This is a repository copy of Optics in Isabelle/HOL.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/117267/

Version: Published Version

Article:

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.
Optics in Isabelle

Simon Foster and Frank Zeyda
University of York, UK

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

June 1, 2017

Abstract
Lenses provide an abstract interface for manipulating data types through spatially-separated views. They are defined abstractly in terms of two functions, \texttt{get}, the return a value from the source type, and \texttt{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 paper [4], which shows how lenses can be used to unify heterogeneous representations of state-spaces in formalised programs.

Contents
1 Interpretation Tools ........................................... 2
1.1 Interpretation Locale ........................................ 2
2 Types of Cardinality 2 or Greater .......................... 3
3 Core Lens Laws .................................................. 4
3.1 Lens Signature .................................................. 4
3.2 Weak Lenses ..................................................... 4
3.3 Well-behaved Lenses ......................................... 5
3.4 Mainly Well-behaved Lenses ................................. 6
3.5 Very Well-behaved Lenses ................................... 6
3.6 Ineffectual Lenses ............................................ 6
3.7 Bijective Lenses .............................................. 7
3.8 Lens Independence ........................................... 8
4 Lens Algebraic Operators .................................. 8
4.1 Lens Composition, Plus, Unit, and Identity .......... 9
4.2 Closure Properties ........................................... 10
4.3 Composition Laws .......................................... 12
4.4 Independence Laws ......................................... 12
4.5 Algebraic Laws ............................................... 14
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)

end

end
2 Types of Cardinality 2 or Greater

theory Two
imports 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

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 identifies the core algebraic hierarchy of lens classes, including laws for well-behaved, very well-behaved, and bijective lenses [3, 1, 5].

record ('a, 'b) lens =
  lens-get :: 'b ⇒ 'a (get1)
  lens-put :: 'b ⇒ 'a ⇒ 'b (put1)

A lens $X : V \rightarrow S$, for source type $S$ and view type $V$, identifies $V$ with a subregion of $S$ [3, 2], 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 $get_X : S \rightarrow V$ that extracts a view from a source, and $put_X : S \Rightarrow V \Rightarrow S$ that updates a view within a given source.

named-theorems lens-defs

definition lens-create :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b (create1) where
  [lens-defs]: create_X v = put_X undefined 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.

3.2 Weak Lenses

Weak lenses are the least constrained class of lenses in our algebraic hierarchy. They simply require that the PutGet law [2, 1] is satisfied, meaning that $get$ is the inverse of $put$. 
locale weak-lens =  
  fixes x :: 'a ⇒ 'b (structure)  
  assumes put-get: get (put σ v) = v
begin

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)

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 update :: ('a ⇒ 'a) ⇒ ('b ⇒ 'b) where  
[lens-defs]: update f σ = put σ (f (get σ))

lemma get-update: get (update f σ) = f (get σ)  
  by (simp add: put-get update-def)

lemma view-determination: put σ u = put ̺ v =⇒ u = v  
  by (metis put-get)

lemma put-inj: inj (put σ)  
  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 [2, 1] 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)

lemma put-surjectivity: ∃ ̺ v. put ̺ v = σ  
  using get-put by blast

lemma source-stability: ∃ v. put σ v = σ  
  using get-put by auto
end

declare wb-lens.get-put [simp]

lemma wb-lens-weak [simp]: wb-lens x =⇒ weak-lens x  
  by (simp-all 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 o g) σ
    by (simp add: put-get put-put update-def)

end
declare mwb-lens.put-put [simp]

lemma mwb-lens-weak [simp]:
  mwb-lens x ⇒ weak-lens x
  by (simp add: mwb-lens-def)

3.5 Very Well-behaved Lenses

Very well-behaved lenses combine all three laws, as in the literature [2, 1].

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

  lemma source-determination: get σ = get φ ⇒ put σ v = put φ v ⇒ σ = φ
    by (metis get-put put-put)

  lemma put-eq:
    [ get σ = k; put σ u = put φ v ] ⇒ put φ k = σ
    by (metis get-put put-put)

end

lemma vwb-lens-wb [simp]: vwb-lens x ⇒ wb-lens x
  by (simp-all add: vwb-lens-def)

lemma vwb-lens-mwb [simp]: vwb-lens x ⇒ mwb-lens x
  by (simp-all add: vwb-lens-def)

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-inf)
qed

lemma ineffectual-const-get:
  \exists v. \forall \sigma. \text{get } \sigma = v
by (metis create-get lens-create-def put-inf)
end

abbreviation eff-lens X ≡ (weak-lens X \land (\neg \text{ief-lens } X))

3.7 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 [2, 1].

locale bij-lens = weak-lens +
  assumes strong-get-put: put \sigma (\text{get } \rho) = \rho
begin

sublocale vwb-lens
proof
  fix \sigma v u
  show put \sigma (\text{get } \sigma) = \sigma
    by (simp add: strong-get-put)
  show put (put \sigma v) u = put \sigma u
    by (metis put-get strong-get-put)
qed

lemma put-surj: surj (put \sigma)
  by (metis strong-get-put surj-def)

lemma put-bij: bij (put \sigma)
  by (simp add: bijI put-inj put-surj)

lemma put-is-create: put \sigma v = create v
  by (metis create-get strong-get-put)

lemma get-create: create (\text{get } \sigma) = \sigma
  by (metis put-get put-is-create source-stability)
end

