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

May 14, 2024

Abstract

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

Contents

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

2 Types of Cardinality 2 or Greater **4**

3 Core Lens Laws **5**
3.1 Lens Signature . . . 5
3.2 Weak Lenses . . . 6
3.3 Well-behaved Lenses . . . 7
3.4 Mainly Well-behaved Lenses . . . 7
3.5 Very Well-behaved Lenses . . . 8
3.6 Ineffectual Lenses . . . 9
3.7 Partially Bijective Lenses . . . 10
3.8 Bijective Lenses . . . 11
3.9 Lens Independence . . . 12
3.10 Lens Compatibility . . . 13

4 Lens Algebraic Operators **13**
4.1 Lens Composition, Plus, Unit, and Identity . . . 14
4.2 Closure Poperties . . . 15
4.3 Composition Laws . . . 17
4.4 Independence Laws . . . 17

4.5 Compatibility Laws . . . 19
4.6 Algebraic Laws . . . 19

5 Order and Equivalence on Lenses **21**
5.1 Sub-lens Relation . . . 21
5.2 Lens Equivalence . . . 22
5.3 Further Algebraic Laws . . . 23
5.4 Bijective Lens Equivalences . . . 28
5.5 Lens Override Laws . . . 30
5.6 Alternative Sublens Characterisation . . . 31
5.7 Alternative Equivalence Characterisation . . . 32
5.8 Ineffectual Lenses as Zero Elements . . . 32

6 Symmetric Lenses **33**
6.1 Partial Symmetric Lenses . . . 33
6.2 Symmetric Lenses . . . 34

7 Scenes **34**
7.1 Overriding Functions . . . 35
7.2 Scene Type . . . 35
7.3 Linking Scenes and Lenses . . . 42
7.4 Function Domain Scene . . . 45

8 Scene Spaces **46**
8.1 Preliminaries . . . 46
8.2 Predicates . . . 50
8.3 Scene space class . . . 50
8.4 Mapping a lens over a scene list . . . 59
8.5 Instances . . . 60
8.6 Scene space and basis lenses . . . 60
8.7 Composite lenses . . . 61

9 Lens Instances **62**
9.1 Function Lens . . . 62
9.2 Function Range Lens . . . 63
9.3 Map Lens . . . 63
9.4 List Lens . . . 63
9.5 Stream Lenses . . . 66
9.6 Record Field Lenses . . . 66
9.7 Locale State Spaces . . . 67
9.8 Type Definition Lens . . . 68
9.9 Mapper Lenses . . . 68
9.10 Lens Interpretation . . . 69
9.11 Tactic . . . 69

10 Lenses **69**

11 Prisms **69**
11.1 Signature and Axioms 69
11.2 Co-dependence 70
11.3 Canonical prisms 70
11.4 Summation 71
11.5 Instances 71
11.6 Lens correspondence 72

12 Channel Types **73**

13 Data spaces **74**

14 Optics Meta-Theory **74**

15 State and Lens integration **74**

1 Interpretation Tools

theory *Interp*
imports *Main*
begin

1.1 Interpretation Locale

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

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

lemma *exists-interp-law*:
$(\exists P.\ Q\ P) = (\exists P.\ Q\ (P\ o\ f))$
  apply (*safe*)
    — Subgoal 1

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

2 Types of Cardinality 2 or Greater

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

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

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

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

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

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

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

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

instance list :: (type) two
```

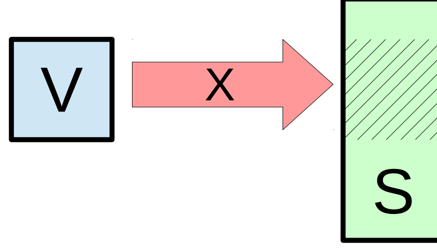

Figure 1: Visualisation of a simple lens

by (*intro-classes*, *auto simp add*: *infinite-UNIV-listI*)

end

3 Core Lens Laws

theory *Lens-Laws*
imports
Two Interp
begin

3.1 Lens Signature

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

record $('a, 'b)$ *lens* =
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**
$lens\text{-}set \equiv (\lambda\ X\ v\ s.\ put_X\ s\ v)$

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.

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**
$lens\text{-}source\ X = \{s.\ \exists\ v\ s'.\ s = put_X\ s'\ v\}$

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* σ *v*) = *v*
begin
  lemma *source-nonempty*: $\exists\ s.\ s \in \mathcal{S}$
    by (*auto simp add*: *lens-source-def*)

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

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

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

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

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

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

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

lemma *put-inj*: *inj* (*put σ*)
by (*simp add*: *injI view-determination*)

end

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

lemma *dom-pget*: *dom* $pget_x = \mathcal{S}_x$
by (*simp add*: *lens-partial-get-def dom-def*)

3.3 Well-behaved Lenses

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

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

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

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

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

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

end

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

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

3.4 Mainly Well-behaved Lenses

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

locale *mwb-lens* = *weak-lens* +

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

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

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

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

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

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

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

lemma *weak-get-via-put*: $s \in \mathcal{S} \Longrightarrow get\ s = (THE\ v.\ put\ s\ v = s)$
by (*rule sym*, *auto intro*!: *the-equality weak-get-put*, *metis put-get*)

end

abbreviation (*input*) *partial-lens* $\equiv$ *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
by (*simp add*: *mwb-lens.axioms*(*1*))

3.5 Very Well-behaved Lenses

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

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

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

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

get can be uniquely determined from *put*

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

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

Observation equivalence is an equivalence relation.

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

end

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

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

lemma *vwb-lens-mwb* [*simp*]: *vwb-lens x* ⟹ *mwb-lens x*
  using *vwb-lens-def* **by** *auto*

lemma *mwb-UNIV-src-is-vwb-lens*:
  ⟦ *mwb-lens X*; $\mathcal{S}_X$ = *UNIV* ⟧ ⟹ *vwb-lens X*
  using *vwb-lens-def wb-lens-axioms-def wb-lens-def* **by** *fastforce*

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

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

3.6 Ineffectual Lenses

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

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

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

