Optics in Isabelle/HOL

Simon Foster, Christian Pardillo-Laursen, and Frank Zeyda
University of York, UK

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

March 17, 2025

Abstract

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

Contents

1 Interpretation Tools

theory *Interp*
imports *Main*
begin

1.1 Interpretation Locale

locale *interp* =
fixes $f :: 'a \Rightarrow 'b$
assumes *f-inj* : *inj f*
begin
lemma *meta-interp-law*:
$(\bigwedge P.\ PROP\ Q\ P) \equiv (\bigwedge P.\ PROP\ Q\ (P\ o\ f))$
  ⟨*proof*⟩

lemma *all-interp-law*:
$(\forall P.\ Q\ P) = (\forall P.\ Q\ (P\ o\ f))$
  ⟨*proof*⟩

lemma *exists-interp-law*:
$(\exists P.\ Q\ P) = (\exists P.\ Q\ (P\ o\ f))$
  ⟨*proof*⟩
end
end

2 Types of Cardinality 2 or Greater

theory *Two*
imports *HOL.Real*
begin

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

class *two* =
  assumes *card-two*: $infinite\ (UNIV :: 'a\ set) \lor card\ (UNIV :: 'a\ set) \geq 2$
begin

lemma *two-diff*: $\exists\ x\ y :: 'a.\ x \neq y$
⟨*proof*⟩
end

instance *bool* :: *two*
⟨*proof*⟩

instance *nat* :: *two*
⟨*proof*⟩

instance *int* :: *two*
⟨*proof*⟩

instance *rat* :: *two*
⟨*proof*⟩

instance *real* :: *two*
⟨*proof*⟩

instance *list* :: (*type*) *two*
⟨*proof*⟩

end

3 Core Lens Laws

theory *Lens-Laws*
imports
Two Interp
begin

3.1 Lens Signature

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

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

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

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

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

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

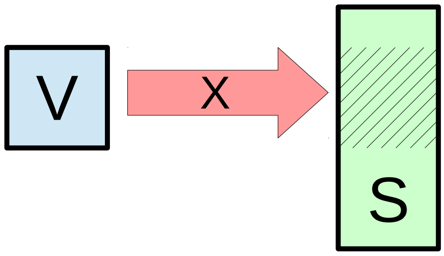

Figure 1: Visualisation of a simple lens

named-theorems *lens-defs*

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

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

A partial version of *lens-get*, which can be useful for partial lenses.

definition *lens-partial-get* :: $('a \Longrightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ option$ (‹*pget*ı›) **where**
lens-partial-get $x\ s = (if\ s \in \mathcal{S}_x\ then\ Some\ (get_x\ s)\ else\ None)$

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

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

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

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

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

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

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

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

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

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

3.2 Weak Lenses

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

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

  lemma *put-closure*: $put\ \sigma\ v \in \mathcal{S}$
    ⟨*proof*⟩

  lemma *create-closure*: $create\ v \in \mathcal{S}$
    ⟨*proof*⟩

  lemma *src-source* [*simp*]: $src \in \mathcal{S}$
    ⟨*proof*⟩

  lemma *create-get*: $get\ (create\ v) = v$
    ⟨*proof*⟩

  lemma *create-inj*: *inj create*
    ⟨*proof*⟩

  lemma *get-update*: $get\ (update\ f\ \sigma) = f\ (get\ \sigma)$
    ⟨*proof*⟩

  lemma *view-determination*:
    assumes $put\ \sigma\ u = put\ \varrho\ v$
    shows $u = v$
    ⟨*proof*⟩

  lemma *put-inj*: $inj\ (put\ \sigma)$
    ⟨*proof*⟩

end

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

lemma *dom-pget*: $dom\ pget_x = \mathcal{S}_x$
  ⟨*proof*⟩

3.3 Well-behaved Lenses

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

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

  lemma *put-twice*: $put\ (put\ \sigma\ v)\ v = put\ \sigma\ v$
    ⟨*proof*⟩

  lemma *put-surjectivity*: $\exists\ \varrho\ v.\ put\ \varrho\ v = \sigma$
    ⟨*proof*⟩

lemma *source-stability*: $\exists\ v.\ put\ \sigma\ v = \sigma$
⟨*proof*⟩

lemma *source-UNIV* [*simp*]: $\mathcal{S} = UNIV$
⟨*proof*⟩

end

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

lemma *wb-lens-weak* [*simp*]: *wb-lens* $x \Longrightarrow$ *weak-lens* x
⟨*proof*⟩

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*: $put\ (put\ \sigma\ v)\ u = put\ \sigma\ u$
begin

lemma *update-comp*: $update\ f\ (update\ g\ \sigma) = update\ (f \circ g)\ \sigma$
⟨*proof*⟩

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

lemma *weak-get-put*: $\sigma \in \mathcal{S} \Longrightarrow put\ \sigma\ (get\ \sigma) = \sigma$
⟨*proof*⟩

lemma *weak-source-determination*:
assumes $\sigma \in \mathcal{S}$ $\varrho \in \mathcal{S}$ $get\ \sigma = get\ \varrho$ $put\ \sigma\ v = put\ \varrho\ v$
shows $\sigma = \varrho$
⟨*proof*⟩

lemma *weak-put-eq*:
assumes $\sigma \in \mathcal{S}$ $get\ \sigma = k$ $put\ \sigma\ u = put\ \varrho\ v$
shows $put\ \varrho\ k = \sigma$
⟨*proof*⟩

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

lemma *weak-get-via-put*: $s \in \mathcal{S} \Longrightarrow get\ s = (THE\ v.\ put\ s\ v = s)$
⟨*proof*⟩

end

abbreviation (*input*) *partial-lens* ≡ *mwb-lens*

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

lemma *mwb-lens-weak* [*simp*]:
mwb-lens $x \Longrightarrow$ *weak-lens* x
⟨*proof*⟩

3.5 Very Well-behaved Lenses

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

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

lemma *source-determination*:
assumes $get\ \sigma = get\ \varrho$ $put\ \sigma\ v = put\ \varrho\ v$
shows $\sigma = \varrho$
⟨*proof*⟩

lemma *put-eq*:
assumes $get\ \sigma = k$ $put\ \sigma\ u = put\ \varrho\ v$
shows $put\ \varrho\ k = \sigma$
⟨*proof*⟩

get can be uniquely determined from *put*

lemma *get-via-put*: $get\ s = (THE\ v.\ put\ s\ v = s)$
⟨*proof*⟩

lemma *get-surj*: *surj get*
⟨*proof*⟩

Observation equivalence is an equivalence relation.

lemma *lens-obs-equiv*: *equivp* (≃)
⟨*proof*⟩

end

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

lemma *vwb-lens-wb* [*simp*]: $vwb\text{-}lens\ x \Longrightarrow wb\text{-}lens\ x$
⟨*proof*⟩

lemma *vwb-lens-mwb* [*simp*]: $vwb\text{-}lens\ x \Longrightarrow mwb\text{-}lens\ x$
⟨*proof*⟩

lemma *mwb-UNIV-src-is-vwb-lens*:
$[\![\ mwb\text{-}lens\ X;\ \mathcal{S}_X = UNIV\ ]\!] \Longrightarrow vwb\text{-}lens\ X$
⟨*proof*⟩

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

lemma *vwb-lens-iff-mwb-UNIV-src*:
$vwb\text{-}lens\ X \longleftrightarrow (mwb\text{-}lens\ X \wedge \mathcal{S}_X = UNIV)$
⟨*proof*⟩

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*: $put\ \sigma\ v = \sigma$
begin

lemma *ief-then-vwb*: *vwb-lens x*
⟨*proof*⟩

sublocale *vwb-lens* ⟨*proof*⟩

lemma *ineffectual-const-get*:
$\exists\ v.\ \ \forall\ \sigma{\in}\mathcal{S}.\ get\ \sigma = v$
⟨*proof*⟩

end

declare *ief-lens.ief-then-vwb* [*simp*]

There is no ineffectual lens when the view type has two or more elements.

lemma *no-ief-two-view*:
assumes *ief-lens* $(x :: {}'a{::}two \Longrightarrow {}'s)$
shows *False*
⟨*proof*⟩

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

3.7 Partially Bijective Lenses

locale *pbij-lens* = *weak-lens* +
assumes *put-det*: $put\ \sigma\ v = put\ \varrho\ v$
begin

sublocale *mwb-lens*
⟨*proof*⟩

lemma *put-is-create*: $put\ \sigma\ v = create\ v$
⟨*proof*⟩

lemma *partial-get-put*: $\varrho \in \mathcal{S} \Longrightarrow put\ \sigma\ (get\ \varrho) = \varrho$
⟨*proof*⟩

end

lemma *pbij-lens-weak* [*simp*]:
$pbij\text{-}lens\ x \Longrightarrow weak\text{-}lens\ x$
⟨*proof*⟩

lemma *pbij-lens-mwb* [*simp*]: $pbij\text{-}lens\ x \Longrightarrow mwb\text{-}lens\ x$
⟨*proof*⟩

lemma *pbij-alt-intro*:
$[\![\ weak\text{-}lens\ X;\ \bigwedge\ s.\ s \in \mathcal{S}_X \Longrightarrow create_X\ (get_X\ s) = s\ ]\!] \Longrightarrow pbij\text{-}lens\ X$
⟨*proof*⟩

3.8 Bijective Lenses

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

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

sublocale *pbij-lens*
⟨*proof*⟩

sublocale *vwb-lens*
⟨*proof*⟩

  lemma *put-bij*: $bij\text{-}betw\ (put\ \sigma)\ UNIV\ UNIV$
    ⟨*proof*⟩

  lemma *get-create*: $create\ (get\ \sigma) = \sigma$
    ⟨*proof*⟩

end

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

lemma *bij-lens-weak* [*simp*]:
  $bij\text{-}lens\ x \Longrightarrow weak\text{-}lens\ x$
  ⟨*proof*⟩

lemma *bij-lens-pbij* [*simp*]:
  $bij\text{-}lens\ x \Longrightarrow pbij\text{-}lens\ x$
  ⟨*proof*⟩

lemma *bij-lens-vwb* [*simp*]: $bij\text{-}lens\ x \Longrightarrow vwb\text{-}lens\ x$
  ⟨*proof*⟩

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

lemma *pbij-vwb-is-bij-lens*:
  $\llbracket\ pbij\text{-}lens\ X;\ vwb\text{-}lens\ X\ \rrbracket \Longrightarrow bij\text{-}lens\ X$
  ⟨*proof*⟩

lemma *bij-lens-iff-pbij-vwb*:
  $bij\text{-}lens\ X \longleftrightarrow (pbij\text{-}lens\ X \wedge vwb\text{-}lens\ X)$
  ⟨*proof*⟩

3.9 Lens Independence

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

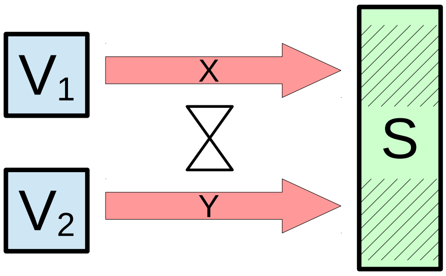

Figure 2: Lens Independence

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

notation *lens-indep* (**infix** ‹⋈› *50*)

lemma *lens-indepI*:
  $[\![\bigwedge u\ v\ \sigma.\ put_x\ (put_y\ \sigma\ v)\ u = put_y\ (put_x\ \sigma\ u)\ v;$
  $\bigwedge v\ \sigma.\ get_x\ (put_y\ \sigma\ v) = get_x\ \sigma;$
  $\bigwedge u\ \sigma.\ get_y\ (put_x\ \sigma\ u) = get_y\ \sigma\ ]\!] \Longrightarrow x \bowtie y$
  ⟨*proof*⟩

Lens independence is symmetric.

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

lemma *lens-indep-comm*:
  $x \bowtie y \Longrightarrow put_x\ (put_y\ \sigma\ v)\ u = put_y\ (put_x\ \sigma\ u)\ v$
  ⟨*proof*⟩

lemma *lens-indep-get* [*simp*]:
  assumes $x \bowtie y$
  shows $get_x\ (put_y\ \sigma\ v) = get_x\ \sigma$
  ⟨*proof*⟩

Characterisation of independence for two very well-behaved lenses

lemma *lens-indep-vwb-iff*:
  assumes *vwb-lens x vwb-lens y*
  shows $x \bowtie y \longleftrightarrow (\forall\ u\ v\ \sigma.\ put_x\ (put_y\ \sigma\ v)\ u = put_y\ (put_x\ \sigma\ u)\ v)$
⟨*proof*⟩

3.10 Lens Compatibility

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

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

lemma *lens-compat-idem* [*simp*]: $x\ \#\#_L\ x$
  ⟨*proof*⟩

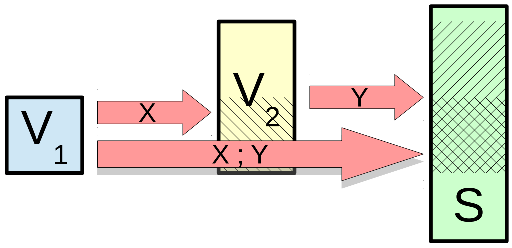

Figure 3: Lens Composition

lemma *lens-compat-sym*: $x\ \#\#_L\ y \Longrightarrow y\ \#\#_L\ x$
⟨*proof*⟩

lemma *lens-indep-compat* [*simp*]: $x \bowtie y \Longrightarrow x\ \#\#_L\ y$
⟨*proof*⟩

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

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

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*]: $X +_L Y = (\!|\ lens\text{-}get = (\lambda\ \sigma.\ (lens\text{-}get\ X\ \sigma,\ lens\text{-}get\ Y\ \sigma))$
$,\ lens\text{-}put = (\lambda\ \sigma\ (u,\ v).\ lens\text{-}put\ X\ (lens\text{-}put\ Y\ \sigma\ v)\ u)\ |\!)$

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*]: $lens\text{-}prod\ X\ Y = (\!|\ lens\text{-}get = map\text{-}prod\ get_X\ get_Y$
$,\ lens\text{-}put = \lambda\ (u,\ v)\ (x,\ y).\ (put_X\ u\ x,\ put_Y\ v\ y)\ |\!)$

