Shallow Expressions

Simon Foster
University of York, UK

simon.foster@york.ac.uk

February 6, 2026

Abstract

Most verification techniques use expressions, for example when assigning to variables or forming assertions over the state. Deep embeddings provide such expressions using a datatype, which allows queries over the syntax, such as calculating the free variables, and performing substitutions. Shallow embeddings, in contrast, model expressions as query functions over the state type, and are more amenable to automating verification tasks. The goal of this library is provide an intuitive implementation of shallow expressions, which nevertheless provides many of the benefits of a deep embedding. We harness the Optics library to provide an algebraic semantics for state variables, and use syntax translations to provide an intuitive lifted expression syntax. Furthermore, we provide a variety of meta-logic-style queries on expressions, such as dependencies on a state variable, and substitution of a variable for an expression. We also provide proof methods, based on the simplifier, to automate the associated proof tasks.

Contents

1 Introduction

This session provides a library for expressions in shallow embeddings, based on the Optics package [5]. It provides the following key features:

1. Parse and print translations for converting between intuitive expression syntax, and state functions using lenses to model variables. The translation uses the type system to determine whether each free variable in an expression is (1) a lens (i.e. a state variable); (2) another expression; (3) a literal, and gives the correct interpretation for each. The lifting mechanism is manifested through the bracket notation, $(\_)_e$, but can usually be hidden behind syntax.

2. Syntax for complex state variable constructions, using the lens operators [5], such as simultaneous assignment, hierarchical state, and initial/final state variables.

3. The "unrestriction" predicate [7, 3], $x \sharp e$, which characterises whether an expression e depends on a particular variable x or not, based on the lens laws. It can often be used as a replacement for syntactically checking for free variables in a deep embedding, as needed in several verification techniques.

4. Semantic substitution of variables for expressions, $e[v/x]$, with support for evaluation from the simplifier. A notation is provided for expressing substitution objects as a sequence of simultaneous variable assignments, $[x_1 \leadsto e_1, x_2 \leadsto e_2, \cdots]$.

5. Collection lenses, $x[i]$, which can be used to model updating a component of a larger structure, for example mutating an element of an array by its index.

6. Supporting transformations and constructors for expressions, such as state extension, state restriction, and quantifiers.

The majority of these concepts have been adapted from Isabelle/UTP [3], but have been generalised for use in other Isabelle-based verification tools. Several proof methods are provided, such as for discharging unrestriction conditions (`unrest`) and evaluating substitutions (`subst_eval`).

The Shallow Expressions library has been applied in the IsaVODEs tool [6], for verifying hybrid systems, and Isabelle/ITrees [4], for verification of process-algebraic languages.

2 Variables as Lenses

theory *Variables*

imports *Optics.Optics*
begin

Here, we implement foundational constructors and syntax translations for state variables, using lens manipulations, for use in shallow expressions.

2.1 Constructors

The following bundle allows us to override the colon operator, which we use for variable namespaces.

bundle *Expression-Syntax*
begin

no-notation
Set.member (‹′(:′)›) **and**
Set.member (‹(‹*notation*=‹*infix* :››-/ : -)› [*51*, *51*] *50*)

end

unbundle *Expression-Syntax*

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

Lenses [3] 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.

definition *univ-var* :: $('\alpha \Longrightarrow '\alpha)$ (**v**) **where**
[*lens-defs*]: $univ\text{-}var = 1_L$

lemma *univ-var-vwb* [*simp*]: *vwb-lens univ-var*
by (*simp add*: *univ-var-def*)

definition *univ-alpha* :: *′s scene* **where**
[*lens-defs*]: $univ\text{-}alpha = \top_S$

definition *emp-alpha* :: *′s scene* **where**
[*lens-defs*]: $emp\text{-}alpha = \bot_S$

definition *var-alpha* :: $('a \Longrightarrow 's) \Rightarrow 's\ scene$ **where**
[*lens-defs*]: $var\text{-}alpha\ x = lens\text{-}scene\ x$

definition *ns-alpha* :: $('b \Longrightarrow 'c) \Rightarrow ('a \Longrightarrow 'b) \Rightarrow 'a \Longrightarrow 'c$ **where**
[*lens-defs*]: $ns\text{-}alpha\ a\ x = x\ ;_L\ a$

definition *var-fst* :: $('a \times 'b \Longrightarrow 's) \Rightarrow ('a \Longrightarrow 's)$ **where**
[*lens-defs*]: $var\text{-}fst\ x = fst_L\ ;_L\ x$

definition *var-snd* :: $('a \times 'b \Longrightarrow 's) \Rightarrow ('b \Longrightarrow 's)$ **where**
[*lens-defs*]: *var-snd* $x = snd_L\ ;_L\ x$

definition *var-pair* :: $('a \Longrightarrow 's) \Rightarrow ('b \Longrightarrow 's) \Rightarrow ('a \times 'b \Longrightarrow 's)$ **where**
[*lens-defs*]: *var-pair* $x\ y = x +_L y$

abbreviation *var-member* :: $('a \Longrightarrow 's) \Rightarrow 's$ *scene* $\Rightarrow$ *bool* (**infix** $\in_v$ *50*) **where**
$x \in_v A \equiv$ *var-alpha* $x \leq A$

lemma *ns-alpha-weak* [*simp*]: ⟦ *weak-lens a*; *weak-lens x* ⟧ ⟹ *weak-lens* (*ns-alpha a x*)
  by (*simp add*: *ns-alpha-def comp-weak-lens*)

lemma *ns-alpha-mwb* [*simp*]: ⟦ *mwb-lens a*; *mwb-lens x* ⟧ ⟹ *mwb-lens* (*ns-alpha a x*)
  by (*simp add*: *ns-alpha-def comp-mwb-lens*)

lemma *ns-alpha-vwb* [*simp*]: ⟦ *vwb-lens a*; *vwb-lens x* ⟧ ⟹ *vwb-lens* (*ns-alpha a x*)
  by (*simp add*: *ns-alpha-def comp-vwb-lens*)

lemma *ns-alpha-indep-1* [*simp*]: $a \bowtie b \Longrightarrow$ *ns-alpha a x* $\bowtie$ *ns-alpha b y*
  by (*simp add*: *lens-indep-left-ext lens-indep-right-ext ns-alpha-def*)

lemma *ns-alpha-indep-2* [*simp*]: $a \bowtie y \Longrightarrow$ *ns-alpha a x* $\bowtie y$
  by (*simp add*: *lens-indep-left-ext ns-alpha-def*)

lemma *ns-alpha-indep-3* [*simp*]: $x \bowtie b \Longrightarrow x \bowtie$ *ns-alpha b y*
  by (*simp add*: *lens-indep-sym*)

lemma *ns-alpha-indep-4* [*simp*]: ⟦ *mwb-lens a*; $x \bowtie y$ ⟧ ⟹ *ns-alpha a x* $\bowtie$ *ns-alpha a y*
  by (*simp add*: *ns-alpha-def*)

lemma *var-fst-mwb* [*simp*]: *mwb-lens x* ⟹ *mwb-lens* (*var-fst x*)
  by (*simp add*: *var-fst-def comp-mwb-lens*)

lemma *var-snd-mwb* [*simp*]: *mwb-lens x* ⟹ *mwb-lens* (*var-snd x*)
  by (*simp add*: *var-snd-def comp-mwb-lens*)

lemma *var-fst-vwb* [*simp*]: *vwb-lens x* ⟹ *vwb-lens* (*var-fst x*)
  by (*simp add*: *var-fst-def comp-vwb-lens*)

lemma *var-snd-vwb* [*simp*]: *vwb-lens x* ⟹ *vwb-lens* (*var-snd x*)
  by (*simp add*: *var-snd-def comp-vwb-lens*)

lemma *var-fst-indep-1* [*simp*]: $x \bowtie y \Longrightarrow$ *var-fst x* $\bowtie y$
  by (*simp add*: *var-fst-def lens-indep-left-ext*)

lemma *var-fst-indep-2* [*simp*]: $x \bowtie y \Longrightarrow x \bowtie$ *var-fst* y
  by (*simp add*: *var-fst-def lens-indep-right-ext*)

lemma *var-snd-indep-1* [*simp*]: $x \bowtie y \Longrightarrow$ *var-snd* $x \bowtie y$
  by (*simp add*: *var-snd-def lens-indep-left-ext*)

lemma *var-snd-indep-2* [*simp*]: $x \bowtie y \Longrightarrow x \bowtie$ *var-snd* y
  by (*simp add*: *var-snd-def lens-indep-right-ext*)

lemma *mwb-var-pair* [*simp*]: $[\![$ *mwb-lens* x; *mwb-lens* y; $x \bowtie y$ $]\!] \Longrightarrow$ *mwb-lens* (*var-pair* x y)
  by (*simp add*: *var-pair-def plus-mwb-lens*)

lemma *vwb-var-pair* [*simp*]: $[\![$ *vwb-lens* x; *vwb-lens* y; $x \bowtie y$ $]\!] \Longrightarrow$ *vwb-lens* (*var-pair* x y)
  by (*simp add*: *var-pair-def*)

lemma *var-pair-pres-indep* [*simp*]:
  $[\![$ $x \bowtie y$; $x \bowtie z$ $]\!] \Longrightarrow x \bowtie$ *var-pair* y z
  $[\![$ $x \bowtie y$; $x \bowtie z$ $]\!] \Longrightarrow$ *var-pair* y $z \bowtie x$
  by (*simp-all add*: *var-pair-def lens-indep-sym*)

definition *res-alpha* :: $('a \Longrightarrow 'b) \Rightarrow ('c \Longrightarrow 'b) \Rightarrow 'a \Longrightarrow 'c$ **where**
[*lens-defs*]: *res-alpha* x a = x $/_L$ a

lemma *idem-scene-var* [*simp*]:
  vwb-lens $x \Longrightarrow$ *idem-scene* (*var-alpha* x)
  by (*simp add*: *lens-defs*)

lemma *var-alpha-combine*: $[\![$ *vwb-lens* x; *vwb-lens* y; $x \bowtie y$ $]\!] \Longrightarrow$ *var-alpha* $x \sqcup_S$ *var-alpha* y = *var-alpha* $(x +_L y)$
  by (*simp add*: *lens-plus-scene var-alpha-def*)

lemma *var-alpha-indep* [*simp*]:
  assumes *vwb-lens* x *vwb-lens* y
  shows *var-alpha* $x \bowtie_S$ *var-alpha* $y \longleftrightarrow x \bowtie y$
  by (*simp add*: *assms*(*1*) *assms*(*2*) *lens-indep-scene var-alpha-def*)

lemma *pre-var-indep-prod* [*simp*]: $x \bowtie a \Longrightarrow$ *ns-alpha* fst_L $x \bowtie a \times_L b$
  using *lens-indep.lens-put-irr2*
  by (*unfold-locales*, *force simp add*: *lens-defs prod.case-eq-if lens-indep-comm*)+

lemma *post-var-indep-prod* [*simp*]: $x \bowtie b \Longrightarrow$ *ns-alpha* snd_L $x \bowtie a \times_L b$
  using *lens-indep.lens-put-irr2*
  by (*unfold-locales*, *force simp add*: *lens-defs prod.case-eq-if lens-indep-comm*)+

lemma *lens-indep-impl-scene-indep-var* [*simp*]:
  $(X \bowtie Y) \Longrightarrow$ *var-alpha* $X \bowtie_S$ *var-alpha* Y
  by (*simp add*: *var-alpha-def*)

declare *lens-scene-override* [*simp*]
declare *uminus-scene-twice* [*simp*]

lemma *var-alpha-override* [*simp*]:
  mwb-lens $X \Longrightarrow s_1 \oplus_S s_2$ *on var-alpha* $X = s_1 \oplus_L s_2$ *on* X
  by (*simp add*: *var-alpha-def*)

lemma *var-alpha-indep-compl* [*simp*]:
  assumes *vwb-lens x vwb-lens y*
  shows *var-alpha* $x \bowtie_S -$ *var-alpha* $y \longleftrightarrow x \subseteq_L y$
  by (*simp add*: *assms scene-le-iff-indep-inv sublens-iff-subscene var-alpha-def*)

lemma *var-alpha-subset* [*simp*]:
  assumes *vwb-lens x vwb-lens y*
  shows *var-alpha* $x \leq$ *var-alpha* $y \longleftrightarrow x \subseteq_L y$
  by (*simp add*: *assms*(*1*) *assms*(*2*) *sublens-iff-subscene var-alpha-def*)

2.2 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** *salpha* **and** *sframe-enum* **and** *sframe*

These non-terminals correspond to the following syntactic entities. Non-terminal *svid* is an atomic variable identifier, and *svids* is a list of identifier. *salpha* is an alphabet or set of variables. *sframe* is a frame. 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* ⇒ *svids* (-)
  -svid-list :: *svid* ⇒ *svids* ⇒ *svids* (-,/ -)
  -svid-alpha :: *svid* (**v**)
  -svid-index :: *id-position* ⇒ *logic* ⇒ *svid* (-′(-′) [*999*] *999*)
  -svid-tuple :: *svids* ⇒ *svid* (′(-′))
  -svid-dot :: *svid* ⇒ *svid* ⇒ *svid* (-:- [*999*,*998*] *998*)
  -svid-res :: *svid* ⇒ *svid* ⇒ *svid* (-↾- [*999*,*998*] *998*)
  -svid-pre :: *svid* ⇒ *svid* (-$^<$ [*997*] *997*)
  -svid-post :: *svid* ⇒ *svid* (-$^>$ [*997*] *997*)
  -svid-fst :: *svid* ⇒ *svid* (-.*1* [*997*] *997*)
  -svid-snd :: *svid* ⇒ *svid* (-.*2* [*997*] *997*)
  -mk-svid-list :: *svids* ⇒ *logic* — Helper function for summing a list of identifiers
  -of-svid-list :: *logic* ⇒ *svids* — Reverse of the above
  -svid-view :: *logic* ⇒ *svid* ($\mathcal{V}$[-]) — View of a symmetric lens

