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, \textit{get}, the return a value from the source type, and \textit{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
5 Order and Equivalence on Lenses

5.1 Sub-lens Relation ........................................... 15
5.2 Lens Equivalence ........................................... 17
5.3 Further Algebraic Laws ...................................... 17
5.4 Bijective Lens Equivalences ............................... 21
5.5 Lens Override Laws ......................................... 23

6 Lens Instances

6.1 Function Lens .................................................. 23
6.2 Function Range Lens ......................................... 24
6.3 Map Lens ....................................................... 24
6.4 List Lens ......................................................... 24
6.5 Record Field Lenses ......................................... 26
6.6 Lens Interpretation ........................................... 26

7 Prisms .............................................................. 27

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)
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.

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)
3 Core Lens Laws

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].

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 \( \text{get}_X : S \Rightarrow V \) that extracts a view from a source, and \( \text{put}_X : S \Rightarrow V \Rightarrow S \) that updates a view within a given source.

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 \( \text{get} \) is the inverse of \( \text{put} \).
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 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]
decclare 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 put-get)

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

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

decclare 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
lemma ineffectual-const-get:
  \exists v. \forall \sigma. get \sigma = v
  by (metis create-get lens-create-def put-inef)
end

abbreviation eff-lens X \equiv (weak-lens X \land \neg 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 fully 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 formulated using the strong GetPut law [2, 1].

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

sublocale vwb-lens
proof
  fix \sigma v u
  show put \sigma (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 (get \sigma) = \sigma
  by (metis put-get put-is-create source-stability)

delclare bij-lens.strong-get-put [simp]
delclare 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 $Y$ $\sigma$ $v$) $u = lens-get X$ $\sigma$
  and lens-put-irr2: lens-get $Y$ (lens-put $X$ $\sigma$ $u$) $v = lens-get Y$ $\sigma$

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

lemma lens-indep1:
  $\forall u v \sigma. lens-put x$ (lens-put $y$ $\sigma$ $v$) $u = lens-put y$ (lens-put $x$ $\sigma$ $u$) $v$;
  $\forall v \sigma. lens-get x$ (lens-put $y$ $\sigma$ $v$) $u = lens-get x$ $\sigma$;
  $\forall u \sigma. lens-get y$ (lens-put $x$ $\sigma$ $u$) $v = lens-get y$ $\sigma$ $\Rightarrow x \bowtie y$
by (simp add: lens-indep-def)

Lens independence is symmetric.

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

lemma lens-indep-comm:
  $x \bowtie y \Rightarrow lens-put x$ (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 $y$ $\sigma$ $v$) $u = lens-get x$ $\sigma$
  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.

definition lens-comp :: ('a ⇒ 'b) ⇒ ('b ⇒ 'c) ⇒ ('a ⇒ 'c) (infixr :: L 80) where
[lens-defs]: lens-comp Y X = (| lens-get = lens-get Y ◦ lens-get X 
   , lens-put = (λ σ v. lens-put X σ (lens-put Y (lens-get X σ) 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 lens-plus :: ('a ⇒ 'c) ⇒ ('b ⇒ 'd) ⇒ ('a × 'b ⇒ 'c × 'd) (infixr +L 75) where
[lens-defs]: X +L Y = (| lens-get = (λ σ. (lens-get X σ, lens-get Y σ)) 
   , lens-put = (λ σ (u, v). lens-put X (lens-put Y σ 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 lens-prod :: ('a ⇒ 'c) ⇒ ('b ⇒ 'd) ⇒ ('a × 'b ⇒ 'c × 'd) (infixr ×L 85) where
[lens-defs]: lens-prod X Y = (| lens-get = map-prod get_X get_Y 
   , lens-put = (λ σ (u, v). put_Y v y (put_X u x, put_Y v y)) |)

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

definition fst-lens :: 'a ⇒ 'a × 'b (fstL) where
[lens-defs]: fstL = (| lens-get = fst, lens-put = (λ (σ, u). (u, g)) |)

definition snd-lens :: 'b ⇒ 'a × 'b (sndL) where
[lens-defs]: sndL = (| lens-get = snd, lens-put = (λ (σ, u). (σ, u)) |)

lemma get-fst-lens [simp]: get_{fstL} (x, y) = x
by (simp add: fst-lens-def)

lemma get-snd-lens [simp]: get_{sndL} (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 swap-lens :: 'a × 'b ⇒ 'b × 'a (swapL) where
swapL ≡ sndL +L fstL

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 ⇒ 'a (ζL) where
The identity lens is a bijective lens where the source and view type are the same.

