Optics in Isabelle/HOL

Simon Foster and Frank Zeyda
University of York, UK

{simon.foster,frank.zeyda}@york.ac.uk

February 23, 2021

Abstract

Lenses provide an abstract interface for manipulating data types through spatially-separated views. They are defined abstractly in terms of two functions, *get*, the return a value from the source type, and *put* that updates the value. We mechanise the underlying theory of lenses, in terms of an algebraic hierarchy of lenses, including well-behaved and very well-behaved lenses, each lens class being characterised by a set of lens laws. We also mechanise a lens algebra in Isabelle that enables their composition and comparison, so as to allow construction of complex lenses. This is accompanied by a large library of algebraic laws. Moreover we also show how the lens classes can be applied by instantiating them with a number of Isabelle data types. This theory development is based on our recent papers [6, 5], which show how lenses can be used to unify heterogeneous representations of state-spaces in formalised programs.

Contents

1 Interpretation Tools **3**
1.1 Interpretation Locale 3

2 Types of Cardinality 2 or Greater **3**

3 Core Lens Laws **4**
3.1 Lens Signature 4
3.2 Weak Lenses 6
3.3 Well-behaved Lenses 7
3.4 Mainly Well-behaved Lenses 7
3.5 Very Well-behaved Lenses 8
3.6 Ineffectual Lenses 9
3.7 Partially Bijective Lenses 9
3.8 Bijective Lenses 10
3.9 Lens Independence 11
3.10 Lens Compatibility 12

4 Lens Algebraic Operators **13**
4.1 Lens Composition, Plus, Unit, and Identity 13
4.2 Closure Poperties 14
4.3 Composition Laws 16
4.4 Independence Laws 17

4.5 Compatibility Laws 18
4.6 Algebraic Laws 19

5 Order and Equivalence on Lenses **20**
5.1 Sub-lens Relation 20
5.2 Lens Equivalence 21
5.3 Further Algebraic Laws 22
5.4 Bijective Lens Equivalences 27
5.5 Lens Override Laws 29
5.6 Alternative Sublens Characterisation 30
5.7 Alternative Equivalence Characterisation 31

6 Symmetric Lenses **31**
6.1 Partial Symmetric Lenses 32
6.2 Symmetric Lenses 32

7 Lens Instances **33**
7.1 Function Lens 33
7.2 Function Range Lens 34
7.3 Map Lens 34
7.4 List Lens 34
7.5 Record Field Lenses 37
7.6 Locale State Spaces 38
7.7 Type Definition Lens 38
7.8 Mapper Lenses 38
7.9 Lens Interpretation 39
7.10 Tactic 39

8 Lenses **39**

9 Prisms **40**
9.1 Signature and Axioms 40
9.2 Co-dependence 40
9.3 Summation 41
9.4 Instances 41
9.5 Lens correspondence 41

10 Channel Types **42**

11 Data spaces **43**

12 Scenes **43**
12.1 Overriding Functions 43
12.2 Scene Type 44
12.3 Linking Scenes and Lenses 49
12.4 Function Domain Scene 51

13 Optics Meta-Theory **52**

14 State and Lens integration **52**

1 Interpretation Tools

```
theory Interp
imports Main
begin
```

1.1 Interpretation Locale

```
locale interp =
fixes f :: 'a ⇒ 'b
assumes f-inj : inj f
begin
lemma meta-interp-law:
(⋀P. PROP Q P) ≡ (⋀P. PROP Q (P o f))
  apply (rule equal-intr-rule)
    — Subgoal 1
   apply (drule-tac x = P o f in meta-spec)
   apply (assumption)
    — Subgoal 2
  apply (drule-tac x = P o inv f in meta-spec)
  apply (simp add: f-inj)
done

lemma all-interp-law:
(∀ P. Q P) = (∀ P. Q (P o f))
  apply (safe)
    — Subgoal 1
   apply (drule-tac x = P o f in spec)
   apply (assumption)
    — Subgoal 2
  apply (drule-tac x = P o inv f in spec)
  apply (simp add: f-inj)
done

lemma exists-interp-law:
(∃ P. Q P) = (∃ P. Q (P o f))
  apply (safe)
    — Subgoal 1
   apply (rule-tac x = P o inv f in exI)
   apply (simp add: f-inj)
    — Subgoal 2
  apply (rule-tac x = P o f in exI)
  apply (assumption)
done
end
end
```

2 Types of Cardinality 2 or Greater

```
theory Two
imports HOL.Real
begin
```

The two class states that a type's carrier is either infinite, or else it has a finite cardinality of at least 2. It is needed when we depend on having at least two distinguishable elements.

```
class two =
  assumes card-two: infinite (UNIV :: 'a set) ∨ card (UNIV :: 'a set) ≥ 2
begin
lemma two-diff: ∃ x y :: 'a. x ≠ y
proof –
  obtain A where finite A card A = 2 A ⊆ (UNIV :: 'a set)
  proof (cases infinite (UNIV :: 'a set))
    case True
    with infinite-arbitrarily-large[of UNIV :: 'a set 2] that
    show ?thesis by auto
  next
    case False
    with card-two that
    show ?thesis
     by (metis UNIV-bool card-UNIV-bool card-image card-le-inj finite.intros(1) finite-insert finite-subset)
  qed
  thus ?thesis
    by (metis (full-types) One-nat-def Suc-1 UNIV-eq-I card.empty card.insert finite.intros(1) insertCI
nat.inject nat.simps(3))
qed
end

instance bool :: two
  by (intro-classes, auto)

instance nat :: two
  by (intro-classes, auto)

instance int :: two
  by (intro-classes, auto simp add: infinite-UNIV-int)

instance rat :: two
  by (intro-classes, auto simp add: infinite-UNIV-char-0)

instance real :: two
  by (intro-classes, auto simp add: infinite-UNIV-char-0)

instance list :: (type) two
  by (intro-classes, auto simp add: infinite-UNIV-listI)

end
```

3 Core Lens Laws

```
theory Lens-Laws
imports
  Two Interp
begin
```

3.1 Lens Signature

This theory introduces the signature of lenses and indentifies the core algebraic hierarchy of lens classes, including laws for well-behaved, very well-behaved, and bijective lenses [4, 2, 8].

```
record ('a, 'b) lens =
```

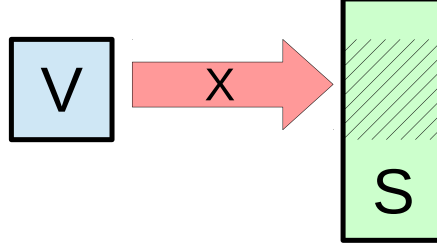

Figure 1: Visualisation of a simple lens

lens-get :: $'b \Rightarrow 'a$ (*get*ı)
lens-put :: $'b \Rightarrow 'a \Rightarrow 'b$ (*put*ı)

type-notation
lens (**infixr** $\Longrightarrow$ *0*)

Alternative parameters ordering, inspired by Back and von Wright's refinement calculus [1], which similarly uses two functions to characterise updates to variables.

abbreviation (*input*) *lens-set* :: $('a \Longrightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ (*lset*ı) **where**
lens-set $\equiv (\lambda\ X\ v\ s.\ put_X\ s\ v)$

A lens $X : V \Longrightarrow S$, for source type S and view type V, identifies V with a subregion of S [4, 3], as illustrated in Figure 1. The arrow denotes X and the hatched area denotes the subregion V it characterises. Transformations on V can be performed without affecting the parts of S outside the hatched area. The lens signature consists of a pair of functions $\mathsf{get}_X : S \Rightarrow V$ that extracts a view from a source, and $\mathsf{put}_X : S \Rightarrow V \Rightarrow S$ that updates a view within a given source.

named-theorems *lens-defs*

lens-source gives the set of constructible sources; that is those that can be built by putting a value into an arbitrary source.

definition *lens-source* :: $('a \Longrightarrow 'b) \Rightarrow 'b\ set$ ($\mathcal{S}$ı) **where**
lens-source $X = \{s.\ \exists\ v\ s'.\ s = put_X\ s'\ v\}$

abbreviation *some-source* :: $('a \Longrightarrow 'b) \Rightarrow 'b$ (*src*ı) **where**
some-source $X \equiv (SOME\ s.\ s \in \mathcal{S}_X)$

definition *lens-create* :: $('a \Longrightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (*create*ı) **where**
[*lens-defs*]: $create_X\ v = put_X\ (src_X)\ v$

Function $create_X\ v$ creates an instance of the source type of X by injecting v as the view, and leaving the remaining context arbitrary.

definition *lens-update* :: $('a \Longrightarrow 'b) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'b)$ (*update*ı) **where**
[*lens-defs*]: *lens-update* $X\ f\ \sigma = put_X\ \sigma\ (f\ (get_X\ \sigma))$

The update function is analogous to the record update function which lifts a function on a view type to one on the source type.

definition *lens-obs-eq* :: $('b \Longrightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\simeq$ı *50*) **where**
[*lens-defs*]: $s_1 \simeq_X s_2 = (s_1 = put_X\ s_2\ (get_X\ s_1))$

This relation states that two sources are equivalent outside of the region characterised by lens X.

definition *lens-override* :: $('b \Longrightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\triangleleft$ *95*) **where**
[*lens-defs*]: $S_1 \triangleleft_X S_2 = put_X\ S_1\ (get_X\ S_2)$

abbreviation (*input*) *lens-override'* :: $'a \Rightarrow 'a \Rightarrow ('b \Longrightarrow 'a) \Rightarrow 'a$ (- $\oplus_L$ - *on* - [*95*,*0*,*96*] *95*) **where**
$S_1 \oplus_L S_2$ *on* $X \equiv S_1 \triangleleft_X S_2$

Lens override uses a lens to replace part of a source type with a given value for the corresponding view.

3.2 Weak Lenses

Weak lenses are the least constrained class of lenses in our algebraic hierarchy. They simply require that the PutGet law [3, 2] is satisfied, meaning that *get* is the inverse of *put*.

locale *weak-lens* =
  fixes $x :: 'a \Longrightarrow 'b$ (**structure**)
  assumes *put-get*: $get\ (put\ \sigma\ v) = v$
begin
  lemma *source-nonempty*: $\exists\ s.\ s \in \mathcal{S}$
    by (*auto simp add*: *lens-source-def*)

  lemma *put-closure*: $put\ \sigma\ v \in \mathcal{S}$
    by (*auto simp add*: *lens-source-def*)

  lemma *create-closure*: $create\ v \in \mathcal{S}$
    by (*simp add*: *lens-create-def put-closure*)

  lemma *src-source* [*simp*]: $src \in \mathcal{S}$
    using *some-in-eq source-nonempty* **by** *auto*

  lemma *create-get*: $get\ (create\ v) = v$
    by (*simp add*: *lens-create-def put-get*)

  lemma *create-inj*: *inj create*
    by (*metis create-get injI*)

  lemma *get-update*: $get\ (update\ f\ \sigma) = f\ (get\ \sigma)$
    by (*simp add*: *put-get lens-update-def*)

  lemma *view-determination*:
    assumes $put\ \sigma\ u = put\ \varrho\ v$
    shows $u = v$
    by (*metis assms put-get*)

  lemma *put-inj*: $inj\ (put\ \sigma)$
    by (*simp add*: *injI view-determination*)

end

declare *weak-lens.put-get* [*simp*]
declare *weak-lens.create-get* [*simp*]

3.3 Well-behaved Lenses

Well-behaved lenses add to weak lenses that requirement that the GetPut law [3, 2] is satisfied, meaning that *put* is the inverse of *get*.

locale *wb-lens* = *weak-lens* +
  assumes *get-put*: *put* σ (*get* σ) = σ
begin

  lemma *put-twice*: *put* (*put* σ *v*) *v* = *put* σ *v*
    by (*metis get-put put-get*)

  lemma *put-surjectivity*: $\exists$ ϱ *v*. *put* ϱ *v* = σ
    using *get-put* **by** *blast*

  lemma *source-stability*: $\exists$ *v*. *put* σ *v* = σ
    using *get-put* **by** *auto*

  lemma *source-UNIV* [*simp*]: $\mathcal{S}$ = *UNIV*
    by (*metis UNIV-eq-I put-closure wb-lens.source-stability wb-lens-axioms*)

end

declare *wb-lens.get-put* [*simp*]

lemma *wb-lens-weak* [*simp*]: *wb-lens x* $\Longrightarrow$ *weak-lens x*
  by (*simp add*: *wb-lens-def*)

3.4 Mainly Well-behaved Lenses

Mainly well-behaved lenses extend weak lenses with the PutPut law that shows how one put override a previous one.

locale *mwb-lens* = *weak-lens* +
  assumes *put-put*: *put* (*put* σ *v*) *u* = *put* σ *u*
begin

  lemma *update-comp*: *update f* (*update g* σ) = *update* ($f \circ g$) σ
    by (*simp add*: *put-get put-put lens-update-def*)

Mainly well-behaved lenses give rise to a weakened version of the *get*−*put* law, where the source must be within the set of constructible sources.

  lemma *weak-get-put*: $\sigma \in \mathcal{S}$ $\Longrightarrow$ *put* σ (*get* σ) = σ
    by (*auto simp add*: *lens-source-def put-get put-put*)

  lemma *weak-source-determination*:
    assumes $\sigma \in \mathcal{S}$ $\varrho \in \mathcal{S}$ *get* σ = *get* ϱ *put* σ *v* = *put* ϱ *v*
    shows $\sigma = \varrho$
    by (*metis assms put-put weak-get-put*)

  lemma *weak-put-eq*:
    assumes $\sigma \in \mathcal{S}$ *get* σ = *k put* σ *u* = *put* ϱ *v*
    shows *put* ϱ *k* = σ
    by (*metis assms put-put weak-get-put*)

Provides *s* is constructible, then *get* can be uniquely determined from *put*

```
  lemma weak-get-via-put: s ∈ 𝒮 ⟹ get s = (THE v. put s v = s)
    by (rule sym, auto intro!: the-equality weak-get-put, metis put-get)

end

abbreviation (input) partial-lens ≡ mwb-lens

declare mwb-lens.put-put [simp]
declare mwb-lens.weak-get-put [simp]

lemma mwb-lens-weak [simp]:
  mwb-lens x ⟹ weak-lens x
  by (simp add: mwb-lens.axioms(1))
```

3.5 Very Well-behaved Lenses

Very well-behaved lenses combine all three laws, as in the literature [3, 2]. The same set of axioms can be found in Back and von Wright's refinement calculus [1], though with different names for the functions.

```
locale vwb-lens = wb-lens + mwb-lens
begin

  lemma source-determination:
    assumes get σ = get ϱ put σ v = put ϱ v
    shows σ = ϱ
    by (metis assms get-put put-put)

 lemma put-eq:
   assumes get σ = k put σ u = put ϱ v
   shows put ϱ k = σ
   using assms weak-put-eq[of σ k u ϱ v] by (simp)
```

get can be uniquely determined from *put*

```
  lemma get-via-put: get s = (THE v. put s v = s)
    by (simp add: weak-get-via-put)

  lemma get-surj: surj get
    by (metis put-get surjI)
```

Observation equivalence is an equivalence relation.

```
  lemma lens-obs-equiv: equivp (≃)
  proof (rule equivpI)
    show reflp (≃)
      by (rule reflpI, simp add: lens-obs-eq-def get-put)
    show symp (≃)
      by (rule sympI, simp add: lens-obs-eq-def, metis get-put put-put)
    show transp (≃)
      by (rule transpI, simp add: lens-obs-eq-def, metis put-put)
  qed

end

abbreviation (input) total-lens ≡ vwb-lens
```

lemma *vwb-lens-wb* [*simp*]: *vwb-lens x* $\Longrightarrow$ *wb-lens x*
by (*simp add*: *vwb-lens-def*)

lemma *vwb-lens-mwb* [*simp*]: *vwb-lens x* $\Longrightarrow$ *mwb-lens x*
using *vwb-lens-def* **by** *auto*

lemma *mwb-UNIV-src-is-vwb-lens*:
$[\![$ *mwb-lens X*; $\mathcal{S}_X$ = *UNIV* $]\!]$ $\Longrightarrow$ *vwb-lens X*
using *vwb-lens-def wb-lens-axioms-def wb-lens-def* **by** *fastforce*

Alternative characterisation: a very well-behaved (i.e. total) lens is a mainly well-behaved (i.e. partial) lens whose source is the universe set.

lemma *vwb-lens-iff-mwb-UNIV-src*:
vwb-lens X $\longleftrightarrow$ (*mwb-lens X* $\wedge$ $\mathcal{S}_X$ = *UNIV*)
by (*meson mwb-UNIV-src-is-vwb-lens vwb-lens-def wb-lens.source-UNIV*)

3.6 Ineffectual Lenses

Ineffectual lenses can have no effect on the view type – application of the *put* function always yields the same source. They are thus, trivially, very well-behaved lenses.

locale *ief-lens* = *weak-lens* +
assumes *put-inef*: *put* σ *v* = σ
begin

sublocale *vwb-lens*
proof
fix σ *v u*
show *put* σ (*get* σ) = σ
by (*simp add*: *put-inef*)
show *put* (*put* σ *v*) *u* = *put* σ *u*
by (*simp add*: *put-inef*)
qed

lemma *ineffectual-const-get*:
$\exists$ *v*. $\forall$ $\sigma \in \mathcal{S}$. *get* σ = *v*
using *put-get put-inef* **by** *auto*

end

abbreviation *eff-lens X* $\equiv$ (*weak-lens X* $\wedge$ ($\neg$ *ief-lens X*))

3.7 Partially Bijective Lenses

locale *pbij-lens* = *weak-lens* +
assumes *put-det*: *put* σ *v* = *put* ϱ *v*
begin

sublocale *mwb-lens*
proof
fix σ *v u*
show *put* (*put* σ *v*) *u* = *put* σ *u*
using *put-det* **by** *blast*
qed

lemma *put-is-create*: *put* σ *v* = *create v*

by (*simp add*: *lens-create-def put-det*)

lemma *partial-get-put*: $\varrho \in \mathcal{S} \Longrightarrow put\ \sigma\ (get\ \varrho) = \varrho$
by (*metis put-det weak-get-put*)

end

lemma *pbij-lens-weak* [*simp*]:
pbij-lens x $\Longrightarrow$ *weak-lens x*
by (*simp-all add*: *pbij-lens-def*)

lemma *pbij-lens-mwb* [*simp*]: *pbij-lens x* $\Longrightarrow$ *mwb-lens x*
by (*simp add*: *mwb-lens-axioms.intro mwb-lens-def pbij-lens.put-is-create*)

lemma *pbij-alt-intro*:
$[\![$ *weak-lens X*; $\bigwedge s.\ s \in \mathcal{S}_X \Longrightarrow create_X\ (get_X\ s) = s$ $]\!] \Longrightarrow$ *pbij-lens X*
by (*metis pbij-lens-axioms-def pbij-lens-def weak-lens.put-closure weak-lens.put-get*)

3.8 Bijective Lenses

Bijective lenses characterise the situation where the source and view type are equivalent: in other words the view type full characterises the whole source type. It is often useful when the view type and source type are syntactically different, but nevertheless correspond precisely in terms of what they observe. Bijective lenses are formulates using the strong GetPut law [3, 2].