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*]: $fst_L = (\!|\ lens\text{-}get = fst,\ lens\text{-}put = (\lambda\ (\sigma,\ \varrho)\ u.\ (u,\ \varrho))\ |\!)$

definition *snd-lens* :: $'b \Longrightarrow 'a \times 'b$ (‹snd_L›) **where**

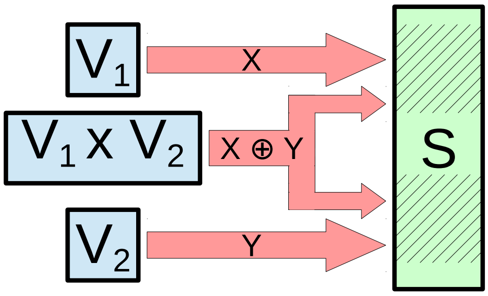

Figure 4: Lens Sum

[*lens-defs*]: snd_L = ⦇ *lens-get* = *snd*, *lens-put* = (λ (σ, ϱ) *u*. (σ, *u*)) ⦈

lemma *get-fst-lens* [*simp*]: get_{fst_L} (*x*, *y*) = *x*
⟨*proof*⟩

lemma *get-snd-lens* [*simp*]: get_{snd_L} (*x*, *y*) = *y*
⟨*proof*⟩

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

abbreviation *swap-lens* :: ′*a* × ′*b* ⟹ ′*b* × ′*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* ⟹ ′*a* (‹0_L›) **where**
[*lens-defs*]: 0_L = ⦇ *lens-get* = (λ -. ()), *lens-put* = (λ σ *x*. σ) ⦈

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

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

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

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

4.2 Closure 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
⟨*proof*⟩

lemma *source-id-lens*: $\mathcal{S}_{1_L}$ = *UNIV*
⟨*proof*⟩

lemma *unit-wb-lens*: *wb-lens* 0_L

⟨*proof*⟩

lemma *source-zero-lens*: $\mathcal{S}_{0_L} = UNIV$
⟨*proof*⟩

lemma *comp-weak-lens*: ⟦ *weak-lens x*; *weak-lens y* ⟧ ⟹ *weak-lens* ($x ;_L y$)
⟨*proof*⟩

lemma *comp-wb-lens*: ⟦ *wb-lens x*; *wb-lens y* ⟧ ⟹ *wb-lens* ($x ;_L y$)
⟨*proof*⟩

lemma *comp-mwb-lens*: ⟦ *mwb-lens x*; *mwb-lens y* ⟧ ⟹ *mwb-lens* ($x ;_L y$)
⟨*proof*⟩

lemma *source-lens-comp*: ⟦ *mwb-lens x*; *mwb-lens y* ⟧ ⟹ $\mathcal{S}_{x ;_L y} = \{s \in \mathcal{S}_y.\ get_y\ s \in \mathcal{S}_x\}$
⟨*proof*⟩

lemma *id-vwb-lens* [*simp*]: *vwb-lens* 1_L
⟨*proof*⟩

lemma *unit-vwb-lens* [*simp*]: *vwb-lens* 0_L
⟨*proof*⟩

lemma *comp-vwb-lens*: ⟦ *vwb-lens x*; *vwb-lens y* ⟧ ⟹ *vwb-lens* ($x ;_L y$)
⟨*proof*⟩

lemma *unit-ief-lens*: *ief-lens* 0_L
⟨*proof*⟩

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* ($x +_L y$)
⟨*proof*⟩

lemma *plus-wb-lens*:
assumes *wb-lens x wb-lens y* $x \bowtie y$
shows *wb-lens* ($x +_L y$)
⟨*proof*⟩

lemma *plus-vwb-lens* [*simp*]:
assumes *vwb-lens x vwb-lens y* $x \bowtie y$
shows *vwb-lens* ($x +_L y$)
⟨*proof*⟩

lemma *source-plus-lens*:
assumes *mwb-lens x mwb-lens y* $x \bowtie y$
shows $\mathcal{S}_{x +_L y} = \mathcal{S}_x \cap \mathcal{S}_y$
⟨*proof*⟩

lemma *prod-mwb-lens*:
⟦ *mwb-lens X*; *mwb-lens Y* ⟧ ⟹ *mwb-lens* ($X \times_L Y$)
⟨*proof*⟩

lemma *prod-wb-lens*:

$[\![$ *wb-lens* X; *wb-lens* Y $]\!] \Longrightarrow$ *wb-lens* $(X \times_L Y)$
⟨*proof*⟩

lemma *prod-vwb-lens*:
$[\![$ *vwb-lens* X; *vwb-lens* Y $]\!] \Longrightarrow$ *vwb-lens* $(X \times_L Y)$
⟨*proof*⟩

lemma *prod-bij-lens*:
$[\![$ *bij-lens* X; *bij-lens* Y $]\!] \Longrightarrow$ *bij-lens* $(X \times_L Y)$
⟨*proof*⟩

lemma *fst-vwb-lens*: *vwb-lens* fst_L
⟨*proof*⟩

lemma *snd-vwb-lens*: *vwb-lens* snd_L
⟨*proof*⟩

lemma *id-bij-lens*: *bij-lens* 1_L
⟨*proof*⟩

lemma *inv-id-lens*: $inv_L\ 1_L = 1_L$
⟨*proof*⟩

lemma *inv-inv-lens*: *bij-lens* $X \Longrightarrow inv_L\ (inv_L\ X) = X$
⟨*proof*⟩

lemma *lens-inv-bij*: *bij-lens* $X \Longrightarrow$ *bij-lens* $(inv_L\ X)$
⟨*proof*⟩

lemma *swap-bij-lens*: *bij-lens* $swap_L$
⟨*proof*⟩

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*: $X ;_L (Y ;_L Z) = (X ;_L Y) ;_L Z$
⟨*proof*⟩

lemma *lens-comp-left-id* [*simp*]: $1_L ;_L X = X$
⟨*proof*⟩

lemma *lens-comp-right-id* [*simp*]: $X ;_L 1_L = X$
⟨*proof*⟩

lemma *lens-comp-anhil* [*simp*]: *wb-lens* $X \Longrightarrow 0_L ;_L X = 0_L$
⟨*proof*⟩

lemma *lens-comp-anhil-right* [*simp*]: *wb-lens* $X \Longrightarrow X ;_L 0_L = 0_L$
⟨*proof*⟩

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*]: $0_L \bowtie X$
⟨*proof*⟩

lemma *zero-lens-indep′* [*simp*]: $X \bowtie 0_L$
⟨*proof*⟩

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

lemma *lens-indep-quasi-irrefl*: $\llbracket$ *wb-lens* x; *eff-lens* x $\rrbracket \Longrightarrow \neg (x \bowtie x)$
⟨*proof*⟩

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

lemma *lens-indep-left-comp* [*simp*]:
$\llbracket$ *mwb-lens* z; $x \bowtie y$ $\rrbracket \Longrightarrow (x ;_L z) \bowtie (y ;_L z)$
⟨*proof*⟩

lemma *lens-indep-right-comp*:
$y \bowtie z \Longrightarrow (x ;_L y) \bowtie (x ;_L z)$
⟨*proof*⟩

lemma *lens-indep-left-ext* [*intro*]:
$y \bowtie z \Longrightarrow (x ;_L y) \bowtie z$
⟨*proof*⟩

lemma *lens-indep-right-ext* [*intro*]:
$x \bowtie z \Longrightarrow x \bowtie (y ;_L z)$
⟨*proof*⟩

lemma *lens-comp-indep-cong-left*:
$\llbracket$ *mwb-lens* Z; $X ;_L Z \bowtie Y ;_L Z$ $\rrbracket \Longrightarrow X \bowtie Y$
⟨*proof*⟩

lemma *lens-comp-indep-cong*:
mwb-lens $Z \Longrightarrow (X ;_L Z) \bowtie (Y ;_L Z) \longleftrightarrow X \bowtie Y$
⟨*proof*⟩

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

lemma *fst-snd-lens-indep* [*simp*]:
$fst_L \bowtie snd_L$
⟨*proof*⟩

lemma *snd-fst-lens-indep* [*simp*]:
$snd_L \bowtie fst_L$
⟨*proof*⟩

lemma *split-prod-lens-indep*:
assumes *mwb-lens* X
shows $(fst_L ;_L X) \bowtie (snd_L ;_L X)$
⟨*proof*⟩

Lens independence is preserved by summation.

lemma *plus-pres-lens-indep* [*simp*]: $\llbracket X \bowtie Z; Y \bowtie Z \rrbracket \Longrightarrow (X +_L Y) \bowtie Z$
⟨*proof*⟩

lemma *plus-pres-lens-indep'* [*simp*]:
$[\![X \bowtie Y; X \bowtie Z]\!] \Longrightarrow X \bowtie Y +_L Z$
⟨*proof*⟩

Lens independence is preserved by product.

lemma *lens-indep-prod*:
$[\![X_1 \bowtie X_2; Y_1 \bowtie Y_2]\!] \Longrightarrow X_1 \times_L Y_1 \bowtie X_2 \times_L Y_2$
⟨*proof*⟩

4.5 Compatibility Laws

lemma *zero-lens-compat* [*simp*]: $0_L \;\#\#_L\; X$
⟨*proof*⟩

lemma *id-lens-compat* [*simp*]: *vwb-lens* $X \Longrightarrow 1_L \;\#\#_L\; X$
⟨*proof*⟩

4.6 Algebraic Laws

Lens plus distributes to the right through composition.

lemma *plus-lens-distr*: *mwb-lens* $Z \Longrightarrow (X +_L Y) ;_L Z = (X ;_L Z) +_L (Y ;_L Z)$
⟨*proof*⟩

The first lens projects the first part of a summation.

lemma *fst-lens-plus*:
wb-lens $y \Longrightarrow fst_L ;_L (x +_L y) = x$
⟨*proof*⟩

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

lemma *snd-lens-plus*:
$[\![$ *wb-lens* $x; x \bowtie y]\!] \Longrightarrow snd_L ;_L (x +_L y) = y$
⟨*proof*⟩

The swap lens switches over a summation.

lemma *lens-plus-swap*:
$X \bowtie Y \Longrightarrow swap_L ;_L (X +_L Y) = (Y +_L X)$
⟨*proof*⟩

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

lemma *fst-snd-id-lens*: $fst_L +_L snd_L = 1_L$
⟨*proof*⟩

lemma *swap-lens-idem*: $swap_L ;_L swap_L = 1_L$
⟨*proof*⟩

lemma *swap-lens-fst*: $fst_L ;_L swap_L = snd_L$
⟨*proof*⟩

lemma *swap-lens-snd*: $snd_L ;_L swap_L = fst_L$
⟨*proof*⟩

The product lens can be rewritten as a sum lens.

lemma *prod-as-plus*: $X \times_L Y = X ;_L fst_L +_L Y ;_L snd_L$

⟨*proof*⟩

lemma *prod-lens-id-equiv*:
$1_L \times_L 1_L = 1_L$
⟨*proof*⟩

lemma *prod-lens-comp-plus*:
$X_2 \bowtie Y_2 \Longrightarrow ((X_1 \times_L Y_1) ;_L (X_2 +_L Y_2)) = (X_1 ;_L X_2) +_L (Y_1 ;_L Y_2)$
⟨*proof*⟩

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 $Y \Longrightarrow (X ;_L Y) /_L Y = X$
⟨*proof*⟩

lemma *lens-quotient-id* [*simp*]: *weak-lens* $X \Longrightarrow (X /_L X) = 1_L$
⟨*proof*⟩

lemma *lens-quotient-id-denom*: $X /_L 1_L = X$
⟨*proof*⟩

lemma *lens-quotient-unit*: *weak-lens* $X \Longrightarrow (0_L /_L X) = 0_L$
⟨*proof*⟩

lemma *lens-obs-eq-zero*: $s_1 \simeq_{0_L} s_2 = (s_1 = s_2)$
⟨*proof*⟩

lemma *lens-obs-eq-one*: $s_1 \simeq_{1_L} s_2$
⟨*proof*⟩

lemma *lens-obs-eq-as-override*: *vwb-lens* $X \Longrightarrow s_1 \simeq_X s_2 \longleftrightarrow (s_2 = s_1 \oplus_L s_2 \ on \ X)$
⟨*proof*⟩

end

5 Order and Equivalence on Lenses

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

5.1 Sub-lens Relation

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

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

Various lens classes are downward closed under the sublens relation.

lemma *sublens-pres-mwb*:
$[\![\ mwb\text{-}lens\ Y;\ X \subseteq_L Y\ ]\!] \Longrightarrow mwb\text{-}lens\ X$
⟨*proof*⟩

lemma *sublens-pres-wb*:
$[\![\, wb\text{-}lens\ Y;\ X \subseteq_L\ Y \,]\!] \Longrightarrow wb\text{-}lens\ X$
⟨*proof*⟩

lemma *sublens-pres-vwb*:
$[\![\, vwb\text{-}lens\ Y;\ X \subseteq_L\ Y \,]\!] \Longrightarrow vwb\text{-}lens\ X$
⟨*proof*⟩

Sublens is a preorder as the following two theorems show.

lemma *sublens-refl* [*simp*]:
$X \subseteq_L X$
⟨*proof*⟩

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

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

lemma *sublens-least*: $wb\text{-}lens\ X \Longrightarrow 0_L \subseteq_L\ X$
⟨*proof*⟩

lemma *sublens-greatest*: $vwb\text{-}lens\ X \Longrightarrow X \subseteq_L\ 1_L$
⟨*proof*⟩

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*:
$[\![\, mwb\text{-}lens\ X;\ Y \subseteq_L\ X \,]\!] \Longrightarrow put_X\ (put_Y\ \sigma\ v)\ u = put_X\ \sigma\ u$
⟨*proof*⟩

lemma *sublens-obs-get*:
$[\![\, mwb\text{-}lens\ X;\ Y \subseteq_L\ X \,]\!] \Longrightarrow\ get_Y\ (put_X\ \sigma\ v) = get_Y\ (put_X\ \varrho\ v)$
⟨*proof*⟩

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*:
$[\![\, X \subseteq_L\ Y;\ Y \bowtie Z \,]\!] \Longrightarrow X \bowtie Z$
⟨*proof*⟩

lemma *sublens-pres-indep'*:
$[\![\, X \subseteq_L\ Y;\ Z \bowtie Y \,]\!] \Longrightarrow Z \bowtie X$
⟨*proof*⟩

lemma *sublens-compat*: $[\![\, vwb\text{-}lens\ X;\ vwb\text{-}lens\ Y;\ X \subseteq_L\ Y \,]\!] \Longrightarrow X\ \#\#_L\ Y$
⟨*proof*⟩

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*:
$[\![\, mwb\text{-}lens\ Y;\ X \subseteq_L\ Y \,]\!] \Longrightarrow mwb\text{-}lens\ (X\ /_L\ Y)$
⟨*proof*⟩