**definition** \( id-lens :: 'a \Rightarrow 'a \) _where_

\[
\begin{align*}
\text{lens-defs: } 1_L &= \langle \text{lens-get} = \text{id}, \text{lens-put} = (\lambda \sigma. \text{id}) \rangle
\end{align*}
\]

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** \( \text{lens-quotient} :: (\forall a \Rightarrow \forall c) \Rightarrow (\forall b \Rightarrow \forall c) \Rightarrow \forall a \Rightarrow \forall b \) _\text{infixr} /_L 90_

\[
\begin{align*}
\text{lens-defs: } X /_L Y &= \langle \text{lens-get} = \lambda \sigma. \text{get}_X (\text{create}_Y \sigma), \text{lens-put} = \lambda \sigma v. \text{get}_Y (\text{put}_X (\text{create}_Y \sigma) v) \rangle
\end{align*}
\]

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

**definition** \( \text{lens-override} :: 'a \Rightarrow 'a \Rightarrow (\forall b \Rightarrow 'a) \Rightarrow 'a \) _\text{on} \( ['95,0', '96] 95 \) _where_

\[
\begin{align*}
\text{lens-defs: } S_1 \oplus_L S_2 \text{ on } X = \text{put}_X S_1 (\text{get}_X S_2)
\end{align*}
\]

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

**definition** \( \text{lens-inv} :: 'a \Rightarrow 'b \Rightarrow 'a \) _\text{on} \( [95,0,96] 95 \) _where_

\[
\begin{align*}
\text{lens-defs: } \text{lens-inv } x &= \langle \text{lens-get} = \text{create}_x, \text{lens-put} = \lambda \sigma. \text{get}_x \rangle
\end{align*}
\]

### 4.2 Closure Properties

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

**lemma** \( \text{id-wb-lens: wb-lens } 1_L \) _by (unfold-locales, simp-all add: id-lens-def)_

**lemma** \( \text{unit-wb-lens: wb-lens } 0_L \) _by (unfold-locales, simp-all add: zero-lens-def)_

**lemma** \( \text{comp-wb-lens: [ wb-lens } x; \text{wb-lens } y \] } \Rightarrow \text{wb-lens } (x ;_L y) \) _by (unfold-locales, simp-all add: lens-comp-def)_

**lemma** \( \text{comp-mwb-lens: [ mwb-lens } x; \text{mwb-lens } y \] } \Rightarrow \text{mwb-lens } (x ;_L y) \) _by (unfold-locales, simp-all add: lens-comp-def)_

**lemma** \( \text{id-vwb-lens [simp]: vwb-lens } 1_L \) _by (unfold-locales, simp-all add: id-lens-def)_

**lemma** \( \text{unit-vwb-lens [simp]: vwb-lens } 0_L \) _by (unfold-locales, simp-all add: zero-lens-def)_
\[ \text{lemma} \ \text{comp-vwb-lens}: \ \begin{array}{c} [ \text{vwb-lens } x; \text{vwb-lens } y ] \ \Rightarrow \ \text{vwb-lens } (x :L y) \\ \text{by} \ (\text{unfold-locales}, \ \text{simp-all add: lens-comp-def}) \end{array} \]

\[ \text{lemma} \ \text{unit-ief-lens}: \ \begin{array}{c} \text{ief-lens } 0_L \\ \text{by} \ (\text{unfold-locales}, \ \text{simp-all add: zero-lens-def}) \end{array} \]

