

Shallow Expressions

Simon Foster
University of York, UK

`simon.foster@york.ac.uk`

June 15, 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	3
2	Variables as Lenses	3
2.1	Constructors	4
2.2	Syntax Translations	7
2.3	Simplifications	10
3	Expressions	10
3.1	Types and Constructs	11
3.2	Lifting Parser and Printer	12
3.3	Reasoning	14
3.4	Algebraic laws	15

4	Unrestriction	16
5	Substitutions	19
5.1	Types and Constants	19
5.2	Syntax Translations	20
5.3	Substitution Laws	22
5.4	Proof rules	24
5.5	Ordering substitutions	25
5.6	Substitution Unrestriction Laws	25
5.7	Conditional Substitution Laws	26
5.8	Substitution Restriction Laws	26
5.9	Evaluation	26
6	Extension and Restriction	28
6.1	Syntax	28
6.2	Laws	29
6.3	Substitutions	30
6.4	Liberation	31
6.5	Definition and Syntax	31
6.6	Laws	31
6.7	Quantifying Lenses	32
6.8	Operators and Syntax	32
6.9	Laws	33
6.10	Cylindric Algebra	34
7	Collections	35
7.1	Partial Lens Definedness	35
7.2	Dynamic Lenses	35
7.3	Overloaded Collection Lens	36
7.4	Syntax for Collection Lenses	37
8	Named Expression Definitions	37
9	Local State	39
10	Shallow Expressions Meta-Theory	42
11	Expression Test Cases and Examples	42
12	Examples of Shallow Expressions	44
12.1	Basic Expressions and Queries	45
12.2	Hierarchical State	46
12.3	Program Semantics	47

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], $x \# e$, 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, $[x_1 \rightsquigarrow e_1, x_2 \rightsquigarrow e_2, \dots]$.
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 unrestricted 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-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-alpha =  $\top_S$ 

```

```

definition emp-alpha :: ' $s$  scene where
[lens-defs]: emp-alpha =  $\perp_S$ 

```