locale *bij-lens* = *weak-lens* +
assumes *strong-get-put*: $put\ \sigma\ (get\ \varrho) = \varrho$
begin

sublocale *pbij-lens*
proof
fix $\sigma\ v\ \varrho$
show $put\ \sigma\ v = put\ \varrho\ v$
by (*metis put-get strong-get-put*)
qed

sublocale *vwb-lens*
proof
fix $\sigma\ v\ u$
show $put\ \sigma\ (get\ \sigma) = \sigma$
by (*simp add*: *strong-get-put*)
qed

lemma *put-bij*: *bij-betw* (*put* σ) *UNIV UNIV*
by (*metis bijI put-inj strong-get-put surj-def*)

lemma *get-create*: $create\ (get\ \sigma) = \sigma$
by (*simp add*: *lens-create-def strong-get-put*)

end

declare *bij-lens.strong-get-put* [*simp*]
declare *bij-lens.get-create* [*simp*]

lemma *bij-lens-weak* [*simp*]:
bij-lens x $\Longrightarrow$ *weak-lens x*

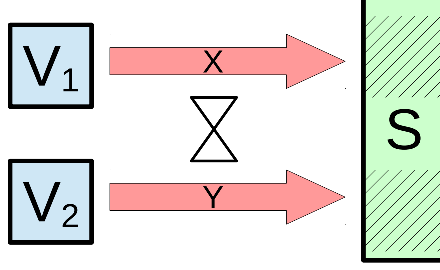

Figure 2: Lens Independence

by (*simp-all add*: *bij-lens-def*)

lemma *bij-lens-pbij* [*simp*]:
bij-lens x $\Longrightarrow$ *pbij-lens x*
by (*metis bij-lens.get-create bij-lens-def pbij-lens-axioms.intro pbij-lens-def weak-lens.put-get*)

lemma *bij-lens-vwb* [*simp*]: *bij-lens x* $\Longrightarrow$ *vwb-lens x*
by (*metis bij-lens.strong-get-put bij-lens-weak mwb-lens.intro mwb-lens-axioms.intro vwb-lens-def wb-lens.intro wb-lens-axioms.intro weak-lens.put-get*)

Alternative characterisation: a bijective lens is a partial bijective lens that is also very well-behaved (i.e. total).

lemma *pbij-vwb-is-bij-lens*:
$[\![$ *pbij-lens X*; *vwb-lens X* $]\!] \Longrightarrow$ *bij-lens X*
by (*unfold-locales*, *simp-all*, *meson pbij-lens.put-det vwb-lens.put-eq*)

lemma *bij-lens-iff-pbij-vwb*:
bij-lens X $\longleftrightarrow$ (*pbij-lens X* $\wedge$ *vwb-lens X*)
using *pbij-vwb-is-bij-lens* **by** *auto*

3.9 Lens Independence

Lens independence shows when two lenses X and Y characterise disjoint regions of the source type, as illustrated in Figure 2. We specify this by requiring that the *put* functions of the two lenses commute, and that the *get* function of each lens is unaffected by application of *put* from the corresponding lens.

locale *lens-indep* =
fixes $X :: 'a \Longrightarrow 'c$ **and** $Y :: 'b \Longrightarrow 'c$
assumes *lens-put-comm*: $put_X\ (put_Y\ \sigma\ v)\ u = put_Y\ (put_X\ \sigma\ u)\ v$
and *lens-put-irr1*: $get_X\ (put_Y\ \sigma\ v) = get_X\ \sigma$
and *lens-put-irr2*: $get_Y\ (put_X\ \sigma\ u) = get_Y\ \sigma$

notation *lens-indep* (**infix** $\bowtie$ *50*)

lemma *lens-indepI*:
$[\![\bigwedge u\ v\ \sigma.\ put_x\ (put_y\ \sigma\ v)\ u = put_y\ (put_x\ \sigma\ u)\ v;$
$\bigwedge v\ \sigma.\ get_x\ (put_y\ \sigma\ v) = get_x\ \sigma;$
$\bigwedge u\ \sigma.\ get_y\ (put_x\ \sigma\ u) = get_y\ \sigma\ ]\!] \Longrightarrow x \bowtie y$
by (*simp add*: *lens-indep-def*)

Lens independence is symmetric.

lemma *lens-indep-sym*: $x \bowtie y \Longrightarrow y \bowtie x$

by (*simp add*: *lens-indep-def*)

lemma *lens-indep-comm*:
$x \bowtie y \Longrightarrow put_x\ (put_y\ \sigma\ v)\ u = put_y\ (put_x\ \sigma\ u)\ v$
by (*simp add*: *lens-indep-def*)

lemma *lens-indep-get* [*simp*]:
assumes $x \bowtie y$
shows $get_x\ (put_y\ \sigma\ v) = get_x\ \sigma$
using *assms lens-indep-def* **by** *fastforce*

Characterisation of independence for two very well-behaved lenses

lemma *lens-indep-vwb-iff*:
assumes *vwb-lens x vwb-lens y*
shows $x \bowtie y \longleftrightarrow (\forall\ u\ v\ \sigma.\ put_x\ (put_y\ \sigma\ v)\ u = put_y\ (put_x\ \sigma\ u)\ v)$
proof
assume $x \bowtie y$
thus $\forall\ u\ v\ \sigma.\ put_x\ (put_y\ \sigma\ v)\ u = put_y\ (put_x\ \sigma\ u)\ v$
by (*simp add*: *lens-indep-comm*)
next
assume *a*: $\forall\ u\ v\ \sigma.\ put_x\ (put_y\ \sigma\ v)\ u = put_y\ (put_x\ \sigma\ u)\ v$
show $x \bowtie y$
proof (*unfold-locales*)
fix $\sigma\ v\ u$
from *a* **show** $put_x\ (put_y\ \sigma\ v)\ u = put_y\ (put_x\ \sigma\ u)\ v$
by *auto*
show $get_x\ (put_y\ \sigma\ v) = get_x\ \sigma$
by (*metis a assms*(*1*) *vwb-lens.put-eq vwb-lens-wb wb-lens-def weak-lens.put-get*)
show $get_y\ (put_x\ \sigma\ u) = get_y\ \sigma$
by (*metis a assms*(*2*) *vwb-lens.put-eq vwb-lens-wb wb-lens-def weak-lens.put-get*)
qed
qed

3.10 Lens Compatibility

Lens compatibility is a weaker notion than independence. It allows that two lenses can overlap so long as they manipulate the source in the same way in that region. It is most easily defined in terms of a function for copying a region from one source to another using a lens.

definition *lens-compat* (**infix** $\#\#_L$ *50*) **where**
[*lens-defs*]: *lens-compat* $X\ Y = (\forall\ s_1\ s_2.\ s_1 \triangleleft_X s_2 \triangleleft_Y s_2 = s_1 \triangleleft_Y s_2 \triangleleft_X s_2)$

lemma *lens-compat-idem* [*simp*]: $x\ \#\#_L\ x$
by (*simp add*: *lens-defs*)

lemma *lens-compat-sym*: $x\ \#\#_L\ y \Longrightarrow y\ \#\#_L\ x$
by (*simp add*: *lens-defs*)

lemma *lens-indep-compat* [*simp*]: $x \bowtie y \Longrightarrow x\ \#\#_L\ y$
by (*simp add*: *lens-override-def lens-compat-def lens-indep-comm*)

end

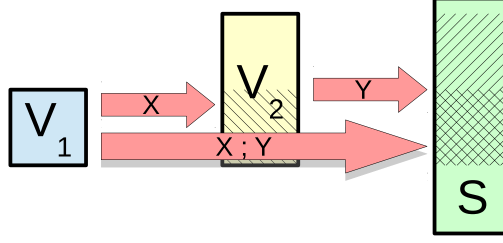

Figure 3: Lens Composition

4 Lens Algebraic Operators

theory *Lens-Algebra*
imports *Lens-Laws*
begin

4.1 Lens Composition, Plus, Unit, and Identity

We introduce the algebraic lens operators; for more information please see our paper [6]. Lens composition, illustrated in Figure 3, constructs a lens by composing the source of one lens with the view of another.

definition $\textit{lens-comp} :: ('a \Longrightarrow 'b) \Rightarrow ('b \Longrightarrow 'c) \Rightarrow ('a \Longrightarrow 'c)$ (**infixl** $;_L$ *80*) **where**
$[\textit{lens-defs}]$: $\textit{lens-comp}\ Y\ X = (\!|\ \textit{lens-get} = \textit{get}_Y \circ \textit{lens-get}\ X$
$,\ \textit{lens-put} = (\lambda\ \sigma\ v.\ \textit{lens-put}\ X\ \sigma\ (\textit{lens-put}\ Y\ (\textit{lens-get}\ X\ \sigma)\ v))\ |\!)$

Lens plus, as illustrated in Figure 4 parallel composes two independent lenses, resulting in a lens whose view is the product of the two underlying lens views.

definition $\textit{lens-plus} :: ('a \Longrightarrow 'c) \Rightarrow ('b \Longrightarrow 'c) \Rightarrow 'a \times 'b \Longrightarrow 'c$ (**infixr** $+_L$ *75*) **where**
$[\textit{lens-defs}]$: $X +_L Y = (\!|\ \textit{lens-get} = (\lambda\ \sigma.\ (\textit{lens-get}\ X\ \sigma,\ \textit{lens-get}\ Y\ \sigma))$
$,\ \textit{lens-put} = (\lambda\ \sigma\ (u,\ v).\ \textit{lens-put}\ X\ (\textit{lens-put}\ Y\ \sigma\ v)\ u)\ |\!)$

The product functor lens similarly parallel composes two lenses, but in this case the lenses have different sources and so the resulting source is also a product.

definition $\textit{lens-prod} :: ('a \Longrightarrow 'c) \Rightarrow ('b \Longrightarrow 'd) \Rightarrow ('a \times 'b \Longrightarrow 'c \times 'd)$ (**infixr** $\times_L$ *85*) **where**
$[\textit{lens-defs}]$: $\textit{lens-prod}\ X\ Y = (\!|\ \textit{lens-get} = \textit{map-prod}\ \textit{get}_X\ \textit{get}_Y$
$,\ \textit{lens-put} = \lambda\ (u,\ v)\ (x,\ y).\ (\textit{put}_X\ u\ x,\ \textit{put}_Y\ v\ y)\ |\!)$

The ***fst*** and ***snd*** lenses project the first and second elements, respectively, of a product source type.

definition $\textit{fst-lens} :: 'a \Longrightarrow 'a \times 'b\ (\textit{fst}_L)$ **where**
$[\textit{lens-defs}]$: $\textit{fst}_L = (\!|\ \textit{lens-get} = \textit{fst},\ \textit{lens-put} = (\lambda\ (\sigma,\ \varrho)\ u.\ (u,\ \varrho))\ |\!)$

definition $\textit{snd-lens} :: 'b \Longrightarrow 'a \times 'b\ (\textit{snd}_L)$ **where**
$[\textit{lens-defs}]$: $\textit{snd}_L = (\!|\ \textit{lens-get} = \textit{snd},\ \textit{lens-put} = (\lambda\ (\sigma,\ \varrho)\ u.\ (\sigma,\ u))\ |\!)$

lemma *get-fst-lens* [*simp*]: $\textit{get}_{\textit{fst}_L}\ (x,\ y) = x$
by (*simp add*: *fst-lens-def*)

lemma *get-snd-lens* [*simp*]: $\textit{get}_{\textit{snd}_L}\ (x,\ y) = y$
by (*simp add*: *snd-lens-def*)

The swap lens is a bijective lens which swaps over the elements of the product source type.

abbreviation $\textit{swap-lens} :: 'a \times 'b \Longrightarrow 'b \times 'a\ (\textit{swap}_L)$ **where**

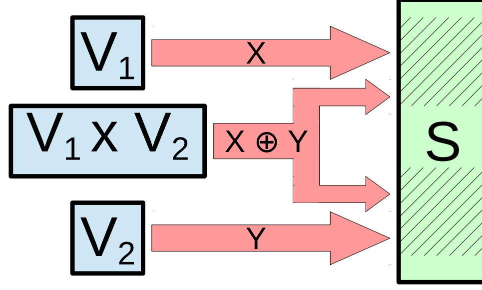

Figure 4: Lens Sum

$swap_L \equiv snd_L +_L fst_L$

The zero lens is an ineffectual lens whose view is a unit type. This means the zero lens cannot distinguish or change the source type.

definition *zero-lens* :: *unit* $\Longrightarrow$ *'a* (0_L) **where**
[*lens-defs*]: 0_L = ⦇ *lens-get* = (λ -. ()), *lens-put* = (λ σ x. σ) ⦈

The identity lens is a bijective lens where the source and view type are the same.

definition *id-lens* :: *'a* $\Longrightarrow$ *'a* (1_L) **where**
[*lens-defs*]: 1_L = ⦇ *lens-get* = *id*, *lens-put* = (λ -. *id*) ⦈

The quotient operator $X /_L Y$ shortens lens X by cutting off Y from the end. It is thus the dual of the composition operator.

definition *lens-quotient* :: (*'a* $\Longrightarrow$ *'c*) ⇒ (*'b* $\Longrightarrow$ *'c*) ⇒ *'a* $\Longrightarrow$ *'b* (**infixr** *'*$/_L$ *90*) **where**
[*lens-defs*]: $X /_L Y$ = ⦇ *lens-get* = λ σ. get_X ($create_Y$ σ)
, *lens-put* = λ σ v. get_Y (put_X ($create_Y$ σ) v) ⦈

Lens inverse take a bijective lens and swaps the source and view types.

definition *lens-inv* :: (*'a* $\Longrightarrow$ *'b*) ⇒ (*'b* $\Longrightarrow$ *'a*) (inv_L) **where**
[*lens-defs*]: *lens-inv x* = ⦇ *lens-get* = $create_x$, *lens-put* = λ σ. get_x ⦈

4.2 Closure Poperties

We show that the core lenses combinators defined above are closed under the key lens classes.

lemma *id-wb-lens*: *wb-lens* 1_L
by (*unfold-locales*, *simp-all add*: *id-lens-def*)

lemma *source-id-lens*: $\mathcal{S}_{1_L}$ = *UNIV*
by (*simp add*: *id-lens-def lens-source-def*)

lemma *unit-wb-lens*: *wb-lens* 0_L
by (*unfold-locales*, *simp-all add*: *zero-lens-def*)

lemma *source-zero-lens*: $\mathcal{S}_{0_L}$ = *UNIV*
by (*simp-all add*: *zero-lens-def lens-source-def*)

lemma *comp-weak-lens*: ⟦ *weak-lens x*; *weak-lens y* ⟧ $\Longrightarrow$ *weak-lens* ($x ;_L y$)
by (*unfold-locales*, *simp-all add*: *lens-comp-def*)

lemma *comp-wb-lens*: ⟦ *wb-lens x*; *wb-lens y* ⟧ $\Longrightarrow$ *wb-lens* ($x ;_L y$)
by (*unfold-locales*, *auto simp add*: *lens-comp-def wb-lens-def weak-lens.put-closure*)

lemma *comp-mwb-lens*: ⟦ *mwb-lens x*; *mwb-lens y* ⟧ ⟹ *mwb-lens* $(x ;_L y)$
by (*unfold-locales*, *auto simp add*: *lens-comp-def mwb-lens-def weak-lens.put-closure*)

lemma *source-lens-comp*: ⟦ *mwb-lens x*; *mwb-lens y* ⟧ ⟹ $\mathcal{S}_{x ;_L y} = \{s \in \mathcal{S}_y.\ get_y\ s \in \mathcal{S}_x\}$
by (*auto simp add*: *lens-comp-def lens-source-def*, *blast*, *metis mwb-lens.put-put mwb-lens-def weak-lens.put-get*)

lemma *id-vwb-lens* [*simp*]: *vwb-lens* 1_L
by (*unfold-locales*, *simp-all add*: *id-lens-def*)

lemma *unit-vwb-lens* [*simp*]: *vwb-lens* 0_L
by (*unfold-locales*, *simp-all add*: *zero-lens-def*)

lemma *comp-vwb-lens*: ⟦ *vwb-lens x*; *vwb-lens y* ⟧ ⟹ *vwb-lens* $(x ;_L y)$
by (*unfold-locales*, *simp-all add*: *lens-comp-def weak-lens.put-closure*)

lemma *unit-ief-lens*: *ief-lens* 0_L
by (*unfold-locales*, *simp-all add*: *zero-lens-def*)

Lens plus requires that the lenses be independent to show closure.

lemma *plus-mwb-lens*:
assumes *mwb-lens x mwb-lens y* $x \bowtie y$
shows *mwb-lens* $(x +_L y)$
using *assms*
apply (*unfold-locales*)
apply (*simp-all add*: *lens-plus-def prod.case-eq-if lens-indep-sym*)
apply (*simp add*: *lens-indep-comm*)
done

lemma *plus-wb-lens*:
assumes *wb-lens x wb-lens y* $x \bowtie y$
shows *wb-lens* $(x +_L y)$
using *assms*
apply (*unfold-locales*, *simp-all add*: *lens-plus-def*)
apply (*simp add*: *lens-indep-sym prod.case-eq-if*)
done

lemma *plus-vwb-lens* [*simp*]:
assumes *vwb-lens x vwb-lens y* $x \bowtie y$
shows *vwb-lens* $(x +_L y)$
using *assms*
apply (*unfold-locales*, *simp-all add*: *lens-plus-def*)
apply (*simp add*: *lens-indep-sym prod.case-eq-if*)
apply (*simp add*: *lens-indep-comm prod.case-eq-if*)
done

lemma *source-plus-lens*:
assumes *mwb-lens x mwb-lens y* $x \bowtie y$
shows $\mathcal{S}_{x +_L y} = \mathcal{S}_x \cap \mathcal{S}_y$
apply (*auto simp add*: *lens-source-def lens-plus-def*)
apply (*meson assms*(*3*) *lens-indep-comm*)
apply (*metis assms*(*1*) *mwb-lens.weak-get-put mwb-lens-weak weak-lens.put-closure*)
done

lemma *prod-mwb-lens*:

$[\![$ *mwb-lens X*; *mwb-lens Y* $]\!] \Longrightarrow$ *mwb-lens* $(X \times_L Y)$
by (*unfold-locales*, *simp-all add*: *lens-prod-def prod.case-eq-if*)

lemma *prod-wb-lens*:
$[\![$ *wb-lens X*; *wb-lens Y* $]\!] \Longrightarrow$ *wb-lens* $(X \times_L Y)$
by (*unfold-locales*, *simp-all add*: *lens-prod-def prod.case-eq-if*)

lemma *prod-vwb-lens*:
$[\![$ *vwb-lens X*; *vwb-lens Y* $]\!] \Longrightarrow$ *vwb-lens* $(X \times_L Y)$
by (*unfold-locales*, *simp-all add*: *lens-prod-def prod.case-eq-if*)

lemma *prod-bij-lens*:
$[\![$ *bij-lens X*; *bij-lens Y* $]\!] \Longrightarrow$ *bij-lens* $(X \times_L Y)$
by (*unfold-locales*, *simp-all add*: *lens-prod-def prod.case-eq-if*)

lemma *fst-vwb-lens*: *vwb-lens* fst_L
by (*unfold-locales*, *simp-all add*: *fst-lens-def prod.case-eq-if*)

lemma *snd-vwb-lens*: *vwb-lens* snd_L
by (*unfold-locales*, *simp-all add*: *snd-lens-def prod.case-eq-if*)

lemma *id-bij-lens*: *bij-lens* 1_L
by (*unfold-locales*, *simp-all add*: *id-lens-def*)

lemma *inv-id-lens*: $inv_L\ 1_L = 1_L$
by (*auto simp add*: *lens-inv-def id-lens-def lens-create-def*)

lemma *inv-inv-lens*: *bij-lens X* $\Longrightarrow inv_L\ (inv_L\ X) = X$
apply (*cases X*)
apply (*auto simp add*: *lens-defs fun-eq-iff*)
apply (*metis* (*no-types*) *bij-lens.strong-get-put bij-lens-def select-convs*(*2*) *weak-lens.put-get*)
done

lemma *lens-inv-bij*: *bij-lens X* $\Longrightarrow$ *bij-lens* $(inv_L\ X)$
by (*unfold-locales*, *simp-all add*: *lens-inv-def lens-create-def*)

lemma *swap-bij-lens*: *bij-lens* $swap_L$
by (*unfold-locales*, *simp-all add*: *lens-plus-def prod.case-eq-if fst-lens-def snd-lens-def*)

