be a HOL identifier, the complete set of variables in the alphabet **v**, 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* (&- [*990*] *990*)
  -sinvar :: *svid* ⇒ *svar* ($- [*990*] *990*)
  -soutvar :: *svid* ⇒ *svar* ($-´ [*990*] *990*)

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 :: *svid* ⇒ *salpha* (- [*990*] *990*)
  -salphavar :: *svar* ⇒ *salpha* (- [*990*] *990*)

-salphaparen :: *salpha* ⇒ *salpha* ('(-'))
-salphacomp :: *salpha* ⇒ *salpha* ⇒ *salpha* (**infixr** ; *75*)
-salphaprod :: *salpha* ⇒ *salpha* ⇒ *salpha* (**infixr** × *85*)
-salpha-all :: *salpha* (Σ)
-salpha-none :: *salpha* (∅)
-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* ({-}$_\alpha$)
-svid-set :: *svids* ⇒ *logic* ({-}$_v$)
-svid-empty :: *logic* ({}$_v$)
-svar :: *svar* ⇒ *logic* ('(-')$_v$)

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. Finally, we set up the translations rules.

translations
— Identifiers
-svid x ⇀ *x*
-svlongid x ⇀ *x*
-svid-alpha ⇌ Σ
-svid-dot x y ⇀ *y* ;$_L$ *x*
-svid-res x y ⇀ *x* /$_L$ *y*
-mk-svid-list (*-svid-unit x*) ⇀ *x*
-mk-svid-list (*-svid-list x xs*) ⇀ *x* +$_L$ *-mk-svid-list xs*
-svid-view a => $\mathcal{V}_a$
-svid-coview a => $\mathcal{C}_a$

— Decorations
-spvar Σ ↽ *CONST pr-var CONST id-lens*
-sinvar Σ ↽ *CONST in-var* 1_L
-soutvar Σ ↽ *CONST out-var* 1_L
-spvar (*-svid-dot x y*) ↽ *CONST pr-var* (*CONST lens-comp y x*)
-sinvar (*-svid-dot x y*) ↽ *CONST in-var* (*CONST lens-comp y x*)
-soutvar (*-svid-dot x y*) ↽ *CONST out-var* (*CONST lens-comp y x*)
-svid-dot x (*-svid-dot y z*) ↽ *-svid-dot x* (*CONST lens-comp z y*)

-spvar (*-svid-res x y*) ↽ *CONST pr-var* (*CONST lens-quotient x y*)
-sinvar (*-svid-res x y*) ↽ *CONST in-var* (*CONST lens-quotient x y*)
-soutvar (*-svid-res x y*) ↽ *CONST out-var* (*CONST lens-quotient x y*)

-spvar x ⇌ *CONST pr-var x*
-sinvar x ⇌ *CONST in-var x*
-soutvar x ⇌ *CONST out-var x*

— Alphabets

-salphaparen a $\rightharpoonup$ *a*
-salphaid x $\rightharpoonup$ *x*
-salphacomp x y $\rightharpoonup$ *x* $+_L$ *y*
-salphaprod a b $\rightleftharpoons$ *a* $\times_L$ *b*
-salphavar x $\rightharpoonup$ *x*
-svar-nil x $\rightharpoonup$ *x*
-svar-cons x xs $\rightharpoonup$ *x* $+_L$ *xs*
-salphaset A $\rightharpoonup$ *A*
(*-svar-cons x* (*-salphamk y*)) $\leftharpoondown$ *-salphamk* (*x* $+_L$ *y*)
x $\leftharpoondown$ *-salphamk x*
-salpha-all $\rightleftharpoons$ 1_L
-salpha-none $\rightleftharpoons$ 0_L

— Quotations
-ualpha-set A $\rightharpoonup$ *A*
-svid-set A $\rightharpoonup$ *-mk-svid-list A*
-svid-empty $\rightharpoonup$ 0_L
-svar x $\rightharpoonup$ *x*

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

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

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

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

end

3 UTP Expressions

theory *utp-expr*
imports
utp-var
begin

3.1 Expression type

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

Before building the predicate model, we will build a model of expressions that generalise alphabetised predicates. Expressions are represented semantically as mapping from the alphabet $'\alpha$ to the expression's type $'a$. This general model will allow us to unify all constructions under one type. The majority definitions in the file are given using the *lifting* package [23], which allows us to reuse much of the existing library of HOL functions.

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

setup-lifting *type-definition-uexpr*

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

nonterminal *uexprs*

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 *uexpr-defs* **and** *ueval* **and** *lit-simps* **and** *lit-norm*

3.2 Core expression constructs

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

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

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

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

The following operator is the general function application for expressions.

lift-definition *uexpr-appl* :: $('a \Rightarrow 'b, 's)$ *uexpr* $\Rightarrow ('a, 's)$ *uexpr* $\Rightarrow ('b, 's)$ *uexpr* (**infixl** |> *85*)
is $\lambda\ f\ x\ s.\ f\ s\ (x\ s)$ **.**

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.

abbreviation *uop* :: $('a \Rightarrow 'b) \Rightarrow ('a, '\alpha)$ *uexpr* $\Rightarrow ('b, '\alpha)$ *uexpr*
  where *uop f e* $\equiv \ll f\gg$ |> *e*

declare [[*coercion-map uop*]] — *uop* is useful as a coercion map

abbreviation *bop* ::
  $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, '\alpha)$ *uexpr* $\Rightarrow ('b, '\alpha)$ *uexpr* $\Rightarrow ('c, '\alpha)$ *uexpr*
  where *bop f u v* $\equiv \ll f\gg$ |> *u* |> *v*

abbreviation *trop* ::
  $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow ('a, '\alpha)$ *uexpr* $\Rightarrow ('b, '\alpha)$ *uexpr* $\Rightarrow ('c, '\alpha)$ *uexpr* $\Rightarrow ('d, '\alpha)$ *uexpr*
  where *trop f u v w* $\equiv \ll f\gg$ |> *u* |> *v* |> *w*

abbreviation *qtop* ::
  $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e) \Rightarrow$
   $('a, '\alpha)$ *uexpr* $\Rightarrow ('b, '\alpha)$ *uexpr* $\Rightarrow ('c, '\alpha)$ *uexpr* $\Rightarrow ('d, '\alpha)$ *uexpr* $\Rightarrow$
   $('e, '\alpha)$ *uexpr*

  where *qtop f u v w x* ≡ ≪*f*≫ |> *u* |> *v* |> *w* |> *x*

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 *uabs* :: (*'a* ⇒ (*'b*, *'α*) *uexpr*) ⇒ (*'a* ⇒ *'b*, *'α*) *uexpr*
is λ *f A x*. *f x A* **.**

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

definition *uIf* :: *bool* ⇒ *'a* ⇒ *'a* ⇒ *'a* **where**
[*uexpr-defs*]: *uIf* = *If*

abbreviation *cond* ::
  (*'a*,*'α*) *uexpr* ⇒ (*bool*, *'α*) *uexpr* ⇒ (*'a*,*'α*) *uexpr* ⇒ (*'a*,*'α*) *uexpr*
  ((*3*- ◁ - ▷/ -) [*52*,*0*,*53*] *52*)
where *P* ◁ *b* ▷ *Q* ≡ *trop uIf b P Q*

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

abbreviation (*input*) *eq-upred* :: (*'a*, *'α*) *uexpr* ⇒ (*'a*, *'α*) *uexpr* ⇒ (*bool*, *'α*) *uexpr* (**infixl** $=_u$ *50*)
where *eq-upred x y* ≡ *bop HOL.eq x y*

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

We also set up syntax for UTP variable expressions.

syntax
  -uuvar :: *svar* ⇒ *logic* (-)

translations
  -uuvar x == *CONST var x*

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

consts
  utrue :: *'a* (*true*)
  ufalse :: *'a* (*false*)

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* [*uexpr-defs*]: *0* = *lit 0*
instance ..
end

instantiation *uexpr* :: (*one*, *type*) *one*

begin
  definition *one-uexpr-def* [*uexpr-defs*]: $1 = lit\ 1$
instance ..

end

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

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

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*)

We can also define the order relation on expressions. Now, unlike the previous group and ring constructs, the order relations ($\leq$) and ($\leq$) 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 [*uexpr-defs*]: $less\text{-}uexpr\ P\ Q = (P \leq Q \wedge \neg\ Q \leq P)$
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

instantiation *uexpr* :: (*equal*, *enum*) *equal*
begin

definition *equal-uexpr* :: $('a, 'b)\ uexpr \Rightarrow ('a, 'b)\ uexpr \Rightarrow bool$ **where**
  $equal\text{-}uexpr\ f\ g \longleftrightarrow (\forall\, x \in set\ enum\text{-}class.enum.\ [\![f]\!]_e\ x = [\![g]\!]_e\ x)$

instance proof qed (*simp-all add*: *equal-uexpr-def uexpr-eq-iff enum-UNIV*)

end

instantiation *uexpr* :: (*enum*, *enum*) *enum*
begin
definition *enum-uexpr* :: ($'a$, $'b$) *uexpr list* **where**
enum-uexpr = *map* mk_e *enum-class.enum*
definition *enum-all-uexpr* :: (($'a$, $'b$) *uexpr* ⇒ *bool*) ⇒ *bool* **where**
enum-all-uexpr P = *enum-class.enum-all* (*P* ∘ mk_e)
definition *enum-ex-uexpr* :: (($'a$, $'b$) *uexpr* ⇒ *bool*) ⇒ *bool* **where**
enum-ex-uexpr P = *enum-class.enum-ex* (*P* ∘ mk_e)

instance
 by (*intro-classes*, *simp-all add*: *equal-uexpr-def enum-uexpr-def enum-all-uexpr-def enum-ex-uexpr-def*)
  (*transfer*, *simp add*: *UNIV-enum enum-distinct enum-all-UNIV comp-def*)+

end

3.4 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.

This operator allows us to get the characteristic set of a type. Essentially this is *UNIV*, but it retains the type syntactically for pretty printing.

definition *set-of* :: *'a itself* ⇒ *'a set* **where**
[*uexpr-defs*]: *set-of t* = *UNIV*

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

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

syntax — Core expression constructs
 -ucoerce :: *logic* ⇒ *type* ⇒ *logic* (**infix** $:_u$ *50*)
 -uabs :: *pttrn* ⇒ *logic* ⇒ *logic* (λ - • - [*0*, *10*] *10*)
 -ulens-ovrd :: *logic* ⇒ *logic* ⇒ *salpha* ⇒ *logic* (- ⊕ - *on* - [*85*, *0*, *86*] *86*)
 -ulens-get :: *logic* ⇒ *svar* ⇒ *logic* (-:- [*900*,*901*] *901*)

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

abbreviation (*input*) *umem* (**infix** $\in_u$ *50*) **where** (*x* $\in_u$ *A*) ≡ *bop* (∈) *x A*
abbreviation (*input*) *uNone* ($None_u$) **where** $None_u$ ≡ ≪*None*≫
abbreviation (*input*) *uSome* ($Some_u$'(-')) **where** $Some_u$(*e*) ≡ *uop Some e*
abbreviation (*input*) *uthe* (the_u'(-')) **where** the_u(*e*) ≡ *uop the e*

syntax — Tuples
 -utuple :: (*'a*, *'α*) *uexpr* ⇒ *utuple-args* ⇒ (*'a* ∗ *'b*, *'α*) *uexpr* ((*1*'(-,/ -')$_u$))
 -utuple-arg :: (*'a*, *'α*) *uexpr* ⇒ *utuple-args* (-)
 -utuple-args :: (*'a*, *'α*) *uexpr* => *utuple-args* ⇒ *utuple-args*   (-,/ -)

translations
  $(x,\ y)_u$ => *CONST bop* (*CONST Pair*) *x y*
  -utuple x (*-utuple-args y z*) => *-utuple x* (*-utuple-arg* (*-utuple y z*))

abbreviation (*input*) *uunit* ($'('')_u$) **where** $()_u \equiv \ll()\gg$
abbreviation (*input*) *ufst* ($\pi_1{}'(-')$) **where** $\pi_1(x) \equiv$ *uop fst x*
abbreviation (*input*) *usnd* ($\pi_2{}'(-')$) **where** $\pi_2(x) \equiv$ *uop snd x*

— Orders

abbreviation (*input*) *uless* (**infix** $<_u$ *50*) **where** $x <_u y \equiv$ *bop* ($<$) *x y*
abbreviation (*input*) *ugreat* (**infix** $>_u$ *50*) **where** $x >_u y \equiv y <_u x$
abbreviation (*input*) *uleq* (**infix** $\leq_u$ *50*) **where** $x \leq_u y \equiv$ *bop* ($\leq$) *x y*
abbreviation (*input*) *ugeq* (**infix** $\geq_u$ *50*) **where** $x \geq_u y \equiv y \leq_u x$

3.5 Evaluation laws for expressions

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

3.6 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 *uexpr-fst* [*simp*]: $\pi_1((e,\ f)_u) = e$
  by (*transfer*, *simp*)

lemma *uexpr-snd* [*simp*]: $\pi_2((e,\ f)_u) = f$
  by (*transfer*, *simp*)

3.7 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-fun-simps* [*lit-simps*]:
  ≪i x y z u≫ = qtop i ≪x≫ ≪y≫ ≪z≫ ≪u≫
  ≪h x y z≫ = trop h ≪x≫ ≪y≫ ≪z≫
  ≪g x y≫ = bop g ≪x≫ ≪y≫
  ≪f x≫ = uop f ≪x≫
  by (*transfer*, *simp*)+

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* [*ueval*]: $[\![$*numeral x*$]\!]_e$ *b* = *numeral x*
  apply (*induct x*)
    apply (*simp add*: *lit.rep-eq one-uexpr-def*)
  apply (*simp add*: *ueval numeral-Bit0 plus-uexpr-def*)
  apply (*simp add*: *ueval numeral-Bit1 plus-uexpr-def one-uexpr-def*)
  done

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

lemma *lit-zero* [*lit-simps*]: *≪0≫ = 0* **by** (*simp add*:*uexpr-defs*)
lemma *lit-one* [*lit-simps*]: *≪1≫ = 1* **by** (*simp add*: *uexpr-defs*)
lemma *lit-plus* [*lit-simps*]: *≪x + y≫ = ≪x≫ + ≪y≫* **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)
lemma *lit-numeral* [*lit-simps*]: *≪numeral n≫ = numeral n* **by** (*simp add*: *numeral-uexpr-simp*)

In general unliteralising converts function applications to corresponding expression liftings. Since some operators, like + and *, have specific operators we also have to use *uIf = If*
0 = ≪0::?'a≫
1 = ≪1::?'a≫
?u + ?v = bop (+) *?u ?v*
(*?P < ?Q*) = (*?P ≤ ?Q ∧ ¬ ?Q ≤ ?P*)
set-of ?t = UNIV 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-uexpr* (*λb. numeral* ($[\![x]\!]_e$ *b*))
  by (*simp add*: *uexpr-appl-def lit.rep-eq*)

lemma *lit-numeral-2*: *Abs-uexpr* (*λ b. numeral v*) = *numeral v*
  by (*metis lit.abs-eq lit-numeral*)

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

The following tactic can be used to evaluate literal expressions. It first literalises UTP expressions, that is pushes as many operators into literals as possible. Then it tries to simplify, and final unliteralises at the end.

method *uexpr-simp* **uses** *simps* = ((*literalise*)*?*, *simp add*: *lit-norm simps*, (*unliteralise*)*?*)

lemma (*1*::(*int*, *'α*) *uexpr*) + *≪2≫ = 4 ⟷ ≪3≫ = 4*
  apply (*literalise*)
  apply (*uexpr-simp*) **oops**

end

4 Expression Type Class Instantiations

theory *utp-expr-insts*
  imports *utp-expr*
begin

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* [*uexpr-defs*]: $- u = uop\ uminus\ u$
instance ..
end

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

instantiation *uexpr* :: (*times*, *type*) *times*
begin
  definition *times-uexpr-def* [*uexpr-defs*]: $u * v = bop\ times\ u\ v$
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**
  [*uexpr-defs*]: *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 [*uexpr-defs*]: *inverse-uexpr u* = *uop inverse u*
instance ..
end

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

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

end

instantiation *uexpr* :: (*abs*, *type*) *abs*
begin
 definition *abs-uexpr-def* [*uexpr-defs*]: *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* :: (*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 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 next theorem lifts powers.

lemma *power-rep-eq* [*ueval*]: $[\![P\ \hat{}\ n]\!]_e = (\lambda\ b.\ [\![P]\!]_e\ b\ \hat{}\ n)$
by (*induct n*, *simp-all add*: *lit.rep-eq one-uexpr-def times-uexpr-def fun-eq-iff uexpr-appl.rep-eq*)

lemma *of-nat-uexpr-rep-eq* [*ueval*]: $[\![\mathit{of\text{-}nat}\ x]\!]_e\ b = \mathit{of\text{-}nat}\ x$
by (*induct x*, *simp-all add*: *uexpr-defs ueval*)

lemma *lit-uminus* [*lit-simps*]: $\ll- x\gg\ = -\ \ll x\gg$ **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)
lemma *lit-minus* [*lit-simps*]: $\ll x - y\gg\ =\ \ll x\gg - \ll y\gg$ **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)
lemma *lit-times* [*lit-simps*]: $\ll x * y\gg\ =\ \ll x\gg * \ll y\gg$ **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)
lemma *lit-divide* [*lit-simps*]: $\ll x\ /\ y\gg\ =\ \ll x\gg\ /\ \ll y\gg$ **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)
lemma *lit-div* [*lit-simps*]: $\ll x\ \mathit{div}\ y\gg\ =\ \ll x\gg\ \mathit{div}\ \ll y\gg$ **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)
lemma *lit-power* [*lit-simps*]: $\ll x\ \hat{}\ n\gg\ =\ \ll x\gg\ \hat{}\ n$ **by** (*simp add*: *lit.rep-eq power-rep-eq uexpr-eq-iff*)

4.1 Expression construction from HOL terms

Sometimes it is convenient to cast HOL terms to UTP expressions, and these simplifications automate this process.

named-theorems *mkuexpr*

lemma *mkuexpr-lens-get* [*mkuexpr*]: $mk_e\ get_x = \&x$
by (*transfer*, *simp add*: *pr-var-def*)

lemma *mkuexpr-zero* [*mkuexpr*]: $mk_e\ (\lambda\ s.\ 0) = 0$
by (*simp add*: *zero-uexpr-def*, *transfer*, *simp*)

lemma *mkuexpr-one* [*mkuexpr*]: $mk_e\ (\lambda\ s.\ 1) = 1$
by (*simp add*: *one-uexpr-def*, *transfer*, *simp*)

lemma *mkuexpr-numeral* [*mkuexpr*]: $mk_e\ (\lambda\ s.\ \mathit{numeral}\ n) = \mathit{numeral}\ n$
using *lit-numeral-2* **by** *blast*

lemma *mkuexpr-lit* [*mkuexpr*]: $mk_e\ (\lambda\ s.\ k) = \ll k\gg$
by (*transfer*, *simp*)

lemma *mkuexpr-pair* [*mkuexpr*]: $mk_e\ (\lambda s.\ (f\ s,\ g\ s)) = (mk_e\ f,\ mk_e\ g)_u$
by (*transfer*, *simp*)

lemma *mkuexpr-plus* [*mkuexpr*]: $mk_e\ (\lambda\ s.\ f\ s + g\ s) = mk_e\ f + mk_e\ g$
by (*simp add*: *plus-uexpr-def*, *transfer*, *simp*)

lemma *mkuexpr-uminus* [*mkuexpr*]: $mk_e\ (\lambda\ s.\ -\ f\ s) = -\ mk_e\ f$
by (*simp add*: *uminus-uexpr-def*, *transfer*, *simp*)

lemma *mkuexpr-minus* [*mkuexpr*]: $mk_e\ (\lambda\ s.\ f\ s - g\ s) = mk_e\ f - mk_e\ g$
by (*simp add*: *minus-uexpr-def*, *transfer*, *simp*)

lemma *mkuexpr-times* [*mkuexpr*]: $mk_e\ (\lambda\ s.\ f\ s * g\ s) = mk_e\ f * mk_e\ g$
by (*simp add*: *times-uexpr-def*, *transfer*, *simp*)

lemma *mkuexpr-divide* [*mkuexpr*]: $mk_e\ (\lambda\ s.\ f\ s\ /\ g\ s) = mk_e\ f\ /\ mk_e\ g$
by (*simp add*: *divide-uexpr-def*, *transfer*, *simp*)

end
theory *utp-expr-funcs*
  imports *utp-expr-insts*
begin

— Polymorphic constructs

abbreviation (*input*) *uceil* $(\lceil\text{-}\rceil_u)$ **where** $\lceil x\rceil_u \equiv$ *uop ceiling x*
abbreviation (*input*) *ufloor* $(\lfloor\text{-}\rfloor_u)$ **where** $\lfloor x\rfloor_u \equiv$ *uop floor x*
abbreviation (*input*) *umin* $(min_u{'}(\text{-},\ \text{-}{'}))$ **where** $min_u(x,\ y) \equiv$ *bop min x y*
abbreviation (*input*) *umax* $(max_u{'}(\text{-},\ \text{-}{'}))$ **where** $max_u(x,\ y) \equiv$ *bop max x y*
abbreviation (*input*) *ugcd* $(gcd_u{'}(\text{-},\ \text{-}{'}))$ **where** $gcd_u(x,\ y) \equiv$ *bop gcd x y*

— Lists / Sequences

abbreviation (*input*) *ucons* (**infixr** $\#_u$ *65*) **where** $x\ \#_u\ xs \equiv$ *bop* $(\#)$ *x xs*
abbreviation (*input*) *uappend* (**infixr** $\hat{}_u$ *80*) **where** $x\ \hat{}_u\ y \equiv$ *bop* $(@)$ *x y*
abbreviation (*input*) *udconcat* (**infixr** $\frown_u$ *90*) **where** $x \frown_u y \equiv$ *bop* $(\frown)$ *x y*
abbreviation (*input*) *ulast* $(last_u{'}(\text{-}{'}))$ **where** $last_u(x) \equiv$ *uop last x*
abbreviation (*input*) *ufront* $(front_u{'}(\text{-}{'}))$ **where** $front_u(x) \equiv$ *uop butlast x*
abbreviation (*input*) *uhead* $(head_u{'}(\text{-}{'}))$ **where** $head_u(x) \equiv$ *uop hd x*
abbreviation (*input*) *utail* $(tail_u{'}(\text{-}{'}))$ **where** $tail_u(x) \equiv$ *uop tl x*
abbreviation (*input*) *utake* $(take_u{'}(\text{-},/\ \text{-}{'}))$ **where** $take_u(n,\ xs) \equiv$ *bop take n xs*
abbreviation (*input*) *udrop* $(drop_u{'}(\text{-},/\ \text{-}{'}))$ **where** $drop_u(n,\ xs) \equiv$ *bop drop n xs*
abbreviation (*input*) *ufilter* (**infixl** $\upharpoonleft_u$ *75*) **where** $xs \upharpoonleft_u A \equiv$ *bop seq-filter xs A*
abbreviation (*input*) *uextract* (**infixl** $\upharpoonright_u$ *75*) **where** $xs \upharpoonright_u A \equiv$ *bop* $(\upharpoonright_l)$ *A xs*
abbreviation (*input*) *uelems* $(elems_u{'}(\text{-}{'}))$ **where** $elems_u(xs) \equiv$ *uop set xs*
abbreviation (*input*) *usorted* $(sorted_u{'}(\text{-}{'}))$ **where** $sorted_u(xs) \equiv$ *uop sorted xs*
abbreviation (*input*) *udistinct* $(distinct_u{'}(\text{-}{'}))$ **where** $distinct_u(xs) \equiv$ *uop set xs*
abbreviation (*input*) *uupto* $(\langle\text{-..-}\rangle)$ **where** $\langle n..k\rangle \equiv$ *bop upto n k*
abbreviation (*input*) *uupt* $(\langle\text{-..<-}\rangle)$ **where** $\langle n..{<}k\rangle \equiv$ *bop upt n k*
abbreviation (*input*) *umap* (map_u) **where** $map_u \equiv$ *bop map*
abbreviation (*input*) *uzip* (zip_u) **where** $zip_u \equiv$ *bop zip*

abbreviation (*input*) *ufinite* $(finite_u{'}(\text{-}{'}))$ **where** $finite_u(x) \equiv$ *uop finite x*
abbreviation (*input*) *uempset* $(\{\}_u)$ **where** $\{\}_u \equiv \ll\{\}\gg$
abbreviation (*input*) *uunion* (**infixl** $\cup_u$ *65*) **where** $A \cup_u B \equiv$ *bop* $(\cup)$ *A B*
abbreviation (*input*) *uinter* (**infixl** $\cap_u$ *70*) **where** $A \cap_u B \equiv$ *bop* $(\cap)$ *A B*
abbreviation (*input*) *uimage* $(\text{-}(\!|\text{-}|\!)_u\ [10,0]\ 10)$ **where** $f(\!|A|\!)_u \equiv$ *bop image f A*
abbreviation (*input*) *uinsert* $(insert_u)$ **where** $insert_u\ x\ xs \equiv$ *bop insert x xs*
abbreviation (*input*) *usubset* (**infix** $\subset_u$ *50*) **where** $A \subset_u B \equiv$ *bop* $(\subset)$ *A B*
abbreviation (*input*) *usubseteq* (**infix** $\subseteq_u$ *50*) **where** $A \subseteq_u B \equiv$ *bop* $(\subseteq)$ *A B*
abbreviation (*input*) *uconverse* $((\text{-}^{\sim})\ [1000]\ 999)$ **where** $P^{\sim} \equiv$ *uop converse P*

syntax — Sets
  -uset :: *args* => $({'}a\ set,\ {'}\alpha)$ *uexpr* $(\{(\text{-})\}_u)$
  -ucarrier :: *type* $\Rightarrow$ *logic* $([\text{-}]_T)$
  -uid :: *type* $\Rightarrow$ *logic* $(id[\text{-}])$
  -uproduct :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infixr** $\times_u$ *80*)
  -urelcomp :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infixr** $;_u$ *75*)

translations
  $\{x,\ xs\}_u$ => $insert_u\ x\ \{xs\}_u$
  $\{x\}_u$ => $insert_u\ x \ll\{\}\gg$

$['a]_T$ == $\ll$*CONST set-of TYPE*$('a)\gg$
id$['a]$ == $\ll$*CONST Id-on* (*CONST set-of TYPE*$('a)$)$\gg$
$A \times_u B$ == *CONST bop CONST Product-Type.Times A B*
$A\ ;_u\ B$ == *CONST bop CONST relcomp A B*

— Sum types

abbreviation (*input*) *uinl* (inl_u'(-')) **where** $inl_u(x) \equiv$ *uop Inl x*
abbreviation (*input*) *uinr* (inr_u'(-')) **where** $inr_u(x) \equiv$ *uop Inr x*

4.2 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* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ((*1*{- :/ - |/ - •/ -}))
-uset-compr-nset :: *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ((*1*{- |/ - •/ -}))
-uset-compr-nfun :: *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ((*1*{- :/ - |/ -}))
-uset-compr-nset-nfun :: *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ((*1*{- |/ -}))
-uset-compr-nvar :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ((*1*{- •/ -}))

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

abbreviation *ZedImage* ::
$(bool \times 'b, '\alpha)$ *uexpr* $\Rightarrow$ $('b$ *set*, $'\alpha)$ *uexpr* **where**
ZedImage PF $\equiv$ *ZedSetCompr* $\ll$*UNIV*$\gg$ (λ *x*::*unit*. *PF*)

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

4.3 Lifting limits

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

definition *ulim-left* :: *'a::order-topology* $\Rightarrow$ $('a \Rightarrow 'b)$ $\Rightarrow$ *'b::t2-space* **where**
[*uexpr-defs*]: *ulim-left* = (λ *p f*. *Lim* (*at-left p*) *f*)

definition *ulim-right* :: *'a::order-topology* $\Rightarrow$ $('a \Rightarrow 'b)$ $\Rightarrow$ *'b::t2-space* **where**
[*uexpr-defs*]: *ulim-right* = (λ *p f*. *Lim* (*at-right p*) *f*)

definition *ucont-on* :: (*'a::topological-space* $\Rightarrow$ *'b::topological-space*) $\Rightarrow$ *'a set* $\Rightarrow$ *bool* **where**
[*uexpr-defs*]: *ucont-on* = (λ *f A*. *continuous-on A f*)

syntax

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

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

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

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

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

lemma *uconcat-units* [*simp*]: $\ll[]\gg \;\hat{}_u\; xs = xs$ $xs \;\hat{}_u \ll[]\gg = xs$
by (*transfer*, *simp*)+

end

5 Unrestriction

theory *utp-unrest*
imports *utp-expr-insts*
begin

5.1 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 [27, 26] 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 [9] and Oliveira's [26] 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
  $\textit{-unrest} :: salpha \Rightarrow logic \Rightarrow logic \Rightarrow logic$ (**infix** $\sharp$ *20*)

translations
  $\textit{-unrest}\ x\ p == CONST\ unrest\ x\ p$
  $\textit{-unrest}\ (\textit{-salphaset}\ (\textit{-salphamk}\ (x +_L y)))\ P\ <= \textit{-unrest}\ (x +_L y)\ P$

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

adhoc-overloading
  unrest unrest-uexpr

lemma *unrest-expr-alt-def*:
  $\textit{weak-lens}\ x \Longrightarrow (x \sharp P) = (\forall\ b\ b'.\ [\![P]\!]_e\ (b \oplus_L b'\ on\ x) = [\![P]\!]_e\ b)$
  by (*transfer*, *metis lens-override-def weak-lens.put-get*)

5.2 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*)

lemma *unrest-svar* [*unrest*]: $(\&x \sharp P) \longleftrightarrow (x \sharp P)$
  by (*transfer*, *simp add*: *lens-defs*)

lemma *unrest-lens-comp* [*unrest*]: $x \sharp e \Longrightarrow x{:}y \sharp e$
  by (*simp add*: *lens-comp-def unrest-uexpr.rep-eq*)

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 one lens is smaller than another, then any unrestriction on the larger lens implies unrestriction on the smaller.

lemma *unrest-sublens*:
fixes P :: ($'a$, $'\alpha$) *uexpr*
assumes $x \sharp P$ $y \subseteq_L x$
shows $y \sharp P$
using *assms*
by (*transfer*, *metis* (*no-types*, *lifting*) *lens.select-convs*(*2*) *lens-comp-def sublens-def*)

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

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

If we can show that an expression is unrestricted on a bijective lens, then is unrestricted on the entire state-space.

lemma *bij-lens-unrest-all*:
fixes P :: ($'a$, $'\alpha$) *uexpr*
assumes *bij-lens* X $X \sharp P$
shows $\Sigma \sharp P$
using *assms bij-lens-equiv-id lens-equiv-def unrest-sublens* **by** *blast*

lemma *bij-lens-unrest-all-eq*:
fixes P :: ($'a$, $'\alpha$) *uexpr*
assumes *bij-lens* X
shows $(\Sigma \sharp P) \longleftrightarrow (X \sharp P)$
by (*meson assms bij-lens-equiv-id lens-equiv-def unrest-sublens*)

If an expression is unrestricted by all variables, then it is unrestricted by any variable

lemma *unrest-all-var*:
fixes e :: ($'a$, $'\alpha$) *uexpr*
assumes $\Sigma \sharp e$
shows $x \sharp e$
by (*metis assms id-lens-def lens.simps*(*2*) *unrest-uexpr.rep-eq*)

We can split an unrestriction composed by lens plus

lemma *unrest-plus-split*:
fixes P :: ($'a$, $'\alpha$) *uexpr*
assumes $x \bowtie y$ *vwb-lens x vwb-lens y*
shows *unrest* $(x +_L y)$ $P \longleftrightarrow (x \sharp P) \wedge (y \sharp P)$
using *assms*
by (*meson lens-plus-right-sublens lens-plus-ub sublens-refl unrest-sublens unrest-var-comp vwb-lens-wb*)

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*]: $\llbracket$ *mwb-lens x*; $x \bowtie y$ $\rrbracket \Longrightarrow \$y \sharp \$x$
by (*simp add*: *unrest-var*)

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

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

lemma *unrest-iuvar-ouvar* [*unrest*]:
fixes $x :: ('a \Longrightarrow '\alpha)$
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 $x :: ('a \Longrightarrow '\alpha)$
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-appl* [*unrest*]: $\llbracket x \sharp f; x \sharp v \rrbracket \Longrightarrow x \sharp f \mid> v$
by (*transfer*, *simp*)

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

lemma *unrest-bop* [*unrest*]: $\llbracket x \sharp u; x \sharp v \rrbracket \Longrightarrow x \sharp$ *bop f u v*
by (*simp add*: *unrest*)

lemma *unrest-trop* [*unrest*]: $\llbracket x \sharp u; x \sharp v; x \sharp w \rrbracket \Longrightarrow x \sharp$ *trop f u v w*
by (*simp add*: *unrest*)

lemma *unrest-qtop* [*unrest*]: $\llbracket x \sharp u; x \sharp v; x \sharp w; x \sharp y \rrbracket \Longrightarrow x \sharp$ *qtop f u v w y*
by (*simp add*: *unrest*)

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

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

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

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

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

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

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

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

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

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

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

lemma *unrest-case-prod* [*unrest*]: $[\![\ \bigwedge i\ j.\ x \sharp P\ i\ j\ ]\!] \Longrightarrow x \sharp$ *case-prod P v*
  by (*simp add*: *prod.split-sel-asm*)

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

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

end

6 Used-by

theory *utp-usedby*
  imports *utp-unrest*
begin

The used-by predicate is the dual of unrestriction. It states that the given lens is an upper-bound on the size of state space the given expression depends on. It is similar to stating that the lens is a valid alphabet for the predicate. For convenience, and because the predicate uses a similar form, we will reuse much of unrestriction's infrastructure.

consts
  usedBy :: $'a \Rightarrow 'b \Rightarrow$ *bool*

syntax
  -usedBy :: *salpha* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (**infix** $\natural$ *20*)

translations
  -usedBy x p == *CONST usedBy x p*
  -usedBy (*-salphaset* (*-salphamk* $(x +_L y)$)) P <= *-usedBy* $(x +_L y)$ P

lift-definition *usedBy-uexpr* :: $('b \Longrightarrow '\alpha) \Rightarrow ('a, '\alpha)$ *uexpr* $\Rightarrow$ *bool*
is $\lambda\ x\ e.\ (\forall\ b\ b'.\ e\ (b' \oplus_L b\ on\ x) = e\ b)$ **.**

adhoc-overloading *usedBy usedBy-uexpr*

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

lemma *usedBy-sublens*:
  fixes $P :: ('a, '\alpha)$ *uexpr*
  assumes $x \natural P$ $x \subseteq_L y$ *vwb-lens* y
  shows $y \natural P$
  using *assms*
  by (*transfer*, *auto*, *metis Lens-Order.lens-override-idem lens-override-def sublens-obs-get vwb-lens-mwb*)

lemma *usedBy-svar* [*unrest*]: $x \natural P \Longrightarrow \&x \natural P$
  by (*transfer*, *simp add*: *lens-defs*)

lemma *usedBy-lens-plus-1* [*unrest*]: $x \natural P \Longrightarrow x;y \natural P$
  by (*transfer*, *simp add*: *lens-defs*)

lemma *usedBy-lens-plus-2* [*unrest*]: $\llbracket\ x \bowtie y;\ y \natural P\ \rrbracket \Longrightarrow x;y \natural P$
  by (*transfer*, *auto simp add*: *lens-defs lens-indep-comm*)

Linking used-by to unrestriction: if x is used-by P, and x is independent of y, then P cannot depend on any variable in y.

lemma *usedBy-indep-uses*:
  fixes $P :: ('a, '\alpha)$ *uexpr*
  assumes $x \natural P$ $x \bowtie y$
  shows $y \sharp P$
  using *assms* **by** (*transfer*, *auto*, *metis lens-indep-get lens-override-def*)

Linking used-by and unrestriction via symmetric lenses.

lemma *psym-lens-unrest*: $\llbracket$ *psym-lens* a; $\mathcal{C}[a] \natural e\ \rrbracket \Longrightarrow \mathcal{V}[a] \sharp e$
  by (*transfer*, *simp add*: *lens-defs*, *metis lens-indep-def psym-lens-def*)

lemma *sym-lens-unrest*: $\llbracket$ *sym-lens* $a\ \rrbracket \Longrightarrow (\mathcal{V}[a] \sharp e) \longleftrightarrow (\mathcal{C}[a] \natural e)$
  by (*auto simp add*: *psym-lens-unrest*) (*transfer*, *simp add*: *lens-defs*, *metis sym-lens.put-region-coregion-cover*)

lemma *sym-lens-unrest'*: $\llbracket$ *sym-lens* $a\ \rrbracket \Longrightarrow (\mathcal{V}[a] \natural e) \longleftrightarrow (\mathcal{C}[a] \sharp e)$
  using *sym-lens-compl sym-lens-unrest* **by** *fastforce*

lemma *usedBy-var* [*unrest*]:
  assumes *vwb-lens* x $y \subseteq_L x$
  shows $x \natural$ *var* y
  using *assms*
  by (*transfer*, *simp add*: *uexpr-defs pr-var-def*)
    (*metis lens-override-def sublens-obs-get vwb-lens-def wb-lens.get-put*)

lemma *usedBy-appl* [*unrest*]: $\llbracket\ x \natural f;\ x \natural v\ \rrbracket \Longrightarrow x \natural f\ |{>}\ v$
  by (*transfer*, *simp*)

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

lemma *usedBy-bop* [*unrest*]: $\llbracket\ x \natural u;\ x \natural v\ \rrbracket \Longrightarrow x \natural$ *bop* f u v
  by (*transfer*, *simp*)

lemma *usedBy-trop* [*unrest*]: $\llbracket\ x \natural u;\ x \natural v;\ x \natural w\ \rrbracket \Longrightarrow x \natural$ *trop* f u v w
  by (*transfer*, *simp*)

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *unrest-var-sep* [*unrest*]:
 vwb-lens $x \Longrightarrow x \natural \&x{:}y$
 by (*transfer*, *simp add*: *lens-defs*)

end

7 UTP Tactics

```
theory utp-tactics
  imports
    utp-expr utp-unrest utp-usedby
keywords update-uexpr-rep-eq-thms :: thy-decl
begin
```

```
declare image-comp [simp]
```

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.

7.1 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
```

7.2 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)
```

7.3 Transfer Tactics

Next, we define the component tactics used for transfer.

7.3.1 Robust Transfer

Robust transfer uses the transfer method of the lifting package.

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

7.3.2 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*]
  usedBy-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*)

7.4 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*)?