```

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

```

```

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

```

```

definition var-fst :: (' $a$   $\times$  ' $b$   $\Longrightarrow$  ' $s$ )  $\Rightarrow$  (' $a$   $\Longrightarrow$  ' $s$ ) where
[lens-defs]: var-fst  $x$  = fst $L$  ; $L$   $x$ 

```

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

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

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

lemma $ns\text{-}alpha\text{-}weak$ $[simp]: \llbracket weak\text{-}lens\ a; weak\text{-}lens\ x \rrbracket \implies weak\text{-}lens\ (ns\text{-}alpha\ a\ x)$
by ($simp\ add: ns\text{-}alpha\text{-}def\ comp\text{-}weak\text{-}lens$)

lemma $ns\text{-}alpha\text{-}mwb$ $[simp]: \llbracket mwb\text{-}lens\ a; mwb\text{-}lens\ x \rrbracket \implies mwb\text{-}lens\ (ns\text{-}alpha\ a\ x)$
by ($simp\ add: ns\text{-}alpha\text{-}def\ comp\text{-}mwb\text{-}lens$)

lemma $ns\text{-}alpha\text{-}vwb$ $[simp]: \llbracket vwb\text{-}lens\ a; vwb\text{-}lens\ x \rrbracket \implies vwb\text{-}lens\ (ns\text{-}alpha\ a\ x)$
by ($simp\ add: ns\text{-}alpha\text{-}def\ comp\text{-}vwb\text{-}lens$)

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

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

lemma $ns\text{-}alpha\text{-}indep\text{-}3$ $[simp]: x \bowtie b \implies x \bowtie ns\text{-}alpha\ b\ y$
by ($simp\ add: lens\text{-}indep\text{-}sym$)

lemma $ns\text{-}alpha\text{-}indep\text{-}4$ $[simp]: \llbracket mwb\text{-}lens\ a; x \bowtie y \rrbracket \implies ns\text{-}alpha\ a\ x \bowtie ns\text{-}alpha\ a\ y$
by ($simp\ add: ns\text{-}alpha\text{-}def$)

lemma $var\text{-}fst\text{-}mwb$ $[simp]: mwb\text{-}lens\ x \implies mwb\text{-}lens\ (var\text{-}fst\ x)$
by ($simp\ add: var\text{-}fst\text{-}def\ comp\text{-}mwb\text{-}lens$)

lemma $var\text{-}snd\text{-}mwb$ $[simp]: mwb\text{-}lens\ x \implies mwb\text{-}lens\ (var\text{-}snd\ x)$
by ($simp\ add: var\text{-}snd\text{-}def\ comp\text{-}mwb\text{-}lens$)

lemma $var\text{-}fst\text{-}vwb$ $[simp]: vwb\text{-}lens\ x \implies vwb\text{-}lens\ (var\text{-}fst\ x)$
by ($simp\ add: var\text{-}fst\text{-}def\ comp\text{-}vwb\text{-}lens$)

lemma $var\text{-}snd\text{-}vwb$ $[simp]: vwb\text{-}lens\ x \implies vwb\text{-}lens\ (var\text{-}snd\ x)$
by ($simp\ add: var\text{-}snd\text{-}def\ comp\text{-}vwb\text{-}lens$)

lemma $var\text{-}fst\text{-}indep\text{-}1$ $[simp]: x \bowtie y \implies var\text{-}fst\ x \bowtie y$
by ($simp\ add: var\text{-}fst\text{-}def\ lens\text{-}indep\text{-}left\text{-}ext$)

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

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

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

lemma *mwb-var-pair* [simp]: $\llbracket \text{mwb-lens } x; \text{mwb-lens } y; x \bowtie y \rrbracket \implies \text{mwb-lens } (\text{var-pair } x \ y)$
by (*simp add: var-pair-def plus-mwb-lens*)

lemma *vwb-var-pair* [simp]: $\llbracket \text{vwb-lens } x; \text{vwb-lens } y; x \bowtie y \rrbracket \implies \text{vwb-lens } (\text{var-pair } x \ y)$
by (*simp add: var-pair-def*)

lemma *var-pair-pres-indep* [simp]:
 $\llbracket x \bowtie y; x \bowtie z \rrbracket \implies x \bowtie \text{var-pair } y \ z$
 $\llbracket x \bowtie y; x \bowtie z \rrbracket \implies \text{var-pair } y \ z \bowtie x$
by (*simp-all add: var-pair-def lens-indep-sym*)

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

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

lemma *var-alpha-combine*: $\llbracket \text{vwb-lens } x; \text{vwb-lens } y; x \bowtie y \rrbracket \implies \text{var-alpha } x \sqcup_S \text{var-alpha } y = \text{var-alpha } (x +_L y)$
by (*simp add: lens-plus-scene var-alpha-def*)

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

lemma *pre-var-indep-prod* [simp]: $x \bowtie a \implies \text{ns-alpha } \text{fst}_L \ x \ \bowtie \ a \ \times_L \ b$
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]: $x \bowtie b \implies \text{ns-alpha } \text{snd}_L \ x \ \bowtie \ a \ \times_L \ b$
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]:
 $(X \ \bowtie \ Y) \implies \text{var-alpha } X \ \bowtie_S \ \text{var-alpha } Y$
by (*simp add: var-alpha-def*)

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

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

lemma *var-alpha-override* [*simp*]:

mwb-lens $X \implies s_1 \oplus_S s_2$ 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* $y \iff x \subseteq_L y$

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* $y \iff x \subseteq_L y$

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* \Rightarrow *svid* (- [999] 999)

-svlongid :: *longid-position* \Rightarrow *svid* (- [999] 999)

:: *svid* \Rightarrow *svids* (-)