4.3 Composition Laws

Lens composition is monoidal, with unit 1_L, as the following theorems demonstrate. It also has 0_L as a right annihilator.

lemma *lens-comp-assoc*: $X ;_L (Y ;_L Z) = (X ;_L Y) ;_L Z$
by (*auto simp add*: *lens-comp-def*)

lemma *lens-comp-left-id* [*simp*]: $1_L ;_L X = X$
by (*simp add*: *id-lens-def lens-comp-def*)

lemma *lens-comp-right-id* [*simp*]: $X ;_L 1_L = X$
by (*simp add*: *id-lens-def lens-comp-def*)

lemma *lens-comp-anhil* [*simp*]: *wb-lens X* $\Longrightarrow 0_L ;_L X = 0_L$
by (*simp add*: *zero-lens-def lens-comp-def comp-def*)

lemma *lens-comp-anhil-right* [*simp*]: *wb-lens* $X \Longrightarrow X ;_L 0_L = 0_L$
  by (*simp add*: *zero-lens-def lens-comp-def comp-def*)

4.4 Independence Laws

The zero lens 0_L is independent of any lens. This is because nothing can be observed or changed using 0_L.

lemma *zero-lens-indep* [*simp*]: $0_L \bowtie X$
  by (*auto simp add*: *zero-lens-def lens-indep-def*)

lemma *zero-lens-indep′* [*simp*]: $X \bowtie 0_L$
  by (*auto simp add*: *zero-lens-def lens-indep-def*)

Lens independence is irreflexive, but only for effectual lenses as otherwise nothing can be observed.

lemma *lens-indep-quasi-irrefl*: ⟦ *wb-lens x*; *eff-lens x* ⟧ $\Longrightarrow \neg (x \bowtie x)$
  by (*auto simp add*: *lens-indep-def ief-lens-def ief-lens-axioms-def*, *metis* (*full-types*) *wb-lens.get-put*)

Lens independence is a congruence with respect to composition, as the following properties demonstrate.

lemma *lens-indep-left-comp* [*simp*]:
  ⟦ *mwb-lens z*; $x \bowtie y$ ⟧ $\Longrightarrow (x ;_L z) \bowtie (y ;_L z)$
  apply (*rule lens-indepI*)
    apply (*auto simp add*: *lens-comp-def*)
   apply (*simp add*: *lens-indep-comm*)
  apply (*simp add*: *lens-indep-sym*)
done

lemma *lens-indep-right-comp*:
  $y \bowtie z \Longrightarrow (x ;_L y) \bowtie (x ;_L z)$
  apply (*auto intro*!: *lens-indepI simp add*: *lens-comp-def*)
    using *lens-indep-comm lens-indep-sym* **apply** *fastforce*
  apply (*simp add*: *lens-indep-sym*)
done

lemma *lens-indep-left-ext* [*intro*]:
  $y \bowtie z \Longrightarrow (x ;_L y) \bowtie z$
  apply (*auto intro*!: *lens-indepI simp add*: *lens-comp-def*)
   apply (*simp add*: *lens-indep-comm*)
  apply (*simp add*: *lens-indep-sym*)
done

lemma *lens-indep-right-ext* [*intro*]:
  $x \bowtie z \Longrightarrow x \bowtie (y ;_L z)$
  by (*simp add*: *lens-indep-left-ext lens-indep-sym*)

lemma *lens-comp-indep-cong-left*:
  ⟦ *mwb-lens Z*; $X ;_L Z \bowtie Y ;_L Z$ ⟧ $\Longrightarrow X \bowtie Y$
  apply (*rule lens-indepI*)
    apply (*rename-tac u v* σ)
    apply (*drule-tac* u=u **and** v=v **and** σ=*create*$_Z$ σ **in** *lens-indep-comm*)
    apply (*simp add*: *lens-comp-def*)
    apply (*meson mwb-lens-weak weak-lens.view-determination*)
   apply (*rename-tac v* σ)

apply (*drule-tac v=v* **and** σ=*create*$_Z$ σ **in** *lens-indep-get*)
apply (*simp add*: *lens-comp-def*)
apply (*drule lens-indep-sym*)
apply (*rename-tac u* σ)
apply (*drule-tac v=u* **and** σ=*create*$_Z$ σ **in** *lens-indep-get*)
apply (*simp add*: *lens-comp-def*)
done

lemma *lens-comp-indep-cong*:
mwb-lens $Z \Longrightarrow (X ;_L Z) \bowtie (Y ;_L Z) \longleftrightarrow X \bowtie Y$
using *lens-comp-indep-cong-left lens-indep-left-comp* **by** *blast*

The first and second lenses are independent since the view different parts of a product source.

lemma *fst-snd-lens-indep* [*simp*]:
$fst_L \bowtie snd_L$
by (*simp add*: *lens-indep-def fst-lens-def snd-lens-def*)

lemma *snd-fst-lens-indep* [*simp*]:
$snd_L \bowtie fst_L$
by (*simp add*: *lens-indep-def fst-lens-def snd-lens-def*)

lemma *split-prod-lens-indep*:
assumes *mwb-lens X*
shows $(fst_L ;_L X) \bowtie (snd_L ;_L X)$
using *assms fst-snd-lens-indep lens-indep-left-comp vwb-lens-mwb* **by** *blast*

Lens independence is preserved by summation.

lemma *plus-pres-lens-indep* [*simp*]: $[\![X \bowtie Z; Y \bowtie Z]\!] \Longrightarrow (X +_L Y) \bowtie Z$
apply (*rule lens-indepI*)
apply (*simp-all add*: *lens-plus-def prod.case-eq-if*)
apply (*simp add*: *lens-indep-comm*)
apply (*simp add*: *lens-indep-sym*)
done

lemma *plus-pres-lens-indep'* [*simp*]:
$[\![X \bowtie Y; X \bowtie Z]\!] \Longrightarrow X \bowtie Y +_L Z$
by (*auto intro*: *lens-indep-sym plus-pres-lens-indep*)

Lens independence is preserved by product.

lemma *lens-indep-prod*:
$[\![X_1 \bowtie X_2; Y_1 \bowtie Y_2]\!] \Longrightarrow X_1 \times_L Y_1 \bowtie X_2 \times_L Y_2$
apply (*rule lens-indepI*)
apply (*auto simp add*: *lens-prod-def prod.case-eq-if lens-indep-comm map-prod-def*)
apply (*simp-all add*: *lens-indep-sym*)
done

4.5 Compatibility Laws

lemma *zero-lens-compat* [*simp*]: $0_L \#\#_L X$
by (*auto simp add*: *zero-lens-def lens-override-def lens-compat-def*)

lemma *id-lens-compat* [*simp*]: *vwb-lens* $X \Longrightarrow 1_L \#\#_L X$
by (*auto simp add*: *id-lens-def lens-override-def lens-compat-def*)

4.6 Algebraic Laws

Lens plus distributes to the right through composition.

lemma *plus-lens-distr*: *mwb-lens* $Z \Longrightarrow (X +_L Y) ;_L Z = (X ;_L Z) +_L (Y ;_L Z)$
by (*auto simp add*: *lens-comp-def lens-plus-def comp-def*)

The first lens projects the first part of a summation.

lemma *fst-lens-plus*:
wb-lens $y \Longrightarrow fst_L ;_L (x +_L y) = x$
by (*simp add*: *fst-lens-def lens-plus-def lens-comp-def comp-def*)

The second law requires independence as we have to apply x first, before y

lemma *snd-lens-plus*:
$[\![$ *wb-lens* x; $x \bowtie y$ $]\!] \Longrightarrow snd_L ;_L (x +_L y) = y$
apply (*simp add*: *snd-lens-def lens-plus-def lens-comp-def comp-def*)
apply (*subst lens-indep-comm*)
apply (*simp-all*)
done

The swap lens switches over a summation.

lemma *lens-plus-swap*:
$X \bowtie Y \Longrightarrow swap_L ;_L (X +_L Y) = (Y +_L X)$
by (*auto simp add*: *lens-plus-def fst-lens-def snd-lens-def id-lens-def lens-comp-def lens-indep-comm*)

The first, second, and swap lenses are all closely related.

lemma *fst-snd-id-lens*: $fst_L +_L snd_L = 1_L$
by (*auto simp add*: *lens-plus-def fst-lens-def snd-lens-def id-lens-def*)

lemma *swap-lens-idem*: $swap_L ;_L swap_L = 1_L$
by (*simp add*: *fst-snd-id-lens lens-indep-sym lens-plus-swap*)

lemma *swap-lens-fst*: $fst_L ;_L swap_L = snd_L$
by (*simp add*: *fst-lens-plus fst-vwb-lens*)

lemma *swap-lens-snd*: $snd_L ;_L swap_L = fst_L$
by (*simp add*: *lens-indep-sym snd-lens-plus snd-vwb-lens*)

The product lens can be rewritten as a sum lens.

lemma *prod-as-plus*: $X \times_L Y = X ;_L fst_L +_L Y ;_L snd_L$
by (*auto simp add*: *lens-prod-def fst-lens-def snd-lens-def lens-comp-def lens-plus-def*)

lemma *prod-lens-id-equiv*:
$1_L \times_L 1_L = 1_L$
by (*auto simp add*: *lens-prod-def id-lens-def*)

lemma *prod-lens-comp-plus*:
$X_2 \bowtie Y_2 \Longrightarrow ((X_1 \times_L Y_1) ;_L (X_2 +_L Y_2)) = (X_1 ;_L X_2) +_L (Y_1 ;_L Y_2)$
by (*auto simp add*: *lens-comp-def lens-plus-def lens-prod-def prod.case-eq-if fun-eq-iff*)

The following laws about quotient are similar to their arithmetic analogues. Lens quotient reverse the effect of a composition.

lemma *lens-comp-quotient*:
weak-lens $Y \Longrightarrow (X ;_L Y) /_L Y = X$
by (*simp add*: *lens-quotient-def lens-comp-def*)

lemma *lens-quotient-id* [*simp*]: *weak-lens* $X \Longrightarrow (X /_L X) = 1_L$
  by (*force simp add*: *lens-quotient-def id-lens-def*)

lemma *lens-quotient-id-denom*: $X /_L 1_L = X$
  by (*simp add*: *lens-quotient-def id-lens-def lens-create-def*)

lemma *lens-quotient-unit*: *weak-lens* $X \Longrightarrow (0_L /_L X) = 0_L$
  by (*simp add*: *lens-quotient-def zero-lens-def*)

lemma *lens-obs-eq-zero*: $s_1 \simeq_{0_L} s_2 = (s_1 = s_2)$
  by (*simp add*: *lens-defs*)

lemma *lens-obs-eq-one*: $s_1 \simeq_{1_L} s_2$
  by (*simp add*: *lens-defs*)

lemma *lens-obs-eq-as-override*: *vwb-lens* $X \Longrightarrow s_1 \simeq_X s_2 \longleftrightarrow (s_2 = s_1 \oplus_L s_2 \text{ on } X)$
  by (*auto simp add*: *lens-defs*; *metis vwb-lens.put-eq*)

end

5 Order and Equivalence on Lenses

theory *Lens-Order*
imports *Lens-Algebra*
begin

5.1 Sub-lens Relation

A lens X is a sub-lens of Y if there is a well-behaved lens Z such that $X = Z ;_L Y$, or in other words if X can be expressed purely in terms of Y.

definition *sublens* :: $('a \Longrightarrow 'c) \Rightarrow ('b \Longrightarrow 'c) \Rightarrow bool$ (**infix** $\subseteq_L$ *55*) **where**
[*lens-defs*]: *sublens* $X\ Y = (\exists\ Z :: ('a, 'b)\ lens.\ vwb\text{-}lens\ Z \wedge X = Z ;_L Y)$

Various lens classes are downward closed under the sublens relation.

lemma *sublens-pres-mwb*:
  $[\![$ *mwb-lens* Y; $X \subseteq_L Y$ $]\!] \Longrightarrow$ *mwb-lens* X
  by (*unfold-locales*, *auto simp add*: *sublens-def lens-comp-def*)

lemma *sublens-pres-wb*:
  $[\![$ *wb-lens* Y; $X \subseteq_L Y$ $]\!] \Longrightarrow$ *wb-lens* X
  by (*unfold-locales*, *auto simp add*: *sublens-def lens-comp-def*)

lemma *sublens-pres-vwb*:
  $[\![$ *vwb-lens* Y; $X \subseteq_L Y$ $]\!] \Longrightarrow$ *vwb-lens* X
  by (*unfold-locales*, *auto simp add*: *sublens-def lens-comp-def*)

Sublens is a preorder as the following two theorems show.

lemma *sublens-refl* [*simp*]:
  $X \subseteq_L X$
  using *id-vwb-lens sublens-def* **by** *fastforce*

lemma *sublens-trans* [*trans*]:
  $[\![X \subseteq_L Y;\ Y \subseteq_L Z]\!] \Longrightarrow X \subseteq_L Z$

apply (*auto simp add*: *sublens-def lens-comp-assoc*)
apply (*rename-tac* Z_1 Z_2)
apply (*rule-tac* x=Z_1 ;$_L$ Z_2 **in** *exI*)
apply (*simp add*: *lens-comp-assoc*)
using *comp-vwb-lens* **apply** *blast*
done

Sublens has a least element – 0_L – and a greatest element – 1_L. Intuitively, this shows that sublens orders how large a portion of the source type a particular lens views, with 0_L observing the least, and 1_L observing the most.

lemma *sublens-least*: *wb-lens* $X \Longrightarrow 0_L \subseteq_L X$
using *sublens-def unit-vwb-lens* **by** *fastforce*

lemma *sublens-greatest*: *vwb-lens* $X \Longrightarrow X \subseteq_L 1_L$
by (*simp add*: *sublens-def*)

If Y is a sublens of X then any put using X will necessarily erase any put using Y. Similarly, any two source types are observationally equivalent by Y when performed following a put using X.

lemma *sublens-put-put*:
$\llbracket$ *mwb-lens* X; $Y \subseteq_L X$ $\rrbracket \Longrightarrow put_X\ (put_Y\ \sigma\ v)\ u = put_X\ \sigma\ u$
by (*auto simp add*: *sublens-def lens-comp-def*)

lemma *sublens-obs-get*:
$\llbracket$ *mwb-lens* X; $Y \subseteq_L X$ $\rrbracket \Longrightarrow get_Y\ (put_X\ \sigma\ v) = get_Y\ (put_X\ \varrho\ v)$
by (*auto simp add*: *sublens-def lens-comp-def*)

Sublens preserves independence; in other words if Y is independent of Z, then also any X smaller than Y is independent of Z.

lemma *sublens-pres-indep*:
$\llbracket X \subseteq_L Y;\ Y \bowtie Z \rrbracket \Longrightarrow X \bowtie Z$
apply (*auto intro*!:*lens-indepI simp add*: *sublens-def lens-comp-def lens-indep-comm*)
apply (*simp add*: *lens-indep-sym*)
done

lemma *sublens-pres-indep'*:
$\llbracket X \subseteq_L Y;\ Z \bowtie Y \rrbracket \Longrightarrow Z \bowtie X$
by (*meson lens-indep-sym sublens-pres-indep*)

lemma *sublens-compat*: $\llbracket$ *vwb-lens* X; *vwb-lens* Y; $X \subseteq_L Y$ $\rrbracket \Longrightarrow X\ \#\#_L\ Y$
unfolding *lens-compat-def lens-override-def*
by (*metis* (*no-types*, *hide-lams*) *sublens-obs-get sublens-put-put vwb-lens-mwb vwb-lens-wb wb-lens.get-put*)

Well-behavedness of lens quotient has sublens as a proviso. This is because we can only remove X from Y if X is smaller than Y.

lemma *lens-quotient-mwb*:
$\llbracket$ *mwb-lens* Y; $X \subseteq_L Y$ $\rrbracket \Longrightarrow$ *mwb-lens* $(X /_L Y)$
by (*unfold-locales*, *auto simp add*: *lens-quotient-def lens-create-def sublens-def lens-comp-def comp-def*)

5.2 Lens Equivalence

Using our preorder, we can also derive an equivalence on lenses as follows. It should be noted that this equality, like sublens, is heterogeneously typed – it can compare lenses whose view

types are different, so long as the source types are the same. We show that it is reflexive, symmetric, and transitive.

definition *lens-equiv* :: $('a \Longrightarrow 'c) \Rightarrow ('b \Longrightarrow 'c) \Rightarrow bool$ (**infix** $\approx_L$ *51*) **where**
[*lens-defs*]: *lens-equiv* $X\ Y = (X \subseteq_L Y \wedge Y \subseteq_L X)$

lemma *lens-equivI* [*intro*]:
  $[\![X \subseteq_L Y; Y \subseteq_L X]\!] \Longrightarrow X \approx_L Y$
  by (*simp add*: *lens-equiv-def*)

lemma *lens-equiv-refl*:
  $X \approx_L X$
  by (*simp add*: *lens-equiv-def*)

lemma *lens-equiv-sym*:
  $X \approx_L Y \Longrightarrow Y \approx_L X$
  by (*simp add*: *lens-equiv-def*)

lemma *lens-equiv-trans* [*trans*]:
  $[\![X \approx_L Y; Y \approx_L Z]\!] \Longrightarrow X \approx_L Z$
  by (*auto intro*: *sublens-trans simp add*: *lens-equiv-def*)

lemma *lens-equiv-pres-indep*:
  $[\![X \approx_L Y; Y \bowtie Z]\!] \Longrightarrow X \bowtie Z$
  using *lens-equiv-def sublens-pres-indep* **by** *blast*

lemma *lens-equiv-pres-indep′*:
  $[\![X \approx_L Y; Z \bowtie Y]\!] \Longrightarrow Z \bowtie X$
  using *lens-equiv-def sublens-pres-indep′* **by** *blast*

lemma *lens-comp-cong-1*: $X \approx_L Y \Longrightarrow X ;_L Z \approx_L Y ;_L Z$
  unfolding *lens-equiv-def*
  by (*metis* (*no-types*, *lifting*) *lens-comp-assoc sublens-def*)

5.3 Further Algebraic Laws

This law explains the behaviour of lens quotient.

lemma *lens-quotient-comp*:
  $[\![$ *weak-lens* $Y; X \subseteq_L Y]\!] \Longrightarrow (X /_L Y) ;_L Y = X$
  by (*auto simp add*: *lens-quotient-def lens-comp-def comp-def sublens-def*)

Plus distributes through quotient.

lemma *lens-quotient-plus*:
  $[\![$ *mwb-lens* $Z; X \subseteq_L Z; Y \subseteq_L Z]\!] \Longrightarrow (X +_L Y) /_L Z = (X /_L Z) +_L (Y /_L Z)$
  apply (*auto simp add*: *lens-quotient-def lens-plus-def sublens-def lens-comp-def comp-def*)
  apply (*rule ext*)
  apply (*rule ext*)
  apply (*simp add*: *prod.case-eq-if*)
done

Laws for for lens plus on the denominator. These laws allow us to extract compositions of fst_L and snd_L terms.