```
sublocale vwb-lens by (fact ief-then-vwb)

lemma ineffectual-const-get:
  ∃ v.  ∀ σ∈𝒮. get σ = v
  using put-get put-inef by auto

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 ⟹ 's)
  shows False
proof −
  obtain x y :: 'a::two where x ≠ y
    using two-diff by auto
  with assms show ?thesis
    by (metis (full-types) ief-lens.axioms(1) ief-lens.put-inef weak-lens.put-get)
qed

abbreviation eff-lens X ≡ (weak-lens X ∧ (¬ ief-lens X))
```

3.7 Partially Bijective Lenses

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

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

  lemma put-is-create: put σ v = create v
    by (simp add: lens-create-def put-det)

  lemma partial-get-put: ϱ ∈ 𝒮 ⟹ put σ (get ϱ) = ϱ
    by (metis put-det weak-get-put)

end

lemma pbij-lens-weak [simp]:
  pbij-lens x ⟹ weak-lens x
  by (simp-all add: pbij-lens-def)

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

lemma pbij-alt-intro:
  ⟦ weak-lens X; ⋀ s. s ∈ 𝒮_X ⟹ create_X (get_X s) = s ⟧ ⟹ pbij-lens X
  by (metis pbij-lens-axioms-def pbij-lens-def weak-lens.put-closure weak-lens.put-get)
```

3.8 Bijective Lenses

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

locale *bij-lens* = *weak-lens* +
  assumes *strong-get-put*: *put* σ (*get* ϱ) = ϱ
begin

sublocale *pbij-lens*
proof
  fix σ v ϱ
  show *put* σ v = *put* ϱ v
    by (*metis put-get strong-get-put*)
qed

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

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

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

end

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

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

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

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

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

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

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

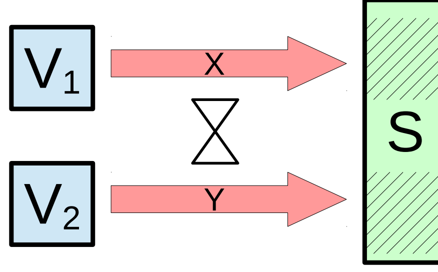

Figure 2: Lens Independence

using *pbij-vwb-is-bij-lens* **by** *auto*

3.9 Lens Independence

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

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

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

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

Lens independence is symmetric.

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

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

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

Characterisation of independence for two very well-behaved lenses

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

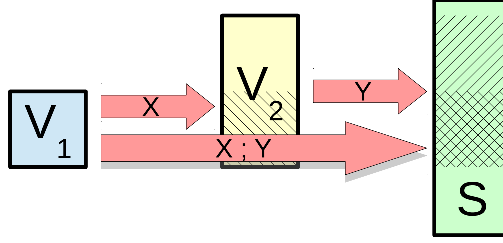

Figure 3: Lens Composition

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

3.10 Lens Compatibility

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

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

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

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

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

end

4 Lens Algebraic Operators

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

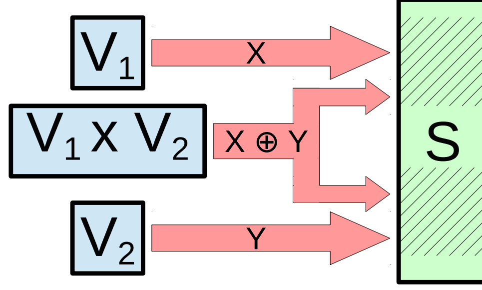

Figure 4: Lens Sum

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-comp* Y X = ⦇ *lens-get* = $get_Y \circ$ *lens-get* X
, *lens-put* = $(\lambda\ \sigma\ v.$ *lens-put* $X\ \sigma$ (*lens-put* Y (*lens-get* $X\ \sigma$) v)) ⦈

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2 Closure Poperties

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

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

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

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

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

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

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

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

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

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

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

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

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

by (*unfold-locales*, *simp-all add*: *zero-lens-def*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3 Composition Laws

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

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

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

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

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

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

4.4 Independence Laws

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

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

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

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

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

unfolding *lens-indep-def ief-lens-def ief-lens-axioms-def*
by (*simp*, *metis* (*full-types*) *wb-lens.get-put*)

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

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

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

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

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

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

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

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

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

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

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

Lens independence is preserved by summation.

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

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

Lens independence is preserved by product.

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

4.5 Compatibility Laws

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

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

4.6 Algebraic Laws

Lens plus distributes to the right through composition.

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

The first lens projects the first part of a summation.

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

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

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

apply (*simp-all*)
done

The swap lens switches over a summation.

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

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

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

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

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

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

The product lens can be rewritten as a sum lens.