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* $X\ Y = (X \subseteq_L Y \land Y \subseteq_L X)$

lemma *lens-equivI* [*intro*]:
$[\![X \subseteq_L Y; Y \subseteq_L X]\!] \Longrightarrow X \approx_L Y$
⟨*proof*⟩

lemma *lens-equiv-refl* [*simp*]:
$X \approx_L X$
⟨*proof*⟩

lemma *lens-equiv-sym*:
$X \approx_L Y \Longrightarrow Y \approx_L X$
⟨*proof*⟩

lemma *lens-equiv-trans* [*trans*]:
$[\![X \approx_L Y; Y \approx_L Z]\!] \Longrightarrow X \approx_L Z$
⟨*proof*⟩

lemma *lens-equiv-pres-indep*:
$[\![X \approx_L Y; Y \bowtie Z]\!] \Longrightarrow X \bowtie Z$
⟨*proof*⟩

lemma *lens-equiv-pres-indep'*:
$[\![X \approx_L Y; Z \bowtie Y]\!] \Longrightarrow Z \bowtie X$
⟨*proof*⟩

lemma *lens-comp-cong-1*: $X \approx_L Y \Longrightarrow X ;_L Z \approx_L Y ;_L Z$
⟨*proof*⟩

5.3 Further Algebraic Laws

This law explains the behaviour of lens quotient.

lemma *lens-quotient-comp*:
$[\![\textit{weak-lens}\ Y; X \subseteq_L Y]\!] \Longrightarrow (X /_L Y) ;_L Y = X$
⟨*proof*⟩

Plus distributes through quotient.

lemma *lens-quotient-plus*:
$[\![\textit{mwb-lens}\ Z; X \subseteq_L Z; Y \subseteq_L Z]\!] \Longrightarrow (X +_L Y) /_L Z = (X /_L Z) +_L (Y /_L Z)$
⟨*proof*⟩

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

lemma *lens-quotient-plus-den1*:
$[\![\textit{weak-lens}\ x; \textit{weak-lens}\ y; x \bowtie y]\!] \Longrightarrow x /_L (x +_L y) = fst_L$
⟨*proof*⟩

lemma *lens-quotient-plus-den2*: $[\![$ *weak-lens* x; *weak-lens* z; $x \bowtie z$; $y \subseteq_L z$ $]\!] \Longrightarrow y /_L (x +_L z) = (y /_L z) ;_L snd_L$
⟨*proof*⟩

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

lemma *plus-pred-sublens*: $[\![$ *mwb-lens* Z; $X \subseteq_L Z$; $Y \subseteq_L Z$; $X \bowtie Y$ $]\!] \Longrightarrow (X +_L Y) \subseteq_L Z$
⟨*proof*⟩

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*:
$X +_L Y +_L Z \subseteq_L (X +_L Y) +_L Z$
⟨*proof*⟩

lemma *lens-plus-sub-assoc-2*:
$(X +_L Y) +_L Z \subseteq_L X +_L Y +_L Z$
⟨*proof*⟩

lemma *lens-plus-assoc*:
$(X +_L Y) +_L Z \approx_L X +_L Y +_L Z$
⟨*proof*⟩

We can similarly show that it is commutative.

lemma *lens-plus-sub-comm*: $X \bowtie Y \Longrightarrow X +_L Y \subseteq_L Y +_L X$
⟨*proof*⟩

lemma *lens-plus-comm*: $X \bowtie Y \Longrightarrow X +_L Y \approx_L Y +_L X$
⟨*proof*⟩

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

lemma *lens-plus-ub* [*simp*]: *wb-lens* $Y \Longrightarrow X \subseteq_L X +_L Y$
⟨*proof*⟩

lemma *lens-plus-right-sublens*:
$[\![$ *vwb-lens* Y; $Y \bowtie Z$; $X \subseteq_L Z$ $]\!] \Longrightarrow X \subseteq_L Y +_L Z$
⟨*proof*⟩

lemma *lens-plus-mono-left*:
$[\![$ $Y \bowtie Z$; $X \subseteq_L Y$ $]\!] \Longrightarrow X +_L Z \subseteq_L Y +_L Z$
⟨*proof*⟩

lemma *lens-plus-mono-right*:
$[\![$ $X \bowtie Z$; $Y \subseteq_L Z$ $]\!] \Longrightarrow X +_L Y \subseteq_L X +_L Z$
⟨*proof*⟩

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* $X \Longrightarrow X ;_L Y \subseteq_L Y$
⟨*proof*⟩

lemma *sublens-comp* [*simp*]:

assumes $\textit{vwb-lens } b \ c \subseteq_L a$
shows $(b ;_L c) \subseteq_L a$
⟨*proof*⟩

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

lemma *lens-unit-plus-sublens-1*: $X \subseteq_L 0_L +_L X$
⟨*proof*⟩

lemma *lens-unit-prod-sublens-2*: $0_L +_L X \subseteq_L X$
⟨*proof*⟩

lemma *lens-plus-left-unit*: $0_L +_L X \approx_L X$
⟨*proof*⟩

lemma *lens-plus-right-unit*: $X +_L 0_L \approx_L X$
⟨*proof*⟩

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

lemma *lens-plus-subcong*: $[\![Y_1 \bowtie Y_2; X_1 \subseteq_L Y_1; X_2 \subseteq_L Y_2]\!] \Longrightarrow X_1 +_L X_2 \subseteq_L Y_1 +_L Y_2$
⟨*proof*⟩

lemma *lens-plus-eq-left*: $[\![X \bowtie Z; X \approx_L Y]\!] \Longrightarrow X +_L Z \approx_L Y +_L Z$
⟨*proof*⟩

lemma *lens-plus-eq-right*: $[\![X \bowtie Y; Y \approx_L Z]\!] \Longrightarrow X +_L Y \approx_L X +_L Z$
⟨*proof*⟩

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*⟩

lemma *prod-lens-sublens-cong*:
$[\![X_1 \subseteq_L X_2; Y_1 \subseteq_L Y_2]\!] \Longrightarrow (X_1 \times_L Y_1) \subseteq_L (X_2 \times_L Y_2)$
⟨*proof*⟩

lemma *prod-lens-equiv-cong*:
$[\![X_1 \approx_L X_2; Y_1 \approx_L Y_2]\!] \Longrightarrow (X_1 \times_L Y_1) \approx_L (X_2 \times_L Y_2)$
⟨*proof*⟩

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*⟩

lemma *lens-get-put-quasi-commute*:
$[\![\textit{vwb-lens } Y; X \subseteq_L Y]\!] \Longrightarrow \textit{get}_Y (\textit{put}_X \ s \ v) = \textit{put}_{X \ /_L \ Y} (\textit{get}_Y \ s) \ v$
⟨*proof*⟩

lemma *lens-put-of-quotient*:
$[\![\textit{vwb-lens } Y; X \subseteq_L Y]\!] \Longrightarrow \textit{put}_Y \ s \ (\textit{put}_{X \ /_L \ Y} \ v_2 \ v_1) = \textit{put}_X \ (\textit{put}_Y \ s \ v_2) \ v_1$
⟨*proof*⟩

If two lenses are both independent and equivalent then they must be ineffectual.

lemma *indep-and-equiv-implies-ief*:
  assumes *wb-lens* x $x \bowtie y$ $x \approx_L y$
  shows *ief-lens* x
⟨*proof*⟩

lemma *indep-eff-implies-not-equiv* [*simp*]:
  fixes x :: $'a::two \Longrightarrow 'b$
  assumes *wb-lens* x $x \bowtie y$
  shows $\neg$ $(x \approx_L y)$
  ⟨*proof*⟩

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 $X \Longrightarrow inv_L\ X\ ;_L\ X = 1_L$
  ⟨*proof*⟩

lemma *bij-lens-inv-right*:
  bij-lens $X \Longrightarrow X\ ;_L\ inv_L\ X = 1_L$
  ⟨*proof*⟩

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 $X \longleftrightarrow X \approx_L 1_L$
  ⟨*proof*⟩

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
  ⟨*proof*⟩

lemma *bij-lens-cong*:
  $X \approx_L Y \Longrightarrow$ *bij-lens* X = *bij-lens* Y
  ⟨*proof*⟩

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 $Y \Longrightarrow Y = X\ ;_L\ Y \Longrightarrow X = 1_L$
  ⟨*proof*⟩

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

lemma *bij-lens-via-comp-id-left*:
  ⟦ *wb-lens* X; *wb-lens* Y; $X\ ;_L\ Y = 1_L$ ⟧ $\Longrightarrow$ *bij-lens* X
  ⟨*proof*⟩

lemma *bij-lens-via-comp-id-right*:
  ⟦ *wb-lens* X; *wb-lens* Y; $X\ ;_L\ Y = 1_L$ ⟧ $\Longrightarrow$ *bij-lens* Y

⟨*proof*⟩

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

lemma *lens-equiv-via-bij*:
assumes *bij-lens* Z $X = Z ;_L Y$
shows $X \approx_L Y$
⟨*proof*⟩

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

lemma *lens-equiv-iff-bij*:
assumes *weak-lens* Y
shows $X \approx_L Y \longleftrightarrow (\exists\ Z.\ \textit{bij-lens}\ Z \wedge X = Z ;_L Y)$
⟨*proof*⟩

lemma *pbij-plus-commute*:
$[\![\ a \bowtie b;\ \textit{mwb-lens}\ a;\ \textit{mwb-lens}\ b;\ \textit{pbij-lens}\ (b +_L a)\ ]\!] \Longrightarrow \textit{pbij-lens}\ (a +_L b)$
⟨*proof*⟩

5.5 Lens Override Laws

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

lemma *lens-override-id* [*simp*]:
$S_1 \oplus_L S_2\ \textit{on}\ 1_L = S_2$
⟨*proof*⟩

lemma *lens-override-unit* [*simp*]:
$S_1 \oplus_L S_2\ \textit{on}\ 0_L = S_1$
⟨*proof*⟩

lemma *lens-override-overshadow*:
assumes *mwb-lens* Y $X \subseteq_L Y$
shows $(S_1 \oplus_L S_2\ \textit{on}\ X) \oplus_L S_3\ \textit{on}\ Y = S_1 \oplus_L S_3\ \textit{on}\ Y$
⟨*proof*⟩

lemma *lens-override-irr*:
assumes $X \bowtie Y$
shows $S_1 \oplus_L (S_2 \oplus_L S_3\ \textit{on}\ Y)\ \textit{on}\ X = S_1 \oplus_L S_2\ \textit{on}\ X$
⟨*proof*⟩

lemma *lens-override-overshadow-left*:
assumes *mwb-lens* X
shows $(S_1 \oplus_L S_2\ \textit{on}\ X) \oplus_L S_3\ \textit{on}\ X = S_1 \oplus_L S_3\ \textit{on}\ X$
⟨*proof*⟩

lemma *lens-override-overshadow-right*:
assumes *mwb-lens* X
shows $S_1 \oplus_L (S_2 \oplus_L S_3\ \textit{on}\ X)\ \textit{on}\ X = S_1 \oplus_L S_3\ \textit{on}\ X$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *lens-override-idem* [*simp*]:
vwb-lens $X \Longrightarrow S \oplus_L S$ *on* $X = S$
⟨*proof*⟩

lemma *lens-override-mwb-idem* [*simp*]:
$[\![$ *mwb-lens* X; $S \in \mathcal{S}_X$ $]\!] \Longrightarrow S \oplus_L S$ *on* $X = S$
⟨*proof*⟩

lemma *lens-override-put-right-in*:
$[\![$ *vwb-lens* A; $X \subseteq_L A$ $]\!] \Longrightarrow S_1 \oplus_L (put_X\ S_2\ v)$ *on* $A = put_X\ (S_1 \oplus_L S_2$ *on* $A)\ v$
⟨*proof*⟩

lemma *lens-override-put-right-out*:
$[\![$ *vwb-lens* A; $X \bowtie A$ $]\!] \Longrightarrow S_1 \oplus_L (put_X\ S_2\ v)$ *on* $A = (S_1 \oplus_L S_2$ *on* $A)$
⟨*proof*⟩

lemma *lens-indep-overrideI*:
assumes *vwb-lens* X *vwb-lens* Y $(\bigwedge s_1\ s_2\ s_3.\ s_1 \oplus_L s_2$ *on* $X \oplus_L s_3$ *on* $Y = s_1 \oplus_L s_3$ *on* $Y \oplus_L s_2$ *on* $X)$
shows $X \bowtie Y$
⟨*proof*⟩

lemma *lens-indep-override-def*:
assumes *vwb-lens* X *vwb-lens* Y
shows $X \bowtie Y \longleftrightarrow (\forall\ s_1\ s_2\ s_3.\ s_1 \oplus_L s_2$ *on* $X \oplus_L s_3$ *on* $Y = s_1 \oplus_L s_3$ *on* $Y \oplus_L s_2$ *on* $X)$
⟨*proof*⟩

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

lemma *override-idem-implies-vwb*:
$[\![$ *mwb-lens* X; $\bigwedge s.\ s \oplus_L s$ *on* $X = s$ $]\!] \Longrightarrow$ *vwb-lens* X
⟨*proof*⟩

5.6 Alternative Sublens Characterisation

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

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

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

lemma *sublens'-prop1*:
assumes *vwb-lens* X $X \subseteq_L' Y$
shows $put_X\ (put_Y\ s_1\ (get_Y\ s_2))\ s_3 = put_Y\ s_1\ (get_Y\ (put_X\ s_2\ s_3))$
⟨*proof*⟩

lemma *sublens'-prop2*:
assumes *vwb-lens* X $X \subseteq_L' Y$
shows $get_X\ (put_Y\ s_1\ (get_Y\ s_2)) = get_X\ s_2$
⟨*proof*⟩

lemma *sublens'-prop3*:
assumes *vwb-lens* X *vwb-lens* Y $X \subseteq_L' Y$
shows $put_Y\ \sigma\ (get_Y\ (put_X\ (put_Y\ \varrho\ (get_Y\ \sigma))\ v)) = put_X\ \sigma\ v$

$\langle proof \rangle$

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

lemma *sublens'-implies-sublens*:
  assumes *vwb-lens* X *vwb-lens* Y $X \subseteq_L{}' Y$
  shows $X \subseteq_L Y$
$\langle proof \rangle$

lemma *sublens-implies-sublens'*:
  assumes *vwb-lens* Y $X \subseteq_L Y$
  shows $X \subseteq_L{}' Y$
  $\langle proof \rangle$

lemma *sublens-iff-sublens'*:
  assumes *vwb-lens* X *vwb-lens* Y
  shows $X \subseteq_L Y \longleftrightarrow X \subseteq_L{}' Y$
  $\langle proof \rangle$