7.5 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 ctxt =>
    let val prove-tac = Basic-Tactics.utp-simp-tac in
      (UTP-Tactics.inst-gen-pred-tac args prove-tac ctxt)
    end)
›
```

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

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

method-setup rel-auto = ‹
  (Scan.lift UTP-Tactics.scan-args) >>
    (fn args => fn ctxt =>
      let val prove-tac = Basic-Tactics.utp-auto-tac in
        (UTP-Tactics.inst-gen-rel-tac args prove-tac ctxt)
      end)
›

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

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

Simpler, one-shot versions of the above tactics, but without the possibility of dynamic arguments.

method *rel-simp'*
 uses *simp*
  = (*simp add*: *uexpr-transfer-laws upred-defs urel-defs alpha-splits*; *simp add*: *upred-defs urel-defs lens-defs prod.case-eq-if relcomp-unfold uexpr-transfer-extra uexpr-rep-eq-thms simp*)

method *rel-auto'*
 uses *simp intro elim dest*
  = (*simp-all add*: *uexpr-transfer-laws upred-defs urel-defs alpha-splits*, (*auto intro*: *intro elim*: *elim dest*: *dest simp add*: *upred-defs urel-defs lens-defs relcomp-unfold uexpr-transfer-laws uexpr-transfer-extra uexpr-rep-eq-thms simp*)?)

method *rel-blast'*

uses *simp intro elim dest*
= (*rel-simp′ simp*: *simp*, *blast intro*: *intro elim*: *elim dest*: *dest*)

end

8 Lifting Parser and Pretty Printer

theory *utp-lift-parser*
  imports *utp-expr-insts*
 keywords *no-utp-lift* :: *thy-decl-block* **and** *utp-lit-vars* :: *thy-decl-block* **and** *utp-expr-vars* :: *thy-decl-block*

begin

8.1 Parser

Here, we derive a parser for UTP expressions that mimicks (and indeed reuses) the syntax of HOL expressions. It has two main features: (1) it lifts HOL functions into UTP expressions using the (|>) construct; and (2) it recognises when a free variable is a declared lens and treats it as a UTP variable, whilst lifting HOL variables. The parser therefore allows free mixing of HOL operators and lenses.

Sometimes it is necessary that operators are handled in a special way however. We, therefore, first create a mutable data structure to store the names of constants that should not be lifted, and arguments of those constants that should not be further processed.

ML ‹
```
structure VarOption = Theory-Data
  (type T = bool
   val empty = false
   val extend = I
   val merge = (fn (x, y) => x orelse y));

structure NoLiftUTP = Theory-Data
  (type T = int list Symtab.table
   val empty = Symtab.empty
   val extend = I
   val merge = Symtab.merge (K true));

val - =
  let fun nolift-const thy (n, opt) =
       let val Const (c, -) = Proof-Context.read-const {proper = true, strict = false} (Proof-Context.init-global
thy) n
          in NoLiftUTP.map (Symtab.update (c, (map Value.parse-int opt))) thy end
  in

  Outer-Syntax.command @{command-keyword no-utp-lift} declare that certain constants should not be
lifted
   (Scan.repeat1 (Parse.term -- Scan.optional (Parse.$$$ ( |-- Parse.!!! (Scan.repeat1 Parse.number
--| Parse.$$$ ))) [])
     >> (fn ns =>
         Toplevel.theory
         (fn thy => Library.foldl (fn (thy, n) => nolift-const thy n) (thy, ns))))
  end;

  Outer-Syntax.command @{command-keyword utp-lit-vars} parse free variables as literals in UTP ex-
pressions
    (Scan.succeed (Toplevel.theory (VarOption.put false)));
  Outer-Syntax.command @{command-keyword utp-expr-vars} parse free variables as expressions in UTP
expressions
    (Scan.succeed (Toplevel.theory (VarOption.put true)));
⟩
```

The core UTP operators should not be lifted. Certain operators have arguments that also should not be processed further by expression lifting. For example, in a substitution update $\sigma(x \mapsto v)$, the lens x (i.e. the second argument) should not be lifted as its target is not an expression. Consequently, constants names in the command **no-utp-lift** can be accompanied by a list of numbers stating the arguments that should be not be further processed.

no-utp-lift
uexpr-appl uop (0) bop (0) trop (0) qtop (0) lit (0)
Groups.zero Groups.one plus uminus minus times divide
var (0) in-var (0) out-var (0) cond numeral (0)
inverse inverse-divide power power2

Add a quotation device for expressions that explicitly stops the lifting parser.

abbreviation (*input*) *quote-uexpr* :: ('a, 's) *uexpr* ⇒ ('a, 's) *uexpr* (@(-) [*999*] *999*) **where** *quote-uexpr p* ≡ *p*

no-utp-lift *quote-uexpr* (*0*)

The following function takes a parser, but not-yet type-checked term, and wherever it encounters an application, it inserts a UTP expression operator. Any operators that have been marked in the above structure will not be lifted. In addition, when it encounters a constant or free variable it will use the type system to determine whether is has a lens type. If it does, then it constructs a UTP variable expression; otherwise it constructs a literal.

FIXME: Actually, this test is a little too coarse for some situations. For example, when the lens is bound by a λ-abstraction the type data is not available, and so it will not necessarily be recognised as a lens. This could either be fixed by adding proper syntactic procedure for determining lenses, or else by using type inference wrt. the bound lambda term.

```
ML ⟨

  val list-appl = Library.foldl (fn (f, x) => Const (@{const-name uexpr-appl}, dummyT) $ f $ x);

  fun utp-lift-aux ctx (Const (n', t), args') =
    — Pre-processing: If we have a ¿ or ¿= operator then we turn these into ¡ and ¡=
    let val pn = (if (Lexicon.is-marked n') then Lexicon.unmark-const n' else n')
        val (args, n) =
          if (pn = @{const-abbrev greater} andalso (length args' = 2))
          then (rev args', @{const-name less})
          else if (pn = @{const-abbrev greater-eq} andalso (length args' = 2))
          then (rev args', @{const-name less-eq})
          else (args', pn)
    in
    — If the leading constructor is an already lifted UTP variable...
    if ((n = @{const-name var}) andalso (length args > 0))
    — ... then we take the first argument as the variable contents, and apply the remaining arguments
    then list-appl (Const (n, t) $ hd args, map (utp-lift ctx) (tl args))
    — Otherwise, if the name of the given constant is in the “no lifting” list...
    else if (member (op =) (Symtab.keys (NoLiftUTP.get (Proof-Context.theory-of ctx))) n)
      — ... then do not lift it, and also do not process any arguments in the given list of integers.
      then let val (SOME aopt) = Symtab.lookup (NoLiftUTP.get (Proof-Context.theory-of ctx)) n in
```

```
          Term.list-comb (Const (n, t), map-index (fn (i, t) => if (member (op =) aopt i) then t else
utp-lift ctx t) args) end
      — If the name is not in the “no lifting” list...
      else
        list-appl
        (case (Type-Infer-Context.const-type ctx n) of
          — ... and it’s a lens, then lift it as a UTP variable...
          SOME (Type (type-name ‹lens-ext›, -)) => Const (@{const-name var}, dummyT) $ (Const
(@{const-name pr-var}, dummyT) $ Const (n', t)) |
          — ... or, if it’s a UTP expression already, then leave it alone...
          SOME (Type (type-name ‹uexpr›, -)) => Const (n, t) |
          — ...otherwise, lift it to a HOL literal.
          - => Const (@{const-name lit}, dummyT) $ Const (n, t)
        , map (utp-lift ctx) args)
    end
    |

  — Free variables are handled similarly to constants; that they are usually lifted. The exception is
when the free variable actually refers to a constant, which can occur if lifting is applied during syntax
translation. In this case, we convert it to a constant first and then apply lifting to it.
  utp-lift-aux ctx (Free (n, t), args) =
    — We first extract the constant table from the context.
    let val consts = (Proof-Context.consts-of ctx)
        val {const-space, ...} = Consts.dest consts
        — The name must be internalised in case it needs qualifying.
        val c = Consts.intern consts n in
        — If the name refers to a declared constant, then we lift it as a constant.
        if (Name-Space.declared const-space c) then
          utp-lift-aux ctx (Const (c, t), args)
        — Otherwise, we simply apply normal lifting.
        else
          case (Syntax.check-term ctx (Free (n, t))) of
            Free (-, Type (type-name ‹lens-ext›, -))
                => list-appl (Const (@{const-name var}, dummyT) $ (Const (@{const-name pr-var},
dummyT) $ Free (n, t)), map (utp-lift ctx) args) |
            Free (-, Type (type-name ‹uexpr›, -)) => list-appl (Free (n, t), map (utp-lift ctx) args) |
            — This case tries to catch indexed predicates of the form P(i)
            Free (-, Type (type-name ‹fun›, [-, Type (type-name ‹uexpr›, -)])) => Term.list-comb (Free
(n, t), args) |
            - => list-appl (if (VarOption.get (Proof-Context.theory-of ctx))
                            then Free (n, t)
                            else Const (@{const-name lit}, dummyT) $ Free (n, t), map (utp-lift ctx) args)
          (* if (Symbol.is-ascii-upper (hd (Symbol.explode n))) then Free (n, t) else Const (@{const-name
lit}, dummyT) $ Free (n, t) *)

    end
    |

  — Bound variables are always lifted as well
  utp-lift-aux ctx (Bound n, args) = list-appl (Const (@{const-name lit}, dummyT) $ Bound n, map
(utp-lift ctx o Term-Position.strip-positions) args) |
  utp-lift-aux - (t, args) = raise TERM (-utp-lift-aux, t :: args)
  and
  (* FIXME: Think more about abstractions; at the moment they are essentially passed over. *)
(*  utp-lift ctx (Abs (x, ty, tm)) = Abs (x, ty, utp-lift ctx tm) | *)
```

```
  utp-lift ctx (Const (syntax-const‹-constrain›, k) $ t $ ty) = (utp-lift ctx t) |
  utp-lift ctx (Abs (x, ty, tm)) = Const (@{const-name uabs}, dummyT) $ Abs (x, ty, utp-lift ctx tm) |
  utp-lift - (Bound n) = (Const (@{const-name lit}, dummyT) $ Bound n) |
  utp-lift ctx t = utp-lift-aux ctx (Term.strip-comb t);
```

— Apply the Isabelle term parser, strip type constraints, perform lifting, and finally type check the resulting lifted term.

```
  fun utp-tr ctx content args =
    let fun err () = raise TERM (utp-tr, args) in
      (case args of
        [(Const (syntax-const‹-constrain›, -)) $ Free (s, -) $ p] =>
          (case Term-Position.decode-position p of
             SOME (pos, -) => (utp-lift ctx (Type.strip-constraints (Syntax.parse-term ctx (content (s,
pos)))))
          | NONE => err ())
      | - => err ())
    end;

›
```

Set up Cartouche syntax using the above.

syntax *-utp* :: ‹*cartouche-position* ⇒ *string*› (*UTP-*)
syntax *-utp* :: ‹*cartouche-position* ⇒ *string*› (***U***-)

```
parse-translation ‹
  [(syntax-const‹-utp›,
    (fn ctx => utp-tr ctx (Symbol-Pos.implode o Symbol-Pos.cartouche-content o Symbol-Pos.explode)))]
›
```

Cartouche parser for UTP expressions. We can either surround the whole of a UTP relation with a the cartouche, or alternatively just the program text.

syntax *-uexpr-cartouche* :: ‹*cartouche-position* ⇒ *logic*› (-)

translations
  -uexpr-cartouche e => *-utp e*

A more conventional parse translation version of the above

syntax
  -UTP :: *logic* ⇒ *logic* (*U*'(-'))
  -UTP :: *logic* ⇒ *logic* (***U***'(-'))

```
parse-translation ‹
  [(@{syntax-const -UTP}, fn ctx => fn term => utp-lift ctx (Term-Position.strip-positions (hd term)))]
›
```

8.2 Examples

A couple of examples

term $U(x\ @\ y)$

utp-expr-vars — Change behaviour so free variables are translated as expressions

term $U(x\ @\ y)$

utp-lit-vars

term UTP‹*f x*›

term ***U*** ‹*f x*›

term UTP‹(*xs* @ *ys*) ! *i*›

term UTP‹$x > y$›

term UTP‹*mm i*›

term UTP‹$\exists\ x.\ f\ x$›

term UTP‹*xs* ! $(x + y)$›

term UTP‹*xs* ! *i*›

term UTP‹$A \cup B$›

term UTP‹$\exists\ x.\ x \leq$ *xs* ! *i*›

term UTP‹$(x \leq 0)$›

term UTP‹(*length xs* $+ 1 + n \leq 0$)›

term UTP‹(*length xs* $+ 1 + n \leq 0$) $\vee$ *true*›

term UTP‹$\exists\ n.$ (*length xs* $+ 1 + n \leq 0$) $\vee$ *true*›

term UTP‹$\{x + y \mid x.\ 1 < x\}$›

term UTP‹$\lambda\ x.\ x + y$›

term UTP‹\$$x + 1 \leq$ \$$y'$›

term UTP‹\$$x'$ = \$$x + 1 \wedge$ \$$y'$ = \$$y$›

locale *test* =
  fixes $x :: \mathit{nat} \Longrightarrow {}'s$ **and** $\mathit{xs} :: \mathit{int\ list} \Longrightarrow {}'s$ **and** $P :: {}'s \Rightarrow ({}'a, {}'s)\ \mathit{uexpr}$
begin

  abbreviation (*input*) $z \equiv x$

The lens x and HOL variable y are automatically distinguished

  term $U(x + y)$

  term UTP‹\$*f v*›

  term UTP‹{*2*<..}›

  term $U(P\ i)$

end

term $\ll x \gg + \$y$

term $\ll x \gg + \$y$

term $\boldsymbol{U}(\&v < \mathit{0})$

term $U(\$y = \mathit{5})$

term $\boldsymbol{U}(\$y' = \mathit{1} + \$y)$

term $U(\$x + \$y + \$z + \$u \mathbin{/} \$f')$

term $\boldsymbol{U}(\$f\ x)$

term $U(\$f\ \$\mathbf{v}')$

term $e \oplus f\ \mathit{on}\ A$

term $U(\$x = v)$

term $U(\$tr' = \$tr\ @\ [a] \wedge \$\mathit{ref} \subseteq \$i{:}\mathit{ref}' \cup \$j{:}\mathit{ref}' \wedge \$x' = \$x + \mathit{1})$

utp-expr-vars

8.3 Linking Parser to Constants

end

9 Substitution

theory *utp-subst*
imports
  utp-expr
  utp-unrest
  utp-tactics
  utp-lift-parser
begin

9.1 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
  $\mathit{usubst} :: {}'s \Rightarrow {}'a \Rightarrow {}'b$ (**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* $=$ $('\beta,\ '\alpha)$ *uexpr*
type-synonym $'\alpha$ *usubst* $=$ $('\alpha,\ '\alpha)$ *uexpr*

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.

We can also represent an arbitrary substitution as below.

lift-definition *subst-nil* :: $('\alpha,\ '\beta)$ *psubst* (nil_s) **is** $\lambda\ s.\ undefined$ **.**

lift-definition *subst-id* :: $'\alpha$ *usubst* (id_s) **is** $\lambda s.\ s$ **.**

lift-definition *subst-comp* :: $('\beta,\ '\gamma)$ *psubst* $\Rightarrow$ $('\alpha,\ '\beta)$ *psubst* $\Rightarrow$ $('\alpha,\ '\gamma)$ *psubst* (**infixl** $\circ_s$ *55*) **is**
$(\circ)$ **.**

lift-definition *inv-subst* :: $('\alpha,\ '\beta)$ *psubst* $\Rightarrow$ $('\beta,\ '\alpha)$ *psubst* (inv_s) **is** *inv* **.**
lift-definition *inj-subst* :: $('\alpha,\ '\beta)$ *psubst* $\Rightarrow$ *bool* (inj_s) **is** *inj* **.**
lift-definition *bij-subst* :: $('\alpha,\ '\beta)$ *psubst* $\Rightarrow$ *bool* (bij_s) **is** *bij* **.**

declare *inj-subst-def* [*uexpr-transfer-extra*]
declare *bij-subst-def* [*uexpr-transfer-extra*]

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.

lift-definition *subst-upd* :: $('\alpha,'\beta)$ *psubst* $\Rightarrow$ $('a \Longrightarrow '\beta)$ $\Rightarrow$ $('a,\ '\alpha)$ *uexpr* $\Rightarrow$ $('\alpha,'\beta)$ *psubst*
is $\lambda\ \sigma\ x\ v\ s.\ put_x\ (\sigma\ s)\ (v\ s)$ **.**

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

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

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

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

syntax

```
  -unrest-usubst :: salpha ⇒ logic ⇒ logic ⇒ logic (infix ♯_s 20)
```

translations

```
  -unrest-usubst x p == CONST unrest-usubst x p
  -unrest-usubst (-salphaset (-salphamk (x +_L y))) P  <= -unrest-usubst (x +_L y) P
```

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).

lift-definition *par-subst* :: $'\alpha$ *usubst* ⇒ ($'a \Longrightarrow '\alpha$) ⇒ ($'b \Longrightarrow '\alpha$) ⇒ $'\alpha$ *usubst* ⇒ $'\alpha$ *usubst* **is**
$\lambda\ \sigma_1\ A\ B\ \sigma_2.\ (\lambda\ s.\ (s \oplus_L (\sigma_1\ s)\ on\ A) \oplus_L (\sigma_2\ s)\ on\ B)$ **.**

no-utp-lift *subst-upd* (*1*) *subst usubst usubst-lookup*

9.2 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** *salphas*

syntax

```
  -smaplet  :: [salpha, logic] => smaplet             (- /↦_s/ -)
             :: smaplet => smaplets             (-)
  -SMaplets :: [smaplet, smaplets] => smaplets (-,/ -)
  -SubstUpd :: ['m usubst, smaplets] => 'm usubst (-/'(-') [900,0] 900)
  -Subst    :: smaplets => logic            ((1[-]))
  -PSubst   :: smaplets => logic             ((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 U(y)
  -SubstUpd m (-smaplet x y)         <= CONST subst-upd m x y
  -Subst ms                        == -SubstUpd id_s ms
  -Subst (-SMaplets ms1 ms2)         <= -SubstUpd (-Subst ms1) ms2
  -PSubst ms                        == -SubstUpd nil_s ms
  -PSubst (-SMaplets ms1 ms2)        <= -SubstUpd (-PSubst ms1) ms2
  -SMaplets ms1 (-SMaplets ms2 ms3) <= -SMaplets (-SMaplets ms1 ms2) ms3
  -subst P es vs => CONST subst (-psubst id_s 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*
-subst P v x <= *CONST usubst* (*CONST subst-upd* id_s *x v*) *P*
-subst P v x <= *-subst P* (*-spvar x*) *v*
-par-subst σ_1 *A B* σ_2 == *CONST par-subst* σ_1 *A B* σ_2

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 and restriction to a substitution maplet.

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

definition *subst-restr* :: $'\alpha$ *usubst* $\Rightarrow$ ($'a \Longrightarrow '\alpha$) $\Rightarrow$ $'\alpha$ *usubst* (**infix** $\rhd_s$ *85*) **where**
[*uexpr-defs*]: *subst-restr* σ x = $[x \mapsto_s \langle\sigma\rangle_s\ x]$

9.3 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*]: $\langle id_s\rangle_s\ x$ = *var x*
  by (*transfer*, *simp*)

lemma *subst-id-var*: id_s = &**v**
  by (*transfer*, *auto simp add*: *lens-defs*)

lemma *subst-upd-id-lam* [*usubst*]: *subst-upd* &**v** *x v* = *subst-upd* id_s *x v*
  by (*simp add*: *subst-id-var*)

lemma *subst-id* [*simp*]: $id_s \circ_s \sigma = \sigma$ $\sigma \circ_s id_s = \sigma$
  by (*transfer*, *auto*)+

lemma *subst-upd-alt-def*: *subst-upd* σ *x v* = *bop* (put_x) σ *v*
  by (*transfer*, *simp*)

lemma *subst-apply-one-lens* [*usubst*]: $\langle\sigma\rangle_s$ $(\&\mathbf{v})_v$ = σ
  by (*transfer*, *simp add*: *lens-defs*)

A substitution update naturally yields the given expression.

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

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

Substitution update is idempotent.

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

lemma *usubst-upd-idem-sub* [*usubst*]:
  assumes $x \subseteq_L y$ *mwb-lens y*
  shows $\sigma(x \mapsto_s u, y \mapsto_s v) = \sigma(y \mapsto_s v)$
  using *assms*
  by (*simp add*: *subst-upd-def assms*, *transfer*, *simp add*: *fun-eq-iff sublens-put-put*)

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* **unfolding** *subst-upd-def*
  by (*transfer*, *auto simp add*: *subst-upd-def assms comp-def lens-indep-comm*)

lemma *usubst-upd-comm2*:
  assumes $z \bowtie y$
  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*
  using *assms* **unfolding** *subst-upd-def*
  by (*transfer*, *auto simp add*: *subst-upd-def assms comp-def lens-indep-comm*)

lemma *subst-upd-pr-var*: $s(\&x \mapsto_s v) = s(x \mapsto_s v)$
  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 $inj_s\ [x \mapsto_s \&y, y \mapsto_s \&x]$
proof (*simp add*: *inj-subst-def*, *rule injI*)
  fix $b_1 :: '\alpha$ **and** $b_2 :: '\alpha$
  assume $[\![[x \mapsto_s \&y, y \mapsto_s \&x]]\!]_e\ b_1 = [\![[x \mapsto_s \&y, y \mapsto_s \&x]]\!]_e\ b_2$
  hence *a*: $put_y\ (put_x\ b_1\ ([\![\&y]\!]_e\ b_1))\ ([\![\&x]\!]_e\ b_1) = put_y\ (put_x\ b_2\ ([\![\&y]\!]_e\ b_2))\ ([\![\&x]\!]_e\ b_2)$
    by (*transfer*, *simp*)
  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 $\Longrightarrow [x \mapsto_s var\ x] = id_s$
  apply (*simp add*: *subst-upd-def subst-id-def id-lens-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-var x*)] = id_s
  apply (*simp add*: *subst-upd-def pr-var-def subst-id-def id-lens-def*)
  apply (*transfer*)
  apply (*rule ext*)
  apply (*auto*)
  done

lemma *subst-sublens-var* [*usubst*]:
  $\llbracket$ *vwb-lens a*; $x \subseteq_L a$ $\rrbracket \Longrightarrow \langle\sigma(a \mapsto_s var\ b)\rangle_s\ x = var\ ((x\ /_L\ a)\ ;_L\ b)$
  by (*transfer*, *auto simp add*: *fun-eq-iff lens-defs*)

lemma *subst-nil-comp* [*usubst*]: $nil_s \circ_s \sigma = nil_s$
  by (*simp add*: *subst-nil-def comp-def*, *transfer*, *simp add*: *comp-def*)

lemma *subst-nil-apply*: $\llbracket nil_s \rrbracket_e\ x$ = *undefined*
  by (*simp add*: *subst-nil.rep-eq*)

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 σ $(x +_L y)$ «(u,v)» = $\sigma(y \mapsto_s$ «v», $x \mapsto_s$ «u»)
  by (*simp add*: *lens-defs uexpr-defs subst-upd-def*, *transfer*, *auto*)

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

lemma *subst-upd-out-lens-plus* [*usubst*]:
  subst-upd σ (*out-var* $(x +_L y)$) «(u,v)» = $\sigma(\$y\acute{} \mapsto_s$ «v», $\$x\acute{} \mapsto_s$ «u»)
  by (*simp add*: *lens-defs uexpr-defs subst-upd-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-def*, *transfer*, *simp*)

lemma *subst-upd-plus* [*usubst*]:
  $x \bowtie y \Longrightarrow$ *subst-upd s* $(x +_L y)$ e = $s(x \mapsto_s fst(e), y \mapsto_s snd(e))$
  by (*simp add*: *subst-upd-def lens-defs*, *transfer*, *auto simp add*: *fun-eq-iff prod.case-eq-if lens-indep-comm*)

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

lemma *usubst-apply-unrest*:
  $\llbracket$ *vwb-lens x*; $x \sharp_s \sigma$ $\rrbracket \Longrightarrow \langle\sigma\rangle_s\ x = var\ x$
  by (*transfer*, *auto simp add*: *fun-eq-iff*)
    (*metis mwb-lens-weak vwb-lens-mwb vwb-lens-wb wb-lens.get-put weak-lens.view-determination*)

There follows various laws about deleting variables from a substitution.

lemma *subst-del-id* [*usubst*]:
  vwb-lens x $\Longrightarrow id_s -_s x = id_s$

by (*simp add*: *subst-del-def subst-upd-def pr-var-def subst-id-def id-lens-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-def*, *transfer*, *simp*)

lemma *subst-del-upd-in* [*usubst*]:
$[\![$ *mwb-lens* a; $x \subseteq_L a$ $]\!] \Longrightarrow \sigma(x \mapsto_s v) -_s a = \sigma -_s a$
by (*simp add*: *subst-del-def subst-upd-def*, *transfer*, *simp add*: *sublens-put-put*)

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-def*, *transfer*, *simp add*: *lens-indep-comm*)

lemma *subst-restr-id* [*usubst*]: *vwb-lens* $x \Longrightarrow id_s \rhd_s x = id_s$
by (*simp add*: *subst-restr-def usubst*)

lemma *subst-restr-upd-in* [*usubst*]:
$[\![$ *vwb-lens* a; $x \subseteq_L a$ $]\!] \Longrightarrow \sigma(x \mapsto_s v) \rhd_s a = (\sigma \rhd_s a)(x \mapsto_s v)$
by (*simp add*: *subst-restr-def usubst subst-upd-def*, *transfer*,
simp add: *fun-eq-iff sublens'-prop1 sublens-implies-sublens' sublens-pres-vwb*)

lemma *subst-restr-upd-out* [*usubst*]:
$[\![$ *vwb-lens* a; $x \bowtie a$ $]\!] \Longrightarrow \sigma(x \mapsto_s v) \rhd_s a = (\sigma \rhd_s a)$
by (*simp add*: *subst-restr-def usubst subst-upd-def*, *transfer*
, *simp add*: *lens-indep.lens-put-irr2*)

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-def*, *transfer*, *auto*)

lemma *subst-unrest-sublens* [*usubst*]: $[\![$ $a \sharp P$; $x \subseteq_L a$ $]\!] \Longrightarrow \sigma(x \mapsto_s v) \dagger P = \sigma \dagger P$
by (*simp add*: *subst-upd-def*, *transfer*, *auto simp add*: *fun-eq-iff*,
metis (*no-types*, *lifting*) *lens.select-convs*(*2*) *lens-comp-def sublens-def*)

lemma *subst-unrest-2* [*usubst*]:
fixes P :: ($'a$, $'\alpha$) *uexpr*
assumes $x \sharp P$ $x \bowtie y$
shows $\sigma(x \mapsto_s u, y \mapsto_s v) \dagger P = \sigma(y \mapsto_s v) \dagger P$
using *assms*
by (*simp add*: *subst-upd-def*, *transfer*, *auto*, *metis lens-indep.lens-put-comm*)

lemma *subst-unrest-3* [*usubst*]:
fixes P :: ($'a$, $'\alpha$) *uexpr*
assumes $x \sharp P$ $x \bowtie y$ $x \bowtie z$
shows $\sigma(x \mapsto_s u,\ y \mapsto_s v,\ z \mapsto_s w) \dagger P = \sigma(y \mapsto_s v,\ z \mapsto_s w) \dagger P$
using *assms*
by (*simp add*: *subst-upd-def*, *transfer*, *auto*, *metis* (*no-types*, *hide-lams*) *lens-indep-comm*)

lemma *subst-unrest-4* [*usubst*]:
fixes P :: ($'a$, $'\alpha$) *uexpr*
assumes $x \sharp P$ $x \bowtie y$ $x \bowtie z$ $x \bowtie u$
shows $\sigma(x \mapsto_s e,\ y \mapsto_s f,\ z \mapsto_s g,\ u \mapsto_s h) \dagger P = \sigma(y \mapsto_s f,\ z \mapsto_s g,\ u \mapsto_s h) \dagger P$
using *assms*

by (*simp add*: *subst-upd-def*, *transfer*, *auto*, *metis* (*no-types*, *hide-lams*) *lens-indep-comm*)

lemma *subst-unrest-5* [*usubst*]:
fixes P :: ($'a$, $'\alpha$) *uexpr*
assumes $x \sharp P$ $x \bowtie y$ $x \bowtie z$ $x \bowtie u$ $x \bowtie v$
shows $\sigma(x \mapsto_s e, y \mapsto_s f, z \mapsto_s g, u \mapsto_s h, v \mapsto_s i) \dagger P = \sigma(y \mapsto_s f, z \mapsto_s g, u \mapsto_s h, v \mapsto_s i) \dagger P$
using *assms*
by (*simp add*: *subst-upd-def*, *transfer*, *auto*, *metis* (*no-types*, *hide-lams*) *lens-indep-comm*)

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

9.4 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*]: $id_s \dagger v = v$
unfolding *subst-id-def lens-defs* **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-uabs* [*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; $x \sharp (\langle\sigma\rangle_s\ x)$; $x \sharp_s \sigma -_s x$ $]\!] \Longrightarrow x \sharp (\sigma \dagger P)$
by (*simp add*: *subst-del-def subst-upd-def unrest-uexpr-def unrest-usubst-def pr-var-def*, *transfer*, *auto*)
(*metis vwb-lens.source-determination*)

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

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

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-case-prod* [*usubst*]:
  fixes P :: $'i \Rightarrow 'j \Rightarrow ('a, '\alpha)$ *uexpr*
  shows $\sigma \dagger$ *case-prod* $(\lambda\ x\ y.\ P\ x\ y)\ v =$ *case-prod* $(\lambda\ x\ y.\ \sigma \dagger P\ x\ y)\ v$
  by (*simp add*: *case-prod-beta'*)

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-power* [*usubst*]: $\sigma \dagger (e\ \hat{}\ n) = (\sigma \dagger e)\ \hat{}\ n$
  by (*simp add*: *power-rep-eq subst.rep-eq uexpr-eq-iff*)

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-numeral* [*usubst*]: $\sigma \dagger$ *numeral* $n =$ *numeral* n
  by (*simp add*: *numeral-uexpr-simp subst-lit*)

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

lemma *subst-subst* [*usubst*]: $\sigma \dagger \varrho \dagger e = (\varrho \circ_s \sigma) \dagger e$
  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 $\varrho(x \mapsto_s v) \circ_s \sigma = (\varrho \circ_s \sigma)(x \mapsto_s \sigma \dagger v)$
  unfolding *subst-upd-def* **by** (*transfer*, *auto*)

lemma *subst-singleton*:

fixes x :: $('a \Longrightarrow '\alpha)$
assumes $x \sharp_s \sigma$
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*

9.5 Ordering substitutions

A simplification procedure to reorder substitutions maplets lexicographically by variable syntax

simproc-setup *subst-order* (*subst-upd* (*subst-upd* σ *x u*) *y v*) =
⟨ (*fn* - => *fn ctx* => *fn ct* =>
    case (*Thm.term-of ct*) *of*
      Const (*utp-subst.subst-upd*, -) \$ (*Const* (*utp-subst.subst-upd*, -) \$ *s* \$ *x* \$ *u*) \$ *y* \$ *v*
    => *if* (*YXML.content-of* (*Syntax.string-of-term ctx x*) > *YXML.content-of*(*Syntax.string-of-term ctx y*))
        then SOME (*mk-meta-eq* @{*thm usubst-upd-comm*})
        else NONE |
     - => *NONE*)
⟩

9.6 Unrestriction laws

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

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

lemma *unrest-usubst-id* [*unrest*]:
mwb-lens x $\Longrightarrow x \sharp_s id_s$
by (*transfer*, *simp*)

lemma *unrest-usubst-upd* [*unrest*]:
$[\![x \bowtie y; x \sharp_s \sigma; x \sharp v]\!] \Longrightarrow x \sharp_s \sigma(y \mapsto_s v)$
by (*transfer*, *simp add*: *lens-indep-comm*)

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

Unrestriction can be demonstrated by showing substitution for its variables is ineffectual.

lemma *unrest-as-subst*: $(x \sharp P) \longleftrightarrow (\forall\ v.\ P[\![\ll v\gg/x]\!] = P)$
by (*transfer*, *auto simp add*: *fun-eq-iff*)

lemma *unrest-by-subst*: $[\![\bigwedge v.\ P[\![\ll v\gg/x]\!] = P]\!] \Longrightarrow x \sharp P$
by (*simp add*: *unrest-as-subst*)

9.7 Conditional Substitution Laws

lemma *usubst-cond-upd-1* [*usubst*]:
$\sigma(x \mapsto_s u) \triangleleft b \triangleright \varrho(x \mapsto_s v) = (\sigma \triangleleft b \triangleright \varrho)(x \mapsto_s (u \triangleleft b \triangleright v))$
by (*simp add*: *subst-upd-def uexpr-defs*, *transfer*, *auto*)

lemma *usubst-cond-upd-2* [*usubst*]:
$\llbracket$ *vwb-lens x*; $x \sharp_s \varrho$ $\rrbracket \Longrightarrow \sigma(x \mapsto_s u) \triangleleft b \triangleright \varrho = (\sigma \triangleleft b \triangleright \varrho)(x \mapsto_s (u \triangleleft b \triangleright \&x))$
by (*simp add*: *subst-upd-def unrest-usubst-def uexpr-defs pr-var-def*, *transfer*, *auto simp add*: *fun-eq-iff*)
(*metis lens-override-def lens-override-idem*)

lemma *usubst-cond-upd-3* [*usubst*]:
$\llbracket$ *vwb-lens x*; $x \sharp_s \sigma$ $\rrbracket \Longrightarrow \sigma \triangleleft b \triangleright \varrho(x \mapsto_s v) = (\sigma \triangleleft b \triangleright \varrho)(x \mapsto_s (\&x \triangleleft b \triangleright v))$
by (*simp add*: *subst-upd-def unrest-usubst-def uexpr-defs pr-var-def*, *transfer*, *auto simp add*: *fun-eq-iff*)
(*metis lens-override-def lens-override-idem*)

9.8 Parallel Substitution Laws

lemma *par-subst-id* [*usubst*]:
$\llbracket$ *vwb-lens A*; *vwb-lens B* $\rrbracket \Longrightarrow id_s\ [A|B]_s\ id_s = id_s$
by (*transfer*, *simp*)

lemma *par-subst-left-empty* [*usubst*]:
$\llbracket$ *vwb-lens A* $\rrbracket \Longrightarrow \sigma\ [\emptyset|A]_s\ \varrho = id_s\ [\emptyset|A]_s\ \varrho$
by (*simp add*: *par-subst-def pr-var-def*)

lemma *par-subst-right-empty* [*usubst*]:
$\llbracket$ *vwb-lens A* $\rrbracket \Longrightarrow \sigma\ [A|\emptyset]_s\ \varrho = \sigma\ [A|\emptyset]_s\ id_s$
by (*simp add*: *par-subst-def pr-var-def*)

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

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

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

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

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

9.9 Power Substitutions

interpretation *subst-monoid*: *monoid-mult subst-id subst-comp*
by (*unfold-locales*, *transfer*, *auto*)

notation *subst-monoid.power* (**infixr** $\hat{}_s$ *80*)

lemma *subst-power-rep-eq*: $\llbracket \sigma\ \hat{}_s\ n \rrbracket_e = \llbracket \sigma \rrbracket_e\ \hat{}\hat{}\ n$
by (*induct n*, *simp-all add*: *subst-id.rep-eq subst-comp.rep-eq*)

update-uexpr-rep-eq-thms

end

10 Meta-level Substitution

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

Meta substitution substitutes 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 $\lambda\ F\ v\ b.\ F\ (v\ b)\ b$ **.**

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

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

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

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

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

lemma *msubst-pair* [*usubst*]: (*P x y*)⟦(*x*, *y*) → $(e, f)_u$⟧ = (*P x y*)⟦*x* → *e*⟧⟦*y* → *f*⟧
  by (*rel-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*→*u*⟧ = *uop f* ((*v x*)⟦*x*→*u*⟧)
  by (*rel-auto*)

lemma *msubst-uop-2* [*usubst*]: (*uop f* (*v x y*))⟦(*x*,*y*)→*u*⟧ = *uop f* ((*v x y*)⟦(*x*,*y*)→*u*⟧)
  by (*pred-simp*, *pred-simp*)

lemma *msubst-bop* [*usubst*]: (*bop f* (*v x*) (*w x*))⟦*x*→*u*⟧ = *bop f* ((*v x*)⟦*x*→*u*⟧) ((*w x*)⟦*x*→*u*⟧)
  by (*rel-auto*)

lemma *msubst-bop-2* [*usubst*]: (*bop f* (*v x y*) (*w x y*))⟦(*x*,*y*)→*u*⟧ = *bop f* ((*v x y*)⟦(*x*,*y*)→*u*⟧) ((*w x y*)⟦(*x*,*y*)→*u*⟧)
  by (*pred-simp*, *pred-simp*)

lemma *msubst-var* [*usubst*]:
  (*utp-expr.var x*)⟦*y*→*u*⟧ = *utp-expr.var x*
  by (*pred-simp*)

lemma *msubst-var-2* [*usubst*]:
  (*utp-expr.var x*)⟦(*y*,*z*)→*u*⟧ = *utp-expr.var x*
  by (*pred-simp*)+

lemma *msubst-unrest* [*unrest*]: ⟦ ⋀ *v*. *x* ♯ *P*(*v*); *x* ♯ *k* ⟧ ⟹ *x* ♯ *P*(*v*)⟦*v*→*k*⟧
  by (*pred-auto*)

end
theory *utp-lift-pretty*
  imports *utp-subst utp-lift-parser*
  keywords *utp-pretty* :: *thy-decl-block* **and** *no-utp-pretty* :: *thy-decl-block* **and** *utp-const* :: *thy-decl-block* **and** *utp-lift-notation* :: *thy-decl-block*
begin

10.1 Pretty Printer

The pretty printer infers when a HOL expression is actually a UTP expression by determing whether it contains operators like *bop*, *lit* etc. If so, it inserts the syntactic UTP quote defined above and then pushes these upwards through the expression syntax as far as possible, removing expression operators along the way. In this way, lifted HOL expressions are printed exactly as the HOL expression with a quote around.

There are two phases to this implementation. Firstly, a collection of print translation functions for each of the combinators for functions, such as *uop* and *bop* insert a UTP quote for each subexpression that is not also headed by such a combinator. This is effectively trying to find "leaf nodes" in an expression. Secondly, a set of translation rules push the UTP quotes upwards, combining where necessary, to the highest possible level, removing the expression operators as they go.

We manifest the pretty printer through two commands that enable and disable it. Disabling allows us to inspect the syntactic structure of a term.