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

lemma bij-lens-weak [simp]:
  bij-lens x \implies weak-lens x
  by (simp-all add: bij-lens-def)

lemma bij-lens-vwb [simp]: bij-lens x \implies vwb-lens x
  by (unfold-locales, simp-all add: bij-lens.put-is-create)
3.8 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 $X :: 'a \Rightarrow 'c$ and $Y :: 'b \Rightarrow 'c$
  assumes lens-put-comm: lens-put $X$ (lens-put $Y$ $\sigma$ $v$) $u$ = lens-put $Y$ (lens-put $X$ $\sigma$ $u$) $v$
  and lens-put-irr1: lens-get $X$ (lens-put $Y$ $\sigma$ $v$) = lens-get $X$ $\sigma$
  and lens-put-irr2: lens-get $Y$ (lens-put $X$ $\sigma$ $u$) = lens-get $Y$ $\sigma$

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

```lemma
lens-indep1:
\[
\land \ u \ v \ \sigma. \ lens-put \ x \ (lens-put \ y \ \sigma \ v) \ u = lens-put \ y \ (lens-put \ x \ \sigma \ u) \ v;
\land \ v \ \sigma. \ lens-get \ x \ (lens-put \ y \ \sigma \ v) = lens-get \ x \ \sigma;
\land \ u \ \sigma. \ lens-get \ y \ (lens-put \ x \ \sigma \ u) = lens-get \ y \ \sigma \ \] \implies \ x \ \bowtie \ y
by (simp add: lens-indep-def)
```

Lens independence is symmetric.

```lemma
lens-indep-sym: $x \bowtie \ y \implies y \bowtie \ x$
  by (simp add: lens-indep-def)
```

```lemma
lens-indep-comm:
\[
x \bowtie \ y \implies lens-put \ x \ (lens-put \ y \ \sigma \ v) \ u = lens-put \ y \ (lens-put \ x \ \sigma \ u) \ v\]
  by (simp add: lens-indep-def)
```

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

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 [4]. Lens composition, illustrated in Figure 3, constructs a lens by composing the source of one lens with the view of another.

\[ \text{definition lens-comp :: } (\text{'}a \Rightarrow \text{'}b) \Rightarrow (\text{'}b \Rightarrow \text{'}c) \Rightarrow (\text{'}a \Rightarrow \text{'c}) \text{ (infixr :L 80) where} \]
\[ \text{lens-defs: lens-comp } X Y = (\text{'} \text{lens-get = lens-get } Y \circ \text{lens-get } X \text{', lens-put = } (\lambda \sigma v. \text{lens-put } X \sigma (\text{lens-put } Y (\text{lens-get } X \sigma v)) \text{')}) \]$\

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.

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

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.

\[ \text{definition lens-prod :: } (\text{'}a \Rightarrow \text{'}c) \Rightarrow (\text{'}b \Rightarrow \text{'}d) \Rightarrow (\text{'}a \times \text{'}b \Rightarrow \text{'}c \times \text{'}d) \text{ (infixr ×L 85) where} \]
\[ \text{lens-defs: lens-prod } X Y = (\text{'} \text{lens-get = map-prod } \text{get}_{X Y} \text{ get}_{Y X} \text{', lens-put = } (\lambda \sigma u v. \text{lens-put } X \sigma (\text{lens-put } Y \sigma u v) u)) \]$\

The \textbf{fst} and \textbf{snd} lenses project the first and second elements, respectively, of a product source type.

\[ \text{definition fst-lens :: } \text{'} a \Rightarrow \text{'a } \times \text{'} b \text{ (fst}_L \text{) where} \]
\[ \text{lens-defs: } \text{fst}_L = (\text{'} \text{lens-get = fst, lens-put = } (\lambda \sigma g u. (u, g)) \text{'}) \]$\

\[ \text{definition snd-lens :: } \text{'b } \Rightarrow \text{'a } \times \text{'} b \text{ (snd}_L \text{) where} \]
\[ \text{lens-defs: } \text{snd}_L = (\text{'} \text{lens-get = snd, lens-put = } (\lambda \sigma g u. (\sigma, u)) \text{'}) \]$\

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

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

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

\[ \text{abbreviation swap-lens :: } \text{'a } \times \text{'} b \Rightarrow \text{'b } \times \text{'a} \text{ (swap}_{L} \text{) where} \]
\[ \text{swap}_{L} \equiv \text{snd}_L +_L \text{fst}_L \]

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.

\[ \text{definition zero-lens :: unit } \Rightarrow \text{'a} \text{ (0}_{L} \text{) where} \]
The identity lens is a bijective lens where the source and view type are the same.

definition id-lens :: 'a ⇒ 'a (1_L) where
    [lens-defs]: 1_L = (| lens-get = id, lens-put = (λ σ. 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 ⇒ 'c) ⇒ ('b ⇒ 'c) ⇒ 'a ⇒ '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 override uses a lens to replace part of a source type with a given value for the corresponding view.

definition lens-override :: 'a ⇒ 'a ⇒ ('b ⇒ 'a) (⊕_L on - [95,0,96] 95) where
    [lens-defs]: S₁ ⊕_L S₂ on X = put_X S₁ (get_X S₂) (inv_L)