lemma *lens-quotient-plus-den1*:
  $[\![$ *weak-lens* x; *weak-lens* y; $x \bowtie y]\!] \Longrightarrow x /_L (x +_L y) = fst_L$
  by (*auto simp add*: *lens-defs prod.case-eq-if fun-eq-iff*, *metis* (*lifting*) *lens-indep-def weak-lens.put-get*)

lemma *lens-quotient-plus-den2*: $[\![$ *weak-lens* x; *weak-lens* z; $x \bowtie z$; $y \subseteq_L z$ $]\!] \Longrightarrow y /_L (x +_L z) = (y /_L z) ;_L snd_L$
by (*auto simp add*: *lens-defs prod.case-eq-if fun-eq-iff lens-indep.lens-put-irr2*)

There follows a number of laws relating sublens and summation. Firstly, sublens is preserved by summation.

lemma *plus-pred-sublens*: $[\![$ *mwb-lens* Z; $X \subseteq_L Z$; $Y \subseteq_L Z$; $X \bowtie Y$ $]\!] \Longrightarrow (X +_L Y) \subseteq_L Z$
apply (*auto simp add*: *sublens-def*)
apply (*rename-tac* Z_1 Z_2)
apply (*rule-tac* x=$Z_1 +_L Z_2$ **in** *exI*)
apply (*auto intro*!: *plus-wb-lens*)
apply (*simp add*: *lens-comp-indep-cong-left plus-vwb-lens*)
apply (*simp add*: *plus-lens-distr*)
done

Intuitively, lens plus is associative. However we cannot prove this using HOL equality due to monomorphic typing of this operator. But since sublens and lens equivalence are both heterogeneous we can now prove this in the following three lemmas.

lemma *lens-plus-sub-assoc-1*:
$X +_L Y +_L Z \subseteq_L (X +_L Y) +_L Z$
apply (*simp add*: *sublens-def*)
apply (*rule-tac* x=$(fst_L ;_L fst_L) +_L (snd_L ;_L fst_L) +_L snd_L$ **in** *exI*)
apply (*auto*)
apply (*rule plus-vwb-lens*)
apply (*simp add*: *comp-vwb-lens fst-vwb-lens*)
apply (*rule plus-vwb-lens*)
apply (*simp add*: *comp-vwb-lens fst-vwb-lens snd-vwb-lens*)
apply (*simp add*: *snd-vwb-lens*)
apply (*simp add*: *lens-indep-left-ext*)
apply (*rule lens-indep-sym*)
apply (*rule plus-pres-lens-indep*)
using *fst-snd-lens-indep fst-vwb-lens lens-indep-left-comp lens-indep-sym vwb-lens-mwb* **apply** *blast*
using *fst-snd-lens-indep lens-indep-left-ext lens-indep-sym* **apply** *blast*
apply (*auto simp add*: *lens-plus-def lens-comp-def fst-lens-def snd-lens-def prod.case-eq-if split-beta'*)[*1*]
done

lemma *lens-plus-sub-assoc-2*:
$(X +_L Y) +_L Z \subseteq_L X +_L Y +_L Z$
apply (*simp add*: *sublens-def*)
apply (*rule-tac* x=$(fst_L +_L (fst_L ;_L snd_L)) +_L (snd_L ;_L snd_L)$ **in** *exI*)
apply (*auto*)
apply (*rule plus-vwb-lens*)
apply (*rule plus-vwb-lens*)
apply (*simp add*: *fst-vwb-lens*)
apply (*simp add*: *comp-vwb-lens fst-vwb-lens snd-vwb-lens*)
apply (*rule lens-indep-sym*)
apply (*rule lens-indep-left-ext*)
using *fst-snd-lens-indep lens-indep-sym* **apply** *blast*
apply (*auto intro*: *comp-vwb-lens simp add*: *snd-vwb-lens*)
apply (*rule plus-pres-lens-indep*)
apply (*simp add*: *lens-indep-left-ext lens-indep-sym*)
apply (*simp add*: *snd-vwb-lens*)
apply (*auto simp add*: *lens-plus-def lens-comp-def fst-lens-def snd-lens-def prod.case-eq-if split-beta'*)[*1*]
done

lemma *lens-plus-assoc*:
$(X +_L Y) +_L Z \approx_L X +_L Y +_L Z$
by (*simp add*: *lens-equivI lens-plus-sub-assoc-1 lens-plus-sub-assoc-2*)

We can similarly show that it is commutative.

lemma *lens-plus-sub-comm*: $X \bowtie Y \Longrightarrow X +_L Y \subseteq_L Y +_L X$
apply (*simp add*: *sublens-def*)
apply (*rule-tac x*=$snd_L +_L fst_L$ **in** *exI*)
apply (*auto*)
apply (*simp add*: *fst-vwb-lens lens-indep-sym snd-vwb-lens*)
apply (*simp add*: *lens-indep-sym lens-plus-swap*)
done

lemma *lens-plus-comm*: $X \bowtie Y \Longrightarrow X +_L Y \approx_L Y +_L X$
by (*simp add*: *lens-equivI lens-indep-sym lens-plus-sub-comm*)

Any composite lens is larger than an element of the lens, as demonstrated by the following four laws.

lemma *lens-plus-ub* [*simp*]: *wb-lens* $Y \Longrightarrow X \subseteq_L X +_L Y$
by (*metis fst-lens-plus fst-vwb-lens sublens-def*)

lemma *lens-plus-right-sublens*:
$[\![$ *vwb-lens* Y; $Y \bowtie Z$; $X \subseteq_L Z$ $]\!] \Longrightarrow X \subseteq_L Y +_L Z$
apply (*auto simp add*: *sublens-def*)
apply (*rename-tac* Z')
apply (*rule-tac x*=$Z' ;_L snd_L$ **in** *exI*)
apply (*auto*)
using *comp-vwb-lens snd-vwb-lens* **apply** *blast*
apply (*metis lens-comp-assoc snd-lens-plus vwb-lens-def*)
done

lemma *lens-plus-mono-left*:
$[\![$ $Y \bowtie Z$; $X \subseteq_L Y$ $]\!] \Longrightarrow X +_L Z \subseteq_L Y +_L Z$
apply (*auto simp add*: *sublens-def*)
apply (*rename-tac* Z')
apply (*rule-tac x*=$Z' \times_L 1_L$ **in** *exI*)
apply (*subst prod-lens-comp-plus*)
apply (*simp-all*)
using *id-vwb-lens prod-vwb-lens* **apply** *blast*
done

lemma *lens-plus-mono-right*:
$[\![$ $X \bowtie Z$; $Y \subseteq_L Z$ $]\!] \Longrightarrow X +_L Y \subseteq_L X +_L Z$
by (*metis prod-lens-comp-plus prod-vwb-lens sublens-def sublens-refl*)

If we compose a lens X with lens Y then naturally the resulting lens must be smaller than Y, as X views a part of Y.

lemma *lens-comp-lb* [*simp*]: *vwb-lens* $X \Longrightarrow X ;_L Y \subseteq_L Y$
by (*auto simp add*: *sublens-def*)

lemma *sublens-comp* [*simp*]:
assumes *vwb-lens* b $c \subseteq_L a$
shows $(b ;_L c) \subseteq_L a$
by (*metis assms sublens-def sublens-trans*)

We can now also show that 0_L is the unit of lens plus

lemma *lens-unit-plus-sublens-1*: $X \subseteq_L 0_L +_L X$
  by (*metis lens-comp-lb snd-lens-plus snd-vwb-lens zero-lens-indep unit-wb-lens*)

lemma *lens-unit-prod-sublens-2*: $0_L +_L X \subseteq_L X$
  apply (*auto simp add*: *sublens-def*)
  apply (*rule-tac x*=$0_L +_L 1_L$ **in** *exI*)
  apply (*auto*)
  apply (*auto simp add*: *lens-plus-def zero-lens-def lens-comp-def id-lens-def prod.case-eq-if comp-def*)
  apply (*rule ext*)
  apply (*rule ext*)
  apply (*auto*)
done

lemma *lens-plus-left-unit*: $0_L +_L X \approx_L X$
  by (*simp add*: *lens-equivI lens-unit-plus-sublens-1 lens-unit-prod-sublens-2*)

lemma *lens-plus-right-unit*: $X +_L 0_L \approx_L X$
  using *lens-equiv-trans lens-indep-sym lens-plus-comm lens-plus-left-unit zero-lens-indep* **by** *blast*

We can also show that both sublens and equivalence are congruences with respect to lens plus and lens product.

lemma *lens-plus-subcong*: $[\![Y_1 \bowtie Y_2; X_1 \subseteq_L Y_1; X_2 \subseteq_L Y_2]\!] \Longrightarrow X_1 +_L X_2 \subseteq_L Y_1 +_L Y_2$
  by (*metis prod-lens-comp-plus prod-vwb-lens sublens-def*)

lemma *lens-plus-eq-left*: $[\![X \bowtie Z; X \approx_L Y]\!] \Longrightarrow X +_L Z \approx_L Y +_L Z$
  by (*meson lens-equiv-def lens-plus-mono-left sublens-pres-indep*)

lemma *lens-plus-eq-right*: $[\![X \bowtie Y; Y \approx_L Z]\!] \Longrightarrow X +_L Y \approx_L X +_L Z$
  by (*meson lens-equiv-def lens-indep-sym lens-plus-mono-right sublens-pres-indep*)

lemma *lens-plus-cong*:
  assumes $X_1 \bowtie X_2$ $X_1 \approx_L Y_1$ $X_2 \approx_L Y_2$
  shows $X_1 +_L X_2 \approx_L Y_1 +_L Y_2$
proof −
  have $X_1 +_L X_2 \approx_L Y_1 +_L X_2$
    by (*simp add*: *assms*(*1*) *assms*(*2*) *lens-plus-eq-left*)
  moreover have ... $\approx_L Y_1 +_L Y_2$
    using *assms*(*1*) *assms*(*2*) *assms*(*3*) *lens-equiv-def lens-plus-eq-right sublens-pres-indep* **by** *blast*
  ultimately show *?thesis*
    using *lens-equiv-trans* **by** *blast*
qed

lemma *prod-lens-sublens-cong*:
  $[\![X_1 \subseteq_L X_2; Y_1 \subseteq_L Y_2]\!] \Longrightarrow (X_1 \times_L Y_1) \subseteq_L (X_2 \times_L Y_2)$
  apply (*auto simp add*: *sublens-def*)
  apply (*rename-tac* Z_1 Z_2)
  apply (*rule-tac x*=$Z_1 \times_L Z_2$ **in** *exI*)
  apply (*auto*)
   using *prod-vwb-lens* **apply** *blast*
  apply (*auto simp add*: *lens-prod-def lens-comp-def prod.case-eq-if*)
  apply (*rule ext*, *rule ext*)
  apply (*auto simp add*: *lens-prod-def lens-comp-def prod.case-eq-if*)
done

lemma *prod-lens-equiv-cong*:
  $[\![X_1 \approx_L X_2; Y_1 \approx_L Y_2]\!] \Longrightarrow (X_1 \times_L Y_1) \approx_L (X_2 \times_L Y_2)$
  by (*simp add*: *lens-equiv-def prod-lens-sublens-cong*)

We also have the following "exchange" law that allows us to switch over a lens product and plus.

lemma *lens-plus-prod-exchange*:
  $(X_1 +_L X_2) \times_L (Y_1 +_L Y_2) \approx_L (X_1 \times_L Y_1) +_L (X_2 \times_L Y_2)$
proof (*rule lens-equivI*)
  show $(X_1 +_L X_2) \times_L (Y_1 +_L Y_2) \subseteq_L (X_1 \times_L Y_1) +_L (X_2 \times_L Y_2)$
    apply (*simp add*: *sublens-def*)
    apply (*rule-tac x*=$((fst_L ;_L fst_L) +_L (fst_L ;_L snd_L)) +_L ((snd_L ;_L fst_L) +_L (snd_L ;_L snd_L))$ **in** *exI*)
    apply (*auto*)
     apply (*auto intro*!: *plus-vwb-lens comp-vwb-lens fst-vwb-lens snd-vwb-lens lens-indep-right-comp*)
       apply (*auto intro*!: *lens-indepI simp add*: *lens-comp-def lens-plus-def fst-lens-def snd-lens-def*)
   apply (*auto simp add*: *lens-prod-def lens-plus-def lens-comp-def fst-lens-def snd-lens-def prod.case-eq-if comp-def*)[*1*]
    apply (*rule ext*, *rule ext*, *auto simp add*: *prod.case-eq-if*)
  done
  show $(X_1 \times_L Y_1) +_L (X_2 \times_L Y_2) \subseteq_L (X_1 +_L X_2) \times_L (Y_1 +_L Y_2)$
    apply (*simp add*: *sublens-def*)
    apply (*rule-tac x*=$((fst_L ;_L fst_L) +_L (fst_L ;_L snd_L)) +_L ((snd_L ;_L fst_L) +_L (snd_L ;_L snd_L))$ **in** *exI*)
    apply (*auto*)
     apply (*auto intro*!: *plus-vwb-lens comp-vwb-lens fst-vwb-lens snd-vwb-lens lens-indep-right-comp*)
       apply (*auto intro*!: *lens-indepI simp add*: *lens-comp-def lens-plus-def fst-lens-def snd-lens-def*)
   apply (*auto simp add*: *lens-prod-def lens-plus-def lens-comp-def fst-lens-def snd-lens-def prod.case-eq-if comp-def*)[*1*]
    apply (*rule ext*, *rule ext*, *auto simp add*: *lens-prod-def prod.case-eq-if*)
  done
qed

lemma *lens-get-put-quasi-commute*:
  $[\![$ *vwb-lens* $Y; X \subseteq_L Y]\!] \Longrightarrow get_Y\ (put_X\ s\ v) = put_{X\ /_L\ Y}\ (get_Y\ s)\ v$
proof –
  assume *a1*: *vwb-lens Y*
  assume *a2*: $X \subseteq_L Y$
  have $\bigwedge l\ la.\ put_{l\ ;_L\ la} = (\lambda b\ c.\ put_{la}\ (b::'b)\ (put_l\ (get_{la}\ b::'a)\ (c::'c)))$
    by (*simp add*: *lens-comp-def*)
  then have $\bigwedge l\ la\ b\ c.\ get_l\ (put_{la\ ;_L\ l}\ (b::'b)\ (c::'c)) = put_{la}\ (get_l\ b::'a)\ c \vee \neg$ *weak-lens l*
    by *force*
  then show *?thesis*
    using *a2 a1* **by** (*metis lens-quotient-comp vwb-lens-wb wb-lens-def*)
qed

lemma *lens-put-of-quotient*:
  $[\![$ *vwb-lens* $Y; X \subseteq_L Y]\!] \Longrightarrow put_Y\ s\ (put_{X\ /_L\ Y}\ v_2\ v_1) = put_X\ (put_Y\ s\ v_2)\ v_1$
proof –
  assume *a1*: *vwb-lens Y*
  assume *a2*: $X \subseteq_L Y$
  have *f3*: $\bigwedge l\ b.\ put_l\ (b::'b)\ (get_l\ b::'a) = b \vee \neg$ *vwb-lens l*
    by *force*
  have *f4*: $\bigwedge b\ c.\ put_{X\ /_L\ Y}\ (get_Y\ b)\ c = get_Y\ (put_X\ b\ c)$
    using *a2 a1* **by** (*simp add*: *lens-get-put-quasi-commute*)
  have $\bigwedge b\ c\ a.\ put_Y\ (put_X\ b\ c)\ a = put_Y\ b\ a$
    using *a2 a1* **by** (*simp add*: *sublens-put-put*)

then show *?thesis*
  using *f4 f3 a1* **by** (*metis mwb-lens.put-put mwb-lens-def vwb-lens-mwb weak-lens.put-get*)
qed

5.4 Bijective Lens Equivalences

A bijective lens, like a bijective function, is its own inverse. Thus, if we compose its inverse with itself we get 1_L.

lemma *bij-lens-inv-left*:
  $bij\text{-}lens\ X \Longrightarrow inv_L\ X\ ;_L\ X = 1_L$
  by (*auto simp add*: *lens-inv-def lens-comp-def lens-create-def comp-def id-lens-def*, *rule ext*, *auto*)

lemma *bij-lens-inv-right*:
  $bij\text{-}lens\ X \Longrightarrow X\ ;_L\ inv_L\ X = 1_L$
  by (*auto simp add*: *lens-inv-def lens-comp-def comp-def id-lens-def*, *rule ext*, *auto*)

The following important results shows that bijective lenses are precisely those that are equivalent to identity. In other words, a bijective lens views all of the source type.

lemma *bij-lens-equiv-id*:
  $bij\text{-}lens\ X \longleftrightarrow X \approx_L 1_L$
  apply (*auto*)
   apply (*rule lens-equivI*)
    apply (*simp-all add*: *sublens-def*)
   apply (*rule-tac x=lens-inv X* **in** *exI*)
   apply (*simp add*: *bij-lens-inv-left lens-inv-bij*)
  apply (*auto simp add*: *lens-equiv-def sublens-def id-lens-def lens-comp-def comp-def*)
  apply (*unfold-locales*)
   apply (*simp*)
  apply (*simp*)
  apply (*metis* (*no-types*, *lifting*) *vwb-lens-wb wb-lens-weak weak-lens.put-get*)
done

For this reason, by transitivity, any two bijective lenses with the same source type must be equivalent.

lemma *bij-lens-equiv*:
  $\llbracket\ bij\text{-}lens\ X;\ X \approx_L Y\ \rrbracket \Longrightarrow bij\text{-}lens\ Y$
  by (*meson bij-lens-equiv-id lens-equiv-def sublens-trans*)

lemma *bij-lens-cong*:
  $X \approx_L Y \Longrightarrow bij\text{-}lens\ X = bij\text{-}lens\ Y$
  by (*meson bij-lens-equiv lens-equiv-sym*)

We can also show that the identity lens 1_L is unique. That is to say it is the only lens which when compose with Y will yield Y.

lemma *lens-id-unique*:
  $weak\text{-}lens\ Y \Longrightarrow Y = X\ ;_L\ Y \Longrightarrow X = 1_L$
  apply (*cases Y*)
  apply (*cases X*)
  apply (*auto simp add*: *lens-comp-def comp-def id-lens-def fun-eq-iff*)
   apply (*metis select-convs*(*1*) *weak-lens.create-get*)
  apply (*metis select-convs*(*1*) *select-convs*(*2*) *weak-lens.put-get*)
done

Consequently, if composition of two lenses X and Y yields 1_L, then both of the composed lenses must be bijective.

lemma *bij-lens-via-comp-id-left*:
  $[\![$ *wb-lens* X; *wb-lens* Y; $X ;_L Y = 1_L$ $]\!] \Longrightarrow$ *bij-lens* X
  apply (*cases* Y)
  apply (*cases* X)
  apply (*auto simp add*: *lens-comp-def comp-def id-lens-def fun-eq-iff*)
  apply (*unfold-locales*)
   apply (*simp-all*)
   using *vwb-lens-wb wb-lens-weak weak-lens.put-get* **apply** *fastforce*
  apply (*metis select-convs*(*1*) *select-convs*(*2*) *wb-lens-weak weak-lens.put-get*)
done

lemma *bij-lens-via-comp-id-right*:
  $[\![$ *wb-lens* X; *wb-lens* Y; $X ;_L Y = 1_L$ $]\!] \Longrightarrow$ *bij-lens* Y
  apply (*cases* Y)
  apply (*cases* X)
  apply (*auto simp add*: *lens-comp-def comp-def id-lens-def fun-eq-iff*)
  apply (*unfold-locales*)
   apply (*simp-all*)
   using *vwb-lens-wb wb-lens-weak weak-lens.put-get* **apply** *fastforce*
  apply (*metis select-convs*(*1*) *select-convs*(*2*) *wb-lens-weak weak-lens.put-get*)
done