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

\[ \text{lemma} \ \text{plus-mwb-lens}: \ \begin{array}{c} \text{assumes} \ \text{mwb-lens } x \ \text{mwb-lens } y \ x \sim y \\ \text{shows} \ \text{mwb-lens } (x +_L y) \\ \text{using} \ \text{assms} \\ \text{apply} \ (\text{unfold-locales}) \\ \text{apply} \ (\text{simp-all add: lens-plus-def prod.case eq-if lens-indep-sym}) \\ \text{apply} \ (\text{simp add: lens-indep-comm}) \\ \text{done} \end{array} \]

\[ \text{lemma} \ \text{plus-wb-lens}: \ \begin{array}{c} \text{assumes} \ \text{wb-lens } x \ \text{wb-lens } y \ x \triangleright y \\ \text{shows} \ \text{wb-lens } (x +_L y) \\ \text{using} \ \text{assms} \\ \text{apply} \ (\text{unfold-locales}, \ \text{simp-all add: lens-plus-def}) \\ \text{apply} \ (\text{simp add: lens-indep-sym prod.case eq-if}) \\ \text{done} \end{array} \]

\[ \text{lemma} \ \text{plus-vwb-lens}: \ \begin{array}{c} \text{assumes} \ \text{vwb-lens } x \ \text{vwb-lens } y \ x \triangleright y \\ \text{shows} \ \text{vwb-lens } (x +_L y) \\ \text{using} \ \text{assms} \\ \text{apply} \ (\text{unfold-locales}, \ \text{simp-all add: lens-plus-def}) \\ \text{apply} \ (\text{simp add: lens-indep-sym prod.case eq-if}) \\ \text{done} \end{array} \]

\[ \text{lemma} \ \text{prod-mwb-lens}: \ \begin{array}{c} [ \text{mwb-lens } X; \text{mwb-lens } Y ] \ \Rightarrow \ \text{mwb-lens } (X \times L Y) \\ \text{by} \ (\text{unfold-locales}, \ \text{simp-all add: lens-prod-def prod.case eq-if}) \end{array} \]

\[ \text{lemma} \ \text{prod-wb-lens}: \ \begin{array}{c} [ \text{wb-lens } X; \text{wb-lens } Y ] \ \Rightarrow \ \text{wb-lens } (X \times L Y) \\ \text{by} \ (\text{unfold-locales}, \ \text{simp-all add: lens-prod-def prod.case eq-if}) \end{array} \]

\[ \text{lemma} \ \text{prod-vwb-lens}: \ \begin{array}{c} [ \text{vwb-lens } X; \text{vwb-lens } Y ] \ \Rightarrow \ \text{vwb-lens } (X \times L Y) \\ \text{by} \ (\text{unfold-locales}, \ \text{simp-all add: lens-prod-def prod.case eq-if}) \end{array} \]

\[ \text{lemma} \ \text{prod-bij-lens}: \ \begin{array}{c} [ \text{bij-lens } X; \text{bij-lens } Y ] \ \Rightarrow \ \text{bij-lens } (X \times L Y) \\ \text{by} \ (\text{unfold-locales}, \ \text{simp-all add: lens-prod-def prod.case eq-if}) \end{array} \]

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

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

lemma inv-id-lens: invL 1L = 1L
  by (auto simp add: lens-inv-def id-lens-def lens-create-def)

lemma lens-inv-bij: bij-lens X ⇒ bij-lens (invL X)
  by (unfold-locales, simp-all add: lens-inv-def lens-create-def)

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

4.3 Composition Laws

Lens composition is monoidal, with unit 1L, as the following theorems demonstrate. It also has 0L 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]: 1L ;L X = X
  by (simp add: id-lens-def lens-comp-def)

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

lemma lens-comp-anhil [simp]: wb-lens X ⇒ 0L ;L X = 0L
  by (simp add: zero-lens-def lens-comp-def comp-def)

4.4 Independence Laws

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

lemma zero-lens-indep [simp]: 0L ⊲ X
  by (auto simp add: zero-lens-def lens-indep-def)

lemma zero-lens-indep' [simp]: X ⊲ 0L
  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 ] ⇒ ¬(x ∞ x)
  by (auto simp add: lens-indep-def ief-lens-def ief-lens-axioms-def, metis (full-types) wb-lens.get-pat)