-svid-coview :: *logic* ⇒ *svid* ($\mathcal{C}$[-]) — Coview of a symmetric lens
-svid-prod :: *svid* ⇒ *svid* ⇒ *svid* (**infixr** × *85*)
-svid-pow2 :: *svid* ⇒ *svid* (-^{2} [*999*] *999*)

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

syntax — Variable sets
-salphaid :: *id-position* ⇒ *salpha* (- [*990*] *990*)
-salphavar :: *svid* ⇒ *salpha* ($- [*990*] *990*)
-salphaparen :: *salpha* ⇒ *salpha* ('(-'))
-salphaunion :: *salpha* ⇒ *salpha* ⇒ *salpha* (**infixr** ∪ *75*)
-salphainter :: *salpha* ⇒ *salpha* ⇒ *salpha* (**infixr** ∩ *75*)
-salphaminus :: *salpha* ⇒ *salpha* ⇒ *salpha* (**infixl** − *65*)
-salphacompl :: *salpha* ⇒ *salpha* (− - [*81*] *80*)
-salpha-all :: *salpha* (Σ)
-salpha-none :: *salpha* (∅)

-salphaset :: *svids* ⇒ *salpha* ({-})
-sframeid :: *id* ⇒ *sframe* (-)
-sframeset :: *svids* ⇒ *sframe-enum* (⦃-⦄)
-sframeunion :: *sframe* ⇒ *sframe* ⇒ *sframe* (**infixr** ∪ *75*)
-sframeinter :: *sframe* ⇒ *sframe* ⇒ *sframe* (**infixr** ∩ *75*)
-sframeminus :: *sframe* ⇒ *sframe* ⇒ *sframe* (**infixl** − *65*)
-sframecompl :: *sframe* ⇒ *sframe* (− - [*81*] *80*)
-sframe-all :: *sframe* (Σ)
-sframe-none :: *sframe* (∅)
-sframe-pre :: *sframe* ⇒ *sframe* (-$^<$ [*989*] *989*)
-sframe-post :: *sframe* ⇒ *sframe* (-$^>$ [*989*] *989*)
-sframe-enum :: *sframe-enum* ⇒ *sframe* (-)
-sframe-alpha :: *sframe-enum* ⇒ *salpha* (-)
-salphamk :: *logic* ⇒ *salpha*
-mk-alpha-list :: *svids* ⇒ *logic*
-mk-frame-list :: *svids* ⇒ *logic*

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
-svid-set :: *svids* ⇒ *logic* ($\{$-$\}_v$)
-svid-empty :: *logic* ($\{\}_v$)
-svar :: *svid* ⇒ *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
$\textit{-svid } x \rightharpoonup x$
$\textit{-svlongid } x \rightharpoonup x$
$\textit{-svid-alpha} \rightleftharpoons \textit{CONST univ-var}$
$\textit{-svid-tuple xs} \rightharpoonup \textit{-mk-svid-list xs}$
$\textit{-svid-dot } x\ y \rightleftharpoons \textit{CONST ns-alpha } x\ y$
$\textit{-svid-index } x\ i \rightharpoonup x\ i$
$\textit{-svid-res } x\ y \rightleftharpoons x /_L\ y$
$\textit{-svid-pre } x \rightleftharpoons \textit{-svid-dot } \textit{fst}_L\ x$
$\textit{-svid-post } x \rightleftharpoons \textit{-svid-dot } \textit{snd}_L\ x$
$\textit{-svid-fst } x \rightleftharpoons \textit{CONST var-fst } x$
$\textit{-svid-snd } x \rightleftharpoons \textit{CONST var-snd } x$
$\textit{-svid-prod } x\ y \rightleftharpoons x \times_L y$
$\textit{-svid-pow2 } x \rightharpoonup x \times_L x$
$\textit{-mk-svid-list } (\textit{-svid-list } x\ \textit{xs}) \rightharpoonup \textit{CONST var-pair } x\ (\textit{-mk-svid-list xs})$
$\textit{-mk-svid-list } x \rightharpoonup x$
$\textit{-mk-alpha-list } (\textit{-svid-list } x\ \textit{xs}) \rightharpoonup \textit{CONST var-alpha } x \sqcup_S \textit{-mk-alpha-list xs}$
$\textit{-mk-alpha-list } x \rightharpoonup \textit{CONST var-alpha } x$

$\textit{-svid-view } a \Rightarrow \mathcal{V}_a$
$\textit{-svid-coview } a \Rightarrow \mathcal{C}_a$

$\textit{-svid-list } (\textit{-svid-tuple } (\textit{-of-svid-list } (\textit{CONST var-pair } x\ y)))\ (\textit{-of-svid-list } z) \leftharpoondown \textit{-of-svid-list } (\textit{CONST var-pair } (\textit{CONST var-pair } x\ y)\ z)$
$\textit{-svid-list } x\ (\textit{-of-svid-list } y) \leftharpoondown \textit{-of-svid-list } (\textit{CONST var-pair } x\ y)$
$x \leftharpoondown \textit{-of-svid-list } x$

$\textit{-svid-tuple } (\textit{-svid-list } x\ y) \leftharpoondown \textit{CONST var-pair } x\ y$
$\textit{-svid-list } x\ \textit{ys} \leftharpoondown \textit{-svid-list } x\ (\textit{-svid-tuple ys})$

— Alphabets
$\textit{-salphaparen } a \rightharpoonup a$
$\textit{-salphaid } x \rightharpoonup x$
$\textit{-salphaunion } x\ y \rightharpoonup x \sqcup_S y$
$\textit{-salphainter } x\ y \rightharpoonup x \sqcap_S y$
$\textit{-salphaminus } x\ y \rightharpoonup x - y$
$\textit{-salphacompl } x \rightharpoonup - x$

$\textit{-salphavar } x \rightleftharpoons \textit{CONST var-alpha } x$
$\textit{-salphaset } A \rightharpoonup \textit{-mk-alpha-list } A$
$\textit{-sframeid } A \rightharpoonup A$
$(\textit{-svid-list } x\ (\textit{-salphamk } y)) \leftharpoondown \textit{-salphamk } (x +_L y)$

$x \leftharpoondown$ *-salphamk x*
-salpha-all $\rightleftharpoons$ *CONST univ-alpha*
-salpha-none $\rightleftharpoons$ *CONST emp-alpha*

— Quotations
-svid-set A $\rightharpoonup$ *-mk-alpha-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*
›

2.3 Simplifications

lemma *get-pre* [*simp*]: $get_{(x^<)_v}\ (s_1, s_2) = get_x\ s_1$
by (*simp add*: *lens-defs*)

lemma *get-post* [*simp*]: $get_{(x^>)_v}\ (s_1, s_2) = get_x\ s_2$
by (*simp add*: *lens-defs*)

lemma *get-prod-decomp*: $get_x\ s = (get_{var\text{-}fst\ x}\ s,\ get_{var\text{-}snd\ x}\ s)$
by (*simp add*: *lens-defs*)

end

3 Expressions

theory *Expressions*
imports *Variables*
keywords *expr-constructor expr-function* :: *thy-decl-block*
begin

3.1 Types and Constructs

named-theorems *expr-defs* **and** *named-expr-defs*

An expression is represented simply as a function from the state space $'s$ to the return type $'a$, which is the simplest shallow model for Isabelle/HOL.

The aim of this theory is to provide transparent conversion between this representation and a more intuitive expression syntax. For example, an expression $x + y$ where x and y are both state variables, can be represented by $\lambda s.\ get_x\ s + get_y\ s$ when both variables are modelled using lenses. Rather than having to write λ-terms directly, it is more convenient to hide this threading of state behind a parser. We introduce the expression bracket syntax, $(\text{-})_e$ to support this.

type-synonym $('a, 's)\ expr = 's \Rightarrow 'a$

The following constructor is used to syntactically mark functions that actually denote expressions. It is semantically vacuous.

definition $SEXP :: ('s \Rightarrow 'a) \Rightarrow ('a, 's)\ expr\ ([\text{-}]_e)$ **where**
$[expr\text{-}defs]$: $SEXP\ x = x$

lemma *SEXP-apply* [*simp*]: $SEXP\ e\ s = (e\ s)$ **by** (*simp add*: *SEXP-def*)

lemma *SEXP-idem* [*simp*]: $[[e]_e]_e = [e]_e$ **by** (*simp add*: *SEXP-def*)

We can create the core constructs of a simple expression language as indicated below.

abbreviation (*input*) $var :: ('a \Longrightarrow 's) \Rightarrow ('a, 's)\ expr$ **where**
$var\ x \equiv (\lambda\ s.\ get_x\ s)$

abbreviation (*input*) $lit :: 'a \Rightarrow ('a, 's)\ expr$ **where**
$lit\ k \equiv (\lambda\ s.\ k)$

abbreviation (*input*) $uop :: ('a \Rightarrow 'b) \Rightarrow ('a, 's)\ expr \Rightarrow ('b, 's)\ expr$ **where**
$uop\ f\ e \equiv (\lambda\ s.\ f\ (e\ s))$

abbreviation (*input*) *bop*
$:: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 's)\ expr \Rightarrow ('b, 's)\ expr \Rightarrow ('c, 's)\ expr$ **where**
$bop\ f\ e_1\ e_2 \equiv (\lambda\ s.\ f\ (e_1\ s)\ (e_2\ s))$

definition $taut :: (bool, 's)\ expr \Rightarrow bool$ **where**
$[expr\text{-}defs]$: $taut\ e = (\forall\ s.\ e\ s)$

definition $expr\text{-}select :: ('a, 's)\ expr \Rightarrow ('b \Longrightarrow 'a) \Rightarrow ('b, 's)\ expr$ **where**
$[expr\text{-}defs,\ code\text{-}unfold]$: $expr\text{-}select\ e\ x = (\lambda\ s.\ get_x\ (e\ s))$

definition $expr\text{-}if :: ('a, 's)\ expr \Rightarrow (bool, 's)\ expr \Rightarrow ('a, 's)\ expr \Rightarrow ('a, 's)\ expr$ **where**
$[expr\text{-}defs,\ code\text{-}unfold]$: $expr\text{-}if\ P\ b\ Q = (\lambda\ s.\ if\ (b\ s)\ then\ P\ s\ else\ Q\ s)$

3.2 Lifting Parser and Printer

The lifting parser creates a parser directive that converts an expression to a *SEXP* boxed λ-term that gives it a semantics. A pretty printer converts a boxed λ-term back to an expression.

nonterminal *sexp*

We create some syntactic constants and define parse and print translations for them.

syntax

```
-sexp-state       :: id
-sexp-quote       :: logic ⇒ logic ('(-')_e)
— Convert the expression to a lambda term, but do not box it.
-sexp-quote-1way :: logic ⇒ logic ('(-')^e)
-sexp-lit         :: logic ⇒ logic («-»)
-sexp-var         :: svid ⇒ logic ($- [990] 990)
-sexp-evar        :: id-position ⇒ logic (@- [999] 999)
-sexp-evar        :: logic ⇒ logic (@'(-') [999] 999)
-sexp-pqt         :: logic ⇒ sexp ([-]_e)
-sexp-taut        :: logic ⇒ logic (‘-‘)
-sexp-select      :: logic ⇒ svid ⇒ logic (-:- [1000, 999] 1000)
-sexp-if          :: logic ⇒ logic ⇒ logic ⇒ logic ((3- ◁ - ▷/ -) [52,0,53] 52)
```

ML-file ‹*Lift-Expr.ML*›

We create a number of attributes for configuring the way the parser works.

declare [[*pretty-print-exprs=true*]]

We can toggle pretty printing of λ expressions using *pretty-print-exprs*.

declare [[*literal-variables=false*]]

Expressions, of course, can contain variables. However, a variable can denote one of three things: (1) a state variable (i.e. a lens); (2) a placeholder for a value (i.e. a HOL literal); and (3) a placeholder for another expression. The attribute *literal-variables* selects option (2) as the default behaviour when true, and option (3) when false.

expr-constructor *expr-select*
expr-constructor *expr-if*

Some constants should not be lifted, since they are effectively constructors for expressions. The command **expr-constructor** allows us to specify such constants to not be lifted. This being the case, the arguments are left unlifted, unless included in a list of numbers before the constant name. The state is passed as the final argument to expression constructors.

parse-translation ‹

```
[(@{syntax-const -sexp-state}, fn ctx => fn term => Syntax.free Lift-Expr.state-id),
```

```
  (@{syntax-const -sexp-quote}
  , fn ctx => fn terms =>
     case terms of
       [Const (@{const-syntax SEXP}, t) $ e] => Const (@{const-name SEXP},
t) $ e |
       [e] =>
          Syntax.const @{const-name SEXP} $ Lift-Expr.mk-lift-expr ctx dummyT
e),
  (@{syntax-const -sexp-quote-1way}
  , fn ctx => fn terms =>
     case terms of
       [e] => Lift-Expr.mk-lift-expr ctx dummyT e)]
›
```

print-translation ‹

```
 [(@{const-syntax SEXP}
  , fn ctx => fn ts =>
    if not (Config.get ctx Lift-Expr.pretty-print-exprs)
    then Term.list-comb (Syntax.const @{syntax-const -sexp-pqt}, ts)
    else
    Syntax.const @{syntax-const -sexp-quote}
      $ Lift-Expr.print-expr ctx (betapply ((hd ts), Syntax.const @{syntax-const
-sexp-state})))]
›
```

translations

```
 -sexp-var x => get_x -sexp-state
 -sexp-taut p == CONST taut (p)_e
 -sexp-select e x == CONST expr-select (e)_e x
 -sexp-if P b Q == CONST expr-if P (b)_e Q
 -sexp-var (-svid-tuple (-of-svid-list (x +_L y))) <= -sexp-var (x +_L y)
```

The main directive is the e subscripted brackets, $(e)_e$. This converts the expression e to a boxed λ term. Essentially, the behaviour is as follows:

1. a new λ abstraction over the state variable s is wrapped around e;
2. every occurrence of a free lens get_x s in e is replaced by $get_x\ s$;
3. every occurrence of an expression variable e is replaced by $e\ s$;
4. every occurrence of any other free variable is left unchanged.

The pretty printer does this in reverse.