-svid-list :: *svid* \Rightarrow *svids* \Rightarrow *svids* (-, / -)

-svid-alpha :: *svid* (**v**)

-svid-index :: *id-position* \Rightarrow *logic* \Rightarrow *svid* (-'(-') [999] 999)

-svid-tuple :: *svids* \Rightarrow *svid* ('(-'))

-svid-dot :: *svid* \Rightarrow *svid* \Rightarrow *svid* (-:- [999,998] 998)

-svid-res :: *svid* \Rightarrow *svid* \Rightarrow *svid* (-|- [999,998] 998)

-svid-pre :: *svid* \Rightarrow *svid* (-< [997] 997)

-svid-post :: *svid* \Rightarrow *svid* (-> [997] 997)

-svid-fst :: *svid* \Rightarrow *svid* (-.1 [997] 997)

-svid-snd :: *svid* \Rightarrow *svid* (-.2 [997] 997)

-mk-svid-list :: *svids* \Rightarrow *logic* — Helper function for summing a list of identifiers

-of-svid-list :: *logic* \Rightarrow *svids* — Reverse of the above

-svid-view :: *logic* \Rightarrow *svid* (\mathcal{V} [-]) — View of a symmetric lens

-svid-coview :: *logic* \Rightarrow *svid* ($\mathcal{C}[-]$) — Coview of a symmetric lens
-svid-prod :: *svid* \Rightarrow *svid* \Rightarrow *svid* (**infixr** \times 85)
-svid-pow2 :: *svid* \Rightarrow *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* \Rightarrow *salpha* (- [990] 990)
-salphavar :: *svid* \Rightarrow *salpha* (\$) [990] 990)
-salphaparen :: *salpha* \Rightarrow *salpha* ('(-'))
-salphaunion :: *salpha* \Rightarrow *salpha* \Rightarrow *salpha* (**infixr** \cup 75)
-salphainter :: *salpha* \Rightarrow *salpha* \Rightarrow *salpha* (**infixr** \cap 75)
-salphaminus :: *salpha* \Rightarrow *salpha* \Rightarrow *salpha* (**infixl** - 65)
-salphacompl :: *salpha* \Rightarrow *salpha* (- - [81] 80)
-salpha-all :: *salpha* (Σ)
-salpha-none :: *salpha* (\emptyset)

-salphaset :: *svids* \Rightarrow *salpha* ({-})
-sframeid :: *id* \Rightarrow *sframe* (-)
-sframeset :: *svids* \Rightarrow *sframe-enum* ($\{\!-\!\}$)
-sframeunion :: *sframe* \Rightarrow *sframe* \Rightarrow *sframe* (**infixr** \cup 75)
-sframeinter :: *sframe* \Rightarrow *sframe* \Rightarrow *sframe* (**infixr** \cap 75)
-sframeminus :: *sframe* \Rightarrow *sframe* \Rightarrow *sframe* (**infixl** - 65)
-sframecompl :: *sframe* \Rightarrow *sframe* (- - [81] 80)
-sframe-all :: *sframe* (Σ)
-sframe-none :: *sframe* (\emptyset)
-sframe-pre :: *sframe* \Rightarrow *sframe* (< [989] 989)
-sframe-post :: *sframe* \Rightarrow *sframe* (> [989] 989)
-sframe-enum :: *sframe-enum* \Rightarrow *sframe* (-)
-sframe-alpha :: *sframe-enum* \Rightarrow *salpha* (-)
-salphamk :: *logic* \Rightarrow *salpha*
-mk-alpha-list :: *svids* \Rightarrow *logic*
-mk-frame-list :: *svids* \Rightarrow *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* \Rightarrow *logic* ($\{-\}_v$)
-svid-empty :: *logic* ($\{\}_v$)
-svar :: *svid* \Rightarrow *logic* ('(-')_v)

For various reasons, the syntax constructors above all yield specific grammar categories and will not parser at the HOL top level (basically this is to do

with us wanting to reuse the syntax for expressions). As a result we provide some quotation constructors above.

Next we need to construct the syntax translations rules. Finally, we set up the translations rules.

translations

— Identifiers

$-svid\ x \rightarrow x$
 $-svlongid\ x \rightarrow x$
 $-svid\text{-}alpha \equiv CONST\ univ\text{-}var$
 $-svid\text{-}tuple\ xs \rightarrow -mk\text{-}svid\text{-}list\ xs$
 $-svid\text{-}dot\ x\ y \equiv CONST\ ns\text{-}alpha\ x\ y$
 $-svid\text{-}index\ x\ i \rightarrow x\ i$
 $-svid\text{-}res\ x\ y \equiv x\ /_L\ y$
 $-svid\text{-}pre\ x \equiv -svid\text{-}dot\ fst_L\ x$
 $-svid\text{-}post\ x \equiv -svid\text{-}dot\ snd_L\ x$
 $-svid\text{-}fst\ x \equiv CONST\ var\text{-}fst\ x$
 $-svid\text{-}snd\ x \equiv CONST\ var\text{-}snd\ x$
 $-svid\text{-}prod\ x\ y \equiv x\ \times_L\ y$
 $-svid\text{-}pow2\ x \rightarrow x\ \times_L\ x$
 $-mk\text{-}svid\text{-}list\ (-svid\text{-}list\ x\ xs) \rightarrow CONST\ var\text{-}pair\ x\ (-mk\text{-}svid\text{-}list\ xs)$
 $-mk\text{-}svid\text{-}list\ x \rightarrow x$
 $-mk\text{-}alpha\text{-}list\ (-svid\text{-}list\ x\ xs) \rightarrow CONST\ var\text{-}alpha\ x\ \sqcup_S\ -mk\text{-}alpha\text{-}list\ xs$
 $-mk\text{-}alpha\text{-}list\ x \rightarrow CONST\ var\text{-}alpha\ x$

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

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

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

— Alphabets

$-salphaparen\ a \rightarrow a$
 $-salphaid\ x \rightarrow x$
 $-salphaunion\ x\ y \rightarrow x\ \sqcup_S\ y$
 $-salphainter\ x\ y \rightarrow x\ \sqcap_S\ y$
 $-salphaminus\ x\ y \rightarrow x - y$
 $-salphacompl\ x \rightarrow -x$

$-salphavar\ x \equiv CONST\ var\text{-}alpha\ x$
 $-salphaset\ A \rightarrow -mk\text{-}alpha\text{-}list\ A$
 $-sframeid\ A \rightarrow A$
 $(-svid\text{-}list\ x\ (-salphamk\ y)) \leftarrow -salphamk\ (x +_L\ y)$