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

definition lens-inv :: ('a ⇒ 'b) ⇒ ('b ⇒ 'a) (inv_L) where
    [lens-defs]: lens-inv x = (| lens-get = create_x, lens-put = λ σ. get_x |)

### 4.2 Closure Properties

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

**lemma** comp-wb-lens: [ wb-lens x; wb-lens y ] ⇒ wb-lens (x ;_L y)
    by (unfold-locales, simp-all add: lens-comp-def)

**lemma** comp-mwb-lens: [ mwb-lens x; mwb-lens y ] ⇒ mwb-lens (x ;_L y)
    by (unfold-locales, simp-all add: lens-comp-def)

**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: \[ \text{vwb-lens} \; x; \; \text{vwb-lens} \; y \implies \text{vwb-lens} \; (x +_L y) \]
by (unfold-locales, simp-all add: lens-comp-def)

lemma unit-ief-lens: \text{ief-lens} \; 0
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 \text{mwb-lens} \; x; \; \text{mwb-lens} \; y \; x \bowtie y
shows \text{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 \text{wb-lens} \; x; \; \text{wb-lens} \; y \; x \bowtie y
shows \text{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:
assumes \text{vwb-lens} \; x; \; \text{vwb-lens} \; y \; x \bowtie y
shows \text{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 prod-mwb-lens:
\[ \text{mwb-lens} \; X; \; \text{mwb-lens} \; Y \implies \text{mwb-lens} \; (X \times_L Y) \]
by (unfold-locales, simp-all add: lens-prod-def prod.case-eq-if)

lemma prod-wb-lens:
\[ \text{wb-lens} \; X; \; \text{wb-lens} \; Y \implies \text{wb-lens} \; (X \times_L Y) \]
by (unfold-locales, simp-all add: lens-prod-def prod.case-eq-if)

lemma prod-vwb-lens:
\[ \text{vwb-lens} \; X; \; \text{vwb-lens} \; Y \implies \text{vwb-lens} \; (X \times_L Y) \]
by (unfold-locales, simp-all add: lens-prod-def prod.case-eq-if)

lemma prod-bij-lens:
\[ \text{bij-lens} \; X; \; \text{bij-lens} \; Y \implies \text{bij-lens} \; (X \times_L Y) \]
by (unfold-locales, simp-all add: lens-prod-def prod.case-eq-if)

lemma fst-vwb-lens: \text{vwb-lens} \; \text{fst}_L
by (unfold-locales, simp-all add: fst-lens-def prod.case-eq-if)

lemma snd-vwb-lens: \text{vwb-lens} \; \text{snd}_L
by (unfold-locales, simp-all add: snd-lens-def prod.case-eq-if)
lemma \textit{id-bij-lens}: \textit{bij-lens} \ 1_L \\
by \ (\text{unfold-locales}, \ simp-all \ add: \ \textit{id-lens-def})

lemma \textit{inv-id-lens}: \textit{inv}_L \ 1_L = 1_L \\
by \ (\text{auto \ simp \ add: \ \textit{lens-inv-def} \ \textit{id-lens-def} \ \textit{lens-create-def})

lemma \textit{lens-inv-bij}: \textit{bij-lens} \ X \Rightarrow \textit{bij-lens} \ (\textit{inv}_L \ X) \\
by \ (\text{unfold-locales}, \ simp-all \ add: \ \textit{lens-inv-def} \ \textit{lens-create-def})

lemma \textit{swap-bij-lens}: \textit{bij-lens} \ \textit{swap}_L \\
by \ (\text{unfold-locales}, \ simp-all \ add: \ \textit{lens-plus-def} \ \textit{prod.eq-if} \ \textit{fst-lens-def} \ \textit{snd-lens-def})

\subsection*{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 \textit{lens-comp-assoc}: \ (X ;_L Y) ;_L Z = X ;_L (Y ;_L Z) \\
by \ (\text{auto \ simp \ add: \ \textit{lens-comp-def})

lemma \textit{lens-comp-left-id} [simp]: \ 1_L ;_L X = X \\
by \ (\text{simp \ add: \ \textit{id-lens-def} \ \textit{lens-comp-def})

lemma \textit{lens-comp-right-id} [simp]: \ X ;_L 1_L = X \\
by \ (\text{simp \ add: \ \textit{id-lens-def} \ \textit{lens-comp-def})

lemma \textit{lens-comp-anhil} [simp]: \textit{wb-lens} \ X \Rightarrow 0_L ;_L X = 0_L \\
by \ (\text{simp \ add: \ \textit{zero-lens-def} \ \textit{lens-comp-def} \ \textit{comp-def})

\subsection*{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 \textit{zero-lens-indep} [simp]: \ 0_L \bowtie X \\
by \ (\text{auto \ simp \ add: \ \textit{zero-lens-def} \ \textit{lens-indep-def})

lemma \textit{zero-lens-indep'} [simp]: \ X \bowtie 0_L \\
by \ (\text{auto \ simp \ add: \ \textit{zero-lens-def} \ \textit{lens-indep-def})