Below is a grammar category for lifted expressions.

nonterminal *sexpr*

syntax *-sexpr* :: *logic* ⇒ *sexpr* (-)

parse-translation ‹
  [(@{*syntax-const -sexpr*}, *fn ctx* => *fn* [*e*] =>
    Syntax.const @{*const-name SEXP*}
          $ (*lambda* (*Syntax.free Lift-Expr.state-id*)
                  (*Lift-Expr.lift-expr ctx dummyT* (*Term-Position.strip-positions e*))))]
›

3.3 Reasoning

lemma *expr-eq-iff*: $P = Q \longleftrightarrow$ ‘$P = Q$‘
  by (*simp add*: *taut-def fun-eq-iff*)

lemma *refine-iff-implies*: $P \leq Q \longleftrightarrow$ ‘$P \longrightarrow Q$‘
  by (*simp add*: *le-fun-def taut-def*)

lemma *taut-True* [*simp*]: ‘*True*‘ = *True*
  by (*simp add*: *taut-def*)

lemma *taut-False* [*simp*]: ‘*False*‘ = *False*
  by (*simp add*: *taut-def*)

lemma *tautI*: $[\![\bigwedge s.\ P\ s]\!] \Longrightarrow$ *taut P*
  by (*simp add*: *taut-def*)

named-theorems *expr-simps*

Lemmas to help automation of expression reasoning

lemma *fst-case-sum* [*simp*]: *fst* (*case p of Inl x* ⇒ (*a1 x*, *a2 x*) | *Inr x* ⇒ (*b1 x*, *b2 x*)) = (*case p of Inl x* ⇒ *a1 x* | *Inr x* ⇒ *b1 x*)
  by (*simp add*: *sum.case-eq-if*)

lemma *snd-case-sum* [*simp*]: *snd* (*case p of Inl x* ⇒ (*a1 x*, *a2 x*) | *Inr x* ⇒ (*b1 x*, *b2 x*)) = (*case p of Inl x* ⇒ *a2 x* | *Inr x* ⇒ *b2 x*)
  by (*simp add*: *sum.case-eq-if*)

lemma *sum-case-apply* [*simp*]: (*case p of Inl x* ⇒ *f x* | *Inr x* ⇒ *g x*) *y* = (*case p of Inl x* ⇒ *f x y* | *Inr x* ⇒ *g x y*)
  by (*simp add*: *sum.case-eq-if*)

Proof methods for simplifying shallow expressions to HOL terms. The first retains the lens structure, and the second removes it when alphabet lenses are present.

method *expr-lens-simp* **uses** *add* =
  ((*simp add*: *expr-simps*)? — Perform any possible simplifications retaining the lens structure
   ;((*simp add*: *fun-eq-iff prod.case-eq-if expr-defs named-expr-defs lens-defs add*)? ; — Explode the rest

(*simp add*: *expr-defs named-expr-defs lens-defs add*)*?*))

method *expr-simp* **uses** *add* = (*expr-lens-simp add*: *alpha-splits add*)

Methods for dealing with tautologies

method *expr-lens-taut* **uses** *add* =
(*rule tautI*;
expr-lens-simp add: *add*)

method *expr-taut* **uses** *add* =
(*rule tautI*;
expr-simp add: *add*;
rename-alpha-vars?)

A method for simplifying shallow expressions to HOL terms and applying *auto*

method *expr-auto* **uses** *add* =
(*expr-simp add*: *add*;
(*auto simp add*: *alpha-splits lens-defs add*)*?*;
(*rename-alpha-vars*)*?* — Rename any logical variables with v subscripts
)

3.4 Algebraic laws

lemma *expr-if-idem* [*simp*]: $P \triangleleft b \triangleright P = P$
by *expr-auto*

lemma *expr-if-sym*: $P \triangleleft b \triangleright Q = Q \triangleleft \neg b \triangleright P$
by *expr-auto*

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

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

lemma *expr-if-true* [*simp*]: $P \triangleleft \mathit{True} \triangleright Q = P$
by *expr-auto*

lemma *expr-if-false* [*simp*]: $P \triangleleft \mathit{False} \triangleright Q = Q$
by *expr-auto*

lemma *expr-if-reach* [*simp*]: $P \triangleleft b \triangleright (Q \triangleleft b \triangleright R) = P \triangleleft b \triangleright R$
by *expr-auto*

lemma *expr-if-disj* [*simp*]: $P \triangleleft b \triangleright (P \triangleleft c \triangleright Q) = P \triangleleft b \vee c \triangleright Q$
by *expr-auto*

lemma *SEXP-expr-if*: $[\mathit{expr\text{-}if}\ P\ b\ Q]_e = \mathit{expr\text{-}if}\ P\ b\ Q$

by (*simp add*: *SEXP-def*)

end

4 Unrestriction

theory *Unrestriction*
imports *Expressions*
begin

Unrestriction means that an expression does not depend on the value of the state space described by the given scene (i.e. set of variables) for its valuation. It is a semantic characterisation of fresh variables.

consts *unrest* :: *'s scene* ⇒ *'p* ⇒ *bool*

definition *unrest-expr* :: *'s scene* ⇒ (*'b*, *'s*) *expr* ⇒ *bool* **where**
[*expr-defs*]: *unrest-expr a e* = (∀ *s s'*. *e* (*s* $\oplus_S$ *s' on a*) = *e s*)

adhoc-overloading *unrest* ⇌ *unrest-expr*

syntax
-unrest :: *salpha* ⇒ *logic* ⇒ *logic* ⇒ *logic* (**infix** ♯ *20*)

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

named-theorems *unrest*

lemma *unrest-empty* [*unrest*]: ∅ ♯ *P*
by (*simp add*: *expr-defs lens-defs*)

lemma *unrest-var-union* [*unrest*]:
⟦ *A* ♯ *P*; *B* ♯ *P* ⟧ ⟹ *A* ∪ *B* ♯ *P*
by (*simp add*: *expr-defs lens-defs*)
(*metis scene-override-union scene-override-unit scene-union-incompat*)

lemma *unrest-neg-union*:
assumes *A* $\#\#_S$ *B* − *A* ♯ *P* − *B* ♯ *P*
shows (− (*A* ∪ *B*)) ♯ *P*
using *assms* **by** (*simp add*: *unrest-expr-def scene-override-commute scene-override-union*)

The following two laws greatly simplify proof when reasoning about unrestricted lens, and so we add them to the expression simplification set.

lemma *unrest-lens* [*expr-simps*]:
mwb-lens x ⟹ ($*x* ♯ *e*) = (∀ *s v*. *e* (put_x *s v*) = *e s*)
by (*simp add*: *unrest-expr-def var-alpha-def comp-mwb-lens lens-override-def*)
(*metis mwb-lens.put-put*)

lemma *unrest-compl-lens* [*expr-simps*]:
mwb-lens x $\Longrightarrow$ ($-$ \$*x* $\sharp$ *e*) = ($\forall$ *s s'*. *e* (*put*$_x$ *s'* (*get*$_x$ *s*)) = *e s*)
by (*simp add*: *unrest-expr-def var-alpha-def comp-mwb-lens lens-override-def scene-override-commute*)

lemma *unrest-subscene*: $[\![$ *idem-scene a*; *a* $\sharp$ *e*; *b* $\subseteq_S$ *a* $]\!]$ $\Longrightarrow$ *b* $\sharp$ *e*
by (*metis subscene-eliminate unrest-expr-def*)

lemma *unrest-lens-comp* [*unrest*]: $[\![$ *mwb-lens x*; *mwb-lens y*; \$*x* $\sharp$ *e* $]\!]$ $\Longrightarrow$ \$*x*:*y* $\sharp$ *e*
by (*simp add*: *unrest-lens*, *simp add*: *lens-comp-def ns-alpha-def*)

lemma *unrest-expr* [*unrest*]: *x* $\sharp$ *e* $\Longrightarrow$ *x* $\sharp$ $(e)_e$
by (*simp add*: *expr-defs*)

lemma *unrest-lit* [*unrest*]: *x* $\sharp$ («*v*»)$_e$
by (*simp add*: *expr-defs*)

lemma *unrest-var* [*unrest*]:
$[\![$ *vwb-lens x*; *idem-scene a*; *var-alpha x* $\bowtie_S$ *a* $]\!]$ $\Longrightarrow$ *a* $\sharp$ (\$*x*)$_e$
by (*auto simp add*: *unrest-expr-def scene-indep-override var-alpha-def*)
(*metis lens-override-def lens-override-idem mwb-lens-weak vwb-lens-mwb weak-lens-def*)

lemma *unrest-var-single* [*unrest*]:
$[\![$ *mwb-lens x*; *x* $\bowtie$ *y* $]\!]$ $\Longrightarrow$ \$*x* $\sharp$ (\$*y*)$_e$
by (*simp add*: *expr-defs lens-indep.lens-put-irr2 lens-indep-sym lens-override-def var-alpha-def*)

lemma *unrest-sublens*:
assumes *mwb-lens x* \$*x* $\sharp$ *P* *y* $\subseteq_L$ *x*
shows \$*y* $\sharp$ *P*
by (*metis assms sublens-pres-mwb sublens-put-put unrest-lens*)

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

lemma *unrest-equiv*:
assumes *mwb-lens y* *x* $\approx_L$ *y* \$*x* $\sharp$ *P*
shows \$*y* $\sharp$ *P*
using *assms lens-equiv-def sublens-pres-mwb unrest-sublens* **by** *blast*

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*:
assumes *bij-lens x* \$*x* $\sharp$ *P*
shows Σ $\sharp$ *P*
by (*metis assms bij-lens-vwb lens-scene-top-iff-bij-lens univ-alpha-def var-alpha-def vwb-lens-iff-mwb-UNIV-src*)

lemma *bij-lens-unrest-all-eq*:
assumes *bij-lens x*
shows (Σ $\sharp$ *P*) $\longleftrightarrow$ (\$*x* $\sharp$ *P*)

by (*metis assms bij-lens-vwb lens-scene-top-iff-bij-lens univ-alpha-def var-alpha-def vwb-lens-mwb*)

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

lemma *unrest-all-var*:
assumes $\Sigma \sharp e$
shows $\$x \sharp e$
by (*metis assms scene-top-greatest top-idem-scene univ-alpha-def unrest-subscene*)

lemma *unrest-pair* [*unrest*]:
assumes *mwb-lens* x *mwb-lens* y $\$x \sharp P$ $\$y \sharp P$
shows $\$(x, y) \sharp P$
using *assms*
by *expr-simp* (*simp add*: *lens-override-def lens-scene.rep-eq scene-override.rep-eq*)

lemma *unrest-pair-split*:
assumes $x \bowtie y$ *vwb-lens* x *vwb-lens* y
shows $(\$(x, y) \sharp P) = ((\$x \sharp P) \wedge (\$y \sharp P))$
using *assms*
by (*metis lens-equiv-scene lens-indep-sym lens-plus-comm lens-plus-right-sublens plus-vwb-lens sublens-refl unrest-pair unrest-sublens var-alpha-def var-pair-def vwb-lens-def*)

lemma *unrest-get* [*unrest*]: $[\![$ *mwb-lens* x; $x \bowtie y$ $]\!] \Longrightarrow \$x \sharp get_y$
by (*expr-simp*, *simp add*: *lens-indep.lens-put-irr2*)

lemma *unrest-conj* [*unrest*]:
$[\![x \sharp P; x \sharp Q]\!] \Longrightarrow x \sharp (P \wedge Q)_e$
by (*auto simp add*: *expr-defs*)

lemma *unrest-not* [*unrest*]:
$[\![x \sharp P]\!] \Longrightarrow x \sharp (\neg P)_e$
by (*auto simp add*: *expr-defs*)

lemma *unrest-disj* [*unrest*]:
$[\![x \sharp P; x \sharp Q]\!] \Longrightarrow x \sharp (P \vee Q)_e$
by (*auto simp add*: *expr-defs*)

lemma *unrest-implies* [*unrest*]:
$[\![x \sharp P; x \sharp Q]\!] \Longrightarrow x \sharp (P \longrightarrow Q)_e$
by (*auto simp add*: *expr-defs*)

lemma *unrest-expr-if* [*unrest*]:
assumes $a \sharp P$ $a \sharp Q$ $a \sharp (e)_e$
shows $a \sharp (P \triangleleft e \triangleright Q)$
using *assms* **by** *expr-simp*

lemma *unrest-uop*:
$[\![x \sharp e]\!] \Longrightarrow x \sharp («f» \ e)_e$

by (*auto simp add*: *expr-defs*)

lemma *unrest-bop*:
$[\![x \sharp e_1; x \sharp e_2]\!] \Longrightarrow x \sharp («f» e_1 e_2)_e$
by (*auto simp add*: *expr-defs*)

lemma *unrest-trop*:
$[\![x \sharp e_1; x \sharp e_2; x \sharp e_3]\!] \Longrightarrow x \sharp («f» e_1 e_2 e_3)_e$
by (*auto simp add*: *expr-defs*)

end

5 Substitutions

theory *Substitutions*
imports *Unrestriction*
begin

5.1 Types and Constants

A substitution is simply a function between two state spaces. Typically, they are used to express mappings from variables to values (e.g. assignments).

type-synonym $('s_1, 's_2)$ *psubst* $= 's_1 \Rightarrow 's_2$
type-synonym $'s$ *subst* $= 's \Rightarrow 's$

There are different ways of constructing an empty substitution.

definition *subst-id* :: $'s$ *subst* ([↝])
where [*expr-defs*, *code-unfold*]: *subst-id* = ($\lambda s.\ s$)

definition *subst-nil* :: $('s_1, 's_2)$ *psubst* (⦇↝⦈)
where [*expr-defs*, *code-unfold*]: ⦇↝⦈ = (λ *s. undefined*)

definition *subst-default* :: $('s_1, 's_2::default)$ *psubst* (⟨↝⟩)
where [*expr-defs*, *code-unfold*]: ⟨↝⟩ = (λ *s. default*)

We can update a substitution by adding a new variable maplet.