```

x ← -salphamk x
-salpha-all ⇒ CONST univ-alpha
-salpha-none ⇒ CONST emp-alpha

```

```

— Quotations
-svid-set A → -mk-alpha-list A
-svid-empty → 0L
-svar x → x

```

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

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

syntax

```
-uvar-ty    :: type ⇒ type ⇒ type
```

parse-translation ‹

```

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

```

2.3 Simplifications

```

lemma get-pre [simp]: get(x<)v (s1, s2) = getx s1
by (simp add: lens-defs)

```

```

lemma get-post [simp]: get(x>)v (s1, s2) = getx s2
by (simp add: lens-defs)

```

```

lemma get-prod-decomp: getx s = (getvar-fst x s, getvar-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 $\lambda s. \text{get}_x s + \text{get}_y s$ 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, $(-)_e$ to support this.

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

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

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

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

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

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

abbreviation $(\text{input}) \text{ var} :: ('a \Longrightarrow 's) \Rightarrow ('a, 's) \text{ expr}$ **where**
 $\text{var } x \equiv (\lambda s. \text{get}_x s)$

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

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

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

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

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

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

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  $\Rightarrow$  logic ('(-)e)
— Convert the expression to a lambda term, but do not box it.
-sexp-quote-1way :: logic  $\Rightarrow$  logic ('(-)e)
-sexp-lit       :: logic  $\Rightarrow$  logic («-»)
-sexp-var       :: svid  $\Rightarrow$  logic ($- [990] 990)
-sexp-evar      :: id-position  $\Rightarrow$  logic (@- [999] 999)
-sexp-evar      :: logic  $\Rightarrow$  logic (@'(-) [999] 999)
-sexp-pqt      :: logic  $\Rightarrow$  sexp ([-]e)
-sexp-taut     :: logic  $\Rightarrow$  logic ('-')
-sexp-select   :: logic  $\Rightarrow$  svid  $\Rightarrow$  logic (-: [1000, 999] 1000)
-sexp-if       :: logic  $\Rightarrow$  logic  $\Rightarrow$  logic  $\Rightarrow$  logic ((?- < - >/ -) [52,0,53] 52)

```

ML-file \langle *Lift-Expr.ML* \rangle

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 \langle

$[[$ $(@$ $\{$ *syntax-const* *sexp-state* $\},$ *fn ctx* \Rightarrow *fn term* \Rightarrow *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 => getx -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* \Rightarrow *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*]: $fst (case p of Inl x \Rightarrow (a1 x, a2 x) | Inr x \Rightarrow (b1 x, b2 x)) = (case p of Inl x \Rightarrow a1 x | Inr x \Rightarrow b1 x)$
by (*simp add: sum.case-eq-if*)

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

lemma *sum-case-apply* [*simp*]: $(case p of Inl x \Rightarrow f x | Inr x \Rightarrow g x) y = (case p of Inl x \Rightarrow f x y | Inr x \Rightarrow g x y)$
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*]: $P \triangleleft \text{True} \triangleright Q = P$
by *expr-auto*

lemma *expr-if-false* [*simp*]: $P \triangleleft \text{False} \triangleright Q = Q$
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*: $[\text{expr-if } P \ b \ Q]_e = \text{expr-if } P \ b \ Q$

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 \Rightarrow 'p \Rightarrow bool

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

adhoc-overloading *unrest* \equiv *unrest-expr*

syntax

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

translations

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

named-theorems *unrest*

lemma *unrest-empty* [*unrest*]: $\emptyset \# P$
 by (simp add: *expr-defs* *lens-defs*)

lemma *unrest-var-union* [*unrest*]:
 [[$A \# P$; $B \# P$]] \Longrightarrow $A \cup B \# P$
 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 $(- (A \cup B)) \# P$

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 \Longrightarrow ($\$x \# e$) = (\forall 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 $x \implies (- \$x \# e) = (\forall s s'. e (put_x s' (get_x s)) = e s)$

by (*simp add: unrest-expr-def var-alpha-def comp-mwb-lens lens-override-def scene-override-commute*)

lemma *unrest-subscene*: $\llbracket idem-scene\ a; a \# e; b \subseteq_S a \rrbracket \implies b \# e$

by (*metis subscene-eliminate unrest-expr-def*)

lemma *unrest-lens-comp* [*unrest*]: $\llbracket mwb-lens\ x; mwb-lens\ y; \$x \# e \rrbracket \implies \$x:y \# e$

by (*simp add: unrest-lens, simp add: lens-comp-def ns-alpha-def*)

lemma *unrest-expr* [*unrest*]: $x \# e \implies x \# (e)_e$

by (*simp add: expr-defs*)

lemma *unrest-lit* [*unrest*]: $x \# (\llbracket v \rrbracket)_e$

by (*simp add: expr-defs*)

lemma *unrest-var* [*unrest*]:

$\llbracket vwb-lens\ x; idem-scene\ a; var-alpha\ x \bowtie_S a \rrbracket \implies a \# (\$x)_e$

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*]:

$\llbracket mwb-lens\ x; x \bowtie y \rrbracket \implies \$x \# (\$y)_e$

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* $x\ \$x \# P\ y \subseteq_L x$

shows $\$y \# P$

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* $y\ x \approx_L y\ \$x \# P$

shows $\$y \# P$

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* $x\ \$x \# P$

shows $\Sigma \# P$

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 $(\Sigma \# P) \longleftrightarrow (\$x \# P)$

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 $\Sigma \# e$

shows $\$x \# e$

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 $\$x \# P$ $\$y \# P$

shows $\$(x, y) \# P$

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 $\$(x, y) \# P = (\$x \# P) \wedge (\$y \# P)$

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*]: $\llbracket \textit{mwb-lens } x; x \bowtie y \rrbracket \Longrightarrow \$x \# \textit{get}_y$

by (*expr-simp, simp add: lens-indep.lens-put-irr2*)

lemma *unrest-conj* [*unrest*]:

$\llbracket x \# P; x \# Q \rrbracket \Longrightarrow x \# (P \wedge Q)_e$

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

lemma *unrest-not* [*unrest*]:

$\llbracket x \# P \rrbracket \Longrightarrow x \# (\neg P)_e$

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

lemma *unrest-disj* [*unrest*]:

$\llbracket x \# P; x \# Q \rrbracket \Longrightarrow x \# (P \vee Q)_e$

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

lemma *unrest-implies* [*unrest*]:

$\llbracket x \# P; x \# Q \rrbracket \Longrightarrow x \# (P \longrightarrow Q)_e$

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

lemma *unrest-expr-if* [*unrest*]:

assumes $a \# P$ $a \# Q$ $a \# (e)_e$

shows $a \# (P \triangleleft e \triangleright Q)$

using *assms* **by** *expr-simp*

lemma *unrest-uop*:

$\llbracket x \# e \rrbracket \Longrightarrow x \# (\llbracket f \rrbracket e)_e$

by (auto simp add: expr-defs)

lemma unrest-bop:

$\llbracket x \# e_1; x \# e_2 \rrbracket \Longrightarrow x \# (\langle\langle f \rangle\rangle e_1 e_2)_e$
 by (auto simp add: expr-defs)

lemma unrest-trop:

$\llbracket x \# e_1; x \# e_2; x \# e_3 \rrbracket \Longrightarrow x \# (\langle\langle f \rangle\rangle e_1 e_2 e_3)_e$
 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* ($\langle\langle \sim \rangle\rangle$)

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

definition *subst-nil* :: ($'s_1, 's_2$) *psubst* ($\langle\langle \sim \rangle\rangle$)

where [*expr-defs*, *code-unfold*]: ($\langle\langle \sim \rangle\rangle$) = ($\lambda s. \text{undefined}$)

definition *subst-default* :: ($'s_1, 's_2::\text{default}$) *psubst* ($\langle\langle \sim \rangle\rangle$)

where [*expr-defs*, *code-unfold*]: ($\langle\langle \sim \rangle\rangle$) = ($\lambda s. \text{default}$)

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* $\sigma x e = (\lambda s. \text{put}_x (\sigma s) (e s))$

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* ($-\uparrow$ [999] 999) **where**

[*expr-defs*, *code-unfold*]: *subst-ext* $a = \text{get}_a$

definition *subst-res* :: ($'s_1 \Longrightarrow 's_2$) $\Rightarrow ('s_1, 's_2)$ *psubst* ($-\downarrow$ [999] 999) **where**

[*expr-defs*, *code-unfold*]: *subst-res* $a = \text{create}_a$

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

definition $subst\text{-}app :: ('s_1, 's_2) psubst \Rightarrow ('a, 's_2) expr \Rightarrow ('a, 's_1) expr$
where $[expr\text{-}defs]: subst\text{-}app \sigma e = (\lambda s. e (\sigma s))$

abbreviation $aext P a \equiv subst\text{-}app (a^\uparrow) P$

abbreviation $ares P a \equiv subst\text{-}app (a^\downarrow) P$

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

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

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

definition $unrest\text{-}usubst :: 's scene \Rightarrow 's subst \Rightarrow bool$
where $[expr\text{-}defs]: unrest\text{-}usubst a \sigma = (\forall s s'. \sigma (s \oplus_S s' \text{ on } a) = (\sigma s) \oplus_S s' \text{ on } a)$

definition $par\text{-}subst :: 's subst \Rightarrow 's scene \Rightarrow 's scene \Rightarrow 's subst \Rightarrow 's subst$
where $[expr\text{-}defs]: par\text{-}subst \sigma_1 A B \sigma_2 = (\lambda s. (s \oplus_S (\sigma_1 s) \text{ on } A) \oplus_S (\sigma_2 s) \text{ on } B)$

definition $subst\text{-}restrict :: 's subst \Rightarrow 's scene \Rightarrow 's subst$ **where**
 $[expr\text{-}defs]: subst\text{-}restrict \sigma a = (\lambda s. s \oplus_S \sigma s \text{ on } a)$

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

definition $sset :: 's scene \Rightarrow 's \Rightarrow 's subst$
where $[expr\text{-}defs, code\text{-}unfold]: sset a s' = (\lambda s. s \oplus_S s' \text{ on } a)$

syntax $\text{-}sset :: salpha \Rightarrow logic \Rightarrow logic (sset[-, -])$
translations $\text{-}sset a P == CONST sset a P$

5.2 Syntax Translations

nonterminal $uexprs$ and $smaplet$ and $smaplets$

syntax

$\text{-}subst\text{-}app :: logic \Rightarrow logic \Rightarrow logic$ **(infix** \dagger **65)**

$\text{-}smaplet :: [svid, logic] \Rightarrow smaplet (- \rightsquigarrow -)$
 $:: smaplet \Rightarrow smaplets (-)$

$\text{-}SMaplets :: [smaplet, smaplets] \Rightarrow smaplets (-, / -)$

— A little syntax utility to extract a list of variable identifiers from a substitution

$\text{-}smaplets\text{-}svids :: smaplets \Rightarrow logic$

$\text{-}SubstUpd :: [logic, smaplets] \Rightarrow logic (-/'(-) [900,0] 900)$

$\text{-}Subst :: smaplets \Rightarrow logic ((1[-]))$