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

lemma \textit{lens-indep-quasi-irrefl}: \ [ \ \textit{wb-lens} \ x; \ \textit{eff-lens} \ x \ ] \Rightarrow \neg \ (x \bowtie x) \\
by \ (\text{auto \ simp \ add: \ \textit{lens-indep-def} \ \textit{ief-lens-def} \ \textit{ief-lens-axioms-def}, \ \textit{metis \ \textit{full-types} \ \textit{wb-lens.get-put}})

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

lemma \textit{lens-indep-left-comp} [simp]: \\
[ \ [ \ \textit{mwb-lens} \ z; \ x \bowtie y \ ] \Rightarrow (x ;_L z) \bowtie (y ;_L z) \ ] \\
apply \ (\text{rule \ \textit{lens-indepI}) \\
apply \ (\text{auto \ simp \ add: \ \textit{lens-comp-def}) \\
apply \ (\text{simp \ add: \ \textit{lens-indep-comm}) \\
apply \ (\text{simp \ add: \ \textit{lens-indep-sym}) \\
de\end{document}

12
lemma lens-indep-right-comp:
\[ y \bowtie z \implies (x ;_L y) \bowtie (x ;_L z) \]
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]:
\[ y \bowtie z \implies (x ;_L y) \bowtie z \]
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 \implies x \bowtie (y ;_L z)
by (simp add: lens-indep-left-ext lens-indep-sym)

lemma lens-comp-indep-cong-left:
\[ \llbracket \text{mwb-lens } Z ; X ;_L Z \bowtie Y ;_L Z \rrbracket \implies X \bowtie Y \]
apply (rule lens-indepI)
apply (rename-tac u v \sigma)
apply (drule-tac u = u and v = v and \sigma = createZ \sigma in lens-indep-comm)
apply (simp add: lens-comp-def)
apply (meson mwb-lens-weak weak-lens.view-determination)
apply (rename-tac v \sigma)
apply (drule-tac v = v and \sigma = createZ \sigma in lens-indep-get)
apply (simp add: lens-comp-def)
apply (rename-tac u \sigma)
apply (drule lens-indep-sym)
apply (rename-tac u \sigma)
apply (drule-tac v = u and \sigma = createZ \sigma in lens-indep-get)
apply (simp add: lens-comp-def)
done

lemma lens-comp-indep-cong:
\[ \text{mwb-lens } Z \implies (X ;_L Z) \bowtie (Y ;_L Z) \iff X \bowtie Y \]
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]:
fstL \bowtie sndL
by (simp add: lens-indep-def fst-lens-def snd-lens-def)

lemma snd-fst-lens-indep [simp]:
sndL \bowtie fstL
by (simp add: lens-indep-def fst-lens-def snd-lens-def)

lemma split-prod-lens-indep:
assumes mwb-lens X
shows (fstL ;_L X) \bowtie (sndL ;_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]: \[ X \bowtie Z ; Y \bowtie Z \rrbracket \implies (X +_L Y) \bowtie Z \]
apply (rule lens-indepI)
apply (simp 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]:
\[
\begin{align*}
X \cong Y; X \cong Z \Rightarrow X \cong Y +_L Z
\end{align*}
\]
by (auto intro: lens-indep-sym plus-pres-lens-indep)

Lens independence is preserved by product.

lemma lens-indep-prod:
\[
\begin{align*}
X_1 \cong X_2; Y_1 \cong Y_2 \Rightarrow X_1 \times_L Y_1 \cong X_2 \times_L Y_2
\end{align*}
\]
apply (rule lens-indepI)
apply (auto simp add: lens-prod-def prod case-eq-if lens-indep-comm map-prod-def)
apply (simp add: lens-indep-sym)
done

4.5 Algebraic Laws

Lens plus distributes to the right through composition.

lemma plus-lens-distr:
\[
\begin{align*}
\text{mwb-lens } Z \Rightarrow (X +_L Y) :_L Z = (X :_L Z) +_L (Y :_L Z)
\end{align*}
\]
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:
\[
\begin{align*}
\text{wb-lens } y \Rightarrow \text{fst}_L (x +_L y) = x
\end{align*}
\]
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:
\[
\begin{align*}
\text{wb-lens } x; x \cong y \Rightarrow \text{snd}_L (x +_L y) = y
\end{align*}
\]
apply (simp add: snd-lens-def lens-plus-def lens-comp-def comp-def)
apply (subst lens-indep-comm)
done

The swap lens switches over a summation.

lemma lens-plus-swap:
\[
\begin{align*}
X \cong Y \Rightarrow \text{swap}_L (X +_L Y) = (Y +_L X)
\end{align*}
\]
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)

14
The product lens can be rewritten as a sum lens.

**lemma** prod-as-plus: \( X \times_L Y = X;_L \mathit{fst}_L +_L Y;_L \mathit{snd}_L \)

by (auto simp add: lens-prod-def \( \mathit{fst}_L \)-def \( \mathit{snd}_L \)-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 \( \mathit{id}_L \)-def)

**lemma** prod-lens-comp-plus:
\( X_2 \bowtie Y_2 \Rightarrow ((X_1 \times_L Y_1);_L (X_2 +_L Y_2)) = (X_1;_L X_2) +_L (Y_1;_L Y_2) \)

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 \( Y \Rightarrow (X;_L Y)/_L Y = X \)

by (simp add: lens-quotient-def lens-comp-def)