ML ‹

›

ML ‹
```
let val utp-tr-rules = map (fn (l, r) => Syntax.Print-Rule ((logic, l), (logic, r)))
  [(U(t) , U(U(t))),

(*
  (-UTP (-uex x P), -uex x (-UTP P)),
  (-UTP (-uall x P), -uall x (-UTP P)),
*)
  (U(-ulens-ovrd e f A), -ulens-ovrd (U(e)) (U(f)) A),

  (-UTP (-SubstUpd m (-smaplet x v)), -SubstUpd (-UTP m) (-smaplet x (-UTP v))),
  (-UTP (-Subst (-smaplet x v)), -Subst (-smaplet x (-UTP v))),
  (-UTP (-subst e v x), -subst (-UTP e) (-UTP v) x),

  (U(σ † e), U(σ) † U(e)),
  (U(f x) , U(f) |> U(x)),
  (U(λ x. f), (λ x • U(f))),
  (U(λ x. f), (λ x . U(f))),

  (U(f x) , CONST uop f U(x)),
  (U(f x y) , CONST bop f U(x) U(y)),
  (U(f x y z) , CONST trop f U(x) U(y) U(z)),
  (U(f x) , -UTP f (-UTP x))]

  val utp-terminals = [@{const-syntax zero-class.zero}, @{const-syntax one-class.one}, @{const-syntax
numeral}, @{const-syntax utrue}, @{const-syntax ufalse}];
 fun utp-consts ctx = @{syntax-const -UTP} :: filter (not o member (op =) utp-terminals) (map
Lexicon.mark-const (Symtab.keys (NoLiftUTP.get (Proof-Context.theory-of ctx))));

fun needs-mark ctx t =
  case t of
    (Const (@{syntax-const -free}, -) $ Free (-, Type (type-name ‹uexpr›, ts))) => true |
    (Const (@{syntax-const -free}, -)
     $ Free (-, Type (syntax-const ‹-ignore-type›, [Type (type-name ‹uexpr›, ts)]))) => true |
    Free (-, -) => true |
    - => false;

fun utp-mark-term ctx t =
  if (needs-mark ctx t) then Const (@{syntax-const -UTP}, dummyT) $ t else t;

fun mark-uexpr-leaf n = (n, fn - => fn typ => fn ts =>
  case typ of
    (Type (type-name ‹uexpr›, -)) => Const (@{syntax-const -UTP}, dummyT) $ Term.list-comb
(Const (n, dummyT), ts) |
    (Type (type-name ‹fun›, [-, Type (type-name ‹uexpr›, -)])) => Const (@{syntax-const -UTP},
dummyT) $ Term.list-comb (Const (n, dummyT), ts) |
  - => raise Match);

fun insert-U args pre ctx ts =
   if (Library.foldl (fn (x, (i, y)) => (not (member (op =) args i) andalso needs-mark ctx y) orelse
x) (false, (Library.map-index (fn x => x) ts)))
   then Library.foldl1 (op $) (pre @ map-index (fn (i, t) => if (member (op =) args i) then t else
utp-mark-term ctx t) ts)
  else raise Match;

fun insert-const-U args c = insert-U args [Const (c, dummyT)];


  (* Function to register a constant c with n arguments as a lifted constant that should be
   aware of U notation. The values in opt are any arguments that should be ignored when
   checking for lifting. *)

  fun mk-remove-U-prtr c n opt =
  let open Ast
     val vars = map (fn i => Variable (x ^ string-of-int i)) (0 upto (n−1))
     val mvars =
       map (fn i =>
            let val v = Variable (x ^ string-of-int i) in
            if (member (op =) opt i) then v else Appl (Constant @{syntax-const -UTP} :: [v])
            end
          ) (0 upto (n−1))
  in
  (Appl (Constant c :: vars), Appl (Constant c :: mvars))
  end;

fun mk-lift-U-prtr c n opt =
  let
    open Ast
    val (l, r) = mk-remove-U-prtr c n opt
  in
  if n = 0 then []
  else
  [
  Syntax.Print-Rule (
  Appl [Constant @{syntax-const -UTP}
       , l],
  r)
  ]
  end;

fun utp-remove-const-U thy (s, opt) =
 let val Const (ct, ty) = Proof-Context.read-const {proper = true, strict = false} (Proof-Context.init-global
thy) s
     val cs = Lexicon.mark-const ct
     val n = length (fst (Term.strip-type ty))
     val args = map Value.parse-int opt in
  (Syntax.Print-Rule (mk-remove-U-prtr cs n args))
  end;

fun add-utp-print-const (s, opt) thy =
 let val Const (ct, ty) = Proof-Context.read-const {proper = true, strict = false} (Proof-Context.init-global
thy) s
     val cs = Lexicon.mark-const ct
     val n = length (fst (Term.strip-type ty))
     val args = map Value.parse-int opt in
  (Sign.add-trrules (mk-lift-U-prtr cs n args) #>
   Sign.print-translation [(cs, insert-const-U args cs)]
  ) thy
  end;


(*
fun utp-consts ctx =
  [@{syntax-const -UTP},
   @{const-syntax lit},
   @{const-syntax var},
   @{const-syntax uop},
   @{const-syntax bop},
   @{const-syntax trop},
   @{const-syntax qtop},
(*     @{const-syntax subst-upd}, *)
   @{const-syntax plus},
   @{const-syntax minus},
   @{const-syntax times},
   @{const-syntax divide}];
```

```
*)

  fun uop-insert-U ctx (f :: ts) = insert-U [] [Const (@{const-syntax uop}, dummyT), f] ctx ts |
  uop-insert-U - - = raise Match;

  fun bop-insert-U ctx (f :: ts) = insert-U [] [Const (@{const-syntax bop}, dummyT), f] ctx ts |
  bop-insert-U - - = raise Match;

  fun trop-insert-U ctx (f :: ts) =
    insert-U [] [Const (@{const-syntax trop}, dummyT), f] ctx ts |
  trop-insert-U - - = raise Match;

  fun appl-insert-U ctx ts = insert-U [] [] ctx ts;

  val print-tr = [ (@{const-syntax var},
                     K (fn ts => if (ts = [])
                                    then Const (var, dummyT)
                                    else Const (@{syntax-const -UTP}, dummyT) $ hd(ts)))
                  , (@{const-syntax lit},
                     K (fn ts => if (ts = [])
                                    then Const (lit, dummyT)
                                    else Const (@{syntax-const -UTP}, dummyT) $ hd(ts)))
                  , (@{const-syntax trop}, trop-insert-U)
                  , (@{const-syntax bop}, bop-insert-U)
                  , (@{const-syntax uop}, uop-insert-U)
(*                  , (@{const-syntax udisj}, insert-const-U @{const-syntax udisj}) *)
                  , (@{const-syntax uexpr-appl}, appl-insert-U)];
  val ty-print-tr = map mark-uexpr-leaf utp-terminals;
  (* FIXME: We should also mark expressions that are free variables *)
  val no-print-tr = [ (@{syntax-const -UTP}, K (fn ts => Term.list-comb (@{print} hd ts, tl ts))) ];
  fun nolift-const thy (n, opt) =
      let val Const (c, -) = Proof-Context.read-const {proper = true, strict = false} (Proof-Context.init-global
thy) n
        in NoLiftUTP.map (Symtab.update (c, (map Value.parse-int opt))) thy end;
  fun utp-lift-notation thy (n, args) =
   let val Const (c, -) = Proof-Context.read-const {proper = true, strict = false} (Proof-Context.init-global
thy) n in
    (Lexicon.mark-const c,
     fn ctx => fn ts =>
      let val ts' = map-index (fn (i, t) => if (not (member (op =) (map Value.parse-int args) i)) then
utp-lift ctx (Term-Position.strip-positions t) else t) ts
       in if (ts = ts') then raise Match else Term.list-comb (Const (c, dummyT), ts') end)
      end;
  in
  Outer-Syntax.command @{command-keyword utp-lift-notation} insert UTP parser quotes into existing
notation
   (Scan.repeat1 (Parse.term -- Scan.optional (Parse.$$$ ( |-- Parse.!!! (Scan.repeat1 Parse.number
--| Parse.$$$ ))) [])
     >> (fn ns =>
         Toplevel.theory
         (fn thy => (Sign.parse-translation (map (utp-lift-notation thy) ns)
                    #> Sign.add-trrules ((map (utp-remove-const-U thy) ns))) thy)));

  Outer-Syntax.command @{command-keyword utp-pretty} enable pretty printing of UTP expressions
```

(Scan.succeed (Toplevel.theory (Isar-Cmd.translations utp-tr-rules #>
Sign.typed-print-translation ty-print-tr #>
Sign.print-translation print-tr
)));
(∗ FIXME: It actually isn't currently possible to disable pretty printing without destroying the term rewriting ∗)

Outer-Syntax.command @{command-keyword no-utp-pretty} disable pretty printing of UTP expressions
(Scan.succeed (Toplevel.theory (Isar-Cmd.no-translations utp-tr-rules #> Sign.print-translation no-print-tr)));

Outer-Syntax.command @{command-keyword utp-const} declare that certain UTP constants should not be lifted
(Scan.repeat1 (Parse.term -- Scan.optional (Parse.$$$ (|-- Parse.!!! (Scan.repeat1 Parse.number --| Parse.$$$))) [])
>> (fn ns =>
Toplevel.theory
(fn thy => Library.foldl (fn (thy, n) => nolift-const thy n |> add-utp-print-const n) (thy, ns))))

end;
›

utp-const
plus minus uminus times divide inverse inverse-divide power power2
subst-upd(1) usubst usubst-lookup(1)
utrue ufalse cond

term ***U***$(3 + \&x)$

utp-pretty

term ***U***$(3 + \&x)$

term *true*

term $U(P \vee \$x = 1 \longrightarrow false)$

term $U(true \wedge q)$

term ***U***$(1 + \&x)$

term $\ll x \gg + \$y$

term $\ll x \gg + \$y$

term ***U***$(\&v < 0)$

term ***U***$(\&v > 0)$

term $U(\$y = 5)$

term ***U***$(\$y' = 1 + \$y)$

term $U(\$x + \$y + \$z + \$u / \$f')$

term $\boldsymbol{U}(\$f\ x)$

term $U(\$f\ \$\mathbf{v}')$

term $e \oplus f\ on\ A$

term $U(\$x = v)$

term $U(\$tr' = \$tr\ @\ [a] \wedge \$ref \subseteq \$i{:}ref' \cup \$j{:}ref' \wedge \$x' = \$x + 1)$

term $U(e[\![v/x]\!])$

term $U((length\ e)[\![1{+}1/\&x]\!])$

term $U([x \mapsto_s 1 + 2])$

end

11 Alphabetised Predicates

theory *utp-pred*
imports
utp-expr-funcs
utp-subst
utp-meta-subst
utp-tactics
utp-lift-parser
utp-lift-pretty
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 [22].

11.1 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

```
  uconj  :: 'a ⇒ 'a ⇒ 'a (infixr ∧ 35)
  udisj  :: 'a ⇒ 'a ⇒ 'a (infixr ∨ 30)
  uimpl  :: 'a ⇒ 'a ⇒ 'a (infixr ⇒ 25)
  uiff   :: 'a ⇒ 'a ⇒ 'a (infixr ⇔ 25)
  unot   :: 'a ⇒ 'a (¬ - [40] 40)
  uex    :: ('a ⟹ 'α) ⇒ 'p ⇒ 'p
  uall   :: ('a ⟹ 'α) ⇒ 'p ⇒ 'p
```

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

utp-const
```
  uex(0) uall(0) unot uconj udisj uimpl uiff
```

abbreviation *shEx* :: ['β ⇒'α *upred*] ⇒ 'α *upred* **where**
shEx P ≡ «*Ex*» |> *uabs P*

abbreviation *shAll* :: ['β ⇒'α *upred*] ⇒ 'α *upred* **where**
shAll P ≡ «*All*» |> *uabs P*

utp-const *shEx shAll*

We set up two versions of each of the quantifiers: *uex* / *uall* and *shEx* / *shAll*. The former pair allows quantification of UTP variables, whilst the latter allows quantification of HOL variables in concert with the literal expression constructor $\boldsymbol{U}(x)$. 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)
  -shEx    :: pttrn ⇒ logic ⇒ logic  (∃ - · - [0, 10] 10)
  -shAll   :: pttrn ⇒ logic ⇒ logic  (∀ - · - [0, 10] 10)
  -shBEx   :: pttrn ⇒ logic ⇒ logic ⇒ logic  (∃ - ∈ - · - [0, 0, 10] 10)
  -shBAll  :: pttrn ⇒ logic ⇒ logic ⇒ logic  (∀ - ∈ - · - [0, 0, 10] 10)
  -shGAll  :: pttrn ⇒ logic ⇒ logic ⇒ logic  (∀ - | - · - [0, 0, 10] 10)
  -shGtAll :: idt ⇒ logic ⇒ logic ⇒ logic (∀ - > - · - [0, 0, 10] 10)
  -shLtAll :: idt ⇒ logic ⇒ logic ⇒ logic (∀ - < - · - [0, 0, 10] 10)
  -uvar-res :: logic ⇒ salpha ⇒ logic (infixl ↾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
  -shEx x P                   == CONST shEx (λ x. P)
  ∃ x ∈ A · P                   => ∃ x · «x» ∈u A ∧ P
  -shAll x P                  == CONST shAll (λ x. P)
  ∀ x ∈ A · P                   => ∀ x · «x» ∈u A ⇒ P
  ∀ x | P · Q                   => ∀ x · P ⇒ Q
```

```
∀ x > y · P            => ∀ x · CONST bop CONST less y «x» ⇒ P
∀ x < y · P            => ∀ x · CONST bop CONST less «x» y ⇒ P

-UTP (-uex x P)         <= -uex x (-UTP P)
-UTP (-uall x P)        <= -uall x (-UTP P)
-UTP (-shEx x P)         <= -shEx x (-UTP P)
-UTP (-shAll x P)        <= -shAll x (-UTP P)
```

11.2 Predicate operators

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

class *refine* = *order*

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

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

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

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

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

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

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

We trivially instantiate our refinement class

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

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

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

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

instantiation *uexpr* :: (*lattice*, *type*) *lattice*
begin
  lift-definition *sup-uexpr* :: ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr*
  is λ*P Q A*. *Lattices.sup* (*P A*) (*Q A*) **.**
  lift-definition *inf-uexpr* :: ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr* ⇒ ('*a*, '*b*) *uexpr*
  is λ*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** λ *A*. *Orderings.bot* **.**
  lift-definition *top-uexpr* :: ('*a*, '*b*) *uexpr* **is** λ *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*
  by (*intro-classes*; *transfer*; *auto simp add*: *INF-SUP-set*)

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

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

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

notation *gfp* (*μ*)
notation *lfp* (*ν*)

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

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

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

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

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

notation
  conj-upred (**infixr** $\wedge_p$ *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 *UINFIMUM* :: *'a set* ⇒ (*'a* ⇒ (*'b*::*complete-lattice*, *'s*) *uexpr*) ⇒ (*'b*, *'s*) *uexpr*
is *λ A F b*. *Sup* {$[\![F\ x]\!]_e$ *b* | *x*. *x* ∈ *A*} **.**

lift-definition *USUPREMUM* :: *'a set* ⇒ (*'a* ⇒ (*'b*::*complete-lattice*, *'s*) *uexpr*) ⇒ (*'b*, *'s*) *uexpr*
is *λ A F b*. *Inf* {$[\![F\ x]\!]_e$ *b* | *x*. *x* ∈ *A*} **.**

update-uexpr-rep-eq-thms

syntax
```
  -USup     :: pttrn ⇒ logic ⇒ logic              (⋀ - • - [0, 10] 10)
  -USup     :: pttrn ⇒ logic ⇒ logic              (⨆ - • - [0, 10] 10)
  -USup-mem :: pttrn ⇒ logic ⇒ logic ⇒ logic        (⋀ -∈- • - [0, 0, 10] 10)
```

-USup-mem :: *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigsqcup$ *-*$\in$*-* $\cdot$ *-* [*0*, *0*, *10*] *10*)
-UInf :: *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigvee$ *-* $\cdot$ *-* [*0*, *10*] *10*)
-UInf :: *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigsqcap$ *-* $\cdot$ *-* [*0*, *10*] *10*)
-UInf-mem :: *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigvee$ *-*$\in$*-* $\cdot$ *-* [*0*, *10*] *10*)
-UInf-mem :: *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* ($\bigsqcap$ *-*$\in$*-* $\cdot$ *-* [*0*, *10*] *10*)

translations
$\bigsqcap$ $x \in A \cdot F$ == *CONST UINFIMUM A* (λ *x*. *F*)
$\bigsqcap$ $x \cdot F$ == $\bigsqcap$ $x \in$ *CONST UNIV* $\cdot$ *F*
$\bigsqcup$ $x \in A \cdot F$ == *CONST USUPREMUM A* (λ *x*. *F*)
$\bigsqcup$ $x \cdot F$ == $\bigsqcup$ $x \in$ *CONST UNIV* $\cdot$ *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 *all* :: ($'a \Longrightarrow '\alpha$) $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* **is**
λ *x P b*. ($\forall$ *v*. *P*(put_x *b v*)) **.**

lift-definition *scex* :: *'s scene* $\Rightarrow$ *'s upred* $\Rightarrow$ *'s upred* **is**
λ *a P b*. $\exists$ b'. *P*($b \oplus_S b'$ *on a*) **.**

lift-definition *scall* :: *'s scene* $\Rightarrow$ *'s upred* $\Rightarrow$ *'s upred* **is**
λ *a P b*. $\forall$ b'. *P*($b \oplus_S b'$ *on 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* $\Rightarrow$ $'\beta$ *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* **.**

declare *taut-def* [*uexpr-transfer-laws*]

The following function extracts the characteristic set of a predicate

lift-definition *upred-set* :: *'a upred* $\Rightarrow$ *'a set* ($[\![\text{-}]\!]_p$) **is**
λ *P*. *Collect P* **.**

Configuration for 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

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* (∈) *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 *par-subst-def* [*upred-defs*]
declare *subst-del-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*,*simp*]
declare *false-alt-def*[*THEN sym*,*simp*]

lemma *upred-set-eqI*: $[\![p]\!]_p = [\![q]\!]_p \Longrightarrow p = q$
  by (*metis eq-iff mem-Collect-eq upred-ref-iff upred-set.rep-eq*)

11.3 Unrestriction Laws

lemma *unrest-allE*:
  $[\![\ \Sigma \sharp P;\ P = true \Longrightarrow Q;\ P = false \Longrightarrow Q\ ]\!] \Longrightarrow Q$
  by (*pred-auto*)

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

lemma *unrest-false* [*unrest*]: $x \sharp$ *false*

by (*pred-auto*)

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

lemma *unrest-disj* [*unrest*]: $[\![x \sharp (P :: '\alpha\ upred); x \sharp Q]\!] \Longrightarrow x \sharp P \vee Q$
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-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 lens-indep-comm*
by (*rel-auto*, *fastforce+*)

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

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

lemma *unrest-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*)

11.4 Used-by laws

lemma *usedBy-not* [*unrest*]:
$[\![x \natural P]\!] \Longrightarrow x \natural (\neg P)$
by (*pred-simp*)

lemma *usedBy-conj* [*unrest*]:
$[\![x \natural P; x \natural Q]\!] \Longrightarrow x \natural (P \wedge Q)$
by (*pred-simp*)

lemma *usedBy-disj* [*unrest*]:
$[\![x \natural P; x \natural Q]\!] \Longrightarrow x \natural (P \vee Q)$
by (*pred-simp*)

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

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

11.5 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*]: $\sigma \dagger (\bigsqcap\ i{\in}A \cdot P(i)) = (\bigsqcap\ i{\in}A \cdot \sigma \dagger P(i))$
by (*pred-auto*)

lemma *subst-USUP* [*usubst*]: $\sigma \dagger (\bigsqcup\ i{\in}A \cdot P(i)) = (\bigsqcup\ i{\in}A \cdot \sigma \dagger P(i))$
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*]:
$x \sharp_s \sigma \Longrightarrow \sigma \dagger (\exists\ x \cdot P) = (\exists\ x \cdot \sigma \dagger P)$
by (*pred-auto*)

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

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

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

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

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

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

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

11.6 Sandbox for conjectures

definition *utp-sandbox* :: $'\alpha$ *upred* $\Rightarrow$ *bool* (*TRY*'(-')) **where**
$TRY(P) = (P = undefined)$

translations
  $P <= CONST\ utp\text{-}sandbox\ P$

end

12 Alphabet Manipulation

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

12.1 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*)?

12.2 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* $\Rightarrow$ $('\beta \Longrightarrow '\alpha) \Rightarrow ('a, '\alpha)$ *uexpr* (**infixr** $\oplus_p$ *95*)
is $\lambda\ P\ x\ b.\ P\ (get_x\ b)$ **.**

utp-const *aext*(*1*)

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 Σ lens identifies the source and view types. Thus an alphabet extension using this has no effect.

lemma *aext-id* [*simp*]: $P \oplus_p 1_L = P$
  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* [*simp*]: $\ll v \gg \oplus_p a = \ll v \gg$
  by (*pred-auto*)

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

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

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

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

lemma *aext-false* [*simp*]: $false \oplus_p a = false$
  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-shEx* [*alpha*]: $(\exists\ x \cdot P\ x) \oplus_p a = (\exists\ x \cdot P\ x \oplus_p a)$
  by (*rel-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-UINF-ind* [*alpha*]: $(\bigsqcap\ x \cdot P\ x) \oplus_p a =(\bigsqcap\ x \cdot (P\ x \oplus_p a))$
  by (*pred-auto*)

lemma *aext-UINF-ind-2* [*alpha*]: $(\bigsqcap\ (i, j) \cdot P\ i\ j) \oplus_p a = (\bigsqcap\ (i, j) \cdot P\ i\ j \oplus_p a)$
  by (*rel-auto*)

lemma *aext-UINF-mem* [*alpha*]: $(\bigsqcap\ x{\in}A \cdot P\ x) \oplus_p a =(\bigsqcap\ x{\in}A \cdot (P\ x \oplus_p a))$
  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*

12.3 Expression 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* $\Rightarrow ('\beta \Longrightarrow '\alpha) \Rightarrow ('a, '\beta)$ *uexpr* (**infixr** $\restriction_e$ *90*)
is λ $P\ x\ b.\ P\ (create_x\ b)$ **.**

utp-const *arestr*(*1*)

update-uexpr-rep-eq-thms

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

lemma *arestr-aext* [*simp*]: *mwb-lens a* $\Longrightarrow (P \oplus_p a) \restriction_e a = P$
  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 $(P \restriction_e a) \oplus_p a = P$
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 *aext-arestr-symlens* [*alpha*]:
  assumes *sym-lens a unrest* $\mathcal{C}_a$ P
  shows $(P \restriction_e \mathcal{V}_a) \oplus_p \mathcal{V}_a = P$
  using *assms*
  by (*rel-auto'*, *metis* (*no-types*, *lifting*) *lens-indep-def sym-lens.indep-region-coregion sym-lens.put-region-coregion-cover*)

Alternative formulation of the above law using used-by instead of unrestriction.

lemma *aext-arestr'* [*alpha*]:
  assumes $a \natural P$
  shows $(P \restriction_e a) \oplus_p a = P$
  by (*rel-simp*, *metis assms lens-override-def usedBy-uexpr.rep-eq*)

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

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

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

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

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

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

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

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

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

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

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

lemma *arestr-eq* [*alpha*]: $(P =_u Q)\upharpoonright_e x = (P\upharpoonright_e x =_u Q\upharpoonright_e x)$
  by (*pred-auto*)

lemma *ares-UINF-ind* [*alpha*]: $(\bigsqcap\ i \cdot P\ i) \upharpoonright_e a = (\bigsqcap\ i \cdot P\ i \upharpoonright_e a)$
  by (*rel-auto*)

lemma *ares-UINF-ind-2* [*alpha*]: $(\bigsqcap\ (i, j) \cdot P\ i\ j) \upharpoonright_e a = (\bigsqcap\ (i, j) \cdot P\ i\ j \upharpoonright_e a)$
  by (*rel-auto*)

12.4 Predicate Alphabet Restriction

In order to restrict the variables of a predicate, we also need to existentially quantify away the other variables. We can't do this at the level of expressions, as quantifiers are not applicable here. Consequently, we need a specialised version of alphabet restriction for predicates. It both restricts the variables using quantification and then removes them from the alphabet type using expression restriction.

definition *upred-ares* :: $'\alpha\ upred \Rightarrow ('\beta \Longrightarrow '\alpha) \Rightarrow '\beta\ upred$
where [*upred-defs*]: $upred\text{-}ares\ P\ a = (P \upharpoonright_v a) \upharpoonright_e a$

utp-const *upred-ares*(*1*)

syntax

-upred-ares :: *logic* ⇒ *salpha* ⇒ *logic* (**infixl** $\restriction_p$ *90*)

translations
-upred-ares P a == *CONST upred-ares P a*

lemma *upred-aext-ares* [*alpha*]:
vwb-lens a $\Longrightarrow$ $P \oplus_p a \restriction_p a = P$
by (*pred-auto*)

lemma *upred-ares-aext* [*alpha*]:
$a \natural P \Longrightarrow (P \restriction_p a) \oplus_p a = P$
by (*pred-auto*)

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

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

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

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

12.5 Alphabet Lens Laws

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

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

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

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

12.6 Substitution Alphabet Extension

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

lift-definition *subst-aext* :: *'α usubst* ⇒ (*'α* $\Longrightarrow$ *'β*) ⇒ *'β usubst* (**infix** $\oplus_s$ *65*)
is $\lambda\ \sigma\ x.\ (\lambda\ s.\ put_x\ s\ (\sigma\ (get_x\ s)))$ **.**

utp-const *subst-aext*(*1*)

update-uexpr-rep-eq-thms

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

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

lemma *apply-subst-ext* [*alpha*]:
  vwb-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*)

lemma *subst-aext-comp* [*usubst*]:
  vwb-lens $a \Longrightarrow (\sigma \oplus_s a) \circ_s (\varrho \oplus_s a) = (\sigma \circ_s \varrho) \oplus_s a$
  by *pred-auto*

lemma *subst-arestr* [*usubst*]: *vwb-lens* $a \Longrightarrow \sigma \dagger (P \upharpoonright_e a) = (((\sigma \oplus_s a) \dagger P) \upharpoonright_e a)$
  by (*pred-auto*)

lemma *subst-lit-aext* [*usubst*]: *weak-lens* $a \Longrightarrow (P \oplus_p a)[\![\ll v\gg/\&a{:}x]\!] = (P[\![\ll v\gg/\&x]\!] \oplus_p a)$
  by (*rel-simp*)

12.7 Substitution Alphabet Restriction

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

lift-definition *subst-ares* :: $'\alpha$ *usubst* $\Rightarrow ('\beta \Longrightarrow '\alpha) \Rightarrow '\beta$ *usubst* (**infix** $\upharpoonright_s$ *65*)
is $\lambda\ \sigma\ x.\ (\lambda\ s.\ get_x\ (\sigma\ (create_x\ s)))$ **.**

utp-const *subst-ares*(*1*)

update-uexpr-rep-eq-thms

lemma *id-subst-res* [*usubst*]:
  mwb-lens $x \Longrightarrow id_s \upharpoonright_s x = id_s$
  by *pred-auto*

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

lemma *subst-ext-res* [*usubst*]:
  mwb-lens $x \Longrightarrow (\sigma \oplus_s x) \upharpoonright_s x = \sigma$
  by (*pred-auto*)

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

13 Lifting Expressions

theory *utp-lift*
  imports
    utp-alphabet utp-lift-pretty
begin

13.1 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 $\lceil P \rceil_< \equiv P \oplus_p fst_L$

notation *lift-pre* $(\text{-}^< \ [999] \ 999)$

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

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 $\lceil P \rceil_> \equiv P \oplus_p snd_L$

notation *lift-post* $(\text{-}^> \ [999] \ 999)$

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

13.2 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*)

13.3 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*)

13.4 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 $\$x\acute{} \sharp \lceil p \rceil_<$
  by (*pred-auto*)

13.5 Parser and Pretty Printer

utp-const *lift-pre drop-pre lift-post drop-post*

term $U((p{::}'a\ upred)^< \leq p^<)$

term $U(1^< \leq p^< \wedge true)$

term $U(p^< \leq p^>)$

end

14 Predicate Calculus Laws

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

14.1 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* $(\leq)$ $(<)$
  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 *taut-conj*: $‘A \wedge B‘ = (‘A‘ \wedge ‘B‘)$
  by (*rel-auto*)

lemma *taut-conj-elim* [*elim!*]:
  $[\![\ ‘A \wedge B‘;\ [\![\ ‘A‘;\ ‘B‘\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
  by (*rel-auto*)

lemma *taut-refine-impl*: $[\![\ Q \sqsubseteq P;\ ‘P‘\ ]\!] \Longrightarrow ‘Q‘$
  by (*rel-auto*)

lemma *taut-shEx-elim*:
  $[\![\ ‘(\exists\ x \cdot P\ x)‘;\ \bigwedge x.\ \Sigma \sharp P\ x;\ \bigwedge x.\ ‘P\ x‘ \Longrightarrow Q\ ]\!] \Longrightarrow Q$

by (*rel-blast*)

Linking refinement and HOL implication

lemma *refine-prop-intro*:
  assumes $\Sigma \sharp P$ $\Sigma \sharp Q$ ‘Q‘ $\Longrightarrow$ ‘P‘
  shows $P \sqsubseteq Q$
  using *assms*
  by (*pred-auto*)

lemma *taut-not*: $\Sigma \sharp P \Longrightarrow (\neg$ ‘P‘$)$ = ‘$\neg\ P$‘
  by (*rel-auto*)

lemma *taut-shAll-intro*:
  $\forall\ x.$ ‘$P\ x$‘ $\Longrightarrow$ ‘$\forall\ x \cdot P\ x$‘
  by (*rel-auto*)

lemma *taut-shAll-intro-2*:
  $\forall\ x\ y.$ ‘$P\ x\ y$‘ $\Longrightarrow$ ‘$\forall\ (x,\ y) \cdot P\ x\ y$‘
  by (*rel-auto*)

lemma *taut-impl-intro*:
  $\llbracket\ \Sigma \sharp P;$ ‘P‘ $\Longrightarrow$ ‘Q‘ $\rrbracket \Longrightarrow$ ‘$P \Rightarrow Q$‘
  by (*rel-auto*)

lemma *upred-eval-taut*:
  ‘$P\llbracket \ll b \gg / \&\mathbf{v} \rrbracket$‘ = $\llbracket P \rrbracket_e b$
  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 *false-sup* [*simp*]: $false \sqcap P = P$ $P \sqcap false = P$
  by (*pred-auto*)+

lemma *true-inf* [*simp*]: $true \sqcup P = P$ $P \sqcup true = P$
  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*]: $(P \wedge \neg P) = \mathit{false}$
  by (*pred-auto*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

by (*pred-auto*)

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

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

14.2 Lattice laws

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

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

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

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

lemma *USUP-mem-false* [*simp*]: $I \neq \{\} \Longrightarrow (\bigsqcup\ i{\in}I \cdot \mathit{false}) = \mathit{false}$
 by (*rel-simp*)

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

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

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

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

lemma *UINF-cong-eq*:
 $[\![\ A = B;\ \bigwedge x.\ x \in A \Longrightarrow `Q_1(x) =_u Q_2(x)`\ ]\!] \Longrightarrow$
  $(\bigsqcap\ x{\in}A \cdot Q_1(x)) = (\bigsqcap\ x{\in}B \cdot Q_2(x))$
 by (*pred-simp*, *metis* (*mono-tags*, *hide-lams*))

lemma *UINF-as-Sup*: $(\bigsqcap\ P \in \mathcal{P} \cdot P) = \bigsqcap\ \mathcal{P}$
 apply (*simp add*: *upred-defs uexpr-appl.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 uexpr-appl.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 uexpr-appl.rep-eq lit.rep-eq Sup-uexpr-def*)
apply (*pred-simp*)
apply (*simp add*: *full-SetCompr-eq*)
done

lemma *UINF-as-Sup-image*: ($\bigsqcap\ P{\in}A \cdot f(P)$) = $\bigsqcap$ (f ' A)
apply (*simp add*: *upred-defs uexpr-appl.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 uexpr-appl.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 (*pred-simp*)
apply (*simp add*: *Setcompr-eq-image*)
done

lemma *USUP-as-Inf-collect'*: ($\bigsqcup P \cdot f(P)$) = ($\bigsqcup P.\ f(P)$)
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 uexpr-appl.rep-eq lit.rep-eq Inf-uexpr-def*)
apply (*pred-simp*)
apply (*rule cong*[*of Inf*])
 apply (*auto*)
done

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*, *auto intro*: *cong*[*of Sup Sup*] *simp add*: *UINF-as-Sup-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 *not-UINF-ind*: ($\neg$ ($\bigsqcap\ i \cdot P(i)$)) = ($\bigsqcup\ i \cdot \neg\ P(i)$)
by (*pred-auto*)

lemma *not-USUP-ind*: ($\neg$ ($\bigsqcup\ i \cdot P(i)$)) = ($\bigsqcap\ i \cdot \neg\ P(i)$)

by (*pred-auto*)

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

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

lemma *UINF-atLeast-first*:
$P(n) \sqcap (\bigsqcap\ i \in \{\mathit{Suc}\ n..\} \cdot P(i)) = (\bigsqcap\ i \in \{n..\} \cdot P(i))$
proof −
have *insert n* $\{\mathit{Suc}\ n..\} = \{n..\}$
by (*auto*)
thus *?thesis*
by (*metis UINF-insert*)
qed

lemma *UINF-atLeast-Suc*:
$(\bigsqcap\ i \in \{\mathit{Suc}\ m..\} \cdot P(i)) = (\bigsqcap\ i \in \{m..\} \cdot P(\mathit{Suc}\ i))$
by (*rel-simp*, *metis* (*full-types*) *Suc-le-D not-less-eq-eq*)

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

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

lemma *USUP-atLeast-first*:
$(P(n) \wedge (\bigsqcup\ i \in \{\mathit{Suc}\ n..\} \cdot P(i))) = (\bigsqcup\ i \in \{n..\} \cdot P(i))$
proof −
have *insert n* $\{\mathit{Suc}\ n..\} = \{n..\}$
by (*auto*)
thus *?thesis*
by (*metis USUP-insert conj-upred-def*)
qed

lemma *USUP-atLeast-Suc*:
$(\bigsqcup\ i \in \{\mathit{Suc}\ m..\} \cdot P(i)) = (\bigsqcup\ i \in \{m..\} \cdot P(\mathit{Suc}\ i))$
by (*rel-simp*, *metis* (*full-types*) *Suc-le-D not-less-eq-eq*)

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 uexpr-appl.rep-eq lit.rep-eq*, *pred-auto*)

lemma *conj-UINF-ind-dist*:
$(P \wedge (\bigsqcap\ Q \cdot F(Q))) = (\bigsqcap\ Q \cdot P \wedge F(Q))$
by *pred-auto*

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

lemma *UINF-conj-UINF* [*simp*]:
  $((\bigsqcap\ i{\in}I \cdot P(i)) \lor (\bigsqcap\ i{\in}I \cdot Q(i))) = (\bigsqcap\ i{\in}I \cdot P(i) \lor Q(i))$
  by (*rel-auto*)

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

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

lemma *UINF-all-cong* [*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-all-cong*:
  assumes $\bigwedge P.\ F(P) = G(P)$
  shows $(\bigsqcup\ P \cdot F(P)) = (\bigsqcup\ P \cdot G(P))$
  by (*simp add*: *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 *USUP-is-forall*: $(\bigsqcup\ x \cdot P(x)) = (\forall\ x \cdot P(x))$
  by (*pred-simp*)

lemma *USUP-ind-is-forall*: $(\bigsqcup\ x{\in}A \cdot P(x)) = (\forall\ x{\in}\ll A\gg \cdot P(x))$
  by (*pred-auto*)

lemma *UINF-is-exists*: $(\bigsqcap\ x \cdot P(x)) = (\exists\ x \cdot P(x))$
  by (*pred-simp*)

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

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

lemma *UINF-upto-expand-first*:
 $m < n \Longrightarrow (\bigsqcap\ i \in \{m..{<}n\} \cdot P(i)) = ((P(m) :: {'\alpha}\ upred) \vee (\bigsqcap\ i \in \{Suc\ m..{<}n\} \cdot P(i)))$
 apply (*rel-auto*) **using** *Suc-leI le-eq-less-or-eq* **by** *auto*

lemma *UINF-upto-expand-last*:
 $(\bigsqcap\ i \in \{0..{<}Suc(n)\} \cdot P(i)) = ((\bigsqcap\ i \in \{0..{<}n\} \cdot P(i)) \vee P(n))$
 apply (*rel-auto*)
 using *less-SucE* **by** *blast*

lemma *UINF-Suc-shift*: $(\bigsqcap\ i \in \{Suc\ 0..{<}Suc\ n\} \cdot P(i)) = (\bigsqcap\ i \in \{0..{<}n\} \cdot P(Suc\ i))$
 apply (*rel-simp*)
 apply (*rule cong*[*of Sup*], *auto*)
 using *less-Suc-eq-0-disj* **by** *auto*

lemma *USUP-upto-expand-first*:
 $(\bigsqcup\ i \in \{0..{<}Suc(n)\} \cdot P(i)) = (P(0) \wedge (\bigsqcup\ i \in \{1..{<}Suc(n)\} \cdot P(i)))$
 apply (*rel-auto*)
 using *not-less* **by** *auto*

lemma *USUP-Suc-shift*: $(\bigsqcup\ i \in \{Suc\ 0..{<}Suc\ n\} \cdot P(i)) = (\bigsqcup\ i \in \{0..{<}n\} \cdot P(Suc\ i))$
 apply (*rel-simp*)
 apply (*rule cong*[*of Inf*], *auto*)
 using *less-Suc-eq-0-disj* **by** *auto*

lemma *UINF-list-conv*:
 $(\bigsqcap\ i \in \{0..{<}length(xs)\} \cdot f\ (xs\ !\ i)) = foldr\ (\vee)\ (map\ f\ xs)\ false$
 apply (*induct xs*)
  apply (*rel-auto*)
 apply (*simp*)
 thm *UINF-upto-expand-first UINF-Suc-shift*
 apply (*simp add*: *UINF-upto-expand-first UINF-Suc-shift*)
 done

lemma *USUP-list-conv*:
 $(\bigsqcup\ i \in \{0..{<}length(xs)\} \cdot f\ (xs\ !\ i)) = foldr\ (\wedge)\ (map\ f\ xs)\ true$
 apply (*induct xs*)
  apply (*rel-auto*)
 apply (*simp-all add*: *USUP-upto-expand-first USUP-Suc-shift*)
 done

lemma *UINF-refines*:
 $[\![\bigwedge i.\ i{\in}I \Longrightarrow P \sqsubseteq Q\ i]\!] \Longrightarrow P \sqsubseteq (\bigsqcap\ i{\in}I \cdot Q\ i)$
 by (*simp add*: *UINF-as-Sup-collect*, *metis SUP-least*)

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*

14.3 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*)+)

14.4 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 *shEx-insert*: $(\exists\ x \in insert_u\ y\ A \cdot P(x)) = (P(x)[\![x \to y]\!] \vee (\exists\ x \in A \cdot P(x)))$
by (*pred-auto*)

lemma *shEx-one-point*: $(\exists\ x \cdot «x» =_u v \wedge P(x)) = P(x)[\![x \to v]\!]$
by (*rel-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*)

14.5 Case Splitting

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

14.6 UTP Quantifiers

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

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

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

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

lemma *all-sub*: $\llbracket$ *mwb-lens* y; $x \subseteq_L y$ $\rrbracket \Longrightarrow (\forall\ x \cdot \forall\ y \cdot P) = (\forall\ y \cdot P)$
  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*)

lemma *ex-conj-contr-left*: $x \sharp P \Longrightarrow (\exists\ x \cdot P \wedge Q) = (P \wedge (\exists\ x \cdot Q))$
  by (*pred-auto*)

lemma *ex-conj-contr-right*: $x \sharp Q \Longrightarrow (\exists\ x \cdot P \wedge Q) = ((\exists\ x \cdot P) \wedge Q)$
  by (*pred-auto*)

lemma *ex-override-def*: *weak-lens* $x \Longrightarrow [\![\exists\ x \cdot P]\!]_e\ b = (\exists\ b'.\ [\![P]\!]_e\ (b \oplus_L b'\ on\ x))$
  by (*rel-simp*, *metis weak-lens.put-get*)

lemma *ex-scene-def*: *mwb-lens* $a \Longrightarrow (\exists\ a \cdot P) = scex\ [\![a]\!]_\sim\ P$
  by (*simp add*: *uexpr-eq-iff ex-override-def scex.rep-eq lens-scene-override*)

lemma *scex-combine*:
  assumes *idem-scene x idem-scene y* $x\ \#\#_S\ y$
  shows $(scex\ x\ (scex\ y\ P)) = (scex\ (x \sqcup_S y)\ P)$
proof −
  have $\bigwedge b\ b'\ b''.\ [\![P]\!]_e\ (b \oplus_S b'\ on\ x \oplus_S b''\ on\ y) \Longrightarrow \exists\, b'.\ [\![P]\!]_e\ (b \oplus_S b'\ on\ (x \sqcup_S y))$
  proof −
    fix $b\ b'\ b''$
    assume *a1*: $[\![P]\!]_e\ (b \oplus_S b'\ on\ x \oplus_S b''\ on\ y)$
    have *f2*: $\forall\, a.\ a \oplus_S a\ on\ x = a$
      by (*simp add*: *assms*(*1*))
    have *f3*: $\forall\, a.\ a \oplus_S a\ on\ y = a$
      by (*metis assms*(*2*) *scene-override-idem*)
    have $\forall\, a\ aa.\ aa \oplus_S a\ on\ y \oplus_S a\ on\ x = aa \oplus_S a\ on\ (y \sqcup_S x)$
      by (*simp add*: *assms*(*3*) *scene-compat-sym scene-override-union*)
    then show $\exists\, a.\ [\![P]\!]_e\ (b \oplus_S a\ on\ (x \sqcup_S y))$
      using *f3 f2 a1* **by** (*metis* (*no-types*) *assms*(*3*) *scene-override-overshadow-left scene-override-union scene-union-commute*)
  qed

thus *?thesis*
 using *assms*(*3*) *scene-override-union* **by** (*rel-auto*, *fastforce*)
qed

lemma *ex-commute-set*: $[\![$ *vwb-lens a*; *vwb-lens b*; $a \#\#_L b$ $]\!] \Longrightarrow (\exists\ a \cdot \exists\ b \cdot P) = (\exists\ b \cdot \exists\ a \cdot P)$
 by (*simp add*: *lens-defs lens-scene.rep-eq scene-compat.rep-eq scene-union-commute scex-combine ex-scene-def*)

14.7 Variable Restriction

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

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

14.8 Conditional laws

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

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

lemma *cond-true-false* [*simp*]: $true \triangleleft b \triangleright false = b$ **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*]:$(P \triangleleft true \triangleright Q) = P$ **by** (*pred-auto*)

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

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*)

14.9 Additional Expression Laws

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

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

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

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

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

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

lemma *ulit-neq* [*simp*]: $x \neq y \Longrightarrow (\ll x\gg =_u \ll y\gg) = \mathit{false}$
  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*)+

14.10 Refinement By Observation

Function to obtain the set of observations of a predicate

definition *obs-upred* :: $'\alpha\ \mathit{upred} \Rightarrow '\alpha\ \mathit{set}$ ($[\![\text{-}]\!]_o$)
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

14.11 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 $U((\exists\ x \cdot \&x = \&y \wedge P) \wedge (\exists\ x \cdot \&x = \&y \wedge \neg\ P)) = \mathit{false}$
using *assms*
by (*pred-simp*, *simp add*: *lens-indep-sym*)

end

15 Healthiness Conditions

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

15.1 Main Definitions

We collect closure laws for healthiness conditions in the following theorem attribute.

named-theorems *closure*

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

A predicate P is healthy, under healthiness function H, if P is a fixed-point of H.

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*

lemma *Healthy-intro*: $H(P) = P \Longrightarrow P$ *is* H
  by (*simp add*: *Healthy-def*)

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

abbreviation *Healthy-carrier* :: $'\alpha$ *health* $\Rightarrow$ $'\alpha$ *upred set* ($[\![\text{-}]\!]_H$)
where $[\![H]\!]_H \equiv \{P.\ P$ *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*:
  $[\![\ F \in [\![\mathcal{H}_1]\!]_H \to [\![\mathcal{H}_2]\!]_H;\ P$ *is* $\mathcal{H}_1\ ]\!] \Longrightarrow \mathcal{H}_2(F(P)) = F(P)$
  using *Healthy-if* **by** *blast*

lemma *Healthy-comp*:
  $[\![\ P$ *is* $\mathcal{H}_1;\ P$ *is* $\mathcal{H}_2\ ]\!] \Longrightarrow P$ *is* $\mathcal{H}_1 \circ \mathcal{H}_2$
  by (*simp add*: *Healthy-def*)

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*:
  $[\![\ P \in F$ ` $A;\ \bigwedge x.\ F\ x$ *is* $H\ ]\!] \Longrightarrow P$ *is* H
  by *blast*

lemma *Healthy-case-prod* [*closure*]:
  $[\![\ \bigwedge x\ y.\ P\ x\ y$ *is* $H\ ]\!] \Longrightarrow$ *case-prod* $P\ v$ *is* H
  by (*simp add*: *prod.case-eq-if*)

lemma *Healthy-SUPREMUM*:
  $A \subseteq [\![H]\!]_H \Longrightarrow$ *SUPREMUM* $A\ H = \bigsqcap\ A$

by (*drule Healthy-carrier-image*, *presburger*)

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

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

lemma *Healthy-mu* [*closure*]:
assumes $\mathit{mono}\ F\ F \in [\![id]\!]_H \to [\![H]\!]_H$
shows $\mu\ F\ \mathit{is}\ H$
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*: $[\![\ A \subseteq [\![H]\!]_H;\ P \in A\ ]\!] \Longrightarrow P\ \mathit{is}\ H$
by *blast*

15.2 Properties of Healthiness Conditions

definition *Idempotent* :: *'α health ⇒ bool* **where**
$\mathit{Idempotent}(H) \longleftrightarrow (\forall\ P.\ H(H(P)) = H(P))$

abbreviation *Monotonic* :: *'α health ⇒ bool* **where**
$\mathit{Monotonic}(H) \equiv \mathit{mono}\ H$

definition *IMH* :: *'α health ⇒ bool* **where**
$\mathit{IMH}(H) \longleftrightarrow \mathit{Idempotent}(H) \wedge \mathit{Monotonic}(H)$

definition *Antitone* :: *'α health ⇒ bool* **where**
$\mathit{Antitone}(H) \longleftrightarrow (\forall\ P\ Q.\ Q \sqsubseteq P \longrightarrow (H(P) \sqsubseteq H(Q)))$

definition *Conjunctive* :: *'α health ⇒ bool* **where**
$\mathit{Conjunctive}(H) \longleftrightarrow (\exists\ Q.\ \forall\ P.\ H(P) = (P \wedge Q))$

definition *FunctionalConjunctive* :: *'α health ⇒ bool* **where**
$\mathit{FunctionalConjunctive}(H) \longleftrightarrow (\exists\ F.\ \forall\ P.\ H(P) = (P \wedge F(P)) \wedge \mathit{Monotonic}(F))$

definition *WeakConjunctive* :: *'α health ⇒ bool* **where**
$\mathit{WeakConjunctive}(H) \longleftrightarrow (\forall\ P.\ \exists\ Q.\ H(P) = (P \wedge Q))$

definition *Disjunctuous* :: *'α health ⇒ bool* **where**
[*upred-defs*]: $\mathit{Disjunctuous}\ H = (\forall\ P\ Q.\ H(P \sqcap Q) = (H(P) \sqcap H(Q)))$

definition *Continuous* :: *'α health ⇒ bool* **where**
[*upred-defs*]: $\mathit{Continuous}\ H = (\forall\ A.\ A \neq \{\} \longrightarrow H\ (\bigsqcap\ A) = \bigsqcap\ (H\ `\ A))$

lemma *Healthy-Idempotent* [*closure*]:
$\mathit{Idempotent}\ H \Longrightarrow H(P)\ \mathit{is}\ H$
by (*simp add*: *Healthy-def Idempotent-def*)

lemma *Healthy-range*: $\mathit{Idempotent}\ H \Longrightarrow \mathit{range}\ H = [\![H]\!]_H$

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*]:
$\llbracket$ *Idempotent f*; *Idempotent g*; $f \circ g = g \circ f$ $\rrbracket \Longrightarrow$ *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-id'* [*closure*]:
mono $(\lambda\ X.\ X)$
by (*simp add*: *monoI*)

lemma *Monotonic-const* [*closure*]:
Monotonic $(\lambda\ x.\ c)$
by (*simp add*: *mono-def*)

lemma *Monotonic-comp* [*intro*]:
$\llbracket$ *Monotonic f*; *Monotonic g* $\rrbracket \Longrightarrow$ *Monotonic* $(f \circ g)$
by (*simp add*: *mono-def*)

lemma *Monotonic-inf* [*closure*]:
assumes *Monotonic P Monotonic Q*
shows *Monotonic* $(\lambda\ X.\ P(X) \sqcap Q(X))$
using *assms* **by** (*simp add*: *mono-def*, *rel-auto*)

lemma *Monotonic-cond* [*closure*]:
assumes *Monotonic P Monotonic Q*
shows *Monotonic* $(\lambda\ X.\ P(X) \triangleleft b \triangleright Q(X))$
by (*simp add*: *assms cond-monotonic*)

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 $HC(P \land Q) = (HC(P) \land Q)$
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 $HC(P \land Q) = (HC(P) \land HC(Q))$

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*]: $\llbracket$ *Continuous H*; $A \neq \{\}$ $\rrbracket \Longrightarrow H\ (\bigsqcap\ A) = (\bigsqcap\ P{\in}A.\ H(P))$
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*]:
$\llbracket$ *Continuous f*; *Continuous g* $\rrbracket \Longrightarrow$ *Continuous* $(f \circ g)$
by (*simp add*: *Continuous-def*)

lemma *Continuous-const* [*closure*]: *Continuous* (λ *X*. *P*)
  by *pred-auto*

lemma *Continuous-cond* [*closure*]:
  assumes *Continuous F Continuous G*
  shows *Continuous* (λ *X*. *F*(*X*) ◁ *b* ▷ *G*(*X*))
  using *assms* **by** (*pred-auto*)

Closure laws derived from continuity

lemma *Sup-Continuous-closed* [*closure*]:
  ⟦ *Continuous H*; ⋀ *i*. *i* ∈ *A* ⟹ *P*(*i*) *is H*; *A* ≠ {} ⟧ ⟹ (⨅ *i*∈*A*. *P*(*i*)) *is H*
  by (*drule ContinuousD*[*of H P ' A*], *simp add*: *UINF-as-Sup*[*THEN sym*])
    (*metis* (*no-types*, *lifting*) *Healthy-def′ SUP-cong image-image*)

lemma *UINF-mem-Continuous-closed* [*closure*]:
  ⟦ *Continuous H*; ⋀ *i*. *i* ∈ *A* ⟹ *P*(*i*) *is H*; *A* ≠ {} ⟧ ⟹ (⨅ *i*∈*A* · *P*(*i*)) *is H*
  by (*simp add*: *Sup-Continuous-closed UINF-as-Sup-collect*)

lemma *UINF-mem-Continuous-closed-pair* [*closure*]:
  assumes *Continuous H* ⋀ *i j*. (*i*, *j*) ∈ *A* ⟹ *P i j is H A* ≠ {}
  shows (⨅ (*i*,*j*)∈*A* · *P i j*) *is H*
proof −
  have (⨅ (*i*,*j*)∈*A* · *P i j*) = (⨅ *x*∈*A* · *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* ⋀ *i j k*. (*i*, *j*, *k*) ∈ *A* ⟹ *P i j k is H A* ≠ {}
  shows (⨅ (*i*,*j*,*k*)∈*A* · *P i j k*) *is H*
proof −
  have (⨅ (*i*,*j*,*k*)∈*A* · *P i j k*) = (⨅ *x*∈*A* · *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-mem-Continuous-closed-quad* [*closure*]:
  assumes *Continuous H* ⋀ *i j k l*. (*i*, *j*, *k*, *l*) ∈ *A* ⟹ *P i j k l is H A* ≠ {}
  shows (⨅ (*i*,*j*,*k*,*l*)∈*A* · *P i j k l*) *is H*
proof −
  have (⨅ (*i*,*j*,*k*,*l*)∈*A* · *P i j k l*) = (⨅ *x*∈*A* · *P* (*fst x*) (*fst* (*snd x*)) (*fst* (*snd* (*snd x*))) (*snd* (*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-mem-Continuous-closed-quint* [*closure*]:
  assumes *Continuous H* ⋀ *i j k l m*. (*i*, *j*, *k*, *l*, *m*) ∈ *A* ⟹ *P i j k l m is H A* ≠ {}

shows ($\bigsqcap$ *(i,j,k,l,m)*∈*A* · *P i j k l m*) *is H*
proof −
  have ($\bigsqcap$ *(i,j,k,l,m)*∈*A* · *P i j k l m*)
        = ($\bigsqcap$ *x*∈*A* · *P* (*fst x*) (*fst* (*snd x*)) (*fst* (*snd* (*snd x*))) (*fst* (*snd* (*snd* (*snd x*)))) (*snd* (*snd* (*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

All continuous functions are also Scott-continuous

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

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

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

end

16 Alphabetised Relations

theory *utp-rel*
imports
  utp-pred-laws
  utp-healthy
  utp-lift
  utp-tactics
  utp-lift-pretty
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 [22].

16.1 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 *inα* :: (*'α* ⟹ *'α* × *'β*) **where**
[*lens-defs*]: *inα* = fst_L

definition *outα* :: (*'β* ⟹ *'α* × *'β*) **where**
[*lens-defs*]: *outα* = 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*]: $x ;_L in\alpha = in\text{-}var\ x$
  by (*simp add*: *fst-lens-def inα-def in-var-def*)

lemma *var-out-alpha* [*simp*]: $x ;_L out\alpha = out\text{-}var\ x$
  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-in-var-unrest* [*usubst*]:
  $in\alpha \sharp_s \sigma \Longrightarrow \langle\sigma\rangle_s\ (in\text{-}var\ x) = \$x$
  by (*rel-simp*, *metis fstI*)

lemma *usubst-lookup-out-var-unrest* [*usubst*]:
  $out\alpha \sharp_s \sigma \Longrightarrow \langle\sigma\rangle_s\ (out\text{-}var\ x) = \$x\acute{}$
  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**
$\lfloor\sigma\rfloor_s \equiv \sigma \upharpoonright_s in\alpha$

utp-const *usubst-rel-lift usubst-rel-drop*

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*)

16.2 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 $('\alpha, '\beta)\ urel = ('\alpha \times '\beta)\ upred$
type-synonym $'\alpha\ hrel = ('\alpha \times '\alpha)\ upred$
type-synonym $('a, '\alpha)\ hexpr = ('a, '\alpha \times '\alpha)\ uexpr$

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

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** ;; *61*)
  uassigns :: $('a, 'b)$ *psubst* $\Rightarrow 'c$ $(\langle \text{-} \rangle_a)$
  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 $\boldsymbol{U}(b^<)$.

definition *lift-rcond* $(\lceil \text{-} \rceil_{\leftarrow})$ **where**
[*upred-defs*]: $\lceil b \rceil_{\leftarrow} = \lceil b \rceil_{<}$

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

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 $((O))$. Since this returns a set, the definition states that the state binding b is an element of this set.

lift-definition *seqr*::$('\alpha, '\beta)$ *urel* $\Rightarrow ('\beta, '\gamma)$ *urel* $\Rightarrow ('\alpha \times '\gamma)$ *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 $\boldsymbol{U}(true)$ and $\boldsymbol{U}(false)$ that are explicitly typed by a homogeneous alphabet.

abbreviation *seqh* :: $'\alpha$ *hrel* $\Rightarrow '\alpha$ *hrel* $\Rightarrow '\alpha$ *hrel* (**infixr** $;;_h$ *61*) **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. This approach, which is originally due to Back [3], permits more general assignment expressions. 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* :: $('\alpha,'\beta)$ *psubst* $\Rightarrow ('\alpha, '\beta)$ *urel*
  is $\lambda\ \sigma\ (b,\ b').\ b' = \sigma(b)$ **.**

adhoc-overloading
  uassigns assigns-r

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_s

adhoc-overloading
  uskip skip-r

Non-deterministic assignment, also known as "choose", assigns an arbitrarily chosen value to the given variable

definition *nd-assign* :: $('a \Longrightarrow '\alpha) \Rightarrow '\alpha$ *hrel* **where**
[*urel-defs*]: *nd-assign* $x = (\bigsqcap\ v \cdot$ *assigns-r* $[x \mapsto_s \ll v \gg])$

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 $\boldsymbol{U}(false)$ (miracle). An assertion is the same, but yields $\boldsymbol{U}(true)$, which is an abort. They are the same as tests, as in Kleene Algebra with Tests [24, 1] (KAT), which embeds a Boolean algebra into a Kleene algebra to represent conditions.

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

notation *rassume* (*?*[-])

utp-lift-notation *rassume*

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

utp-lift-notation *rassert*

We also encode "naked" guarded commands [8, **?**] by composing an assumption with a relation.

definition *rgcmd* :: $'a$ *upred* $\Rightarrow 'a$ *hrel* $\Rightarrow 'a$ *hrel* $(\text{-} \longrightarrow_r \text{-}\ [55, 56]\ 55)$ **where**

[*urel-defs*]: *rgcmd b P* = (*rassume b* ;; *P*)

utp-lift-notation *rgcmd* (*1*)

We define two variants of while loops based on strongest and weakest fixed points. The former is $\boldsymbol{U}$(*false*) for an infinite loop, and the latter is $\boldsymbol{U}$(*true*).

definition *while-top* :: $'\alpha$ *cond* ⇒ $'\alpha$ *hrel* ⇒ $'\alpha$ *hrel* ($while^{\top}$ - *do* - *od*) **where**
[*urel-defs*]: *while-top b P* = (ν *X* · (*P* ;; *X*) ◁ *b* ▷$_r$ *II*)

notation *while-top* (*while* - *do* - *od*)

utp-lift-notation *while-top* (*1*)

definition *while-bot* :: $'\alpha$ *cond* ⇒ $'\alpha$ *hrel* ⇒ $'\alpha$ *hrel* ($while_{\perp}$ - *do* - *od*) **where**
[*urel-defs*]: *while-bot b P* = (μ *X* · (*P* ;; *X*) ◁ *b* ▷$_r$ *II*)

utp-lift-notation *while-bot* (*1*)

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

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

utp-lift-notation *while-inv* (*2*)

While loops with invariant decoration – total correctness.

definition *while-inv-bot* :: $'\alpha$ *cond* ⇒ $'\alpha$ *cond* ⇒ $'\alpha$ *hrel* ⇒ $'\alpha$ *hrel* ($while_{\perp}$ - *invr* - *do* - *od 71*) **where**
[*urel-defs*]: *while-inv-bot b p S* = *while-bot b S*

utp-lift-notation *while-inv-bot* (*2*)

While loops with invariant and variant decorations – total correctness.

definition *while-vrt* ::
  $'\alpha$ *cond* ⇒ $'\alpha$ *cond* ⇒ (*nat*, $'\alpha$) *uexpr* ⇒ $'\alpha$ *hrel* ⇒ $'\alpha$ *hrel* (*while* - *invr* - *vrt* - *do* - *od*) **where**
[*urel-defs*]: *while-vrt b p v S* = *while-bot b S*

utp-lift-notation *while-vrt* (*3*)

translations
  ?[*b*] <= *?*[*U*(*b*)]
  {*b*}$_\perp$ <= {*U*(*b*)}$_\perp$
  while b do P od <= *while U*(*b*) *do P od*
  while b invr c do P od <= *while U*(*b*) *invr U*(*c*) *do P od*

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

definition *rel-var-res* :: $'\alpha$ *hrel* ⇒ ($'a \Longrightarrow '\alpha$) ⇒ $'\alpha$ *hrel* (**infix** $\restriction_\alpha$ *80*) **where**
[*urel-defs*]: *P* $\restriction_\alpha$ *x* = (∃ \$*x* · ∃ \$*x*´ · *P*)

Alphabet extension and restriction add additional variables by the given lens in both their primed and unprimed versions.

definition *rel-aext* :: $'\beta$ *hrel* ⇒ ($'\beta \Longrightarrow '\alpha$) ⇒ $'\alpha$ *hrel*
where [*upred-defs*]: *rel-aext P a* = *P* $\oplus_p$ (*a* $\times_L$ *a*)

definition *rel-ares* :: $'\alpha$ *hrel* ⇒ ($'\beta \Longrightarrow '\alpha$) ⇒ $'\beta$ *hrel*

where [*upred-defs*]: $rel\text{-}ares\ P\ a = (P \upharpoonright_p (a \times a))$

We next describe frames and antiframes with the help of lenses. A frame states that P defines how variables in a changed, and all those outside of a remain the same. An antiframe describes the converse: all variables outside a are specified by P, and all those in remain the same. For more information please see [25].

definition $frame :: ('a \Longrightarrow '\alpha) \Rightarrow '\alpha\ hrel \Rightarrow '\alpha\ hrel$ **where**
[*urel-defs*]: $frame\ a\ P = (P \wedge \$\mathbf{v}' =_u \$\mathbf{v} \oplus \$\mathbf{v}'\ on\ \&a)$

definition $antiframe :: ('a \Longrightarrow '\alpha) \Rightarrow '\alpha\ hrel \Rightarrow '\alpha\ hrel$ **where**
[*urel-defs*]: $antiframe\ a\ P = (P \wedge \$\mathbf{v}' =_u \$\mathbf{v}' \oplus \$\mathbf{v}\ on\ \&a)$

Frame extension combines alphabet extension with the frame operator to both add additional variables and then frame those.

definition $rel\text{-}frext :: ('\beta \Longrightarrow '\alpha) \Rightarrow '\beta\ hrel \Rightarrow '\alpha\ hrel$ **where**
[*upred-defs*]: $rel\text{-}frext\ a\ P = frame\ a\ (rel\text{-}aext\ P\ a)$

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*]: $nameset\ a\ P = (P \upharpoonright_v \{\$\mathbf{v},\$a'\})$

The modify and freeze operators below are analogous to the frame and antiframe, but they discard updates to variables outside (inside) the frame, rather than requiring that they do not change.

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

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

16.3 Syntax Translations

— Alternative traditional conditional syntax

abbreviation (*input*) *rifthenelse* ((*if* (-)/ *then* (-)/ *else* (-)/ *fi*))
  where $rifthenelse\ b\ P\ Q \equiv P \triangleleft b \triangleright_r Q$

abbreviation (*input*) *rifthen* ((*if* (-)/ *then* (-)/ *fi*))
  where $rifthen\ b\ P \equiv rifthenelse\ b\ P\ II$

utp-lift-notation *rifthenelse* (*1 2*)
utp-lift-notation *rifthen* (*1*)

syntax
  — Iterated sequential composition
  $\text{-}seqr\text{-}iter :: pttrn \Rightarrow 'a\ list \Rightarrow '\sigma\ hrel \Rightarrow '\sigma\ hrel$ ((*3*;; - : - ·/ -) [*0*, *0*, *10*] *10*)
  — Single and multiple assignement
  $\text{-}assignment :: svids \Rightarrow uexprs \Rightarrow '\alpha\ hrel$ ('(-') := '(-'))
  $\text{-}assignment :: svids \Rightarrow uexprs \Rightarrow '\alpha\ hrel$ (**infixr** := *62*)
  — Non-deterministic assignment
  $\text{-}nd\text{-}assign :: svids \Rightarrow logic$ (- := ∗ [*62*] *62*)
  — Substitution constructor
  $\text{-}mk\text{-}usubst :: svids \Rightarrow uexprs \Rightarrow '\alpha\ usubst$