```

-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 σ1 A B σ2 == CONST par-subst σ1 A B σ2
-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*]: $[\rightsquigarrow] \dagger P = P$
by (*expr-auto*)

lemma *subst-unrest* [*usubst*]:
 $\llbracket \text{vub-lens } x; \$x \# v \rrbracket \implies \sigma(x \rightsquigarrow e) \dagger v = \sigma \dagger v$
by *expr-auto*

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

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

lemma *subst-id-var*: $[\rightsquigarrow] = (\$v)_e$
by *expr-simp*

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

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

lemma *subst-default-id* [*simp*]: $\langle [\rightsquigarrow] \rangle \circ_s \sigma = \langle [\rightsquigarrow] \rangle$
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*: $\text{subst-app } \sigma [e]_e = [\text{subst-app } \sigma e]_e$
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 $(\forall k. P[\langle\langle k \rangle\rangle/x])_e$

lemma *subst-get* [*usubst*]: $\sigma \dagger \text{get}_x = \langle\sigma\rangle_s x$
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*: $\sigma \dagger (\langle\langle f \rangle\rangle e)_e = (\langle\langle f \rangle\rangle (\sigma \dagger e))_e$
by (*simp add: expr-defs*)

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

lemma *subst-lit* [*usubst*]: $\sigma \dagger (\langle\langle v \rangle\rangle)_e = (\langle\langle v \rangle\rangle)_e$
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 $x \implies [x \rightsquigarrow \$x] = [\rightsquigarrow]$
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*: $\llbracket \text{vwb-lens } x; \text{vwb-lens } y; x \bowtie y \rrbracket \implies \text{inj } [(x, y) \rightsquigarrow (\$y, \$x)]$
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* $x \wedge v$. ‘ $P \llbracket \llbracket v \rrbracket / x \rrbracket$ ’
shows ‘ P ’
using *assms* **by** (*expr-simp*, *metis* *vwb-lens.put-eq*)

lemma *eq-substI*:

assumes *vwb-lens* $x \wedge v$. $P[\llbracket v \rrbracket/x] = Q[\llbracket v \rrbracket/x]$
shows $P = Q$
using *assms* **by** (*expr-simp*, *metis vwb-lens.put-eq*)

lemma *bool-eq-substI*:

assumes *vwb-lens* x $P[\text{True}/x] = Q[\text{True}/x]$ $P[\text{False}/x] = Q[\text{False}/x]$
shows $P = Q$
by (*metis (full-types) assms eq-substI*)

lemma *less-eq-substI*:

assumes *vwb-lens* $x \wedge v$. $P[\llbracket v \rrbracket/x] \leq Q[\llbracket v \rrbracket/x]$
shows $P \leq Q$
using *assms* **by** (*expr-simp*, *metis le-funE le-funI vwb-lens-def wv-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) =
 \langle (*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*)
 \rangle

5.6 Substitution Unrestriction Laws

lemma *unrest-subst-lens* [*expr-simps*]: *mwb-lens* $x \implies (\$x \#_s \sigma) = (\forall s v. \sigma (\text{put}_x s v) = \text{put}_x (\sigma s) v)$

by (*simp add: unrest-usubst-def, metis lens-override-def mwb-lens-weak weak-lens.create-get*)

lemma *unrest-subst-empty* [*unrest*]: $x \#_s [\rightsquigarrow]$

by (*expr-simp*)

lemma *unrest-subst-upd* [*unrest*]: $\llbracket \text{vwb-lens } x; x \bowtie y; \$x \# (e)_e; \$x \#_s \sigma \rrbracket \implies \$x \#_s \sigma (y \rightsquigarrow e)$

by (*expr-auto add: lens-indep-comm*)

lemma *unrest-subst-upd-compl* [*unrest*]: $\llbracket \text{vwb-lens } x; y \subseteq_L x; -\$x \# (e)_e; -\$x \#_s \sigma \rrbracket \implies -\$x \#_s \sigma (y \rightsquigarrow e)$

by (*expr-auto, simp add: lens-override-def scene-override-commute*)

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

$\llbracket \$x \# P; \$x \#_s \sigma \rrbracket \implies \$x \# (\sigma \dagger P)$
by (*expr-auto*)

lemma *unrest-sset* [*unrest*]:

$x \bowtie y \implies \$x \#_s \text{sset}[\$y, v]$

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*]:

$\llbracket \text{vwb-lens } x; \$x \#_s \varrho \rrbracket \implies \sigma(x \rightsquigarrow u) \triangleleft b \triangleright \varrho = (\sigma \triangleleft b \triangleright \varrho)(x \rightsquigarrow (u \triangleleft b \triangleright (\$x)_e))$

by (*expr-auto*, *metis lens-override-def lens-override-idem*)

lemma *subst-cond-upd-3* [*usubst*]:

$\llbracket \text{vwb-lens } x; \$x \#_s \sigma \rrbracket \implies \sigma \triangleleft b \triangleright \varrho(x \rightsquigarrow v) = (\sigma \triangleleft b \triangleright \varrho)(x \rightsquigarrow ((\$x)_e \triangleleft b \triangleright v))$

by (*expr-auto*, *metis lens-override-def lens-override-idem*)

lemma *expr-if-bool-var-left*: $\text{vwb-lens } x \implies P[\text{True}/x] \triangleleft \$x \triangleright Q = P \triangleleft \$x \triangleright Q$

by (*expr-simp*, *metis (full-types) lens-override-def lens-override-idem*)

lemma *expr-if-bool-var-right*: $\text{vwb-lens } x \implies P \triangleleft \$x \triangleright Q[\text{False}/x] = P \triangleleft \$x \triangleright Q$

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*]: $\text{idem-scene } a \implies [\rightsquigarrow] \upharpoonright_s a = [\rightsquigarrow]$

by *expr-simp*

lemma *subst-restrict-out* [*usubst*]: $\llbracket \text{vwb-lens } x; \text{vwb-lens } a; x \bowtie a \rrbracket \implies \sigma(x \rightsquigarrow e)$

$\upharpoonright_s \$a = \sigma \upharpoonright_s \a

by (*expr-simp add: lens-indep.lens-put-irr2*)

lemma *subst-restrict-in* [*usubst*]: $\llbracket \text{vwb-lens } x; \text{vwb-lens } y; x \subseteq_L y \rrbracket \implies \sigma(x \rightsquigarrow e)$

$\upharpoonright_s \$y = (\sigma \upharpoonright_s \$y)(x \rightsquigarrow e)$

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*]: $weak\text{-}lens\ x \implies get_x ((\sigma(x \rightsquigarrow e)) s) = e$
by (*simp add: subst-upd-def SEXP-def*)

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

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

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

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

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

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

lemma *unrest-sset-lens* [*unrest*]: $\llbracket mwb\text{-}lens\ x; mwb\text{-}lens\ y; x \bowtie y \rrbracket \implies \$x \#_s$
 $sset[\$y, s]$
by (*simp add: sset-def unrest-subst-lens lens-indep-comm lens-override-def*)

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

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

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

lemma *subst-apply-unrest*:
 $\llbracket vwb\text{-}lens\ x; \$x \#_s \sigma \rrbracket \implies \langle \sigma \rangle_s x = var\ x$

proof –

assume 1: *vwb-lens* x and $\$x \#_s \sigma$

hence $\forall s\ s'. \sigma (s \oplus_L s' \text{ on } x) = \sigma s \oplus_L s' \text{ on } x$

by (*simp add: unrest-usubst-def*)

```

thus  $\langle \sigma \rangle_s x = \text{var } x$ 
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  $\Rightarrow$  svid  $\Rightarrow$  logic (infixl  $\uparrow$  80)
-ares    :: logic  $\Rightarrow$  svid  $\Rightarrow$  logic (infixl  $\downarrow$  80)
-pre     :: logic  $\Rightarrow$  logic (-< [999] 1000)
-post    :: logic  $\Rightarrow$  logic (-> [999] 1000)
-drop-pre :: logic  $\Rightarrow$  logic (-< [999] 1000)
-drop-post :: logic  $\Rightarrow$  logic (-> [999] 1000)

```

translations

```

-aext P a == CONST aext P a
-ares P a == CONST ares P a
-pre P == -aext (P)e fstL

```

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

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*]: $weak-lens\ a \implies P \uparrow a \downarrow a = P$
by (*simp add: expr-defs*)

lemma *aext-ares* [*alpha*]: $\llbracket mwb-lens\ a; (-\ \$a) \# P \rrbracket \implies P \downarrow a \uparrow a = P$
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*]: $\llbracket mwb-lens\ x; x \bowtie a \rrbracket \implies \$x \# (P \uparrow a)$
by (*expr-simp add: lens-indep.lens-put-irr2*)

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

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

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

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

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

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

6.3 Substitutions

definition *subst-aext* :: 'a subst \Rightarrow ('a \Longrightarrow 'b) \Rightarrow 'b subst
where [*expr-defs*]: *subst-aext* σ $x = (\lambda s. \text{put}_x s (\sigma (\text{get}_x s)))$

definition *subst-ares* :: 'b subst \Rightarrow ('a \Longrightarrow 'b) \Rightarrow 'a subst
where [*expr-defs*]: *subst-ares* σ $x = (\lambda s. \text{get}_x (\sigma (\text{create}_x s)))$

syntax

-*subst-aext* :: logic \Rightarrow *svld* \Rightarrow logic (**infixl** \uparrow_s 80)

-*subst-ares* :: logic \Rightarrow *svld* \Rightarrow logic (**infixl** \downarrow_s 80)

translations

-*subst-aext* P $a == \text{CONST } \text{subst-aext } P$ a

-*subst-ares* P $a == \text{CONST } \text{subst-ares } P$ a

lemma *unrest-subst-aext* [*unrest*]: $x \bowtie a \Longrightarrow \$x \#_s (\sigma \uparrow_s a)$

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 $x \Longrightarrow [\rightsquigarrow] \uparrow_s x = [\rightsquigarrow]$

by *expr-auto*

lemma *upd-subst-ext* [*alpha*]:

vwb-lens $x \Longrightarrow \sigma(y \rightsquigarrow e) \uparrow_s x = (\sigma \uparrow_s x)(x:y \rightsquigarrow e \uparrow x)$

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* $a \Longrightarrow [\rightsquigarrow] \downarrow_s a = [\rightsquigarrow]$

by *expr-auto*

lemma *upd-subst-res-in*:

$\llbracket \text{mwb-lens } a; x \subseteq_L a \rrbracket \Longrightarrow \sigma(x \rightsquigarrow e) \downarrow_s a = (\sigma \downarrow_s a)(x \dagger a \rightsquigarrow e \downarrow a)$

by (*expr-simp*, *fastforce*)

lemma *upd-subst-res-out*:

$\llbracket \text{mwb-lens } a; x \bowtie a \rrbracket \Longrightarrow \sigma(x \rightsquigarrow e) \downarrow_s a = \sigma \downarrow_s a$

by (*simp add: expr-defs lens-indep-sym*)

lemma *subst-ext-lens-apply*: $\llbracket \text{mwb-lens } a; -\$a \#_s \sigma \rrbracket \Longrightarrow (a^\uparrow \circ_s \sigma) \dagger P = ((\sigma \downarrow_s a) \dagger P) \uparrow a$

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

syntax

-liberate :: $\text{logic} \Rightarrow \text{salpha} \Rightarrow \text{logic}$ (**infixl** \ 80)

translations

-liberate $P \ x == \text{CONST } \text{liberate } P \ x$

-liberate $P \ x <= \text{-liberate } (P)_e \ x$

expr-constructor *liberate* (0)

6.6 Laws

lemma *liberate-lens* [*expr-simps*]:

$\text{mwb-lens } x \Longrightarrow P \ \backslash \ \$x = (\lambda \ s. \exists \ s'. P \ (s \triangleleft_x \ s'))$

by (*simp add: liberate-def*)

lemma *liberate-lens'*: $\text{mwb-lens } x \Longrightarrow P \ \backslash \ \$x = (\lambda \ s. \exists \ v. P \ (\text{put}_x \ s \ v))$

by (*auto simp add: liberate-def lens-defs fun-eq-iff*)

(*metis mwb-lens-weak weak-lens.put-get*)

lemma *liberate-as-subst*: $\text{vwb-lens } x \Longrightarrow e \ \backslash \ \$x = (\exists \ v. e \llbracket \langle v \rangle / x \rrbracket)_e$

by (*expr-simp*, *metis vwb-lens.put-eq*)

lemma *unrest-liberate*: $a \# P \ \backslash \ a$

by (*expr-simp*)

lemma *unrest-liberate-iff*: $(a \# P) \longleftrightarrow (P \ \backslash \ a = P)$

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

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

definition *all-expr* :: $('a \implies 's) \Rightarrow (bool, 's) \text{ expr} \Rightarrow (bool, 's) \text{ expr}$ **where**
[*expr-defs*]: *all-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* $x P == \text{CONST } \text{ex-expr } x P$
-*ex1-expr* $x P == \text{CONST } \text{ex1-expr } x P$
-*all-expr* $x P == \text{CONST } \text{all-expr } x P$

6.9 Laws

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

lemma *ex-unrest-iff*: $\llbracket \text{mwb-lens } x \rrbracket \implies (\$x \# P) \longleftrightarrow (\exists x \bullet P) = P$
by (*simp add: ex-is-liberation unrest-liberate-iff*)

lemma *ex-unrest*: $\llbracket \text{mwb-lens } x; \$x \# P \rrbracket \implies (\exists x \bullet P) = P$
using *ex-unrest-iff* **by** *blast*

lemma *unrest-ex-in* [*unrest*]:
 $\llbracket \text{mwb-lens } y; x \subseteq_L y \rrbracket \implies \$x \# (\exists y \bullet P)$
by (*simp add: ex-expr-def sublens-pres-mwb sublens-put-put unrest-lens*)

lemma *unrest-ex-out* [*unrest*]:
 $\llbracket \text{mwb-lens } x; \$x \# P; x \bowtie y \rrbracket \implies \$x \# (\exists y \bullet P)$
by (*simp add: ex-expr-def unrest-lens, metis lens-indep.lens-put-comm*)

lemma *subst-ex-out* [*usubst*]: $\llbracket \text{mwb-lens } x; \$x \#_s \sigma \rrbracket \implies \sigma \dagger (\exists x \bullet P) = (\exists x \bullet \sigma \dagger P)$
by (*expr-simp*)

lemma *subst-lit-ex-indep* [*usubst*]:
 $y \bowtie x \implies \sigma(y \rightsquigarrow \llbracket v \rrbracket) \dagger (\exists x \bullet P) = \sigma \dagger (\exists x \bullet [y \rightsquigarrow \llbracket v \rrbracket] \dagger P)$
by (*expr-simp*, *simp add: lens-indep.lens-put-comm*)

lemma *subst-ex-in* [*usubst*]:
 $\llbracket \text{vwb-lens } a; x \subseteq_L a \rrbracket \implies \sigma(x \rightsquigarrow e) \dagger (\exists a \bullet P) = \sigma \dagger (\exists a \bullet P)$
by (*expr-simp*, *force*)

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

lemma *ex-as-subst*: $\text{vwb-lens } x \implies (\exists x \bullet e) = (\exists v. e[\llbracket v \rrbracket/x])_e$
by *expr-auto*

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

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

lemma *ex-true* [*simp*]: $(\exists x \bullet (\text{True})_e) = (\text{True})_e$
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*: $wb\text{-lens } x \implies 'P \longrightarrow (\exists x \bullet P)'$
by (*expr-simp, metis wb-lens.get-put*)

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

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

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

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

lemma *ex-C6*:
assumes $wb\text{-lens } x \bowtie y \bowtie z$
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 $weak\text{-lens } x \bowtie y$
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 \implies 's) \Rightarrow (bool, 's) *expr* **where**
[expr-defs]: lens-defined *x* = ($\$v \in \mathcal{S}_{\langle x \rangle}$)_{*e*}

syntax *-lens-defined* :: *svid* \Rightarrow *logic* (**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 \implies 's)) \Rightarrow ('s \Rightarrow 'i) \Rightarrow ('a \implies 's) **where**
[lens-defs]: dyn-lens *f* *x* = (\lfloor *lens-get* = (λ *s*. *get_f* (*x* *s*) *s*), *lens-put* = (λ *s* *v*. *put_f* (*x* *s*) *s* *v*) \rfloor)

lemma *dyn-lens-mwb* [*simp*]: $\llbracket \bigwedge i. \text{mwb-lens } (f\ i); \bigwedge i. \$f(i) \# e \rrbracket \implies \text{mwb-lens } (dyn-lens\ f\ e)$