definition *subst-upd* :: $('s_1, 's_2)$ *psubst* $\Rightarrow ('a \Longrightarrow 's_2) \Rightarrow ('a, 's_1)$ *expr* $\Rightarrow ('s_1, 's_2)$ *psubst*
where [*expr-defs*, *code-unfold*]: *subst-upd* $\sigma\ x\ e = (\lambda\ s.\ put_x\ (\sigma\ s)\ (e\ s))$

The next two operators extend and restrict the alphabet of a substitution.

definition *subst-ext* :: $('s_1 \Longrightarrow 's_2) \Rightarrow ('s_2, 's_1)$ *psubst* $(\text{-}^{\uparrow}$ [*999*] *999*) **where** [*expr-defs*, *code-unfold*]: *subst-ext* $a = get_a$

definition *subst-res* :: $('s_1 \Longrightarrow 's_2) \Rightarrow ('s_1, 's_2)$ *psubst* $(\text{-}_{\downarrow}$ [*999*] *999*) **where** [*expr-defs*, *code-unfold*]: *subst-res* $a = create_a$

Application of a substitution to an expression is effectively function composition.

definition *subst-app* :: $('s_1,\ 's_2)$ *psubst* ⇒ $('a,\ 's_2)$ *expr* ⇒ $('a,\ 's_1)$ *expr*
where [*expr-defs*]: *subst-app* σ e = (λ s. e (σ s))

abbreviation *aext* P a ≡ *subst-app* $(a^{\uparrow})$ P
abbreviation *ares* P a ≡ *subst-app* $(a_{\downarrow})$ P

We can also lookup the expression a variable is mapped to.

definition *subst-lookup* :: $('s_1,\ 's_2)$ *psubst* ⇒ $('a \Longrightarrow 's_2)$ ⇒ $('a,\ 's_1)$ *expr* $(\langle\text{-}\rangle_s)$
where [*expr-defs*, *code-unfold*]: $\langle\sigma\rangle_s$ x = (λ s. get_x (σ s))

definition *subst-comp* :: $('s_1,\ 's_2)$ *psubst* ⇒ $('s_3,\ 's_1)$ *psubst* ⇒ $('s_3,\ 's_2)$ *psubst* (**infixl** $\circ_s$ *55*)
where [*expr-defs*, *code-unfold*]: *subst-comp* = *comp*

definition *unrest-usubst* :: *'s scene* ⇒ *'s subst* ⇒ *bool*
where [*expr-defs*]: *unrest-usubst* a σ = ($\forall$ s s'. σ ($s \oplus_S s'$ *on* a) = (σ s) $\oplus_S$ s' *on* a)

definition *par-subst* :: *'s subst* ⇒ *'s scene* ⇒ *'s scene* ⇒ *'s subst* ⇒ *'s subst*
where [*expr-defs*]: *par-subst* σ_1 A B σ_2 = (λ s. ($s \oplus_S$ (σ_1 s) *on* A) $\oplus_S$ (σ_2 s) *on* B)

definition *subst-restrict* :: *'s subst* ⇒ *'s scene* ⇒ *'s subst* **where**
[*expr-defs*]: *subst-restrict* σ a = (λ s. $s \oplus_S \sigma$ s *on* a)

Create a substitution that copies the region from the given scene from a given state. This is used primarily in calculating unrestriction conditions.

definition *sset* :: *'s scene* ⇒ *'s* ⇒ *'s subst*
where [*expr-defs*, *code-unfold*]: *sset* a s' = (λ s. $s \oplus_S s'$ *on* a)

syntax *-sset* :: *salpha* ⇒ *logic* ⇒ *logic* (*sset*[-, -])
translations *-sset* a P == *CONST sset* a P

5.2 Syntax Translations

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

syntax
-subst-app :: *logic* ⇒ *logic* ⇒ *logic* (**infix** † *65*)
-smaplet :: [*svid*, *logic*] => *smaplet* (- ⇝ -)
:: *smaplet* => *smaplets* (-)
-SMaplets :: [*smaplet*, *smaplets*] => *smaplets* (-,/ -)
— A little syntax utility to extract a list of variable identifiers from a substitution
-smaplets-svids :: *smaplets* ⇒ *logic*
-SubstUpd :: [*logic*, *smaplets*] => *logic* (-/'(-') [*900*,*0*] *900*)
-Subst :: *smaplets* => *logic* ((*1*[-]))

-PSubst :: *smaplets* => *logic* ((*1*⦇-⦈))

-DSubst :: *smaplets* ⇒ *logic* ((*1*⦉-⦊))

-psubst :: [*logic*, *svids*, *uexprs*] ⇒ *logic*

-subst :: *logic* ⇒ *uexprs* ⇒ *svids* ⇒ *logic* ((-⟦-'/-⟧) [*990*,*0*,*0*] *991*)

-uexprs :: [*logic*, *uexprs*] => *uexprs* (-,/ -)

:: *logic* => *uexprs* (-)

-par-subst :: *logic* ⇒ *salpha* ⇒ *salpha* ⇒ *logic* ⇒ *logic* (- [-|-]$_s$ - [*100*,*0*,*0*,*101*] *101*)

-subst-restrict :: *logic* ⇒ *salpha* ⇒ *logic* (**infixl** ↾$_s$ *85*)

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

translations

-subst-app σ e == *CONST subst-app σ e*

-subst-app σ e <= *-subst-app σ* (*e*)$_e$

-SubstUpd m (*-SMaplets xy ms*) == *-SubstUpd* (*-SubstUpd m xy*) *ms*

-SubstUpd m (*-smaplet x y*) == *CONST subst-upd m x* (*y*)$_e$

-Subst ms == *-SubstUpd* [⤳] *ms*

-Subst (*-SMaplets ms1 ms2*) <= *-SubstUpd* (*-Subst ms1*) *ms2*

-PSubst ms == *-SubstUpd* ⦇⤳⦈ *ms*

-PSubst (*-SMaplets ms1 ms2*) <= *-SubstUpd* (*-PSubst ms1*) *ms2*

-DSubst ms == *-SubstUpd* ⦉⤳⦊ *ms*

-DSubst (*-SMaplets ms1 ms2*) <= *-SubstUpd* (*-DSubst ms1*) *ms2*

-SMaplets ms1 (*-SMaplets ms2 ms3*) <= *-SMaplets* (*-SMaplets ms1 ms2*) *ms3*

-smaplets-svids (*-SMaplets* (*-smaplet x e*) *ms*) => *x* $+_L$ (*-smaplets-svids ms*)

-smaplets-svids (*-smaplet x e*) => *x*

-subst P es vs => *CONST subst-app* (*-psubst* [⤳] *vs es*) *P*

-psubst m (*-svid-list x xs*) (*-uexprs v vs*) => *-psubst* (*-psubst m x v*) *xs vs*

-psubst m x v => *CONST subst-upd m x* (*v*)$_e$

-subst P v x <= *CONST subst-app* (*CONST subst-upd* [⤳] *x* (*v*)$_e$) *P*

-subst P v x <= *-subst-app* (*-Subst* (*-smaplet x v*)) *P*

-subst P v x <= *-subst* (*-sexp-quote P*) *v x*

-subst P v (*-svid-tuple* (*-of-svid-list* (*x* $+_L$ *y*))) <= *-subst P v* (*x* $+_L$ *y*)

-par-subst σ_1 *A B* σ_2 == *CONST par-subst* σ_1 *A B* σ_2

-subst-restrict σ a == *CONST subst-restrict σ a*

-unrest-usubst x p == *CONST unrest-usubst x p*

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

expr-constructor *subst-app* (*1*) — Only the second parameter (1) should be treated as a lifted expression.

expr-constructor *subst-id*

expr-constructor *subst-nil*

expr-constructor *subst-default*

expr-constructor *subst-upd*

expr-constructor *subst-lookup*

ML-file ‹*Expr-Util.ML*›

5.3 Substitution Laws

named-theorems *usubst* **and** *usubst-eval*

lemma *subst-id-apply* [*usubst*]: $[\leadsto] \dagger P = P$
by (*expr-auto*)

lemma *subst-unrest* [*usubst*]:
$[\![$ *vwb-lens* x; $\$x \sharp v$ $]\!] \Longrightarrow \sigma(x \leadsto e) \dagger v = \sigma \dagger v$
by *expr-auto*

lemma *subst-lookup-id* [*usubst*]: $\langle[\leadsto]\rangle_s\ x = [var\ x]_e$
by *expr-simp*

lemma *subst-lookup-aext* [*usubst*]: $\langle a^{\uparrow}\rangle_s\ x = [get_{ns\text{-}alpha\ a\ x}]_e$
by *expr-auto*

lemma *subst-id-var*: $[\leadsto] = (\$\mathbf{v})_e$
by *expr-simp*

lemma *subst-upd-id-lam* [*usubst*]: *subst-upd* $(\$\mathbf{v})_e\ x\ v =$ *subst-upd* $[\leadsto]\ x\ v$
by *expr-simp*

lemma *subst-id* [*simp*]: $[\leadsto] \circ_s \sigma = \sigma\ \sigma \circ_s [\leadsto] = \sigma$
by *expr-auto+*

lemma *subst-default-id* [*simp*]: $\langle\!|\leadsto|\!\rangle \circ_s \sigma = \langle\!|\leadsto|\!\rangle$
by (*simp add*: *expr-defs comp-def*)

lemma *subst-lookup-one-lens* [*usubst*]: $\langle\sigma\rangle_s\ 1_L = \sigma$
by *expr-simp*

The following law can break expressions abstraction, so it is not by default a "usubst" law.

lemma *subst-apply-SEXP*: *subst-app* $\sigma\ [e]_e = [$*subst-app* $\sigma\ e]_e$
by *expr-simp*

lemma *subst-apply-twice* [*usubst*]:
$\varrho \dagger (\sigma \dagger e) = (\sigma \circ_s \varrho) \dagger e$
by *expr-simp*

lemma *subst-apply-twice-SEXP* [*usubst*]:
$\varrho \dagger [\sigma \dagger e]_e = (\sigma \circ_s \varrho) \dagger [e]_e$
by *expr-simp*

term $(f\ (\sigma \dagger e))_e$

term $(\forall\ x.\ x + \$y > \$z)_e$

term $(\forall\ k.\ P[\![«k»/x]\!])_e$

lemma *subst-get* [*usubst*]: $\sigma \dagger get_x = \langle\sigma\rangle_s\ x$
  by (*simp add*: *expr-defs*)

lemma *subst-var* [*usubst*]: $\sigma \dagger (\$x)_e = \langle\sigma\rangle_s\ x$
  by (*simp add*: *expr-defs*)

We can't use this as simplification unfortunately as the expression structure is too ambiguous to support automatic rewriting.

lemma *subst-uop*: $\sigma \dagger («f»\ e)_e = («f»\ (\sigma \dagger e))_e$
  by (*simp add*: *expr-defs*)

lemma *subst-bop*: $\sigma \dagger («f»\ e_1\ e_2)_e = («f»\ (\sigma \dagger e_1)\ (\sigma \dagger e_2))_e$
  by (*simp add*: *expr-defs*)

lemma *subst-lit* [*usubst*]: $\sigma \dagger («v»)_e = («v»)_e$
  by (*expr-simp*)

lemmas *subst-basic-ops* [*usubst*] =
  subst-bop[**where** *f*=*conj*]
  subst-bop[**where** *f*=*disj*]
  subst-bop[**where** *f*=*implies*]
  subst-uop[**where** *f*=*Not*]
  subst-bop[**where** *f*=*HOL.eq*]
  subst-bop[**where** *f*=*less*]
  subst-bop[**where** *f*=*less-eq*]
  subst-bop[**where** *f*=*Set.member*]
  subst-bop[**where** *f*=*inf*]
  subst-bop[**where** *f*=*sup*]
  subst-bop[**where** *f*=*Pair*]

A substitution update naturally yields the given expression.

lemma *subst-lookup-upd* [*usubst*]:
  assumes *weak-lens x*
  shows $\langle\sigma(x \rightsquigarrow v)\rangle_s\ x = (v)_e$
  using *assms* **by** (*simp add*: *expr-defs*)

lemma *subst-lookup-upd-diff* [*usubst*]:
  assumes $x \bowtie y$
  shows $\langle\sigma(y \rightsquigarrow v)\rangle_s\ x = \langle\sigma\rangle_s\ x$
  using *assms* **by** (*simp add*: *expr-defs*)

lemma *subst-lookup-pair* [*usubst*]:
  $\langle\sigma\rangle_s\ (x +_L y) = ((\langle\sigma\rangle_s\ x, \langle\sigma\rangle_s\ y))_e$
  by (*expr-simp*)

Substitution update is idempotent.

lemma *usubst-upd-idem* [*usubst*]:
assumes *mwb-lens x*
shows $\sigma(x \rightsquigarrow u,\ x \rightsquigarrow v) = \sigma(x \rightsquigarrow v)$
using *assms* **by** (*simp add*: *expr-defs*)

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

Substitution updates commute when the lenses are independent.

lemma *subst-upd-comm*:
assumes $x \bowtie y$
shows $\sigma(x \rightsquigarrow u,\ y \rightsquigarrow v) = \sigma(y \rightsquigarrow v,\ x \rightsquigarrow u)$
using *assms* **unfolding** *subst-upd-def*
by (*auto simp add*: *subst-upd-def assms comp-def lens-indep-comm*)

lemma *subst-upd-comm2*:
assumes $z \bowtie y$
shows $\sigma(x \rightsquigarrow u,\ y \rightsquigarrow v,\ z \rightsquigarrow s) = \sigma(x \rightsquigarrow u,\ z \rightsquigarrow s,\ y \rightsquigarrow v)$
using *assms* **unfolding** *subst-upd-def*
by (*auto simp add*: *subst-upd-def assms comp-def lens-indep-comm*)

lemma *subst-upd-var-id* [*usubst*]:
vwb-lens x $\Longrightarrow [x \rightsquigarrow \$x] = [\rightsquigarrow]$
by (*simp add*: *subst-upd-def subst-id-def id-lens-def SEXP-def*)

lemma *subst-upd-pair* [*usubst*]:
$\sigma((x,\ y) \rightsquigarrow (e,\ f)) = \sigma(y \rightsquigarrow f,\ x \rightsquigarrow e)$
by (*simp add*: *subst-upd-def lens-defs SEXP-def fun-eq-iff*)

lemma *subst-upd-comp* [*usubst*]:
$\varrho(x \rightsquigarrow v) \circ_s \sigma = (\varrho \circ_s \sigma)(x \rightsquigarrow \sigma \dagger v)$
by (*simp add*: *expr-defs fun-eq-iff*)

lemma *swap-subst-inj*: $[\![$ *vwb-lens x*; *vwb-lens y*; $x \bowtie y$ $]\!] \Longrightarrow$ *inj* $[(x,\ y) \rightsquigarrow (\$y,\ \$x)]$
by (*simp add*: *expr-defs lens-defs inj-on-def*)
(*metis lens-indep.lens-put-irr2 lens-indep-get vwb-lens.source-determination vwb-lens-def wb-lens-weak weak-lens.put-get*)

