Article:
Foster, Simon David orcid.org/0000-0002-9889-9514 and Zeyda, Frank (2017) Optics in Isabelle/HOL. Archive of Formal Proofs. pp. 1-28.

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, *get*, the return a value from the source type, and *put* that updates the value. We mechanise the underlying theory of lenses, in terms of an algebraic hierarchy of lenses, including well-behaved and very well-behaved lenses, each lens class being characterised by a set of lens laws. We also mechanise a lens algebra in Isabelle that enables their composition and comparison, so as to allow construction of complex lenses. This is accompanied by a large library of algebraic laws. Moreover we also show how the lens classes can be applied by instantiating them with a number of Isabelle data types. This theory development is based on our recent paper [4], which shows how lenses can be used to unify heterogeneous representations of state-spaces in formalised programs.

Contents

1 Interpretation Tools

theory *Interp*
imports *Main*
begin

1.1 Interpretation Locale

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

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

lemma exists-interp-law:
(∃P. Q P) = (∃P. Q (P o f))
  apply (safe)
```

— Subgoal 1
apply (*rule-tac x = P o inv f* **in** *exI*)
apply (*simp add*: *f-inj*)
— Subgoal 2
apply (*rule-tac x = P o f* **in** *exI*)
apply (*assumption*)
done
end
end

2 Types of Cardinality 2 or Greater

theory *Two*
imports *Real*
begin

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

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

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

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

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

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

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

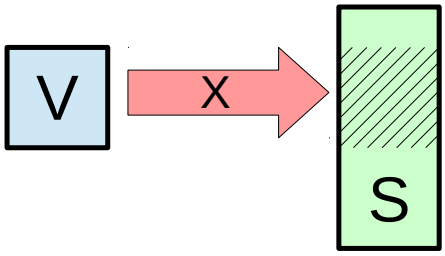

Figure 1: Visualisation of a simple lens

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

end

3 Core Lens Laws

theory *Lens-Laws*
imports
  Two Interp
begin

3.1 Lens Signature

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

record ($'a$, $'b$) *lens* =
  lens-get :: $'b \Rightarrow 'a$ (*get*1)
  lens-put :: $'b \Rightarrow 'a \Rightarrow 'b$ (*put*1)

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

A lens $X : V \Longrightarrow S$, 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 $\mathit{get}_X : S \Rightarrow V$ that extracts a view from a source, and $\mathit{put}_X : S \Rightarrow V \Rightarrow S$ that updates a view within a given source.

named-theorems *lens-defs*

definition *lens-create* :: $('a \Longrightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (*create*1) **where**
[*lens-defs*]: $\mathit{create}_X\ v = \mathit{put}_X\ \mathit{undefined}\ v$

Function $\mathit{create}_X\ v$ 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 :: $'a \Longrightarrow 'b$ (**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* :: $('a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'b)$ **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* ϱ v $\Longrightarrow$ 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 put-get*)

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

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

end

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

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

3.5 Very Well-behaved Lnses

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* ϱ $\Longrightarrow$ *put* σ *v* = *put* ϱ *v* $\Longrightarrow$ σ = ϱ
    by (*metis get-put put-put*)

 lemma *put-eq*:
   $[\![$ *get* σ = *k*; *put* σ *u* = *put* ϱ *v* $]\!]$ $\Longrightarrow$ *put* ϱ *k* = σ
   by (*metis get-put put-put*)

end

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

lemma *vwb-lens-mwb* [*simp*]: *vwb-lens x* $\Longrightarrow$ *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-inef*)
qed

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* $\wedge$ ($\neg$ *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 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*: $put\ \sigma\ (get\ \varrho) = \varrho$
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* σ)
by (*metis strong-get-put surj-def*)

lemma *put-bij*: *bij* (*put* σ)
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*)

end

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

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

lemma *bij-lens-vwb* [*simp*]: *bij-lens x* $\Longrightarrow$ *vwb-lens x*
by (*unfold-locales*, *simp-all add*: *bij-lens.put-is-create*)

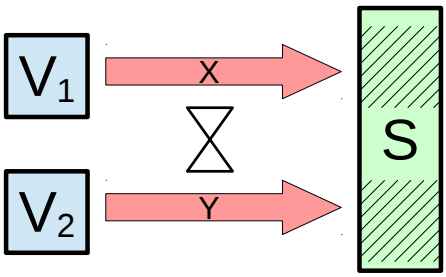

Figure 2: Lens Independence

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 $X :: 'a \Longrightarrow 'c$ **and** $Y :: 'b \Longrightarrow 'c$
  assumes *lens-put-comm*: *lens-put* X (*lens-put* Y σ v) u = *lens-put* Y (*lens-put* X σ u) v
  and *lens-put-irr1*: *lens-get* X (*lens-put* Y σ v) = *lens-get* X σ
  and *lens-put-irr2*: *lens-get* Y (*lens-put* X σ u) = *lens-get* Y σ