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

lemma lens-indep-left-comp [simp]:
  [ mwb-lens z; x ∞ y ] ⇒ (x ;L z) ∞ (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:
\[ y \bowtie z \implies (x : L y) \bowtie (x : L z) \]
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:
\[[ \text{mwb-lens } Z ; X : L Z \bowtie Y : L Z ] \implies X \bowtie Y \]
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 (rename-tac u σ)
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:
\[ \text{mwb-lens } Z \implies (X : L Z) \bowtie (Y : L Z) \iff X \bowtie Y \]
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]:
\[ \text{fst} \bowtie \text{snd} \]
by (simp add: lens-indep-def fst-lens-def snd-lens-def)

lemma snd-fst-lens-indep [simp]:
\[ \text{snd} \bowtie \text{fst} \]
by (simp add: lens-indep-def fst-lens-def snd-lens-def)

lemma split-prod-lens-indep:
assumes mwb-lens X
shows \((\text{fst}_{L} ; X) \bowtie (\text{snd}_{L} ; X)\)
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]:
\[ [X \bowtie Z ; Y \bowtie Z ] \implies (X +_{L} Y) \bowtie Z \]
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]:
  \[ X \bowtie Y; X \bowtie Z \] \Rightarrow 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:
  \[ X_1 \bowtie X_2; Y_1 \bowtie Y_2 \] \Rightarrow 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 Algebraic Laws

Lens plus distributes to the right through composition.

lemma plus-lens-distr:
   mwb-lens Z = \Rightarrow (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 = \Rightarrow fst_L \; (x +_L y) = x
by (simp add: lens-comp-def lens-plus-def comp-def)

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

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

The swap lens switches over a summation.

lemma lens-plus-swap:
   X \bowtie Y \Rightarrow swap_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)

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: swap-lens-def lens-comp-def)

lemma swap-lens-fst: fst_L \; L \; swap_L = snd_L
by (simp add: lens-comp-def)

lemma swap-lens-snd: snd_L \; L \; swap_L = fst_L
by (simp add: lens-comp-def)

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

**Lemma prod-as-plus:** \( X \times_L Y = X \circ \text{fst}_{L,Y} +_L Y \circ \text{snd}_{L,Y} \)
by (auto simp add: lens-prod-def \text{fst-lens-def} \text{snd-lens-def} \text{lens-comp-def} \text{lens-plus-def})

**Lemma prod-lens-id-equiv:** \( 1_L \times_L 1_L = 1_L \)
by (auto simp add: lens-prod-def \text{id-lens-def})

**Lemma prod-lens-comp-plus:** \( X_2 \circ \text{fst}_{L,Y} +_L Y_2 \Rightarrow (X_1 \circ \text{fst}_{L,Y} +_L Y_2) = (X_1 \circ \text{fst}_{L,Y}) +_L (Y_1 \circ \text{fst}_{L,Y}) \)
by (auto simp add: lens-comp-def \text{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:** \( \text{weak-lens } Y \Rightarrow (X \circ \text{fst}_{L,Y}) /_L Y = X \)
by (simp add: \text{lens-quotient-def} \text{lens-comp-def})

**Lemma lens-quotient-id:** \( \text{weak-lens } X \Rightarrow (X /_L 1_L) = 1_L \)
by (force simp add: \text{lens-quotient-def} \text{id-lens-def})

**Lemma lens-quotient-id-denom:** \( X /_L 1_L = X \)
by (simp add: \text{lens-quotient-def} \text{zero-lens-def})

**Lemma lens-quotient-unit:** \( \text{weak-lens } X \Rightarrow (0_L /_L X) = 0_L \)
by (simp add: \text{lens-quotient-def} \text{zero-lens-def})

end

5 Order and Equivalence on Lenses

theory \text{Lens-Order}
imports \text{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:** \( (\text{a} \Rightarrow \text{b} \Rightarrow \text{c}) \Rightarrow (\text{b} \Rightarrow \text{c}) \Rightarrow \text{bool} \) (infix \( \subseteq_L \) 55) where

[lens-defs]: sublens \( X \ Y = (\exists Z :: (\text{a}, \text{b}) \text{ lens. wb-lens } Z \ \\
\wedge X = Z ;_L Y) \)

Various lens classes are downward closed under the sublens relation.