Importantly, an equivalence between two lenses can be demonstrated by showing that one lens can be converted to the other by application of a suitable bijective lens Z. This Z lens converts the view type of one to the view type of the other.

lemma *lens-equiv-via-bij*:
  assumes *bij-lens* Z $X = Z ;_L Y$
  shows $X \approx_L Y$
  using *assms*
  apply (*auto simp add*: *lens-equiv-def sublens-def*)
   using *bij-lens-vwb* **apply** *blast*
  apply (*rule-tac x=lens-inv Z* **in** *exI*)
  apply (*auto simp add*: *lens-comp-assoc bij-lens-inv-left*)
   using *bij-lens-vwb lens-inv-bij* **apply** *blast*
done

Indeed, we actually have a stronger result than this – the equivalent lenses are precisely those than can be converted to one another through a suitable bijective lens. Bijective lenses can thus be seen as a special class of "adapter" lens.

lemma *lens-equiv-iff-bij*:
  assumes *weak-lens* Y
  shows $X \approx_L Y \longleftrightarrow (\exists\ Z.$ *bij-lens* $Z \wedge X = Z ;_L Y)$
  apply (*rule iffI*)
   apply (*auto simp add*: *lens-equiv-def sublens-def lens-id-unique*)[*1*]
   apply (*rename-tac* Z_1 Z_2)
   apply (*rule-tac* $x{=}Z_1$ **in** *exI*)
   apply (*simp*)
   apply (*subgoal-tac* $Z_2 ;_L Z_1 = 1_L$)
    apply (*meson bij-lens-via-comp-id-right vwb-lens-wb*)
   apply (*metis assms lens-comp-assoc lens-id-unique*)
  using *lens-equiv-via-bij* **apply** *blast*
done

lemma *pbij-plus-commute*:
  $[\![$ $a \bowtie b$; *mwb-lens* a; *mwb-lens* b; *pbij-lens* $(b +_L a)$ $]\!] \Longrightarrow$ *pbij-lens* $(a +_L b)$

apply (*unfold-locales*, *simp-all add*:*lens-defs lens-indep-sym prod.case-eq-if*)
using *lens-indep.lens-put-comm pbij-lens.put-det* **apply** *fastforce*
done

5.5 Lens Override Laws

The following laws are analogus to the equivalent laws for functions.

lemma *lens-override-id* [*simp*]:
$S_1 \oplus_L S_2$ *on* $1_L = S_2$
by (*simp add*: *lens-override-def id-lens-def*)

lemma *lens-override-unit* [*simp*]:
$S_1 \oplus_L S_2$ *on* $0_L = S_1$
by (*simp add*: *lens-override-def zero-lens-def*)

lemma *lens-override-overshadow*:
assumes *mwb-lens* Y $X \subseteq_L Y$
shows $(S_1 \oplus_L S_2$ *on* $X) \oplus_L S_3$ *on* $Y = S_1 \oplus_L S_3$ *on* Y
using *assms* **by** (*simp add*: *lens-override-def sublens-put-put*)

lemma *lens-override-irr*:
assumes $X \bowtie Y$
shows $S_1 \oplus_L (S_2 \oplus_L S_3$ *on* $Y)$ *on* $X = S_1 \oplus_L S_2$ *on* X
using *assms* **by** (*simp add*: *lens-override-def*)

lemma *lens-override-overshadow-left*:
assumes *mwb-lens* X
shows $(S_1 \oplus_L S_2$ *on* $X) \oplus_L S_3$ *on* $X = S_1 \oplus_L S_3$ *on* X
by (*simp add*: *assms lens-override-def*)

lemma *lens-override-overshadow-right*:
assumes *mwb-lens* X
shows $S_1 \oplus_L (S_2 \oplus_L S_3$ *on* $X)$ *on* $X = S_1 \oplus_L S_3$ *on* X
by (*simp add*: *assms lens-override-def*)

lemma *lens-override-plus*:
$X \bowtie Y \Longrightarrow S_1 \oplus_L S_2$ *on* $(X +_L Y) = (S_1 \oplus_L S_2$ *on* $X) \oplus_L S_2$ *on* Y
by (*simp add*: *lens-indep-comm lens-override-def lens-plus-def*)

lemma *lens-override-idem* [*simp*]:
vwb-lens $X \Longrightarrow S \oplus_L S$ *on* $X = S$
by (*simp add*: *lens-override-def*)

lemma *lens-override-mwb-idem* [*simp*]:
$\llbracket$ *mwb-lens* X; $S \in \mathcal{S}_X$ $\rrbracket \Longrightarrow S \oplus_L S$ *on* $X = S$
by (*simp add*: *lens-override-def*)

lemma *lens-override-put-right-in*:
$\llbracket$ *vwb-lens* A; $X \subseteq_L A$ $\rrbracket \Longrightarrow S_1 \oplus_L (put_X\ S_2\ v)$ *on* $A = put_X\ (S_1 \oplus_L S_2$ *on* $A)\ v$
by (*simp add*: *lens-override-def lens-get-put-quasi-commute lens-put-of-quotient*)

lemma *lens-override-put-right-out*:
$\llbracket$ *vwb-lens* A; $X \bowtie A$ $\rrbracket \Longrightarrow S_1 \oplus_L (put_X\ S_2\ v)$ *on* $A = (S_1 \oplus_L S_2$ *on* $A)$
by (*simp add*: *lens-override-def lens-indep.lens-put-irr2*)

lemma *lens-indep-overrideI*:
assumes *vwb-lens X vwb-lens Y* ($\bigwedge$ s_1 s_2 s_3. $s_1 \oplus_L s_2$ *on* $X \oplus_L s_3$ *on* $Y = s_1 \oplus_L s_3$ *on* $Y \oplus_L s_2$ *on* X)
shows $X \bowtie Y$
using *assms*
apply (*unfold-locales*)
apply (*simp-all add*: *lens-override-def*)
apply (*metis mwb-lens-def vwb-lens-mwb weak-lens.put-get*)
apply (*metis lens-override-def lens-override-idem mwb-lens-def vwb-lens-mwb weak-lens.put-get*)
apply (*metis mwb-lens-weak vwb-lens-mwb vwb-lens-wb wb-lens.get-put weak-lens.put-get*)
done

lemma *lens-indep-override-def*:
assumes *vwb-lens X vwb-lens Y*
shows $X \bowtie Y \longleftrightarrow (\forall\ s_1\ s_2\ s_3.\ s_1 \oplus_L s_2$ *on* $X \oplus_L s_3$ *on* $Y = s_1 \oplus_L s_3$ *on* $Y \oplus_L s_2$ *on* X)
by (*metis assms*(*1*) *assms*(*2*) *lens-indep-comm lens-indep-overrideI lens-override-def*)

Alternative characterisation of very-well behaved lenses: override is idempotent.

lemma *override-idem-implies-vwb*:
$[\![$ *mwb-lens X*; $\bigwedge$ *s*. $s \oplus_L s$ *on* $X = s$ $]\!] \Longrightarrow$ *vwb-lens X*
by (*unfold-locales*, *simp-all add*: *lens-defs*)

5.6 Alternative Sublens Characterisation

The following definition is equivalent to the above when the two lenses are very well behaved.

definition *sublens'* :: $('a \Longrightarrow 'c) \Rightarrow ('b \Longrightarrow 'c) \Rightarrow$ *bool* (**infix** $\subseteq_L''$ *55*) **where**
[*lens-defs*]: *sublens'* $X\ Y = (\forall\ s_1\ s_2\ s_3.\ s_1 \oplus_L s_2$ *on* $Y \oplus_L s_3$ *on* $X = s_1 \oplus_L s_2 \oplus_L s_3$ *on X on Y*)

We next prove some characteristic properties of our alternative definition of sublens.

lemma *sublens'-prop1*:
assumes *vwb-lens X* $X \subseteq_L' Y$
shows $put_X\ (put_Y\ s_1\ (get_Y\ s_2))\ s_3 = put_Y\ s_1\ (get_Y\ (put_X\ s_2\ s_3))$
using *assms*
by (*simp add*: *sublens'-def*, *metis lens-override-def mwb-lens-def vwb-lens-mwb weak-lens.put-get*)

lemma *sublens'-prop2*:
assumes *vwb-lens X* $X \subseteq_L' Y$
shows $get_X\ (put_Y\ s_1\ (get_Y\ s_2)) = get_X\ s_2$
using *assms* **unfolding** *sublens'-def*
by (*metis lens-override-def vwb-lens-wb wb-lens-axioms-def wb-lens-def weak-lens.put-get*)

lemma *sublens'-prop3*:
assumes *vwb-lens X vwb-lens Y* $X \subseteq_L' Y$
shows $put_Y\ \sigma\ (get_Y\ (put_X\ (put_Y\ \varrho\ (get_Y\ \sigma))\ v)) = put_X\ \sigma\ v$
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *mwb-lens-def sublens'-prop1 vwb-lens.put-eq vwb-lens-mwb weak-lens.put-get*)

Finally we show our two definitions of sublens are equivalent, assuming very well behaved lenses.

lemma *sublens'-implies-sublens*:
assumes *vwb-lens X vwb-lens Y* $X \subseteq_L' Y$
shows $X \subseteq_L Y$
proof –
have *vwb-lens* $(X /_L Y)$
by (*unfold-locales*

,*auto simp add*: *assms lens-quotient-def lens-comp-def lens-create-def sublens′-prop1 sublens′-prop2*)
  moreover have $X = X /_L Y ;_L Y$
  proof −
    have $get_X = (\lambda\sigma.\ get_X\ (create_Y\ \sigma)) \circ get_Y$
      by (*rule ext*, *simp add*: *assms*(*1*) *assms*(*3*) *lens-create-def sublens′-prop2*)
    moreover have $put_X = (\lambda\sigma\ v.\ put_Y\ \sigma\ (get_Y\ (put_X\ (create_Y\ (get_Y\ \sigma))\ v)))$
      by (*rule ext*, *rule ext*, *simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *lens-create-def sublens′-prop3*)
    ultimately show *?thesis*
      by (*simp add*: *lens-quotient-def lens-comp-def*)
  qed
  ultimately show *?thesis*
    using *sublens-def* **by** *blast*
qed

lemma *sublens-implies-sublens′*:
  assumes *vwb-lens Y* $X \subseteq_L Y$
  shows $X \subseteq_L{}' Y$
  by (*metis assms lens-override-def lens-override-put-right-in sublens′-def*)

lemma *sublens-iff-sublens′*:
  assumes *vwb-lens X vwb-lens Y*
  shows $X \subseteq_L Y \longleftrightarrow X \subseteq_L{}' Y$
  using *assms sublens′-implies-sublens sublens-implies-sublens′* **by** *blast*

5.7 Alternative Equivalence Characterisation

definition *lens-equiv′* :: $('a \Longrightarrow 'c) \Rightarrow ('b \Longrightarrow 'c) \Rightarrow bool$ (**infix** $\approx_L{}''$ *51*) **where**
[*lens-defs*]: *lens-equiv′* $X\ Y = (\forall\ s_1\ s_2.\ (s_1 \oplus_L s_2\ on\ X = s_1 \oplus_L s_2\ on\ Y))$

lemma *lens-equiv-iff-lens-equiv′*:
  assumes *vwb-lens X vwb-lens Y*
  shows $X \approx_L Y \longleftrightarrow X \approx_L{}' Y$
  apply (*simp add*: *lens-equiv-def sublens-iff-sublens′ assms*)
  apply (*auto simp add*: *lens-defs assms*)
  apply (*metis assms*(*2*) *mwb-lens.put-put vwb-lens-mwb vwb-lens-wb wb-lens.get-put*)
  done

end

6 Symmetric Lenses

theory *Lens-Symmetric*
  imports *Lens-Order*
begin

A characterisation of Hofmann's "Symmetric Lenses" [7], where a lens is accompanied by its complement.

record (*′a*, *′b*, *′s*) *slens* =
  view :: $'a \Longrightarrow 's$ ($\mathcal{V}_1$) — The region characterised
  coview :: $'b \Longrightarrow 's$ ($\mathcal{C}_1$) — The complement of the region

type-notation
  slens (<-, -> ⟺ - [*0*, *0*, *0*] *0*)

declare *slens.defs* [*lens-defs*]

definition *slens-compl* :: $(<'a, 'c> \Longleftrightarrow 'b) \Rightarrow <'c, 'a> \Longleftrightarrow 'b$ ($-_L$ - [*81*] *80*) **where**
[*lens-defs*]: *slens-compl a* = ⦇ *view* = *coview a*, *coview* = *view a* ⦈

lemma *view-slens-compl* [*simp*]: $\mathcal{V}_{-_L\ a} = \mathcal{C}_a$
  by (*simp add*: *slens-compl-def*)

lemma *coview-slens-compl* [*simp*]: $\mathcal{C}_{-_L\ a} = \mathcal{V}_a$
  by (*simp add*: *slens-compl-def*)

6.1 Partial Symmetric Lenses

locale *psym-lens* =
  fixes $S :: <'a, 'b> \Longleftrightarrow 's$ (**structure**)
  assumes
    mwb-region [*simp*]: *mwb-lens* $\mathcal{V}$ **and**
    mwb-coregion [*simp*]: *mwb-lens* $\mathcal{C}$ **and**
    indep-region-coregion [*simp*]: $\mathcal{V} \bowtie \mathcal{C}$ **and**
    pbij-region-coregion [*simp*]: *pbij-lens* $(\mathcal{V} +_L \mathcal{C})$

declare *psym-lens.mwb-region* [*simp*]
declare *psym-lens.mwb-coregion* [*simp*]
declare *psym-lens.indep-region-coregion* [*simp*]

lemma *psym-lens-compl* [*simp*]: *psym-lens a* $\Longrightarrow$ *psym-lens* $(-_L\ a)$
  apply (*simp add*: *slens-compl-def*)
  apply (*rule psym-lens.intro*)
  apply (*simp-all*)
  using *lens-indep-sym psym-lens.indep-region-coregion* **apply** *blast*
  using *lens-indep-sym pbij-plus-commute psym-lens-def* **apply** *blast*
  done

6.2 Symmetric Lenses

locale *sym-lens* =
  fixes $S :: <'a, 'b> \Longleftrightarrow 's$ (**structure**)
  assumes
    vwb-region: *vwb-lens* $\mathcal{V}$ **and**
    vwb-coregion: *vwb-lens* $\mathcal{C}$ **and**
    indep-region-coregion: $\mathcal{V} \bowtie \mathcal{C}$ **and**
    bij-region-coregion: *bij-lens* $(\mathcal{V} +_L \mathcal{C})$
begin

sublocale *psym-lens*
proof (*rule psym-lens.intro*)
  show *mwb-lens* $\mathcal{V}$
    by (*simp add*: *vwb-region*)
  show *mwb-lens* $\mathcal{C}$
    by (*simp add*: *vwb-coregion*)
  show $\mathcal{V} \bowtie \mathcal{C}$
    using *indep-region-coregion* **by** *blast*
  show *pbij-lens* $(\mathcal{V} +_L \mathcal{C})$
    by (*simp add*: *bij-region-coregion*)
qed

lemma *put-region-coregion-cover*:

$put_{\mathcal{V}}$ $(put_{\mathcal{C}}\ s_1\ (get_{\mathcal{C}}\ s_2))\ (get_{\mathcal{V}}\ s_2) = s_2$
proof −
have $put_{\mathcal{V}}\ (put_{\mathcal{C}}\ s_1\ (get_{\mathcal{C}}\ s_2))\ (get_{\mathcal{V}}\ s_2) = put_{\mathcal{V}\ +_L\ \mathcal{C}}\ s_1\ (get_{\mathcal{V}\ +_L\ \mathcal{C}}\ s_2)$
by (*simp add*: *lens-defs*)
also have ... $= s_2$
by (*simp add*: *bij-region-coregion*)
finally show *?thesis* **.**
qed

end

declare *sym-lens.vwb-region* [*simp*]
declare *sym-lens.vwb-coregion* [*simp*]
declare *sym-lens.indep-region-coregion* [*simp*]

lemma *sym-lens-psym* [*simp*]: *sym-lens x* $\Longrightarrow$ *psym-lens x*
by (*simp add*: *psym-lens-def sym-lens.bij-region-coregion*)

lemma *sym-lens-compl* [*simp*]: *sym-lens a* $\Longrightarrow$ *sym-lens* $(-_L\ a)$
apply (*simp add*: *slens-compl-def*)
apply (*rule sym-lens.intro*, *simp-all*)
using *lens-indep-sym sym-lens.indep-region-coregion* **apply** *blast*
using *bij-lens-equiv lens-plus-comm sym-lens-def* **apply** *blast*
done

end

7 Lens Instances

theory *Lens-Instances*
imports *Lens-Order Lens-Symmetric HOL−Eisbach.Eisbach*
keywords *alphabet statespace* :: *thy-defn*
begin

In this section we define a number of concrete instantiations of the lens locales, including functions lenses, list lenses, and record lenses.