We can also prove the closure law for lens quotient

lemma *lens-quotient-vwb*: $[\![$ *vwb-lens* x; *vwb-lens* y; $x \subseteq_L y$ $]\!] \Longrightarrow$ *vwb-lens* $(x /_L y)$
  $\langle proof \rangle$

lemma *lens-quotient-indep*:
  $[\![$ *vwb-lens* x; *vwb-lens* y; *vwb-lens* a; $x \bowtie y$; $x \subseteq_L a$; $y \subseteq_L a$ $]\!] \Longrightarrow (x /_L a) \bowtie (y /_L a)$
  $\langle proof \rangle$

lemma *lens-quotient-bij*: $[\![$ *vwb-lens* x; *vwb-lens* y; $y \approx_L x$ $]\!] \Longrightarrow$ *bij-lens* $(x /_L y)$
  $\langle proof \rangle$

5.7 Alternative Equivalence Characterisation

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

lemma *lens-equiv-iff-lens-equiv'*:
  assumes *vwb-lens* X *vwb-lens* Y
  shows $X \approx_L Y \longleftrightarrow X \approx_L{}' Y$
  $\langle proof \rangle$

5.8 Ineffectual Lenses as Zero Elements

lemma *ief-lens-then-zero*: *ief-lens* $x \Longrightarrow x \approx_L 0_L$
  $\langle proof \rangle$

lemma *ief-lens-iff-zero*: *vwb-lens* $x \Longrightarrow$ *ief-lens* $x \longleftrightarrow x \approx_L 0_L$
  $\langle proof \rangle$

end

6 Symmetric Lenses

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

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

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

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

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

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

lemma *view-slens-compl* [*simp*]: $\mathcal{V}_{-_L\ a} = \mathcal{C}_a$
  ⟨*proof*⟩

lemma *coview-slens-compl* [*simp*]: $\mathcal{C}_{-_L\ a} = \mathcal{V}_a$
  ⟨*proof*⟩

6.1 Partial Symmetric Lenses

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

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

lemma *psym-lens-compl* [*simp*]: *psym-lens a* ⟹ *psym-lens* ($-_L$ *a*)
  ⟨*proof*⟩

6.2 Symmetric Lenses

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

sublocale *psym-lens*
⟨*proof*⟩

lemma *put-region-coregion-cover*:
  $put_{\mathcal{V}}\ (put_{\mathcal{C}}\ s_1\ (get_{\mathcal{C}}\ s_2))\ (get_{\mathcal{V}}\ s_2) = s_2$
⟨*proof*⟩

end

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

lemma *sym-lens-psym* [*simp*]: *sym-lens* $x \Longrightarrow$ *psym-lens* x
⟨*proof*⟩

lemma *sym-lens-compl* [*simp*]: *sym-lens* $a \Longrightarrow$ *sym-lens* $(-_L\ a)$
⟨*proof*⟩

end

7 Scenes

theory *Scenes*
imports *Lens-Symmetric*
begin

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

7.1 Overriding Functions

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

locale *overrider* =
fixes F :: $'s \Rightarrow 's \Rightarrow 's$ (**infixl** ‹▷› *65*)
assumes
ovr-overshadow-left: $x \triangleright y \triangleright z = x \triangleright z$ **and**
ovr-overshadow-right: $x \triangleright (y \triangleright z) = x \triangleright z$
begin
lemma *ovr-assoc*: $x \triangleright (y \triangleright z) = x \triangleright y \triangleright z$
⟨*proof*⟩
end

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

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

7.2 Scene Type

typedef $'s$ *scene* = $\{F :: 's \Rightarrow 's \Rightarrow 's.$ *overrider* $F\}$
⟨*proof*⟩

setup-lifting *type-definition-scene*

lift-definition *idem-scene* :: *'s scene* ⇒ *bool* **is** *idem-overrider* ⟨*proof*⟩

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

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

lemma *equiv-region*: *equiv UNIV* (*region X*)
⟨*proof*⟩

lemma *equiv-coregion*: *equiv UNIV* (*coregion X*)
⟨*proof*⟩

lemma *region-coregion-Id*:
idem-scene X ⟹ *region X* ∩ *coregion X* = *Id*
⟨*proof*⟩

lemma *state-eq-iff*: *idem-scene S* ⟹ *x* = *y* ⟷ (*x*, *y*) ∈ *region S* ∧ (*x*, *y*) ∈ *coregion S*
⟨*proof*⟩

lift-definition *scene-override* :: *'a* ⇒ *'a* ⇒ (*'a scene*) ⇒ *'a* (‹- $\oplus_S$ - *on* -› [*95*,*0*,*96*] *95*)
is $\lambda\ s_1\ s_2\ F.\ F\ s_1\ s_2$ ⟨*proof*⟩

abbreviation (*input*) *scene-copy* :: *'a scene* ⇒ *'a* ⇒ (*'a* ⇒ *'a*) (‹*cp*-›) **where**
$cp_A\ s \equiv (\lambda\ s'.\ s' \oplus_S s\ on\ A)$

lemma *scene-override-idem* [*simp*]: *idem-scene X* ⟹ $s \oplus_S s$ *on X* = *s*
⟨*proof*⟩

lemma *scene-override-overshadow-left* [*simp*]:
$S_1 \oplus_S S_2$ *on X* $\oplus_S S_3$ *on X* = $S_1 \oplus_S S_3$ *on X*
⟨*proof*⟩

lemma *scene-override-overshadow-right* [*simp*]:
$S_1 \oplus_S (S_2 \oplus_S S_3$ *on X*) *on X* = $S_1 \oplus_S S_3$ *on X*
⟨*proof*⟩

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

lemma *scene-equiv-region*: *idem-scene X* ⟹ *region X* = $\{(S_1, S_2).\ S_1 \approx_S S_2$ *on X*$\}$
⟨*proof*⟩

lift-definition *scene-indep* :: *'a scene* ⇒ *'a scene* ⇒ *bool* (**infix** ‹$\bowtie_S$› *50*)
is λ *F G*. $(\forall\ s_1\ s_2\ s_3.\ G\ (F\ s_1\ s_2)\ s_3 = F\ (G\ s_1\ s_3)\ s_2)$ ⟨*proof*⟩

lemma *scene-indep-override*:
$X \bowtie_S Y = (\forall\ s_1\ s_2\ s_3.\ s_1 \oplus_S s_2$ *on X* $\oplus_S s_3$ *on Y* = $s_1 \oplus_S s_3$ *on Y* $\oplus_S s_2$ *on X*)
⟨*proof*⟩

lemma *scene-indep-copy*:
$X \bowtie_S Y = (\forall\ s_1\ s_2.\ cp_X\ s_1 \circ cp_Y\ s_2 = cp_Y\ s_2 \circ cp_X\ s_1)$

⟨*proof*⟩

lemma *scene-indep-sym*:
  $X \bowtie_S Y \Longrightarrow Y \bowtie_S X$
  ⟨*proof*⟩

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

lift-definition *scene-compat* :: *'a scene* ⇒ *'a scene* ⇒ *bool* (**infix** ‹$\#\#_S$› *50*)
is $\lambda\ F\ G.\ (\forall\ s_1\ s_2.\ G\ (F\ s_1\ s_2)\ s_2 = F\ (G\ s_1\ s_2)\ s_2)$ ⟨*proof*⟩

lemma *scene-compat-copy*:
  $X\ \#\#_S\ Y = (\forall\ s.\ cp_X\ s \circ cp_Y\ s = cp_Y\ s \circ cp_X\ s)$
  ⟨*proof*⟩

lemma *scene-indep-compat* [*simp*]: $X \bowtie_S Y \Longrightarrow X\ \#\#_S\ Y$
  ⟨*proof*⟩

lemma *scene-compat-refl*: $X\ \#\#_S\ X$
  ⟨*proof*⟩

lemma *scene-compat-sym*: $X\ \#\#_S\ Y \Longrightarrow Y\ \#\#_S\ X$
  ⟨*proof*⟩

lemma *scene-override-commute-indep*:
  assumes $X \bowtie_S Y$
  shows $S_1 \oplus_S S_2\ on\ X \oplus_S S_3\ on\ Y = S_1 \oplus_S S_3\ on\ Y \oplus_S S_2\ on\ X$
  ⟨*proof*⟩

instantiation *scene* :: (*type*) {*bot*, *top*, *uminus*, *sup*, *inf*}
begin
  lift-definition *bot-scene* :: *'a scene* **is** $\lambda\ x\ y.\ x$ ⟨*proof*⟩
  lift-definition *top-scene* :: *'a scene* **is** $\lambda\ x\ y.\ y$ ⟨*proof*⟩
  lift-definition *uminus-scene* :: *'a scene* ⇒ *'a scene* **is** $\lambda\ F\ x\ y.\ F\ y\ x$
    ⟨*proof*⟩

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

  lift-definition *sup-scene* :: *'a scene* ⇒ *'a scene* ⇒ *'a scene*
    is $\lambda\ F\ G.\ if\ (\forall\ s_1\ s_2.\ G\ (F\ s_1\ s_2)\ s_2 = F\ (G\ s_1\ s_2)\ s_2)\ then\ (\lambda\ s_1\ s_2.\ G\ (F\ s_1\ s_2)\ s_2)\ else\ (\lambda\ s_1\ s_2.\ s_1)$
    ⟨*proof*⟩
  definition *inf-scene* :: *'a scene* ⇒ *'a scene* ⇒ *'a scene* **where**
    [*lens-defs*]: $inf\text{-}scene\ X\ Y = -\ (sup\ (-\ X)\ (-\ Y))$
  instance ⟨*proof*⟩
end

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

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

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

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

instantiation *scene* :: (*type*) *minus*
begin
definition *minus-scene* :: *'a scene* ⇒ *'a scene* ⇒ *'a scene* **where**
minus-scene A B = $A \sqcap_S (- B)$
instance ⟨*proof*⟩
end

lemma *bot-idem-scene* [*simp*]: *idem-scene* $\bot_S$
⟨*proof*⟩

lemma *top-idem-scene* [*simp*]: *idem-scene* $\top_S$
⟨*proof*⟩

lemma *uminus-top-scene* [*simp*]: $- \top_S = \bot_S$
⟨*proof*⟩

lemma *uminus-bot-scene* [*simp*]: $- \bot_S = \top_S$
⟨*proof*⟩

lemma *uminus-scene-twice*: $- (- (X :: {'s\ scene})) = X$
⟨*proof*⟩

lemma *scene-override-id* [*simp*]: $S_1 \oplus_S S_2$ *on* $\top_S = S_2$
⟨*proof*⟩

lemma *scene-override-unit* [*simp*]: $S_1 \oplus_S S_2$ *on* $\bot_S = S_1$
⟨*proof*⟩

lemma *scene-override-commute*: $S_2 \oplus_S S_1$ *on* $(- X) = S_1 \oplus_S S_2$ *on* X
⟨*proof*⟩

lemma *scene-union-incompat*: $\neg X \mathbin{\#\#_S} Y \Longrightarrow X \sqcup_S Y = \bot_S$
⟨*proof*⟩

lemma *scene-override-union*: $X \mathbin{\#\#_S} Y \Longrightarrow S_1 \oplus_S S_2$ *on* $(X \sqcup_S Y) = (S_1 \oplus_S S_2$ *on* $X) \oplus_S S_2$ *on* Y
⟨*proof*⟩

lemma *scene-override-inter*: $-X \mathbin{\#\#_S} -Y \Longrightarrow S_1 \oplus_S S_2$ *on* $(X \sqcap_S Y) = S_1 \oplus_S S_1 \oplus_S S_2$ *on* X *on* Y
⟨*proof*⟩

lemma *scene-equiv-bot* [*simp*]: $a \approx_S b$ *on* $\bot_S$
⟨*proof*⟩

lemma *scene-equiv-refl* [*simp*]: *idem-scene* $a \Longrightarrow s \approx_S s$ *on* a
⟨*proof*⟩

lemma *scene-equiv-sym* [*simp*]: *idem-scene* $a \Longrightarrow s_1 \approx_S s_2$ *on* $a \Longrightarrow s_2 \approx_S s_1$ *on* a
⟨*proof*⟩

lemma *scene-union-unit* [*simp*]: $X \sqcup_S \bot_S = X$ $\bot_S \sqcup_S X = X$

⟨*proof*⟩

lemma *scene-indep-bot* [*simp*]: $X \bowtie_S \bot_S$
⟨*proof*⟩

A unitary scene admits only one element, and therefore top and bottom are the same.

lemma *unit-scene-top-eq-bot*: $(\bot_S :: \textit{unit scene}) = \top_S$
⟨*proof*⟩

lemma *idem-scene-union* [*simp*]: $[\![\textit{idem-scene } A; \textit{idem-scene } B]\!] \Longrightarrow \textit{idem-scene } (A \sqcup_S B)$
⟨*proof*⟩

lemma *scene-union-annhil*: $\textit{idem-scene } X \Longrightarrow X \sqcup_S \top_S = \top_S$
⟨*proof*⟩

lemma *scene-union-pres-compat*: $[\![A \ \#\#_S \ B; A \ \#\#_S \ C]\!] \Longrightarrow A \ \#\#_S \ (B \sqcup_S C)$
⟨*proof*⟩

lemma *scene-indep-pres-compat*: $[\![A \bowtie_S B; A \bowtie_S C]\!] \Longrightarrow A \bowtie_S (B \sqcup_S C)$
⟨*proof*⟩

lemma *scene-indep-self-compl*: $A \bowtie_S -A$
⟨*proof*⟩

lemma *scene-compat-self-compl*: $A \ \#\#_S \ -A$
⟨*proof*⟩

lemma *scene-compat-bot* [*simp*]: $a \ \#\#_S \ \bot_S \ \bot_S \ \#\#_S \ a$
⟨*proof*⟩

lemma *scene-compat-top* [*simp*]:
$\textit{idem-scene } a \Longrightarrow a \ \#\#_S \ \top_S$
$\textit{idem-scene } a \Longrightarrow \top_S \ \#\#_S \ a$
⟨*proof*⟩

lemma *scene-union-assoc*:
assumes $X \ \#\#_S \ Y \ X \ \#\#_S \ Z \ Y \ \#\#_S \ Z$
shows $X \sqcup_S (Y \sqcup_S Z) = (X \sqcup_S Y) \sqcup_S Z$
⟨*proof*⟩

lemma *scene-inter-indep*:
assumes $\textit{idem-scene } X \ \textit{idem-scene } Y \ X \bowtie_S Y$
shows $X \sqcap_S Y = \bot_S$
⟨*proof*⟩

lemma *scene-union-indep-uniq*:
assumes $\textit{idem-scene } X \ \textit{idem-scene } Y \ \textit{idem-scene } Z \ X \bowtie_S Z \ Y \bowtie_S Z \ X \sqcup_S Z = Y \sqcup_S Z$
shows $X = Y$
⟨*proof*⟩

lemma *scene-union-idem*: $X \sqcup_S X = X$
⟨*proof*⟩

lemma *scene-union-compl*: $\textit{idem-scene } X \Longrightarrow X \sqcup_S - X = \top_S$
⟨*proof*⟩

lemma *scene-inter-idem*: $X \sqcap_S X = X$
⟨*proof*⟩

lemma *scene-union-commute*: $X \sqcup_S Y = Y \sqcup_S X$
⟨*proof*⟩

lemma *scene-inter-compl*: *idem-scene* $X \Longrightarrow X \sqcap_S - X = \bot_S$
⟨*proof*⟩

lemma *scene-demorgan1*: $-(X \sqcup_S Y) = -X \sqcap_S -Y$
⟨*proof*⟩

lemma *scene-demorgan2*: $-(X \sqcap_S Y) = -X \sqcup_S -Y$
⟨*proof*⟩

lemma *scene-inter-commute*: $X \sqcap_S Y = Y \sqcap_S X$
⟨*proof*⟩

lemma *scene-union-inter-distrib*:
$[\![$ *idem-scene* x; $x \bowtie_S y$; $x \bowtie_S z$; $y \#\#_S z$ $]\!] \Longrightarrow x \sqcup_S y \sqcap_S z = (x \sqcup_S y) \sqcap_S (x \sqcup_S z)$
⟨*proof*⟩

lemma *idem-scene-uminus* [*simp*]: *idem-scene* $X \Longrightarrow$ *idem-scene* $(- X)$
⟨*proof*⟩

lemma *scene-minus-cancel*: $[\![$ $a \bowtie_S b$; *idem-scene* a; *idem-scene* b $]\!] \Longrightarrow a \sqcup_S (b \sqcap_S - a) = a \sqcup_S b$
⟨*proof*⟩