**Lemma sublens-pres-mwb:** \( \forall \text{mwb-lens } Y; X \subseteq_L Y \Rightarrow \text{mwb-lens } X \)
by (unfold-locales, auto simp add: \text{sublens-def} \text{lens-comp-def})

**Lemma sublens-pres-wb:** \( \forall \text{wb-lens } Y; X \subseteq_L Y \Rightarrow \text{wb-lens } X \)
by (unfold-locales, auto simp add: \text{sublens-def} \text{lens-comp-def})

**Lemma sublens-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]:**
\[ X \subseteq Y \subseteq Z \implies X \subseteq Z \]
apply (auto simp add: sublens-def lens-comp-assoc)
apply (rename_tac Z \(Z_1\) \(Z_2\))
apply (rule-tac \(x=Z_1\); \(Z_2\) 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:**
\[ \text{vwb-lens } X = \implies 0_L \subseteq L X \]
using sublens-def unit-vwb-lens by fastforce

**Lemma sublens-greatest:**
\[ \text{vwb-lens } X = \implies X \subseteq L 1_L \]
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:**
\[ \text{mwb-lens } X; Y \subseteq L X \implies \text{put } X (\text{put } Y \sigma v) u = \text{put } X \sigma u \]
by (auto simp add: sublens-def lens-comp-def)

**Lemma sublens-obs-get:**
\[ \text{mwb-lens } X; Y \subseteq L X \implies \text{get } Y (\text{put } X \sigma v) = \text{get } Y (\text{put } X \circ v) \]
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-indep1 simp add: sublens-def lens-comp-def lens-indep-comm)
apply (simp add: lens-indep-sym)
done

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

**definition** `lens-equiv :: ('a => 'c) => ('b => 'c) => bool (infix \approx_{L} 51)` where

```latex
\text{lens-defs}: \text{lens-equiv } X \ Y = (X \subseteq_{L} Y \land Y \subseteq_{L} X)
```

**lemma** `lens-equivI [intro]`:

\[ [ X \subseteq_{L} Y; Y \subseteq_{L} X ] \implies X \approx_{L} Y \]

by `(simp add: lens-equiv-def)

**lemma** `lens-equiv-refl`:

\[ X \approx_{L} Y \]

by `(simp add: lens-equiv-def sublens-refl)

**lemma** `lens-equiv-sym`:

\[ X \approx_{L} Y = \implies Y \approx_{L} X \]

by `(simp add: lens-equiv-def)

**lemma** `lens-equiv-trans [trans]`:

\[ [ X \approx_{L} Y; Y \approx_{L} Z ] \implies X \approx_{L} Z \]

by `(auto intro: sublens-trans simp add: lens-equiv-def)`

5.3 Further Algebraic Laws

This law explains the behaviour of lens quotient.

**lemma** `lens-quotient-comp`:

\[ [ \text{weak-lens } Y; X \subseteq_{L} Y \implies (X /_{L} Y) /_{L} Y = X \]

by `(auto simp add: lens-quotient-def lens-comp-def sublens-def)`

Plus distributes through quotient.

**lemma** `lens-quotient-plus`:

\[ [ \text{mwb-lens } Z; X \subseteq_{L} Y \implies (X +_{L} Y) /_{L} Z = (X /_{L} Z) +_{L} (Y /_{L} Z) ] \]

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

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

**lemma** `plus-pred-sublens`:

\[ [ \text{mwb-lens } Z; X \subseteq_{L} Y \implies (X +_{L} Y) \subseteq_{L} Z \]

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 \( x = (\text{fst}_L +_L \text{fst}_L, \text{snd}_L +_L \text{snd}_L) \) in exI)  
apply (auto simp add: lens-plus-def lens-comp-def fst-lens-def snd-lens-def prod.case-eq-if split-beta')

**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 \( x = (\text{fst}_L +_L (\text{fst}_L +_L \text{snd}_L), \text{snd}_L +_L \text{snd}_L) \) in exI)  
apply (auto simp add: lens-plus-def lens-comp-def fst-lens-def snd-lens-def prod.case-eq-if split-beta')

**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**:  
\[ X \bowtie Y \implies X +_L Y \subseteq_L Y +_L X \]  
apply (simp add: sublens-def)  
apply (rule-tac \( x = \text{snd}_L +_L \text{fst}_L \) in exI)  
apply (auto simp add: lens-plus-def lens-comp-def lens-plus-swap)