```
— Alphabetised skip
-skip-ra        :: salpha ⇒ logic (II-)
— Frame
-frame          :: salpha ⇒ logic ⇒ logic (-:[-] [99,0] 100)
— Antiframe
-antiframe      :: salpha ⇒ logic ⇒ logic (-:⟦-⟧ [79,0] 80)
— Relational Alphabet Extension
-rel-aext  :: logic ⇒ salpha ⇒ logic (infixl $\oplus_r$ 90)
— Relational Alphabet Restriction
-rel-ares  :: logic ⇒ salpha ⇒ logic (infixl $\restriction_r$ 90)
— Frame Extension
-rel-frext :: salpha ⇒ logic ⇒ logic (-:[-]$^+$ [99,0] 100)
— Nameset
-nameset        :: salpha ⇒ logic ⇒ logic (ns - • - [0,10] 10)
— Modify
-modify         :: salpha ⇒ logic ⇒ logic (mdf - • - [0,10] 10)
— Freeze
-freeze         :: salpha ⇒ logic ⇒ logic (frz - • - [0,10] 10)
```

translations

```
;; x : l • P ⇌ (CONST seqr-iter) l (λx. P)
-mk-usubst σ (-svid-unit x) v ⇌ σ(&x $\mapsto_s$ v)
-mk-usubst σ (-svid-list x xs) (-uexprs v vs) ⇌ (-mk-usubst (σ(&x $\mapsto_s$ v)) xs vs)
-assignment xs vs => CONST uassigns (-mk-usubst $id_s$ xs vs)
-assignment x v <= CONST uassigns (CONST subst-upd $id_s$ x v)
-assignment x v <= -assignment (-spvar x) v
-assignment x v <= -assignment x (-UTP v)
-nd-assign x => CONST nd-assign (-mk-svid-list x)
-nd-assign x <= CONST nd-assign x
x,y := u,v <= CONST uassigns (CONST subst-upd (CONST subst-upd $id_s$ (CONST pr-var x) u)
(CONST pr-var y) v)
-skip-ra v ⇌ CONST skip-ra v
-frame x P => CONST frame x P
-frame (-salphaset (-salphamk x)) P <= CONST frame x P
-antiframe x P => CONST antiframe x P
-antiframe (-salphaset (-salphamk x)) P <= CONST antiframe x P
-nameset x P == CONST nameset x P
-modify x P == CONST modify x P
-freeze x P == CONST freeze x P
-rel-aext P a == CONST rel-aext P a
-rel-ares P a == CONST rel-ares P a
-rel-frext a P == CONST rel-frext a 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*) *'α hrel* <= (*type*) (*bool*, *'α*) *hexpr*

16.4 Relation Properties

We describe some properties of relations, including functional and injective relations. We also provide operators for extracting the domain and range of a UTP relation.

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

definition *uinj* :: (*'a*, *'b*) *urel* ⇒ *bool*
where [*urel-defs*]: *uinj R* ⟷ *II* ⊑ *R* ;; R^-

definition *Pre* :: (*'α*, *'β*) *urel* ⇒ *'α upred*
where [*upred-defs*]: *Pre P* = $\lfloor \exists\ \$\mathbf{v}' \cdot P \rfloor_<$

definition *Post* :: (*'α*, *'β*) *urel* ⇒ *'β upred*
where [*upred-defs*]: *Post P* = $\lfloor \exists\ \$\mathbf{v} \cdot P \rfloor_>$

utp-const *Pre Post*

— Configuration for UTP tactics.

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

16.5 Introduction laws

lemma *urel-refine-ext*:
  ⟦ ⋀ *s s'*. *P*⟦≪*s*≫,≪*s'*≫/$**v**,$**v**´⟧ ⊑ *Q*⟦≪*s*≫,≪*s'*≫/$**v**,$**v**´⟧ ⟧ ⟹ *P* ⊑ *Q*
  by (*rel-auto*)

lemma *urel-eq-ext*:
  ⟦ ⋀ *s s'*. *P*⟦≪*s*≫,≪*s'*≫/$**v**,$**v**´⟧ = *Q*⟦≪*s*≫,≪*s'*≫/$**v**,$**v**´⟧ ⟧ ⟹ *P* = *Q*
  by (*rel-auto*)

16.6 Unrestriction Laws

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

lemma *unrest-ouvar* [*unrest*]: *inα* ♯ $*x*´
  by (*metis inα-def lift-post-var snd-fst-lens-indep unrest-aext-indep*)

lemma *unrest-semir-undash* [*unrest*]:
  fixes *x* :: (*'a* ⟹ *'α*)
  assumes $*x* ♯ *P*
  shows $*x* ♯ *P* ;; *Q*
  using *assms* **by** (*rel-auto*)

lemma *unrest-semir-dash* [*unrest*]:

fixes $x :: ('a \Longrightarrow 'α)$
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-lift-rcond* [*unrest*]:
$x \sharp \lceil b \rceil_{<} \Longrightarrow x \sharp \lceil b \rceil_{\leftarrow}$
by (*simp add*: *lift-rcond-def*)

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

lemma *unrest-outα-var* [*unrest*]:
$[\![mwb\text{-}lens\ x;\ out\alpha \sharp (P :: ('a, ('\alpha \times '\beta))\ uexpr)]\!] \Longrightarrow \$x´ \sharp P$
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*]:
$vwb\text{-}lens\ x \Longrightarrow \$x \sharp (P \upharpoonright_{\alpha} x)$
by (*simp add*: *rel-var-res-def unrest*)

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

lemma *unrest-out-alpha-usubst-rel-lift* [*unrest*]:
$out\alpha \sharp_s \lceil \sigma \rceil_s$
by (*rel-auto*)

lemma *unrest-in-rel-aext* [*unrest*]: $x \bowtie y \Longrightarrow \$y \sharp P \oplus_r x$
  by (*simp add*: *rel-aext-def unrest-aext-indep*)

lemma *unrest-out-rel-aext* [*unrest*]: $x \bowtie y \Longrightarrow \$y' \sharp P \oplus_r x$
  by (*simp add*: *rel-aext-def unrest-aext-indep*)

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

lemma *rel-aext-seq* [*alpha*]:
  $weak\text{-}lens\ a \Longrightarrow (P ;; Q) \oplus_r a = (P \oplus_r a ;; Q \oplus_r a)$
  apply (*rel-auto*)
  apply (*rename-tac aa b y*)
  apply (*rule-tac* $x{=}create_a\ y$ **in** *exI*)
  apply (*simp*)
  done

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

16.7 Substitution laws

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

lemma *subst-seq-right* [*usubst*]:
  $in\alpha \sharp_s \sigma \Longrightarrow \sigma \dagger (P ;; Q) = P ;; (\sigma \dagger Q)$
  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
  $\bigwedge P\ Q\ \sigma.\ \sigma(\$x \mapsto_s \textit{numeral } n) \dagger (P\ ;;\ Q) = \sigma \dagger (P[\![\textit{numeral } n/\$x]\!]\ ;;\ Q)$
  $\bigwedge P\ Q\ \sigma.\ \sigma(\$x\acute{} \mapsto_s \textit{numeral } n) \dagger (P\ ;;\ Q) = \sigma \dagger (P\ ;;\ Q[\![\textit{numeral } n/\$x\acute{}]\!])$
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*]:
$out\alpha \sharp_s \sigma \Longrightarrow \sigma \dagger II = \langle\lfloor\sigma\rfloor_s\rangle_a$
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 *subst-rel-lift-seq* [*usubst*]:
$\lceil\sigma\rceil_s \dagger (P\ ;;\ Q) = (\lceil\sigma\rceil_s \dagger P)\ ;;\ Q$
by (*rel-auto*)

lemma *subst-rel-lift-comp* [*usubst*]:
$\lceil\sigma\rceil_s \circ_s \lceil\varrho\rceil_s = \lceil\sigma \circ_s \varrho\rceil_s$
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 $\$x\acute{} \sharp_s \lceil P\rceil_s$
by *pred-simp*

lemma *subst-lift-cond* [*usubst*]: $\lceil\sigma\rceil_s \dagger \lceil s\rceil_\leftarrow = \lceil\sigma \dagger s\rceil_\leftarrow$

by (*rel-auto*)

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

16.8 Alphabet laws

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

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

lemma *rcond-lift-true* [*simp*]:
$\lceil \textit{true} \rceil_{\leftarrow} = \textit{true}$
by *rel-auto*

lemma *rcond-lift-false* [*simp*]:
$\lceil \textit{false} \rceil_{\leftarrow} = \textit{false}$
by *rel-auto*

lemma *rel-ares-aext* [*alpha*]:
$\textit{vwb-lens } a \Longrightarrow (P \oplus_r a) \upharpoonright_r a = P$
by (*rel-auto*)

lemma *rel-aext-ares* [*alpha*]:
$\{\$a, \$a'\} \natural P \Longrightarrow P \upharpoonright_r a \oplus_r a = P$
by (*rel-auto*)

lemma *rel-aext-uses* [*unrest*]:
$\textit{vwb-lens } a \Longrightarrow \{\$a, \$a'\} \natural (P \oplus_r a)$
by (*rel-auto*)

16.9 Framing

The following operator states that a relation only modifies variables within a.

abbreviation *modifies* :: *'s hrel* ⇒ (*'a* ⟹ *'s*) ⇒ *bool* **where**
modifies P a ≡ *P is frame a*

abbreviation *not-modifies* :: *'s hrel* ⇒ (*'a* ⟹ *'s*) ⇒ *bool* **where**
not-modifies P a ≡ *P is antiframe a*

syntax
-modifies :: *logic* ⇒ *salpha* ⇒ *logic* (**infix** *mods 30*)
-not-modifies :: *logic* ⇒ *salpha* ⇒ *logic* (**infix** *nmods 30*)

translations
-modifies P x == *CONST modifies P x*
-not-modifies P x == *CONST not-modifies P x*

lemma *mods-skip* [*closure*]:
vwb-lens a ⟹ *II mods a*
by (*rel-auto*)

lemma *mods-assigns* [*closure*]:
  $[\![$ *mwb-lens a*; $\sigma \rhd_s a = \sigma$ $]\!] \Longrightarrow \langle\sigma\rangle_a$ *mods a*
  by (*rel-auto*)

lemma *mods-disj* [*closure*]:
  assumes *P mods a Q mods a*
  shows $(P \vee Q)$ *mods a*
proof −
  have $(a{:}[P] \vee a{:}[Q])$ *mods a*
    by (*rel-auto*)
  thus *?thesis* **by** (*simp add*: *Healthy-if assms*)
qed

lemma *mods-cond* [*closure*]:
  assumes *P mods a Q mods a*
  shows $P \triangleleft b \triangleright_r Q$ *mods a*
proof −
  have $a{:}[P] \triangleleft b \triangleright_r a{:}[Q]$ *mods a*
    by (*rel-auto*)
  thus *?thesis* **by** (*simp add*: *Healthy-if assms*)
qed

lemma *mods-seq* [*closure*]:
  assumes *mwb-lens a P mods a Q mods a*
  shows *P* ;; *Q mods a*
proof −
  from *assms*(*1*) **have** $a{:}[P]$;; $a{:}[Q]$ *mods a*
    by (*rel-auto*, *metis mwb-lens.put-put*)
  thus *?thesis*
    by (*simp add*: *Healthy-if assms*)
qed

lemma *nmods-intro*:
  $[\![$ *vwb-lens x*; $\bigwedge v.\ x := \ll v \gg$;; $P = P$;; $x := \ll v \gg$ $]\!] \Longrightarrow$ *P nmods x*
  by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put wb-lens.put-twice*)

lemma *nmods-skip* [*closure*]: *vwb-lens a* $\Longrightarrow$ *II nmods a*
  by *rel-auto*

lemma *nmods-seq* [*closure*]:
  assumes *weak-lens a P nmods a Q nmods a*
  shows *P* ;; *Q nmods a*
  using *assms* **by** (*rel-auto'*, *metis weak-lens.put-get*)

lemma *nmods-cond* [*closure*]:
  assumes *P nmods a Q nmods a*
  shows $P \triangleleft b \triangleright_r Q$ *nmods a*
  using *assms* **by** (*rel-auto'*)

lemma *nmods-gcmd* [*closure*]: *P nmods a* $\Longrightarrow$ $(b \longrightarrow_r P)$ *nmods a*
  by (*rel-auto*)

lemma *nmods-choice* [*closure*]: $[\![$ *P nmods a*; *Q nmods a* $]\!] \Longrightarrow P \sqcap Q$ *nmods a*
  by (*rel-auto*)

lemma *nmods-assigns* [*closure*]:
  $[\![$ *vwb-lens x*; $x \sharp_s \sigma$ $]\!] \Longrightarrow \langle\sigma\rangle_a$ *nmods x*
  by (*rel-auto*, *metis vwb-lens.put-eq*)

lemma *nmods-assign* [*closure*]: $[\![$ *vwb-lens y*; $x \bowtie y$ $]\!] \Longrightarrow x := v$ *nmods y*
  by (*rel-auto*, *metis lens-indep.lens-put-comm vwb-lens-wb wb-lens.get-put*)

lemma *nmods-frext-comp* [*closure*]:$[\![$ *vwb-lens a*; *vwb-lens x*; *P nmods x* $]\!] \Longrightarrow a{:}[P]^+$ *nmods* $\&a{:}x$
  by (*rel-auto*, *metis lens-override-def lens-override-idem*)

lemma *nmods-frext-indep* [*closure*]:$[\![$ *vwb-lens a*; *vwb-lens x*; $x \bowtie a$ $]\!] \Longrightarrow a{:}[P]^+$ *nmods x*
  by (*rel-auto*, *metis lens-indep-get lens-override-def lens-override-idem*)

lemma *nmods-UINF* [*closure*]: $[\![\bigwedge v.\ P\ v$ *nmods x* $]\!] \Longrightarrow (\bigsqcap\ v \cdot P\ v)$ *nmods x*
  by (*rel-auto*)

lemma *nmods-guard* [*closure*]: *vwb-lens x* $\Longrightarrow$ *?*[*p*] *nmods x*
  by (*rel-auto*)

lemma *nmods-miracle* [*closure*]: *false nmods x*
  by *rel-auto*

lemma *nmods-disj* [*closure*]: $[\![$ *P nmods a*; *Q nmods a* $]\!] \Longrightarrow (P \vee Q)$ *nmods a*
  by (*rel-auto*)

no-utp-lift *rcond uassigns id seqr useq uskip rcond rassume rassert*
  frame antiframe modify freeze conv-r
  rgcmd while-top while-bot while-inv while-inv-bot while-vrt

end

17 Fixed-points and Recursion

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

17.1 Fixed-point Laws

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

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

lemma *nu-id*: $(\nu\ X \cdot X) = false$
  by (*meson lfp-lowerbound utp-pred-laws.bot.extremum-unique*)

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

lemma *mu-refine-intro*:

assumes $(C \Rightarrow S) \sqsubseteq F(C \Rightarrow S)$ $(C \wedge \mu\ F) = (C \wedge \nu\ F)$
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

17.2 Obtaining Unique Fixed-points

Obtaining termination proofs via approximation chains. Theorems and proofs adapted from Chapter 2, page 63 of the UTP book [22].