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

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

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

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

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

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

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

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

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

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

lemma *lens-obs-eq-as-override*: $vwb\text{-}lens\ X \Longrightarrow s_1 \simeq_X s_2 \longleftrightarrow (s_2 = s_1 \oplus_L s_2\ on\ X)$

by (*auto simp add*: *lens-defs*; *metis vwb-lens.put-eq*)

end

5 Order and Equivalence on Lenses

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

5.1 Sub-lens Relation

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

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

Various lens classes are downward closed under the sublens relation.

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

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

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

Sublens is a preorder as the following two theorems show.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Lens Equivalence

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

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

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

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

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

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

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

lemma *lens-equiv-pres-indep'*:
$\llbracket X \approx_L Y; Z \bowtie Y \rrbracket \Longrightarrow Z \bowtie X$
using *lens-equiv-def sublens-pres-indep'* **by** *blast*

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

5.3 Further Algebraic Laws

This law explains the behaviour of lens quotient.

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

Plus distributes through quotient.

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

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

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

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

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

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

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

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

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

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

We can similarly show that it is commutative.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 −
have $x \bowtie x$
using *assms*(*2*) *assms*(*3*) *lens-equiv-pres-indep'* **by** *blast*
thus *?thesis*
using *assms*(*1*) *lens-indep-quasi-irrefl vwb-lens-wb wb-lens-weak* **by** *blast*
qed

lemma *indep-eff-implies-not-equiv* [*simp*]:
fixes $x :: 'a::two \Longrightarrow 'b$

assumes *wb-lens x x* $\bowtie$ *y*
shows $\neg$ ($x \approx_L y$)
using *assms indep-and-equiv-implies-ief no-ief-two-view* **by** *blast*

5.4 Bijective Lens Equivalences

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

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

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

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

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

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

lemma *bij-lens-equiv*:
$[\![$ *bij-lens X*; $X \approx_L Y$ $]\!]$ $\Longrightarrow$ *bij-lens Y*
by (*meson bij-lens-equiv-id lens-equiv-def sublens-trans*)

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

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

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

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

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

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

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

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

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

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

lemma *pbij-plus-commute*:
  ⟦ $a \bowtie b$; *mwb-lens a*; *mwb-lens b*; *pbij-lens* $(b +_L a)$ ⟧ ⟹ *pbij-lens* $(a +_L b)$

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

5.5 Lens Override Laws

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

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

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

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

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

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

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

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

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

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

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* $A)\ v$
by (*simp add*: *lens-override-def lens-get-put-quasi-commute lens-put-of-quotient*)

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* $A)$
by (*simp add*: *lens-override-def lens-indep.lens-put-irr2*)

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

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

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

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

5.6 Alternative Sublens Characterisation

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

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

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

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

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

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

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

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

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

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

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

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* $(x /_L y)$
by (*unfold-locales*)
(*simp-all add*: *sublens′-def lens-quotient-def lens-quotient-mwb sublens-iff-sublens′ lens-create-def sublens′-prop1 sublens′-prop2*)

lemma *lens-quotient-indep*:
$[\![$ *vwb-lens* x; *vwb-lens* y; *vwb-lens* a; $x \bowtie y$; $x \subseteq_L a$; $y \subseteq_L a$ $]\!] \Longrightarrow (x /_L a) \bowtie (y /_L a)$
by (*unfold-locales*)
(*simp-all add*: *lens-quotient-def sublens-iff-sublens′ lens-create-def lens-indep.lens-put-comm sublens′-prop1 sublens′-prop2 lens-indep.lens-put-irr2*)

lemma *lens-quotient-bij*: $[\![$ *vwb-lens* x; *vwb-lens* y; $y \approx_L x$ $]\!] \Longrightarrow$ *bij-lens* $(x /_L y)$
by (*metis lens-comp-quotient lens-equiv-iff-bij lens-equiv-sym vwb-lens-wb wb-lens-weak*)

5.7 Alternative Equivalence Characterisation

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

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

5.8 Ineffectual Lenses as Zero Elements

lemma *ief-lens-then-zero*: *ief-lens* $x \Longrightarrow x \approx_L 0_L$
by (*simp add*: *lens-equiv-iff-lens-equiv′ lens-equiv′-def*)

(*simp add*: *ief-lens.put-inef lens-override-def*)

lemma *ief-lens-iff-zero*: *vwb-lens x* $\Longrightarrow$ *ief-lens x* $\longleftrightarrow$ $x \approx_L 0_L$
by (*metis ief-lens-axioms-def ief-lens-def ief-lens-then-zero lens-equiv-def lens-override-def lens-override-unit sublens'-prop3 sublens-implies-sublens' unit-vwb-lens vwb-lens-wb wb-lens-weak*)

end

6 Symmetric Lenses

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

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

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

type-notation
slens (<-, -> $\Longleftrightarrow$ - [*0*, *0*, *0*] *0*)

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

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

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

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

6.1 Partial Symmetric Lenses

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

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

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

6.2 Symmetric Lenses

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

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

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

end

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

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

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

end

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

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

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

7.2 Scene Type

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

setup-lifting *type-definition-scene*

lift-definition *idem-scene* :: $'s\ scene \Rightarrow bool$ **is** *idem-overrider* **.**