**lemma** **lens-plus-comm**:  
\[ X \bowtie Y \implies X +_L Y \approx_L Y +_L X \]  
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: \( \text{wb-lens } Y \implies X \subseteq L X +_L Y \)
  by (metis fst-lens-plus fst-vwb-lens sublens-def)

lemma lens-plus-right-sublens:
  \[ \forall Y ; X \subseteq L Y \implies X \subseteq L X +_L L Y \]
  apply (auto simp add: sublens-def)
  apply (rename_tac \( Z' \))
  apply (rule_tac x=\( Z' \times L 1 \) in exI)
  apply (simp)
  using comp-vwb-lens snd-vwb-lens apply blast
  apply (simp add: lens-comp-assoc snd-lens-plus)
  done

lemma lens-plus-mono-left:
  \[ \forall Y \supseteq \subseteq Z ; X \subseteq L Y \implies X +_L Z \subseteq L X +_L L Z \]
  apply (auto simp add: sublens-def)
  apply (rename_tac \( Z' \))
  apply (rule_tac x=\( Z' \times L 1 \) in exI)
  apply (auto)
  apply (rule plus-vwb-lens)
  apply (simp-all)
  using id-vwb-lens prod-vwb-lens apply blast
  done

lemma lens-plus-mono-right:
  \[ \forall X \supseteq \subseteq Z ; Y \subseteq L Z \implies X +_L Y \subseteq L X +_L L Z \]
  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]: \( \text{vwb-lens } X \implies X \subseteq L X ; L Y \)
  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 (simp)
  apply (auto)
  apply (auto)
  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`: \[ Y_1 \bowtie Y_2; X_1 \subseteq_L Y_1; X_2 \subseteq_L Y_2 \implies X_1 +_L X_2 \subseteq_L Y_1 +_L Y_2 \]

by (metis `prod-lens-comp-plus` `prod-vwb-lens` `sublens-def`)

**Lemma** `lens-plus-eq-left`: \[ X \bowtie Z; X \approx_L Y \implies X +_L Z \approx_L Y +_L Z \]

by (meson `lens-equiv-def` `lens-plus-mono-left` `sublens-pres-indep`)

**Lemma** `lens-plus-eq-right`: \[ X \bowtie Y; Y \approx_L Z \implies X +_L Y \approx_L X +_L Z \]

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 \( \ldots \approx_L Y_1 +_L Y_2 \)

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`:

\[ X_1 \subseteq L X_2; Y_1 \subseteq L Y_2 \implies (X_1 \times_L Y_1) \subseteq_L (X_2 \times_L Y_2) \]

apply (auto simp add: `sublens-def`)

apply (rename-tac `Z` `Z`)

apply (rule-tac `x`=`Z` \times L `Z` in `ext`)

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`:

\[ X_1 \approx_L X_2; Y_1 \approx_L Y_2 \implies (X_1 \times_L Y_1) \approx_L (X_2 \times_L Y_2) \]

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`:

\( (X_1 +_L X_2) \times (Y_1 +_L Y_2) \approx_L (X_1 \times (Y_1 +_L Y_2)) +_L (X_2 \times (Y_1 +_L Y_2)) \)

**Proof**

(rule `lens-equiv`)

show \( (X_1 +_L X_2) \times (Y_1 +_L Y_2) \subseteq_L (X_1 \times (Y_1 +_L Y_2)) +_L (X_2 \times (Y_1 +_L Y_2)) \)

apply (simp add: `sublens-def`)

apply (rule-tac `x`=(`fst` \_L `fst\_L`) \+ L (`fst\_L` \_L `snd\_L`) \+ L (`snd\_L` \_L `snd\_L`)) 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 intro!: `lens-indepI` simp add: `lens-comp-def` `lens-plus-def` `fst-lens-def` `snd-lens-def` `comp-def`[1])

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

**Done**
show \((X_1 \times_L Y_1) +_L (X_2 \times_L Y_2) \subseteq (X_1 +_L X_2) \times_L (Y_1 +_L Y_2)\)
apply (simp add: sublens-def)
apply (rule-tac \(x=((\text{fst}_L \; ;_L \text{fst}_L) +_L (\text{fst}_L ;_L \text{snd}_L)) +_L ((\text{snd}_L ;_L \text{fst}_L) +_L (\text{snd}_L ;_L \text{snd}_L))\) 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-plus-def lens-comp-def lens-plus-def lens-comp-def lens-indep-right-comp)
apply (auto simp add: lens-comp-def comp-vwb-lens fst-vwb-lens snd-vwb-lens lens-indep-right comp-def)
done

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 \(I_L\).