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

definition *chain* :: *'a chain* ⇒ *bool* **where**
chain Y = ((*Y 0* = *false*) ∧ (∀ *i*. *Y* (*Suc i*) ⊑ *Y i*))

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

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

lemma *chainE*:
assumes *chain Y* ⋀ *i*. ⟦ *Y 0* = *false*; *Y* (*Suc i*) ⊑ *Y i* ⟧ ⟹ *P*
shows *P*
using *assms* **by** (*simp add*: *chain-def*)

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

Constructive chains

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

lemma *constrI*:
assumes *chain E* ⋀ *X n*. ((*F*(*X*) ∧ *E*(*n* + *1*)) = (*F*(*X* ∧ *E*(*n*)) ∧ *E* (*n* + *1*)))
shows *constr F E*
using *assms* **by** (*auto simp add*: *constr-def*)

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

lemma *chain-pred-terminates*:
assumes *constr F E mono F*
shows (⨅ (*range E*) ∧ μ *F*) = (⨅ (*range E*) ∧ ν *F*)
proof −
from *assms* **have** ∀ *n*. (*E n* ∧ μ *F*) = (*E n* ∧ ν *F*)
proof (*rule-tac allI*)

```
    fix $n$
    from $assms$ show $(E\ n \wedge \mu\ F) = (E\ n \wedge \nu\ F)$
    proof $(induct\ n)$
      case $0$ thus $?case$ by $(simp\ add{:}\ constr\text{-}def)$
    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 $gfp\text{-}unfold[OF\ hyp(3),\ THEN\ sym]$ by $(simp\ add{:}\ constr\text{-}def)$
        also from $hyp$ have $... = (E\ (n + 1) \wedge F\ (E\ n \wedge \mu\ F))$
          by $(metis\ conj\text{-}comm\ constr\text{-}def)$
        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 $(metis\ (no\text{-}types,\ lifting)\ conj\text{-}comm\ constr\text{-}def\ lfp\text{-}unfold)$
        ultimately show $?thesis$
          by $simp$
      qed
    qed
  qed
  thus $?thesis$
    by $(auto\ intro{:}\ L274)$
qed
```

```
theorem $constr\text{-}fp\text{-}uniq$:
  assumes $constr\ F\ E\ mono\ F\ \bigsqcap\ (range\ E) = C$
  shows $(C \wedge \mu\ F) = (C \wedge \nu\ F)$
  using $assms(1)\ assms(2)\ assms(3)\ chain\text{-}pred\text{-}terminates$ by $blast$
```

17.3 Noetherian Induction Instantiation

Contribution from Yakoub Nemouchi.The following generalization was used by Tobias Nipkow and Peter Lammich in *Refine_Monadic*

```
lemma $wf\text{-}fixp\text{-}uinduct\text{-}pure\text{-}ueq\text{-}gen$:
  assumes $fixp\text{-}unfold{:}\ fp\ B = B\ (fp\ B)$
  and             $WF{:}\ wf\ R$
  and      $induct\text{-}step$:
          $\bigwedge f\ st.\ [\![\bigwedge st'.\ (st',st) \in R \implies (((pre \wedge \lceil e\rceil_< =_u \ll st'\gg) \Rightarrow post) \sqsubseteq f)]\!]$
              $\implies fp\ B = f \implies ((pre \wedge \lceil e\rceil_< =_u \ll st\gg) \Rightarrow post) \sqsubseteq (B\ f)$
        shows $((pre \Rightarrow post) \sqsubseteq fp\ B)$
proof $-$
  { fix $st$
    have $((pre \wedge \lceil e\rceil_< =_u \ll st\gg) \Rightarrow post) \sqsubseteq (fp\ B)$
    using $WF$ proof $(induction\ rule{:}\ wf\text{-}induct\text{-}rule)$
      case $(less\ x)$
      hence $(pre \wedge \lceil e\rceil_< =_u \ll x\gg \Rightarrow post) \sqsubseteq B\ (fp\ B)$
        by $(rule\ induct\text{-}step,\ rel\text{-}blast,\ simp)$
      then show $?case$
        using $fixp\text{-}unfold$ by $auto$
    qed
  }
  thus $?thesis$
  by $pred\text{-}simp$
```

qed

The next lemma shows that using substitution also work. However it is not that generic nor practical for proof automation ...

```
lemma refine-usubst-to-ueq:
  vwb-lens E ⟹ (pre ⇒ post)⟦«st'»/$E⟧ ⊑ f⟦«st'»/$E⟧ = (((pre ∧ $E =_u «st'») ⇒ post) ⊑ f)
  by (rel-auto, metis vwb-lens-wb wb-lens.get-put)
```

By instantiation of ⟦?fp ?B = ?B (?fp ?B); wf ?R; ⋀f st. ⟦⋀st'. (st', st) ∈ ?R ⟹ (?pre ∧ bop (=) (?e$^<$) **U**(st') ⇒ ?post) ⊑ f; ?fp ?B = f⟧ ⟹ (?pre ∧ bop (=) (?e$^<$) **U**(st) ⇒ ?post) ⊑ ?B f⟧ ⟹ (?pre ⇒ ?post) ⊑ ?fp ?B with μ and lifting of the well-founded relation we have ...

```
lemma mu-rec-total-pure-rule:
  assumes WF: wf R
  and     M: mono B
  and     induct-step:
          ⋀ f st.  ⟦(pre ∧ (⌈e⌉<,«st»)u ∈u «R» ⇒ post) ⊑ f⟧
              ⟹ μ B = f ⟹(pre ∧ ⌈e⌉< =u «st» ⇒ post) ⊑ (B f)
        shows (pre ⇒ post) ⊑ μ B
proof (rule wf-fixp-uinduct-pure-ueq-gen[where fp=μ and pre=pre and B=B and R=R and e=e])
  show μ B = B (μ B)
    by (simp add: M def-gfp-unfold)
  show wf R
    by (fact WF)
  show ⋀f st. (⋀st'. (st', st) ∈ R ⟹ (pre ∧ ⌈e⌉< =u «st'» ⇒ post) ⊑ f) ⟹
              μ B = f ⟹
              (pre ∧ ⌈e⌉< =u «st» ⇒ post) ⊑ B f
    by (rule induct-step, rel-simp, simp)
qed
```

```
lemma nu-rec-total-pure-rule:
  assumes WF: wf R
  and     M: mono B
  and     induct-step:
          ⋀ f st.  ⟦(pre ∧ (⌈e⌉<,«st»)u ∈u «R» ⇒ post) ⊑ f⟧
              ⟹ ν B = f ⟹(pre ∧ ⌈e⌉< =u «st» ⇒ post) ⊑ (B f)
        shows (pre ⇒ post) ⊑ ν B
proof (rule wf-fixp-uinduct-pure-ueq-gen[where fp=ν and pre=pre and B=B and R=R and e=e])
  show ν B = B (ν B)
    by (simp add: M def-lfp-unfold)
  show wf R
    by (fact WF)
  show ⋀f st. (⋀st'. (st', st) ∈ R ⟹ (pre ∧ ⌈e⌉< =u «st'» ⇒ post) ⊑ f) ⟹
              ν B = f ⟹
              (pre ∧ ⌈e⌉< =u «st» ⇒ post) ⊑ B f
    by (rule induct-step, rel-simp, simp)
qed
```

Since B **U**(pre ∧ (E$^<$, st) ∈ R ⇒ post) ⊑ B (μ B) and mono B, thus, ⟦wf ?R; Monotonic ?B; ⋀f st. ⟦(?pre ∧ bop (∈) (bop Pair (?e$^<$) **U**(st)) **U**(?R) ⇒ ?post) ⊑ f; μ ?B = f⟧ ⟹ (?pre ∧ bop (=) (?e$^<$) **U**(st) ⇒ ?post) ⊑ ?B f⟧ ⟹ (?pre ⇒ ?post) ⊑ μ ?B can be expressed as follows

```
lemma mu-rec-total-utp-rule:
  assumes WF: wf R
```

  and M: *mono B*
  and *induct-step*:
  $\bigwedge st.\ (pre \wedge \lceil e\rceil_< =_u \ll st\gg \Rightarrow post) \sqsubseteq (B\ ((pre \wedge (\lceil e\rceil_<,\ll st\gg)_u \in_u \ll R\gg \Rightarrow post)))$
 shows $(pre \Rightarrow post) \sqsubseteq \mu\ B$
proof (*rule mu-rec-total-pure-rule*[**where** R=R **and** e=e], *simp-all add*: *assms*)
 show $\bigwedge f\ st.\ (pre \wedge (\lceil e\rceil_<, \ll st\gg)_u \in_u \ll R\gg \Rightarrow post) \sqsubseteq f \Longrightarrow \mu\ B = f \Longrightarrow (pre \wedge \lceil e\rceil_< =_u \ll st\gg \Rightarrow post) \sqsubseteq B\ f$
  by (*simp add*: *M induct-step monoD order-subst2*)
qed

lemma *nu-rec-total-utp-rule*:
 assumes *WF*: *wf R*
  and M: *mono B*
  and *induct-step*:
  $\bigwedge st.\ (pre \wedge \lceil e\rceil_< =_u \ll st\gg \Rightarrow post) \sqsubseteq (B\ ((pre \wedge (\lceil e\rceil_<,\ll st\gg)_u \in_u \ll R\gg \Rightarrow post)))$
 shows $(pre \Rightarrow post) \sqsubseteq \nu\ B$
proof (*rule nu-rec-total-pure-rule*[**where** R=R **and** e=e], *simp-all add*: *assms*)
 show $\bigwedge f\ st.\ (pre \wedge (\lceil e\rceil_<, \ll st\gg)_u \in_u \ll R\gg \Rightarrow post) \sqsubseteq f \Longrightarrow \nu\ B = f \Longrightarrow (pre \wedge \lceil e\rceil_< =_u \ll st\gg \Rightarrow post) \sqsubseteq B\ f$
  by (*simp add*: *M induct-step monoD order-subst2*)
qed

end

18 Sequent Calculus

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

definition *sequent* :: $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* $\Rightarrow$ *bool* (**infixr** $\Vdash$ *15*) **where**
[*upred-defs*]: *sequent* $P\ Q = (Q \sqsubseteq P)$

utp-lift-notation *sequent*

abbreviation *sequent-triv* ($\Vdash$ - [*15*] *15*) **where** $\Vdash P \equiv (true \Vdash P)$

translations
 $\Vdash P$ <= *true* $\Vdash P$

Conversion of UTP sequent to Isabelle proposition

lemma *sequentI*: $[\![\bigwedge s.\ [\![\Gamma]\!]_e\ s \Longrightarrow [\![\varphi]\!]_e\ s]\!] \Longrightarrow \Gamma \Vdash \varphi$
 by (*rel-auto*)

lemma *sTrue*: $P \Vdash true$
 by *pred-auto*

lemma *sAx*: $P \Vdash P$
 by *pred-auto*

lemma *sNotI*: $\Gamma \wedge P \Vdash false \Longrightarrow \Gamma \Vdash \neg\ P$
 by *pred-auto*

lemma *sConjI*: $[\![\Gamma \Vdash P; \Gamma \Vdash Q]\!] \Longrightarrow \Gamma \Vdash P \wedge Q$
 by *pred-auto*

lemma *sImplI*: $[\![\ P \wedge \Gamma \Vdash Q\ ]\!] \Longrightarrow \Gamma \Vdash (P \Rightarrow Q)$
  by *pred-auto*

lemma *sAsmDisj*:
  $[\![\ A \Vdash C;\ B \Vdash C\ ]\!] \Longrightarrow A \vee B \Vdash C$
  by (*rel-auto*)

lemma *sDisjI1*: $P \Vdash Q \Longrightarrow P \Vdash (Q \vee R)$
  by (*rel-auto*)

lemma *sDisjI2*: $P \Vdash R \Longrightarrow P \Vdash (Q \vee R)$
  by (*rel-auto*)

lemma *sVarEqI*:
  assumes *wb-lens x* $(\&x = v \wedge P) \Vdash (Q[\![v/\&x]\!])$
  shows $(\&x = v \wedge P) \Vdash Q$
  using *assms* **by** (*rel-simp*, *metis wb-lens.get-put*)

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

lemma *sWk1*: $P \Vdash R \Longrightarrow P \wedge Q \Vdash R$
  by (*rel-auto*)

lemma *sWk2*: $Q \Vdash R \Longrightarrow P \wedge Q \Vdash R$
  by (*rel-auto*)

end

19 Relational Calculus Laws

theory *utp-rel-laws*
  imports
    utp-rel
    utp-recursion
    utp-lift-parser
begin

19.1 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*)

Alternative expression of conditional using assumptions and choice

lemma *rcond-rassume-expand*: $P \triangleleft b \triangleright_r Q = ([b]^\top \ ;;\ P) \sqcap ([(\neg\ b)]^\top \ ;;\ Q)$

by (*rel-auto*)

19.2 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*)

19.3 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 *Monotonic-seqr-tail* [*closure*]:
assumes *Monotonic F*
shows *Monotonic* $(\lambda\ X.\ P\ ;;\ F(X))$
by (*simp add*: *assms monoD monoI seqr-mono*)

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

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

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

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

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

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

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

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

lemma *seqr-unfold*:
$(P\ ;;\ Q) = (\exists\ v \cdot P\llbracket \ll v \gg / \$\mathbf{v}' \rrbracket \wedge Q\llbracket \ll v \gg / \$\mathbf{v} \rrbracket)$
by (*rel-auto*)

lemma *seqr-unfold-heterogeneous*:
$(P\ ;;\ Q) = (\exists\ v \cdot (Pre(P\llbracket \ll v \gg / \$\mathbf{v}' \rrbracket))^{<} \wedge (Post(Q\llbracket \ll v \gg / \$\mathbf{v} \rrbracket))^{>})$
by (*rel-auto*)

lemma *seqr-middle*:
assumes *vwb-lens x*
shows $(P\ ;;\ Q) = (\exists\ v \cdot P\llbracket \ll v \gg / \$x' \rrbracket\ ;;\ Q\llbracket \ll v \gg / \$x \rrbracket)$
using *assms*
by (*rel-auto'*, *metis vwb-lens-wb wb-lens.source-stability*)

lemma *seqr-left-one-point*:

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

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

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

lemma *cond-inter-var-split*:

assumes *vwb-lens x*
shows $(P \triangleleft \$x' \triangleright Q)$;; $R = (P⟦true/\$x'⟧$;; $R⟦true/\$x⟧ \vee Q⟦false/\$x'⟧$;; $R⟦false/\$x⟧)$
proof –
have $(P \triangleleft \$x' \triangleright Q)$;; $R = ((\$x' \wedge P)$;; $R \vee (\neg \$x' \wedge Q)$;; $R)$
by (*simp add*: *cond-def seqr-or-distl*)
also have ... = $((P \wedge \$x')$;; $R \vee (Q \wedge \neg\$x')$;; $R)$
by (*rel-auto*)
also have ... = $(P⟦true/\$x'⟧$;; $R⟦true/\$x⟧ \vee Q⟦false/\$x'⟧$;; $R⟦false/\$x⟧)$
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 $(P$;; $(Q \wedge \$x')) = ((P$;; $Q) \wedge \$x')$
by (*rel-auto*)

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

lemma *shEx-mem-lift-seq-1* [*uquant-lift*]:
assumes $out\alpha \sharp A$
shows $((\exists\ x \in A \cdot P\ x)$;; $Q) = (\exists\ x \in A \cdot (P\ x$;; $Q))$
using *assms* **by** *rel-blast*

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

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

lemma *shEx-mem-lift-seq-2* [*uquant-lift*]:
  assumes $in\alpha \sharp A$
  shows $(P\ ;;\ (\exists\ x \in A \cdot Q\ x)) = (\exists\ x \in A \cdot (P\ ;;\ Q\ x))$
  using *assms* **by** *rel-blast*

19.4 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*)

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

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

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

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

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

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

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

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

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

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

19.7 Assignment Laws

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

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

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

lemma *assign-subst* [*usubst*]:
$[\![$ *mwb-lens x*; *mwb-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 *assign-vacuous-skip*:
assumes *vwb-lens x*
shows (x := &x) = *II*
using *assms* **by** *rel-auto*

The following law shows the case for the above law when x is only mainly-well behaved. We require that the state is one of those in which x is well defined using and assumption.

lemma *assign-vacuous-assume*:
assumes *mwb-lens x*
shows $[\&\mathbf{v} \in \ll\mathcal{S}_x\gg]^\top$;; (x := &x) = $[\&\mathbf{v} \in \ll\mathcal{S}_x\gg]^\top$
using *assms* **by** *rel-auto*

lemma *assign-simultaneous*:
assumes *vwb-lens y x* $\bowtie$ *y*
shows (x,y) := (e, &y) = (x := e)
by (*simp add*: *assms usubst-upd-comm usubst-upd-var-id*)

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

lemma *assigns-comp*: $(\langle f\rangle_a$;; $\langle g\rangle_a) = \langle g \circ_s f\rangle_a$
  by (*rel-auto*)

lemma *assigns-cond*: $(\langle f\rangle_a \triangleleft b \triangleright_r \langle g\rangle_a) = \langle f \triangleleft b \triangleright g\rangle_a$
  by (*rel-auto*)

lemma *assigns-r-conv*:
  $bij_s\ f \Longrightarrow \langle f\rangle_a{}^- = \langle inv_s\ f\rangle_a$
  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 $\$x \sharp b$ $out\alpha \sharp b$
  shows $(b \wedge x := v) = (x := v \wedge b^-)$
  using *assms* **by** (*rel-blast*)

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

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

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

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 $out\alpha \sharp b$
  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*: $inj_s\ f \Longrightarrow uinj\ \langle f\rangle_a$
  by (*rel-simp*, *simp add*: *inj-eq*)

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

by (*metis assigns-r-uinj pr-var-def swap-usubst-inj*)

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

19.8 Non-deterministic Assignment Laws

lemma *nd-assign-comp*:
  $x \bowtie y \Longrightarrow x := * ;; y := * = x,y := *$
  apply (*rel-auto*) **using** *lens-indep-comm* **by** *fastforce+*

lemma *nd-assign-assign*:
  $[\![\ vwb\text{-}lens\ x;\ x \sharp e\ ]\!] \Longrightarrow x := * ;; x := e = x := e$
  by (*rel-auto*)

19.9 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 $(\$x)^- = \$x\acute{}$
  by *pred-auto*

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

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

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

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

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

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

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

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

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

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

19.10 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*)

19.11 Frame and Antiframe Laws

named-theorems *frame*

lemma *frame-all* [*frame*]: $\Sigma{:}[P] = P$
  by (*rel-auto*)

lemma *frame-none* [*frame*]:
  $\emptyset{:}[P] = (P \wedge II)$
  by (*rel-auto*)

lemma *frame-commute*:
  assumes $\$y \sharp P$ $\$y\acute{} \sharp P$ $\$x \sharp Q$ $\$x\acute{} \sharp Q$ $x \bowtie y$
  shows $x{:}[P] ;; y{:}[Q] = y{:}[Q] ;; x{:}[P]$
  apply (*insert assms*)
  apply (*rel-auto*)
   apply (*rename-tac* $s\ s'\ s_0$)
   apply (*subgoal-tac* $(s \oplus_L s'$ *on* $y) \oplus_L s_0$ *on* $x = s_0 \oplus_L s'$ *on* y)
    apply (*metis lens-indep-get lens-indep-sym lens-override-def*)
   apply (*simp add*: *lens-indep.lens-put-comm lens-override-def*)
  apply (*rename-tac* $s\ s'\ s_0$)
  apply (*subgoal-tac* $put_y\ (put_x\ s\ (get_x\ (put_x\ s_0\ (get_x\ s'))))\ (get_y\ (put_y\ s\ (get_y\ s_0)))$
                    $= put_x\ s_0\ (get_x\ s')$)
   apply (*metis lens-indep-get lens-indep-sym*)
  apply (*metis lens-indep.lens-put-comm*)
  done

lemma *frame-miracle* [*simp*]:

x:[*false*] = *false*
by (*rel-auto*)

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

lemma *frame-assign-in* [*frame*]:
$\llbracket$ *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 *frame-conj-true* [*frame*]:
$\llbracket$ {\$$x$,\$$x'$} $\natural$ P; *vwb-lens* x $\rrbracket \Longrightarrow$ ($P \wedge x$:[*true*]) = x:[P]
by (*rel-auto*)

lemma *frame-is-assign* [*frame*]:
vwb-lens $x \Longrightarrow x$:[\$$x' =_u \lceil v \rceil_<$] = $x := v$
by (*rel-auto*)

lemma *frame-seq* [*frame*]:
$\llbracket$ *vwb-lens* x; {\$$x$,\$$x'$} $\natural$ P; {\$$x$,\$$x'$} $\natural$ Q $\rrbracket \Longrightarrow x$:[$P$;; Q] = x:[P] ;; x:[Q]
apply (*rel-auto*)
 apply (*metis mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens-def weak-lens.put-get*)
apply (*metis mwb-lens.put-put vwb-lens-mwb*)
done

lemma *frame-assign-commute-unrest*:
assumes *vwb-lens* x $x \bowtie a$ $a \sharp v$ \$$x \sharp P$ \$$x' \sharp P$
shows $x := v$;; a:[P] = a:[P] ;; $x := v$
using *assms*
apply (*rel-auto*)
apply (*metis* (*no-types*, *lifting*) *lens-indep.lens-put-irr2 lens-indep-comm*)
apply (*metis* (*no-types*, *hide-lams*) *lens-indep-def*)
done

lemma *frame-to-antiframe* [*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 *rel-frext-miracle* [*frame*]:
a:[*false*]$^+$ = *false*
by (*rel-auto*)

lemma *rel-frext-skip* [*frame*]:
vwb-lens $a \Longrightarrow a$:[*II*]$^+$ = *II*
by (*rel-auto*)

lemma *rel-frext-seq* [*frame*]:
vwb-lens $a \Longrightarrow a$:[P ;; Q]$^+$ = (a:[P]$^+$;; a:[Q]$^+$)
apply (*rel-auto*)
 apply (*rename-tac* s s' s_0)
 apply (*rule-tac* x=put_a s s_0 **in** *exI*)
 apply (*auto*)
apply (*metis mwb-lens.put-put vwb-lens-mwb*)
done

lemma *rel-frext-assigns* [*frame*]:
  vwb-lens a $\Longrightarrow$ a:$[\langle\sigma\rangle_a]^+ = \langle\sigma \oplus_s a\rangle_a$
  by (*rel-auto*)

lemma *rel-frext-rcond* [*frame*]:
  a:$[P \triangleleft b \triangleright_r Q]^+ = (a$:$[P]^+ \triangleleft b \oplus_p a \triangleright_r a$:$[Q]^+)$
  by (*rel-auto*)

lemma *rel-frext-commute*:
  $x \bowtie y \Longrightarrow x$:$[P]^+$;; y:$[Q]^+ = y$:$[Q]^+$;; x:$[P]^+$
  apply (*rel-auto*)
   apply (*rename-tac a c b*)
   apply (*subgoal-tac* $\bigwedge b\ a.\ get_y\ (put_x\ b\ a) = get_y\ b$)
    apply (*metis* (*no-types*, *hide-lams*) *lens-indep-comm lens-indep-get*)
   apply (*simp add*: *lens-indep.lens-put-irr2*)
  apply (*subgoal-tac* $\bigwedge b\ c.\ get_x\ (put_y\ b\ c) = get_x\ b$)
   apply (*subgoal-tac* $\bigwedge b\ a.\ get_y\ (put_x\ b\ a) = get_y\ b$)
    apply (*metis* (*mono-tags*, *lifting*) *lens-indep-comm*)
   apply (*simp-all add*: *lens-indep.lens-put-irr2*)
  done

lemma *antiframe-disj* [*frame*]: $(x$:$[\![P]\!] \vee x$:$[\![Q]\!]) = x$:$[\![P \vee Q]\!]$
  by (*rel-auto*)

lemma *antiframe-seq* [*frame*]:
  $[\![$ *vwb-lens x*; $\$x\acute{}\ \sharp\ P$; $\$x\ \sharp\ Q$ $]\!]$ $\Longrightarrow$ $(x$:$[\![P]\!]$;; x:$[\![Q]\!]) = x$:$[\![P$;; $Q]\!]$
  apply (*rel-auto*)
   apply (*metis vwb-lens-wb wb-lens-def weak-lens.put-get*)
  apply (*metis vwb-lens-wb wb-lens.put-twice wb-lens-def weak-lens.put-get*)
  done

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*]

19.12 Modify and Freeze Laws

Assignments made to modify variables are retained, but lost for frozen ones.

lemma *modify-assigns*: (*mdf a* $\cdot$ $\langle\sigma\rangle_a) = \langle\sigma \rhd_s a\rangle_a$
  by (*rel-auto*)

lemma *modify-assign*:
  vwb-lens x $\Longrightarrow$ (*mdf x* $\cdot$ $x := v) = x := v$
  by (*simp add*: *modify-assigns usubst*)

lemma *freeze-assigns*: (*frz a* $\cdot$ $\langle\sigma\rangle_a) = \langle\sigma -_s a\rangle_a$
  by (*rel-auto*)

lemma *freeze-assign*:
  vwb-lens x $\Longrightarrow$ (*frz x* $\cdot$ $x := v$) = *II*

by (*simp add*: *freeze-assigns usubst skip-r-def*)

lemma *frame-modify-same-fixpoints*:
mwb-lens a $\Longrightarrow$ *P mods a* $\longleftrightarrow$ *P is modify a*
by (*rel-simp*, *metis mwb-lens-weak weak-lens-def*)

lemma *antiframe-freeze-same-fixpoints*:
mwb-lens a $\Longrightarrow$ *P is antiframe a* $\longleftrightarrow$ *P is freeze a*
by (*rel-simp*, *metis mwb-lens.put-put*)

19.13 While Loop Laws

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

theorem *while-false*: *while false do P od* = *II*
by (*subst while-unfold*, *rel-auto*)

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

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

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

theorem *while-bot-true*: $\mathit{while}_\bot\ \mathit{true}\ \mathit{do}\ P\ \mathit{od} = (\mu\ X \cdot P \;;;\; X)$
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

19.14 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*)+)

declare *upred-semiring.power-Suc* [*simp del*]

We introduce the power syntax derived from semirings

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

translations
  P ^ *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*:
  $[\![P$ ^ $i]\!]_e = (\lambda\ b.\ b \in (\{p.\ [\![P]\!]_e\ p\}$ ^^ $i))$
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 upred-semiring.power-Suc*)
qed

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

update-uexpr-rep-eq-thms

lemma *Sup-power-expand*:
  fixes *P* :: *nat* $\Rightarrow$ *'a::complete-lattice*
  shows $P(0) \sqcap (\bigsqcap i.\ P(i+1)) = (\bigsqcap i.\ P(i))$
proof –
  have *UNIV* = *insert* (*0::nat*) {*1..*}
    by *auto*
  moreover have $(\bigsqcap i.\ P(i)) = \bigsqcap\ (P$ ` *UNIV*)
    by (*blast*)
  moreover have $\bigsqcap\ (P$ ` *insert 0* {*1..*}) = $P(0) \sqcap$ *SUPREMUM* {*1..*} *P*
    by (*simp*)
  moreover have *SUPREMUM* {*1..*} $P = (\bigsqcap i.\ P(i+1))$

```
    by (simp add: atLeast-Suc-greaterThan greaterThan-0)
  ultimately show ?thesis
    by (simp only:)
qed

lemma Sup-upto-Suc: (⨅i∈{0..Suc n}. P ^ i) = (⨅i∈{0..n}. P ^ i) ⊓ P ^ Suc n
proof −
  have (⨅i∈{0..Suc n}. P ^ i) = (⨅i∈insert (Suc n) {0..n}. P ^ i)
    by (simp add: atLeast0-atMost-Suc)
  also have ... = P ^ Suc n ⊓ (⨅i∈{0..n}. P ^ 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 ⊑ ((P ;; Q) ⊓ R) ⟹ Q ⊑ P ^ n ;; R
proof (induct n)
  case 0
  then show ?case by (auto)
next
  case (Suc n)
  then show ?case
   by (auto simp add: upred-semiring.power-Suc, metis (no-types, hide-lams) dual-order.trans order-refl
seqr-assoc seqr-mono)
qed

lemma upower-inductr:
  assumes Q ⊑ R ⊓ (Q ;; P)
  shows Q ⊑ R ;; (P ^ n)
using assms proof (induct n)
  case 0
  then show ?case by auto
next
  case (Suc n)
  have R ;; P ^ Suc n = (R ;; P ^ n) ;; P
    by (metis seqr-assoc upred-semiring.power-Suc2)
  also have Q ;; P ⊑ ...
    by (meson Suc.hyps assms eq-iff seqr-mono)
  also have Q ⊑ ...
    using assms by auto
  finally show ?case .
qed

lemma SUP-atLeastAtMost-first:
  fixes P :: nat ⇒ 'a::complete-lattice
  assumes m ≤ n
  shows (⨅i∈{m..n}. P(i)) = P(m) ⊓ (⨅i∈{Suc m..n}. P(i))
  by (metis SUP-insert assms atLeastAtMost-insertL)

lemma upower-seqr-iter: P ^ n = (;; Q : replicate n P · Q)
  by (induct n, simp-all add: upred-semiring.power-Suc)

lemma assigns-power: ⟨f⟩ₐ ^ n = ⟨f ^ₛ n⟩ₐ
  by (induct n, rel-auto+)
```

19.15 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

19.16 Kleene Plus

purge-notation *trancl* ((-^+) [*1000*] *999*)

definition *uplus* :: $'\alpha$ *hrel* $\Rightarrow$ $'\alpha$ *hrel* (-^+ [*999*] *999*) **where**
[*upred-defs*]: $P^+ = P\ ;;\ P^\star$

lemma *uplus-power-def*: $P^+ = (\bigsqcap\ i \cdot P\ \hat{}\ (Suc\ i))$
  by (*simp add*: *uplus-def ustar-def seq-UINF-distl′ UINF-atLeast-Suc upred-semiring.power-Suc*)

19.17 Omega

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

19.18 Relation Algebra Laws

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

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*)

19.19 Kleene Algebra Laws

lemma *ustar-alt-def*: $P^\star = (\bigsqcap\ i \cdot P\ \hat{}\ i)$
  by (*simp add*: *ustar-def*)

theorem *ustar-sub-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 ⊑ R Q ⊑ P ;; Q
  shows Q ⊑ P⋆ ;; R
proof −
  have P⋆ ;; R = (⨅ i. P ˆ i ;; R)
    by (simp add: ustar-def UINF-as-Sup-collect′ seq-SUP-distr)
  also have Q ⊑ ...
    by (simp add: SUP-least assms upower-inductl)
  finally show ?thesis .
qed

theorem ustar-inductr:
  assumes Q ⊑ R Q ⊑ Q ;; P
  shows Q ⊑ R ;; P⋆
proof −
  have R ;; P⋆ = (⨅ i. R ;; P ˆ i)
    by (simp add: ustar-def UINF-as-Sup-collect′ seq-SUP-distl)
  also have Q ⊑ ...
    by (simp add: SUP-least assms upower-inductr)
  finally show ?thesis .
qed

lemma ustar-refines-nu: (ν X · (P ;; X) ⊓ II) ⊑ P⋆
  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⋆ = (ν X · (P ;; X) ⊓ II)
proof (rule antisym)
  show (ν X · (P ;; X) ⊓ II) ⊑ P⋆
    by (simp add: ustar-refines-nu)
  show P⋆ ⊑ (ν X · (P ;; X) ⊓ II)
    by (metis lfp-lowerbound upred-semiring.add-commute ustar-sub-unfoldl)
qed

lemma ustar-unfoldl: P⋆ = II ⊓ (P ;; P⋆)
  apply (simp add: ustar-as-nu)
  apply (subst lfp-unfold)
   apply (rule monoI)
   apply (rel-auto)+
  done
```

While loop can be expressed using Kleene star

```
lemma while-star-form:
  while b do P od = (P ◁ b ▷ᵣ II)⋆ ;; [(¬b)]⊤
proof −
  have 1: Continuous (λX. P ;; X ◁ b ▷ᵣ II)
    by (rel-auto)
  have while b do P od = (⨅i. ((λX. P ;; X ◁ b ▷ᵣ II) ˆˆ i) false)
    by (simp add: 1 false-upred-def sup-continuous-Continuous sup-continuous-lfp while-top-def)
  also have ... = ((λX. P ;; X ◁ b ▷ᵣ II) ˆˆ 0) false ⊓ (⨅i. ((λX. P ;; X ◁ b ▷ᵣ II) ˆˆ (i+1)) false)
    by (subst Sup-power-expand, simp)
  also have ... = (⨅i. ((λX. P ;; X ◁ b ▷ᵣ II) ˆˆ (i+1)) false)
    by (simp)
  also have ... = (⨅i. (P ◁ b ▷ᵣ II)ˆi ;; (false ◁ b ▷ᵣ II))
```

  proof (*rule SUP-cong*, *simp-all*)
    fix *i*
    show P ;; $((\lambda X.\ P$;; $X \triangleleft b \triangleright_r II)$ ^^ $i)$ *false* $\triangleleft b \triangleright_r II = (P \triangleleft b \triangleright_r II)$ ^ i ;; $(\mathit{false} \triangleleft b \triangleright_r II)$
    proof (*induct i*)
      case *0*
      then show *?case* **by** *simp*
    next
      case (*Suc i*)
      then show *?case*
        by (*simp add*: *upred-semiring.power-Suc*)
          (*metis* (*no-types*, *lifting*) *RA1 comp-cond-left-distr cond-L6 upred-semiring.mult.left-neutral*)
    qed
  qed
  also have ... = $(\bigsqcap i \in \{0..\} \cdot (P \triangleleft b \triangleright_r II)\hat{}i$;; $[(\neg b)]^\top)$
    by (*rel-auto*)
  also have ... = $(P \triangleleft b \triangleright_r II)^\star$;; $[(\neg b)]^\top$
    by (*metis seq-UINF-distr ustar-def*)
  finally show *?thesis* **.**
qed

19.20 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*)

19.21 Refinement Laws

lemma *skip-r-refine*:
  $(p \Rightarrow p) \sqsubseteq II$
  by *pred-blast*

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

lemma *pre-weak-rel*:
  assumes *‘pre* $\Rightarrow$ *I‘*
  and $(I \Rightarrow \mathit{post}) \sqsubseteq P$
  shows $(\mathit{pre} \Rightarrow \mathit{post}) \sqsubseteq P$
  using *assms* **by**(*rel-auto*)

lemma *cond-refine-rel*:
  assumes $S \sqsubseteq (\lceil b \rceil_< \wedge P)$ $S \sqsubseteq (\lceil \neg b \rceil_< \wedge Q)$
  shows $S \sqsubseteq P \triangleleft b \triangleright_r Q$
  by (*metis aext-not assms*(*1*) *assms*(*2*) *cond-def lift-rcond-def utp-pred-laws.le-sup-iff*)

lemma *seq-refine-pred*:
  assumes $(\lceil b \rceil_< \Rightarrow \lceil s \rceil_>) \sqsubseteq P$ **and** $(\lceil s \rceil_< \Rightarrow \lceil c \rceil_>) \sqsubseteq Q$
  shows $(\lceil b \rceil_< \Rightarrow \lceil c \rceil_>) \sqsubseteq (P$;; $Q)$
  using *assms* **by** *rel-auto*

lemma *seq-refine-unrest*:
  assumes $out\alpha \sharp b$ $in\alpha \sharp c$
  assumes $(b \Rightarrow \lceil s \rceil_>) \sqsubseteq P$ **and** $(\lceil s \rceil_< \Rightarrow c) \sqsubseteq Q$
  shows $(b \Rightarrow c) \sqsubseteq (P$;; $Q)$

using *assms* **by** *rel-blast*

19.22 Preain and Postge Laws

named-theorems *prepost*

lemma *Pre-conv-Post* [*prepost*]:
$Pre(P^-) = Post(P)$
by (*rel-auto*)

lemma *Post-conv-Pre* [*prepost*]:
$Post(P^-) = Pre(P)$
by (*rel-auto*)

lemma *Pre-skip* [*prepost*]:
$Pre(II) = true$
by (*rel-auto*)

lemma *Pre-assigns* [*prepost*]:
$Pre(\langle\sigma\rangle_a) = true$
by (*rel-auto*)

lemma *Pre-miracle* [*prepost*]:
$Pre(false) = false$
by (*rel-auto*)

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

lemma *Pre-seq*:
$Pre(P\ ;;\ Q) = Pre(P\ ;;\ [Pre(Q)]^\top)$
by (*rel-auto*)

lemma *Pre-disj* [*prepost*]:
$Pre(P \vee Q) = (Pre(P) \vee Pre(Q))$
by (*rel-auto*)

lemma *Pre-inf* [*prepost*]:
$Pre(P \sqcap Q) = (Pre(P) \vee Pre(Q))$
by (*rel-auto*)

lemma *Pre-conj-rel-aext* [*prepost*]:
$[\![$ *vwb-lens a*; *vwb-lens b*; $a \bowtie b$ $]\!] \Longrightarrow Pre(P \oplus_r a \wedge Q \oplus_r b) = (Pre(P \oplus_r a) \wedge Pre(Q \oplus_r b))$
by (*rel-auto*, *metis* (*no-types*, *lifting*) *lens-indep-def mwb-lens-def vwb-lens-mwb weak-lens-def*)

If P uses on the variables in a and Q does not refer to the variables of $\boldsymbol{U}(\$a')$ then we can distribute.

lemma *Pre-conj-indep* [*prepost*]: $[\![$ $\{\$a,\$a'\} \natural P$; $\$a' \sharp Q$; *vwb-lens a* $]\!] \Longrightarrow Pre(P \wedge Q) = (Pre(P) \wedge Pre(Q))$
by (*rel-auto*, *metis lens-override-def lens-override-idem*)

lemma *assume-Pre* [*prepost*]:
$[Pre(P)]^\top\ ;;\ P = P$
by (*rel-auto*)

end

20 UTP Theories

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

Here, we mechanise a representation of UTP theories using locales [4]. We also link them to the HOL-Algebra library [5], which allows us to import properties from complete lattices and Galois connections.

20.1 Complete lattice of predicates

definition *upred-lattice* :: ($'\alpha$ *upred*) *gorder* ($\mathcal{P}$) **where**
upred-lattice = (| *carrier* = *UNIV*, *eq* = (=), *le* = ($\sqsubseteq$) |)

$\mathcal{P}$ is the complete lattice of alphabetised predicates. All other theories will be defined relative to it.