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*]: $\llbracket \bigwedge i. \text{vwb-lens } (f\ i); \bigwedge i. \$f(i) \# e \rrbracket \implies \text{vwb-lens } (\text{dyn-lens } f\ e)$

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*: $\llbracket \bigwedge i. \text{mwb-lens } (f\ i); \bigwedge i. \$f(i) \# e \rrbracket \implies \mathcal{S}_{\text{dyn-lens } f\ e} = \{s. s \in \mathcal{S}_f (e\ s)\}$

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*]: $\llbracket \bigwedge i. f\ i \bowtie x \rrbracket \implies \langle \text{subst-upd } \sigma\ (\text{dyn-lens } f\ k)\ e \rangle_s\ x = \langle \sigma \rangle_s\ x$

by (*expr-simp, metis (mono-tags, lifting) lens-indep.lens-put-irr2*)

lemma *get-upd-dyn-lens* [*usubst-eval*]: $\llbracket \bigwedge i. f\ i \bowtie x \rrbracket \implies \text{get}_x\ (\text{subst-upd } \sigma\ (\text{dyn-lens } f\ k)\ e\ s) = \text{get}_x\ (\sigma\ s)$

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* \implies *'s*)

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

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

ad hoc overloading

collection-lens \equiv *fun-collection-lens* **and**

collection-lens \equiv *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-collection-lens*} *i* = ($\lambda f. \text{fun-upd } f\ i$)

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*: $\mathcal{S}_{\text{list-collection-lens } i} = \{xs. i < \text{length } xs\}$

by (*simp add: list-collection-lens-def source-list-lens*)

lemma *put-list-collection-lens* [*simp*]:
 $put_{list-collection-lens} i = (\lambda xs. list-augment xs i)$
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* $f x i \equiv dyn-lens (\lambda k. ns-alpha x (f k)) i$

syntax
 $-svid-collection :: svid \Rightarrow logic \Rightarrow svid (-[-] [999, 0] 999)$

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

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

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

lemma *defined-list-collection-lens* [*simp*]:
 $\llbracket vwb-lens x; \$x \# e \rrbracket \Longrightarrow \mathbf{D}(x[e]) = (e < length(\$x))_e$
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*]:
 $vwb-lens x \Longrightarrow lens-defined (ns-alpha x (list-collection-lens i)) = (\llbracket i \rrbracket < length(\$x))_e$
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*]:
 $mwb-lens x \Longrightarrow get_x (subst-upd \sigma (dyn-lens-poly cl x (e)_e) v s)$
 $= lens-put (cl (e (\sigma s))) (get_x (\sigma s)) (v s)$
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 {verbose = false}
          (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 {verbose = false}
          (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-tyt ctx st
      val tyt = Syntax.read-tyt ctx t
      in named-expr n tyt stateT term ctx
  end

end;

val - =
let
  open Expr-Def;
in
  Outer-Syntax.local-theory command-keyword ‹definition› 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.tyt) - -- Scan.optional
  (@{keyword over} |-- Parse.tyt) -) --| @{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 :: pptrn => type
-state-tuple :: type => pptrn => logic
-state-lenses :: pptrn => 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 1L

```

```

end

```

10 Shallow Expressions Meta-Theory

```

theory Shallow-Expressions

```

```

  imports

```

```

    Variables Expressions Unrestriction Substitutions Extension Liberation Quanti-
    fiers 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 $[[\textit{literal-variables}]]$

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 $[\lambda s. \textit{get}_x\ s + g]_e$. 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 $[\lambda s. \textit{get}_x\ s + g\ s]_e$, the we have requested that g is treated as an expression variable. We can do similar with literal, as show below.

term $(f + \langle\langle x \rangle\rangle)_e$

Some further examples follow.

term $(\langle\langle f \rangle\rangle (\@e))_e$

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 $SEXP(\lambda s. \textit{get}_x\ s + e\ s + v)$

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 $[[\textit{pretty-print-exprs}, \textit{literal-variables=false}]]$

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 [λ s. getx 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-is-big → 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-is-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 # (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-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-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-greater' x = (v1 > @x)e
```

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

term (*v1-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-less *x* = (*v1 < x*)_e

expr-function *v1-less*

This means, we can parse terms like (*v1-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-less'* *x* = (*v1 < x*)

term (*v1-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* × *st* **is** *v1[>] = v1[<] + 1*

Definitional equations for named expressions are collected in the theorem attribute *v1-less' ?x = (\$v1 < ?x)*_e

v1-is-big = (*100 < \$v1*)_e

inc-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 $'s\ prog = 's \times 's \Rightarrow bool$

definition $seq :: 's\ prog \Rightarrow 's\ prog \Rightarrow 's\ prog$ (**infixr** $::;$ 85) **where**
 $[[expr-defs]]: seq\ P\ Q = (\exists\ s. P[[\langle s \rangle / \mathbf{v}^{\triangleright}]] \wedge Q[[\langle s \rangle / \mathbf{v}^{\triangleleft}]])$ _e

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

definition $assign :: ('a \Rightarrow 's) \Rightarrow ('a, 's)\ expr \Rightarrow 's\ prog$ **where**
 $[[expr-defs]]: assign\ x\ e = (\$x^{\triangleright} = e^{\triangleleft} \wedge \mathbf{v}^{\triangleright} \simeq_{\langle x \rangle} \mathbf{v}^{\triangleleft})$ _e

syntax

$-assign :: svid \Rightarrow logic \Rightarrow logic$ ($- ::= -$ [86, 87] 87)

$-ifthenelse :: logic \Rightarrow logic \Rightarrow logic \Rightarrow logic$ (*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 $mwb-lens\ x\ mwb-lens\ y\ x \bowtie y\ \$y \# (e)$ _e $\$x \# (f)$ _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 # (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 # ("Adam")e by unrest
  show $adam:name # ("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.