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

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

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

done

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

lemma *region-coregion-Id*:
$idem\text{-}scene\ X \Longrightarrow region\ X \cap coregion\ X = Id$
by (*transfer*, *auto*, *metis idem-overrider.ovr-idem*)

lemma *state-eq-iff*: $idem\text{-}scene\ S \Longrightarrow x = y \longleftrightarrow (x, y) \in region\ S \wedge (x, y) \in coregion\ S$
by (*metis IntE IntI pair-in-Id-conv region-coregion-Id*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *bot-idem-scene* [*simp*]: *idem-scene* $\bot_S$
  by (*transfer*, *unfold-locales*, *simp-all*)

lemma *top-idem-scene* [*simp*]: *idem-scene* $\top_S$
  by (*transfer*, *unfold-locales*, *simp-all*)

lemma *uminus-top-scene* [*simp*]: $- \top_S = \bot_S$
  by (*transfer*, *simp*)

lemma *uminus-bot-scene* [*simp*]: $- \bot_S = \top_S$
  by (*transfer*, *simp*)

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

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

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

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

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

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

lemma *scene-override-inter*: $-X \#\#_S -Y \Longrightarrow S_1 \oplus_S S_2$ *on* $(X \sqcap_S Y) = S_1 \oplus_S S_1 \oplus_S S_2$ *on* X *on* Y
  by (*simp add*: *inf-scene-def scene-override-commute scene-override-union*)

lemma *scene-equiv-bot* [*simp*]: $a \approx_S b$ *on* $\bot_S$
  by (*simp add*: *scene-equiv-def*)

lemma *scene-equiv-refl* [*simp*]: *idem-scene* $a \Longrightarrow s \approx_S s$ *on* a
  by (*simp add*: *scene-equiv-def*)

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
  by (*metis scene-equiv-def scene-override-idem scene-override-overshadow-right*)

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

lemma *scene-indep-bot* [*simp*]: $X \bowtie_S \bot_S$
by (*transfer*, *simp*)

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

lemma *unit-scene-top-eq-bot*: $(\bot_S :: \mathit{unit\ scene}) = \top_S$
by (*transfer*, *simp*)

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

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

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

lemma *scene-indep-pres-compat*: $[\![A \bowtie_S B; A \bowtie_S C]\!] \Longrightarrow A \bowtie_S (B \sqcup_S C)$
by (*transfer*, *auto*)

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

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

lemma *scene-compat-bot* [*simp*]: $a \#\#_S \bot_S$ $\bot_S \#\#_S a$
by (*transfer*, *simp*)+

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

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

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

lemma *scene-union-indep-uniq*:
assumes *idem-scene* X *idem-scene* Y *idem-scene* Z $X \bowtie_S Z$ $Y \bowtie_S Z$ $X \sqcup_S Z = Y \sqcup_S Z$

shows $X = Y$
using *assms* **apply** (*transfer*, *simp*)
by (*metis* (*no-types*, *opaque-lifting*) *ext idem-overrider.ovr-idem overrider-def*)

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

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

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

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

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

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

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

lemma *scene-inter-commute*: $X \sqcap_S Y = Y \sqcap_S X$
by (*simp add*: *inf-scene-def scene-union-commute*)

lemma *scene-union-inter-distrib*:
$[\![$ *idem-scene* x; $x \bowtie_S y$; $x \bowtie_S z$; $y \#\#_S z$ $]\!] \Longrightarrow x \sqcup_S y \sqcap_S z = (x \sqcup_S y) \sqcap_S (x \sqcup_S z)$
apply (*simp add*: *inf-scene-def*, *transfer*)
apply (*auto simp add*: *fun-eq-iff*)
apply (*unfold overrider-def idem-overrider-def idem-overrider-axioms-def*)
apply *metis+*
done

lemma *idem-scene-uminus* [*simp*]: *idem-scene* $X \Longrightarrow$ *idem-scene* $(- X)$
by (*simp add*: *uminus-scene-def idem-scene-def Abs-scene-inverse idem-overrider-axioms-def idem-overrider-def overrider.intro*)

lemma *scene-minus-cancel*: $[\![$ $a \bowtie_S b$; *idem-scene* a; *idem-scene* b $]\!] \Longrightarrow a \sqcup_S (b \sqcap_S - a) = a \sqcup_S b$
apply (*simp add*: *lens-defs*, *transfer*, *auto simp add*: *fun-eq-iff*)
apply (*metis* (*mono-tags*, *lifting*) *overrider.ovr-overshadow-left*)
apply (*metis* (*no-types*, *opaque-lifting*) *idem-overrider.ovr-idem overrider.ovr-overshadow-right*)
done

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* $x\ y = (x \leq y \wedge \neg\ y \leq x)$
instance ..
end

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

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

lemma *subscene-trans*: $[\![$ *idem-scene* Y; $X \subseteq_S Y$; $Y \subseteq_S Z\ ]\!] \Longrightarrow X \subseteq_S Z$
by (*simp add*: *less-eq-scene-def*, *transfer*, *auto*, *metis* (*no-types*, *opaque-lifting*) *idem-overrider.ovr-idem*)

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

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

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

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

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

lemma *scene-union-ub*: $[\![$ *idem-scene* Y; $X \bowtie_S Y\ ]\!] \Longrightarrow X \leq (X \sqcup_S Y)$
by (*simp add*: *less-eq-scene-def*, *transfer*, *auto*)
(*metis* (*no-types*, *opaque-lifting*) *idem-overrider.ovr-idem overrider.ovr-overshadow-right*)

lemma *scene-union-lb*: $[\![\ a\ \#\#_S\ b;\ a \leq c;\ b \leq c\ ]\!] \Longrightarrow a \sqcup_S b \leq c$
by (*simp add*: *less-eq-scene-def scene-override-union*)

lemma *scene-union-mono*: $[\![\ a \subseteq_S c;\ b \subseteq_S c;\ a\ \#\#_S\ b;$ *idem-scene* a; *idem-scene* $b\ ]\!] \Longrightarrow a \sqcup_S b \subseteq_S c$
by (*simp add*: *less-eq-scene-def*, *transfer*, *auto*)