```
interpretation upred-lattice: complete-lattice 𝒫
proof (unfold-locales, simp-all add: upred-lattice-def)
  fix A :: 'α upred set
  show ∃ s. is-lub (|carrier = UNIV, eq = (=), le = (⊑)|) s A
    apply (rule-tac x=⨆ A in exI)
    apply (rule least-UpperI)
       apply (auto intro: Inf-greatest simp add: Inf-lower Upper-def)
    done
  show ∃ i. is-glb (|carrier = UNIV, eq = (=), le = (⊑)|) i A
    apply (rule-tac x=⨅ 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 𝒫
  by (simp add: upred-lattice.weak.weak-complete-lattice-axioms)

lemma upred-lattice-eq [simp]:
  (.=𝒫) = (=)
  by (simp add: upred-lattice-def)

lemma upred-lattice-le [simp]:
  le 𝒫 P Q = (P ⊑ Q)
  by (simp add: upred-lattice-def)

lemma upred-lattice-carrier [simp]:
  carrier 𝒫 = UNIV
  by (simp add: upred-lattice-def)

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

lemma upred-lattice-Idempotent [simp]: Idem𝒫 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*)

20.2 UTP theories hierarchy

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

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*) = (=)
by (*simp add*: *utp-order-def*)

lemma *utp-order-le* [*simp*]:
le (*utp-order T*) = ($\sqsubseteq$)
by (*simp add*: *utp-order-def*)

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

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

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

lemma *isotone-utp-orderI*: *Monotonic H* $\Longrightarrow$ *isotone* (*utp-order X*) (*utp-order Y*) *H*
by (*auto simp add*: *mono-def isotone-def utp-weak-partial-order*)

lemma *Mono-utp-orderI*:
$[\![\bigwedge P\ Q.\ [\![P \sqsubseteq Q;\ P\ is\ H;\ Q\ is\ H]\!] \Longrightarrow F(P) \sqsubseteq F(Q)]\!] \Longrightarrow Mono_{utp\text{-}order\ H}\ F$
by (*auto simp add*: *isotone-def utp-weak-partial-order*)

The UTP order can equivalently be characterised as the fixed point lattice, *fpl*.

lemma *utp-order-fpl*: *utp-order H* = *fpl* $\mathcal{P}$ *H*
by (*auto simp add*: *utp-order-def upred-lattice-def fps-def Healthy-def*)

20.3 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 *hcond* :: $'\alpha$ *upred* $\Rightarrow$ $'\alpha$ *upred* ($\mathcal{H}$)

```
  assumes HCond-Idem: ℋ(ℋ(P)) = ℋ(P)
begin

  abbreviation thy-order :: 'α upred gorder where
  thy-order ≡ utp-order ℋ

  abbreviation umono ≡ Mono_thy-order

  lemma HCond-Idempotent [closure,intro]: Idempotent ℋ
    by (simp add: Idempotent-def HCond-Idem)

  sublocale utp-po: partial-order utp-order ℋ
    by (unfold-locales, simp-all add: utp-order-def)
```

We need to remove some transitivity rules to stop them being applied in calculations

```
  declare utp-po.trans [trans del]

  lemma refine-monoE:
    assumes umono F x is ℋ y is ℋ x ⊑ y
    shows (x is ℋ ⟹ y is ℋ ⟹ F x ⊑ F y ⟹ thesis) ⟹ thesis
    using assms by (simp add: isotone-def)

end

locale utp-theory-lattice = utp-theory +
  assumes uthy-lattice: complete-lattice (utp-order ℋ)
begin

sublocale complete-lattice utp-order ℋ
  rewrites le thy-order = (⊑)
  and eq thy-order = (=)
  and ⋀ A. A ⊆ carrier thy-order ⟷ A ⊆ ⟦ℋ⟧_H
  and ⋀ P. P ∈ carrier thy-order ⟷ P is ℋ
  and carrier thy-order → carrier thy-order = ⟦ℋ⟧_H → ⟦ℋ⟧_H
  and Lattice.sup thy-order (carrier thy-order) = Lattice.sup thy-order ⟦ℋ⟧_H
  and Lattice.inf thy-order (carrier thy-order) = Lattice.inf thy-order ⟦ℋ⟧_H
  by (simp-all add: uthy-lattice)

declare top-closed [simp del]
declare bottom-closed [simp del]
```

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 [5], such as the Knaster-Tarski theorem. We can also retrieve lattice operators as below.

```
abbreviation utp-top (⊤)
where utp-top ≡ top (utp-order ℋ)

abbreviation utp-bottom (⊥)
where utp-bottom ≡ bottom (utp-order ℋ)

abbreviation utp-join (infixl ⊔ 65) where
utp-join ≡ join (utp-order ℋ)

abbreviation utp-meet (infixl ⊓ 70) where
utp-meet ≡ meet (utp-order ℋ)
```

abbreviation *utp-sup* ($\bigsqcup$- [*90*] *90*) **where**
utp-sup ≡ *Lattice.sup* (*utp-order* $\mathcal{H}$)

abbreviation *utp-inf* ($\bigsqcap$- [*90*] *90*) **where**
utp-inf ≡ *Lattice.inf* (*utp-order* $\mathcal{H}$)

abbreviation *utp-gfp* ($\boldsymbol{\nu}$) **where**
utp-gfp ≡ *GREATEST-FP* (*utp-order* $\mathcal{H}$)

abbreviation *utp-lfp* ($\boldsymbol{\mu}$) **where**
utp-lfp ≡ *LEAST-FP* (*utp-order* $\mathcal{H}$)

The following theorem and proof was contributed by Yakoub Nemouchi.

```
lemma lfp-ordinal-induct [case-names M H step union]:
  assumes M:‹Mono_thy-order F›
  assumes H:‹F ∈ ⟦𝓗⟧_H → ⟦𝓗⟧_H›
  assumes P-f:‹⋀S. P S ⟹ S ⊑ μ F ⟹ S is 𝓗 ⟹ P (F S)›
  assumes P-Union:‹⋀M. M ⊆ ⟦𝓗⟧_H ⟹ (⋀S. S ∈ M ⟹ P S) ⟹ P (⨆ M)›
  shows ‹P (μ F)›
proof −
  let ?M = ‹{S. S ⊑ μ F ∧ P S ∧ (S is 𝓗)}›
  from P-Union have ‹P (⨆ ?M)›
    by (metis (no-types, lifting) Collect-mono mem-Collect-eq)
  also have ‹⨆ ?M = μ F›
  proof (rule antisym)
    show ‹⨆ ?M ⊑ μ F›
      by (subst sup-least, auto simp add: Collect-mono)
    then have ‹F (⨆ ?M) ⊑ F (μ F)›
      by (metis (mono-tags, lifting) Collect-mono LFP-closed M sup-closed refine-monoE)
    then have ‹F (⨆ ?M) ⊑ μ F›
      by (metis (no-types, lifting) H LFP-weak-unfold M)
    then have ‹F (⨆ ?M) ∈ ?M›
      using P-Union
      apply simp
      apply (subst P-f)
         apply simp-all
         apply (simp add: calculation)
        apply (simp add: ‹⨆ ?M ⊑ μ F›)
       apply (simp add: Collect-mono-iff)
      using H
      apply (elim PiE)
       apply simp-all
      apply (simp add: Collect-mono)
      done
    then have F (⨆ ?M)  ⊑ ⨆ ?M
      by (simp add: Collect-mono sup-upper)
    then show μ F  ⊑ ⨆ ?M
      by (simp add: Collect-mono LFP-lowerbound)
  qed
  finally show ?thesis .
qed

end
```

syntax
  -tmu :: *logic* ⇒ *pttrn* ⇒ *logic* ⇒ *logic* ($\boldsymbol{\mu}$1 - · - [*0*, *10*] *10*)
  -tnu :: *logic* ⇒ *pttrn* ⇒ *logic* ⇒ *logic* ($\boldsymbol{\nu}$1 - · - [*0*, *10*] *10*)

notation *gfp* (μ)
notation *lfp* (ν)

translations
  $\boldsymbol{\mu}_H$ *X* · *P* == *CONST LEAST-FP* (*CONST utp-order H*) (λ *X*. *P*)
  $\boldsymbol{\nu}_H$ *X* · *P* == *CONST GREATEST-FP* (*CONST utp-order H*) (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* ∘ $\mathcal{H}$)
  proof −
    have {*P*. (*P is* $\mathcal{H}$) ∧ *F P* ⊑ *P*} = {*P*. (*P is* $\mathcal{H}$) ∧ *F* ($\mathcal{H}$ *P*) ⊑ *P*}
      by (*auto simp add*: *Healthy-def*)
    thus *?thesis*
      by (*simp add*: *LEAST-FP-def*)
  qed

  lemma *GFP-healthy-comp*: $\boldsymbol{\nu}$ *F* = $\boldsymbol{\nu}$ (*F* ∘ $\mathcal{H}$)
  proof −
    have {*P*. (*P is* $\mathcal{H}$) ∧ *P* ⊑ *F P*} = {*P*. (*P is* $\mathcal{H}$) ∧ *P* ⊑ *F* ($\mathcal{H}$ *P*)}
      by (*auto simp add*: *Healthy-def*)
    thus *?thesis*
      by (*simp add*: *GREATEST-FP-def*)
  qed

  lemma *top-healthy* [*closure*]: ⊤ *is* $\mathcal{H}$
    using *weak.top-closed* **by** *auto*

  lemma *bottom-healthy* [*closure*]: ⊥ *is* $\mathcal{H}$
    using *weak.bottom-closed* **by** *auto*

  lemma *utp-top*: *P is* $\mathcal{H}$ ⟹ *P* ⊑ ⊤
    using *weak.top-higher* **by** *auto*

  lemma *utp-bottom*: *P is* $\mathcal{H}$ ⟹ ⊥ ⊑ *P*
    using *weak.bottom-lower* **by** *auto*

end

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

lemma *upred-bottom*: $\bot_{\mathcal{P}}$ = *true*
  by *fastforce*

One way of obtaining a complete lattice is showing that the healthiness conditions are monotone, which the below locale characterises.

```
locale utp-theory-mono = utp-theory +
  assumes HCond-Mono [closure,intro]: Monotonic ℋ

sublocale utp-theory-mono ⊆ utp-theory-lattice
proof –
  interpret weak-complete-lattice fpl 𝒫 ℋ
    by (rule Knaster-Tarski, auto)

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

  hence complete-lattice (utp-order ℋ)
    by (simp add: utp-order-def, simp add: upred-lattice-def)

  thus utp-theory-lattice ℋ
    by (simp add: utp-theory-axioms utp-theory-lattice.intro utp-theory-lattice-axioms.intro)
qed
```

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

```
context utp-theory-mono
begin

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

lemma healthy-bottom: ⊥ = ℋ(true)
proof –
  have ⊥ = ⊥fpl 𝒫 ℋ
    by (simp add: utp-order-fpl)
  also have ... = ℋ ⊥𝒫
    using Knaster-Tarski-idem-extremes(2)[of 𝒫 ℋ]
    by (simp add: HCond-Idempotent HCond-Mono)
  also have ... = ℋ true
    by (simp add: upred-bottom)
   finally show ?thesis .
qed

lemma healthy-inf:
  assumes A ⊆ ⟦ℋ⟧H
  shows ⨅ A = ℋ (⨅ A)
  using Knaster-Tarski-idem-inf-eq[OF upred-weak-complete-lattice, of ℋ]
 by (simp, metis HCond-Idempotent HCond-Mono assms partial-object.simps(3) upred-lattice-def upred-lattice-inf
utp-order-def)
```

```
end

locale utp-theory-continuous = utp-theory +
  assumes HCond-Cont [closure,intro]: Continuous $\mathcal{H}$

sublocale utp-theory-continuous ⊆ utp-theory-mono
proof
  show Monotonic $\mathcal{H}$
    by (simp add: Continuous-Monotonic HCond-Cont)
qed

context utp-theory-continuous
begin

  lemma healthy-inf-cont:
    assumes $A \subseteq [\![\mathcal{H}]\!]_H$ $A \neq \{\}$
    shows $\bigsqcap A = \bigsqcap A$
  proof −
    have $\bigsqcap A = \bigsqcap (\mathcal{H}`A)$
      using Continuous-def HCond-Cont assms(1) assms(2) healthy-inf by auto
    also have ... = $\bigsqcap A$
      by (unfold Healthy-carrier-image[OF assms(1)], simp)
    finally show ?thesis .
  qed

  lemma healthy-inf-def:
    assumes $A \subseteq [\![\mathcal{H}]\!]_H$
    shows $\bigsqcap A = (if\ (A = \{\})\ then\ \top\ else\ (\bigsqcap A))$
    using assms healthy-inf-cont weak.weak-inf-empty by auto

  lemma healthy-meet-cont:
    assumes P is $\mathcal{H}$ Q is $\mathcal{H}$
    shows $P \sqcap Q = P \sqcap Q$
    using healthy-inf-cont[of {P, Q}] assms
    by (simp add: Healthy-if meet-def)

  lemma meet-is-healthy [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 disj-is-healthy [closure]:
    ⟦ P is $\mathcal{H}$; Q is $\mathcal{H}$ ⟧ ⟹ $(P \vee Q)$ is $\mathcal{H}$
    by (simp add: disj-upred-def meet-is-healthy)

  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)

  lemma inf-empty: $\bigsqcap\ \{\} = \top$
    by (simp add: healthy-inf-def)

  lemma inf-all: $\bigsqcap\ [\![\mathcal{H}]\!]_H = \bot$
    using weak-inf-carrier by auto
```

The UTP theory lfp operator can be rewritten to the alphabetised predicate lfp when in a continuous context.

```
theorem utp-lfp-def:
  assumes Monotonic F $F \in [\![\mathcal{H}]\!]_H \to [\![\mathcal{H}]\!]_H$
  shows $\boldsymbol{\mu}\ F = (\mu\ X \cdot F(\mathcal{H}(X)))$
proof (rule antisym)
  have ne: $\{P.\ (P\ is\ \mathcal{H}) \wedge F\ P \sqsubseteq P\} \neq \{\}$
  proof -
    have $F\ \top \sqsubseteq \top$
      using assms(2) utp-top weak.top-closed by force
    thus ?thesis
      by (auto)
  qed
  show $\boldsymbol{\mu}\ F \sqsubseteq (\mu\ X \cdot F\ (\mathcal{H}\ X))$
  proof -
    have $\bigsqcap\{P.\ (P\ is\ \mathcal{H}) \wedge F(P) \sqsubseteq P\} \sqsubseteq \bigsqcap\{P.\ F(\mathcal{H}(P)) \sqsubseteq P\}$
    proof -
      have 1: $\bigwedge\ P.\ F(\mathcal{H}(P)) = \mathcal{H}(F(\mathcal{H}(P)))$
        by (metis HCond-Idem Healthy-def assms(2) funcset-mem mem-Collect-eq)
      show ?thesis
      proof (rule Sup-least, auto)
        fix P
        assume a: $F\ (\mathcal{H}\ P) \sqsubseteq P$
        hence F: $(F\ (\mathcal{H}\ P)) \sqsubseteq (\mathcal{H}\ P)$
          by (metis 1 HCond-Mono mono-def)
        show $\bigsqcap\{P.\ (P\ is\ \mathcal{H}) \wedge F\ P \sqsubseteq P\} \sqsubseteq P$
        proof (rule Sup-upper2[of F $(\mathcal{H}\ P)$])
          show $F\ (\mathcal{H}\ P) \in \{P.\ (P\ is\ \mathcal{H}) \wedge F\ P \sqsubseteq P\}$
          proof (auto)
            show $F\ (\mathcal{H}\ P)\ is\ \mathcal{H}$
              by (metis 1 Healthy-def)
            show $F\ (F\ (\mathcal{H}\ P)) \sqsubseteq F\ (\mathcal{H}\ P)$
              using F 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: LEAST-FP-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: LEAST-FP-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

  lemma UINF-ind-Healthy [closure]:
    assumes ⋀ i. P(i) is ℋ
    shows (⨅ i · P(i)) is ℋ
    by (simp add: closure assms)

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 hcond for hcond :: 'α hrel ⇒ 'α hrel (ℋ) +
  assumes Healthy-Sequence [closure]: ⟦ P is ℋ; Q is ℋ ⟧ ⟹ (P ;; Q) is ℋ
begin

lemma upower-Suc-Healthy [closure]:
  assumes P is ℋ
  shows P ^ Suc n is ℋ
  by (induct n, simp-all add: closure assms upred-semiring.power-Suc)

end

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

  lemma seq-cont-Sup-distl:
    assumes P is ℋ A ⊆ ⟦ℋ⟧_H A ≠ {}
    shows P ;; (⨅ A) = ⨅ {P ;; Q | Q. Q ∈ A }
  proof –
    have {P ;; Q | Q. Q ∈ A } ⊆ ⟦ℋ⟧_H
      using Healthy-Sequence assms(1) assms(2) by (auto)
    thus ?thesis
      by (simp add: healthy-inf-cont seq-Sup-distl setcompr-eq-image assms)
  qed

  lemma seq-cont-Sup-distr:
    assumes Q is ℋ A ⊆ ⟦ℋ⟧_H A ≠ {}
    shows (⨅ A) ;; Q = ⨅ {P ;; Q | P. P ∈ A }
  proof –
    have {P ;; Q | P. P ∈ A } ⊆ ⟦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

  lemma uplus-healthy [closure]:
    assumes P is ℋ
    shows P⁺ is ℋ
    by (simp add: uplus-power-def closure assms)

end
```

There also exist UTP theories with units. 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-units* =
  utp-theory-rel +
  fixes *utp-unit* ($\mathcal{II}$)
  assumes *Healthy-Unit* [*closure*]: $\mathcal{II}$ *is* $\mathcal{H}$
begin

We can characterise the theory Kleene star by lifting the relational one.

definition *utp-star* (-$\star$ [*999*] *999*) **where**
[*upred-defs*]: *utp-star P* = ($P^\star$;; *utp-unit*)

We can then characterise tests as refinements of units.

definition *utp-test* :: *'a hrel* ⇒ *bool* **where**
[*upred-defs*]: *utp-test b* = ($\mathcal{II} \sqsubseteq b$)

end

locale *utp-theory-left-unital* =
  utp-theory-units +
  assumes *Unit-Left*: *P is* $\mathcal{H} \Longrightarrow (\mathcal{II}$;; $P) = P$

locale *utp-theory-right-unital* =
  utp-theory-units +
  assumes *Unit-Right*: *P is* $\mathcal{H} \Longrightarrow (P$;; $\mathcal{II}) = P$

locale *utp-theory-unital* =
  utp-theory-left-unital + *utp-theory-right-unital*
begin

lemma *Unit-self* [*simp*]:
  $\mathcal{II}$;; $\mathcal{II} = \mathcal{II}$
  by (*simp add*: *Healthy-Unit Unit-Right*)

lemma *utest-intro*:
  $\mathcal{II} \sqsubseteq P \Longrightarrow$ *utp-test P*
  by (*simp add*: *utp-test-def*)

lemma *utest-Unit* [*closure*]:
  utp-test $\mathcal{II}$
  by (*simp add*: *utp-test-def*)

end

locale *utp-theory-mono-unital* =
  utp-theory-unital $\mathcal{H}$ $\mathcal{II}$ + *utp-theory-mono* **for** $\mathcal{II}$
begin

lemma *utest-Top* [*closure*]: *utp-test* ⊤
  by (*simp add*: *Healthy-Unit utp-test-def utp-top*)

end

locale *utp-theory-cont-unital* = *utp-theory-cont-rel* + *utp-theory-unital hcond*
begin

sublocale *utp-theory-mono-unital* $\mathcal{H}$ $\mathcal{II}$
  by (*simp add*: *utp-theory-mono-axioms utp-theory-mono-unital-def utp-theory-unital-axioms*)

end

locale *utp-theory-unital-zerol* =
  utp-theory-unital +
  utp-theory-lattice +
  assumes *Top-Left-Zero*: P *is* $\mathcal{H} \Longrightarrow \top$;; $P = \top$

locale *utp-theory-cont-unital-zerol* =
  utp-theory-unital-zerol + *utp-theory-cont-unital hcond utp-unit*
begin

lemma *Top-test-Right-Zero*:
  assumes b *is* $\mathcal{H}$ *utp-test* b
  shows b ;; $\top = \top$
proof −
  have $b \sqcap \mathcal{II} = \mathcal{II}$
    by (*meson assms*(*2*) *semilattice-sup-class.le-iff-sup utp-test-def*)
  then show *?thesis*
   by (*metis* (*no-types*) *Top-Left-Zero Unit-Left assms*(*1*) *meet-top top-healthy upred-semiring.distrib-right*)
qed

end

20.4 Theory of relations

interpretation *rel-theory*: *utp-theory-mono-unital id skip-r*
  rewrites *rel-theory.utp-top* = *false*
  and *rel-theory.utp-bottom* = *true*
  and *carrier* (*utp-order id*) = *UNIV*
  and (P *is id*) = *True*
proof −
  show *utp-theory-mono-unital id II*
    by (*unfold-locales*, *simp-all add*: *Healthy-def*)
  then interpret *utp-theory-mono-unital id skip-r*
    by *simp*
  show *utp-top* = *false utp-bottom* = *true*
    by (*simp-all add*: *healthy-top healthy-bottom*)
  show *carrier* (*utp-order id*) = *UNIV* (P *is id*) = *True*
    by (*auto simp add*: *utp-order-def Healthy-def*)
qed

A more sophisticated UTP theory that characterises relations that only modify a region of the state space characterised by a lens a.

theorem *frame-theory*:
  assumes *vwb-frame*: *vwb-lens* a
  shows *utp-theory-cont-unital* (*frame* a) *II*
proof
  fix P Q
  show a:[a:[P]] = a:[P]
    by *rel-auto*
  show P *mods* $a \Longrightarrow Q$ *mods* $a \Longrightarrow P$;; Q *mods* a

using *vwb-frame mods-seq vwb-lens-mwb* **by** *blast*
show *Continuous* (*frame a*)
by (*rel-auto*)
show *II mods a*
by (*simp add*: *vwb-frame mods-skip*)
qed (*simp-all*)

20.5 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-order H1*, *orderB* = *utp-order H2*, *lower* = $\mathcal{H}_2$, *upper* = $\mathcal{H}_1$ $|\!)$

lemma *mk-conn-orderA* [*simp*]: $\mathcal{X}_{H1 \Leftarrow\langle\mathcal{H}_1,\mathcal{H}_2\rangle\Rightarrow H2}$ = *utp-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-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* ∘ *H* = *H* ∘ *ex x*
shows *retract* ((*ex x* ∘ *H*) $\Leftarrow\langle$*ex x*, *H*$\rangle\Rightarrow$ *H*)
proof (*unfold-locales*, *simp-all*)
show $H \in [\![ex\ x \circ H]\!]_H \to [\![H]\!]_H$
using *Healthy-Idempotent assms* **by** *blast*
from *assms*(*1*) *assms*(*3*)[*THEN sym*] **show** $ex\ x \in [\![H]\!]_H \to [\![ex\ x \circ H]\!]_H$
by (*simp add*: *Pi-iff Healthy-def fun-eq-iff exists-twice*)
fix *P Q*
assume *P is* (*ex x* ∘ *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* ∘ *H*)
  thus (∃ *x* · *H P*) ⊑ *P*
    by (*simp add*: *Healthy-def*)
qed

corollary *ex-retract-id*:
  assumes *vwb-lens x*
  shows *retract* (*ex x* ⇐⟨*ex x*, *id*⟩⇒ *id*)
  using *assms ex-retract*[**where** *H*=*id*] **by** (*auto*)
end

21 Relational Hoare calculus

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

21.1 Hoare Triple Definitions and Tactics

definition *hoare-r* :: *'α cond* ⇒ (*'α*, *'β*) *urel* ⇒ *'β cond* ⇒ *bool* ($\{\!|$-$|\!\}$/ -/ $\{\!|$-$|\!\}_u$) **where**
$\{\!|p|\!\}Q\{\!|r|\!\}_u = ((\lceil p\rceil_< \Rightarrow \lceil r\rceil_>) \sqsubseteq Q)$

notation *hoare-r* (**{**-**}**/ -/ **{**-**}**)

utp-lift-notation *hoare-r* (*1*)

translations **{**b**}**P**{**c**}** <= **{**$U(b)$**}**P**{**$U(c)$**}**

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

named-theorems *hoare* **and** *hoare-safe*

method *hoare-split* **uses** *hr* =
  ((*simp add*: *assigns-comp assigns-cond usubst*)*?*, — Combine Assignments where possible
   (*auto*
    intro: *hoare intro*!: *hoare-safe hr*
    simp add: *conj-comm conj-assoc usubst unrest*))[*1*] — Apply Hoare logic laws

method *hoare-auto* **uses** *hr* = (*hoare-split hr*: *hr*; (*rel-simp'*)*?*, *auto?*)

21.2 Basic Laws

lemma *hoare-meaning*:
  $\{\!|P|\!\}S\{\!|Q|\!\}_u = (\forall\ s\ s'.\ [\![P]\!]_e\ s \land [\![S]\!]_e\ (s, s') \longrightarrow [\![Q]\!]_e\ s')$
  by (*rel-auto*)

lemma *hoare-alt-def*: $\{\!|b|\!\}P\{\!|c|\!\}_u \longleftrightarrow (P\ ;;\ ?[c]) \sqsubseteq (?[b]\ ;;\ P)$
  by (*rel-auto*)

lemma *hoare-assume*: $\{\!|P|\!\}S\{\!|Q|\!\}_u \Longrightarrow ?[P]\ ;;\ S = ?[P]\ ;;\ S\ ;;\ ?[Q]$
  by (*rel-auto*)

lemma *hoare-pre-assume-1*: $\{\!\{b \wedge c\}\!\}P\{\!\{d\}\!\}_u = \{\!\{c\}\!\}?[b]$;; $P\{\!\{d\}\!\}_u$
by (*rel-auto*)

lemma *hoare-pre-assume-2*: $\{\!\{b \wedge c\}\!\}P\{\!\{d\}\!\}_u = \{\!\{b\}\!\}?[c]$;; $P\{\!\{d\}\!\}_u$
by (*rel-auto*)

lemma *hoare-test* [*hoare-safe*]: '$p \wedge b \Rightarrow q$' $\Longrightarrow \{\!\{p\}\!\}?[b]\{\!\{q\}\!\}_u$
by (*rel-simp*)

lemma *hoare-gcmd* [*hoare-safe*]: $\{\!\{p \wedge b\}\!\}P\{\!\{q\}\!\}_u \Longrightarrow \{\!\{p\}\!\}b \longrightarrow_r P\{\!\{q\}\!\}_u$
by (*rel-auto*)

lemma *hoare-r-conj* [*hoare-safe*]: $[\![\ \{\!\{p\}\!\}Q\{\!\{r\}\!\}_u;\ \{\!\{p\}\!\}Q\{\!\{s\}\!\}_u\ ]\!] \Longrightarrow \{\!\{p\}\!\}Q\{\!\{r \wedge s\}\!\}_u$
by *rel-auto*

lemma *hoare-r-weaken-pre* [*hoare*]:
$\{\!\{p\}\!\}Q\{\!\{r\}\!\}_u \Longrightarrow \{\!\{p \wedge q\}\!\}Q\{\!\{r\}\!\}_u$
$\{\!\{q\}\!\}Q\{\!\{r\}\!\}_u \Longrightarrow \{\!\{p \wedge q\}\!\}Q\{\!\{r\}\!\}_u$
by *rel-auto*+

lemma *pre-str-hoare-r*:
assumes '$p_1 \Rightarrow p_2$' **and** $\{\!\{p_2\}\!\}C\{\!\{q\}\!\}_u$
shows $\{\!\{p_1\}\!\}C\{\!\{q\}\!\}_u$
using *assms* **by** *rel-auto*

lemma *post-weak-hoare-r*:
assumes $\{\!\{p\}\!\}C\{\!\{q_2\}\!\}_u$ **and** '$q_2 \Rightarrow q_1$'
shows $\{\!\{p\}\!\}C\{\!\{q_1\}\!\}_u$
using *assms* **by** *rel-auto*

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

21.3 Sequence Laws

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

lemma *seq-hoare-invariant* [*hoare-safe*]: $[\![\ \{\!\{p\}\!\}Q_1\{\!\{p\}\!\}_u\ ;\ \{\!\{p\}\!\}Q_2\{\!\{p\}\!\}_u\ ]\!] \Longrightarrow \{\!\{p\}\!\}Q_1$;; $Q_2\{\!\{p\}\!\}_u$
by *rel-auto*

lemma *seq-hoare-stronger-pre-1* [*hoare-safe*]:
$[\![\ \{\!\{p \wedge q\}\!\}Q_1\{\!\{p \wedge q\}\!\}_u\ ;\ \{\!\{p \wedge q\}\!\}Q_2\{\!\{q\}\!\}_u\ ]\!] \Longrightarrow \{\!\{p \wedge q\}\!\}Q_1$;; $Q_2\{\!\{q\}\!\}_u$
by *rel-auto*

lemma *seq-hoare-stronger-pre-2* [*hoare-safe*]:
$[\![\ \{\!\{p \wedge q\}\!\}Q_1\{\!\{p \wedge q\}\!\}_u\ ;\ \{\!\{p \wedge q\}\!\}Q_2\{\!\{p\}\!\}_u\ ]\!] \Longrightarrow \{\!\{p \wedge q\}\!\}Q_1$;; $Q_2\{\!\{p\}\!\}_u$
by *rel-auto*

lemma *seq-hoare-inv-r-2* [*hoare*]: $[\![\ \{\!\{p\}\!\}Q_1\{\!\{q\}\!\}_u\ ;\ \{\!\{q\}\!\}Q_2\{\!\{q\}\!\}_u\ ]\!] \Longrightarrow \{\!\{p\}\!\}Q_1$;; $Q_2\{\!\{q\}\!\}_u$
by *rel-auto*

lemma *seq-hoare-inv-r-3* [*hoare*]: $[\![\ \{\!\{p\}\!\}Q_1\{\!\{p\}\!\}_u\ ;\ \{\!\{p\}\!\}Q_2\{\!\{q\}\!\}_u\ ]\!] \Longrightarrow \{\!\{p\}\!\}Q_1$;; $Q_2\{\!\{q\}\!\}_u$
by *rel-auto*

21.4 Assignment Laws

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

lemma *assigns-backward-hoare-r*:
$\{\!|\sigma \dagger p|\!\}\langle\sigma\rangle_a\{\!|p|\!\}_u$
by *rel-auto*

lemma *assign-floyd-hoare-r*:
assumes *vwb-lens x*
shows $\{\!|p|\!\}$ *assign-r x e* $\{\!|\exists\ v\ .\ p[\![\ll v\gg/x]\!] \wedge \&x = e[\![\ll v\gg/x]\!]|\!\}_u$
using *assms*
by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *assigns-init-hoare* [*hoare-safe*]:
$[\![$ *vwb-lens x*; $x \sharp p$; $x \sharp v$; $\{\!|\&x = v \wedge p|\!\}S\{\!|q|\!\}_u$ $]\!] \Longrightarrow \{\!|p|\!\}x := v\ ;;\ S\{\!|q|\!\}_u$
by (*rel-auto*)

lemma *assigns-init-hoare-general*:
$[\![$ *vwb-lens x*; $\bigwedge x_0.\ \{\!|\&x = v[\![\ll x_0\gg/\&x]\!] \wedge p[\![\ll x_0\gg/\&x]\!]|\!\}S\{\!|q|\!\}_u$ $]\!] \Longrightarrow \{\!|p|\!\}x := v\ ;;\ S\{\!|q|\!\}_u$
by (*rule seq-hoare-r*, *rule assign-floyd-hoare-r*, *simp*, *rel-auto*)

lemma *assigns-final-hoare* [*hoare-safe*]:
$\{\!|p|\!\}S\{\!|\sigma \dagger q|\!\}_u \Longrightarrow \{\!|p|\!\}S\ ;;\ \langle\sigma\rangle_a\{\!|q|\!\}_u$
by (*rel-auto*)

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

lemma *skip-hoare-impl-r* [*hoare-safe*]: $`p \Rightarrow q` \Longrightarrow \{\!|p|\!\}II\{\!|q|\!\}_u$
by *rel-auto*

21.5 Conditional Laws

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

lemma *cond-hoare-r-wp*:
assumes $\{\!|p'|\!\}S\{\!|q|\!\}_u$ **and** $\{\!|p''|\!\}T\{\!|q|\!\}_u$
shows $\{\!|(b \wedge p') \vee (\neg b \wedge p'')|\!\}S \triangleleft b \triangleright_r T\{\!|q|\!\}_u$
using *assms* **by** *pred-simp*

lemma *cond-hoare-r-sp*:
assumes $\langle\{\!|b \wedge p|\!\}S\{\!|q|\!\}_u\rangle$ **and** $\langle\{\!|\neg b \wedge p|\!\}T\{\!|s|\!\}_u\rangle$
shows $\langle\{\!|p|\!\}S \triangleleft b \triangleright_r T\{\!|q \vee s|\!\}_u\rangle$
using *assms* **by** *pred-simp*

lemma *hoare-ndet* [*hoare-safe*]:
assumes $\{\!|pre|\!\}P\{\!|post|\!\}_u$ $\{\!|pre|\!\}Q\{\!|post|\!\}_u$
shows $\{\!|pre|\!\}(P \sqcap Q)\{\!|post|\!\}_u$
using *assms* **by** (*rel-auto*)

lemma *hoare-disj* [*hoare-safe*]:
assumes $\{\!|pr|\!\}P\{\!|post|\!\}_u$ $\{\!|pr|\!\}Q\{\!|post|\!\}_u$
shows $\{\!|pr|\!\}(P \vee Q)\{\!|post|\!\}_u$

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

lemma *hoare-UINF* [*hoare-safe*]:
assumes $\bigwedge$ *i*. $i \in A \Longrightarrow \{\!|pre|\!\}P(i)\{\!|post|\!\}_u$
shows $\{\!|pre|\!\}(\bigsqcap\ i \in A \cdot P(i))\{\!|post|\!\}_u$
using *assms* **by** (*rel-auto*)

21.6 Recursion Laws

lemma *nu-hoare-r-partial*:
assumes *induct-step*:
$\bigwedge$ *st P*. $\{\!|p|\!\}P\{\!|q|\!\}_u \Longrightarrow \{\!|p|\!\}F\ P\{\!|q|\!\}_u$
shows $\{\!|p|\!\}\nu\ F\ \{\!|q|\!\}_u$
by (*meson hoare-r-def induct-step lfp-lowerbound order-refl*)

lemma *mu-hoare-r*:
assumes *WF*: *wf R*
assumes *M*:*mono F*
assumes *induct-step*:
$\bigwedge$ *st P*. $\{\!|p \wedge (e,\ll st\gg) \in \ll R\gg|\!\}P\{\!|q|\!\}_u \Longrightarrow \{\!|p \wedge e = \ll st\gg|\!\}F\ P\{\!|q|\!\}_u$
shows $\{\!|p|\!\}\mu\ F\ \{\!|q|\!\}_u$
unfolding *hoare-r-def*
proof (*rule mu-rec-total-utp-rule*[*OF WF M* , *of - e*], *goal-cases*)
case (*1 st*)
then show *?case*
using *induct-step*[*unfolded hoare-r-def*, *of* $(\lceil p\rceil_< \wedge (\lceil e\rceil_<, \ll st\gg)_u \in_u \ll R\gg \Rightarrow \lceil q\rceil_>)$ *st*]
by (*simp add*: *alpha*)
qed

lemma *mu-hoare-r'*:
assumes *WF*: *wf R*
assumes *M*:*mono F*
assumes *induct-step*:
$\bigwedge$ *st P*. $\{\!|p \wedge (e,\ll st\gg) \in \ll R\gg|\!\}\ P\ \{\!|q|\!\}_u \Longrightarrow \{\!|p \wedge e = \ll st\gg|\!\}\ F\ P\ \{\!|q|\!\}_u$
assumes *I0*: ‘$p' \Rightarrow p$‘
shows $\{\!|p'|\!\}\ \mu\ F\ \{\!|q|\!\}_u$
by (*meson I0 M WF induct-step mu-hoare-r pre-str-hoare-r*)

21.7 Iteration Rules

lemma *iter-hoare-r* [*hoare-safe*]: $\{\!|P|\!\}S\{\!|P|\!\}_u \Longrightarrow \{\!|P|\!\}S^\star\{\!|P|\!\}_u$
by (*rel-simp'*, *metis* (*mono-tags*, *hide-lams*) *mem-Collect-eq rtrancl-induct*)

lemma *while-hoare-r* [*hoare-safe*]:
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-top-def hoare-r-def*, *rule-tac lfp-lowerbound*) (*rel-auto*)

lemma *while-invr-hoare-r* [*hoare-safe*]:
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*)

lemma *while-r-minimal-partial*:
assumes *seq-step*: ‘$p \Rightarrow invar$‘

```
  assumes induct-step: {|invar∧ b|} C {|invar|}_u
  shows {|p|}while b do C od{|¬b ∧ invar|}_u
  using induct-step pre-str-hoare-r seq-step while-hoare-r by blast

lemma approx-chain:
  (⨅n::nat. ⌈p ∧ v <_u «n»⌉_<) = ⌈p⌉_<
  by (rel-auto)
```

Total correctness law for Hoare logic, based on constructive chains. This is limited to variants that have naturals numbers as their range.

```
lemma while-term-hoare-r:
  assumes ⋀ z::nat. {|p ∧ b ∧ v = «z»|}S{|p ∧ v < «z»|}_u
  shows {|p|}while_⊥ b do S od{|¬b ∧ p|}_u
proof –
  have (⌈p⌉_< ⇒ ⌈¬ b ∧ p⌉_>) ⊑ (μ X · S ;; X ◁ b ▷_r II)
  proof (rule mu-refine-intro)

    from assms show (⌈p⌉_< ⇒ ⌈¬ b ∧ p⌉_>) ⊑ S ;; (⌈p⌉_< ⇒ ⌈¬ b ∧ p⌉_>) ◁ b ▷_r II
      by (rel-auto)

    let ?E = λ n. ⌈p ∧ v <_u «n»⌉_<
    show (⌈p⌉_< ∧ (μ X · S ;; X ◁ b ▷_r II)) = (⌈p⌉_< ∧ (ν X · S ;; X ◁ b ▷_r II))
    proof (rule constr-fp-uniq[where E=?E])

      show (⨅n. ?E(n)) = ⌈p⌉_<
        by (rel-auto)

      show mono (λX. S ;; X ◁ b ▷_r II)
        by (simp add: cond-mono monoI seqr-mono)

      show constr (λX. S ;; X ◁ b ▷_r II) ?E
      proof (rule constrI)

        show chain ?E
        proof (rule chainI)
          show ⌈p ∧ v <_u «0»⌉_< = false
            by (rel-auto)
          show ⋀i. ⌈p ∧ v <_u «Suc i»⌉_< ⊑ ⌈p ∧ v <_u «i»⌉_<
            by (rel-auto)
        qed

        from assms
        show ⋀X n. (S ;; X ◁ b ▷_r II ∧ ⌈p ∧ v <_u «n + 1»⌉_<) =
                    (S ;; (X ∧ ⌈p ∧ v <_u «n»⌉_<) ◁ b ▷_r II ∧ ⌈p ∧ v <_u «n + 1»⌉_<)
          apply (rel-auto)
          using less-antisym less-trans apply blast
          done
      qed
    qed
  qed

  thus ?thesis
    by (simp add: hoare-r-def while-bot-def)
qed
```

lemma *while-vrt-hoare-r* [*hoare-safe*]:
  assumes $\bigwedge$ *z::nat*. $\{\!|p \wedge b \wedge v = \ll z \gg|\!\}S\{\!|p \wedge v < \ll z \gg|\!\}_u$ *'pre* $\Rightarrow$ *p' '*$(\neg b \wedge p) \Rightarrow$ *post'*
  shows $\{\!|pre|\!\}$*while b invr p vrt v do S od*$\{\!|post|\!\}_u$
  apply (*rule hoare-r-conseq*[*OF - assms*(*2*) *assms*(*3*)])
  apply (*simp add*: *while-vrt-def*)
  apply (*rule while-term-hoare-r*[**where** *v*=*v*, *OF assms*(*1*)])
  done