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

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

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

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

lemma *subscene-refl*: $X \subseteq_S X$
⟨*proof*⟩

lemma *subscene-trans*: $[\![$ *idem-scene* Y; $X \subseteq_S Y$; $Y \subseteq_S Z$ $]\!] \Longrightarrow X \subseteq_S Z$
⟨*proof*⟩

lemma *subscene-antisym*: $[\![$ *idem-scene* Y; $X \subseteq_S Y$; $Y \subseteq_S X$ $]\!] \Longrightarrow X = Y$
⟨*proof*⟩

lemma *subscene-copy-def*:

assumes *idem-scene X idem-scene Y*
shows $X \subseteq_S Y = (\forall\ s_1\ s_2.\ cp_X\ s_1 \circ cp_Y\ s_2 = cp_Y\ (cp_X\ s_1\ s_2))$
⟨*proof*⟩

lemma *subscene-eliminate*:
$[\![$ *idem-scene* $Y;\ X \leq Y\ ]\!] \Longrightarrow s_1 \oplus_S s_2$ *on* $X \oplus_S s_3$ *on* $Y = s_1 \oplus_S s_3$ *on* Y
⟨*proof*⟩

lemma *scene-bot-least*: $\bot_S \leq X$
⟨*proof*⟩

lemma *scene-top-greatest*: $X \leq \top_S$
⟨*proof*⟩

lemma *scene-union-ub*: $[\![$ *idem-scene* $Y;\ X \bowtie_S Y\ ]\!] \Longrightarrow X \leq (X \sqcup_S Y)$
⟨*proof*⟩

lemma *scene-union-lb*: $[\![\ a\ \#\#_S\ b;\ a \leq c;\ b \leq c\ ]\!] \Longrightarrow a \sqcup_S b \leq c$
⟨*proof*⟩

lemma *scene-union-mono*: $[\![\ a \subseteq_S c;\ b \subseteq_S c;\ a\ \#\#_S\ b;$ *idem-scene* $a;$ *idem-scene* $b\ ]\!] \Longrightarrow a \sqcup_S b \subseteq_S c$
⟨*proof*⟩

lemma *scene-le-then-compat*: $[\![$ *idem-scene* $X;$ *idem-scene* $Y;\ X \leq Y\ ]\!] \Longrightarrow X\ \#\#_S\ Y$
⟨*proof*⟩

lemma *indep-then-compl-in*: $A \bowtie_S B \Longrightarrow A \leq -B$
⟨*proof*⟩

lemma *scene-le-iff-indep-inv*:
$A \bowtie_S - B \longleftrightarrow A \leq B$
⟨*proof*⟩

lift-definition *scene-comp* :: *'a scene* ⇒ (*'a* ⟹ *'b*) ⇒ *'b scene* (**infixl** ‹$;_S$› *80*)
is $\lambda\ S\ X\ a\ b.$ *if* (*vwb-lens* X) *then* $put_X\ a\ (S\ (get_X\ a)\ (get_X\ b))$ *else* a
⟨*proof*⟩

lemma *scene-comp-idem* [*simp*]: *idem-scene* $S \Longrightarrow$ *idem-scene* $(S\ ;_S\ X)$
⟨*proof*⟩

lemma *scene-comp-lens-indep* [*simp*]: $X \bowtie Y \Longrightarrow (A\ ;_S\ X) \bowtie_S (A\ ;_S\ Y)$
⟨*proof*⟩

lemma *scene-comp-indep* [*simp*]: $A \bowtie_S B \Longrightarrow (A\ ;_S\ X) \bowtie_S (B\ ;_S\ X)$
⟨*proof*⟩

lemma *scene-comp-bot* [*simp*]: $\bot_S\ ;_S\ x = \bot_S$
⟨*proof*⟩

lemma *scene-comp-id-lens* [*simp*]: $A\ ;_S\ 1_L = A$
⟨*proof*⟩

lemma *scene-union-comp-distl*: $a\ \#\#_S\ b \Longrightarrow (a \sqcup_S b)\ ;_S\ x = (a\ ;_S\ x) \sqcup_S (b\ ;_S\ x)$
⟨*proof*⟩

lemma *scene-comp-assoc*: $[\![$ *vwb-lens* X; *vwb-lens* Y $]\!] \Longrightarrow A ;_S X ;_S Y = A ;_S (X ;_L Y)$
⟨*proof*⟩

lift-definition *scene-quotient* :: $'b$ *scene* $\Rightarrow ('a \Longrightarrow 'b) \Rightarrow 'a$ *scene* (**infixl** ‹'/$_S$› *80*)
is λ S X a b. *if* (*vwb-lens* $X \wedge (\forall s_1\ s_2\ s_3.\ S\ (s_1 \triangleleft_X s_2)\ s_3 = s_1 \triangleleft_X S\ s_2\ s_3)$) *then* get_X (S ($create_X$ a) ($create_X$ b)) *else* a
⟨*proof*⟩

lemma *scene-quotient-idem*: *idem-scene* $S \Longrightarrow$ *idem-scene* $(S /_S X)$
⟨*proof*⟩

lemma *scene-quotient-indep*: $A \bowtie_S B \Longrightarrow (A /_S X) \bowtie_S (B /_S X)$
⟨*proof*⟩

lemma *scene-bot-quotient* [*simp*]: $\bot_S /_S X = \bot_S$
⟨*proof*⟩

lemma *scene-comp-quotient*: *vwb-lens* $X \Longrightarrow (A ;_S X) /_S X = A$
⟨*proof*⟩

lemma *scene-quot-id-lens* [*simp*]: $(A /_S 1_L) = A$
⟨*proof*⟩

7.3 Linking Scenes and Lenses

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

lift-definition *lens-scene* :: $('v \Longrightarrow 's) \Rightarrow 's$ *scene* (‹$[\![$-$]\!]_\sim$›) **is**
λ X s_1 s_2. *if* (*mwb-lens* X) *then* $s_1 \oplus_L s_2$ *on* X *else* s_1
⟨*proof*⟩

lemma *vwb-impl-idem-scene* [*simp*]:
vwb-lens $X \Longrightarrow$ *idem-scene* $[\![X]\!]_\sim$
⟨*proof*⟩

lemma *idem-scene-impl-vwb*:
$[\![$ *mwb-lens* X; *idem-scene* $[\![X]\!]_\sim$ $]\!] \Longrightarrow$ *vwb-lens* X
⟨*proof*⟩

lemma *lens-compat-scene*: $[\![$ *mwb-lens* X; *mwb-lens* Y $]\!] \Longrightarrow X \#\#_L Y \longleftrightarrow [\![X]\!]_\sim \#\#_S [\![Y]\!]_\sim$
⟨*proof*⟩

Next we show some important congruence properties

lemma *zero-lens-scene*: $[\![0_L]\!]_\sim = \bot_S$
⟨*proof*⟩

lemma *one-lens-scene*: $[\![1_L]\!]_\sim = \top_S$
⟨*proof*⟩

lemma *scene-comp-top-scene* [*simp*]: *vwb-lens* $x \Longrightarrow \top_S ;_S x = [\![x]\!]_\sim$
⟨*proof*⟩

lemma *scene-comp-lens-scene-indep* [*simp*]: $x \bowtie y \Longrightarrow [\![x]\!]_\sim \bowtie_S a ;_S y$
⟨*proof*⟩

lemma *lens-scene-override*:

$mwb\text{-}lens\ X \Longrightarrow s_1 \oplus_S s_2\ on\ [\![X]\!]_\sim = s_1 \oplus_L s_2\ on\ X$
⟨proof⟩

lemma *lens-indep-scene*:
assumes *vwb-lens X vwb-lens Y*
shows $(X \bowtie Y) \longleftrightarrow [\![X]\!]_\sim \bowtie_S [\![Y]\!]_\sim$
⟨proof⟩

lemma *lens-indep-impl-scene-indep* [*simp*]:
$(X \bowtie Y) \Longrightarrow [\![X]\!]_\sim \bowtie_S [\![Y]\!]_\sim$
⟨proof⟩

lemma *get-scene-override-indep*: $[\![\ vwb\text{-}lens\ x;\ [\![x]\!]_\sim \bowtie_S a\ ]\!] \Longrightarrow get_x\ (s \oplus_S s'\ on\ a) = get_x\ s$
⟨proof⟩

lemma *put-scene-override-indep*:
$[\![\ vwb\text{-}lens\ x;\ [\![x]\!]_\sim \bowtie_S a\ ]\!] \Longrightarrow put_x\ s\ v \oplus_S s'\ on\ a = put_x\ (s \oplus_S s'\ on\ a)\ v$
⟨proof⟩

lemma *get-scene-override-le*: $[\![\ vwb\text{-}lens\ x;\ [\![x]\!]_\sim \le a\ ]\!] \Longrightarrow get_x\ (s \oplus_S s'\ on\ a) = get_x\ s'$
⟨proof⟩

lemma *put-scene-override-le*: $[\![\ vwb\text{-}lens\ x;\ idem\text{-}scene\ a;\ [\![x]\!]_\sim \le a\ ]\!] \Longrightarrow put_x\ s\ v \oplus_S s'\ on\ a = s \oplus_S s'\ on\ a$
⟨proof⟩

lemma *put-scene-override-le-distrib*:
$[\![\ vwb\text{-}lens\ x;\ idem\text{-}scene\ A;\ [\![x]\!]_\sim \le A\ ]\!] \Longrightarrow put_x\ (s_1 \oplus_S s_2\ on\ A)\ v = (put_x\ s_1\ v) \oplus_S (put_x\ s_2\ v)\ on\ A$
⟨proof⟩

lemma *lens-plus-scene*:
$[\![\ vwb\text{-}lens\ X;\ vwb\text{-}lens\ Y;\ X \bowtie Y\ ]\!] \Longrightarrow [\![X +_L Y]\!]_\sim = [\![X]\!]_\sim \sqcup_S [\![Y]\!]_\sim$
⟨proof⟩

lemma *subscene-implies-sublens'*: $[\![\ vwb\text{-}lens\ X;\ vwb\text{-}lens\ Y\ ]\!] \Longrightarrow [\![X]\!]_\sim \le [\![Y]\!]_\sim \longleftrightarrow X \subseteq_L' Y$
⟨proof⟩

lemma *sublens'-implies-subscene*: $[\![\ vwb\text{-}lens\ X;\ vwb\text{-}lens\ Y;\ X \subseteq_L' Y\ ]\!] \Longrightarrow [\![X]\!]_\sim \le [\![Y]\!]_\sim$
⟨proof⟩

lemma *sublens-iff-subscene*:
assumes *vwb-lens X vwb-lens Y*
shows $X \subseteq_L Y \longleftrightarrow [\![X]\!]_\sim \le [\![Y]\!]_\sim$
⟨proof⟩

lemma *lens-scene-indep-compl* [*simp*]:
assumes *vwb-lens x vwb-lens y*
shows $[\![x]\!]_\sim \bowtie_S - [\![y]\!]_\sim \longleftrightarrow x \subseteq_L y$
⟨proof⟩

lemma *lens-scene-comp*: $[\![\ vwb\text{-}lens\ X;\ vwb\text{-}lens\ Y\ ]\!] \Longrightarrow [\![X ;_L Y]\!]_\sim = [\![X]\!]_\sim ;_S Y$
⟨proof⟩

lemma *scene-comp-pres-indep*: $[\![\ idem\text{-}scene\ a;\ idem\text{-}scene\ b;\ a \bowtie_S [\![x]\!]_\sim\ ]\!] \Longrightarrow a \bowtie_S b ;_S x$

⟨*proof*⟩

lemma *scene-comp-le*: $A ;_S X \leq [\![X]\!]_\sim$
⟨*proof*⟩

lemma *scene-quotient-comp*: $[\![$ *vwb-lens* X; *idem-scene* A; $A \leq [\![X]\!]_\sim$ $]\!] \Longrightarrow (A /_S X) ;_S X = A$
⟨*proof*⟩

lemma *lens-scene-quotient*: $[\![$ *vwb-lens* Y; $X \subseteq_L Y$ $]\!] \Longrightarrow [\![X /_L Y]\!]_\sim = [\![X]\!]_\sim /_S Y$
⟨*proof*⟩

lemma *scene-union-quotient*: $[\![A \mathbin{\#\#_S} B; A \leq [\![X]\!]_\sim; B \leq [\![X]\!]_\sim]\!] \Longrightarrow (A \sqcup_S B) /_S X = (A /_S X) \sqcup_S (B /_S X)$
⟨*proof*⟩

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

lemma *lens-equiv-scene*:
assumes *vwb-lens* X *vwb-lens* Y
shows $X \approx_L Y \longleftrightarrow [\![X]\!]_\sim = [\![Y]\!]_\sim$
⟨*proof*⟩

lemma *lens-scene-top-iff-bij-lens*: *mwb-lens* $x \Longrightarrow [\![x]\!]_\sim = \top_S \longleftrightarrow$ *bij-lens* x
⟨*proof*⟩