**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:**
\(\text{bij-lens } X \leftarrow\rightarrow X \approx_L 1_L\)
apply (auto)
apply (rule lens-equivI)
apply (simp-all add: sublens-def)
apply (rule-tac \(x=\text{inv}_L X\) in ext)
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:**
\([\text{bij-lens } X; X \approx_L Y]\) \(\Longrightarrow\) \(\text{bij-lens } Y\)
by (meson bij-lens-equiv-id lens-equiv-def sublens-trans)

We can also show that the identity lens \(I_L\) 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 \((\text{metis} \ \text{select-convs}(1) \ \text{weak-lens.create-get})\)
apply \((\text{metis} \ \text{select-convs}(1) \ \text{select-convs}(2) \ \text{weak-lens.put-get})\)
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:
\[
\begin{align*}
[\ [ \ \text{wb-lens} \ X ; \ \text{wb-lens} \ Y ; \ X ;_L \ Y = 1_L \ ] ] & \implies \text{bij-lens} \ X \\
\text{apply} & (\text{cases} \ Y) \\
\text{apply} & (\text{cases} \ X) \\
\text{apply} & (\text{auto simp add: lens-comp-def comp-def id-lens-def fun-eq-iff}) \\
\text{apply} & (\text{unfold-locales}) \\
\text{apply} & (\text{simp-all}) \\
\text{using} & \text{vwb-lens-wb wb-lens-weak weak-lens.put-get apply fastforce} \\
\text{apply} & (\text{metis select-convs}(1) \ \text{select-convs}(2) \ \text{weak-lens-weak} \ \text{put-get}) \\
\text{done}
\end{align*}
\]

**lemma** bij-lens-via-comp-id-right:
\[
\begin{align*}
[\ [ \ \text{wb-lens} \ X ; \ \text{wb-lens} \ Y ; \ X ;_L \ Y = 1_L \ ] ] & \implies \text{bij-lens} \ Y \\
\text{apply} & (\text{cases} \ Y) \\
\text{apply} & (\text{cases} \ X) \\
\text{apply} & (\text{auto simp add: lens-comp-def comp-def id-lens-def fun-eq-iff}) \\
\text{apply} & (\text{unfold-locale}) \\
\text{apply} & (\text{simp-all}) \\
\text{using} & \text{vwb-lens-wb wb-lens-weak weak-lens.put-get apply fastforce} \\
\text{apply} & (\text{metis select-convs}(1) \ \text{select-convs}(2) \ \text{weak-lens-weak} \ \text{put-get}) \\
\text{done}
\end{align*}
\]

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:
\[
\begin{align*}
\text{assumes} & \ \text{bij-lens} \ Z \ X = Z ;_L \ Y \\
\text{shows} & \ X \approx_L Y \\
\text{using} & \text{assms} \\
\text{apply} & (\text{auto simp add: lens-equiv-def sublens-def}) \\
\text{using} & \text{bij-lens-vwb apply blast} \\
\text{apply} & (\text{rule-tac} \ \text{x=}\text{lens-inv} \ Z \ \text{in} \ \text{exI}) \\
\text{apply} & (\text{auto simp add: lens-comp-assoc bij-lens-inv-left}) \\
\text{using} & \text{bij-lens-vwb lens-inv-bij apply blast} \\
\text{apply} & (\text{simp add: bij-lens-inv-left lens-comp-assoc[THEN sym]}) \\
\text{done}
\end{align*}
\]

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:
\[
\begin{align*}
\text{assumes} & \ \text{weak-lens} \ Y \\
\text{shows} & \ X \approx_L Y \iff (\exists \ Z. \ \text{bij-lens} \ Z \land X = Z ;_L \ Y) \\
\text{apply} & (\text{rule iffI}) \\
\text{apply} & (\text{auto simp add: lens-equiv-def sublens-def lens-id-unique}[1]) \\
\text{apply} & (\text{rename-tac} \ Z_1 \ Z_2) \\
\text{apply} & (\text{rule-tac} \ \text{x=Z}_1 \ \text{in} \ \text{exI}) \\
\text{apply} & (\text{simp}) \\
\end{align*}
\]
apply (subgoal-tac Z₂ :L Z₁ = 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