**lemma** lens-quotient-id: weak-lens \( X \Rightarrow (X/;_L X) = 1_L \)

by (force simp add: lens-quotient-def \( \mathit{id}_L \)-def)

**lemma** lens-quotient-id-denom: \( X/;_L 1_L = X \)

by (simp add: lens-quotient-def \( \mathit{id}_L \)-def lens-create-def)

**lemma** lens-quotient-unit: weak-lens \( X \Rightarrow (0/;_L X) = 0_L \)

by (simp add: lens-quotient-def \( \mathit{zero}_L \)-def)

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** substlens :: \((a \Rightarrow \ \mathit{'}c\Rightarrow \ (a \Rightarrow \mathit{'}b\Rightarrow \mathit{'}c) \Rightarrow \mathit{bool}) \) (infix \( \subseteq_L \) 55) where
[lens-defs]: substlens \( X \) \( Y \) \( = \exists \ Z :: (a, b \ \mathit{lens} \ \mathit{wblens} Z \wedge X = Z;_L Y) \)

Various lens classes are downward closed under the sublens relation.

**lemma** substlens-pres-mwb:
\( \exists \ \mathit{mwb} \)-lens \( Y \); \( X \subseteq_L Y \) \( \Rightarrow \mathit{mwb} \)-lens \( X \)

by (unfold-locales, auto simp add: substlens-def lens-comp-def)

**lemma** substlens-pres-wb:
\( \exists \ \mathit{wb} \)-lens \( Y \); \( X \subseteq_L Y \) \( \Rightarrow \mathit{wb} \)-lens \( X \)

by (unfold-locales, auto simp add: substlens-def lens-comp-def)

**lemma** substlens-pres-vwb:

15
Sublens is a preorder as the following two theorems show.

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

**lemma sublens-trans [trans]:**
\[ X \subseteq_L Y; Y \subseteq_L Z \implies X \subseteq_L Z \]
**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 ex1)
**apply** (simp add: lens-comp-assoc)
**using** comp-vwb-lens **apply** blast

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:**
\[ \forall X. \text{vwb-lens } X \implies 0_L \subseteq_L X \]
**using** sublens-def unit-vwb-lens **by** fastforce

**lemma sublens-greatest:**
\[ \forall X. \text{vwb-lens } X \implies X \subseteq_L 1_L \]
**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:**
\[ X \subseteq_L Y; Y \subseteq_L X \implies \text{put}_X (\text{put}_Y \sigma v) u = \text{put}_X \sigma u \]
**by** (auto simp add: sublens-def lens-comp-def)

**lemma sublens-obs-get:**
\[ X \subseteq_L Y; Y \bowtie Z \implies \text{get}_Y (\text{put}_X \sigma v) = \text{get}_Y (\text{put}_X \circ v) \]
**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:**
\[ X \subseteq_L Y; Y \bowtie Z \implies X \bowtie Z \]
**apply** (auto intro!: lens-indepI simp add: sublens-def lens-comp-def lens-indep-comm)
**apply** (simp add: lens-indep-sym)

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:**
\[ X \subseteq_L Y \implies \text{mwb-lens } (X / L Y) \]
**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.

\textbf{definition} \textit{lens-equiv} :: (\textit{'}a \Rightarrow \textit{'}c) \Rightarrow (\textit{'}b \Rightarrow \textit{'}c) \Rightarrow \textit{bool} (\textit{infix} \approx_L 51) \textit{ where}

\textbf{lemma} \textit{lens-equivI} [\textit{intro}]:
\[
[X \subseteq_L Y; Y \subseteq_L X] \Rightarrow X \approx_L Y
\]
by (simp add: lens-equiv-def)

\textbf{lemma} \textit{lens-equiv-refl}:
\[
X \approx_L X
\]
by (simp add: lens-equiv-def sublens-refl)

\textbf{lemma} \textit{lens-equiv-sym}:
\[
X \approx_L Y = \Rightarrow Y \approx_L X
\]
by (simp add: lens-equiv-def)

\textbf{lemma} \textit{lens-equiv-trans} [\textit{trans}]:
\[
[X \approx_L Y; Y \approx_L Z] \Rightarrow X \approx_L Z
\]
by (auto intro: sublens-trans simp add: lens-equiv-def)

5.3 Further Algebraic Laws

This law explains the behaviour of lens quotient.

\textbf{lemma} \textit{lens-quotient-comp}:
\[
[X \subseteq_L Y; Y \subseteq_L X] \Rightarrow (X /L Y) \subseteq_L Z
\]
by (auto simp add: lens-quotient-def sublens-def)

Plus distributes through quotient.