7.1 Function Lens

A function lens views the valuation associated with a particular domain element $'a$. We require that range type of a lens function has cardinality of at least 2; this ensures that properties of independence are provable.

definition *fun-lens* :: $'a \Rightarrow ('b::two \Longrightarrow ('a \Rightarrow 'b))$ **where**
[*lens-defs*]: *fun-lens* $x = (\!|\ lens\text{-}get = (\lambda\ f.\ f\ x),\ lens\text{-}put = (\lambda\ f\ u.\ f(x := u))\ |\!)$

lemma *fun-vwb-lens*: *vwb-lens* (*fun-lens x*)
by (*unfold-locales*, *simp-all add*: *fun-lens-def*)

Two function lenses are independent if and only if the domain elements are different.

lemma *fun-lens-indep*:
fun-lens x $\bowtie$ *fun-lens y* $\longleftrightarrow x \neq y$
proof −
obtain $u\ v :: 'a::two$ **where** $u \neq v$

using *two-diff* **by** *auto*
thus *?thesis*
by (*auto simp add*: *fun-lens-def lens-indep-def*)
qed

7.2 Function Range Lens

The function range lens allows us to focus on a particular region of a function's range.

definition *fun-ran-lens* :: $('c \Longrightarrow 'b) \Rightarrow (('a \Rightarrow 'b) \Longrightarrow '\alpha) \Rightarrow (('a \Rightarrow 'c) \Longrightarrow '\alpha)$ **where**
[*lens-defs*]: *fun-ran-lens X Y* = ⦇ *lens-get* = λ *s*. $get_X \circ get_Y$ *s*
, *lens-put* = λ *s v*. put_Y *s* (λ *x*::'*a*. put_X (get_Y *s x*) (*v x*)) ⦈

lemma *fun-ran-mwb-lens*: ⟦ *mwb-lens X*; *mwb-lens Y* ⟧ ⟹ *mwb-lens* (*fun-ran-lens X Y*)
by (*unfold-locales*, *auto simp add*: *fun-ran-lens-def*)

lemma *fun-ran-wb-lens*: ⟦ *wb-lens X*; *wb-lens Y* ⟧ ⟹ *wb-lens* (*fun-ran-lens X Y*)
by (*unfold-locales*, *auto simp add*: *fun-ran-lens-def*)

lemma *fun-ran-vwb-lens*: ⟦ *vwb-lens X*; *vwb-lens Y* ⟧ ⟹ *vwb-lens* (*fun-ran-lens X Y*)
by (*unfold-locales*, *auto simp add*: *fun-ran-lens-def*)

7.3 Map Lens

The map lens allows us to focus on a particular region of a partial function's range. It is only a mainly well-behaved lens because it does not satisfy the PutGet law when the view is not in the domain.

definition *map-lens* :: $'a \Rightarrow ('b \Longrightarrow ('a \rightharpoonup 'b))$ **where**
[*lens-defs*]: *map-lens x* = ⦇ *lens-get* = (λ *f*. *the* (*f x*)), *lens-put* = (λ *f u*. *f*(*x* ↦ *u*)) ⦈

lemma *map-mwb-lens*: *mwb-lens* (*map-lens x*)
by (*unfold-locales*, *simp-all add*: *map-lens-def*)

lemma *source-map-lens*: $\mathcal{S}_{map\text{-}lens\ x} = \{f.\ x \in dom(f)\}$
by (*force simp add*: *map-lens-def lens-source-def*)

7.4 List Lens

The list lens allows us to view a particular element of a list. In order to show it is mainly well-behaved we need to define to additional list functions. The following function adds a number undefined elements to the end of a list.

definition *list-pad-out* :: '*a list* ⇒ *nat* ⇒ '*a list* **where**
list-pad-out xs k = *xs* @ *replicate* (*k* + *1* − *length xs*) *undefined*

The following function is like *list-update* but it adds additional elements to the list if the list isn't long enough first.

definition *list-augment* :: '*a list* ⇒ *nat* ⇒ '*a* ⇒ '*a list* **where**
list-augment xs k v = (*list-pad-out xs k*)[*k* := *v*]

The following function is like (!) but it expressly returns *undefined* when the list isn't long enough.

definition *nth'* :: '*a list* ⇒ *nat* ⇒ '*a* **where**
nth' xs i = (*if* (*length xs* > *i*) *then xs* ! *i else undefined*)

We can prove some additional laws about list update and append.

lemma *list-update-append-lemma1*: $i < length\ xs \Longrightarrow xs[i := v]\ @\ ys = (xs\ @\ ys)[i := v]$
by (*simp add*: *list-update-append*)

lemma *list-update-append-lemma2*: $i < length\ ys \Longrightarrow xs\ @\ ys[i := v] = (xs\ @\ ys)[i + length\ xs := v]$
by (*simp add*: *list-update-append*)

We can also prove some laws about our new operators.

lemma *nth′-0* [*simp*]: $nth'\ (x\ \#\ xs)\ 0 = x$
by (*simp add*: *nth′-def*)

lemma *nth′-Suc* [*simp*]: $nth'\ (x\ \#\ xs)\ (Suc\ n) = nth'\ xs\ n$
by (*simp add*: *nth′-def*)

lemma *list-augment-0* [*simp*]:
$list\text{-}augment\ (x\ \#\ xs)\ 0\ y = y\ \#\ xs$
by (*simp add*: *list-augment-def list-pad-out-def*)

lemma *list-augment-Suc* [*simp*]:
$list\text{-}augment\ (x\ \#\ xs)\ (Suc\ n)\ y = x\ \#\ list\text{-}augment\ xs\ n\ y$
by (*simp add*: *list-augment-def list-pad-out-def*)

lemma *list-augment-twice*:
$list\text{-}augment\ (list\text{-}augment\ xs\ i\ u)\ j\ v = (list\text{-}pad\text{-}out\ xs\ (max\ i\ j))[i{:=}u,\ j{:=}v]$
apply (*auto simp add*: *list-augment-def list-pad-out-def list-update-append-lemma1 replicate-add*[*THEN sym*] *max-def*)
apply (*metis Suc-le-mono add.commute diff-diff-add diff-le-mono le-add-diff-inverse2*)
done

lemma *list-augment-last* [*simp*]:
$list\text{-}augment\ (xs\ @\ [y])\ (length\ xs)\ z = xs\ @\ [z]$
by (*induct xs*, *simp-all*)

lemma *list-augment-idem* [*simp*]:
$i < length\ xs \Longrightarrow list\text{-}augment\ xs\ i\ (xs\ !\ i) = xs$
by (*simp add*: *list-augment-def list-pad-out-def*)

We can now prove that *list-augment* is commutative for different (arbitrary) indices.

lemma *list-augment-commute*:
$i \neq j \Longrightarrow list\text{-}augment\ (list\text{-}augment\ \sigma\ j\ v)\ i\ u = list\text{-}augment\ (list\text{-}augment\ \sigma\ i\ u)\ j\ v$
by (*simp add*: *list-augment-twice list-update-swap max.commute*)

We can also prove that we can always retrieve an element we have added to the list, since *list-augment* extends the list when necessary. This isn't true of *list-update*.

lemma *nth-list-augment*: $list\text{-}augment\ xs\ k\ v\ !\ k = v$
by (*simp add*: *list-augment-def list-pad-out-def*)

lemma *nth′-list-augment*: $nth'\ (list\text{-}augment\ xs\ k\ v)\ k = v$
by (*auto simp add*: *nth′-def nth-list-augment list-augment-def list-pad-out-def*)

The length is expanded if not already long enough, or otherwise left as it is.

lemma *length-list-augment-1*: $k \geq length\ xs \Longrightarrow length\ (list\text{-}augment\ xs\ k\ v) = Suc\ k$
by (*simp add*: *list-augment-def list-pad-out-def*)

lemma *length-list-augment-2*: $k < \mathit{length}\ xs \Longrightarrow \mathit{length}\ (\mathit{list\text{-}augment}\ xs\ k\ v) = \mathit{length}\ xs$
by (*simp add*: *list-augment-def list-pad-out-def*)

We also have it that *list-augment* cancels itself.

lemma *list-augment-same-twice*: $\mathit{list\text{-}augment}\ (\mathit{list\text{-}augment}\ xs\ k\ u)\ k\ v = \mathit{list\text{-}augment}\ xs\ k\ v$
by (*simp add*: *list-augment-def list-pad-out-def*)

lemma *nth'-list-augment-diff*: $i \neq j \Longrightarrow nth'\ (\mathit{list\text{-}augment}\ \sigma\ i\ v)\ j = nth'\ \sigma\ j$
by (*auto simp add*: *list-augment-def list-pad-out-def nth-append nth'-def*)

Finally we can create the list lenses, of which there are three varieties. One that allows us to view an index, one that allows us to view the head, and one that allows us to view the tail. They are all mainly well-behaved lenses.

definition *list-lens* :: $\mathit{nat} \Rightarrow ('a::two \Longrightarrow 'a\ \mathit{list})$ **where**
[*lens-defs*]: $\mathit{list\text{-}lens}\ i = (\!|\ \mathit{lens\text{-}get} = (\lambda\ xs.\ nth'\ xs\ i)$
$,\ \mathit{lens\text{-}put} = (\lambda\ xs\ x.\ \mathit{list\text{-}augment}\ xs\ i\ x)\ |\!)$

abbreviation *hd-lens* (hd_L) **where** $\mathit{hd\text{-}lens} \equiv \mathit{list\text{-}lens}\ 0$

definition *tl-lens* :: $'a\ \mathit{list} \Longrightarrow 'a\ \mathit{list}$ (tl_L) **where**
[*lens-defs*]: $\mathit{tl\text{-}lens} = (\!|\ \mathit{lens\text{-}get} = (\lambda\ xs.\ tl\ xs)$
$,\ \mathit{lens\text{-}put} = (\lambda\ xs\ xs'.\ hd\ xs\ \#\ xs')\ |\!)$

lemma *list-mwb-lens*: *mwb-lens* (*list-lens x*)
by (*unfold-locales*, *simp-all add*: *list-lens-def nth'-list-augment list-augment-same-twice*)

The set of constructible sources is precisely those where the length is greater than the given index.

lemma *source-list-lens*: $\mathcal{S}_{\mathit{list\text{-}lens}\ i} = \{xs.\ \mathit{length}\ xs > i\}$
apply (*auto simp add*: *lens-source-def list-lens-def*)
apply (*metis length-list-augment-1 length-list-augment-2 lessI not-less*)
apply (*metis list-augment-idem*)
done

lemma *tail-lens-mwb*:
mwb-lens tl_L
by (*unfold-locales*, *simp-all add*: *tl-lens-def*)

lemma *source-tail-lens*: $\mathcal{S}_{tl_L} = \{xs.\ xs \neq []\}$
using *list.exhaust-sel* **by** (*auto simp add*: *tl-lens-def lens-source-def*)

Independence of list lenses follows when the two indices are different.

lemma *list-lens-indep*:
$i \neq j \Longrightarrow \mathit{list\text{-}lens}\ i \bowtie \mathit{list\text{-}lens}\ j$
by (*simp add*: *list-lens-def lens-indep-def list-augment-commute nth'-list-augment-diff*)

lemma *hd-tl-lens-indep* [*simp*]:
$hd_L \bowtie tl_L$
apply (*rule lens-indepI*)
apply (*simp-all add*: *list-lens-def tl-lens-def*)
apply (*metis hd-conv-nth hd-def length-greater-0-conv list.case*(*1*) *nth'-def nth'-list-augment*)
apply (*metis* (*full-types*) *hd-conv-nth hd-def length-greater-0-conv list.case*(*1*) *nth'-def*)
apply (*metis One-nat-def diff-Suc-Suc diff-zero length-0-conv length-list-augment-1 length-tl linorder-not-less list.exhaust list.sel*(*2*) *list.sel*(*3*) *list-augment-0 not-less-zero*)

done

lemma *hd-tl-lens-pbij*: *pbij-lens* $(hd_L +_L tl_L)$
by (*unfold-locales*, *auto simp add*: *lens-defs*)

7.5 Record Field Lenses

We also add support for record lenses. Every record created can yield a lens for each field. These cannot be created generically and thus must be defined case by case as new records are created. We thus create a new Isabelle outer syntax command **alphabet** which enables this. We first create syntax that allows us to obtain a lens from a given field using the internal record syntax translations.

abbreviation (*input*) *fld-put f* ≡ (λ σ u. f (λ-. u) σ)

syntax
-FLDLENS :: *id* ⇒ *logic* (*FLDLENS* -)
translations
FLDLENS x => (| *lens-get* = *x*, *lens-put* = *CONST fld-put* (*-update-name x*) |)

We also allow the extraction of the "base lens", which characterises all the fields added by a record without the extension.

syntax
-BASELENS :: *id* ⇒ *logic* (*BASELENS* -)

abbreviation (*input*) *base-lens t e m* ≡ (| *lens-get* = *t*, *lens-put* = λ *s v*. *e v* (*m s*) |)

ML ‹
```
fun baselens-tr [Free (name, -)] =
  let
    val extend = Free (name ^.extend, dummyT);
    val truncate = Free (name ^.truncate, dummyT);
    val more = Free (name ^.more, dummyT);
  in
    Const (@{const-syntax base-lens}, dummyT) $ truncate $ extend $ more
  end
| baselens-tr - = raise Match;
```
›

parse-translation ‹[(@{*syntax-const -BASELENS*}, *K baselens-tr*)]›

We also introduce the **alphabet** command that creates a record with lenses for each field. For each field a lens is created together with a proof that it is very well-behaved, and for each pair of lenses an independence theorem is generated. Alphabets can also be extended which yields sublens proofs between the extension field lens and record extension lenses.

ML-file ‹*Lens-Lib.ML*›
ML-file ‹*Lens-Record.ML*›

The following theorem attribute stores splitting theorems for alphabet types which which is useful for proof automation.

named-theorems *alpha-splits*

7.6 Locale State Spaces

Alternative to the alphabet command, we also introduce the statespace command, which implements Schirmer and Wenzel's locale-based approach to state space modelling [9].

It has the advantage of allowing multiple inheritance of state spaces, and also variable names are fully internalised with the locales. The approach is also far simpler than record-based state spaces.

It has the disadvantage that variables may not be fully polymorphic, unlike for the alphabet command. This makes it in general unsuitable for UTP theory alphabets.

ML-file ‹*Lens-Statespace.ML*›

7.7 Type Definition Lens

Every type defined by a **typedef** command induces a partial bijective lens constructed using the abstraction and representation functions.

context *type-definition*
begin

definition *typedef-lens* :: $'b \Longrightarrow 'a$ ($typedef_L$) **where**
[*lens-defs*]: $typedef_L = (\!|\ lens\text{-}get = Abs,\ lens\text{-}put = (\lambda\ s.\ Rep)\ |\!)$

lemma *pbij-typedef-lens* [*simp*]: *pbij-lens* $typedef_L$
  by (*unfold-locales*, *simp-all add*: *lens-defs Rep-inverse*)

lemma *source-typedef-lens*: $\mathcal{S}_{typedef_L} = A$
  using *Rep-cases* **by** (*auto simp add*: *lens-source-def lens-defs Rep*)

lemma *bij-typedef-lens-UNIV*: $A = UNIV \Longrightarrow$ *bij-lens* $typedef_L$
  by (*auto intro*: *pbij-vwb-is-bij-lens simp add*: *mwb-UNIV-src-is-vwb-lens source-typedef-lens*)

end

7.8 Mapper Lenses

definition *lmap-lens* ::
  $(('\alpha \Rightarrow '\beta) \Rightarrow ('\gamma \Rightarrow '\delta)) \Rightarrow$
   $(('\beta \Rightarrow '\alpha) \Rightarrow '\delta \Rightarrow '\gamma) \Rightarrow$
   $('\gamma \Rightarrow '\alpha) \Rightarrow$
   $('\beta \Longrightarrow '\alpha) \Rightarrow$
   $('\delta \Longrightarrow '\gamma)$ **where**
  [*lens-defs*]:
  $lmap\text{-}lens\ f\ g\ h\ l = (\!|$
  $lens\text{-}get = f\ (get_l)$,
  $lens\text{-}put = g\ o\ (put_l)\ o\ h\ |\!)$

The parse translation below yields a heterogeneous mapping lens for any record type. This achieved through the utility function above that constructs a functorial lens. This takes as input a heterogeneous mapping function that lifts a function on a record's extension type to an update on the entire record, and also the record's "more" function. The first input is given twice as it has different polymorphic types, being effectively a type functor construction which are not explicitly supported by HOL. We note that the *more-update* function does something similar to the extension lifting, but is not precisely suitable here since it only considers homogeneous functions, namely of type $'a \Rightarrow 'a$ rather than $'a \Rightarrow 'b$.

syntax
-lmap :: *id* ⇒ *logic* (*lmap*[-])

```
ML ‹
  fun lmap-tr [Free (name, -)] =
    let
      val extend = Free (name ^.extend, dummyT);
      val truncate = Free (name ^.truncate, dummyT);
      val more = Free (name ^.more, dummyT);
      val map-ext = Abs (f, dummyT,
                    Abs (r, dummyT,
                      extend $ (truncate $ Bound 0) $ (Bound 1 $ (more $ (Bound 0)))))

    in
      Const (@{const-syntax lmap-lens}, dummyT) $ map-ext $ map-ext $ more
    end
  | lmap-tr - = raise Match;
›
```

parse-translation ‹[(@{*syntax-const -lmap*}, *K lmap-tr*)]›

7.9 Lens Interpretation

named-theorems *lens-interp-laws*

locale *lens-interp* = *interp*
begin
declare *meta-interp-law* [*lens-interp-laws*]
declare *all-interp-law* [*lens-interp-laws*]
declare *exists-interp-law* [*lens-interp-laws*]

end

7.10 Tactic

A simple tactic for simplifying lens expressions

declare *split-paired-all* [*alpha-splits*]

method *lens-simp* = (*simp add*: *alpha-splits lens-defs prod.case-eq-if*)

end

8 Lenses

theory *Lenses*
imports
Lens-Laws
Lens-Algebra
Lens-Order
Lens-Symmetric
Lens-Instances
begin end

9 Prisms

theory *Prisms*
  imports *Lenses*
begin

9.1 Signature and Axioms

Prisms are like lenses, but they act on sum types rather than product types [8]. See https://hackage.haskell.org/package/lens-4.15.2/docs/Control-Lens-Prism.html for more information.

record ('*v*, '*s*) *prism* =
  prism-match :: '*s* ⇒ '*v option* (*match*ı)
  prism-build :: '*v* ⇒ '*s* (*build*ı)

type-notation
  prism (**infixr** $\Longrightarrow_\triangle$ *0*)

locale *wb-prism* =
  fixes *x* :: '*v* $\Longrightarrow_\triangle$ '*s* (**structure**)
  assumes *match-build*: *match* (*build v*) = *Some v*
  and *build-match*: *match s* = *Some v* ⟹ *s* = *build v*
begin

  lemma *build-match-iff*: *match s* = *Some v* ⟷ *s* = *build v*
    using *build-match match-build* **by** *blast*

  lemma *range-build*: *range build* = *dom match*
    using *build-match match-build* **by** *fastforce*
end

declare *wb-prism.match-build* [*simp*]
declare *wb-prism.build-match* [*simp*]

9.2 Co-dependence

The relation states that two prisms construct disjoint elements of the source. This can occur, for example, when the two prisms characterise different constructors of an algebraic datatype.

definition *prism-diff* :: ('*a* $\Longrightarrow_\triangle$ '*s*) ⇒ ('*b* $\Longrightarrow_\triangle$ '*s*) ⇒ *bool* (**infix** ∇ *50*) **where**
prism-diff X Y = (*range* $build_X$ ∩ *range* $build_Y$ = {})

lemma *prism-diff-intro*:
  (⋀ s_1 s_2. $build_X$ s_1 = $build_Y$ s_2 ⟹ *False*) ⟹ *X* ∇ *Y*
  by (*auto simp add*: *prism-diff-def*)

lemma *prism-diff-irrefl*: ¬ *X* ∇ *X*
  by (*simp add*: *prism-diff-def*)

lemma *prism-diff-sym*: *X* ∇ *Y* ⟹ *Y* ∇ *X*
  by (*auto simp add*: *prism-diff-def*)

lemma *prism-diff-build*: *X* ∇ *Y* ⟹ $build_X$ *u* ≠ $build_Y$ *v*
  by (*simp add*: *disjoint-iff-not-equal prism-diff-def*)

9.3 Summation

definition *prism-plus* :: $('a \Longrightarrow_\triangle 's) \Rightarrow ('b \Longrightarrow_\triangle 's) \Rightarrow 'a + 'b \Longrightarrow_\triangle 's$ (**infixl** $+_\triangle$ *85*)
where
$X +_\triangle Y = (\!|$ *prism-match* = (λ *s*. *case* (*match*$_X$ *s*, *match*$_Y$ *s*) *of*
(*Some u*, -) ⇒ *Some* (*Inl u*) |
(*None*, *Some v*) ⇒ *Some* (*Inr v*) |
(*None*, *None*) ⇒ *None*),
prism-build = (λ *v*. *case v of Inl x* ⇒ *build*$_X$ *x* | *Inr y* ⇒ *build*$_Y$ *y*) $|\!)$

9.4 Instances

definition *prism-suml* :: ('*a*, '*a* + '*b*) *prism* (*Inl*$_\triangle$) **where**
[*lens-defs*]: *prism-suml* = $(\!|$ *prism-match* = (λ *v*. *case v of Inl x* ⇒ *Some x* | - ⇒ *None*), *prism-build* = *Inl* $|\!)$

definition *prism-sumr* :: ('*b*, '*a* + '*b*) *prism* (*Inr*$_\triangle$) **where**
[*lens-defs*]: *prism-sumr* = $(\!|$ *prism-match* = (λ *v*. *case v of Inr x* ⇒ *Some x* | - ⇒ *None*), *prism-build* = *Inr* $|\!)$

lemma *wb-prim-suml*: *wb-prism Inl*$_\triangle$
apply (*unfold-locales*)
apply (*simp-all add*: *prism-suml-def sum.case-eq-if*)
apply (*metis option.inject option.simps*(*3*) *sum.collapse*(*1*))
done

lemma *wb-prim-sumr*: *wb-prism Inr*$_\triangle$
apply (*unfold-locales*)
apply (*simp-all add*: *prism-sumr-def sum.case-eq-if*)
apply (*metis option.distinct*(*1*) *option.inject sum.collapse*(*2*))
done

lemma *prism-suml-indep-sumr* [*simp*]: *Inl*$_\triangle$ ∇ *Inr*$_\triangle$
by (*auto simp add*: *prism-diff-def lens-defs*)