7.4 Function Domain Scene

lift-definition *fun-dom-scene* :: *'a set* ⇒ (*'a* ⇒ *'b::two*) *scene* (‹*fds*›) **is**
λ *A f g. override-on f g A* ⟨*proof*⟩

lemma *fun-dom-scene-empty*: $fds(\{\}) = \bot_S$
⟨*proof*⟩

lemma *fun-dom-scene-union*: $fds(A \cup B) = fds(A) \sqcup_S fds(B)$
⟨*proof*⟩

lemma *fun-dom-scene-compl*: $fds(- A) = - fds(A)$
⟨*proof*⟩

lemma *fun-dom-scene-inter*: $fds(A \cap B) = fds(A) \sqcap_S fds(B)$
⟨*proof*⟩

lemma *fun-dom-scene-UNIV*: $fds(UNIV) = \top_S$
⟨*proof*⟩

lemma *fun-dom-scene-indep* [*simp*]:
$fds(A) \bowtie_S fds(B) \longleftrightarrow A \cap B = \{\}$
⟨*proof*⟩

lemma *fun-dom-scene-always-compat* [*simp*]: $fds(A) \mathbin{\#\#_S} fds(B)$
⟨*proof*⟩

lemma *fun-dom-scene-le* [*simp*]: $fds(A) \subseteq_S fds(B) \longleftrightarrow A \subseteq B$
⟨*proof*⟩

Hide implementation details for scenes

lifting-update *scene.lifting*

lifting-forget *scene.lifting*

end

8 Scene Spaces

theory *Scene-Spaces*
  imports *Scenes*
begin

8.1 Preliminaries

abbreviation *foldr-scene* :: *'a scene list* ⇒ *'a scene* (‹$\bigsqcup_S$›) **where**
foldr-scene as ≡ *foldr* ($\sqcup_S$) *as* $\bot_S$

lemma *pairwise-indep-then-compat* [*simp*]: *pairwise* ($\bowtie_S$) *A* ⟹ *pairwise* ($\#\#_S$) *A*
  ⟨*proof*⟩

lemma *pairwise-compat-foldr*:
  ⟦ *pairwise* ($\#\#_S$) (*set as*); ∀ *b* ∈ *set as*. *a* $\#\#_S$ *b* ⟧ ⟹ *a* $\#\#_S$ $\bigsqcup_S$ *as*
  ⟨*proof*⟩

lemma *foldr-scene-indep*:
  ⟦ *pairwise* ($\#\#_S$) (*set as*); ∀ *b* ∈ *set as*. *a* $\bowtie_S$ *b* ⟧ ⟹ *a* $\bowtie_S$ $\bigsqcup_S$ *as*
  ⟨*proof*⟩

lemma *foldr-compat-dist*:
  pairwise ($\#\#_S$) (*set as*) ⟹ *foldr* ($\sqcup_S$) (*map* (λ*a*. *a* $;_S$ *x*) *as*) $\bot_S$ = $\bigsqcup_S$ *as* $;_S$ *x*
  ⟨*proof*⟩

lemma *foldr-compat-quotient-dist*:
  ⟦ *pairwise* ($\#\#_S$) (*set as*); ∀ *a*∈*set as*. *a* ≤ ⟦*x*⟧$_\sim$ ⟧ ⟹ *foldr* ($\sqcup_S$) (*map* (λ*a*. *a* $/_S$ *x*) *as*) $\bot_S$ = $\bigsqcup_S$ *as* $/_S$ *x*
  ⟨*proof*⟩

lemma *foldr-scene-union-add-tail*:
  ⟦ *pairwise* ($\#\#_S$) (*set xs*); ∀ *x*∈*set xs*. *x* $\#\#_S$ *b* ⟧ ⟹ $\bigsqcup_S$ *xs* $\sqcup_S$ *b* = *foldr* ($\sqcup_S$) *xs b*
  ⟨*proof*⟩

lemma *pairwise-Diff*: *pairwise R A* ⟹ *pairwise R* (*A* − *B*)
  ⟨*proof*⟩

lemma *scene-compats-members*: ⟦ *pairwise* ($\#\#_S$) *A*; *x* ∈ *A*; *y* ∈ *A* ⟧ ⟹ *x* $\#\#_S$ *y*
  ⟨*proof*⟩

corollary *foldr-scene-union-removeAll*:
  assumes *pairwise* ($\#\#_S$) (*set xs*) *x* ∈ *set xs*
  shows $\bigsqcup_S$ (*removeAll x xs*) $\sqcup_S$ *x* = $\bigsqcup_S$ *xs*
⟨*proof*⟩

lemma *foldr-scene-union-eq-sets*:
  assumes *pairwise* ($\#\#_S$) (*set xs*) *set xs* = *set ys*
  shows $\bigsqcup_S$ *xs* = $\bigsqcup_S$ *ys*
⟨*proof*⟩

lemma *foldr-scene-removeAll*:
  assumes *pairwise* $(\#\#_S)$ *(set xs)*
  shows $x \sqcup_S \bigsqcup_S$ *(removeAll x xs)* $= x \sqcup_S \bigsqcup_S$ *xs*
  ⟨*proof*⟩

lemma *pairwise-Collect*: *pairwise R A* $\Longrightarrow$ *pairwise R* $\{x \in A.\ P\ x\}$
  ⟨*proof*⟩

lemma *removeAll-overshadow-filter*:
  removeAll x (*filter* (λ*xa. xa* $\notin A - \{x\}$) *xs*) = *removeAll x* (*filter* (λ *xa. xa* $\notin A$) *xs*)
  ⟨*proof*⟩

corollary *foldr-scene-union-filter*:
  assumes *pairwise* $(\#\#_S)$ *(set xs) set ys* $\subseteq$ *set xs*
  shows $\bigsqcup_S$ *xs* $= \bigsqcup_S$ (*filter* ($\lambda x.\ x \notin$ *set ys*) *xs*) $\sqcup_S \bigsqcup_S$ *ys*
⟨*proof*⟩

lemma *foldr-scene-append*:
  ⟦ *pairwise* $(\#\#_S)$ (*set* (*xs* @ *ys*)) ⟧ $\Longrightarrow \bigsqcup_S$ (*xs* @ *ys*) $= \bigsqcup_S$ *xs* $\sqcup_S \bigsqcup_S$ *ys*
  ⟨*proof*⟩

lemma *foldr-scene-concat*:
  ⟦ *pairwise* $(\#\#_S)$ (*set* (*concat xs*)) ⟧ $\Longrightarrow \bigsqcup_S$ (*concat xs*) $= \bigsqcup_S$ (*map* $\bigsqcup_S$ *xs*)
  ⟨*proof*⟩

8.2 Predicates

All scenes in the set are independent

definition *scene-indeps* :: *'s scene set* $\Rightarrow$ *bool* **where**
scene-indeps = *pairwise* $(\bowtie_S)$

All scenes in the set cover the entire state space

definition *scene-span* :: *'s scene list* $\Rightarrow$ *bool* **where**
scene-span S = (*foldr* $(\sqcup_S)$ *S* $\bot_S = \top_S$)

cf. *finite-dimensional-vector-space*, which scene spaces are based on.

8.3 Scene space class

class *scene-space* =
  fixes *Vars* :: *'a scene list*
  assumes *idem-scene-Vars* [*simp*]: $\bigwedge x.\ x \in$ *set Vars* $\Longrightarrow$ *idem-scene x*
  and *indep-Vars*: *scene-indeps* (*set Vars*)
  and *span-Vars*: *scene-span Vars*
begin

lemma *scene-space-compats* [*simp*]: *pairwise* $(\#\#_S)$ (*set Vars*)
  ⟨*proof*⟩

lemma *Vars-ext-lens-indep*: ⟦ $a\ ;_S\ x \neq b\ ;_S\ x$; $a \in$ *set Vars*; $b \in$ *set Vars* ⟧ $\Longrightarrow a\ ;_S\ x \bowtie_S b\ ;_S\ x$
  ⟨*proof*⟩

inductive-set *scene-space* :: *'a scene set* **where**
bot-scene-space [*intro*]: $\bot_S \in$ *scene-space* |
Vars-scene-space [*intro*]: $x \in$ *set Vars* $\Longrightarrow x \in$ *scene-space* |

union-scene-space [*intro*]: $\llbracket x \in \textit{scene-space}; y \in \textit{scene-space} \rrbracket \Longrightarrow x \sqcup_S y \in \textit{scene-space}$

lemma *idem-scene-space*: $a \in \textit{scene-space} \Longrightarrow \textit{idem-scene } a$
⟨*proof*⟩

lemma *set-Vars-scene-space* [*simp*]: $\textit{set Vars} \subseteq \textit{scene-space}$
⟨*proof*⟩

lemma *pairwise-compat-Vars-subset*: $\textit{set xs} \subseteq \textit{set Vars} \Longrightarrow \textit{pairwise } (\#\#_S) \ (\textit{set xs})$
⟨*proof*⟩

lemma *scene-space-foldr*: $\textit{set xs} \subseteq \textit{scene-space} \Longrightarrow \bigsqcup_S xs \in \textit{scene-space}$
⟨*proof*⟩

lemma *top-scene-eq*: $\top_S = \bigsqcup_S \textit{Vars}$
⟨*proof*⟩

lemma *top-scene-space*: $\top_S \in \textit{scene-space}$
⟨*proof*⟩

lemma *Vars-compat-scene-space*: $\llbracket b \in \textit{scene-space}; x \in \textit{set Vars} \rrbracket \Longrightarrow x \ \#\#_S \ b$
⟨*proof*⟩

lemma *scene-space-compat*: $\llbracket a \in \textit{scene-space}; b \in \textit{scene-space} \rrbracket \Longrightarrow a \ \#\#_S \ b$
⟨*proof*⟩

corollary *scene-space-union-assoc*:
assumes $x \in \textit{scene-space}$ $y \in \textit{scene-space}$ $z \in \textit{scene-space}$
shows $x \sqcup_S (y \sqcup_S z) = (x \sqcup_S y) \sqcup_S z$
⟨*proof*⟩

lemma *scene-space-vars-decomp*: $a \in \textit{scene-space} \Longrightarrow \exists xs. \ \textit{set xs} \subseteq \textit{set Vars} \land \textit{foldr } (\sqcup_S) \ xs \ \bot_S = a$
⟨*proof*⟩

lemma *scene-space-vars-decomp-iff*: $a \in \textit{scene-space} \longleftrightarrow (\exists xs. \ \textit{set xs} \subseteq \textit{set Vars} \land a = \textit{foldr } (\sqcup_S) \ xs \ \bot_S)$
⟨*proof*⟩

lemma $\textit{fold } (\sqcup_S) \ (\textit{map } (\lambda x. \ x \ ;_S \ a) \ \textit{Vars}) \ b = \llbracket a \rrbracket_\sim \sqcup_S b$
⟨*proof*⟩

lemma *Vars-indep-foldr*:
assumes $x \in \textit{set Vars}$ $\textit{set xs} \subseteq \textit{set Vars}$
shows $x \bowtie_S \bigsqcup_S (\textit{removeAll } x \ xs)$
⟨*proof*⟩

lemma *Vars-indeps-foldr*:
assumes $\textit{set xs} \subseteq \textit{set Vars}$
shows $\textit{foldr } (\sqcup_S) \ xs \ \bot_S \bowtie_S \textit{foldr } (\sqcup_S) \ (\textit{filter } (\lambda x. \ x \notin \textit{set xs}) \ \textit{Vars}) \ \bot_S$
⟨*proof*⟩

lemma *uminus-var-other-vars*:
assumes $x \in \textit{set Vars}$
shows $- x = \textit{foldr } (\sqcup_S) \ (\textit{removeAll } x \ \textit{Vars}) \ \bot_S$
⟨*proof*⟩

lemma *uminus-vars-other-vars*:
assumes $set\ xs \subseteq set\ Vars$
shows $- \bigsqcup_S xs = \bigsqcup_S (filter\ (\lambda x.\ x \notin set\ xs)\ Vars)$
⟨*proof*⟩

lemma *scene-space-uminus*: $[\![\ a \in scene\text{-}space\ ]\!] \Longrightarrow - a \in scene\text{-}space$
⟨*proof*⟩

lemma *scene-space-inter*: $[\![\ a \in scene\text{-}space;\ b \in scene\text{-}space\ ]\!] \Longrightarrow a \sqcap_S b \in scene\text{-}space$
⟨*proof*⟩

lemma *scene-union-foldr-remove-element*:
assumes $set\ xs \subseteq set\ Vars$
shows $a \sqcup_S \bigsqcup_S xs = a \sqcup_S \bigsqcup_S (removeAll\ a\ xs)$
⟨*proof*⟩

lemma *scene-union-foldr-Cons-removeAll*:
assumes $set\ xs \subseteq set\ Vars\ a \in set\ xs$
shows $foldr\ (\sqcup_S)\ xs\ \bot_S = foldr\ (\sqcup_S)\ (a\ \#\ removeAll\ a\ xs)\ \bot_S$
⟨*proof*⟩

lemma *scene-union-foldr-Cons-removeAll'*:
assumes $set\ xs \subseteq set\ Vars\ a \in set\ Vars$
shows $foldr\ (\sqcup_S)\ (a\ \#\ xs)\ \bot_S = foldr\ (\sqcup_S)\ (a\ \#\ removeAll\ a\ xs)\ \bot_S$
⟨*proof*⟩

lemma *scene-in-foldr*: $[\![\ a \in set\ xs;\ set\ xs \subseteq set\ Vars\ ]\!] \Longrightarrow a \subseteq_S \bigsqcup_S xs$
⟨*proof*⟩

lemma *scene-union-foldr-subset*:
assumes $set\ xs \subseteq set\ ys\ set\ ys \subseteq set\ Vars$
shows $\bigsqcup_S xs \subseteq_S \bigsqcup_S ys$
⟨*proof*⟩

lemma *union-scene-space-foldrs*:
assumes $set\ xs \subseteq set\ Vars\ set\ ys \subseteq set\ Vars$
shows $(foldr\ (\sqcup_S)\ xs\ \bot_S) \sqcup_S (foldr\ (\sqcup_S)\ ys\ \bot_S) = foldr\ (\sqcup_S)\ (xs\ @\ ys)\ \bot_S$
⟨*proof*⟩

lemma *scene-space-ub*:
assumes $a \in scene\text{-}space\ b \in scene\text{-}space$
shows $a \subseteq_S a \sqcup_S b$
⟨*proof*⟩

lemma *scene-compl-subset-iff*:
assumes $a \in scene\text{-}space\ b \in scene\text{-}space$
shows $- a \subseteq_S -b \longleftrightarrow b \subseteq_S a$
⟨*proof*⟩