5.5 Lens Override Laws

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

lemma lens-override-id [simp]:
S₁ ⊕L S₂ on 1_L = S₂
by (simp add: lens-override-def id-lens-def)

lemma lens-override-unit [simp]:
S₁ ⊕L S₂ on 0_L = S₁
by (simp add: lens-override-def zero-lens-def)

lemma lens-override-overshadow:
assumes mwb-lens Y X ⊆ L Y
shows (S₁ ⊕L S₂ on X) ⊕L S₃ on Y = S₁ ⊕L S₃ on Y
using assms by (simp add: lens-override-def sublens-put-put)

lemma lens-override-plus:
X ⊓ Y ⊳ S₁ ⊕L S₂ on (X +ₐ Y) = (S₁ ⊕L S₂ on X) ⊕L S₂ 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 ⇒ ('b::two ⇒ ('a ⇒ 'b)) where
[lens-defs]: fun-lens x = (λ f. f x), lens-put = (λ 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 ⊓ fun-lens y ←→ x ≠ y
proof –
  obtain u v :: 'a::two where u ≠ 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 ⇒ 'b ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'c) ⇒ 'a
where
[lens-defs]: fun-ran-lens X Y = [] fun-get = λ s. get_X ◦ 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)

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 ⇒ ('b ⇒ ('a ⇒ 'b)) where
[lens-defs]: map-lens x = [] lens-get = (λ f. the (f x)), lens-put = (λ f a. f (x ↦ u)) []

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 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 op ! 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 ⇒ xs[i := v] @ ys = (xs @ ys)[i := v]
by (simp add: list-update-append)

lemma list-update-append-lemma2: \( i < \text{length} \, ys \implies xs \odot ys[i := v] = (xs \odot ys)[i + \text{length} \, xs := v] \)
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 \# list-augment xs n y
by (simp add: list-augment-def list-pad-out-def)

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

lemma list-augment-twine:
list-augment (list-augment xs i u) j v = 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 \text{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-commute)

We can also prove that we can always retrieve an element we have added to the list, since \text{list-augment} extends the list when necessary. This isn’t true of \text{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: \( \text{nth}' \, (\text{list-augment} \, xs \, k \, v) \, k = v \)
by (auto simp add: nth'-def nth-list-augment list-augment-def list-pad-out-def)

We also have it that \text{list-augment} cancels itself.

lemma list-augment-same-twine: 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 :: \( \text{nat} \Rightarrow (\text{a}\cdot\text{two} \Rightarrow \text{'}a\,\text{list}) \) where
[lens-defs]: list-lens i = (| lens-get = (\lambda \, xs. nth' zs i)
, lens-put = (\lambda \, xs. list-augment xs i z) |)
abbreviation \( \text{hd-lens} \equiv \text{list-lens} \ 0 \)

definition \( \text{tl-lens} :: 'a \text{ list} \implies 'a \text{ list} \) where
\[ \text{tl-lens} = \lambda \text{xs}. \text{tl} \text{xs} \]
\[ , \text{tl-lens} = \lambda \text{xs} \text{xs}' \. \text{hd} \text{xs} \neq \text{xs}' \]

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)

lemma tail-lens-mwb:
\[ \text{mwb-lens} \text{tl-lens} \]
\by (unfold-locales, simp-all add: tl-lens-def)

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

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

lemma hd-tl-lens-indep [simp]:
\[ \text{hd-lens} \bowtie \text{tl-lens} \]
\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'-list-augment)
\apply (metis (full-types) hd-conv-nth hd-def length-greater-0-conv list.case(1) nth'-def)
\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(4) 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)
\done

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.

abbreviation \( (\text{input}) \text{fld-put} \ f \equiv (\lambda \sigma \ u. \ f (\lambda -. \ u) \sigma) \)

syntax -FLDLENS :: \( \text{id} \Rightarrow ('a \Rightarrow ') \) (FLDLENS -)
translations FDLENS \( x \Rightarrow (| \text{lens-get} = x, \text{lens-put} = \text{CONST fld-put} (\text{-update-name} \ x) |) \)

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.

ML-file Lens-Record.ML

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

named-theorems alpha-splits

6.6 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
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 (match1)
  prism-build :: 'v ⇒ 's (build1)

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 = () prism-match = (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 = () 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) ∣

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