9.5 Lens correspondence

Every well-behaved prism can be represented by a partial bijective lens. We prove this by exhibiting conversion functions and showing they are (almost) inverses.

definition *prism-lens* :: ('*a*, '*s*) *prism* ⇒ ('*a* ⟹ '*s*) **where**
prism-lens X = $(\!|$ *lens-get* = (λ *s*. *the* (*match*$_X$ *s*)), *lens-put* = (λ *s v*. *build*$_X$ *v*) $|\!)$

definition *lens-prism* :: ('*a* ⟹ '*s*) ⇒ ('*a*, '*s*) *prism* **where**
lens-prism X = $(\!|$ *prism-match* = (λ *s*. *if* ($s \in \mathcal{S}_X$) *then Some* (*get*$_X$ *s*) *else None*)
, *prism-build* = *create*$_X$ $|\!)$

lemma *get-prism-lens*: *get*$_{prism\text{-}lens\ X}$ = *the* ∘ *match*$_X$
by (*simp add*: *prism-lens-def fun-eq-iff*)

lemma *src-prism-lens*: $\mathcal{S}_{prism\text{-}lens\ X}$ = *range* (*build*$_X$)
by (*auto simp add*: *prism-lens-def lens-source-def*)

lemma *create-prism-lens*: *create*$_{prism\text{-}lens\ X}$ = *build*$_X$
by (*simp add*: *prism-lens-def lens-create-def fun-eq-iff*)

lemma *prism-lens-inverse*:
  wb-prism X $\Longrightarrow$ *lens-prism* (*prism-lens X*) = *X*
  unfolding *lens-prism-def src-prism-lens create-prism-lens get-prism-lens*
  by (*auto intro*: *prism.equality simp add*: *fun-eq-iff domIff wb-prism.range-build*)

Function *lens-prism* is almost inverted by *prism-lens*. The *put* functions are identical, but the *get* functions differ when applied to a source where the prism *X* is undefined.

lemma *lens-prism-put-inverse*:
  pbij-lens X $\Longrightarrow$ $put_{prism\text{-}lens\ (lens\text{-}prism\ X)} = put_X$
  unfolding *prism-lens-def lens-prism-def*
  by (*auto simp add*: *fun-eq-iff pbij-lens.put-is-create*)

lemma *wb-prism-implies-pbij-lens*:
  wb-prism X $\Longrightarrow$ *pbij-lens* (*prism-lens X*)
  by (*unfold-locales*, *simp-all add*: *prism-lens-def*)

lemma *pbij-lens-implies-wb-prism*:
  assumes *pbij-lens X*
  shows *wb-prism* (*lens-prism X*)
proof (*unfold-locales*)
  fix *s v*
  show $match_{lens\text{-}prism\ X}\ (build_{lens\text{-}prism\ X}\ v) = Some\ v$
    by (*simp add*: *lens-prism-def weak-lens.create-closure assms*)
  assume *a*: $match_{lens\text{-}prism\ X}\ s = Some\ v$
  show $s = build_{lens\text{-}prism\ X}\ v$
  proof (*cases* $s \in \mathcal{S}_X$)
    case *True*
    with *a assms* **show** *?thesis*
      by (*simp add*: *lens-prism-def lens-create-def*,
          metis mwb-lens.weak-get-put pbij-lens.put-det pbij-lens-mwb)
  next
    case *False*
    with *a assms* **show** *?thesis* **by** (*simp add*: *lens-prism-def*)
  qed
qed

ML-file ⟨*Prism-Lib.ML*⟩

end

10 Channel Types

theory *Channel-Type*
  imports *Prisms*
  keywords *chantype* :: *thy-defn*
begin

A channel type is a simplified algebraic datatype where each constructor has exactly one parameter, and it is wrapped up as a prism. It a dual of an alphabet type.

definition *ctor-prism* :: $('a \Rightarrow 'd) \Rightarrow ('d \Rightarrow bool) \Rightarrow ('d \Rightarrow 'a) \Rightarrow ('a \Longrightarrow_\triangle 'd)$ **where**
[*lens-defs*]:
ctor-prism ctor disc sel = ⦇ *prism-match* = (λ *d*. *if* (*disc d*) *then Some* (*sel d*) *else None*)
                , *prism-build* = *ctor* ⦈

lemma *wb-ctor-prism-intro*:
assumes
$\bigwedge$ *v. disc* (*ctor v*)
$\bigwedge$ *v. sel* (*ctor v*) = *v*
$\bigwedge$ *s. disc s* $\Longrightarrow$ *ctor* (*sel s*) = *s*
shows *wb-prism* (*ctor-prism ctor disc sel*)
by (*unfold-locales*, *simp-all add*: *lens-defs assms*)
(*metis assms*(*3*) *option.distinct*(*1*) *option.sel*)

lemma *ctor-codep-intro*:
assumes $\bigwedge$ *x y. ctor1 x* $\neq$ *ctor2 y*
shows *ctor-prism ctor1 disc1 sel1* ∇ *ctor-prism ctor2 disc2 sel2*
by (*rule prism-diff-intro*, *simp add*: *lens-defs assms*)

ML-file *Channel-Type.ML*

end

11 Data spaces

theory *Dataspace*
imports *Lenses Prisms*
keywords *dataspace* :: *thy-defn* **and** *constants variables channels*
begin

A data space is like a more sophisticated version of a locale-based state space. It allows us to introduce both variables, modelled by lenses, and channels, modelled by prisms. It also allows local constants, and assumptions over them.

ML-file *Dataspace.ML*

end

12 Scenes

theory *Scenes*
imports *Lens-Instances*
begin

Like lenses, scenes characterise a region of a source type. However, unlike lenses, scenes do not explicitly assign a view type to this region, and consequently they have just one type parameter. This means they can be more flexibly composed, and in particular it is possible to show they form nice algebraic structures in Isabelle/HOL. They are mainly of use in characterising sets of variables, where, of course, we do not care about the types of those variables and therefore representing them as lenses is inconvenient.

12.1 Overriding Functions

Overriding functions provide an abstract way of replacing a region of an existing source with the corresponding region of another source.

locale *overrider* =
fixes *F* :: $'s \Rightarrow 's \Rightarrow 's$ (**infixl** $\triangleright$ *65*)
assumes

ovr-overshadow-left: $x \triangleright y \triangleright z = x \triangleright z$ **and**
ovr-overshadow-right: $x \triangleright (y \triangleright z) = x \triangleright z$
begin
lemma *ovr-assoc*: $x \triangleright (y \triangleright z) = x \triangleright y \triangleright z$
by (*simp add*: *ovr-overshadow-left ovr-overshadow-right*)
end

locale *idem-overrider* = *overrider* +
assumes *ovr-idem*: $x \triangleright x = x$

declare *overrider.ovr-overshadow-left* [*simp*]
declare *overrider.ovr-overshadow-right* [*simp*]
declare *idem-overrider.ovr-idem* [*simp*]

12.2 Scene Type

typedef *'s scene* = $\{F :: 's \Rightarrow 's \Rightarrow 's.$ *overrider* $F\}$
by (*rule-tac* $x=\lambda\ x\ y.\ x$ **in** *exI*, *simp*, *unfold-locales*, *simp-all*)

setup-lifting *type-definition-scene*

lift-definition *idem-scene* :: *'s scene* ⇒ *bool* **is** *idem-overrider* **.**

lift-definition *region* :: *'s scene* ⇒ *'s rel*
is $\lambda\ F.\ \{(s_1, s_2).\ (\forall\ s.\ F\ s\ s_1 = F\ s\ s_2)\}$ **.**

lift-definition *coregion* :: *'s scene* ⇒ *'s rel*
is $\lambda\ F.\ \{(s_1, s_2).\ (\forall\ s.\ F\ s_1\ s = F\ s_2\ s)\}$ **.**

lemma *equiv-region*: *equiv UNIV* (*region X*)
apply (*transfer*)
apply (*rule equivI*)
apply (*rule refl-onI*)
apply (*auto*)
apply (*rule symI*)
apply (*auto*)
apply (*rule transI*)
apply (*auto*)
done

lemma *equiv-coregion*: *equiv UNIV* (*coregion X*)
apply (*transfer*)
apply (*rule equivI*)
apply (*rule refl-onI*)
apply (*auto*)
apply (*rule symI*)
apply (*auto*)
apply (*rule transI*)
apply (*auto*)
done

lemma *region-coregion-Id*:
idem-scene X ⟹ *region X* ∩ *coregion X* = *Id*
by (*transfer*, *auto*, *metis idem-overrider.ovr-idem*)

lemma *state-eq-iff*: *idem-scene S* ⟹ $x = y \longleftrightarrow (x, y) \in$ *region S* $\wedge\ (x, y) \in$ *coregion S*

by (*metis IntE IntI pair-in-Id-conv region-coregion-Id*)

lift-definition *scene-override* :: $'a \Rightarrow 'a \Rightarrow ('a\ scene) \Rightarrow 'a$ (- $\oplus_S$ - *on* - [*95*,*0*,*96*] *95*)
is $\lambda\ s_1\ s_2\ F.\ F\ s_1\ s_2$ **.**

abbreviation (*input*) *scene-copy* :: $'a\ scene \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a)$ (cp_{-}) **where**
$cp_A\ s \equiv (\lambda\ s'.\ s' \oplus_S s\ on\ A)$

lemma *scene-override-idem* [*simp*]: $idem\text{-}scene\ X \Longrightarrow s \oplus_S s\ on\ X = s$
by (*transfer*, *simp*)

lemma *scene-override-overshadow-left* [*simp*]:
$S_1 \oplus_S S_2\ on\ X \oplus_S S_3\ on\ X = S_1 \oplus_S S_3\ on\ X$
by (*transfer*, *simp*)

lemma *scene-override-overshadow-right* [*simp*]:
$S_1 \oplus_S (S_2 \oplus_S S_3\ on\ X)\ on\ X = S_1 \oplus_S S_3\ on\ X$
by (*transfer*, *simp*)

definition *scene-equiv* :: $'a \Rightarrow 'a \Rightarrow ('a\ scene) \Rightarrow bool$ (- $\approx_S$ - *on* - [*65*,*0*,*66*] *65*) **where**
[*lens-defs*]: $S_1 \approx_S S_2\ on\ X = (S_1 \oplus_S S_2\ on\ X = S_1)$

lemma *scene-equiv-region*: $idem\text{-}scene\ X \Longrightarrow region\ X = \{(S_1, S_2).\ S_1 \approx_S S_2\ on\ X\}$
by (*simp add*: *lens-defs*, *transfer*, *auto*)
(*metis idem-overrider.ovr-idem*, *metis overrider.ovr-overshadow-right*)

lift-definition *scene-indep* :: $'a\ scene \Rightarrow 'a\ scene \Rightarrow bool$ (**infix** $\bowtie_S$ *50*)
is $\lambda\ F\ G.\ (\forall\ s_1\ s_2\ s_3.\ G\ (F\ s_1\ s_2)\ s_3 = F\ (G\ s_1\ s_3)\ s_2)$ **.**

lemma *scene-indep-override*:
$X \bowtie_S Y = (\forall\ s_1\ s_2\ s_3.\ s_1 \oplus_S s_2\ on\ X \oplus_S s_3\ on\ Y = s_1 \oplus_S s_3\ on\ Y \oplus_S s_2\ on\ X)$
by (*transfer*, *auto*)

lemma *scene-indep-copy*:
$X \bowtie_S Y = (\forall\ s_1\ s_2.\ cp_X\ s_1 \circ cp_Y\ s_2 = cp_Y\ s_2 \circ cp_X\ s_1)$
by (*auto simp add*: *scene-indep-override comp-def fun-eq-iff*)

lemma *scene-indep-sym*:
$X \bowtie_S Y \Longrightarrow Y \bowtie_S X$
by (*transfer*, *auto*)

Compatibility is a weaker notion than independence; the scenes can overlap but they must agree when they do.

lift-definition *scene-compat* :: $'a\ scene \Rightarrow 'a\ scene \Rightarrow bool$ (**infix** $\#\#_S$ *50*)
is $\lambda\ F\ G.\ (\forall\ s_1\ s_2.\ G\ (F\ s_1\ s_2)\ s_2 = F\ (G\ s_1\ s_2)\ s_2)$ **.**

lemma *scene-compat-copy*:
$X\ \#\#_S\ Y = (\forall\ s.\ cp_X\ s \circ cp_Y\ s = cp_Y\ s \circ cp_X\ s)$
by (*transfer*, *auto simp add*: *fun-eq-iff*)

lemma *scene-indep-compat* [*simp*]: $X \bowtie_S Y \Longrightarrow X\ \#\#_S\ Y$
by (*transfer*, *auto*)

lemma *scene-compat-refl*: $X\ \#\#_S\ X$
by (*transfer*, *simp*)

lemma *scene-compat-sym*: $X \#\#_S\ Y \Longrightarrow Y \#\#_S\ X$
  by (*transfer*, *simp*)

lemma *scene-override-commute-indep*:
  assumes $X \bowtie_S Y$
  shows $S_1 \oplus_S S_2$ *on* $X \oplus_S S_3$ *on* $Y = S_1 \oplus_S S_3$ *on* $Y \oplus_S S_2$ *on* X
  using *assms*
  by (*transfer*, *auto*)

instantiation *scene* :: (*type*) {*bot*, *top*, *uminus*, *sup*, *inf*}
begin
  lift-definition *bot-scene* :: *'s scene* **is** $\lambda\ x\ y.\ x$ **by** (*unfold-locales*, *simp-all*)
  lift-definition *top-scene* :: *'s scene* **is** $\lambda\ x\ y.\ y$ **by** (*unfold-locales*, *simp-all*)
  lift-definition *uminus-scene* :: *'s scene* ⇒ *'s scene* **is** $\lambda\ F\ x\ y.\ F\ y\ x$
    by (*unfold-locales*, *simp-all*)

Scene union requires that the two scenes are at least compatible. If they are not, the result is the bottom scene.

  lift-definition *sup-scene* :: *'s scene* ⇒ *'s scene* ⇒ *'s scene*
    is $\lambda\ F\ G.$ *if* $(\forall\ s_1\ s_2.\ G\ (F\ s_1\ s_2)\ s_2 = F\ (G\ s_1\ s_2)\ s_2)$ *then* $(\lambda\ s_1\ s_2.\ G\ (F\ s_1\ s_2)\ s_2)$ *else* $(\lambda\ s_1\ s_2.\ s_1)$
    by (*unfold-locales*, *auto*, *metis overrider.ovr-overshadow-right*)
  definition *inf-scene* :: *'s scene* ⇒ *'s scene* ⇒ *'s scene* **where**
    [*lens-defs*]: *inf-scene* $X\ Y = -\ (sup\ (-\ X)\ (-\ Y))$
  instance **..**
end

abbreviation *union-scene* :: *'s scene* ⇒ *'s scene* ⇒ *'s scene* (**infixl** $\sqcup_S$ *65*)
where *union-scene* ≡ *sup*

abbreviation *inter-scene* :: *'s scene* ⇒ *'s scene* ⇒ *'s scene* (**infixl** $\sqcap_S$ *70*)
where *inter-scene* ≡ *inf*

abbreviation *top-scene* :: *'s scene* ($\top_S$)
where *top-scene* ≡ *top*

abbreviation *bot-scene* :: *'s scene* ($\bot_S$)
where *bot-scene* ≡ *bot*

lemma *uminus-scene-twice*: $-\ (-\ (X :: 's\ scene)) = X$
  by (*transfer*, *simp*)

lemma *scene-override-id* [*simp*]: $S_1 \oplus_S S_2$ *on* $\top_S = S_2$
  by (*transfer*, *simp*)

lemma *scene-override-unit* [*simp*]: $S_1 \oplus_S S_2$ *on* $\bot_S = S_1$
  by (*transfer*, *simp*)

lemma *scene-override-commute*: $S_2 \oplus_S S_1$ *on* $(-\ X) = S_1 \oplus_S S_2$ *on* X
  by (*transfer*, *simp*)

lemma *scene-union-incompat*: $\neg\ X \#\#_S\ Y \Longrightarrow X \sqcup_S Y = \bot_S$
  by (*transfer*, *auto*)

lemma *scene-override-union*: $X \#\#_S\ Y \Longrightarrow S_1 \oplus_S S_2\ on\ (X \sqcup_S Y) = (S_1 \oplus_S S_2\ on\ X) \oplus_S S_2\ on\ Y$
by (*transfer*, *auto*)

lemma *scene-union-unit*: $X \sqcup_S \bot_S = X$
by (*transfer*, *simp*)

lemma *idem-scene-union* [*simp*]: $\llbracket$ *idem-scene* A; *idem-scene* B $\rrbracket \Longrightarrow$ *idem-scene* $(A \sqcup_S B)$
apply (*transfer*, *auto*)
apply (*unfold-locales*, *auto*)
apply (*metis overrider.ovr-overshadow-left*)
apply (*metis overrider.ovr-overshadow-right*)
done

lemma *scene-union-annhil*: *idem-scene* $X \Longrightarrow X \sqcup_S \top_S = \top_S$
by (*transfer*, *simp*)

lemma *scene-union-pres-compat*: $\llbracket A \#\#_S B;\ A \#\#_S C \rrbracket \Longrightarrow A \#\#_S (B \sqcup_S C)$
by (*transfer*, *auto*)

lemma *scene-indep-self-compl*: $A \bowtie_S -A$
by (*transfer*, *simp*)

lemma *scene-compat-self-compl*: $A \#\#_S -A$
by (*transfer*, *simp*)

lemma *scene-union-assoc*:
assumes $X \#\#_S\ Y\ X \#\#_S\ Z\ Y \#\#_S\ Z$
shows $X \sqcup_S (Y \sqcup_S Z) = (X \sqcup_S Y) \sqcup_S Z$
using *assms* **by** (*transfer*, *auto*)

lemma *scene-inter-indep*:
assumes *idem-scene* X *idem-scene* Y $X \bowtie_S Y$
shows $X \sqcap_S Y = \bot_S$
using *assms*
unfolding *lens-defs*
apply (*transfer*, *auto*)
apply (*metis* (*no-types*, *hide-lams*) *idem-overrider.ovr-idem overrider.ovr-assoc overrider.ovr-overshadow-right*)
apply (*metis* (*no-types*, *hide-lams*) *idem-overrider.ovr-idem overrider.ovr-overshadow-right*)
done

lemma *scene-union-idem*: $X \sqcup_S X = X$
by (*transfer*, *simp*)

lemma *scene-union-compl*: *idem-scene* $X \Longrightarrow X \sqcup_S - X = \top_S$
by (*transfer*, *auto*)

lemma *scene-inter-idem*: $X \sqcap_S X = X$
by (*simp add*: *inf-scene-def*, *transfer*, *auto*)

lemma *scene-union-commute*: $X \sqcup_S Y = Y \sqcup_S X$
by (*transfer*, *auto*)

lemma *scene-inter-compl*: *idem-scene* $X \Longrightarrow X \sqcap_S - X = \bot_S$
by (*simp add*: *inf-scene-def*, *transfer*, *auto*)

lemma *scene-demorgan1*: $-(X \sqcup_S Y) = -X \sqcap_S -Y$
  by (*simp add*: *inf-scene-def*, *transfer*, *auto*)

lemma *scene-demorgan2*: $-(X \sqcap_S Y) = -X \sqcup_S -Y$
  by (*simp add*: *inf-scene-def*, *transfer*, *auto*)

lemma *scene-compat-top*: *idem-scene* $X \Longrightarrow X \#\#_S \top_S$
  by (*transfer*, *simp*)

instantiation *scene* :: (*type*) *ord*
begin

X is a subscene of Y provided that overriding with first Y and then X can be rewritten using the complement of X.