\textbf{lemma} \textit{lens-quotient-plus}:
\[
[X \subseteq_L Y; Y \subseteq_L Z] \Rightarrow (X +L Y) /L Z = (X /L Z) +L (Y /L Z)
\]
apply (auto simp add: lens-quotient-def sublens-def)
done

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

\textbf{lemma} \textit{plus-pred-sublens}:
\[
[X \subseteq_L Y; Y \subseteq_L Z; X \bowtie \bowtie Y] \Rightarrow (X +L Y) \subseteq_L Z
\]
apply (auto simp add: sublens-def)
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=(fstL ;_L fstL) +_L (sndL ;_L fstL) +_L sndL 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: 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
apply (auto simp add: lens-plus-def lens-comp-def fst-lens-def snd-lens-def prod.case-eq-if split-beta'[I])
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=(fstL +_L (fstL ;_L sndL)) +_L (sndL ;_L sndL) in exI)
apply (auto)
apply (rule plus-vwb-lens)
apply (simp add: fst-vwb-lens lens-indep-sym plus-vwb-lens snd-vwb-lens)
apply (simp add: lens-indep-sym lens-plus-swap)
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:**
\[ X \triangleright Y \implies X +_L Y \subseteq_L Y +_L X \]
apply (simp add: sublens-def)
apply (rule_tac x=sndL +_L fstL in exI)
apply (auto)
apply (simp add: fst-vwb-lens lens-indep-sym plus-vwb-lens snd-vwb-lens)
done

**Lemma lens-plus-comm:**
\[ X \triangleright Y \implies X +_L Y \approx_L Y +_L X \]
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: wb-lens Y \implies X \subseteq L X + L Y
by (metis fst-lens-plus fst-vwb-lens sublens-def)

lemma lens-plus-right-sublens:
\[ [\text{vwb-lens Y}; Y \sqsupset Z; X \subseteq L Z ] \implies X \subseteq L Y + L Z \]
apply (auto simp add: sublens-def)
apply (rename-tac Z')
apply (rule-tac x=Z'; L snd in exI)
apply (auto)
using comp-vwb-lens snd-vwb-lens apply blast
apply (simp add: lens-comp-assoc snd-lens-plus)
done

lemma lens-plus-mono-left:
\[ [\text{vwb-lens Y}; X \subseteq L Y ] \implies X + L Z \subseteq L Y + L Z \]
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:
\[ [\text{vwb-lens Y}; X \subseteq L Z ] \implies X + L Y \subseteq L Y + L Z \]
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 \implies X ; L Y \subseteq L Y
by (auto simp add: sublens-def)

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 (rule plus-vwb-lens)
apply (simp-all)
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-sublens-cong`:
\[
[ X_1 \subseteq_L X_2; Y_1 \subseteq_L Y_2 ] \implies (X_1 \times_L Y_1) \subseteq_L (X_2 \times_L Y_2)
\]
by (metis prod-lens-comp-plus prod-vwb-lens sublens-def)

**Lemma** `lens-plus-eq-left`:
\[
[ X \approx; Y \approx ] \implies X +_L Y \approx L Y +_L Z
\]
by (meson lens-equiv-def lens-plus-mono-left lens-sublens-pres-indep)

**Lemma** `lens-plus-eq-right`:
\[
[ X \approx; Y \approx ] \implies X +_L Y \approx L X +_L Z
\]
by (meson lens-equiv-def lens-sublens-pres-sym lens-plus-mono-right lens-sublens-pres-indep)

**Lemma** `lens-plus-cong`:
assumes \( X_1 \approx X_2 \), \( X_1 \approx L Y_1 \), \( X_2 \approx L Y_2 \)
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: lens-plus-eq-left)
  moreover have ... \( \approx L Y_1 +_L Y_2 \)
    using lens-equiv-def lens-plus-right substlens-pres-indep
ultimately show \( \ldots \approx \) thesis
using lens-eqv-trans
qed

**Lemma** `prod-lens-sublens-cong`:
\[
[ X_1 \subseteq X_2; Y_1 \subseteq Y_2 ] \implies (X_1 \times Y_1) \subseteq_L (X_2 \times Y_2)
\]
by (auto simp add: sublens-def)

**Lemma** `prod-lens-equiv-cong`:
\[
[ X_1 \approx X_2; Y_1 \approx Y_2 ] \implies (X_1 \times Y_1) \approx_L (X_2 \times Y_2)
\]
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-equiv)
  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: substlens-def)
  apply (rule-tac x=(fstL ;L fstL) +_L (fstL ;L sndL)) +_L (sndL ;L fstL) +_L (sndL ;L sndL)
  by (cases)
  apply (auto)
apply (auto intro!: plus-vwb-lens comp-vwb-lens substlens-pres-lens-equiv-comp)
apply (auto intro!: lens-indepI simp add: lens-comp-def substlens-pres-lens-equiv-comp)
done

20
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=\(((f_{stL} :_L f_{stL}) +_L (f_{stL} :_L s_{ndL})) +_L ((s_{ndL} :_L f_{stL}) +_L (s_{ndL} :_L s_{ndL}))\) in ext)

apply (auto)

apply (auto intro!: plus-vwb-lens comp-vwb-lens fst-vwb-lens snd-vwb-lens lens-indep-right-comp)

apply (auto simp add: lens-comp-def lens-plus-def fst-lens-def snd-lens-def)

apply (auto intro!: lens-indepI simp add: lens-comp-def lens-plus-def fst-lens-def snd-lens-def prod.case-eq-if)

apply (auto simp add: lens-prod-def lens-plus-def lens-comp-def prod.case-eq-if)

apply (rule ext, rule ext, auto simp add: lens-prod-def prod.case-eq-if)

done

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:

\[ \text{bij-lens } X \Longrightarrow \text{inv}_L X : L X = 1_L \]

by (auto simp add: lens-inv-def lens-comp-def comp-def id-lens-def, rule ext, auto)

lemma bij-lens-inv-right:

\[ \text{bij-lens } X \Longrightarrow X :_L \text{inv}_L X = 1_L \]

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:

\[ \text{bij-lens } X \iff X \approx_L 1_L \]

apply (auto simp add: lens-equiv-def sublens-def id-lens-def, unfold-locales)

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:

\[ \begin{array}{c}
\text{bij-lens } X; X \approx_L Y \Longrightarrow \text{bij-lens } Y \\
\end{array} \]

by (meson bij-lens-equiv-id lens-equiv-def sublens-trans)

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:

\[ \text{weak-lens } Y \Longrightarrow Y = X :_L Y \Longrightarrow X = 1_L \]

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 $I_L$, then both of the composed lenses must be bijective.