5.4 Proof rules

In proof, a lens can always be substituted for an arbitrary but fixed value.

lemma *taut-substI*:
assumes *vwb-lens x* $\bigwedge v.$ ‘$P[\![«v»/x]\!]$‘
shows ‘P‘
using *assms* **by** (*expr-simp*, *metis vwb-lens.put-eq*)

lemma *eq-substI*:
assumes *vwb-lens x* $\bigwedge$ *v*. $P⟦«v»/x⟧ = Q⟦«v»/x⟧$
shows $P = Q$
using *assms* **by** (*expr-simp*, *metis vwb-lens.put-eq*)

lemma *bool-eq-substI*:
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-substI*)

lemma *less-eq-substI*:
assumes *vwb-lens x* $\bigwedge$ *v*. $P⟦«v»/x⟧ \leq Q⟦«v»/x⟧$
shows $P \leq Q$
using *assms* **by** (*expr-simp*, *metis le-funE le-funI vwb-lens-def wb-lens.source-stability*)

5.5 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 (@{*const-name subst-upd*}, -) $ (*Const* (@{*const-name subst-upd*}, -) $ *s* $ *x* $ *u*) $ *y* $ *v*
=> *if* (*XML.content-of* (*YXML.parse-body* (*Syntax.string-of-term ctx x*)) > *XML.content-of* (*YXML.parse-body* (*Syntax.string-of-term ctx y*)))
then SOME (*mk-meta-eq* @{*thm subst-upd-comm*})
else NONE |
- => *NONE*)
›

5.6 Substitution Unrestriction Laws

lemma *unrest-subst-lens* [*expr-simps*]: *mwb-lens x* $\Longrightarrow$ ($\$x \sharp_s \sigma$) = ($\forall$ *s v*. σ (put_x *s v*) = put_x (σ *s*) *v*)
by (*simp add*: *unrest-usubst-def*, *metis lens-override-def mwb-lens-weak weak-lens.create-get*)

lemma *unrest-subst-empty* [*unrest*]: $x \sharp_s [\leadsto]$
by (*expr-simp*)

lemma *unrest-subst-upd* [*unrest*]: ⟦ *vwb-lens x*; $x \bowtie y$; $\$x \sharp (e)_e$; $\$x \sharp_s \sigma$ ⟧ $\Longrightarrow$ $\$x \sharp_s \sigma(y \leadsto e)$
by (*expr-auto add*: *lens-indep-comm*)

lemma *unrest-subst-upd-compl* [*unrest*]: ⟦ *vwb-lens x*; $y \subseteq_L x$; $-\$x \sharp (e)_e$; $-\$x \sharp_s \sigma$ ⟧ $\Longrightarrow -\$x \sharp_s \sigma(y \leadsto e)$
by (*expr-auto*, *simp add*: *lens-override-def scene-override-commute*)

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

$[\![\ \$x \ \sharp \ P; \ \$x \ \sharp_s \ \sigma \ ]\!] \Longrightarrow \$x \ \sharp \ (\sigma \dagger P)$
by (*expr-auto*)

lemma *unrest-sset* [*unrest*]:
$x \bowtie y \Longrightarrow \$x \ \sharp_s \ sset[\$y, \ v]$
by (*expr-auto*, *meson lens-indep-impl-scene-indep scene-override-commute-indep*)

5.7 Conditional Substitution Laws

lemma *subst-cond-upd-1* [*usubst*]:
$\sigma(x \rightsquigarrow u) \triangleleft b \triangleright \varrho(x \rightsquigarrow v) = (\sigma \triangleleft b \triangleright \varrho)(x \rightsquigarrow (u \triangleleft b \triangleright v))$
by *expr-auto*

lemma *subst-cond-upd-2* [*usubst*]:
$[\![\ vwb\text{-}lens \ x; \ \$x \ \sharp_s \ \varrho \ ]\!] \Longrightarrow \sigma(x \rightsquigarrow u) \triangleleft b \triangleright \varrho = (\sigma \triangleleft b \triangleright \varrho)(x \rightsquigarrow (u \triangleleft b \triangleright (\$x)_e))$
by (*expr-auto*, *metis lens-override-def lens-override-idem*)

lemma *subst-cond-upd-3* [*usubst*]:
$[\![\ vwb\text{-}lens \ x; \ \$x \ \sharp_s \ \sigma \ ]\!] \Longrightarrow \sigma \triangleleft b \triangleright \varrho(x \rightsquigarrow v) = (\sigma \triangleleft b \triangleright \varrho)(x \rightsquigarrow ((\$x)_e \triangleleft b \triangleright v))$
by (*expr-auto*, *metis lens-override-def lens-override-idem*)

lemma *expr-if-bool-var-left*: $vwb\text{-}lens \ x \Longrightarrow P[\![True/x]\!] \triangleleft \$x \triangleright Q = P \triangleleft \$x \triangleright Q$
by (*expr-simp*, *metis* (*full-types*) *lens-override-def lens-override-idem*)

lemma *expr-if-bool-var-right*: $vwb\text{-}lens \ x \Longrightarrow P \triangleleft \$x \triangleright Q[\![False/x]\!] = P \triangleleft \$x \triangleright Q$
by (*expr-simp*, *metis* (*full-types*) *lens-override-def lens-override-idem*)

lemma *subst-expr-if* [*usubst*]: $\sigma \dagger (P \triangleleft B \triangleright Q) = (\sigma \dagger P) \triangleleft (\sigma \dagger B) \triangleright (\sigma \dagger Q)$
by *expr-simp*

5.8 Substitution Restriction Laws

lemma *subst-restrict-id* [*usubst*]: $idem\text{-}scene \ a \Longrightarrow [\rightsquigarrow] \upharpoonright_s a = [\rightsquigarrow]$
by *expr-simp*

lemma *subst-restrict-out* [*usubst*]: $[\![\ vwb\text{-}lens \ x; \ vwb\text{-}lens \ a; \ x \bowtie a \ ]\!] \Longrightarrow \sigma(x \rightsquigarrow e) \upharpoonright_s \$a = \sigma \upharpoonright_s \$a$
by (*expr-simp add*: *lens-indep.lens-put-irr2*)

lemma *subst-restrict-in* [*usubst*]: $[\![\ vwb\text{-}lens \ x; \ vwb\text{-}lens \ y; \ x \subseteq_L y \ ]\!] \Longrightarrow \sigma(x \rightsquigarrow e) \upharpoonright_s \$y = (\sigma \upharpoonright_s \$y)(x \rightsquigarrow e)$
by (*expr-auto*)

lemma *subst-restrict-twice* [*simp*]: $\sigma \upharpoonright_s a \upharpoonright_s a = \sigma \upharpoonright_s a$
by *expr-simp*

5.9 Evaluation

lemma *subst-SEXP* [*usubst-eval*]: $\sigma \dagger [\lambda \ s. \ e \ s]_e = [\lambda \ s. \ e \ (\sigma \ s)]_e$
by (*simp add*: *SEXP-def subst-app-def fun-eq-iff*)

lemma *get-subst-id* [*usubst-eval*]: get_x ([$\rightsquigarrow$] s) = get_x s
by (*simp add*: *subst-id-def*)

lemma *get-subst-upd-same* [*usubst-eval*]: *weak-lens* $x \Longrightarrow get_x$ (($\sigma(x \rightsquigarrow e)$) s) = e s
by (*simp add*: *subst-upd-def SEXP-def*)

lemma *get-subst-upd-indep* [*usubst-eval*]:
$x \bowtie y \Longrightarrow get_x$ (($\sigma(y \rightsquigarrow e)$) s) = get_x (σ s)
by (*simp add*: *subst-upd-def*)

lemma *unrest-ssubst*: ($a \sharp P$) $\longleftrightarrow$ ($\forall$ s'. *sset* a s' $\dagger$ P = $(P)_e$)
by (*auto simp add*: *expr-defs fun-eq-iff*)

lemma *unrest-ssubst-expr*: ($a \sharp (P)_e$) = ($\forall s'$. *sset*[a, s'] $\dagger$ $(P)_e$ = $(P)_e$)
by (*simp add*: *unrest-ssubst*)

lemma *get-subst-sset-out* [*usubst-eval*]: $\llbracket$ *vwb-lens* x; *var-alpha* $x \bowtie_S a$ $\rrbracket \Longrightarrow get_x$ (*sset* a s' s) = get_x s
by (*simp add*: *expr-defs var-alpha-def get-scene-override-indep*)

lemma *get-subst-sset-in* [*usubst-eval*]: $\llbracket$ *vwb-lens* x; *var-alpha* $x \leq a$ $\rrbracket \Longrightarrow get_x$ (*sset* a s' s) = get_x s'
by (*simp add*: *get-scene-override-le sset-def var-alpha-def*)

lemma *get-subst-ext* [*usubst-eval*]: get_x (*subst-ext* a s) = $get_{ns\text{-}alpha\ a\ x}$ s
by (*expr-simp*)

lemma *unrest-sset-lens* [*unrest*]: $\llbracket$ *mwb-lens* x; *mwb-lens* y; $x \bowtie y$ $\rrbracket \Longrightarrow \$x \sharp_s$ *sset*[$\$y$, s]
by (*simp add*: *sset-def unrest-subst-lens lens-indep-comm lens-override-def*)

lemma *get-subst-restrict-out* [*usubst-eval*]: $\llbracket$ *vwb-lens* a; $x \bowtie a$ $\rrbracket \Longrightarrow get_x$ (($\sigma \upharpoonright_s$ $\$a$) s) = get_x s
by (*expr-simp*)

lemma *get-subst-restrict-in* [*usubst-eval*]: $\llbracket$ *vwb-lens* a; $x \subseteq_L a$ $\rrbracket \Longrightarrow get_x$ (($\sigma \upharpoonright_s$ $\$a$) s) = get_x (σ s)
by (*expr-simp*, *force*)

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

lemma *subst-apply-unrest*:
$\llbracket$ *vwb-lens* x; $\$x \sharp_s \sigma$ $\rrbracket \Longrightarrow \langle\sigma\rangle_s$ x = *var* x
proof −
assume *1*: *vwb-lens* x **and** $\$x \sharp_s \sigma$
hence $\forall s\ s'$. σ ($s \oplus_L s'$ *on* x) = σ $s \oplus_L s'$ *on* x
by (*simp add*: *unrest-usubst-def*)

thus $\langle\sigma\rangle_s$ *x* = *var x*
by (*metis 1 lens-override-def lens-override-idem mwb-lens-weak subst-lookup-def vwb-lens-mwb weak-lens.put-get*)
qed

A tactic for proving unrestrictions by evaluating a special kind of substitution.

method *unrest* **uses** *add* = (*simp add*: *add unrest unrest-ssubst-expr var-alpha-combine usubst usubst-eval*)

A tactic for evaluating substitutions.

method *subst-eval* **uses** *add* = (*simp add*: *add usubst-eval usubst unrest*)

We can exercise finer grained control over substitutions with the following method.

declare *vwb-lens-mwb* [*lens*]
declare *mwb-lens-weak* [*lens*]

method *subst-eval′* = (*simp only*: *lens usubst-eval usubst unrest SEXP-apply*)

end

6 Extension and Restriction

theory *Extension*
imports *Substitutions*
begin

It is often necessary to coerce an expression into a different state space using a lens, for example when the state space grows to add additional variables. Extension and restriction is provided by *aext* and *ares* respectively. Here, we provide syntax translations and reasoning support for these.