  definition *less-eq-scene* :: *'a scene* ⇒ *'a scene* ⇒ *bool* **where**
  [*lens-defs*]: *less-eq-scene* $X\ Y = (\forall\ s_1\ s_2\ s_3.\ s_1 \oplus_S s_2$ *on* $Y \oplus_S s_3$ *on* $X = s_1 \oplus_S (s_2 \oplus_S s_3$ *on* $X)$ *on* $Y)$
  definition *less-scene* :: *'a scene* ⇒ *'a scene* ⇒ *bool* **where**
  [*lens-defs*]: *less-scene* $x\ y = (x \le y \wedge \neg\ y \le x)$
instance ..
end

abbreviation *subscene* :: *'a scene* ⇒ *'a scene* ⇒ *bool* (**infix** $\subseteq_S$ *55*)
where *subscene* $X\ Y \equiv X \le Y$

lemma *subscene-refl*: $X \subseteq_S X$
  by (*simp add*: *less-eq-scene-def*)

lemma *subscene-trans*: ⟦ *idem-scene* Y; $X \subseteq_S Y$; $Y \subseteq_S Z$ ⟧ $\Longrightarrow X \subseteq_S Z$
  by (*simp add*: *less-eq-scene-def*, *transfer*, *auto*, *metis* (*no-types*, *hide-lams*) *idem-overrider.ovr-idem*)

lemma *subscene-antisym*: ⟦ *idem-scene* Y; $X \subseteq_S Y$; $Y \subseteq_S X$ ⟧ $\Longrightarrow X = Y$
  apply (*simp add*: *less-eq-scene-def*, *transfer*, *auto*)
  apply (*rule ext*)
  apply (*rule ext*)
  apply (*metis* (*full-types*) *idem-overrider.ovr-idem overrider.ovr-overshadow-left*)
  done

lemma *subscene-copy-def*:
  assumes *idem-scene X idem-scene Y*
  shows $X \subseteq_S Y = (\forall\ s_1\ s_2.\ cp_X\ s_1 \circ cp_Y\ s_2 = cp_Y\ (cp_X\ s_1\ s_2))$
  using *assms*
  by (*simp add*: *less-eq-scene-def fun-eq-iff*, *transfer*, *auto*)

lemma *subscene-eliminate*:
  ⟦ *idem-scene* Y; $X \le Y$ ⟧ $\Longrightarrow s_1 \oplus_S s_2$ *on* $X \oplus_S s_3$ *on* $Y = s_1 \oplus_S s_3$ *on* Y
  by (*metis less-eq-scene-def scene-override-overshadow-left scene-override-idem*)

lemma *scene-bot-least*: $\bot_S \le X$
  unfolding *less-eq-scene-def* **by** (*transfer*, *auto*)

lemma *scene-top-greatest*: $X \le \top_S$
  unfolding *less-eq-scene-def* **by** (*transfer*, *auto*)

lemma *scene-union-ub*: ⟦ *idem-scene* Y; $X \bowtie_S Y$ ⟧ $\Longrightarrow X \le (X \sqcup_S Y)$

by (*simp add*: *less-eq-scene-def*, *transfer*, *auto*)
(*metis* (*no-types*, *hide-lams*) *idem-overrider.ovr-idem overrider.ovr-overshadow-right*)

lemma *scene-le-then-compat*: $\llbracket$ *idem-scene X*; *idem-scene Y*; $X \le Y$ $\rrbracket \Longrightarrow X \#\#_S Y$
unfolding *less-eq-scene-def*
by (*transfer*, *auto*, *metis* (*no-types*, *lifting*) *idem-overrider.ovr-idem overrider-def*)

lemma *indep-then-compl-in*: $A \bowtie_S B \Longrightarrow A \le -B$
unfolding *less-eq-scene-def* **by** (*transfer*, *simp*)

lift-definition *scene-comp* :: *'a scene* ⇒ (*'a* ⟹ *'b*) ⇒ *'b scene* (**infixl** $;_S$ *80*)
is λ *S X a b*. *if* (*vwb-lens X*) *then* put_X *a* (*S* (get_X *a*) (get_X *b*)) *else a*
by (*unfold-locales*, *auto*)

lemma *scene-comp-idem* [*simp*]: *idem-scene S* ⟹ *idem-scene* ($S ;_S X$)
by (*transfer*, *unfold-locales*, *simp-all*)

lemma *scene-comp-lens-indep* [*simp*]: $X \bowtie Y \Longrightarrow (A ;_S X) \bowtie_S (A ;_S Y)$
by (*transfer*, *auto simp add*: *lens-indep.lens-put-comm lens-indep.lens-put-irr2*)

lemma *scene-comp-indep* [*simp*]: $A \bowtie_S B \Longrightarrow (A ;_S X) \bowtie_S (B ;_S X)$
by (*transfer*, *auto*)

12.3 Linking Scenes and Lenses

The following function extracts a scene from a very well behaved lens

lift-definition *lens-scene* :: (*'v* ⟹ *'s*) ⇒ *'s scene* ($\llbracket$-$\rrbracket_\sim$) **is**
λ *X* s_1 s_2. *if* (*mwb-lens X*) *then* $s_1 \oplus_L s_2$ *on X else* s_1
by (*unfold-locales*, *auto simp add*: *lens-override-def*)

lemma *vwb-impl-idem-scene* [*simp*]:
vwb-lens X ⟹ *idem-scene* $\llbracket X \rrbracket_\sim$
by (*transfer*, *unfold-locales*, *auto simp add*: *lens-override-overshadow-left lens-override-overshadow-right*)

lemma *idem-scene-impl-vwb*:
$\llbracket$ *mwb-lens X*; *idem-scene* $\llbracket X \rrbracket_\sim$ $\rrbracket$ ⟹ *vwb-lens X*
apply (*cases mwb-lens X*)
apply (*transfer*, *unfold idem-overrider-def overrider-def*, *auto*)
apply (*simp add*: *idem-overrider-axioms-def override-idem-implies-vwb*)
done

lemma *lens-compat-scene*: $\llbracket$ *mwb-lens X*; *mwb-lens Y* $\rrbracket \Longrightarrow X \#\#_L Y \longleftrightarrow \llbracket X \rrbracket_\sim \#\#_S \llbracket Y \rrbracket_\sim$
by (*auto simp add*: *lens-scene.rep-eq scene-compat.rep-eq lens-defs*)

Next we show some important congruence properties

lemma *zero-lens-scene*: $\llbracket 0_L \rrbracket_\sim = \bot_S$
by (*transfer*, *simp*)

lemma *one-lens-scene*: $\llbracket 1_L \rrbracket_\sim = \top_S$
by (*transfer*, *simp*)

lemma *lens-scene-override*:
mwb-lens X ⟹ $s_1 \oplus_S s_2$ *on* $\llbracket X \rrbracket_\sim = s_1 \oplus_L s_2$ *on X*
by (*transfer*, *simp*)

lemma *lens-indep-scene*:
  assumes *vwb-lens X vwb-lens Y*
  shows $(X \bowtie Y) \longleftrightarrow [\![X]\!]_\sim \bowtie_S [\![Y]\!]_\sim$
  using *assms*
  by (*auto*, (*simp add*: *scene-indep-override*, *transfer*, *simp add*: *lens-indep-override-def*)+)

lemma *lens-indep-impl-scene-indep* [*simp*]:
  $(X \bowtie Y) \Longrightarrow [\![X]\!]_\sim \bowtie_S [\![Y]\!]_\sim$
  by (*transfer*, *auto simp add*: *lens-indep-comm lens-override-def*)

lemma *lens-plus-scene*:
  $[\![$ *vwb-lens X*; *vwb-lens Y*; $X \bowtie Y$ $]\!] \Longrightarrow [\![X +_L Y]\!]_\sim = [\![X]\!]_\sim \sqcup_S [\![Y]\!]_\sim$
  by (*transfer*, *auto simp add*: *lens-override-plus lens-indep-override-def lens-indep-overrideI plus-vwb-lens*)

lemma *subscene-implies-sublens'*: $[\![$ *vwb-lens X*; *vwb-lens Y* $]\!] \Longrightarrow [\![X]\!]_\sim \leq [\![Y]\!]_\sim \longleftrightarrow X \subseteq_L{}' Y$
  by (*simp add*: *lens-defs less-eq-scene-def*, *transfer*, *simp add*: *lens-override-def*)

lemma *sublens'-implies-subscene*: $[\![$ *vwb-lens X*; *vwb-lens Y*; $X \subseteq_L{}' Y$ $]\!] \Longrightarrow [\![X]\!]_\sim \leq [\![Y]\!]_\sim$
  by (*simp add*: *lens-defs less-eq-scene-def*, *auto simp add*: *lens-override-def lens-scene-override*)

lemma *sublens-iff-subscene*:
  assumes *vwb-lens X vwb-lens Y*
  shows $X \subseteq_L Y \longleftrightarrow [\![X]\!]_\sim \leq [\![Y]\!]_\sim$
  by (*simp add*: *assms sublens-iff-sublens' subscene-implies-sublens'*)

Equality on scenes is sound and complete with respect to lens equivalence.

lemma *lens-equiv-scene*:
  assumes *vwb-lens X vwb-lens Y*
  shows $X \approx_L Y \longleftrightarrow [\![X]\!]_\sim = [\![Y]\!]_\sim$
proof
  assume *a*: $X \approx_L Y$
  show $[\![X]\!]_\sim = [\![Y]\!]_\sim$
    by (*meson a assms lens-equiv-def sublens-iff-subscene subscene-antisym vwb-impl-idem-scene*)
next
  assume *b*: $[\![X]\!]_\sim = [\![Y]\!]_\sim$
  show $X \approx_L Y$
    by (*simp add*: *assms b lens-equiv-def sublens-iff-subscene subscene-refl*)
qed

definition *lens-insert* :: $('a \Longrightarrow 'b) \Rightarrow 'b\ scene \Rightarrow 'b\ scene$ ($insert_S$) **where**
lens-insert x A = (*if* $([\![x]\!]_\sim \leq A)$ *then* $[\![x]\!]_\sim \sqcup_S A$ *else A*)

lemma *lens-insert-idem*: $insert_S\ x\ (insert_S\ x\ A) = insert_S\ x\ A$
  apply (*auto simp add*: *lens-insert-def less-eq-scene-def*)
  apply (*transfer*)
  apply (*auto simp add*: *lens-override-overshadow-left*)
  apply (*metis lens-override-overshadow-left*)
  done

Membership operations. These have slightly hacky definitions at the moment in order to cater for *mwb-lens*. See if they can be generalised?

definition *lens-member* :: $('a \Longrightarrow 'b) \Rightarrow 'b\ scene \Rightarrow bool$ (**infix** $\in_S$ *50*) **where**
[*lens-defs*]:
lens-member x A = $((\forall\ s_1\ s_2\ s_3.\ s_1 \oplus_S s_2\ on\ A \oplus_L s_3\ on\ x = s_1 \oplus_S (s_2 \oplus_L s_3\ on\ x)\ on\ A) \land$
  $(\forall\ b\ b'.\ get_x\ (b \oplus_S b'\ on\ A) = get_x\ b'))$

lemma *lens-member-override*: $x \in_S A \Longrightarrow s_1 \oplus_S s_2$ *on* $A \oplus_L s_3$ *on* $x = s_1 \oplus_S (s_2 \oplus_L s_3$ *on* $x)$ *on* A
using *lens-member-def* **by** *force*

lemma *lens-member-put*:
assumes *vwb-lens x idem-scene a* $x \in_S a$
shows $put_x\ s\ v \oplus_S s$ *on* $a = s$
by (*metis* (*full-types*) *assms lens-member-override lens-override-def scene-override-idem vwb-lens.put-eq*)

lemma *lens-member-top*: $x \in_S \top_S$
by (*auto simp add*: *lens-member-def*)

abbreviation *lens-not-member* :: $('a \Longrightarrow 'b) \Rightarrow 'b$ *scene* $\Rightarrow$ *bool* (**infix** $\notin_S$ *50*) **where**
$x \notin_S A \equiv (x \in_S - A)$

lemma *lens-member-get-override* [*simp*]: $x \in_S a \Longrightarrow get_x\ (b \oplus_S b'$ *on* $a) = get_x\ b'$
by (*simp add*: *lens-member-def*)

lemma *lens-not-member-get-override* [*simp*]: $x \notin_S a \Longrightarrow get_x\ (b \oplus_S b'$ *on* $a) = get_x\ b$
by (*simp add*: *lens-member-def scene-override-commute*)

12.4 Function Domain Scene

lift-definition *fun-dom-scene* :: $'a$ *set* $\Rightarrow ('a \Rightarrow 'b::two)$ *scene* (*fds*) **is**
λ *A f g. override-on f g A* **by** (*unfold-locales*, *simp-all add*: *override-on-def fun-eq-iff*)

lemma *fun-dom-scene-empty*: $fds(\{\}) = \bot_S$
by (*transfer*, *simp*)

lemma *fun-dom-scene-union*: $fds(A \cup B) = fds(A) \sqcup_S fds(B)$
by (*transfer*, *auto simp add*: *fun-eq-iff override-on-def*)

lemma *fun-dom-scene-compl*: $fds(- A) = - fds(A)$
by (*transfer*, *auto simp add*: *fun-eq-iff override-on-def*)

lemma *fun-dom-scene-inter*: $fds(A \cap B) = fds(A) \sqcap_S fds(B)$
by (*simp add*: *inf-scene-def fun-dom-scene-union*[*THEN sym*] *fun-dom-scene-compl*[*THEN sym*])

lemma *fun-dom-scene-UNIV*: $fds(UNIV) = \top_S$
by (*transfer*, *auto simp add*: *fun-eq-iff override-on-def*)

lemma *fun-dom-scene-indep* [*simp*]:
$fds(A) \bowtie_S fds(B) \longleftrightarrow A \cap B = \{\}$
by (*transfer*, *auto simp add*: *override-on-def fun-eq-iff*, *meson two-diff*)

lemma *fun-dom-scene-always-compat* [*simp*]: $fds(A)\ \#\#_S\ fds(B)$
by (*transfer*, *simp add*: *override-on-def fun-eq-iff*)

lemma *fun-dom-scene-le* [*simp*]: $fds(A) \subseteq_S fds(B) \longleftrightarrow A \subseteq B$
unfolding *less-eq-scene-def*
by (*transfer*, *auto simp add*: *override-on-def fun-eq-iff*, *meson two-diff*)

Hide implementation details for scenes

lifting-update *scene.lifting*
lifting-forget *scene.lifting*

end

13 Optics Meta-Theory

theory *Optics*
 imports *Lenses Prisms Scenes Dataspace Channel-Type*
begin end

14 State and Lens integration

theory *Lens-State*
imports
 HOL−Library.State-Monad
 Lens-Algebra
begin

Inspired by Haskell's lens package

definition *zoom* :: $('a \Longrightarrow 'b) \Rightarrow ('a, 'c)$ *state* $\Rightarrow ('b, 'c)$ *state* **where**
zoom l m = *State* (λ*b*. *case run-state m* (*lens-get l b*) *of* (*c*, *a*) ⇒ (*c*, *lens-put l b a*))

definition *use* :: $('a \Longrightarrow 'b) \Rightarrow ('b, 'a)$ *state* **where**
use l = *zoom l State-Monad.get*

definition *modify* :: $('a \Longrightarrow 'b) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('b, unit)$ *state* **where**
modify l f = *zoom l* (*State-Monad.update f*)

definition *assign* :: $('a \Longrightarrow 'b) \Rightarrow 'a \Rightarrow ('b, unit)$ *state* **where**
assign l b = *zoom l* (*State-Monad.set b*)

context begin

qualified abbreviation *add l n* ≡ *modify l* (λ*x*. *x* + *n*)
qualified abbreviation *sub l n* ≡ *modify l* (λ*x*. *x* − *n*)
qualified abbreviation *mul l n* ≡ *modify l* (λ*x*. *x* ∗ *n*)
qualified abbreviation *inc l* ≡ *add l 1*
qualified abbreviation *dec l* ≡ *sub l 1*

end

bundle *lens-state-notation* **begin**
 notation *zoom* (**infixr** ▷ *80*)
 notation *modify* (**infix** %= *80*)
 notation *assign* (**infix** .= *80*)
 notation *Lens-State.add* (**infix** += *80*)
 notation *Lens-State.sub* (**infix** −= *80*)
 notation *Lens-State.mul* (**infix** ∗= *80*)
 notation *Lens-State.inc* (- ++)
 notation *Lens-State.dec* (- −−)
end

context includes *lens-state-notation* **begin**

lemma *zoom-comp1*: $l1 \triangleright l2 \triangleright s = (l2 ;_L l1) \triangleright s$
unfolding *zoom-def lens-comp-def*

by (*auto split*: *prod.splits*)

lemma *zoom-zero*[*simp*]: *zero-lens* ▷ *s* = *s*
unfolding *zoom-def zero-lens-def*
by *simp*

lemma *zoom-id*[*simp*]: *id-lens* ▷ *s* = *s*
unfolding *zoom-def id-lens-def*
by *simp*

end

lemma (**in** *mwb-lens*) *zoom-comp2*[*simp*]: *zoom x m* ≫= (λ*a*. *zoom x* (*n a*)) = *zoom x* (*m* ≫= *n*)
unfolding *zoom-def State-Monad.bind-def*
by (*auto split*: *prod.splits simp*: *put-get put-put*)

lemma (**in** *wb-lens*) *use-alt-def*: *use x* = *map-state* (*lens-get x*) *State-Monad.get*
unfolding *State-Monad.get-def use-def zoom-def*
by (*simp add*: *comp-def get-put*)

lemma (**in** *wb-lens*) *modify-alt-def*: *modify x f* = *State-Monad.update* (*update f*)
unfolding *modify-def zoom-def lens-update-def State-Monad.update-def State-Monad.get-def State-Monad.set-def State-Monad.bind-def*
by (*auto*)

lemma (**in** *wb-lens*) *modify-id*[*simp*]: *modify x* (λ*x*. *x*) = *State-Monad.return* ()
unfolding *lens-update-def modify-alt-def*
by (*simp add*: *get-put*)

lemma (**in** *mwb-lens*) *modify-comp*[*simp*]: *bind* (*modify x f*) (λ-. *modify x g*) = *modify x* (*g* ∘ *f*)
unfolding *modify-def*
by *simp*

end

Acknowledgements. This work is partly supported by EU H2020 project *INTO-CPS*, grant agreement 644047. http://into-cps.au.dk/. We would also like to thank Prof. Burkhart Wolff and Dr. Achim Brucker for their generous and helpful comments on our work, and particularly their invaluable advice on Isabelle mechanisation and ML coding.

References

[1] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[2] S. Fischer, Z. Hu, and H. Pacheco. A clear picture of lens laws. In *MPC 2015*, pages 215–223. Springer, 2015.

[3] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.

[4] J. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[5] 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.

[6] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC)*, volume 9965 of *LNCS*. Springer, 2016.

[7] M. Hofmann, B. Pierce, and D. Wagner. Symmetric lenses. In *POPL*, pages 371–384. IEEE, 2011.

[8] M. Pickering, J. Gibbons, and N. Wu. Profunctor optics: Modular data accessors. *The Art, Science, and Engineering of Programming*, 1(2), 2017.

[9] N. Schirmer and M. Wenzel. State spaces – the locale way. In *SSV 2009*, volume 254 of *ENTCS*, pages 161–179, 2009.