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

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

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

by (*auto simp add*: *less-eq-scene-def scene-indep-override scene-override-commute*)

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

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

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

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

lemma *scene-comp-bot* [*simp*]: $\bot_S ;_S x = \bot_S$
by (*transfer*, *auto*)

lemma *scene-comp-id-lens* [*simp*]: $A ;_S 1_L = A$
by (*transfer*, *auto*, *simp add*: *id-lens-def*)

lemma *scene-union-comp-distl*: $a \#\#_S b \Longrightarrow (a \sqcup_S b) ;_S x = (a ;_S x) \sqcup_S (b ;_S x)$
by (*transfer*, *auto simp add*: *fun-eq-iff*)

lemma *scene-comp-assoc*: ⟦ *vwb-lens X*; *vwb-lens Y* ⟧ ⟹ $A ;_S X ;_S Y = A ;_S (X ;_L Y)$
by (*transfer*, *auto simp add*: *lens-comp-def fun-eq-iff*)
(*metis comp-vwb-lens lens-comp-def*)

lift-definition *scene-quotient* :: *'b scene* ⇒ (*'a* ⟹ *'b*) ⇒ *'a scene* (**infixl** $'/_S$ *80*)
is λ *S X a b*. *if* (*vwb-lens X* ∧ ($\forall s_1\ s_2\ s_3$. *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*
by (*unfold-locales*, *auto simp add*: *lens-create-def lens-override-def*)
(*metis* (*no-types*, *lifting*) *overrider.ovr-overshadow-right*)

lemma *scene-quotient-idem*: *idem-scene S* ⟹ *idem-scene* ($S /_S X$)
by (*transfer*, *unfold-locales*, *auto simp add*: *lens-create-def lens-override-def*)
(*metis* (*no-types*, *lifting*) *overrider.ovr-overshadow-right*)

lemma *scene-quotient-indep*: $A \bowtie_S B \Longrightarrow (A /_S X) \bowtie_S (B /_S X)$
by (*transfer*, *auto simp add*: *lens-create-def lens-override-def*)

lemma *scene-bot-quotient* [*simp*]: $\bot_S /_S X = \bot_S$
by (*transfer*, *auto*)

lemma *scene-comp-quotient*: *vwb-lens X* ⟹ $(A ;_S X) /_S X = A$
by (*transfer*, *auto simp add*: *fun-eq-iff lens-override-def*)

lemma *scene-quot-id-lens* [*simp*]: $(A /_S 1_L) = A$
by (*transfer*, *simp*, *simp add*: *lens-defs*)

7.3 Linking Scenes and Lenses

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

lift-definition *lens-scene* :: (*'v* ⟹ *'s*) ⇒ *'s scene* ($[\![-]\!]_\sim$) **is**
λ *X* s_1 s_2. *if* (*mwb-lens X*) *then* $s_1 \oplus_L s_2$ *on X else* s_1

by (*unfold-locales*, *auto simp add*: *lens-override-def*)

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

lemma *idem-scene-impl-vwb*:
$[\![$ *mwb-lens X*; *idem-scene* $[\![X]\!]_\sim$ $]\!] \Longrightarrow$ *vwb-lens X*
apply (*cases mwb-lens X*)
apply (*transfer*, *unfold idem-overrider-def overrider-def*, *auto*)
apply (*simp add*: *idem-overrider-axioms-def override-idem-implies-vwb*)
done

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

Next we show some important congruence properties

lemma *zero-lens-scene*: $[\![0_L]\!]_\sim = \bot_S$
by (*transfer*, *simp*)

lemma *one-lens-scene*: $[\![1_L]\!]_\sim = \top_S$
by (*transfer*, *simp*)

lemma *scene-comp-top-scene* [*simp*]: *vwb-lens x* $\Longrightarrow \top_S\ ;_S\ x = [\![x]\!]_\sim$
by (*transfer*, *simp add*: *fun-eq-iff lens-override-def*)

lemma *scene-comp-lens-scene-indep* [*simp*]: $x \bowtie y \Longrightarrow [\![x]\!]_\sim \bowtie_S a\ ;_S\ y$
by (*transfer*, *simp add*: *lens-indep.lens-put-comm lens-indep.lens-put-irr2 lens-override-def*)

lemma *lens-scene-override*:
mwb-lens X $\Longrightarrow s_1 \oplus_S s_2$ *on* $[\![X]\!]_\sim = s_1 \oplus_L s_2$ *on X*
by (*transfer*, *simp*)

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

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

lemma *get-scene-override-indep*: $[\![$ *vwb-lens x*; $[\![x]\!]_\sim \bowtie_S a$ $]\!] \Longrightarrow get_x\ (s \oplus_S s'$ *on* $a) = get_x\ s$
proof −
assume *a1*: $[\![x]\!]_\sim \bowtie_S a$
assume *a2*: *vwb-lens x*
then have $\forall\, b\ ba\ bb.\ bb \oplus_S b \oplus_S ba$ *on a on* $[\![x]\!]_\sim = bb \oplus_S b$ *on* $[\![x]\!]_\sim$
using *a1* **by** (*metis idem-scene-uminus indep-then-compl-in scene-indep-sym scene-override-commute subscene-eliminate vwb-impl-idem-scene*)
then show *?thesis*
using *a2* **by** (*metis lens-override-def lens-scene-override mwb-lens-def vwb-lens-mwb weak-lens.put-get*)
qed

lemma *put-scene-override-indep*:

$\llbracket$ *vwb-lens x*; $\llbracket x\rrbracket_\sim \bowtie_S a$ $\rrbracket \Longrightarrow put_x\ s\ v \oplus_S s'$ *on* $a = put_x\ (s \oplus_S s'$ *on* $a)\ v$
by (*transfer*, *auto*)
(*metis lens-override-def mwb-lens-weak vwb-lens-mwb weak-lens.put-get*)

lemma *get-scene-override-le*: $\llbracket$ *vwb-lens x*; $\llbracket x\rrbracket_\sim \le a$ $\rrbracket \Longrightarrow get_x\ (s \oplus_S s'$ *on* $a) = get_x\ s'$
by (*metis get-scene-override-indep scene-le-iff-indep-inv scene-override-commute*)

lemma *put-scene-override-le*: $\llbracket$ *vwb-lens x*; *idem-scene a*; $\llbracket x\rrbracket_\sim \le a$ $\rrbracket \Longrightarrow put_x\ s\ v \oplus_S s'$ *on* $a = s \oplus_S s'$ *on a*
by (*metis lens-override-idem lens-override-put-right-in lens-scene-override sublens-refl subscene-eliminate vwb-lens-mwb*)

lemma *put-scene-override-le-distrib*:
$\llbracket$ *vwb-lens x*; *idem-scene A*; $\llbracket x\rrbracket_\sim \le A$ $\rrbracket \Longrightarrow put_x\ (s_1 \oplus_S s_2$ *on* $A)\ v = (put_x\ s_1\ v) \oplus_S (put_x\ s_2\ v)$ *on* A
by (*metis put-scene-override-indep put-scene-override-le scene-le-iff-indep-inv scene-override-commute*)

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

lemma *subscene-implies-sublens'*: $\llbracket$ *vwb-lens X*; *vwb-lens Y* $\rrbracket \Longrightarrow \llbracket X\rrbracket_\sim \le \llbracket Y\rrbracket_\sim \longleftrightarrow X \subseteq_L' Y$
by (*simp add*: *lens-defs*, *transfer*, *simp add*: *lens-override-def*)

lemma *sublens'-implies-subscene*: $\llbracket$ *vwb-lens X*; *vwb-lens Y*; $X \subseteq_L' Y$ $\rrbracket \Longrightarrow \llbracket X\rrbracket_\sim \le \llbracket Y\rrbracket_\sim$
by (*simp add*: *lens-defs*, *auto simp add*: *lens-override-def lens-scene-override*)

lemma *sublens-iff-subscene*:
assumes *vwb-lens X vwb-lens Y*
shows $X \subseteq_L Y \longleftrightarrow \llbracket X\rrbracket_\sim \le \llbracket Y\rrbracket_\sim$
by (*simp add*: *assms sublens-iff-sublens' subscene-implies-sublens'*)

lemma *lens-scene-indep-compl* [*simp*]:
assumes *vwb-lens x vwb-lens y*
shows $\llbracket x\rrbracket_\sim \bowtie_S - \llbracket y\rrbracket_\sim \longleftrightarrow x \subseteq_L y$
by (*simp add*: *assms scene-le-iff-indep-inv sublens-iff-subscene*)

lemma *lens-scene-comp*: $\llbracket$ *vwb-lens X*; *vwb-lens Y* $\rrbracket \Longrightarrow \llbracket X ;_L Y\rrbracket_\sim = \llbracket X\rrbracket_\sim ;_S Y$
by (*transfer*, *simp add*: *fun-eq-iff comp-vwb-lens*)
(*simp add*: *lens-comp-def lens-override-def*)

lemma *scene-comp-pres-indep*: $\llbracket$ *idem-scene a*; *idem-scene b*; $a \bowtie_S \llbracket x\rrbracket_\sim$ $\rrbracket \Longrightarrow a \bowtie_S b ;_S x$
by (*transfer*, *auto*)
(*metis* (*no-types*, *opaque-lifting*) *lens-override-def lens-override-idem vwb-lens-def wb-lens-weak weak-lens.put-get*)

lemma *scene-comp-le*: $A ;_S X \le \llbracket X\rrbracket_\sim$
unfolding *less-eq-scene-def* **by** (*transfer*, *auto simp add*: *fun-eq-iff lens-override-def*)

lemma *scene-quotient-comp*: $\llbracket$ *vwb-lens X*; *idem-scene A*; $A \le \llbracket X\rrbracket_\sim$ $\rrbracket \Longrightarrow (A /_S X) ;_S X = A$
unfolding *less-eq-scene-def*
proof (*transfer*, *simp add*: *fun-eq-iff*, *safe*)
fix $Xa :: 'a \Longrightarrow 'b$ **and** $Aa :: 'b \Rightarrow 'b \Rightarrow 'b$ **and** $x :: 'b$ **and** $xa :: 'b$
assume *a1*: *vwb-lens Xa*
assume *a2*: *overrider Aa*