6.1 Syntax

syntax
-aext :: *logic* ⇒ *svid* ⇒ *logic* (**infixl** ↑ *80*)
-ares :: *logic* ⇒ *svid* ⇒ *logic* (**infixl** ↓ *80*)
-pre :: *logic* ⇒ *logic* ($-^<$ [*999*] *1000*)
-post :: *logic* ⇒ *logic* ($-^>$ [*999*] *1000*)
-drop-pre :: *logic* ⇒ *logic* ($-_<$ [*999*] *1000*)
-drop-post :: *logic* ⇒ *logic* ($-_>$ [*999*] *1000*)

translations
-aext P a == *CONST aext P a*
-ares P a == *CONST ares P a*
-pre P == *-aext* $(P)_e$ fst_L

-post P == *-aext* $(P)_e$ snd_L
-drop-pre P == *-ares* $(P)_e$ fst_L
-drop-post P == *-ares* $(P)_e$ snd_L

expr-constructor *aext*
expr-constructor *ares*

named-theorems *alpha*

6.2 Laws

lemma *aext-var* [*alpha*]: $(\$x)_e \uparrow a = (\$a{:}x)_e$
by (*simp add*: *expr-defs lens-defs*)

lemma *ares-aext* [*alpha*]: *weak-lens* $a \Longrightarrow P \uparrow a \downarrow a = P$
by (*simp add*: *expr-defs*)

lemma *aext-ares* [*alpha*]: $\llbracket$ *mwb-lens* a; $(- \$a) \sharp P$ $\rrbracket \Longrightarrow P \downarrow a \uparrow a = P$
unfolding *unrest-compl-lens*
by (*auto simp add*: *expr-defs fun-eq-iff lens-create-def*)

lemma *expr-pre* [*simp*]: $e^< (s_1, s_2) = (e)_e\ s_1$
by (*simp add*: *subst-ext-def subst-app-def*)

lemma *expr-post* [*simp*]: $e^> (s_1, s_2) = (@e)_e\ s_2$
by (*simp add*: *subst-ext-def subst-app-def*)

lemma *unrest-aext-expr-lens* [*unrest*]: $\llbracket$ *mwb-lens* x; $x \bowtie a$ $\rrbracket \Longrightarrow \$x \sharp (P \uparrow a)$
by (*expr-simp add*: *lens-indep.lens-put-irr2*)

lemma *unrest-init-pre* [*unrest*]: $\llbracket$ *mwb-lens* x; $\$x \sharp e$ $\rrbracket \Longrightarrow \$x^< \sharp e^<$
by *expr-auto*

lemma *unrest-init-post* [*unrest*]: *mwb-lens* $x \Longrightarrow \$x^< \sharp e^>$
by *expr-auto*

lemma *unrest-fin-pre* [*unrest*]: *mwb-lens* $x \Longrightarrow \$x^> \sharp e^<$
by *expr-auto*

lemma *unrest-fin-post* [*unrest*]: $\llbracket$ *mwb-lens* x; $\$x \sharp e$ $\rrbracket \Longrightarrow \$x^> \sharp e^>$
by *expr-auto*

lemma *aext-get-fst* [*usubst*]: *aext* (get_x) fst_L = $get_{ns\text{-}alpha\ fst_L\ x}$
by *expr-simp*

lemma *aext-get-snd* [*usubst*]: *aext* (get_x) snd_L = $get_{ns\text{-}alpha\ snd_L\ x}$
by *expr-simp*

6.3 Substitutions

definition *subst-aext* :: $'a$ *subst* ⇒ ($'a \Longrightarrow 'b$) ⇒ $'b$ *subst*
  where [*expr-defs*]: *subst-aext* σ x = (λ *s*. put_x *s* (σ (get_x *s*)))

definition *subst-ares* :: $'b$ *subst* ⇒ ($'a \Longrightarrow 'b$) ⇒ $'a$ *subst*
  where [*expr-defs*]: *subst-ares* σ x = (λ *s*. get_x (σ ($create_x$ *s*)))

syntax
  -subst-aext :: *logic* ⇒ *svid* ⇒ *logic* (**infixl** $\uparrow_s$ *80*)
  -subst-ares :: *logic* ⇒ *svid* ⇒ *logic* (**infixl** $\downarrow_s$ *80*)

translations
  -subst-aext P a == *CONST subst-aext P a*
  -subst-ares P a == *CONST subst-ares P a*

lemma *unrest-subst-aext* [*unrest*]: $x \bowtie a \Longrightarrow \$x \sharp_s (\sigma \uparrow_s a)$
  by (*expr-simp*)
    (*metis lens-indep-def lens-override-def lens-scene.rep-eq scene-override.rep-eq*)

lemma *subst-id-ext* [*usubst*]:
  vwb-lens $x \Longrightarrow [\leadsto] \uparrow_s x = [\leadsto]$
  by *expr-auto*

lemma *upd-subst-ext* [*alpha*]:
  vwb-lens $x \Longrightarrow \sigma(y \leadsto e) \uparrow_s x = (\sigma \uparrow_s x)(x{:}y \leadsto e \uparrow x)$
  by *expr-auto*

lemma *apply-subst-ext* [*alpha*]:
  vwb-lens $x \Longrightarrow (\sigma \dagger e) \uparrow x = (\sigma \uparrow_s x) \dagger (e \uparrow x)$
  by (*expr-auto*)

lemma *subst-aext-compose* [*alpha*]: $(\sigma \uparrow_s x) \uparrow_s y = \sigma \uparrow_s y{:}x$
  by (*expr-simp*)

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

lemma *subst-id-res*: *mwb-lens* $a \Longrightarrow [\leadsto] \downarrow_s a = [\leadsto]$
  by *expr-auto*

lemma *upd-subst-res-in*:
  ⟦ *mwb-lens a*; $x \subseteq_L a$ ⟧ $\Longrightarrow \sigma(x \leadsto e) \downarrow_s a = (\sigma \downarrow_s a)(x \restriction a \leadsto e \downarrow a)$
  by (*expr-simp*, *fastforce*)

lemma *upd-subst-res-out*:
  ⟦ *mwb-lens a*; $x \bowtie a$ ⟧ $\Longrightarrow \sigma(x \leadsto e) \downarrow_s a = \sigma \downarrow_s a$
  by (*simp add*: *expr-defs lens-indep-sym*)

lemma *subst-ext-lens-apply*: $\llbracket$ *mwb-lens* a; $-\$a \sharp_s \sigma$ $\rrbracket \Longrightarrow (a^{\uparrow} \circ_s \sigma) \dagger P = ((\sigma \downarrow_s a) \dagger P) \uparrow a$
by (*expr-simp*, *simp add*: *lens-override-def scene-override-commute*)

end

6.4 Liberation

theory *Liberation*
imports *Extension*
begin

Liberation [1] is an operator that removes dependence on a number of variables. It is similar to existential quantification, but is defined over a scene (a variable set).

6.5 Definition and Syntax

definition *liberate* :: $('s \Rightarrow bool) \Rightarrow 's\ scene \Rightarrow ('s \Rightarrow bool)$ **where**
[*expr-defs*]: $liberate\ P\ x = (\lambda\ s.\ \exists\ s'.\ P\ (s \oplus_S s'\ on\ x))$

syntax
-liberate :: $logic \Rightarrow salpha \Rightarrow logic$ (**infixl** $\backslash$ *80*)

translations
-liberate $P\ x$ == *CONST liberate* $P\ x$
-liberate $P\ x$ <= *-liberate* $(P)_e\ x$

expr-constructor *liberate* (*0*)

6.6 Laws

lemma *liberate-lens* [*expr-simps*]:
mwb-lens $x \Longrightarrow P \backslash \$x = (\lambda s.\ \exists s'.\ P\ (s \triangleleft_x s'))$
by (*simp add*: *liberate-def*)

lemma *liberate-lens'*: *mwb-lens* $x \Longrightarrow P \backslash \$x = (\lambda s.\ \exists v.\ P\ (put_x\ s\ v))$
by (*auto simp add*: *liberate-def lens-defs fun-eq-iff*)
(*metis mwb-lens-weak weak-lens.put-get*)

lemma *liberate-as-subst*: *vwb-lens* $x \Longrightarrow e \backslash \$x = (\exists\ v.\ e\llbracket «v»/x \rrbracket)_e$
by (*expr-simp*, *metis vwb-lens.put-eq*)

lemma *unrest-liberate*: $a \sharp P \backslash a$
by (*expr-simp*)

lemma *unrest-liberate-iff*: $(a \sharp P) \longleftrightarrow (P \backslash a = P)$
by (*expr-simp*, *metis* (*full-types*) *scene-override-overshadow-left*)

lemma *liberate-none* [*simp*]: $P \setminus \emptyset = P$
  by (*expr-simp*)

lemma *liberate-idem* [*simp*]: $P \setminus a \setminus a = P \setminus a$
  by (*expr-simp*)

lemma *liberate-commute* [*simp*]: $a \bowtie_S b \Longrightarrow P \setminus a \setminus b = P \setminus b \setminus a$
  using *scene-override-commute-indep* **by** (*expr-auto*, *fastforce+*)

lemma *liberate-true* [*simp*]: $(True)_e \setminus a = (True)_e$
  by (*expr-simp*)

lemma *liberate-false* [*simp*]: $(False)_e \setminus a = (False)_e$
  by (*expr-simp*)

lemma *liberate-disj* [*simp*]: $(P \vee Q)_e \setminus a = (P \setminus a \vee Q \setminus a)_e$
  by (*expr-auto*)

end

6.7 Quantifying Lenses

theory *Quantifiers*
  imports *Liberation*
begin

We define operators to existentially and universally quantify an expression over a lens.

6.8 Operators and Syntax

definition *ex-expr* :: $('a \Longrightarrow 's) \Rightarrow (bool, 's)\ expr \Rightarrow (bool, 's)\ expr$ **where**
[*expr-defs*]: $ex\text{-}expr\ x\ e = (\lambda\ s.\ (\exists\ v.\ e\ (put_x\ s\ v)))$

definition *ex1-expr* :: $('a \Longrightarrow 's) \Rightarrow (bool, 's)\ expr \Rightarrow (bool, 's)\ expr$ **where**
[*expr-defs*]: $ex1\text{-}expr\ x\ e = (\lambda\ s.\ (\exists!\ v.\ e\ (put_x\ s\ v)))$

definition *all-expr* :: $('a \Longrightarrow 's) \Rightarrow (bool, 's)\ expr \Rightarrow (bool, 's)\ expr$ **where**
[*expr-defs*]: $all\text{-}expr\ x\ e = (\lambda\ s.\ (\forall\ v.\ e\ (put_x\ s\ v)))$

expr-constructor *ex-expr* (*1*)
expr-constructor *ex1-expr* (*1*)
expr-constructor *all-expr* (*1*)

syntax
  -ex-expr :: $svid \Rightarrow logic \Rightarrow logic$ ($\exists$ - • - [*0*, *20*] *20*)
  -ex1-expr :: $svid \Rightarrow logic \Rightarrow logic$ ($\exists_1$ - • - [*0*, *20*] *20*)
  -all-expr :: $svid \Rightarrow logic \Rightarrow logic$ ($\forall$ - • - [*0*, *20*] *20*)

translations
  -ex-expr x P == CONST ex-expr x P
  -ex1-expr x P == CONST ex1-expr x P
  -all-expr x P == CONST all-expr x P

6.9 Laws

lemma *ex-is-liberation*: *mwb-lens x* $\Longrightarrow$ ($\exists$ *x* $\bullet$ *P*) = (*P* $\setminus$ \$*x*)
  by (*expr-auto*, *metis mwb-lens-weak weak-lens.put-get*)

lemma *ex-unrest-iff*: $[\![$ *mwb-lens x* $]\!]$ $\Longrightarrow$ (\$*x* $\sharp$ *P*) $\longleftrightarrow$ ($\exists$ *x* $\bullet$ *P*) = *P*
  by (*simp add*: *ex-is-liberation unrest-liberate-iff*)

lemma *ex-unrest*: $[\![$ *mwb-lens x*; \$*x* $\sharp$ *P* $]\!]$ $\Longrightarrow$ ($\exists$ *x* $\bullet$ *P*) = *P*
  using *ex-unrest-iff* **by** *blast*

lemma *unrest-ex-in* [*unrest*]:
  $[\![$ *mwb-lens y*; *x* $\subseteq_L$ *y* $]\!]$ $\Longrightarrow$ \$*x* $\sharp$ ($\exists$ *y* $\bullet$ *P*)
  by (*simp add*: *ex-expr-def sublens-pres-mwb sublens-put-put unrest-lens*)

lemma *unrest-ex-out* [*unrest*]:
  $[\![$ *mwb-lens x*; \$*x* $\sharp$ *P*; *x* $\bowtie$ *y* $]\!]$ $\Longrightarrow$ \$*x* $\sharp$ ($\exists$ *y* $\bullet$ *P*)
  by (*simp add*: *ex-expr-def unrest-lens*, *metis lens-indep.lens-put-comm*)

lemma *subst-ex-out* [*usubst*]: $[\![$ *mwb-lens x*; \$*x* $\sharp_s$ σ $]\!]$ $\Longrightarrow$ σ $\dagger$ ($\exists$ *x* $\bullet$ *P*) = ($\exists$ *x* $\bullet$ σ $\dagger$ *P*)
  by (*expr-simp*)

lemma *subst-lit-ex-indep* [*usubst*]:
  y $\bowtie$ *x* $\Longrightarrow$ σ(*y* $\leadsto$ «*v*») $\dagger$ ($\exists$ *x* $\bullet$ *P*) = σ $\dagger$ ($\exists$ *x* $\bullet$ [*y* $\leadsto$ «*v*»] $\dagger$ *P*)
  by (*expr-simp*, *simp add*: *lens-indep.lens-put-comm*)

lemma *subst-ex-in* [*usubst*]:
  $[\![$ *vwb-lens a*; *x* $\subseteq_L$ *a* $]\!]$ $\Longrightarrow$ σ(*x* $\leadsto$ *e*) $\dagger$ ($\exists$ *a* $\bullet$ *P*) = σ $\dagger$ ($\exists$ *a* $\bullet$ *P*)
  by (*expr-simp*, *force*)

declare *lens-plus-right-sublens* [*simp*]

lemma *ex-as-subst*: *vwb-lens x* $\Longrightarrow$ ($\exists$ *x* $\bullet$ *e*) = ($\exists$ *v*. *e*$[\![$«*v*»/*x*$]\!]$)$_e$
  by *expr-auto*

lemma *ex-twice* [*simp*]: *mwb-lens x* $\Longrightarrow$ ($\exists$ *x* $\bullet$ $\exists$ *x* $\bullet$ *P*) = ($\exists$ *x* $\bullet$ *P*)
  by (*expr-simp*)

lemma *ex-commute*: *x* $\bowtie$ *y* $\Longrightarrow$ ($\exists$ *x* $\bullet$ $\exists$ *y* $\bullet$ *P*) = ($\exists$ *y* $\bullet$ $\exists$ *x* $\bullet$ *P*)
  by (*expr-auto*, *metis lens-indep-comm*)+

lemma *ex-true* [*simp*]: ($\exists$ *x* $\bullet$ (*True*)$_e$) = (*True*)$_e$
  by *expr-simp*

lemma *ex-false* [*simp*]: $(\exists\ x \bullet (False)_e) = (False)_e$
  by (*expr-simp*)

lemma *ex-disj* [*simp*]: $(\exists\ x \bullet (P \vee Q)_e) = ((\exists\ x \bullet P) \vee (\exists\ x \bullet Q))_e$
  by (*expr-auto*)

lemma *ex-plus*:
  $(\exists\ (y,x) \bullet P) = (\exists\ x \bullet \exists\ y \bullet P)$
  by (*expr-auto*)

lemma *all-as-ex*: $(\forall\ x \bullet P) = (\neg\ (\exists\ x \bullet \neg\ P))_e$
  by (*expr-auto*)

lemma *ex-as-all*: $(\exists\ x \bullet P) = (\neg\ (\forall\ x \bullet \neg\ P))_e$
  by (*expr-auto*)