**Lemma bij-lens-via-comp-id-left:**

\[ \text{wb-lens } X ; \text{wb-lens } Y ; X \circ_L Y = I_L \implies \text{bij-lens } X \]
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
apply fastforce
apply (metis select-convs(1) select-convs(2) wb-lens-weak weak-lens)
put-get
apply fastforce
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:**

\[ \text{wb-lens } X ; \text{wb-lens } Y ; X \circ_L Y = I_L \implies \text{bij-lens } Y \]
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 $Z X$ = $Z$
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
apply (simp add: bij-lens-inv-left lens-comp-assoc[THEN sym])
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 $X \approx_L Y \iff (\exists Z. \text{bij-lens } Z \land X = Z \circ_L Y)$
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 vw-lens-wb)
apply (metis assms lens-comp-assoc lens-id-unique)
using lens-equiv-via-bij apply blast

done

5.5 Lens Override Laws

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

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

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

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

**lemma** lens-override-plus:
\[ X \cong Y \implies S_1 \oplus_L S_2 \text{ on } (X +_L Y) = (S_1 \oplus_L S_2 \text{ on } X) \oplus_L S_2 \text{ on } Y \]
by (simp add: lens-indep-comm lens-override-def lens-plus-def)

end

6 Lens Instances

**theory** Lens-Instances
**imports** Lens-Order
**keywords** alphabet :: thy-decl-block
begin

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

6.1 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 :: \( 'a \Rightarrow (\tau::two \Rightarrow (\tau \Rightarrow \nu)) \) where
[lens-defs]: fun-lens x = (\lambda f. f x, lens-get = (\lambda f. f x), lens-put = (\lambda f u. f(x := u)) )

**lemma** fun-wb-lens: wb-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 \cong fun-lens y \iff 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

6.2 Function Range Lens

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

definition fun-ran-lens :: \( ('c \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \) where
[lens-defs]: fun-ran-lens \( X \equiv \emptyset \) lens-get = \( \lambda s. \text{get}_X \circ \text{get}_Y \) s, lens-put = \( \lambda s v. \text{put}_Y s (\lambda x:'a. \text{put}_X (\text{get}_Y s x) (v x)) \)

lemma fun-ran-mwb-lens: \([ mwb-lens X; mwb-lens Y ] \) \Rightarrow mwb-lens (fun-ran-lens X Y)
by (unfold-locales, auto simp add: fun-ran-lens-def)

lemma fun-ran-wb-lens: \([ wb-lens X; wb-lens Y ] \) \Rightarrow wb-lens (fun-ran-lens X Y)
by (unfold-locales, auto simp add: fun-ran-lens-def)

lemma fun-ran-vwb-lens: \([ vwb-lens X; vwb-lens Y ] \) \Rightarrow vwb-lens (fun-ran-lens X Y)
by (unfold-locales, auto simp add: fun-ran-lens-def)

6.3 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 :: \( 'a \Rightarrow ('b \Rightarrow 'a \Rightarrow 'b) \) where
[lens-defs]: map-lens \( x = \emptyset \) lens-get = \( \lambda f. \text{the} (f x) \), lens-put = \( \lambda f a. f(x \mapsto a) \)

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

6.4 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 :: \( 'a \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \) where
list-pad-out \( xs k \) = \( xs \# \text{replicate} (k + 1 - \text{length} xs) \) 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 :: \( 'a \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \) where
list-augment \( xs k v \) = \( \text{list-pad-out} \) \( xs k \) \( \# v \)

The following function is like op ! but it expressly returns undefined when the list isn’t long enough.

definition nth' :: \( 'a \Rightarrow nat \Rightarrow 'a \) where
nth' \( xs i \) = (if \( \text{length} xs > i \) then \( \text{xs} ! i \) else undefined)

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

lemma list-update-append-lemma1: \( i < \text{length} xs \Rightarrow xs[i := v] @ ys = (xs @ ys)[i := v] \)
by (simp add: list-update-append)

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

We can also prove some laws about our new operators.