assume *a3*: *idem-overrider Aa*
assume *a4*: $\forall s_1\ s_2\ s_3.\ Aa\ (s_1 \triangleleft_{Xa}\ s_2)\ s_3 = s_1 \triangleleft_{Xa}\ Aa\ s_2\ s_3$
have $\bigwedge b.\ Aa\ b\ b = b$
using *a3* **by** *simp*
then have $Aa\ x\ (put_{Xa}\ src_{Xa}\ (get_{Xa}\ xa)) = Aa\ x\ xa$
by (*metis a2 a4 lens-override-def overrider.ovr-overshadow-right*)
then show $put_{Xa}\ x\ (get_{Xa}\ (Aa\ (create_{Xa}\ (get_{Xa}\ x))\ (create_{Xa}\ (get_{Xa}\ xa)))) = Aa\ x\ xa$
using *a4 a1* **by** (*metis lens-create-def lens-override-def vwb-lens-def wb-lens.get-put wb-lens-weak weak-lens.put-get*)
qed

lemma *lens-scene-quotient*: $[\![$ *vwb-lens* Y; $X \subseteq_L Y$ $]\!] \Longrightarrow [\![X /_L\ Y]\!]_\sim = [\![X]\!]_\sim /_S\ Y$
by (*metis lens-quotient-comp lens-quotient-vwb lens-scene-comp scene-comp-quotient sublens-pres-vwb vwb-lens-def wb-lens-weak*)

lemma *scene-union-quotient*: $[\![\ A \mathbin{\#\#_S} B;\ A \le [\![X]\!]_\sim;\ B \le [\![X]\!]_\sim\ ]\!] \Longrightarrow (A \sqcup_S B) /_S\ X = (A /_S\ X) \sqcup_S (B /_S\ X)$
unfolding *less-eq-scene-def*
by (*case-tac vwb-lens X*; *transfer*, *auto simp add*: *lens-create-def lens-override-def*)

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

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

lemma *lens-scene-top-iff-bij-lens*: *mwb-lens x* $\Longrightarrow [\![x]\!]_\sim = \top_S \longleftrightarrow$ *bij-lens x*
apply (*transfer*)
apply (*auto simp add*: *fun-eq-iff lens-override-def*)
apply (*unfold-locales*)
apply *auto*
done

7.4 Function Domain Scene

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

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

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

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

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

by (*simp add*: *inf-scene-def fun-dom-scene-union*[*THEN sym*] *fun-dom-scene-compl*[*THEN sym*])

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

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

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

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

Hide implementation details for scenes

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

end

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* ($\bowtie_S$) $A \Longrightarrow$ *pairwise* ($\#\#_S$) A
by (*simp add*: *pairwise-alt*)

lemma *pairwise-compat-foldr*:
⟦ *pairwise* ($\#\#_S$) (*set as*); ∀ *b* ∈ *set as*. *a* $\#\#_S$ *b* ⟧ ⟹ *a* $\#\#_S$ $\bigsqcup_S$ *as*
apply (*induct as*)
apply (*simp*)
apply (*auto simp add*: *pairwise-insert scene-union-pres-compat*)
done

lemma *foldr-scene-indep*:
⟦ *pairwise* ($\#\#_S$) (*set as*); ∀ *b* ∈ *set as*. *a* $\bowtie_S$ *b* ⟧ ⟹ *a* $\bowtie_S$ $\bigsqcup_S$ *as*
apply (*induct as*)
apply (*simp*)
apply (*auto intro*: *scene-indep-pres-compat simp add*: *pairwise-insert*)
done

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*
apply (*induct as*)
apply (*simp*)
apply (*auto simp add*: *pairwise-insert*)

apply (*metis pairwise-compat-foldr scene-compat-refl scene-union-comp-distl*)
done

lemma *foldr-compat-quotient-dist*:
$[\![$ *pairwise* $(\#\#_S)$ (*set as*); $\forall$ $a \in$*set as*. $a \leq [\![x]\!]_\sim$ $]\!] \Longrightarrow$ *foldr* $(\sqcup_S)$ (*map* ($\lambda a.$ a $/_S$ x) *as*) $\bot_S = \bigsqcup_S$ *as* $/_S$ x
apply (*induct as*)
apply (*auto simp add*: *pairwise-insert*)
apply (*subst scene-union-quotient*)
apply *simp-all*
using *pairwise-compat-foldr scene-compat-refl* **apply** *blast*
apply (*meson foldr-scene-indep scene-indep-sym scene-le-iff-indep-inv*)
done

lemma *foldr-scene-union-add-tail*:
$[\![$ *pairwise* $(\#\#_S)$ (*set xs*); $\forall$ $x \in$*set xs*. x $\#\#_S$ b $]\!] \Longrightarrow \bigsqcup_S$ *xs* $\sqcup_S$ b = *foldr* $(\sqcup_S)$ *xs b*
apply (*induct xs*)
apply (*simp*)
apply (*simp*)
apply (*subst scene-union-assoc*[*THEN sym*])
apply (*auto simp add*: *pairwise-insert*)
using *pairwise-compat-foldr scene-compat-refl* **apply** *blast*
apply (*meson pairwise-compat-foldr scene-compat-sym*)
done

lemma *pairwise-Diff*: *pairwise R A* $\Longrightarrow$ *pairwise R* $(A - B)$
using *pairwise-mono* **by** *fastforce*

lemma *scene-compats-members*: $[\![$ *pairwise* $(\#\#_S)$ A; $x \in A$; $y \in A$ $]\!] \Longrightarrow x$ $\#\#_S$ y
by (*metis pairwise-def scene-compat-refl*)