lemma *inter-scene-space-foldrs*:
assumes $set\ xs \subseteq set\ Vars\ set\ ys \subseteq set\ Vars$
shows $\bigsqcup_S xs \sqcap_S \bigsqcup_S ys = \bigsqcup_S (filter\ (\lambda\ x.\ x \in set\ xs \cap set\ ys)\ Vars)$
⟨*proof*⟩

lemma *scene-inter-distrib-lemma*:
assumes $set\ xs \subseteq set\ Vars\ set\ ys \subseteq set\ Vars\ set\ zs \subseteq set\ Vars$
shows $\bigsqcup_S xs \sqcup_S (\bigsqcup_S ys \sqcap_S \bigsqcup_S zs) = (\bigsqcup_S xs \sqcup_S \bigsqcup_S ys) \sqcap_S (\bigsqcup_S xs \sqcup_S \bigsqcup_S zs)$
⟨*proof*⟩

lemma *scene-union-inter-distrib*:
assumes $a \in scene\text{-}space\ b \in scene\text{-}space\ c \in scene\text{-}space$
shows $a \sqcup_S b \sqcap_S c = (a \sqcup_S b) \sqcap_S (a \sqcup_S c)$
⟨*proof*⟩

lemma *finite-distinct-lists-subset*:
assumes *finite A*
shows $finite\ \{xs.\ distinct\ xs \wedge set\ xs \subseteq A\}$
⟨*proof*⟩

lemma *foldr-scene-union-remdups*: $set\ xs \subseteq set\ Vars \Longrightarrow \bigsqcup_S (remdups\ xs) = \bigsqcup_S xs$
⟨*proof*⟩

lemma *scene-space-as-lists*:
$scene\text{-}space = \{\bigsqcup_S xs \mid xs.\ distinct\ xs \wedge set\ xs \subseteq set\ Vars\}$
⟨*proof*⟩

lemma *finite-scene-space*: *finite scene-space*
⟨*proof*⟩

lemma *scene-space-inter-assoc*:
assumes $x \in scene\text{-}space\ y \in scene\text{-}space\ z \in scene\text{-}space$
shows $(x \sqcap_S y) \sqcap_S z = x \sqcap_S (y \sqcap_S z)$
⟨*proof*⟩

lemma *scene-inter-union-distrib*:
assumes $x \in scene\text{-}space\ y \in scene\text{-}space\ z \in scene\text{-}space$
shows $x \sqcap_S (y \sqcup_S z) = (x \sqcap_S y) \sqcup_S (x \sqcap_S z)$
⟨*proof*⟩

lemma *scene-union-inter-minus*:
assumes $a \in scene\text{-}space\ b \in scene\text{-}space$
shows $a \sqcup_S (b \sqcap_S - a) = a \sqcup_S b$
⟨*proof*⟩

lemma *scene-union-foldr-minus-element*:
assumes $a \in scene\text{-}space\ set\ xs \subseteq scene\text{-}space$
shows $a \sqcup_S \bigsqcup_S xs = a \sqcup_S \bigsqcup_S (map\ (\lambda\ x.\ x \sqcap_S - a)\ xs)$
⟨*proof*⟩

lemma *scene-space-in-foldr*: $[\![\ a \in set\ xs;\ set\ xs \subseteq scene\text{-}space\ ]\!] \Longrightarrow a \subseteq_S \bigsqcup_S xs$
⟨*proof*⟩

lemma *scene-space-foldr-lb*:
$[\![\ a \in scene\text{-}space;\ set\ xs \subseteq scene\text{-}space;\ \forall\ b{\in}set\ xs.\ b \le a\ ]\!] \Longrightarrow \bigsqcup_S xs \subseteq_S a$
⟨*proof*⟩

lemma *var-le-union-choice*:
$[\![\ x \in set\ Vars;\ a \in scene\text{-}space;\ b \in scene\text{-}space;\ x \le a \sqcup_S b\ ]\!] \Longrightarrow (x \le a \vee x \le b)$
⟨*proof*⟩

lemma *var-le-union-iff*:
$\llbracket\ x \in set\ Vars;\ a \in scene\text{-}space;\ b \in scene\text{-}space\ \rrbracket \Longrightarrow x \le a \sqcup_S b \longleftrightarrow (x \le a \vee x \le b)$
⟨*proof*⟩

Vars may contain the empty scene, as we want to allow vacuous lenses in alphabets

lemma *le-vars-then-equal*: $\llbracket\ x \in set\ Vars;\ y \in set\ Vars;\ x \le y;\ x \ne \bot_S\ \rrbracket \Longrightarrow x = y$
⟨*proof*⟩

end

lemma *foldr-scene-union-eq-scene-space*:
$\llbracket\ set\ xs \subseteq scene\text{-}space;\ set\ xs = set\ ys\ \rrbracket \Longrightarrow \bigsqcup_S xs = \bigsqcup_S ys$
⟨*proof*⟩

8.4 Mapping a lens over a scene list

definition *map-lcomp* :: $'b\ scene\ list \Rightarrow ('b \Longrightarrow 'a) \Rightarrow 'a\ scene\ list$ **where**
$map\text{-}lcomp\ ss\ a = map\ (\lambda\ x.\ x\ ;_S\ a)\ ss$

lemma *map-lcomp-dist*:
$\llbracket\ pairwise\ (\#\#_S)\ (set\ xs);\ vwb\text{-}lens\ a\ \rrbracket \Longrightarrow \bigsqcup_S (map\text{-}lcomp\ xs\ a) = \bigsqcup_S xs\ ;_S\ a$
⟨*proof*⟩

lemma *map-lcomp-Vars-is-lens* [*simp*]: $vwb\text{-}lens\ a \Longrightarrow \bigsqcup_S (map\text{-}lcomp\ Vars\ a) = \llbracket a \rrbracket_\sim$
⟨*proof*⟩

lemma *set-map-lcomp* [*simp*]: $set\ (map\text{-}lcomp\ xs\ a) = (\lambda x.\ x\ ;_S\ a)\ `\ set\ xs$
⟨*proof*⟩

8.5 Instances

instantiation *unit* :: *scene-space*
begin

definition *Vars-unit* :: *unit scene list* **where** [*simp*]: $Vars\text{-}unit = []$

instance
⟨*proof*⟩

end

instantiation *prod* :: (*scene-space*, *scene-space*) *scene-space*
begin

definition *Vars-prod* :: $('a \times 'b)\ scene\ list$ **where** $Vars\text{-}prod = map\text{-}lcomp\ Vars\ fst_L\ @\ map\text{-}lcomp\ Vars\ snd_L$

instance ⟨*proof*⟩

end

8.6 Scene space and basis lenses

locale *var-lens* = *vwb-lens* +
assumes *lens-in-scene-space*: $\llbracket x \rrbracket_\sim \in scene\text{-}space$

declare *var-lens.lens-in-scene-space* [*simp*]
declare *var-lens.axioms*(*1*) [*simp*]

locale *basis-lens* = *vwb-lens* +
  assumes *lens-in-basis*: $[\![x]\!]_\sim \in$ *set Vars*

sublocale *basis-lens* ⊆ *var-lens*
  ⟨*proof*⟩

declare *basis-lens.lens-in-basis* [*simp*]

Effectual variable and basis lenses need to have at least two view elements

abbreviation (*input*) *evar-lens* :: (*'a*::*two* ⟹ *'s*::*scene-space*) ⇒ *bool*
  where *evar-lens* ≡ *var-lens*

abbreviation (*input*) *ebasis-lens* :: (*'a*::*two* ⟹ *'s*::*scene-space*) ⇒ *bool*
  where *ebasis-lens* ≡ *basis-lens*

lemma *basis-then-var* [*simp*]: *basis-lens x* ⟹ *var-lens x*
  ⟨*proof*⟩

lemma *basis-lens-intro*: ⟦ *vwb-lens x*; $[\![x]\!]_\sim \in$ *set Vars* ⟧ ⟹ *basis-lens x*
  ⟨*proof*⟩

8.7 Composite lenses

locale *composite-lens* = *vwb-lens* +
  assumes *comp-in-Vars*: (λ *a*. *a* $;_S$ *x*) ‘ *set Vars* ⊆ *set Vars*
begin

lemma *Vars-closed-comp*: *a* ∈ *set Vars* ⟹ *a* $;_S$ *x* ∈ *set Vars*
  ⟨*proof*⟩

lemma *scene-space-closed-comp*:
  assumes *a* ∈ *scene-space*
  shows *a* $;_S$ *x* ∈ *scene-space*
⟨*proof*⟩

sublocale *var-lens*
⟨*proof*⟩

end

lemma *composite-implies-var-lens* [*simp*]:
  composite-lens x ⟹ *var-lens x*
  ⟨*proof*⟩

The extension of any lens in the scene space remains in the scene space

lemma *composite-lens-comp* [*simp*]:
  ⟦ *composite-lens a*; *var-lens x* ⟧ ⟹ *var-lens* (*x* $;_L$ *a*)
  ⟨*proof*⟩

lemma *comp-composite-lens* [*simp*]:
  ⟦ *composite-lens a*; *composite-lens x* ⟧ ⟹ *composite-lens* (*x* $;_L$ *a*)

⟨*proof*⟩

A basis lens within a composite lens remains a basis lens (i.e. it remains atomic)

lemma *composite-lens-basis-comp* [*simp*]:
$[\![$ *composite-lens a*; *basis-lens x* $]\!] \Longrightarrow$ *basis-lens* $(x ;_L a)$
⟨*proof*⟩

lemma *id-composite-lens*: *composite-lens* 1_L
⟨*proof*⟩

lemma *fst-composite-lens*: *composite-lens* fst_L
⟨*proof*⟩

lemma *snd-composite-lens*: *composite-lens* snd_L
⟨*proof*⟩

end

9 Lens Instances

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

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

9.1 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* = $(\lambda\ f.\ f\ x)$, *lens-put* = $(\lambda\ f\ u.\ f(x := u))$ ⦈

lemma *fun-vwb-lens*: *vwb-lens* (*fun-lens* x)
⟨*proof*⟩

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*⟩

9.2 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* = $\lambda\ s.\ get_X \circ get_Y\ s$
, *lens-put* = $\lambda\ s\ v.\ put_Y\ s\ (\lambda\ x::'a.\ put_X\ (get_Y\ s\ x)\ (v\ x))$ ⦈

lemma *fun-ran-mwb-lens*: $[\![$ *mwb-lens X*; *mwb-lens Y* $]\!] \Longrightarrow$ *mwb-lens* (*fun-ran-lens X Y*)

⟨*proof*⟩

lemma *fun-ran-wb-lens*: ⟦ *wb-lens X*; *wb-lens Y* ⟧ ⟹ *wb-lens* (*fun-ran-lens X Y*)
⟨*proof*⟩

lemma *fun-ran-vwb-lens*: ⟦ *vwb-lens X*; *vwb-lens Y* ⟧ ⟹ *vwb-lens* (*fun-ran-lens X Y*)
⟨*proof*⟩

9.3 Map Lens

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

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

lemma *map-mwb-lens*: *mwb-lens* (*map-lens x*)
⟨*proof*⟩

lemma *source-map-lens*: $\mathcal{S}_{map\text{-}lens\ x} = \{f.\ x \in dom(f)\}$
⟨*proof*⟩

lemma *pget-map-lens*: $pget_{map\text{-}lens\ k}\ f = f\ k$
⟨*proof*⟩

9.4 List Lens

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

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

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

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

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

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

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

lemma *list-update-append-lemma1*: *i* < *length xs* ⟹ *xs*[*i* := *v*] @ *ys* = (*xs* @ *ys*)[*i* := *v*]
⟨*proof*⟩

lemma *list-update-append-lemma2*: *i* < *length ys* ⟹ *xs* @ *ys*[*i* := *v*] = (*xs* @ *ys*)[*i* + *length xs* := *v*]
⟨*proof*⟩

We can also prove some laws about our new operators.

lemma *nth'-0* [*simp*]: *nth'* (*x* # *xs*) *0* = *x*

⟨*proof*⟩

lemma *nth′-Suc* [*simp*]: $nth'\ (x\ \#\ xs)\ (Suc\ n) = nth'\ xs\ n$
⟨*proof*⟩

lemma *list-augment-0* [*simp*]:
$list\text{-}augment\ (x\ \#\ xs)\ 0\ y = y\ \#\ xs$
⟨*proof*⟩

lemma *list-augment-Suc* [*simp*]:
$list\text{-}augment\ (x\ \#\ xs)\ (Suc\ n)\ y = x\ \#\ list\text{-}augment\ xs\ n\ y$
⟨*proof*⟩

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

lemma *list-augment-last* [*simp*]:
$list\text{-}augment\ (xs\ @\ [y])\ (length\ xs)\ z = xs\ @\ [z]$
⟨*proof*⟩

lemma *list-augment-idem* [*simp*]:
$i < length\ xs \Longrightarrow list\text{-}augment\ xs\ i\ (xs\ !\ i) = xs$
⟨*proof*⟩

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

lemma *list-augment-commute*:
$i \neq j \Longrightarrow list\text{-}augment\ (list\text{-}augment\ \sigma\ j\ v)\ i\ u = list\text{-}augment\ (list\text{-}augment\ \sigma\ i\ u)\ j\ v$
⟨*proof*⟩

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*: $list\text{-}augment\ xs\ k\ v\ !\ k = v$
⟨*proof*⟩

lemma *nth′-list-augment*: $nth'\ (list\text{-}augment\ xs\ k\ v)\ k = v$
⟨*proof*⟩

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

lemma *length-list-augment-1*: $k \geq length\ xs \Longrightarrow length\ (list\text{-}augment\ xs\ k\ v) = Suc\ k$
⟨*proof*⟩

lemma *length-list-augment-2*: $k < length\ xs \Longrightarrow length\ (list\text{-}augment\ xs\ k\ v) = length\ xs$
⟨*proof*⟩

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

lemma *list-augment-same-twice*: $list\text{-}augment\ (list\text{-}augment\ xs\ k\ u)\ k\ v = list\text{-}augment\ xs\ k\ v$
⟨*proof*⟩

lemma *nth′-list-augment-diff*: $i \neq j \Longrightarrow nth'\ (list\text{-}augment\ \sigma\ i\ v)\ j = nth'\ \sigma\ j$
⟨*proof*⟩

The definition of *list-augment* is not good for code generation, since it produces undefined values even when padding out is not required. Here, we defined a code equation that avoids this.

lemma *list-augment-code* [*code*]:
$list\text{-}augment\ xs\ k\ v = (if\ (k < length\ xs)\ then\ list\text{-}update\ xs\ k\ v\ else\ list\text{-}update\ (list\text{-}pad\text{-}out\ xs\ k)\ k\ v)$
⟨*proof*⟩