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

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

Lens independence is symmetric.

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

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

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

4 Lens Algebraic Operators

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

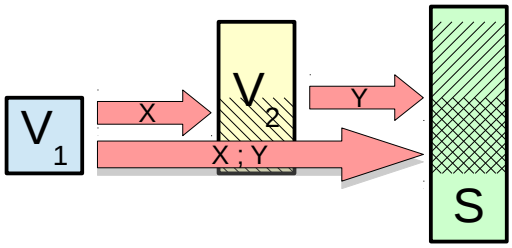

Figure 3: Lens Composition

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

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

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

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

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

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

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

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

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

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

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

abbreviation *swap-lens* :: $'a \times 'b \Longrightarrow 'b \times 'a$ ($swap_L$) **where**
$swap_L \equiv snd_L +_L fst_L$

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

definition *zero-lens* :: *unit* $\Longrightarrow 'a$ (0_L) **where**

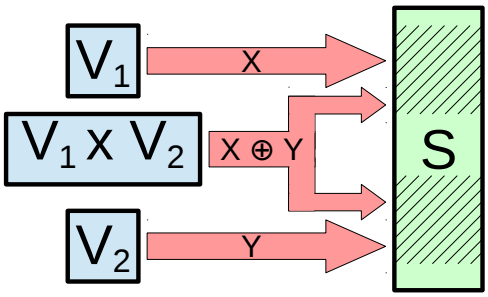

Figure 4: Lens Sum

[*lens-defs*]: 0_L = ⦇ *lens-get* = (λ -. ()), *lens-put* = (λ σ *x*. σ) ⦈

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

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

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

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

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

definition *lens-override* :: $'a \Rightarrow 'a \Rightarrow ('b \Longrightarrow 'a) \Rightarrow 'a$ (- $\oplus_L$ - *on* - [*95*,*0*,*96*] *95*) **where**
[*lens-defs*]: $S_1 \oplus_L S_2$ *on* X = put_X S_1 (get_X S_2)

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

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

4.2 Closure Poperties

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

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

lemma *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*: $\llbracket$ *vwb-lens x*; *vwb-lens y* $\rrbracket \Longrightarrow$ *vwb-lens* $(x ;_L y)$
by (*unfold-locales*, *simp-all add*: *lens-comp-def*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3 Composition Laws

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

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

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

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

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

4.4 Independence Laws

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

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

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

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

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

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

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

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

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

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

lemma *lens-comp-indep-cong-left*:
$[\![$ *mwb-lens* Z; $X ;_L Z \bowtie Y ;_L Z$ $]\!] \Longrightarrow X \bowtie Y$
apply (*rule lens-indepI*)
apply (*rename-tac* u v σ)
apply (*drule-tac* u=u **and** v=v **and** σ=$create_Z$ σ **in** *lens-indep-comm*)
apply (*simp add*: *lens-comp-def*)
apply (*meson mwb-lens-weak weak-lens.view-determination*)
apply (*rename-tac* v σ)
apply (*drule-tac* v=v **and** σ=$create_Z$ σ **in** *lens-indep-get*)
apply (*simp add*: *lens-comp-def*)
apply (*drule lens-indep-sym*)
apply (*rename-tac* u σ)
apply (*drule-tac* v=u **and** σ=$create_Z$ σ **in** *lens-indep-get*)
apply (*simp add*: *lens-comp-def*)
done

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

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

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

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

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

Lens independence is preserved by summation.

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

apply (*rule lens-indepI*)
apply (*simp-all add*: *lens-plus-def prod.case-eq-if*)
apply (*simp add*: *lens-indep-comm*)
apply (*simp add*: *lens-indep-sym*)
done

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

Lens independence is preserved by product.

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

4.5 Algebraic Laws

Lens plus distributes to the right through composition.

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

The first lens projects the first part of a summation.

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

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

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

The swap lens switches over a summation.