General total correctness law based on well-founded induction

lemma *while-wf-hoare-r*:
  assumes *WF*: *wf R*
  assumes *I0*: *'pre* $\Rightarrow$ *p'*
  assumes *induct-step*:$\bigwedge$ *st*. $\{\!|b \wedge p \wedge e = \ll st \gg|\!\}Q\{\!|p \wedge (e, \ll st \gg) \in \ll R \gg|\!\}_u$
  assumes *PHI*:*'*$(\neg b \wedge p) \Rightarrow$ *post'*
  shows $\{\!|pre|\!\}$*while*$_\bot$ *b invr p do Q od*$\{\!|post|\!\}_u$
unfolding *hoare-r-def while-inv-bot-def while-bot-def*
proof (*rule pre-weak-rel*[*of -* $\lceil p \rceil_<$])
  from *I0* **show** *'*$\lceil pre \rceil_< \Rightarrow \lceil p \rceil_<$*'*
    by *rel-auto*
  show $(\lceil p \rceil_< \Rightarrow \lceil post \rceil_>) \sqsubseteq (\mu\ X \cdot Q\ ;;\ X \triangleleft b \triangleright_r II)$
  proof (*rule mu-rec-total-utp-rule*[**where** *e*=*e*, *OF WF*])
    show *Monotonic* $(\lambda X.\ Q\ ;;\ X \triangleleft b \triangleright_r II)$
      by (*simp add*: *closure*)
    have *induct-step'*: $\bigwedge$ *st*. $(\lceil b \wedge p \wedge\ \ e =_u \ll st \gg \rceil_< \Rightarrow (\lceil p \wedge (e, \ll st \gg)_u \in_u \ll R \gg \rceil_>)) \sqsubseteq Q$
      using *induct-step* **by** *rel-auto*
    with *PHI*
    show $\bigwedge$*st*. $(\lceil p \rceil_< \wedge \lceil e \rceil_< =_u \ll st \gg \Rightarrow \lceil post \rceil_>) \sqsubseteq Q\ ;;\ (\lceil p \rceil_< \wedge (\lceil e \rceil_<, \ll st \gg)_u \in_u \ll R \gg \Rightarrow \lceil post \rceil_>)$
$\triangleleft b \triangleright_r II$
      by (*rel-auto*)
  qed
qed

21.8 Frame Rules

Frame rule: If starting *S* in a state satisfying *pestablishesq* in the final state, then we can insert an invariant predicate *r* when *S* is framed by *a*, provided that *r* does not refer to variables in the frame, and *q* does not refer to variables outside the frame.

lemma *frame-hoare-r*:
  assumes *vwb-lens a a* $\sharp$ *r a* $\natural$ *q* $\{\!|p|\!\}P\{\!|q|\!\}_u$
  shows $\{\!|p \wedge r|\!\}a{:}[P]\{\!|q \wedge r|\!\}_u$
  using *assms*
  by (*rel-auto*, *metis*)

lemma *frame-strong-hoare-r* [*hoare-safe*]:
  assumes *vwb-lens a a* $\sharp$ *r a* $\natural$ *q* $\{\!|p \wedge r|\!\}S\{\!|q|\!\}_u$
  shows $\{\!|p \wedge r|\!\}a{:}[S]\{\!|q \wedge r|\!\}_u$
  using *assms* **by** (*rel-auto*, *metis*)

lemma *frame-hoare-r'* [*hoare-safe*]:
  assumes *vwb-lens a a* $\sharp$ *r a* $\natural$ *q* $\{\!|r \wedge p|\!\}S\{\!|q|\!\}_u$
  shows $\{\!|r \wedge p|\!\}a{:}[S]\{\!|r \wedge q|\!\}_u$
  using *assms*
  by (*simp add*: *frame-strong-hoare-r utp-pred-laws.inf.commute*)

lemma *antiframe-hoare-r*:

assumes *vwb-lens a a* ♮ *r a* ♯ *q* $\{\!|p|\!\}P\{\!|q|\!\}_u$
shows $\{\!|p \wedge r|\!\}$ *a*:⟦*P*⟧ $\{\!|q \wedge r|\!\}_u$
using *assms* **by** (*rel-auto*, *metis*)

lemma *antiframe-strong-hoare-r*:
assumes *vwb-lens a a* ♮ *r a* ♯ *q* $\{\!|p \wedge r|\!\}P\{\!|q|\!\}_u$
shows $\{\!|p \wedge r|\!\}$ *a*:⟦*P*⟧ $\{\!|q \wedge r|\!\}_u$
using *assms* **by** (*rel-auto*, *metis*)

lemma *nmods-invariant*:
assumes *S nmods a a* ♮ *p*
shows **{***p***}***S***{***p***}**
using *assms* **by** (*rel-auto*, *metis*)

end

22 Weakest Liberal Precondition Calculus

theory *utp-wlp*
imports *utp-hoare*
begin

The calculus we here define is termed "weakest precondition" in the UTP book, however it is in reality the liberal version that does not account for termination.

named-theorems *wp*

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

consts
uwlp :: *'a* ⇒ *'b* ⇒ *'c* (**infix** *wlp 60*)

definition *wlp-upred* :: (*'α*, *'β*) *urel* ⇒ *'β cond* ⇒ *'α cond* **where**
wlp-upred Q r = $\lfloor\neg\ (Q\ ;;\ (\neg\ \lceil r\rceil_<))$ :: (*'α*, *'β*) $urel\rfloor_<$

utp-const *uwlp* (*0*)

adhoc-overloading
uwlp wlp-upred

declare *wlp-upred-def* [*urel-defs*]

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

lemma *wlp-conj* [*wp*]: (*P wlp* (*b* ∧ *c*)) = ((*P wlp b*) ∧ (*P wlp c*))
by (*rel-auto*)

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

lemma *wlp-nd-assign* [*wp*]: (*x* := ∗) *wlp b* = (∀ *v* · *b*⟦≪*v*≫/&*x*⟧)
by (*simp add*: *nd-assign-def wp*, *rel-auto*)

lemma *wlp-rel-aext-unrest* [*wp*]: ⟦ *vwb-lens a*; *a* ♯ *b* ⟧ ⟹ *a*:$[P]^+$ *wlp b* = ((*P wlp false*) $\oplus_p$ *a* ∨ *b*)

by $(\mathit{rel\text{-}simp},\ \mathit{metis\ mwb\text{-}lens\text{-}def\ vwb\text{-}lens\text{-}def\ weak\text{-}lens.put\text{-}get})$

lemma $\mathit{wlp\text{-}rel\text{-}aext\text{-}usedby}$ $[\mathit{wp}]$: $[\![\ \mathit{vwb\text{-}lens}\ a;\ a \natural\ b\ ]\!] \Longrightarrow a{:}[P]^{+}\ \mathit{wlp}\ b = (P\ \mathit{wlp}\ (b \upharpoonright_e a)) \oplus_p a$
by $(\mathit{rel\text{-}auto},\ \mathit{metis\ mwb\text{-}lens\text{-}def\ vwb\text{-}lens\text{-}mwb\ weak\text{-}lens.put\text{-}get})$

theorem $\mathit{wlp\text{-}skip\text{-}r}$ $[\mathit{wp}]$:
$\mathit{II}\ \mathit{wlp}\ r = r$
by $\mathit{rel\text{-}auto}$

theorem $\mathit{wlp\text{-}abort}$ $[\mathit{wp}]$:
$r \neq \mathit{true} \Longrightarrow \mathit{true}\ \mathit{wlp}\ r = \mathit{false}$
by $\mathit{rel\text{-}auto}$

theorem $\mathit{wlp\text{-}seq\text{-}r}$ $[\mathit{wp}]$: $(P\ ;;\ Q)\ \mathit{wlp}\ r = P\ \mathit{wlp}\ (Q\ \mathit{wlp}\ r)$
by $\mathit{rel\text{-}auto}$

theorem $\mathit{wlp\text{-}choice}$ $[\mathit{wp}]$: $(P \sqcap Q)\ \mathit{wlp}\ R = (P\ \mathit{wlp}\ R \wedge Q\ \mathit{wlp}\ R)$
by $(\mathit{rel\text{-}auto})$

theorem $\mathit{wlp\text{-}choice}'$ $[\mathit{wp}]$: $(P \vee Q)\ \mathit{wlp}\ R = (P\ \mathit{wlp}\ R \wedge Q\ \mathit{wlp}\ R)$
by $(\mathit{rel\text{-}auto})$

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

lemma $\mathit{wlp\text{-}UINF\text{-}ind}$ $[\mathit{wp}]$: $(\prod\ i \cdot P(i))\ \mathit{wlp}\ b = (\forall\ i \cdot P(i)\ \mathit{wlp}\ b)$
by $(\mathit{rel\text{-}auto})$

lemma $\mathit{wlp\text{-}test}$ $[\mathit{wp}]$: $?[b]\ \mathit{wlp}\ c = (b \Rightarrow c)$
by $(\mathit{rel\text{-}auto})$

lemma $\mathit{wlp\text{-}gcmd}$ $[\mathit{wp}]$: $(b \longrightarrow_r P)\ \mathit{wlp}\ c = (b \Rightarrow P\ \mathit{wlp}\ c)$
by $(\mathit{simp\ add}{:}\ \mathit{rgcmd\text{-}def\ wp})$

lemma $\mathit{wlp\text{-}USUP\text{-}pre}$ $[\mathit{wp}]$:
fixes $Q :: \text{-} \Rightarrow {}'s\ \mathit{upred}$
shows $P\ \mathit{wlp}\ (\bigwedge\ i{\in}A \cdot Q(i)) = U(\forall\ i{\in}\ll A\gg.\ P\ \mathit{wlp}\ Q\ i)$
by $(\mathit{rel\text{-}auto};\ \mathit{blast})$

theorem $\mathit{wlp\text{-}hoare\text{-}link}$:
$\{\!|p|\!\}Q\{\!|r|\!\}_u \longleftrightarrow `p \Rightarrow Q\ \mathit{wlp}\ r`$
by $\mathit{rel\text{-}auto}$

We can use the above theorem as a means to discharge Hoare triples with the following tactic

method $\mathit{hoare\text{-}wlp\text{-}auto}$ **uses** $\mathit{defs} = (\mathit{simp\ add}{:}\ \mathit{wlp\text{-}hoare\text{-}link\ wp\ unrest\ usubst\ defs};\ \mathit{rel\text{-}auto})$

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

theorem $\mathit{wlp\text{-}eq\text{-}intro}$: $[\![\ \bigwedge\ r.\ P\ \mathit{wlp}\ r = Q\ \mathit{wlp}\ r\ ]\!] \Longrightarrow P = Q$
by $(\mathit{rel\text{-}auto\ robust},\ \mathit{fastforce}+)$

end

23 Weakest Precondition Calculus

theory *utp-wp*
 imports *utp-wlp*
begin

This calculus is like the liberal version, but also accounts for termination. It is equivalent to the relational preimage.

consts
 uwp :: $'a \Rightarrow 'b \Rightarrow 'c$

utp-const *uwp*(*0*)

utp-lift-notation *uwp* (*0*)

syntax
 -uwp :: *logic* ⇒ *logic* ⇒ *logic* (**infix** *wp 60*)

translations
 -uwp P b == *CONST uwp P b*

definition *wp-upred* :: $('\alpha, '\beta)$ *urel* ⇒ $'\beta$ *cond* ⇒ $'\alpha$ *cond* **where**
[*upred-defs*]: *wp-upred P b* = *Pre*(*P* ;; *?*[*b*])

adhoc-overloading
 uwp wp-upred

term *P wp true*

theorem *refine-iff-wp*:
 fixes *P Q* :: $('\alpha, '\beta)$ *urel*
 shows $P \sqsubseteq Q \longleftrightarrow (\forall\ b.$ '*P wp b* ⇒ *Q wp b*'$)$
 apply (*rel-auto*)
 oops

theorem *wp-refine-iff*: $(\forall\ r.$ '*Q wp r* ⇒ *P wp r*'$) \longleftrightarrow P \sqsubseteq Q$
 by (*rel-auto robust*; *fastforce*)

theorem *wp-refine-intro*: $(\bigwedge\ r.$ '*Q wp r* ⇒ *P wp r*'$) \Longrightarrow P \sqsubseteq Q$
 using *wp-refine-iff* **by** *blast*

theorem *wp-eq-iff*: $(\forall\ r.$ *P wp r* = *Q wp r*$) \longrightarrow P = Q$
 by (*rel-auto robust*; *fastforce*)

theorem *wp-eq-intro*: $(\bigwedge\ r.$ *P wp r* = *Q wp r*$) \Longrightarrow P = Q$
 by (*simp add*: *wp-eq-iff*)

lemma *wp-true*: *P wp true* = *Pre*(*P*)
 by (*rel-auto*)

lemma *wp-false* [*wp*]: *P wp false* = *false*
 by (*rel-auto*)

lemma *wp-abort* [*wp*]: *false wp b* = *false*
 by (*rel-auto*)

lemma *wp-seq* [*wp*]: $(P \;;;\; Q)$ *wp* $b = P$ *wp* $(Q$ *wp* $b)$
  by (*simp add*: *wp-upred-def*, *metis Pre-seq RA1*)

lemma *wp-disj* [*wp*]: $(P \vee Q)$ *wp* $b = (P$ *wp* $b \vee Q$ *wp* $b)$
  by (*rel-auto*)

lemma *wp-ndet* [*wp*]: $(P \sqcap Q)$ *wp* $b = (P$ *wp* $b \vee Q$ *wp* $b)$
  by (*rel-auto*)

lemma *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*

lemma *wp-UINF-mem* [*wp*]: $(\prod i{\in}I \cdot P(i))$ *wp* $b = (\prod i{\in}I \cdot P(i)$ *wp* $b)$
  by (*rel-auto*)

lemma *wp-UINF-ind* [*wp*]: $(\prod i \cdot P(i))$ *wp* $b = (\prod i \cdot P(i)$ *wp* $b)$
  by (*rel-auto*)

lemma *wp-UINF-ind-2* [*wp*]: $(\prod (i, j) \cdot P\ i\ j)$ *wp* $b = (\bigvee (i, j) \cdot (P\ i\ j)$ *wp* $b)$
  by (*rel-auto*)

lemma *wp-UINF-ind-3* [*wp*]: $(\prod (i, j, k) \cdot P\ i\ j\ k)$ *wp* $b = (\bigvee (i, j, k) \cdot (P\ i\ j\ k)$ *wp* $b)$
  by (*rel-blast*)

lemma *wp-test* [*wp*]: $?[b]$ *wp* $c = (b \wedge c)$
  by (*rel-auto*)

lemma *wp-gcmd* [*wp*]: $(b \longrightarrow_r P)$ *wp* $c = (b \wedge P$ *wp* $c)$
  by (*rel-auto*)

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

lemma *wp-assigns* [*wp*]: $\langle\sigma\rangle_a$ *wp* $b = \sigma \dagger b$
  by (*rel-auto*)

lemma *wp-nd-assign* [*wp*]: $(x := *)$ *wp* $b = (\exists\ v \cdot b[\![\ll v\gg/\&x]\!])$
  by (*simp add*: *nd-assign-def wp*, *rel-auto*)

lemma *wp-rel-frext* [*wp*]:
  assumes *vwb-lens a* $a \sharp q$
  shows $a{:}[P]^{+}$ *wp* $(p \oplus_p a \wedge q) = ((P$ *wp* $p) \oplus_p a \wedge q)$
  using *assms*
  by (*rel-auto*, *metis* (*full-types*), *metis mwb-lens-def vwb-lens-mwb weak-lens.put-get*)

lemma *wp-rel-aext-unrest* [*wp*]: $[\![$ *vwb-lens a*; $a \sharp b$ $]\!] \Longrightarrow a{:}[P]^{+}$ *wp* $b = (b \wedge (P$ *wp true*$) \oplus_p a)$
  by (*rel-auto*, *metis*, *metis mwb-lens-def vwb-lens-mwb weak-lens.put-get*)

lemma *wp-rel-aext-usedby* [*wp*]: $[\![$ *vwb-lens a*; $a \natural b$ $]\!] \Longrightarrow a{:}[P]^{+}$ *wp* $b = (P$ *wp* $(b \upharpoonright_e a)) \oplus_p a$
  by (*rel-auto*, *metis mwb-lens-def vwb-lens-mwb weak-lens.put-get*)

lemma *wp-wlp-conjugate*: P *wp* $b = (\neg P$ *wlp* $(\neg b))$
  by (*rel-auto*)

Weakest Precondition and Weakest Liberal Precondition are equivalent for terminating deterministic programs.

lemma *wlp-wp-equiv-lem*: $[\![(mk_e\ (Pair\ a))\ \dagger\ II]\!]_e\ a$
  by (*rel-auto*)

lemma *wlp-wp-equiv-total-det*: $(\forall\ \ b\ .\ P\ wp\ b = P\ wlp\ b) \longleftrightarrow (Pre(P) = true \land ufunctional\ P)$
  apply (*rel-auto*)
    apply *blast*
   apply (*rename-tac a b y*)
  apply (*subgoal-tac* $[\![(mk_e\ (Pair\ a))\ \dagger\ II]\!]_e\ b$)
  apply (*simp add*: *assigns-r.rep-eq skip-r-def subst.rep-eq subst-id.rep-eq Abs-uexpr-inverse*)
  using *wlp-wp-equiv-lem* **apply** *fastforce*
  apply *blast*
  done

lemma *total-det-then-wlp-wp-equiv*: $[\![\ Pre(P) = true;\ ufunctional\ P\ ]\!] \Longrightarrow P\ wp\ b = P\ wlp\ b$
  using *wlp-wp-equiv-total-det* **by** *blast*

lemma *Pre-as-wp*: $Pre(P) = P\ wp\ true$
  by (*simp add*: *wp-true*)

lemma *nmods-via-wp*:
  $[\![\ vwb\text{-}lens\ x;\ \bigwedge\ v.\ P\ wp\ (\&x = \ll v \gg) = U(\&x = \ll v \gg)\ ]\!] \Longrightarrow P\ nmods\ x$
  by (*rel-auto*, *metis vwb-lens.put-eq*)

method *wp-calc* =
  (*rule wp-refine-intro wp-eq-intro*, *wp-tac*)

method *wp-auto* = (*wp-calc*, *rel-auto*)

end

24 Dynamic Logic

theory *utp-dynlog*
  imports *utp-sequent utp-wp*
begin

24.1 Definitions

named-theorems *dynlog-simp* **and** *dynlog-intro*

definition *dBox* :: $('\alpha,\ '\beta)\ urel \Rightarrow\ '\beta\ upred \Rightarrow\ '\alpha\ upred$ ([-]- [*0*,*999*] *999*)
where [*upred-defs*]: $dBox\ A\ \Phi = A\ wlp\ \Phi$

definition *dDia* :: $('\alpha,\ '\beta)\ urel \Rightarrow\ '\beta\ upred \Rightarrow\ '\alpha\ upred$ (<->- [*0*,*999*] *999*)
where [*upred-defs*]: $dDia\ A\ \Phi = A\ wp\ \Phi$

utp-const *dBox*(*0*) *dDia*(*0*)

lemma *dDia-dBox-def*: $<A>\Phi = (\neg\ [A](\neg\ \Phi))$
  by (*simp add*: *dBox-def dDia-def wp-wlp-conjugate*)

Correspondence between Hoare logic and Dynamic Logic

lemma *hoare-as-dynlog*: $\{\!|p|\!\}Q\{\!|r|\!\}_u = (p \Vdash [Q]r)$
  by (*rel-auto*)

24.2 Box Laws

lemma *dBox-false* [*dynlog-simp*]: $[false]\Phi = true$
by (*rel-auto*)

lemma *dBox-skip* [*dynlog-simp*]: $[II]\Phi = \Phi$
by (*rel-auto*)

lemma *dBox-assigns* [*dynlog-simp*]: $[\langle\sigma\rangle_a]\Phi = (\sigma \dagger \Phi)$
by (*simp add*: *dBox-def wlp-assigns-r*)

lemma *dBox-choice* [*dynlog-simp*]: $[P \sqcap Q]\Phi = ([P]\Phi \wedge [Q]\Phi)$
by (*rel-auto*)

lemma *dBox-seq*: $[P ;; Q]\Phi = [P][Q]\Phi$
by (*simp add*: *dBox-def wlp-seq-r*)

lemma *dBox-star-unfold*: $[P^\star]\Phi = (\Phi \wedge [P][P^\star]\Phi)$
by (*metis dBox-choice dBox-seq dBox-skip ustar-unfoldl*)

lemma *dBox-star-induct*: ‘$(\Phi \wedge [P^\star](\Phi \Rightarrow [P]\Phi)) \Rightarrow [P^\star]\Phi$‘
by (*rel-simp*, *metis* (*mono-tags*, *lifting*) *mem-Collect-eq rtrancl-induct*)

lemma *dBox-test*: $[?[p]]\Phi = (p \Rightarrow \Phi)$
by (*rel-auto*)

24.3 Diamond Laws

lemma *dDia-false* [*dynlog-simp*]: $<false>\Phi = false$
by (*simp add*: *dBox-false dDia-dBox-def*)

lemma *dDia-skip* [*dynlog-simp*]: $<II>\Phi = \Phi$
by (*simp add*: *dBox-skip dDia-dBox-def*)

lemma *dDia-assigns* [*dynlog-simp*]: $<\langle\sigma\rangle_a>\Phi = (\sigma \dagger \Phi)$
by (*simp add*: *dBox-assigns dDia-dBox-def subst-not*)

lemma *dDia-choice*: $<P \sqcap Q>\Phi = (<P>\Phi \vee <Q>\Phi)$
by (*simp add*: *dBox-def dDia-dBox-def wlp-choice*)

lemma *dDia-seq*: $<P ;; Q>\Phi = <P><Q>\Phi$
by (*simp add*: *dBox-def dDia-dBox-def wlp-seq-r*)

lemma *dDia-test*: $<?[p]>\Phi = (p \wedge \Phi)$
by (*rel-auto*)

24.4 Sequent Laws

lemma *sBoxSeq* [*dynlog-simp*]: $\Gamma \Vdash [P ;; Q]\Phi \equiv \Gamma \Vdash [P][Q]\Phi$
by (*simp add*: *dBox-def wlp-seq-r*)

lemma *sBoxTest* [*dynlog-intro*]: $\Gamma \Vdash (b \Rightarrow \Psi) \Longrightarrow \Gamma \Vdash [?[b]]\Psi$
by (*rel-auto*)

lemma *sBoxAssignFwd* [*dynlog-intro*]:
assumes *vwb-lens x* $\bigwedge x_0.\ ((\Gamma[\![\ll x_0\gg/\&x]\!] \wedge \&x = v[\![\ll x_0\gg/\&x]\!]) \Vdash \Phi)$

shows $(\Gamma \Vdash [x := v]\Phi)$
proof −
have $\{\!|\Gamma|\!\}$ $x := v$;; *II* $\{\!|\Phi|\!\}_u$
by (*metis* (*no-types*) *assigns-init-hoare-general assms*(*1*) *assms*(*2*) *dBox-skip hoare-as-dynlog utp-pred-laws.inf-commu*
then show *?thesis*
by (*simp add*: *hoare-as-dynlog*)
qed

lemma *sBoxAssignFwd-simp* [*dynlog-simp*]: $[\![$ *vwb-lens* x; $x \sharp v$; $x \sharp \Gamma$ $]\!] \Longrightarrow (\Gamma \Vdash [x := v]\Phi) = ((\&x = v \wedge \Gamma) \Vdash \Phi)$
by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *sBoxIndStar*: $\Vdash [\Phi \Rightarrow [P]\Phi]_u \Longrightarrow \Phi \Vdash [P^\star]\Phi$
by (*rel-simp*, *metis* (*mono-tags*, *lifting*) *mem-Collect-eq rtrancl-induct*)

end

25 Blocks (Abstract Local Variables)

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

25.1 Extending and Contracting Substitutions

definition *subst-ext* :: $('\alpha \Longrightarrow '\beta) \Rightarrow ('\alpha, '\beta)$ *psubst* (ext_s) **where**
— Extend state space, setting local state to an arbitrary value
[*upred-defs*]: $ext_s\ a = (\!|\&a \mapsto_s \&\mathbf{v}|\!)$

definition *subst-con* :: $('\alpha \Longrightarrow '\beta) \Rightarrow ('\beta, '\alpha)$ *psubst* (con_s) **where**
— Contract the state space with get
[*upred-defs*]: $con_s\ a = \&a$

lemma *subst-con-alt-def*: $con_s\ a = (\!|\mathbf{v} \mapsto_s \&a|\!)$
unfolding *subst-con-def* **by** (*rel-auto*)

lemma *subst-ext-con* [*usubst*]: *mwb-lens* $a \Longrightarrow con_s\ a \circ_s ext_s\ a = id_s$
by (*rel-simp*)

lemma *subst-apply-con* [*usubst*]: $\langle con_s\ a\rangle_s\ x = \&a{:}x$
by (*rel-simp*)

Variables in the global state space will be retained after a state is contracted

lemma *subst-con-update-sublens* [*usubst*]:
$[\![$ *mwb-lens* a; $x \subseteq_L a$ $]\!] \Longrightarrow con_s\ a \circ_s$ *subst-upd* $\sigma\ x\ v =$ *subst-upd* $(con_s\ a \circ_s \sigma)\ (x /_L a)\ v$
by (*simp add*: *subst-con-def usubst alpha*, *rel-simp*)

Variables in the local state space will be lost after a state is contracted

lemma *subst-con-update-indep* [*usubst*]:
$[\![$ *mwb-lens* x; *mwb-lens* a; $a \bowtie x$ $]\!] \Longrightarrow con_s\ a \circ_s$ *subst-upd* $\sigma\ x\ v = (con_s\ a \circ_s \sigma)$
by (*simp add*: *subst-con-alt-def usubst alpha*)

lemma *subst-ext-apply* [*usubst*]: $\langle ext_s\ a\rangle_s\ x = \&x \upharpoonright_e a$
apply (*rel-simp*)
oops

25.2 Generic Blocks

We ensure that the initial values of local are arbitrarily chosen using the non-deterministic choice operator.

definition *block-open* :: $(<'a, 'c> \Longleftrightarrow 'b) \Rightarrow ('a, 'b)$ *urel* (*open*$_{-}$) **where**
[*upred-defs*]: *block-open a* $= \langle ext_s\ \mathcal{V}_a\rangle_a\ ;;\ \mathcal{C}[a] := *$

lemma *block-open-alt-def*:
  sym-lens a $\Longrightarrow$ *block-open a* $= \langle ext_s\ \mathcal{V}_a\rangle_a\ ;;\ (\$\mathcal{V}[a]' =_u \$\mathcal{V}[a])$
  by (*rel-auto*, *metis lens-indep-vwb-iff sym-lens.put-region-coregion-cover sym-lens-def*)

definition *block-close* :: $(<'a, 'c> \Longleftrightarrow 'b) \Rightarrow ('b, 'a)$ *urel* (*close*$_{-}$) **where**
[*upred-defs*]: *block-close a* $= \langle con_s\ \mathcal{V}_a\rangle_a$

lemma *wp-open-block* [*wp*]: *psym-lens a* $\Longrightarrow$ $open_a$ *wp b* $= (\exists\ v \cdot (\!|\&\mathcal{V}[a] \mapsto_s \&\mathbf{v}, \&\mathcal{C}[a] \mapsto_s \ll v\gg|\!)\ \dagger$ $b)$
  by (*simp add*: *block-open-def subst-ext-def wp usubst unrest*)

lemma *wp-close-block* [*wp*]: *psym-lens a* $\Longrightarrow$ $close_a$ *wp b* $= con_s\ \mathcal{V}_a \dagger b$
  by (*simp add*: *block-close-def subst-ext-def wp usubst unrest*)

lemma *block-open-conv*:
  sym-lens a $\Longrightarrow$ ${open_a}^{-} = close_a$
  by (*rel-auto*, *metis lens-indep-def sym-lens.put-region-coregion-cover sym-lens-def*)

lemma *block-open-close*:
  psym-lens a $\Longrightarrow$ $open_a\ ;;\ close_a = II$
  by (*rel-auto*)

I needed this property for the assignment open law below.

lemma *usubst-prop*: $\sigma \oplus_s a = [a \mapsto_s \&a \dagger \sigma]$
  by (*rel-simp*)

lemma *block-assigns-open*:
  psym-lens a $\Longrightarrow \langle\sigma\rangle_a\ ;;\ open_a = open_a\ ;;\ \langle\sigma \oplus_s \mathcal{V}_a\rangle_a$
  apply (*wp-calc*)
  apply (*simp add*: *usubst-prop usubst*)
  apply (*rel-auto*)
  done

lemma *block-assign-open*:
  psym-lens a $\Longrightarrow x := v\ ;;\ open_a = open_a\ ;;\ \mathcal{V}[a]{:}x := (v \oplus_p \mathcal{V}_a)$
  by (*simp add*: *block-assigns-open*, *rel-auto*)

lemma *block-assign-local-close*:
  $\mathcal{V}_a \bowtie x \Longrightarrow x := v\ ;;\ close_a = close_a$
  by (*rel-auto*)

lemma *block-assign-global-close*:
  $[\![$ *psym-lens a*; $x \subseteq_L \mathcal{V}_a\ ;\ \mathcal{V}[a] \natural\ v\ ]\!] \Longrightarrow (x := v)\ ;;\ close_a = close_a\ ;;\ (x{\restriction}\mathcal{V}[a] := (v \restriction_e \mathcal{V}_a))$
  by (*rel-simp*)

lemma *block-assign-global-close'*:
  $[\![$ *sym-lens a*; $x \subseteq_L \mathcal{V}_a\ ;\ \mathcal{C}[a] \sharp\ v\ ]\!] \Longrightarrow (x := v)\ ;;\ close_a = close_a\ ;;\ (x{\restriction}\mathcal{V}[a] := (v \restriction_e \mathcal{V}_a))$
  by (*rule block-assign-global-close*, *simp-all add*: *sym-lens-unrest'*)

lemma *hoare-block* [*hoare-safe*]:
  assumes *psym-lens a*
  shows $\{\!\{p \oplus_p \mathcal{V}_a\}\!\}P\{\!\{q \oplus_p \mathcal{V}_a\}\!\}_u \Longrightarrow \{\!\{p\}\!\}open_a \;;;\; P \;;;\; close_a\{\!\{q\}\!\}_u$
  using *assms* **by** (*rel-simp*)

lemma $vwb\text{-}lens\ a \Longrightarrow a{:}[P]^+ = a{:}[\langle con_s\ a\rangle_a \;;;\; P \;;;\; \langle ext_s\ a\rangle_a \;;;\; (\$a\acute{} =_u \$a)]$
  by (*rel-auto*)

end

26 State Variable Declaration Parser

theory *utp-state-parser*
  imports *utp-blocks*
begin

This theory sets up a parser for state blocks, as an alternative way of providing lenses to a predicate. A program with local variables can be represented by a predicate indexed by a tuple of lenses, where each lens represents a variable. These lenses must then be supplied with respect to a suitable state space. Instead of creating a type to represent this alphabet, we can create a product type for the state space, with an entry for each variable. Then each variable becomes a composition of the fst_L and snd_L lenses to index the correct position in the variable vector.

We first creation a vacuous definition that will mark when an indexed predicate denotes a state block.

definition *state-block* :: ($'v \Rightarrow 'p$) $\Rightarrow$ $'v \Rightarrow 'p$ **where**
[*upred-defs*]: *state-block f x = f x*

We declare a number of syntax translations to produce lens and product types, to obtain a type for the overall state space, to construct a tuple that denotes the lens vector parameter, to construct the vector itself, and finally to construct the state declaration.

syntax
  -lensT :: *type* ⇒ *type* ⇒ *type* (*LENSTYPE'*(-, -'))
  -pairT :: *type* ⇒ *type* ⇒ *type* (*PAIRTYPE'*(-, -'))
  -state-type :: *pttrn* ⇒ *type*
  -state-tuple :: *type* ⇒ *pttrn* ⇒ *logic*
  -state-lenses :: *pttrn* ⇒ *logic* ⇒ *logic*
  -state-decl :: *pttrn* ⇒ *logic* ⇒ *logic* (*alpha* - • - [*0*, *10*] *10*)
  -state-decl-in :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* (*alpha* - *in* - • - [*0*, *0*, *10*] *10*)

translations
  (*type*) *PAIRTYPE*(*'a*, *'b*) => (*type*) *'a* × *'b*
  (*type*) *LENSTYPE*(*'a*, *'b*) => (*type*) *'a* ⟹ -

  -state-type (*-constrain x t*) => *t*
  -state-type (*CONST Pair* (*-constrain x t*) *vs*) => *-pairT t* (*-state-type vs*)

  -state-tuple st (*-constrain x t*) => *-constrain x* (*-lensT t st*)
  -state-tuple st (*CONST Pair* (*-constrain x t*) *vs*) =>
    CONST Product-Type.Pair (*-constrain x* (*-lensT t st*)) (*-state-tuple st vs*)

  -state-decl-in vs loc P =>
    CONST state-block (*-abs* (*-state-tuple* (*-state-type vs*) *vs*) *P*) (*-state-lenses vs loc*)

```
  -state-decl vs P =>
    CONST state-block (-abs (-state-tuple (-state-type vs) vs) P) (-state-lenses vs 1_L)
  -state-decl vs P <= CONST state-block (-abs vs P) k
```

ML ‹

›

parse-translation ‹

```
  let
    open HOLogic; open Syntax;
    fun lensT s t = Type (@{type-name lens-ext}, [s, t, HOLogic.unitT]);
    fun lens-comp a b c = Const (@{const-syntax lens-comp}, lensT a b --> lensT b c --> lensT a
c);
   fun fst-lens t = Const (@{const-syntax fst-lens}, Type (@{type-name lens-ext}, [t, dummyT, unitT]));
    val snd-lens = Const (@{const-syntax snd-lens}, dummyT);
    fun id-lens t = Const (@{const-syntax id-lens},Type (@{type-name lens-ext}, [t, dummyT, unitT]));
     fun lens-syn-typ t = const @{type-syntax lens-ext} $ t $ const @{type-syntax dummy} $ const
@{type-syntax unit};
    fun constrain t ty = const @{syntax-const -constrain} $ t $ ty;

    (* Construct a tuple of n lenses, whose source type is product of the types in ts, and each lens
       has an element of the type: prod-lens [t0, t1 ... ] 1 : t1 ==> t0 * t1 * ... *)
    fun prod-lens ts i =
      let open Syntax; open Library; fun lens-compf (x, y) = const @{const-name lens-comp} $ x $ y in
      if (length ts = 1)
      then Const (@{const-name id-lens}, lensT (nth ts i) (nth ts i))
      else if (length ts = i + 1)
      then foldl lens-compf (Const (@{const-name snd-lens}, lensT (nth ts i) dummyT), replicate (i−1)
(const @{const-name snd-lens}))
      else foldl lens-compf (Const (@{const-name fst-lens}, lensT (nth ts i) dummyT), replicate i (const
@{const-name snd-lens}))
      end;

    (* Construct a tuple of lenses for each of the possible locally declared variables *)
    fun state-lenses ts sty st =
      foldr1 (fn (x, y) => pair-const dummyT dummyT $ x $ y) (map (fn i => lens-comp dummyT sty
dummyT $ prod-lens ts i $ st) (upto (0, length ts − 1)));

    fun
      (* Add up the number of variable declarations in the tuple *)
      var-decl-num (Const (@{const-syntax Product-Type.Pair},-) $ - $ vs) = var-decl-num vs + 1 |
      var-decl-num - = 1;

    fun
       var-decl-typs (Const (@{const-syntax Product-Type.Pair},-) $ (Const (-constrain, -) $ - $ typ) $
vs) = Syntax-Phases.decode-typ typ :: var-decl-typs vs |
      var-decl-typs (Const (-constrain, -) $ - $ typ) = [Syntax-Phases.decode-typ typ] |
      var-decl-typs - = [];

    fun state-lens ctx [vs, loc] = (state-lenses (var-decl-typs vs) (mk-tupleT (var-decl-typs vs)) loc);
  in
  [(-state-lenses, state-lens)]
  end
```

⟩

26.1 Variable Block Syntax

definition *vblock* :: $(<'a, 'b> \Longleftrightarrow 'c) \Rightarrow ('d \Rightarrow 'c\ hrel) \Rightarrow 'd \Rightarrow 'a\ hrel$ **where**
[*upred-defs*]: $vblock\ sl\ f\ x = open_{sl}\ ;;\ f\ x\ ;;\ close_{sl}$

syntax
  -var-block-in :: $pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic$ (*var - in - • -* [*0*, *0*, *10*] *10*)

translations
  -var-block-in vs sl P => *CONST vblock sl* (*-abs* (*-state-tuple* (*-state-type vs*) *vs*) *P*) (*-state-lenses vs* $\mathcal{C}_{sl}$)

26.2 Examples

term *alpha* (*x*::*int*, *y*::*real*, *z*::*int*) • $y := \&x + \&z$

lemma *alpha p* • $II = II$
  by (*rel-auto*)

end

27 Relational Operational Semantics

theory *utp-rel-opsem*
  imports
    utp-rel-laws
    utp-hoare
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 [22].