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

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

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

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

lemma list-augment-twice:
list-augment $(\text{list-augment} xs i u) j v = \text{list-pad-out} xs (\max i j)[i := u, j := v]$
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

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

lemma list-augment-commute:
i \neq j \implies list-augment (list-augment \sigma j v) i u = list-augment (list-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-augment xs k v \# k = v
by (simp add: list-augment-def list-pad-out-def)

lemma nth'-list-augment: nth'(list-augment xs k v) k = v
by (auto simp add: nth'-def nth-list-augment list-augment-def list-pad-out-def)

We also have it that list-augment cancels itself.

lemma list-augment-same-twice: list-augment (list-augment xs k v) k v = list-augment xs k v
by (simp add: list-augment-def list-pad-out-def)

lemma nth'-list-augment-diff: i \neq j \implies nth'(list-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 :: \nat \Rightarrow ('a::two \Rightarrow 'a list) where
[lens-defs]: list-lens i = (\lambda xs. nth' xs i)
, list-get = (\lambda xs. nth' xs i)
, list-put = (\lambda xs x. list-augment xs i x)
abbreviation \( \text{hd-lens} \equiv \text{list-lens} \, 0 \)

\begin{itemize}
  \item \textbf{definition} \( \text{tl-lens} :: \text{'a list} \Rightarrow \text{'a list} \)
  \begin{align*}
  \text{lens-defs: tl-lens} &= \langle \text{lens-get} = (\lambda \, \text{xs.} \, \text{tl} \, \text{xs}) \right. \\
  \text{, lens-put} &= (\lambda \, \text{xs} \, \text{xs'} \, \text{hd} \, \text{xs} \# \text{xs'}) \rangle
  \end{align*}
\end{itemize}

\begin{itemize}
  \item \textbf{lemma} \( \text{list-mwb-lens} \, \text{mwb-lens} \, \text{tl-lens} \)
  \begin{itemize}
    \item \text{by} (\text{unfold-locales, simp-all add: list-lens-def nth'list-augment list-augment-same-twice})
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \textbf{lemma} \( \text{tail-lens-mwb} \)
  \begin{itemize}
    \item \text{by} (\text{unfold-locales, simp-all add: tl-lens-def})
  \end{itemize}
\end{itemize}

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

\begin{itemize}
  \item \textbf{lemma} \( \text{list-lens-indep} \)
  \begin{itemize}
    \item \( i \neq j \Rightarrow \text{list-lens} \, i \, \ll \ll \, \text{list-lens} \, j \)
    \item \text{by (simp add: list-lens-def lens-indep-def list-augment-commute nth'list-augment-diff})
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \textbf{lemma} \( \text{hd-tl-lens-indep} \) [simp]:
    \begin{itemize}
      \item \( \text{hd-lens} \, \ll \ll \, \text{tl-lens} \)
      \item \text{apply (rule lens-indepI)}
      \item \text{apply (simp-add: list-lens-def tl-lens-def)}
      \item \text{apply (metis hd-conv-nth hd-def length-greater-0-conv list.case(1) nth'-list-augment)}
      \item \text{apply (metis Nitpick.size-list-simp(2) One-nat-def add-Suc-right append.simps(1) append-Nil2 diff-Suc-Suc diff-zero hd-Cons-tl list.inject list.size(1) list-augment-0 list-augment-def list-augment-same-twice list-pad-out-def nth-list-augment replicate.simps(1) replicate.simps(2) tl-Nil)}
      \item \text{done}
    \end{itemize}
\end{itemize}

\subsection{6.5 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 \texttt{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.

\begin{itemize}
  \item \textbf{abbreviation} \( \text{input} \) \( f \equiv (\lambda \, \sigma \, u. \, f \, (\lambda \, - \). \, u) \, \sigma) \)
  \item \textbf{syntax} \( \text{FLDLENS} :: \text{id} \Rightarrow (\text{'}a \Rightarrow \text{'}r) \) (FLDLENS -)
  \item \textbf{translations} \( \text{FLDLENS} \, x \Rightarrow \langle \text{lens-get} = x, \text{lens-put} = \text{CONST fld-put} \, (-update-name \, x) \rangle \)
\end{itemize}

We also introduce the \texttt{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.

\textbf{ML-file} \texttt{Lens-Record.ML}

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

\begin{itemize}
  \item \textbf{named-theorems} \alpha-splits
\end{itemize}

\subsection{6.6 Lens Interpretation}

\begin{itemize}
  \item \textbf{named-theorems} \text{ Lens-Interp-laws}
\end{itemize}
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
end

7 Prisms

theory Prisms
  imports Main
begin
Prisms are like lenses, but they act on sum types rather than product types. For now we do not support many properties about them. 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)
  prism-build :: 'v ⇒ 's (build)

locale wb-prism =
  fixes x :: ('v, 's) prism (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

definition prism-suml :: ('a, 'a + 'b) prism where
prism-suml = (λ v. case v of Inl x ⇒ Some x | - ⇒ None), prism-build = Inl |

lemma wb-prim-suml: wb-prism prism-suml
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

definition prism-diff :: ('a, 's) prism ⇒ ('b, 's) prism ⇒ bool (infix ∇ 50) where
prism-diff X Y = (range build X ∩ range build Y = {})

lemma prism-diff-build: X ∇ Y ⇒ build X u ≠ build Y v
  by (simp add: disjoint-iff-not-equal prism-diff-def)

definition prism-plus :: ('a, 's) prism ⇒ ('b, 's) prism ⇒ ('a + 'b, 's) prism (infixl +P 85) where
X +P Y = (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 \(= (\lambda v . \text{case } v \text{ of } \text{Inl } x \Rightarrow \text{build } X x \mid \text{Inr } y \Rightarrow \text{build } Y y) \) \)

theory Lenses
imports
   Lens-Laws
   Lens-Algebra
   Lens-Order
   Lens-Instances
   Prisms
begin 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