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

definition *list-lens* :: *nat* ⇒ (′*a*::*two* ⟹ ′*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* (‹hd_L›) **where** *hd-lens* ≡ *list-lens 0*

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

lemma *list-mwb-lens*: *mwb-lens* (*list-lens x*)
⟨*proof*⟩

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

lemma *source-list-lens*: $\mathcal{S}_{list\text{-}lens\ i} = \{xs.\ length\ xs > i\}$
⟨*proof*⟩

lemma *tail-lens-mwb*:
mwb-lens tl_L
⟨*proof*⟩

lemma *source-tail-lens*: $\mathcal{S}_{tl_L} = \{xs.\ xs \neq []\}$
⟨*proof*⟩

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

lemma *list-lens-indep*:
$i \neq j \Longrightarrow list\text{-}lens\ i \bowtie list\text{-}lens\ j$
⟨*proof*⟩

lemma *hd-tl-lens-indep* [*simp*]:
$hd_L \bowtie tl_L$
⟨*proof*⟩

lemma *hd-tl-lens-pbij*: *pbij-lens* ($hd_L +_L tl_L$)
⟨*proof*⟩

9.5 Stream Lenses

primrec *stream-update* :: ′*a stream* ⇒ *nat* ⇒ ′*a* ⇒ ′*a stream* **where**
$stream\text{-}update\ xs\ 0\ a = a\#\#(stl\ xs)\ |$
$stream\text{-}update\ xs\ (Suc\ n)\ a = shd\ xs\ \#\#\ (stream\text{-}update\ (stl\ xs)\ n\ a)$

lemma *stream-update-snth*: $(stream\text{-}update\ xs\ n\ a)\ !!\ n = a$
⟨*proof*⟩

lemma *stream-update-unchanged*: $i \neq j \Longrightarrow (stream\text{-}update\ xs\ i\ a)\ !!\ j = xs\ !!\ j$
⟨*proof*⟩

lemma *stream-update-override*: *stream-update* (*stream-update xs n a*) *n b* = *stream-update xs n b*
⟨*proof*⟩

lemma *stream-update-nth*: *stream-update* σ *i* (σ !! *i*) = σ
⟨*proof*⟩

definition *stream-lens* :: *nat* ⇒ (*'a::two* ⟹ *'a stream*) **where**
[*lens-defs*]: *stream-lens i* = ⦇ *lens-get* = (λ *xs*. *snth xs i*)
, *lens-put* = (λ *xs x*. *stream-update xs i x*)⦈

lemma *stream-vwb-lens*: *vwb-lens* (*stream-lens i*)
⟨*proof*⟩

9.6 Record Field Lenses

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

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

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

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

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

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

⟨*ML*⟩

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.

named-theorems *lens*

⟨*ML*⟩

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

named-theorems *alpha-splits*

We supply a helpful tactic to remove the subscripted v characters from subgoals. These exist because the internal names of record fields have them.

method *rename-alpha-vars* = *tactic* ‹ *Lens-Utils.rename-alpha-vars* ›

9.7 Locale State Spaces

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

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

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

⟨*ML*⟩

9.8 Type Definition Lens

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

context *type-definition*
begin

definition *typedef-lens* :: $'b \Longrightarrow 'a$ (‹$typedef_L$›) **where**
[*lens-defs*]: $typedef_L = ⦇$ *lens-get* = *Abs*, *lens-put* = (λ *s*. *Rep*) ⦈

lemma *pbij-typedef-lens* [*simp*]: *pbij-lens* $typedef_L$
⟨*proof*⟩

lemma *source-typedef-lens*: $\mathcal{S}_{typedef_L} = A$
⟨*proof*⟩

lemma *bij-typedef-lens-UNIV*: $A = UNIV \Longrightarrow$ *bij-lens* $typedef_L$
⟨*proof*⟩

end

9.9 Mapper Lenses

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

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

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

⟨*ML*⟩

9.10 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

9.11 Tactic

A simple tactic for simplifying lens expressions

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

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

end

10 Lenses

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

11 Prisms

theory *Prisms*
imports *Lenses*
begin

11.1 Signature and Axioms

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

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

type-notation
prism (**infixr** ‹$\Longrightarrow_{\triangle}$› *0*)

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

  lemma *build-match-iff*: *match s* = *Some v* $\longleftrightarrow$ *s* = *build v*
    ⟨*proof*⟩

  lemma *range-build*: *range build* = *dom match*
    ⟨*proof*⟩

  lemma *inj-build*: *inj build*
    ⟨*proof*⟩

end

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

11.2 Co-dependence

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

definition *prism-diff* :: $('a \Longrightarrow_\triangle 's) \Rightarrow ('b \Longrightarrow_\triangle 's) \Rightarrow bool$ (**infix** ‹∇› *50*) **where**
[*lens-defs*]: *prism-diff X Y* = (*range* $build_X$ ∩ *range* $build_Y$ = {})

lemma *prism-diff-intro*:
  $(\bigwedge s_1\ s_2.\ build_X\ s_1 = build_Y\ s_2 \Longrightarrow False) \Longrightarrow X \nabla Y$
  ⟨*proof*⟩

lemma *prism-diff-irrefl*: $\neg X \nabla X$
  ⟨*proof*⟩

lemma *prism-diff-sym*: $X \nabla Y \Longrightarrow Y \nabla X$
  ⟨*proof*⟩

lemma *prism-diff-build*: $X \nabla Y \Longrightarrow build_X\ u \neq build_Y\ v$
  ⟨*proof*⟩

lemma *prism-diff-build-match*: $[\![\textit{wb-prism}\ X;\ X \nabla Y]\!] \Longrightarrow match_X\ (build_Y\ v) = None$
  ⟨*proof*⟩

11.3 Canonical prisms

definition *prism-id* :: $('a \Longrightarrow_\triangle 'a)$ (‹$1_\triangle$›) **where**
[*lens-defs*]: *prism-id* = ⦇ *prism-match* = *Some*, *prism-build* = *id* ⦈

lemma *wb-prism-id*: *wb-prism* $1_\triangle$
  ⟨*proof*⟩

lemma *prism-id-never-diff*: $\neg 1_\triangle \nabla X$
  ⟨*proof*⟩

11.4 Summation

definition *prism-plus* :: $('a \Longrightarrow_\triangle 's) \Rightarrow ('b \Longrightarrow_\triangle 's) \Rightarrow 'a + 'b \Longrightarrow_\triangle 's$ (**infixl** ‹$+_\triangle$› *85*)
where
[*lens-defs*]: $X +_\triangle Y = (\!|$ *prism-match* $= (\lambda\ s.\ case\ (match_X\ s,\ match_Y\ s)\ of$
$(Some\ u,\ \text{-}) \Rightarrow Some\ (Inl\ u)\ |$
$(None,\ Some\ v) \Rightarrow Some\ (Inr\ v)\ |$
$(None,\ None) \Rightarrow None),$
prism-build $= (\lambda\ v.\ case\ v\ of\ Inl\ x \Rightarrow build_X\ x\ |\ Inr\ y \Rightarrow build_Y\ y)\ |\!)$

lemma *prism-plus-wb* [*simp*]: $[\![\ wb\text{-}prism\ X;\ wb\text{-}prism\ Y;\ X\ \nabla\ Y\ ]\!] \Longrightarrow wb\text{-}prism\ (X +_\triangle Y)$
⟨*proof*⟩

lemma *build-plus-Inl* [*simp*]: $build_{c\ +_\triangle\ d}\ (Inl\ x) = build_c\ x$
⟨*proof*⟩

lemma *build-plus-Inr* [*simp*]: $build_{c\ +_\triangle\ d}\ (Inr\ y) = build_d\ y$
⟨*proof*⟩

lemma *prism-diff-preserved-1* [*simp*]: $[\![\ X\ \nabla\ Y;\ X\ \nabla\ Z\ ]\!] \Longrightarrow X\ \nabla\ Y +_\triangle Z$
⟨*proof*⟩

lemma *prism-diff-preserved-2* [*simp*]: $[\![\ X\ \nabla\ Z;\ Y\ \nabla\ Z\ ]\!] \Longrightarrow X +_\triangle Y\ \nabla\ Z$
⟨*proof*⟩

The following two lemmas are useful for reasoning about prism sums

lemma *Bex-Sum-iff*: $(\exists x{\in}A{<}{+}{>}B.\ P\ x) \longleftrightarrow (\exists\ x{\in}A.\ P\ (Inl\ x)) \lor (\exists\ y{\in}B.\ P\ (Inr\ y))$
⟨*proof*⟩

lemma *Ball-Sum-iff*: $(\forall x{\in}A{<}{+}{>}B.\ P\ x) \longleftrightarrow (\forall\ x{\in}A.\ P\ (Inl\ x)) \land (\forall\ y{\in}B.\ P\ (Inr\ y))$
⟨*proof*⟩

11.5 Instances

definition *prism-suml* :: $('a,\ 'a + 'b)\ prism$ (‹$Inl_\triangle$›) **where**
[*lens-defs*]: $prism\text{-}suml = (\!|\ prism\text{-}match = (\lambda\ v.\ case\ v\ of\ Inl\ x \Rightarrow Some\ x\ |\ \text{-} \Rightarrow None),\ prism\text{-}build = Inl\ |\!)$

definition *prism-sumr* :: $('b,\ 'a + 'b)\ prism$ (‹$Inr_\triangle$›) **where**
[*lens-defs*]: $prism\text{-}sumr = (\!|\ prism\text{-}match = (\lambda\ v.\ case\ v\ of\ Inr\ x \Rightarrow Some\ x\ |\ \text{-} \Rightarrow None),\ prism\text{-}build = Inr\ |\!)$

lemma *wb-prim-suml* [*simp*]: $wb\text{-}prism\ Inl_\triangle$
⟨*proof*⟩

lemma *wb-prim-sumr* [*simp*]: $wb\text{-}prism\ Inr_\triangle$
⟨*proof*⟩

lemma *prism-suml-indep-sumr* [*simp*]: $Inl_\triangle\ \nabla\ Inr_\triangle$
⟨*proof*⟩

lemma *prism-sum-plus*: $Inl_\triangle +_\triangle Inr_\triangle = 1_\triangle$
⟨*proof*⟩

11.6 Lens correspondence

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

definition *prism-lens* :: ('*a*, '*s*) *prism* ⇒ ('*a* ⟹ '*s*) **where**
prism-lens X = ⦇ *lens-get* = (λ *s*. *the* ($match_X$ *s*)), *lens-put* = (λ *s v*. $build_X$ *v*) ⦈

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

lemma *mwb-prism-lens*: *wb-prism a* ⟹ *mwb-lens* (*prism-lens a*)
⟨*proof*⟩

lemma *get-prism-lens*: $get_{prism\text{-}lens\ X}$ = *the* ∘ $match_X$
⟨*proof*⟩

lemma *src-prism-lens*: $\mathcal{S}_{prism\text{-}lens\ X}$ = *range* ($build_X$)
⟨*proof*⟩

lemma *create-prism-lens*: $create_{prism\text{-}lens\ X}$ = $build_X$
⟨*proof*⟩

lemma *prism-lens-inverse*:
wb-prism X ⟹ *lens-prism* (*prism-lens X*) = *X*
⟨*proof*⟩

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

lemma *lens-prism-put-inverse*:
pbij-lens X ⟹ $put_{prism\text{-}lens\ (lens\text{-}prism\ X)}$ = put_X
⟨*proof*⟩

lemma *wb-prism-implies-pbij-lens*:
wb-prism X ⟹ *pbij-lens* (*prism-lens X*)
⟨*proof*⟩

lemma *pbij-lens-implies-wb-prism*:
assumes *pbij-lens X*
shows *wb-prism* (*lens-prism X*)
⟨*proof*⟩

⟨*ML*⟩

end

12 Channel Types

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

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

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

lemma *wb-ctor-prism-intro*:
assumes
⋀ *v*. *disc* (*ctor v*)
⋀ *v*. *sel* (*ctor v*) = *v*
⋀ *s*. *disc s* ⟹ *ctor* (*sel s*) = *s*
shows *wb-prism* (*ctor-prism ctor disc sel*)
⟨*proof*⟩

lemma *ctor-codep-intro*:
assumes ⋀ *x y*. *ctor1 x* ≠ *ctor2 y*
shows *ctor-prism ctor1 disc1 sel1* ∇ *ctor-prism ctor2 disc2 sel2*
⟨*proof*⟩

⟨*ML*⟩

end

13 Data spaces

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

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

⟨*ML*⟩

end

14 Optics Meta-Theory

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

15 State and Lens integration

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

Inspired by Haskell's lens package

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

definition *use* :: ('*a* ⟹ '*b*) ⇒ ('*b*, '*a*) *state* **where**
use l = *zoom l State-Monad.get*

definition *modify* :: ('*a* ⟹ '*b*) ⇒ ('*a* ⇒ '*a*) ⇒ ('*b*, *unit*) *state* **where**
modify l f = *zoom l* (*State-Monad.update f*)

definition *assign* :: ('*a* ⟹ '*b*) ⇒ '*a* ⇒ ('*b*, *unit*) *state* **where**
assign l b = *zoom l* (*State-Monad.set b*)

context begin

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

end

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

context includes *lens-state-syntax* **begin**

lemma *zoom-comp1*: *l1* ▷ *l2* ▷ *s* = (*l2* $;_L$ *l1*) ▷ *s*
⟨*proof*⟩

lemma *zoom-zero*[*simp*]: *zero-lens* ▷ *s* = *s*
⟨*proof*⟩

lemma *zoom-id*[*simp*]: *id-lens* ▷ *s* = *s*
⟨*proof*⟩

end

lemma (**in** *mwb-lens*) *zoom-comp2*[*simp*]: *zoom x m* ≫= (λ*a*. *zoom x* (*n a*)) = *zoom x* (*m* ≫= *n*)
⟨*proof*⟩

lemma (**in** *wb-lens*) *use-alt-def*: *use x* = *map-state* (*lens-get x*) *State-Monad.get*
⟨*proof*⟩

lemma (**in** *wb-lens*) *modify-alt-def*: *modify x f* = *State-Monad.update* (*update f*)
⟨*proof*⟩

lemma (**in** *wb-lens*) *modify-id*[*simp*]: *modify* x ($\lambda x.\ x$) = *State-Monad.return* ()
⟨*proof*⟩

lemma (**in** *mwb-lens*) *modify-comp*[*simp*]: *bind* (*modify* x f) (λ-. *modify* x g) = *modify* x ($g \circ f$)
⟨*proof*⟩

end

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

References

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

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

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

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

[5] S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197, October 2020.

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

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

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

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