6.10 Cylindric Algebra

lemma *ex-C1*: $(\exists\ x \bullet (False)_e) = (False)_e$
  by (*expr-auto*)

lemma *ex-C2*: *wb-lens* $x \Longrightarrow$ ‘$P \longrightarrow (\exists\ x \bullet P)$‘
  by (*expr-simp*, *metis wb-lens.get-put*)

lemma *ex-C3*: *mwb-lens* $x \Longrightarrow (\exists\ x \bullet (P \wedge (\exists\ x \bullet Q)))_e = ((\exists\ x \bullet P) \wedge (\exists\ x \bullet Q))_e$
  by (*expr-auto*)

lemma *ex-C4a*: $x \approx_L y \Longrightarrow (\exists\ x \bullet \exists\ y \bullet P) = (\exists\ y \bullet \exists\ x \bullet P)$
  by (*expr-simp*, *metis* (*mono-tags*, *lifting*) *lens.select-convs*(*2*))

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

lemma *ex-C5*:
  fixes $x :: ('a \Longrightarrow '\alpha)$
  shows $(\$x = \$x)_e = (True)_e$
  by *simp*

lemma *ex-C6*:
  assumes *wb-lens* x $x \bowtie y$ $x \bowtie z$
  shows $(\$y = \$z)_e = (\exists\ x \bullet \$y = \$x \wedge \$x = \$z)_e$
  using *assms*
  by (*expr-simp*, *metis lens-indep-def*)

lemma *ex-C7*:
  assumes *weak-lens* x $x \bowtie y$
  shows $((\exists\ x \bullet \$x = \$y \wedge P) \wedge (\exists\ x \bullet \$x = \$y \wedge \neg\ P))_e = (False)_e$

using *assms* **by** (*expr-simp*, *simp add*: *lens-indep-sym*)

end

7 Collections

theory *Collections*
imports *Substitutions*
begin

A lens whose source is a collection type (e.g. a list or map) can be divided into several lenses, corresponding to each of the elements in the collection. This can be used to support update of an individual collection element, such as an array update. Here, we provide the infrastructure to support such collection lenses [2].

7.1 Partial Lens Definedness

Partial lenses (e.g. *mwb-lens*) are only defined for certain states. For example, the list lens is defined only when the source list is of a sufficient length. Below, we define a predicate that characterises the states in a which such a lens is defined.

definition *lens-defined* :: $('a \Longrightarrow 's) \Rightarrow (bool, 's)$ *expr* **where**
[*expr-defs*]: *lens-defined* $x = (\$\mathbf{v} \in \mathcal{S}_{«x»})_e$

syntax *-lens-defined* :: *svid* ⇒ *logic* ($\mathbf{D}$'(-'))
translations *-lens-defined x* == *CONST lens-defined x*

expr-constructor *lens-defined*

7.2 Dynamic Lenses

Dynamics lenses [2] are used to model elements of a lens indexed by some type $'i$. The index is typically used to select different elements of a collection. The lens is "dynamic" because the particular index is provided by an expression $'s \Rightarrow 'i$, which can change from state to state. We normally assume that this expression does not refer to the indexed lens itself.

definition *dyn-lens* :: $('i \Rightarrow ('a \Longrightarrow 's)) \Rightarrow ('s \Rightarrow 'i) \Rightarrow ('a \Longrightarrow 's)$ **where**
[*lens-defs*]: *dyn-lens* $f\ x = (\!|\ lens\text{-}get = (\lambda\ s.\ get_{f\ (x\ s)}\ s),\ lens\text{-}put = (\lambda\ s\ v.\ put_{f\ (x\ s)}\ s\ v)\ |\!)$

lemma *dyn-lens-mwb* [*simp*]: $[\![\bigwedge i.\ mwb\text{-}lens\ (f\ i);\ \bigwedge i.\ \$f(i) \sharp\ e\ ]\!] \Longrightarrow mwb\text{-}lens\ (dyn\text{-}lens\ f\ e)$
apply (*unfold-locales*, *auto simp add*: *expr-defs lens-defs lens-indep.lens-put-irr2*)
apply (*metis lens-override-def mwb-lens-weak weak-lens.put-get*)
apply (*metis lens-override-def mwb-lens.put-put*)

done

lemma *ind-lens-vwb* [*simp*]: $\llbracket \bigwedge i.\ vwb\text{-}lens\ (f\ i);\ \bigwedge i.\ \$f(i)\ \sharp\ e\ \rrbracket \Longrightarrow vwb\text{-}lens\ (dyn\text{-}lens\ f\ e)$
by (*unfold-locales*, *auto simp add*: *lens-defs expr-defs lens-indep.lens-put-irr2 lens-scene-override*)
(*metis mwb-lens-weak vwb-lens-mwb weak-lens.put-get*, *metis mwb-lens.put-put vwb-lens-mwb*)

lemma *src-dyn-lens*: $\llbracket \bigwedge i.\ mwb\text{-}lens\ (f\ i);\ \bigwedge i.\ \$f(i)\ \sharp\ e\ \rrbracket \Longrightarrow \mathcal{S}_{dyn\text{-}lens\ f\ e} = \{s.\ s \in \mathcal{S}_{f\ (e\ s)}\}$
by (*auto simp add*: *lens-defs expr-defs lens-source-def lens-scene-override unrest*)
(*metis mwb-lens.put-put*)+

lemma *subst-lookup-dyn-lens* [*usubst*]: $\llbracket \bigwedge i.\ f\ i \bowtie x\ \rrbracket \Longrightarrow \langle subst\text{-}upd\ \sigma\ (dyn\text{-}lens\ f\ k)\ e\rangle_s\ x = \langle\sigma\rangle_s\ x$
by (*expr-simp*, *metis* (*mono-tags*, *lifting*) *lens-indep.lens-put-irr2*)

lemma *get-upd-dyn-lens* [*usubst-eval*]: $\llbracket \bigwedge i.\ f\ i \bowtie x\ \rrbracket \Longrightarrow get_x\ (subst\text{-}upd\ \sigma\ (dyn\text{-}lens\ f\ k)\ e\ s) = get_x\ (\sigma\ s)$
by (*expr-simp*, *metis lens-indep.lens-put-irr2*)

7.3 Overloaded Collection Lens

The following polymorphic constant is used to provide implementations of different collection lenses. Type *'k* is the index into the collection. For the list collection lens, the index has type *nat*.

consts *collection-lens* :: $'k \Rightarrow ('a \Longrightarrow 's)$

definition [*lens-defs*]: *fun-collection-lens* = *fun-lens*
definition [*lens-defs*]: *list-collection-lens* = *list-lens*

adhoc-overloading
collection-lens ⇌ *fun-collection-lens* **and**
collection-lens ⇌ *list-collection-lens*

lemma *vwb-fun-collection-lens* [*simp*]: *vwb-lens* (*fun-collection-lens k*)
by (*simp add*: *fun-collection-lens-def fun-vwb-lens*)

lemma *put-fun-collection-lens* [*simp*]:
$put_{fun\text{-}collection\text{-}lens\ i} = (\lambda f.\ fun\text{-}upd\ f\ i)$
by (*simp add*: *fun-collection-lens-def fun-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*)

lemma *put-list-collection-lens* [*simp*]:
$put_{list\text{-}collection\text{-}lens\ i} = (\lambda\ xs.\ list\text{-}augment\ xs\ i)$
by (*simp add*: *list-collection-lens-def list-lens-def*)

7.4 Syntax for Collection Lenses

We add variable identifier syntax for collection lenses, which allows us to write $x[i]$ for some collection and index.

abbreviation *dyn-lens-poly f x i* ≡ *dyn-lens* (λ *k*. *ns-alpha x* (*f k*)) *i*

syntax
-svid-collection :: *svid* ⇒ *logic* ⇒ *svid* (-[-] [*999*, *0*] *999*)

translations
-svid-collection x e == *CONST dyn-lens-poly CONST collection-lens x* $(e)_e$

lemma *source-ns-alpha*: ⟦ *mwb-lens a*; *mwb-lens x* ⟧ ⟹ $\mathcal{S}_{ns\text{-}alpha\ a\ x} = \{s \in \mathcal{S}_a.\ get_a\ s \in \mathcal{S}_x\}$
by (*simp add*: *ns-alpha-def source-lens-comp*)

lemma *defined-vwb-lens* [*simp*]: *vwb-lens x* ⟹ $\mathbf{D}(x) = (True)_e$
by (*simp add*: *lens-defined-def*)
(*metis UNIV-I vwb-lens-iff-mwb-UNIV-src*)

lemma *defined-list-collection-lens* [*simp*]:
⟦ *vwb-lens x*; $x ♯ *e* ⟧ ⟹ $\mathbf{D}(x[e]) = (e < length(\$x))_e$
by (*simp add*: *lens-defined-def src-dyn-lens unrest source-ns-alpha source-list-collection-lens*)
(*simp add*: *lens-defs wb-lens.source-UNIV*)

lemma *lens-defined-list-code* [*code-unfold*]:
vwb-lens x ⟹ *lens-defined* (*ns-alpha x* (*list-collection-lens i*)) = («*i*» < *length*($*x*))$_e$
by (*simp add*: *lens-defined-def src-dyn-lens unrest source-ns-alpha source-list-collection-lens*)
(*simp add*: *lens-defs wb-lens.source-UNIV*)

The next theorem allows the simplification of a collection lens update.

lemma *get-subst-upd-dyn-lens* [*simp*]:
mwb-lens x ⟹ get_x (*subst-upd* σ (*dyn-lens-poly cl x* $(e)_e$) *v s*)
= *lens-put* (*cl* (*e* (σ *s*))) (get_x (σ *s*)) (*v s*)
by *expr-simp*

end

8 Named Expression Definitions

theory *Named-Expressions*
imports *Expressions*
keywords *edefinition expression* :: *thy-decl-block* **and** *over*

begin

Here, we add a command that allows definition of a named expression. It provides a more concise version of **definition** and inserts the expression brackets.

ML ‹

```
structure Expr-Def =
struct

  fun mk-expr-def-eq ctx term =
    case (Type.strip-constraints term) of
      Const (@{const-name HOL.eq}, b) $ c $ t =>
        (fst (dest-Free (fst (Term.strip-comb c))),
        @{const Trueprop} $ (Const (@{const-name HOL.eq}, b) $ c $ (Syntax.const
@{const-name SEXP}
            $ (lambda (Syntax.free Lift-Expr.state-id)
                    (Lift-Expr.lift-expr ctx dummyT (Term-Position.strip-positions
t)))))) |
      - => raise Match;

  val expr-defs = [[Token.make-string (Binding.name-of @{binding expr-defs}, Po-
sition.none)]];

  fun expr-def attr decl term ctx =
  let val named-expr-defs = @{attributes [named-expr-defs]}
      val (n, eq) = mk-expr-def-eq ctx term
      val (thm, ctx0) = Specification.definition
                (Option.map (fn x => fst (Proof-Context.read-var x ctx)) decl) [] []
                ((fst attr, map (Attrib.check-src ctx) (named-expr-defs @ snd attr)),
eq) (snd (Local-Theory.begin-nested ctx))
      val ctx1 = ExprFun-Const.exprfun-const n (Local-Theory.end-nested ctx0)
  in (thm, ctx1)
  end

  fun named-expr n typ stateT expr ctx =
    let val named-expr-defs = @{attributes [named-expr-defs]}
        val term = Const (@{const-name HOL.eq}, dummyT) $ Syntax.free n $ expr
        val ctx' = snd (Specification.definition
                (SOME (Binding.name n, SOME (stateT --> typ), Mixfix.NoSyn))
[] []
                ((Binding.name (n ^ -def), named-expr-defs), snd (mk-expr-def-eq
ctx term)) (snd (Local-Theory.begin-nested ctx)))
        (* When adding an expression in a locale, the named recorded below may be
the
          localised version, which may not work correctly. This may lead to unexpected
            behaviour when there are two locales each with a constant of the same
name. *)
        val ctx2 = ExprFun-Const.exprfun-const n (Local-Theory.end-nested ctx')
        in ctx2
```

```
  end

  fun named-expr-cmd n t st expr ctx =
    let val term = Syntax.parse-term ctx expr
        val stateT = Syntax.read-typ ctx st
        val typ = Syntax.read-typ ctx t
        in named-expr n typ stateT term ctx
  end

end;

val - =
let
  open Expr-Def;
in
  Outer-Syntax.local-theory command-keyword ‹edefinition› 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 (expr-def attr decl (Syntax.parse-term ctx term) ctx))))
end

val - =
let
  open Expr-Def;
in
  Outer-Syntax.local-theory command-keyword ‹expression› define named expressions
   ((((Parse.name -- Scan.optional (@{keyword ::} |-- Parse.typ) - -- Scan.optional
(@{keyword over} |-- Parse.typ) -) --| @{keyword is}) -- Parse.term) >> (fn
(((n, t), st), expr) =>
        (named-expr-cmd n t st expr)))
end;

›
```

end

9 Local State

theory *Local-State*
  imports *Expressions*
begin

This theory supports ad-hoc extension of an alphabet type with a tuple of lenses constructed using successive applications of fst_L and snd_L. It effectively allows local variables, since we always add a collection of new variables.