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*: $(\sigma, \langle\varrho\rangle_a) \rightarrow_u (\varrho \circ_s \sigma, II)$
  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 alpha*)

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 alpha*)

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*)

Theorem linking Hoare calculus and operational semantics. If we start Q in a state σ_0 satisfying p, and Q reaches final state σ_1 then r holds in this final state.

theorem *hoare-opsem-link*:
  $\{\!|p|\!\}Q\{\!|r|\!\}_u = (\forall\ \sigma_0\ \sigma_1.\ `\sigma_0 \dagger p` \wedge (\sigma_0,\ Q) \rightarrow_u (\sigma_1,\ II) \longrightarrow `\sigma_1 \dagger r`)$
  apply (*rel-auto*)
  apply (*rename-tac a b*)
  apply (*metis* (*full-types*) *lit.rep-eq*)
  done

declare *trel.simps* [*simp del*]

end

28 Symbolic Evaluation of Relational Programs

theory *utp-sym-eval*
  imports *utp-rel-opsem*
begin

The following operator applies a variable context Γ as an assignment, and composes it with a relation P for the purposes of evaluation.

definition *utp-sym-eval* :: *'s usubst* ⇒ *'s hrel* ⇒ *'s hrel* (**infixr** ⊨ *55*) **where**
[*upred-defs*]: *utp-sym-eval* Γ *P* = (⟨Γ⟩$_a$;; *P*)

named-theorems *symeval*

lemma *seq-symeval* [*symeval*]: Γ ⊨ *P* ;; *Q* = (Γ ⊨ *P*) ;; *Q*
  by (*rel-auto*)

lemma *assigns-symeval* [*symeval*]: Γ ⊨ ⟨σ⟩$_a$ = (σ $\circ_s$ Γ) ⊨ *II*
  by (*rel-auto*)

lemma *term-symeval* [*symeval*]: (Γ ⊨ *II*) ;; *P* = Γ ⊨ *P*
  by (*rel-auto*)

lemma *if-true-symeval* [*symeval*]: ⟦ Γ † *b* = *true* ⟧ ⟹ Γ ⊨ (*P* ◃ *b* ▹$_r$ *Q*) = Γ ⊨ *P*
  by (*simp add*: *utp-sym-eval-def usubst assigns-r-comp*)

lemma *if-false-symeval* [*symeval*]: ⟦ Γ † *b* = *false* ⟧ ⟹ Γ ⊨ (*P* ◃ *b* ▹$_r$ *Q*) = Γ ⊨ *Q*
  by (*simp add*: *utp-sym-eval-def usubst assigns-r-comp*)

lemma *while-true-symeval* [*symeval*]: ⟦ Γ † *b* = *true* ⟧ ⟹ Γ ⊨ *while b do P od* = Γ ⊨ (*P* ;; *while b do P od*)
  by (*subst while-unfold*, *simp add*: *symeval*)

lemma *while-false-symeval* [*symeval*]: ⟦ Γ † *b* = *false* ⟧ ⟹ Γ ⊨ *while b do P od* = Γ ⊨ *II*
  by (*subst while-unfold*, *simp add*: *symeval*)

lemma *while-inv-true-symeval* [*symeval*]: ⟦ Γ † *b* = *true* ⟧ ⟹ Γ ⊨ *while b invr S do P od* = Γ ⊨ (*P* ;; *while b do P od*)
  by (*metis while-inv-def while-true-symeval*)

lemma *while-inv-false-symeval* [*symeval*]: ⟦ Γ † *b* = *false* ⟧ ⟹ Γ ⊨ *while b invr S do P od* = Γ ⊨ *II*
  by (*metis while-false-symeval while-inv-def*)

method *sym-eval* = (*simp add*: *symeval usubst lit-simps*[*THEN sym*]), (*simp del*: *One-nat-def add*: *One-nat-def*[*THEN sym*])*?*

syntax
  -terminated :: *logic* ⇒ *logic* (*terminated*: - [*999*] *999*)

translations
  terminated: Γ == Γ ⊨ *II*

Below are some theorems linking symbolic evaluation and Hoare logic.

lemma *hoare-symeval-link-1*: ⦃*b*⦄*P*⦃*c*⦄$_u$ = (∀ s_1 s_2. 's_1 † *b*' ∧ ((s_1 ⊨ *P*) ⊑ (s_2 ⊨ *II*)) ⟶ 's_2 † *c*')
  by (*simp add*: *utp-sym-eval-def usubst hoare-opsem-link trel.simps*)

lemma *hoare-symeval-link-2*: ⦃*b*⦄*P*⦃*c*⦄$_u$ ⟹ 's_1 † *b*' ∧ ((s_1 ⊨ *P*) = (s_2 ⊨ *II*)) ⟶ 's_2 † *c*'
  by (*rel-blast*)

end

29 Strongest Postcondition Calculus

theory *utp-sp*
imports *utp-wp*
begin

named-theorems *sp*

method *sp-tac* = (*simp add*: *sp*)

consts
  $usp :: {'a} \Rightarrow {'b} \Rightarrow {'c}$ (**infix** *sp 60*)

definition *sp-upred* :: $'\alpha$ *cond* $\Rightarrow$ $('\alpha, '\beta)$ *urel* $\Rightarrow$ $'\beta$ *cond* **where**
sp-upred $p\ Q = \lfloor(\lceil p\rceil_{>} \ ;;\ Q) :: ('\alpha, '\beta)\ urel\rfloor_{>}$

no-utp-lift *usp*

adhoc-overloading
  usp sp-upred

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

lemma *sp-false* [*sp*]: $p\ sp\ false = false$
  by (*rel-simp*)

lemma *sp-true* [*sp*]: $q \neq false \Longrightarrow q\ sp\ true = true$
  by (*rel-auto*)

lemma *sp-assign-r* [*sp*]:
  $vwb\text{-}lens\ x \Longrightarrow (p\ sp\ x := e) = (\exists\ v \cdot p⟦«v»/x⟧ \wedge \&x =_u e⟦«v»/x⟧)$
  by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*, *metis vwb-lens.put-eq*)

lemma *sp-assigns-r* [*sp*]:
  $(p\ sp\ \langle\sigma\rangle_a) = (\exists\ v \cdot [p⟦«v»/\&\mathbf{v}⟧]_u \wedge \&\mathbf{v} =_u \sigma⟦«v»/\&\mathbf{v}⟧)$
  by (*rel-auto*)

lemma *sp-convr* [*sp*]: $b\ sp\ P^{-} = P\ wp\ b$
  by (*rel-auto*)

lemma *wp-convr* [*wp*]: $P^{-}\ wp\ b = b\ sp\ P$
  by (*rel-auto*)

lemma *sp-seqr* [*sp*]: $b\ sp\ (P\ ;;\ Q) = (b\ sp\ P)\ sp\ Q$
  by (*rel-auto*)

lemma *sp-is-post-condition*:
  $⦃p⦄C⦃p\ sp\ C⦄_u$
  by *rel-blast*

lemma *sp-it-is-the-strongest-post*:
  $`p\ sp\ C \Rightarrow Q` \Longrightarrow ⦃p⦄C⦃Q⦄_u$
  by *rel-blast*

theorem *sp-hoare-link*:
  $⦃p⦄Q⦃r⦄_u \longleftrightarrow `p\ sp\ Q \Rightarrow r`$

by *rel-auto*

lemma *sp-while-r* [*sp*]:
  assumes ‹‘*pre* ⇒ *I*‘› **and** ‹$\{\!|I \wedge b|\!\}C\{\!|I'|\!\}_u$› **and** ‹‘*I*′ ⇒ *I*‘›
  shows (*pre sp invar I while*$_\bot$ *b do C od*) = (¬*b* ∧ *I*)
  unfolding *sp-upred-def*
  oops

theorem *sp-eq-intro*: $[\![\bigwedge r.\ r\ sp\ P = r\ sp\ Q]\!] \Longrightarrow P = Q$
  by (*rel-auto robust*, *fastforce*+)

lemma *wlp-sp-sym*:
  ‘*prog wlp* (*true sp prog*)‘
  by *rel-auto*

lemma *it-is-pre-condition*:$\{\!|C\ wlp\ Q|\!\}C\{\!|Q|\!\}_u$
  by *rel-blast*

end

30 Concurrent Programming

theory *utp-concurrency*
  imports
    utp-hoare
    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 [22].

30.1 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 :: $'\beta_1$

We set up syntax for the three variable classes.

syntax
  -svarprior :: *svid* (<)

-svarl :: *svid* (*0*)
-svarr :: *svid* (*1*)

translations
-svarprior == *CONST mrg-prior*
-svarl == *CONST mrg-left*
-svarr == *CONST mrg-right*

30.2 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*, $'\alpha)$ *urel*

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

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

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*]: $swap_m = (0{:}\mathbf{v},1{:}\mathbf{v}) := (\&1{:}\mathbf{v},\&0{:}\mathbf{v})$

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$(M) \equiv (swap_m \;;;\; M)$

30.3 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*) *urel* **where**
[*upred-defs*]: $U0 = (\$0{:}\mathbf{v}\acute{} =_u \$\mathbf{v})$

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

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*]: $U0\alpha = (1_L \times_L$ *mrg-left*)

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

We then create the following intuitive syntax for separating simulations.

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

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

$\lceil P \rceil_0$ 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*: $(P ;; \mathit{U0}) = \lceil P \rceil_0$
  by (*rel-auto*)

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

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

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

lemma *U0α-indep-right-uvar* [*simp*]: $\mathit{vwb\text{-}lens}\ x \Longrightarrow \mathit{U0}\alpha \bowtie \mathit{out\text{-}var}\ (x ;_L \mathit{mrg\text{-}right})$
  by (*force intro*: *plus-pres-lens-indep fst-snd-lens-indep lens-indep-left-comp*
          simp add: *U0α-def out-var-def prod-as-plus*)

lemma *U1α-indep-left-uvar* [*simp*]: $\mathit{vwb\text{-}lens}\ x \Longrightarrow \mathit{U1}\alpha \bowtie \mathit{out\text{-}var}\ (x ;_L \mathit{mrg\text{-}left})$
  by (*force intro*: *plus-pres-lens-indep fst-snd-lens-indep lens-indep-left-comp*
          simp add: *U1α-def out-var-def prod-as-plus*)

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

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

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

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

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

lemma *U1-skip* [*alpha*]: $\lceil \mathit{II} \rceil_1 = (\$1{:}\mathbf{v}' =_u \$\mathbf{v})$
  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-var x*) $;_L$ *U0α* = *in-var x*
by (*simp add*: *U0α-def alpha-in-var in-var-prod-lens*)

lemma *U0α-comp-out-var* [*alpha*]: (*out-var x*) $;_L$ *U0α* = *out-var* (*x* $;_L$ *mrg-left*)
by (*simp add*: *U0α-def alpha-out-var id-wb-lens out-var-prod-lens*)

lemma *U1α-comp-in-var* [*alpha*]: (*in-var x*) $;_L$ *U1α* = *in-var x*
by (*simp add*: *U1α-def alpha-in-var in-var-prod-lens*)

lemma *U1α-comp-out-var* [*alpha*]: (*out-var x*) $;_L$ *U1α* = *out-var* (*x* $;_L$ *mrg-right*)
by (*simp add*: *U1α-def alpha-out-var id-wb-lens out-var-prod-lens*)

30.4 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* :: *'α merge* ⇒ (('α, 'α, ('α, 'α, 'α) *mrg*) *mrg*, 'α) *urel* (**M***3*'(-')) **where**
[*upred-defs*]: *ThreeWayMerge M* = (($*0*:**v**´ $=_u$ $*0*:**v** ∧ $*1*:**v**´ $=_u$ $*1*:*0*:**v** ∧ $<:**v**´ $=_u$ $<:**v**) ;; *M* ;; *U0* ∧ $*1*:**v**´ $=_u$ $*1*:*1*:**v** ∧ $<:**v**´ $=_u$ $<:**v**) ;; *M*

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

abbreviation $rotate_m$ **where** $rotate_m$ ≡ (*0*:**v**,*1*:*0*:**v**,*1*:*1*:**v**) := (&*1*:*0*:**v**,&*1*:*1*:**v**,&*0*:**v**)

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

definition *AssocMerge* :: *'α merge* ⇒ *bool* **where**
[*upred-defs*]: *AssocMerge M* = ($rotate_m$;; **M***3*(*M*) = **M***3*(*M*))

30.5 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* ≡ (*P* ;; *U0*) ∧ (*Q* ;; *U1*) ∧ $<´ $=_u$ $**v**

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 :: ('α, 'β) *urel* ⇒ (('α, 'β, 'γ) *mrg*, 'δ) *urel* ⇒ ('α, 'γ) *urel* ⇒ ('α, 'δ) *urel*
(- $\|_-$ - [*85*,*0*,*86*] *85*)
where [*upred-defs*]: *P* $\|_M$ *Q* = (*P* $\|_s$ *Q* ;; *M*)

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

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

30.6 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*)

lemma *unrest-merge-vars* [*unrest*]: $\$1{:}x' \sharp \lceil P \rceil_0\ \$<:x' \sharp \lceil P \rceil_0\ \$0{:}x' \sharp \lceil P \rceil_1\ \$<:x' \sharp \lceil P \rceil_1$
  by (*rel-auto*)+

30.7 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 coercsions. We need quite a number of variants to support this which are below.

lemma *U0-seq-subst*: $(P\ ;;\ U0)[\![\ll v \gg/\$0{:}x']\!] = (P[\![\ll v \gg/\$x']\!]\ ;;\ U0)$
  by (*rel-auto*)

lemma *U1-seq-subst*: $(P\ ;;\ U1)[\![\ll v \gg/\$1{:}x']\!] = (P[\![\ll v \gg/\$x']\!]\ ;;\ U1)$
  by (*rel-auto*)

lemma *lit-pbm-subst* [*usubst*]:
  fixes $x :: (- \Longrightarrow '\alpha)$
  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 $x :: (- \Longrightarrow '\alpha)$
  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 $x :: (- \Longrightarrow '\alpha)$
  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
    $\bigwedge$ $P\ Q\ M\ \sigma.\ \sigma(\$x \mapsto_s \mathit{numeral}\ n) \dagger (P \parallel_M Q) = \sigma \dagger ((P[\![\mathit{numeral}\ n/\$x]\!]) \parallel_{M[\![\mathit{numeral}\ n/\$<:x]\!]} (Q[\![\mathit{numeral}\ n/\$x]\!]))$
    $\bigwedge$ $P\ Q\ M\ \sigma.\ \sigma(\$x\acute{} \mapsto_s \mathit{numeral}\ n) \dagger (P \parallel_M Q) = \sigma \dagger (P \parallel_{M[\![\mathit{numeral}\ n/\$x\acute{}]\!]} Q)$
  by (*rel-auto*)+

30.8 Parallel-by-merge laws

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

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

lemma *par-by-merge-right-false* [*simp*]:
  $P \parallel_M \mathit{false} = \mathit{false}$
  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 $P \;;; \mathit{true} = \mathit{true}$ $Q \;;; \mathit{true} = \mathit{true}$
  shows $P \parallel_{\mathit{skip}_m} Q = \mathit{II}$
  using *assms* **by** (*rel-auto*)

lemma *skip-merge-swap*: $\mathit{swap}_m \;;; \mathit{skip}_m = \mathit{skip}_m$
  by (*rel-auto*)

lemma *par-sep-swap*: $P \parallel_s Q \;;; \mathit{swap}_m = Q \parallel_s P$
  by (*rel-auto*)

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

lemma *par-by-merge-commute-swap*:
  shows $P \parallel_M Q = Q \parallel_{\mathit{swap}_m \;;; M} P$
proof −
  have $Q \parallel_{\mathit{swap}_m \;;; M} P = ((((Q \;;; U0) \land (P \;;; U1) \land \$<:\mathbf{v}\acute{} =_u \$\mathbf{v}) \;;; \mathit{swap}_m) \;;; M)$
    by (*simp add*: *par-by-merge-def seqr-assoc*)
  also have ... $= (((Q \;;; U0 \;;; \mathit{swap}_m) \land (P \;;; U1 \;;; \mathit{swap}_m) \land \$<:\mathbf{v}\acute{} =_u \$\mathbf{v}) \;;; M)$
    by (*rel-auto*)
  also have ... $= (((Q \;;; U1) \land (P \;;; U0) \land \$<:\mathbf{v}\acute{} =_u \$\mathbf{v}) \;;; M)$
    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*)

lemma *par-by-merge-mono*:
  assumes $P_1 \sqsubseteq P_2$ $Q_1 \sqsubseteq Q_2$
  shows $P_1 \parallel_M Q_1 \sqsubseteq P_2 \parallel_M Q_2$
  by (*meson assms dual-order.trans par-by-merge-mono-1 par-by-merge-mono-2*)

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 $(P \parallel_M Q) \parallel_M R = ((P \;;;\; U0) \wedge (Q \;;;\; U0 \;;;\; U1) \wedge (R \;;;\; U1 \;;;\; U1) \wedge \$<:\mathbf{v}' =_u \$\mathbf{v}) \;;;\; \mathbf{M}3(M)$
    by (*rel-blast*)
  also have ... $= ((P \;;;\; U0) \wedge (Q \;;;\; U0 \;;;\; U1) \wedge (R \;;;\; U1 \;;;\; U1) \wedge \$<:\mathbf{v}' =_u \$\mathbf{v}) \;;;\; rotate_m \;;;\; \mathbf{M}3(M)$
    using *AssocMerge-def assms*(*2*) **by** *force*
  also have ... $= ((Q \;;;\; U0) \wedge (R \;;;\; U0 \;;;\; U1) \wedge (P \;;;\; U1 \;;;\; U1) \wedge \$<:\mathbf{v}' =_u \$\mathbf{v}) \;;;\; \mathbf{M}3(M)$
    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-or-left*:
  $(P \vee Q) \parallel_M R = (P \parallel_M R \vee Q \parallel_M R)$
  by (*rel-auto*)

theorem *par-by-merge-or-right*:
  $P \parallel_M (Q \vee R) = (P \parallel_M Q \vee P \parallel_M R)$
  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*)

30.9 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*]: $M[a|b]_\sigma = (\$\mathbf{v}' =_u (\$\!<\!:\mathbf{v} \oplus \$0{:}\mathbf{v}\ on\ \&a) \oplus \$1{:}\mathbf{v}\ on\ \&b)$

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*:
$P\ |a|b|_\sigma\ Q = (\exists\ (st_0,\ st_1) \cdot P[\![\ll st_0 \gg/\$\mathbf{v}']\!] \wedge Q[\![\ll st_1 \gg/\$\mathbf{v}']\!] \wedge \$\mathbf{v}' =_u (\$\mathbf{v} \oplus \ll st_0 \gg\ on\ \&a) \oplus \ll st_1 \gg\ on\ \&b)$
by (*rel-auto*)

lemma *StateParallel-form'*:
assumes *vwb-lens a vwb-lens b a* $\bowtie$ *b*
shows $P\ |a|b|_\sigma\ Q = \{\&a,\&b\}{:}[(P \upharpoonright_v \{\$\mathbf{v},\$a'\}) \wedge (Q \upharpoonright_v \{\$\mathbf{v},\$b'\})]$
using *assms*
apply (*simp add*: *StateParallel-form*, *rel-auto*)
   apply (*metis vwb-lens-wb wb-lens-axioms-def wb-lens-def*)
  apply (*metis vwb-lens-wb wb-lens.get-put*)
 apply (*simp add*: *lens-indep-comm*)
apply (*metis* (*no-types*, *hide-lams*) *lens-indep-comm vwb-lens-wb wb-lens-def weak-lens.put-get*)
done

We can frame all the variables that the parallel operator refers to

lemma *StateParallel-frame*:
assumes *vwb-lens a vwb-lens b a* $\bowtie$ *b*
shows $\{\&a,\&b\}{:}[P\ |a|b|_\sigma\ Q] = P\ |a|b|_\sigma\ Q$
using *assms*
apply (*simp add*: *StateParallel-form*, *rel-auto*)
using *lens-indep-comm* **apply** *fastforce+*
done

Parallel Hoare logic rule. This employs something similar to separating conjunction in the postcondition, but we explicitly require that the two conjuncts only refer to variables on the left and right of the parallel composition explicitly.

theorem *StateParallel-hoare* [*hoare*]:
  assumes $\{\!|c|\!\}P\{\!|d_1|\!\}_u$ $\{\!|c|\!\}Q\{\!|d_2|\!\}_u$ $a \bowtie b$ $a \natural d_1$ $b \natural d_2$
  shows $\{\!|c|\!\}P\ |a|b|_\sigma\ Q\{\!|d_1 \wedge d_2|\!\}_u$
proof –
  — Parallelise the specification
  from *assms*(*4*,*5*)
  have *1*:$(\lceil c\rceil_< \Rightarrow \lceil d_1 \wedge d_2\rceil_>) \sqsubseteq (\lceil c\rceil_< \Rightarrow \lceil d_1\rceil_>)\ |a|b|_\sigma\ (\lceil c\rceil_< \Rightarrow \lceil d_2\rceil_>)$ (**is** *?lhs* ⊑ *?rhs*)
    by (*simp add*: *StateParallel-form*, *rel-auto*, *metis assms*(*3*) *lens-indep-comm*)
  — Prove Hoare rule by monotonicity of parallelism
  have *2*:*?rhs* $\sqsubseteq P\ |a|b|_\sigma\ Q$
  proof (*rule par-by-merge-mono*)
    show $(\lceil c\rceil_< \Rightarrow \lceil d_1\rceil_>) \sqsubseteq P$
      using *assms*(*1*) *hoare-r-def* **by** *auto*
    show $(\lceil c\rceil_< \Rightarrow \lceil d_2\rceil_>) \sqsubseteq Q$
      using *assms*(*2*) *hoare-r-def* **by** *auto*
  qed
  show *?thesis*
    unfolding *hoare-r-def* **using** *1 2 order-trans* **by** *auto*
qed

Specialised version of the above law where an invariant expression referring to variables outside the frame is preserved.

theorem *StateParallel-frame-hoare* [*hoare*]:
  assumes *vwb-lens a vwb-lens b* $a \bowtie b$ $a \natural d_1$ $b \natural d_2$ $a \sharp c_1$ $b \sharp c_1$ $\{\!|c_1 \wedge c_2|\!\}P\{\!|d_1|\!\}_u$ $\{\!|c_1 \wedge c_2|\!\}Q\{\!|d_2|\!\}_u$
  shows $\{\!|c_1 \wedge c_2|\!\}P\ |a|b|_\sigma\ Q\{\!|c_1 \wedge d_1 \wedge d_2|\!\}_u$
proof –
  have $\{\!|c_1 \wedge c_2|\!\}\{\&a,\&b\}:[P\ |a|b|_\sigma\ Q]\{\!|c_1 \wedge d_1 \wedge d_2|\!\}_u$
    by (*auto intro*!: *frame-hoare-r′ StateParallel-hoare simp add*: *assms unrest plus-vwb-lens*)
  thus *?thesis*
    by (*simp add*: *StateParallel-frame assms*)
qed

end

31 Collections

theory *utp-collection*
  imports *utp-lift-pretty utp-pred*
begin

31.1 Partial Lens Definedness

definition *src-pred* :: $('a \Longrightarrow 's) \Rightarrow 's\ upred$ ($\mathbf{S}$′(-′)) **where**
[*upred-defs*]: *src-pred x* = $(\&\mathbf{v} \in_u \ll\mathcal{S}_x\gg)$

lemma *wb-lens-src-true* [*simp*]: *wb-lens x* $\Longrightarrow \mathbf{S}(x) = true$
  by (*rel-simp*, *simp add*: *wb-lens.source-UNIV*)

31.2 Indexed Lenses

definition *ind-lens* :: $('i \Rightarrow ('a \Longrightarrow 's)) \Rightarrow ('i, 's)\ uexpr \Rightarrow ('a \Longrightarrow 's)$ **where**
[*lens-defs*]: *ind-lens f x* = (| *lens-get* = $(\lambda\ s.\ get_{f\ (\llbracket x\rrbracket_e\ s)}\ s)$, *lens-put* = $(\lambda\ s\ v.\ put_{f\ (\llbracket x\rrbracket_e\ s)}\ s\ v)$ |)

lemma *ind-lens-mwb* [*simp*]: $\llbracket \bigwedge i.\ mwb\text{-}lens\ (F\ i); \bigwedge i.\ unrest\ (F\ i)\ x \rrbracket \Longrightarrow mwb\text{-}lens\ (ind\text{-}lens\ F\ x)$
  by (*unfold-locales*, *auto simp add*: *lens-defs lens-indep.lens-put-irr2 unrest-uexpr.rep-eq*)

lemma *ind-lens-vwb* [*simp*]: $\llbracket \bigwedge i.\ vwb\text{-}lens\ (F\ i);\ \bigwedge i.\ unrest\ (F\ i)\ x\ \rrbracket \Longrightarrow vwb\text{-}lens\ (ind\text{-}lens\ F\ x)$
  by (*unfold-locales*, *auto simp add*: *lens-defs lens-indep.lens-put-irr2 unrest-uexpr.rep-eq*)

lemma *src-ind-lens*: $\llbracket \bigwedge i.\ unrest\ (f\ i)\ e\ \rrbracket \Longrightarrow \mathcal{S}_{ind\text{-}lens\ f\ e} = \{s.\ s \in \mathcal{S}_{f\ (\llbracket e \rrbracket_e\ s)}\}$
  apply (*auto simp add*: *lens-defs lens-source-def unrest unrest-uexpr.rep-eq*)
  apply (*blast*)
  apply *metis*
  done

31.3 Overloaded Collection Lens

consts *collection-lens* :: $'k \Rightarrow ('a \Longrightarrow 's)$

definition [*lens-defs*]: *fun-collection-lens* = *fun-lens*
definition [*lens-defs*]: *pfun-collection-lens* = *pfun-lens*
definition [*lens-defs*]: *ffun-collection-lens* = *ffun-lens*
definition [*lens-defs*]: *list-collection-lens* = *list-lens*

lemma *vwb-fun-collection-lens* [*simp*]: *vwb-lens* (*fun-collection-lens k*)
  by (*simp add*: *fun-collection-lens-def fun-vwb-lens*)

lemma *mwb-pfun-collection-lens* [*simp*]: *mwb-lens* (*pfun-collection-lens k*)
  by (*simp add*: *pfun-collection-lens-def*)

lemma *mwb-ffun-collection-lens* [*simp*]: *mwb-lens* (*ffun-collection-lens k*)
  by (*simp add*: *ffun-collection-lens-def*)

lemma *mwb-list-collection-lens* [*simp*]: *mwb-lens* (*list-collection-lens i*)
  by (*simp add*: *list-collection-lens-def list-mwb-lens*)

lemma *source-list-collection-lens*: $\mathcal{S}_{list\text{-}collection\text{-}lens\ i} = \{xs.\ i < length\ xs\}$
  by (*simp add*: *list-collection-lens-def source-list-lens*)

adhoc-overloading
  collection-lens fun-collection-lens **and**
  collection-lens pfun-collection-lens **and**
  collection-lens ffun-collection-lens **and**
  collection-lens list-collection-lens

31.4 Syntax for Collection Lens

abbreviation *ind-lens-poly f x i* ≡ *ind-lens* (λ *k*. *f k* $;_L$ *x*) *i*

utp-lift-notation *ind-lens-poly* (*0 1*)

syntax
  -svid-collection :: *svid* ⇒ *logic* ⇒ *svid* (-[-] [*999*, *0*] *999*)

translations
  -svid-collection x i == *CONST ind-lens-poly CONST collection-lens x i*

lemma *src-list-collection-lens* [*simp*]:
  $\llbracket\ vwb\text{-}lens\ x;\ x \sharp\ i\ \rrbracket \Longrightarrow \mathbf{S}(ind\text{-}lens\text{-}poly\ list\text{-}collection\text{-}lens\ x\ i) = U(i < length(\&x))$
  apply (*simp add*: *upred-defs src-ind-lens unrest source-list-collection-lens source-lens-comp*)
  apply (*transfer*, *auto simp add*: *fun-eq-iff lens-defs wb-lens.source-UNIV*)

done

end

32 Definition Command for UTP

theory *utp-definition*
  imports *utp-pred*
  keywords *utp-def* :: *thy-decl-block*
begin

A first attempt at a definition command for UTP that (1) uses the lifting parser for the expression on the RHS and (2) adds the definitional equation to *'?x'* $= All$ $[\![?x]\!]_e$

$\boldsymbol{U}(true) = \bot$

$\boldsymbol{U}(false) = \top$

$(\wedge) = (\sqcup)$

$(\vee) = (\sqcap)$

$unot = uminus$

$diff\text{-}upred = (-)$

$par\text{-}subst \equiv map\text{-}fun\ Rep\text{-}uexpr\ (map\text{-}fun\ id\ (map\text{-}fun\ id\ (map\text{-}fun\ Rep\text{-}uexpr\ mk_e)))\ (\lambda\sigma_1\ A\ B\ \sigma_2\ s.\ s \lhd_A \sigma_1\ s \lhd_B \sigma_2\ s)$

$unrest\text{-}usubst \equiv map\text{-}fun\ id\ (map\text{-}fun\ Rep\text{-}uexpr\ id)\ (\lambda x\ \sigma.\ \forall\,\varrho\ v.\ \sigma\ (put_x\ \varrho\ v) = put_x\ (\sigma\ \varrho)\ v)$

$uIf = If$

$\boldsymbol{U}(0) = \boldsymbol{U}(0{::}?'a)$

$\boldsymbol{U}(1) = \boldsymbol{U}(1{::}?'a)$

$?u + ?v = bop\ (+)\ ?u\ ?v$

$(?P < ?Q) = (?P \leq ?Q \wedge \neg\ ?Q \leq ?P)$

$set\text{-}of\ ?t = UNIV$

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

$?u - ?v = bop\ (-)\ ?u\ ?v$

$?u * ?v = bop\ (*)\ ?u\ ?v$

$?u\ div\ ?v = bop\ (div)\ ?u\ ?v$

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

$?u\ mod\ ?v = bop\ (mod)\ ?u\ ?v$

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

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

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

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

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

$?\sigma -_s ?x = ?\sigma(?x \mapsto_s \boldsymbol{U}(\&?x))$

$?\sigma \rhd_s ?x = [?x \mapsto_s \langle ?\sigma\rangle_s\ ?x].$

ML ‹
structure UTP-Def =
struct

```
  fun mk-utp-def-eq ctx term =
    case (Type.strip-constraints term) of
      Const (@{const-name HOL.eq}, b) $ c $ t =>
        @{const Trueprop} $ (Const (@{const-name HOL.eq}, b) $ c $ utp-lift ctx t) |
      - => raise Match;

  val upred-defs = [[Token.make-string (Binding.name-of @{binding upred-defs}, Position.none)]];

  fun utp-def attr decl term ctx =
    Specification.definition
      (Option.map (fn x => fst (Proof-Context.read-var x ctx)) decl) [] []
      ((fst attr, map (Attrib.check-src ctx) (upred-defs @ snd attr)), mk-utp-def-eq ctx term) ctx

end

val - =
let
  open UTP-Def;
in
  Outer-Syntax.local-theory command-keyword‹utp-def› UTP constant definition
    (Scan.option Parse-Spec.constdecl −− (Parse-Spec.opt-thm-name : −− Parse.prop) −−
      Parse-Spec.if-assumes −− Parse.for-fixes >> (fn (((decl, (attr, term)), -), -) =>
        (fn ctx => snd (utp-def attr decl (Syntax.parse-term ctx term) ctx))))
end
›
```

end

33 UTP Schema Types

theory *utp-schema*
  imports *utp-definition*
  keywords *schema* :: *thy-decl-block*
begin

Create a type with invariants attached; similar to a Z schema.

ML ‹
```
val - =
  Outer-Syntax.command @{command-keyword schema} define a new schema type
    (Parse-Spec.overloaded −− (Parse.type-args-constrained −− Parse.binding) −−
      (@{keyword =} |−− Scan.option (Parse.typ −−| @{keyword +}) −−
           Scan.repeat1 Parse.const-binding) −− Scan.optional (@{keyword where} |−− (Scan.repeat1
(Scan.option (Parse.binding −−| Parse.$$$ :) |−− Parse.term))) [true]
    >> (fn (((overloaded, x), (y, z)), ts) =>
        let (∗ Get the new type name ∗)
            val n = Binding.name-of (snd x)
            (∗ Produce a list of type variables ∗)
            val varl = fold (fn - => fn y => -,  ˆ y) (1 upto length (fst x)) 'a
            (∗ Name for the new invariant ∗)
            val invn = n ˆ -inv
            val itb = Binding.make (invn ˆ -def, Position.none)
            val upred = Lexicon.unmark-type @{type-syntax upred}
            val ib = (SOME (Binding.make (invn, Position.none), SOME ((( ˆ varl ˆ ) ˆ n ˆ -scheme)
ˆ upred), NoSyn))
            open HOLogic in
        Toplevel.theory
          (Lens-Utils.add-alphabet-cmd {overloaded = overloaded} x y z
           #> Named-Target.theory-map
              (fn ctx =>
               let val invs = Library.foldr1 HOLogic.mk-conj (map (Syntax.parse-term ctx) ts)
                   val sinv = case y of
                      NONE => invs |
                      SOME t => case (Syntax.parse-typ ctx t) of
                        Type (n, -) => (case (Syntax.parse-term ctx (n ^ -inv)) of
                          Const (syntax-const ‹-type-constraint-›, -) $ Const (n', -) => HOLogic.mk-conj
(Const (n', dummyT), invs) | - => invs) |
                        - => invs
               in
                 snd (UTP-Def.utp-def (itb, []) ib (mk-eq (Free (invn, dummyT), sinv)) ctx)
               end)
           #> Named-Target.theory-map
              (fn ctx =>
               let val Const (cn, -) = Syntax.read-term ctx invn
                   val varl =
                     if (length (fst x) = 0)
                     then
                     else ( ^ foldr1 (fn (x, y) => -,  ^ x) (map (fn - => -) (1 upto length (fst x))) ^ )
                   val ty = Syntax.read-typ ctx (varl ^ n ^   ^ upred) in
               Specification.abbreviation Syntax.mode-default (SOME (Binding.make (n, Position.none),
SOME ty, NoSyn)) [] (Logic.mk-equals (Free (n, dummyT), Const (cn, dummyT))) false ctx
               end)
)
        end));
›
```

end

34 Meta-theory for the Standard Core

theory *utp*
imports
  utp-var
  utp-expr
  utp-expr-insts
  utp-expr-funcs
  utp-unrest
  utp-usedby
  utp-subst
  utp-meta-subst
  utp-alphabet
  utp-lift
  utp-pred
  utp-pred-laws
  utp-recursion
  utp-dynlog
  utp-rel
  utp-rel-laws
  utp-sequent
  utp-state-parser
  utp-lift-parser

utp-lift-pretty
utp-sym-eval
utp-tactics
utp-hoare
utp-wlp
utp-wp
utp-sp
utp-theory
utp-concurrency
utp-collection
utp-rel-opsem
utp-blocks
utp-definition
utp-schema
begin recall-syntax end

35 Overloaded Expression Constructs

theory *utp-expr-ovld*
imports *utp*
begin

35.1 Overloadable Constants

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 :: $'f \Rightarrow 'k \Rightarrow 'v$
— Overriding
uovrd :: $'f \Rightarrow 'f \Rightarrow 'f$
— 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* ⇒ (*'k* ⇒ *'v*) ⇒ (*'k*, *'v*) *ffun* **where**
ffun-entries d f = *graph-ffun* {(*k*, *f k*) | *k*. *k* ∈ *d*}

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

adhoc-overloading
uempty 0 **and**
uapply rel-apply **and** *uapply fun-apply* **and** *uapply nth* **and** *uapply pfun-app* **and** *uapply ffun-app* **and**
uovrd rel-override **and** *uovrd plus*
uupd rel-update **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**
uran Range **and** *uran pran* **and** *uran fran* **and** *uran set* **and**
udomres rel-domres **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*

35.2 Syntax Translations

syntax
-uundef :: *logic* ($\bot_u$)
-umap-empty :: *logic* ($[]_u$)
-uapply :: (*'a* ⇒ *'b*, *'α*) *uexpr* ⇒ *utuple-args* ⇒ (*'b*, *'α*) *uexpr* ($-'(-')_a$ [*999*,*0*] *999*)
-uovrd :: *logic* ⇒ *logic* ⇒ *logic* (**infixl** ⊕ *65*)
-umaplet :: [*logic*, *logic*] => *umaplet* (- /↦/ -)
:: *umaplet* => *umaplets* (-)
-UMaplets :: [*umaplet*, *umaplets*] => *umaplets* (-,/ -)
-UMapUpd :: [*logic*, *umaplets*] => *logic* ($-/'(-')_u$ [*900*,*0*] *900*)
-UMap :: *umaplets* => *logic* ($(1[-]_u)$)
-ucard :: *logic* ⇒ *logic* ($\#_u'(-')$)
-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*)
-uentries :: *logic* ⇒ *logic* ⇒ *logic* ($entr_u'(-,-')$)

translations
— Pretty printing for adhoc-overloaded constructs
$f(x)_a$ <= *CONST uapply f x*
f ⊕ *g* <= *CONST uovrd f g*
$dom_u(f)$ <= *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*
0 <= *CONST uempty* — We have to do this so we don't see uempty. Is there a better way of printing?

— 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*
$f \oplus g$ == *CONST bop CONST uovrd f g*
$\#_u(xs)$ == *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*

35.3 Simplifications

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

lemma *lit-plus-appl* [*lit-norm*]: $\ll(+)\gg(x)_a(y)_a = x + y$ **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)
lemma *lit-minus-appl* [*lit-norm*]: $\ll(-)\gg(x)_a(y)_a = x - y$ **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)
lemma *lit-mult-appl* [*lit-norm*]: $\ll times \gg(x)_a(y)_a = x * y$ **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)
lemma *lit-divide-apply* [*lit-norm*]: $\ll(/)\gg(x)_a(y)_a = x / y$ **by** (*simp add*: *uexpr-defs*, *transfer*, *simp*)

lemma *pfun-entries-apply* [*simp*]:
  $(entr_u(d,f) :: (('k, 'v)\ pfun, '\alpha)\ uexpr)(i)_a = ((\ll f \gg(i)_a) \lhd i \in_u d \rhd \bot_u)$
  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 $(m(k \mapsto v)_u)(i)_a = v \lhd i =_u k \rhd m(i)_a$
  by (*rel-auto*)

35.4 Indexed Assignment

syntax
  — Indexed assignment
  -assignment-upd :: *svid* ⇒ *logic* ⇒ *logic* ⇒ *logic* ((-[-] :=/ -) [*63*, *0*, *0*] *62*)

translations
  — Indexed assignment uses the overloaded collection update function *uupd*.
  -assignment-upd x k v => $x := \&x(k \mapsto v)_u$

end

36 Meta-theory for the Standard Core with Overloaded Constructs

theory *utp-full*
  imports *utp utp-expr-ovld*
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] R.-J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[4] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In *Proc. 5th Intl. Conf. on Mathematical Knowledge Management (MKM)*, volume 4108 of *LNCS*, pages 31–43. Springer, 2006.

[5] C. Ballarin et al. The Isabelle/HOL Algebra Library. *Isabelle/HOL*, October 2017. https://isabelle.in.tum.de/library/HOL/HOL-Algebra/document.pdf.

[6] A. Cavalcanti and J. Woodcock. A tutorial introduction to designs in unifying theories of programming. In *Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*, pages 40–66. Springer, 2004.

[7] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.

[8] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[9] 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.

[10] 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.

[11] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.

[12] S. Foster, J. Baxter, A. Cavalcanti, A. Miyazawa, and J. Woodcock. Automating verification of state machines with reactive designs and Isabelle/UTP. In *Proc. 15th. Intl. Conf. on Formal Aspects of Component Software*, volume 11222 of *LNCS*. Springer, October 2018.

[13] S. Foster, K. Ye, A. Cavalcanti, and J. Woodcock. Calculational verification of reactive programs with reactive relations and Kleene algebra. In *Proc. 17th Intl. Conf. on Relational and Algebraic Methods in Computer Science (RAMICS)*, volume 11194 of *LNCS*. Springer, October 2018.

[14] S. Foster and F. Zeyda. Optics. *Archive of Formal Proofs*, May 2017. http://isa-afp.org/entries/Optics.html, Formal proof development.

[15] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *UTP*, LNCS 8963, pages 21–41. Springer, 2014.

[16] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC)*, volume 9965 of *LNCS*. Springer, 2016.

[17] D. Harel. Dynamic logic. In *Handbook of Philosophical Logic*, volume 165 of *SYLI*, pages 497–604. Springer, 1984.

[18] E. C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14:133–158, 1990.

[19] E. C. R. Hehner. *A Practical Theory of Programming*. Springer, 1993.

[20] E. C. R. Hehner and A. J. Malton. Termination conventions and comparative semantics. *Acta Informatica*, 25, 1988.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[22] T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[23] 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.

[24] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997.

[25] C. Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.

[26] M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In *UTP 2006*, volume 4010 of *LNCS*, pages 123–140. Springer, 2007.

[27] 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.

[28] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *UTP*, LNCS 10134. Springer, 2016.