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

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

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

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

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

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

The product lens can be rewritten as a sum lens.

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

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

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

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

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

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

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

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

end

5 Order and Equivalence on Lenses

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

5.1 Sub-lens Relation

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

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

Various lens classes are downward closed under the sublens relation.

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

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

lemma *sublens-pres-vwb*:

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

Sublens is a preorder as the following two theorems show.

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

lemma *sublens-trans* [*trans*]:
$[\![X \subseteq_L Y;\ Y \subseteq_L Z]\!] \Longrightarrow X \subseteq_L Z$
apply (*auto simp add*: *sublens-def lens-comp-assoc*)
apply (*rename-tac* Z_1 Z_2)
apply (*rule-tac* $x{=}Z_1$ $;_L$ Z_2 **in** *exI*)
apply (*simp add*: *lens-comp-assoc*)
using *comp-vwb-lens* **apply** *blast*
done

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

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

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

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

lemma *sublens-put-put*:
$[\![\textit{mwb-lens } X;\ Y \subseteq_L X]\!] \Longrightarrow \textit{put}_X\ (\textit{put}_Y\ \sigma\ v)\ u = \textit{put}_X\ \sigma\ u$
by (*auto simp add*: *sublens-def lens-comp-def*)

lemma *sublens-obs-get*:
$[\![\textit{mwb-lens } X;\ Y \subseteq_L X]\!] \Longrightarrow \textit{get}_Y\ (\textit{put}_X\ \sigma\ v) = \textit{get}_Y\ (\textit{put}_X\ \varrho\ v)$
by (*auto simp add*: *sublens-def lens-comp-def*)

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

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

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*:
$[\![\textit{mwb-lens } Y;\ X \subseteq_L Y]\!] \Longrightarrow \textit{mwb-lens } (X\ /_L\ Y)$
by (*unfold-locales*, *auto simp add*: *lens-quotient-def lens-create-def sublens-def lens-comp-def comp-def*)

5.2 Lens Equivalence

Using our preorder, we can also derive an equivalence on lenses as follows. It should be noted that this equality, like sublens, is heterogeneously typed – it can compare lenses whose view types are different, so long as the source types are the same. We show that it is reflexive, symmetric, and transitive.

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

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

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

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

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

5.3 Further Algebraic Laws

This law explains the behaviour of lens quotient.

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

Plus distributes through quotient.

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

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

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

Intuitively, lens plus is associative. However we cannot prove this using HOL equality due to monomorphic typing of this operator. But since sublens and lens equivalence are both

heterogeneous we can now prove this in the following three lemmas.

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

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

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

We can similarly show that it is commutative.

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

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

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

laws.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Bijective Lens Equivalences

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

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

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

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

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

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

lemma *bij-lens-equiv*:
$[\![$ *bij-lens* X; $X \approx_L Y$ $]\!] \Longrightarrow$ *bij-lens* Y
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*:
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 1_L, then both of the composed lenses must be bijective.