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 (localstate (-)/ over (-) [0, 10] 10)
  -lvar-abs :: id ⇒ type ⇒ logic ⇒ logic

translations
  (type) PAIRTYPE('a, 'b) => (type) 'a × 'b
  (type) LENSTYPE('a, 'b) => (type) 'a ⟹ 'b

  -state-type (-constrain x t) => t
 -state-type (CONST Product-Type.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)

ML ‹
signature LIFT-EXPR-LVAR =
sig
val lift-expr-lvar: string −> term −> term
val lift-lvar: string −> term −> term
end

structure Lift-Expr-LVar: LIFT-EXPR-LVAR =
struct

fun lift-expr-lvar n (Free (x, t')) =
  let open Syntax; open Lift-Expr
  in
  if x = n then const @{const-name lens-get} $ Free (n, t') $ Bound 0
  else Free (x, t')
  end |
lift-expr-lvar n (Free (x, t') $ Bound 0) =
  lift-expr-lvar n (Free (x, t')) |
lift-expr-lvar n (Abs (x, t', trm)) =
  if x = n then Abs (x, t', trm) else Abs (x, t', lift-expr-lvar n trm) |
lift-expr-lvar n (t1 $ t2) = lift-expr-lvar n t1 $ lift-expr-lvar n t2 |
lift-expr-lvar - trm = trm
```

```
fun lift-lvar n (Const (@{const-name SEXP}, t') $ e)
  = (Const (@{const-name SEXP}, t') $ lift-expr-lvar n e) |
lift-lvar n (t1 $ t2) = lift-lvar n t1 $ lift-lvar n t2 |
lift-lvar n (Abs (x, t', trm)) =
  if x = n then Abs (x, t', trm) else Abs (x, t', lift-lvar n trm) |
lift-lvar - trm = trm

end
›
```

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) dum-
myT), replicate (i−1) (const @{const-name snd-lens}))
       else foldl lens-compf (Const (@{const-name fst-lens}, lensT (nth ts i) dum-
myT), 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*),
  (@{*syntax-const* -*lvar-abs*}
  , *fn ctx* => *fn terms* =>
   let val [*Free* (*x*, -), *tty*, *trm*] = *terms*
     val ty = *Syntax-Phases.decode-typ tty*
     val lens = *Type* (@{*type-name lens-ext*}, [*ty*, *dummyT*, @{*typ unit*}])
     val trm' = *Lift-Expr-LVar.lift-lvar x trm*
   in Abs (*x*, *lens*, *abstract-over* (*free x*, *trm'*)) *end*)]
 end
›

term *localstate* (*x*::*int*, *y*::*int*) *over* 1_L

end

10 Shallow Expressions Meta-Theory

theory *Shallow-Expressions*
 imports
  Variables Expressions Unrestriction Substitutions Extension Liberation Quantifiers Collections
  Named-Expressions Local-State
begin end

11 Expression Test Cases and Examples

theory *Expressions-Tests*
 imports *Expressions Named-Expressions*
begin

Some examples of lifted expressions follow. For now, we turn off the pretty printer so that we can see the results of the parser.

declare [[*pretty-print-exprs=false*]]

term $(f + g)_e$ — Lift an expression and insert -*sexp-pqt* for pretty printing

term $(f + g)^e$ — Lift an expression and don't insert *-sexp-pqt*

The default behaviour of our parser is to recognise identifiers as expression variables. So, the above expression becomes the term $[\lambda s.\ f\ s + g\ s]_e$. We can easily change this using the attribute *literal-variables*:

declare [[*literal-variables*]]

term $(f + g)_e$

Now, f and g are both parsed as literals, and so the term is $[\lambda s.\ f + g]_e$. Alternatively, we could have a lens in the expression, by marking a free variable with a dollar :

term $(\$x + g)_e$

This gives the term $[\lambda s.\ get_x\ s + g]_e$. Although we have default behaviours for parsing, we can use different markup to coerce identifiers to particular variable kinds.

term $(\$x + @g)_e$

This gives $[\lambda s.\ get_x\ s + g\ s]_e$, the we have requested that g is treated as an expression variable. We can do similar with literal, as show below.

term $(f + «x»)_e$

Some further examples follow.

term $(«f»\ (@e))_e$

term $(@f + @g)_e$

term $(@x)_e$

term $(\$x{:}y{:}z)_e$

term $((\$x{:}y){:}z)_e$

term $(x{::}nat)_e$

term $(\forall\ x{::}nat.\ x > 2)_e$

term $SEXP(\lambda\ s.\ get_x\ s + e\ s + v)$

term $(v \in \$xs \cup (\$f)\ ys \cup \{\} \wedge @e)_e$

We now turn pretty printing back on, so we can see how the user sees expressions.

declare [[*pretty-print-exprs*, *literal-variables=false*]]

term $(\$x^< = \$x^>)_e$

term ($\$x.1 = \$y.2)_e$

The pretty printer works even when we don't use the parser, as shown below.

term $[\lambda\ s.\ get_x\ s + e\ s + v]_e$

By default, dollars are printed next to free variables that are lenses. However, we can alter this behaviour with the attribute *mark-state-variables*:

declare [[*mark-state-variables=false*]]

term $(\$x + e + v)_e$

This way, the x variable is indistinguishable when printed from the e and v. Usually, this information can be inferred from the types of the entities:

alphabet *st* =
  x :: *int*

term $(x + e + v)_e$

expression *x-is-big* **over** *st* **is** $x > 1000$

term $(x\text{-}is\text{-}big \longrightarrow x > 0)_e$

Here, x is a lens defined by the **alphabet** command, and so the lifting translation treats it as a state variable. This is hidden from the user.

dataspace *testspace* =
  variables *z* :: *int*

declare [[*literal-variables*]]

context *testspace*
begin

edefinition *z-is-bigger y* = $(z > y)$

term $(z\text{-}is\text{-}bigger\ (z + 1))_e$

end

end

12 Examples of Shallow Expressions

theory *Shallow-Expressions-Examples*
  imports *Shallow-Expressions*
begin

12.1 Basic Expressions and Queries

We define some basic variables using the **alphabet** command, process some simple expressions, and then perform some unrestriction queries and substitution transformations.

declare [[*literal-variables*]]

alphabet *st* =
  v1 :: *int*
  v2 :: *int*
  v3 :: *string*

term $(v1 > a)_e$

declare [[*pretty-print-exprs=false*]]

term $(v1 > a)_e$

declare [[*pretty-print-exprs*]]

lemma $\$v2 \sharp (v1 > 5)_e$
  by *unrest*

lemma $(v1 > 5)_e[\![v2/v1]\!] = (v2 > 5)_e$
  by *subst-eval*

We sometimes would like to define "constructors" for expressions. These are functions that produce expressions, and may also have expressions as arguments. Unlike for other functions, during lifting the state is not passed to the arguments, but is passed to the constructor constant itself. An example is given below:

definition *v1-greater* :: *int* ⇒ (*bool*, *st*) *expr* **where**
$v1\text{-}greater\ x = (v1 > x)_e$

expr-constructor *v1-greater*

Definition *v1-greater* is a constructor for an expression, and so it should not be lifted. Therefore we use the command **expr-constructor** to specify this, which modifies the behaviour of the lifting parser, and means that $(v1\text{-}greater\ 7)_e$ is correctly translated.

If it is desired that one or more of the arguments is an expression, then this can be specified using an optional list of numbers. In the example below, the first argument is an expression.

definition *v1-greater'* :: (*int*, *st*) *expr* ⇒ (*bool*, *st*) *expr* **where**
$v1\text{-}greater'\ x = (v1 > @x)_e$

expr-constructor *v1-greater'* (*0*)

term $(v1\text{-}greater'\ (v1\ +\ 1))_e$

We also sometimes wish to have functions that return expressions, whose arguments should be lifted. We can achieve this using the **expr-function** command:

definition *v1-less* :: *int* ⇒ (*bool*, *st*) *expr* **where**
$v1\text{-}less\ x = (v1\ <\ x)_e$

expr-function *v1-less*

This means, we can parse terms like $(v1\text{-}less\ (\$v1\ +\ 1))_e$ – notice that this returns an expression and takes an expression as an input. Alternatively, we can achieve the same effect with the **edefinition** command, which is like **definition**, but uses the expression parse and lifts the arguments as expressions. It is typically used for user-level functions that depend on the state.

edefinition *v1-less′* **where** $v1\text{-}less'\ x = (v1\ <\ x)$

term $(v1\text{-}less'\ (v1\ +\ 1))_e$

In addition, we can define an expression using the command below, which automatically performs expression lifting in the defining term. These constants are also set as expression constructors.

expression *v1-is-big* **over** *st* **is** $v1\ >\ 100$

expression *inc-v1* **over** $st \times st$ **is** $v1^{>}\ =\ v1^{<}\ +\ 1$

Definitional equations for named expressions are collected in the theorem attribute $v1\text{-}less'\ ?x = (\$v1\ <\ ?x)_e$
$v1\text{-}is\text{-}big = (100\ <\ \$v1)_e$
$inc\text{-}v1\ = ((\$v1)^{>}\ =\ (\$v1)^{<}\ +\ 1)_e$.

thm *named-expr-defs*

12.2 Hierarchical State

alphabet *person* =
  name :: *string*
  age :: *nat*

alphabet *company* =
  adam :: *person*
  bella :: *person*
  carol :: *person*

term $(\$adam{:}age > \$carol{:}age)_e$

term $(\$adam{:}name \neq \$bella{:}name)_e$

12.3 Program Semantics

We give a predicative semantics to a simple imperative programming language with sequence, conditional, and assignment, using lenses and shallow expressions. We then use these definitions to prove some basic laws of programming.

declare [[*literal-variables=false*]]

type-synonym *'s prog* = *'s* × *'s* ⇒ *bool*

definition *seq* :: *'s prog* ⇒ *'s prog* ⇒ *'s prog* (**infixr** ;; *85*) **where**
[*expr-defs*]: $seq\ P\ Q = (\exists\ s.\ P[\![«s»/\mathbf{v}^>]\!] \wedge Q[\![«s»/\mathbf{v}^<]\!])_e$

definition *ifthenelse* :: (*bool*, *'s*) *expr* ⇒ *'s prog* ⇒ *'s prog* ⇒ *'s prog* **where**
[*expr-defs*]: $ifthenelse\ b\ P\ Q = (if\ b^<\ then\ P\ else\ Q)_e$

definition *assign* :: (*'a* ⟹ *'s*) ⇒ (*'a*, *'s*) *expr* ⇒ *'s prog* **where**
[*expr-defs*]: $assign\ x\ e = (\$x^> = e^< \wedge \mathbf{v}^> \simeq_{«x»} \mathbf{v}^<)_e$

syntax
  -assign :: *svid* ⇒ *logic* ⇒ *logic* (- ::= - [*86*, *87*] *87*)
  -ifthenelse :: *logic* ⇒ *logic* ⇒ *logic* ⇒ *logic* (*IF* - *THEN* - *ELSE* - [*0*, *0*, *84*] *84*)

The syntax translations insert the expression brackets, which means the expressions are lifted, without this being visible to the user.

translations
  -assign x e == *CONST assign x* $(e)_e$
  -ifthenelse b P Q == *CONST ifthenelse* $(b)_e$ *P Q*

lemma *seq-assoc*: *P* ;; (*Q* ;; *R*) = (*P* ;; *Q*) ;; *R*
  by *expr-auto*

lemma *ifthenelse-seq-distr*: (*IF B THEN P ELSE Q*) ;; *R* = *IF B THEN P* ;; *R ELSE Q* ;; *R*
  by *expr-auto*

lemma *assign-twice*:
  assumes *mwb-lens x*
  shows *x* ::= *e* ;; *x* ::= *f* = *x* ::= $f[\![e/x]\!]$
  using *assms*
  apply *expr-simp*
  apply (*metis mwb-lens.put-put mwb-lens-weak weak-lens.put-get*)
  done

lemma *assign-commute*:
  assumes *mwb-lens x mwb-lens y x* ⋈ *y* $\$y \sharp (e)_e$ $\$x \sharp (f)_e$

shows $(x ::= e\ ;;\ y ::= f) = (y ::= f\ ;;\ x ::= e)$
using *assms*
apply *expr-simp*
apply *safe*
apply (*metis lens-indep-def mwb-lens-weak weak-lens.put-get*)+
done

lemma *assign-combine*:
assumes *mwb-lens x mwb-lens y* $x \bowtie y$ $\$x \sharp (f)_e$
shows $(x ::= e\ ;;\ y ::= f) = (x, y) ::= (e, f)$
using *assms*
apply *expr-simp*
apply *safe*
apply (*simp-all add*: *lens-indep.lens-put-comm*)
apply (*metis mwb-lens-weak weak-lens.put-get*)
done

Below, we apply the assignment commutativity law in a small example:

declare [[*literal-variables*]]

lemma *assign-commute-example*:
adam:*name* ::= ''*Adam*'' ;; *bella*:*name* ::= ''*Bella*'' =
bella:*name* ::= ''*Bella*'' ;; *adam*:*name* ::= ''*Adam*''
proof (*rule assign-commute*)
— We show the two variables satisfy the lens axioms
show *mwb-lens* $(adam{:}name)_v$ **by** *simp*
show *mwb-lens* $(bella{:}name)_v$ **by** *simp*

— We show the two variables are independent
show $(adam{:}name)_v \bowtie (bella{:}name)_v$ **by** *simp*

— We show that neither assigned expression depends on the opposite variable
show $\$bella{:}name \sharp (''Adam'')_e$ **by** *unrest*
show $\$adam{:}name \sharp (''Bella'')_e$ **by** *unrest*
qed

end

References

[1] B. Dongol, I. Hayes, L. Meinicke, and G. Struth. Cylindric Kleene lattices for program construction. In *MPC*, volume 11825 of *LNCS*, pages 192–225. Springer, October 2019.

[2] S. Foster and J. Baxter. Automated algebraic reasoning for collections and local variables with lenses. In *RAMiCS*, volume 12062 of *LNCS*. Springer, 2020.

[3] S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197, October 2020.

[4] S. Foster, C.-K. Hur, and J. Woodcock. Unifying model execution and deductive verification with Interaction Trees in Isabelle/HOL. *ACM Trans. on Software Engineering Methodology (TOSEM)*, 2024.

[5] S. Foster, C. Pardillo-Laursen, and F. Zeyda. Optics. *Archive of Formal Proofs*, May 2017. https://isa-afp.org/entries/Optics.html, Formal proof development.

[6] J. J. Huerta y Munive, S. Foster, M. Gleirscher, G. Struth, C. P. Laursen, and T. Hickman. IsaVODEs: Interactive Verification of Cyber-Physical Systems at Scale. *Journal of Automated Reasoning*, 2024.

[7] M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In *UTP 2006*, volume 4010 of *LNCS*, pages 123–140. Springer, 2007.