```
lemma bij-lens-via-comp-id-left:
 ⟦ wb-lens X; wb-lens Y; X ;_L Y = 1_L ⟧ ⟹ bij-lens X
 apply (cases Y)
 apply (cases X)
 apply (auto simp add: lens-comp-def comp-def id-lens-def fun-eq-iff)
 apply (unfold-locales)
  apply (simp-all)
  using vwb-lens-wb wb-lens-weak weak-lens.put-get apply fastforce
 apply (metis select-convs(1) select-convs(2) wb-lens-weak weak-lens.put-get)
done

lemma bij-lens-via-comp-id-right:
 ⟦ wb-lens X; wb-lens Y; X ;_L Y = 1_L ⟧ ⟹ bij-lens Y
 apply (cases Y)
 apply (cases X)
 apply (auto simp add: lens-comp-def comp-def id-lens-def fun-eq-iff)
 apply (unfold-locales)
  apply (simp-all)
  using vwb-lens-wb wb-lens-weak weak-lens.put-get apply fastforce
 apply (metis select-convs(1) select-convs(2) wb-lens-weak weak-lens.put-get)
done
```

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

```
lemma lens-equiv-via-bij:
 assumes bij-lens Z X = Z ;_L Y
 shows X ≈_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 ≈_L Y ⟷ (∃ Z. bij-lens Z ∧ X = Z ;_L Y)
 apply (rule iffI)
  apply (auto simp add: lens-equiv-def sublens-def lens-id-unique)[1]
  apply (rename-tac Z_1 Z_2)
  apply (rule-tac x=Z_1 in exI)
  apply (simp)
```

apply (*subgoal-tac* Z_2 ;$_L$ Z_1 = 1_L)
apply (*meson bij-lens-via-comp-id-right vwb-lens-wb*)
apply (*metis assms lens-comp-assoc lens-id-unique*)
using *lens-equiv-via-bij* **apply** *blast*
done

5.5 Lens Override Laws

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

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

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

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

lemma *lens-override-plus*:
$X \bowtie Y \Longrightarrow S_1 \oplus_L S_2$ *on* $(X +_L Y) = (S_1 \oplus_L S_2$ *on* $X) \oplus_L S_2$ *on* Y
by (*simp add*: *lens-indep-comm lens-override-def lens-plus-def*)
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* :: $'a \Rightarrow ('b{::}two \Longrightarrow ('a \Rightarrow 'b))$ **where**
[*lens-defs*]: *fun-lens* x = (| *lens-get* = (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 $x \bowtie$ *fun-lens* $y \longleftrightarrow x \neq y$
proof −
obtain u v :: $'a{::}two$ **where** $u \neq v$

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

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

lemma *fun-ran-mwb-lens*: ⟦ *mwb-lens X*; *mwb-lens Y* ⟧ $\Longrightarrow$ *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* ⟧ $\Longrightarrow$ *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* ⟧ $\Longrightarrow$ *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* :: $'a \Rightarrow ('b \Longrightarrow ('a \rightharpoonup 'b))$ **where**
[*lens-defs*]: *map-lens x* = (| *lens-get* = (λ *f*. *the* (*f x*)), *lens-put* = (λ *f u*. $f(x \mapsto u)$) |)

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* $\Rightarrow$ *nat* $\Rightarrow$ '*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* $\Rightarrow$ *nat* $\Rightarrow$ '*a* $\Rightarrow$ '*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* $\Rightarrow$ *nat* $\Rightarrow$ '*a* **where**
nth' xs i = (*if* (*length xs* > *i*) *then xs* ! *i else undefined*)

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

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

by (*simp add*: *list-update-append*)

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

We can also prove some laws about our new operators.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

abbreviation *hd-lens* ≡ *list-lens 0*

definition *tl-lens* :: *'a list* ⟹ *'a list* **where**
[*lens-defs*]: *tl-lens* = (| *lens-get* = (λ *xs*. *tl xs*)
, *lens-put* = (λ *xs xs'*. *hd xs* # *xs'*) |)

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

lemma *tail-lens-mwb*:
mwb-lens 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 ≠ *j* ⟹ *list-lens i* ⋈ *list-lens j*
by (*simp add*: *list-lens-def lens-indep-def list-augment-commute nth'-list-augment-diff*)

lemma *hd-tl-lens-indep* [*simp*]:
hd-lens ⋈ *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'-def 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 **alphabet** which enables this. We first create syntax that allows us to obtain a lens from a given field using the internal record syntax translations.

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

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

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

ML-file *Lens-Record.ML*

The following theorem attribute stores splitting theorems for alphabet types which 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* (*match*$_1$)
  prism-build :: '*v* ⇒ '*s* (*build*$_1$)

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*),

$$prism\text{-}build = (\lambda\ v.\ case\ v\ of\ Inl\ x \Rightarrow build_X\ x \mid Inr\ y \Rightarrow build_Y\ y)\ ||$$

end
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 particurlarly their invaluable advice on Isabelle mechanisation and ML coding.

References

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

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

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

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

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