corollary *foldr-scene-union-removeAll*:
assumes *pairwise* $(\#\#_S)$ (*set xs*) $x \in$ *set xs*
shows $\bigsqcup_S$ (*removeAll x xs*) $\sqcup_S$ $x = \bigsqcup_S$ *xs*
using *assms* **proof** (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
have *x-compat*: $\bigwedge$ $z.$ $z \in$ *set xs* $\Longrightarrow x$ $\#\#_S$ z
using *Cons.prems*(*1*) *Cons.prems*(*2*) *scene-compats-members* **by** *auto*

from *Cons* **have** *x-compats*: x $\#\#_S$ $\bigsqcup_S$ (*removeAll x xs*)
by (*metis* (*no-types*, *lifting*) *insert-Diff list.simps*(*15*) *pairwise-compat-foldr pairwise-insert removeAll-id set-removeAll x-compat*)

from *Cons* **have** *a-compats*: a $\#\#_S$ $\bigsqcup_S$ (*removeAll x xs*)
by (*metis* (*no-types*, *lifting*) *insert-Diff insert-iff list.simps*(*15*) *pairwise-compat-foldr pairwise-insert scene-compat-refl set-removeAll x-compats*)

from *Cons* **show** *?case*
proof (*cases* $x \in$ *set xs*)
case *True*
with *Cons* **show** *?thesis*
by (*auto simp add*: *pairwise-insert scene-union-commute*)

```
      (metis a-compats scene-compats-members scene-union-assoc scene-union-idem,
       metis (full-types) a-compats scene-union-assoc scene-union-commute x-compats)
  next
    case False
    with Cons show ?thesis
      by (simp add: scene-union-commute)
  qed
qed

lemma foldr-scene-union-eq-sets:
  assumes pairwise (##S) (set xs) set xs = set ys
  shows ⨆S xs = ⨆S ys
using assms proof (induct xs arbitrary: ys)
  case Nil
  then show ?case
    by simp
next
  case (Cons a xs)
  hence ys: set ys = insert a (set (removeAll a ys))
    by (auto)
  then show ?case
   by (metis (no-types, lifting) Cons.hyps Cons.prems(1) Cons.prems(2) Diff-insert-absorb foldr-scene-union-removeAll
insertCI insert-absorb list.simps(15) pairwise-insert set-removeAll)
qed

lemma foldr-scene-removeAll:
  assumes pairwise (##S) (set xs)
  shows x ⊔S ⨆S (removeAll x xs) = x ⊔S ⨆S xs
   by (metis (mono-tags, opaque-lifting) assms foldr-Cons foldr-scene-union-eq-sets insertCI insert-Diff
list.simps(15) o-apply removeAll.simps(2) removeAll-id set-removeAll)

lemma pairwise-Collect: pairwise R A ⟹ pairwise R {x ∈ A. P x}
  by (simp add: pairwise-def)

lemma removeAll-overshadow-filter:
  removeAll x (filter (λxa. xa ∉ A − {x}) xs) = removeAll x (filter (λ xa. xa ∉ A) xs)
  apply (simp add: removeAll-filter-not-eq)
  apply (rule filter-cong)
   apply (simp)
  apply auto
  done

corollary foldr-scene-union-filter:
  assumes pairwise (##S) (set xs) set ys ⊆ set xs
  shows ⨆S xs = ⨆S (filter (λx. x ∉ set ys) xs) ⊔S ⨆S ys
using assms proof (induct xs arbitrary: ys)
  case Nil
  then show ?case by (simp)
next
  case (Cons x xs)
  show ?case
  proof (cases x ∈ set ys)
    case True
    with Cons have 1: set ys − {x} ⊆ set xs
      by (auto)
```

have *2*: $x \#\#_S \bigsqcup_S$ (*removeAll x ys*)
by (*metis Cons.prems*(*1*) *Cons.prems*(*2*) *True foldr-scene-removeAll foldr-scene-union-removeAll pairwise-subset scene-compat-bot*(*2*) *scene-compat-sym scene-union-incompat scene-union-unit*(*1*))
have *3*: $\bigwedge$ *P*. $x \#\#_S \bigsqcup_S$ (*filter P xs*)
by (*meson Cons.prems*(*1*) *Cons.prems*(*2*) *True filter-is-subset in-mono pairwise-compat-foldr pairwise-subset scene-compats-members set-subset-Cons*)
have *4*: $\bigwedge$ *P*. $\bigsqcup_S$ (*filter P xs*) $\#\#_S \bigsqcup_S$ (*removeAll x ys*)
by (*rule pairwise-compat-foldr*)
(*metis Cons.prems*(*1*) *Cons.prems*(*2*) *pairwise-Diff pairwise-subset set-removeAll*,
metis (*no-types*, *lifting*) *1 Cons.prems*(*1*) *filter-is-subset pairwise-compat-foldr pairwise-subset scene-compat-sym scene-compats-members set-removeAll set-subset-Cons subsetD*)
have $\bigsqcup_S$ (*x # xs*) = $x \sqcup_S \bigsqcup_S$ *xs*
by *simp*
also have ... = $x \sqcup_S$ ($\bigsqcup_S$ (*filter* (λ*xa*. *xa* $\notin$ *set ys* − {*x*}) *xs*) $\sqcup_S \bigsqcup_S$ (*removeAll x ys*))
using *1 Cons*(*1*)[**where** *ys*=*removeAll x ys*] *Cons*(*2*) **by** (*simp add*: *pairwise-insert*)
also have ... = ($x \sqcup_S \bigsqcup_S$ (*filter* (λ*xa*. *xa* $\notin$ *set ys* − {*x*}) *xs*)) $\sqcup_S \bigsqcup_S$ (*removeAll x ys*)
by (*simp add*: *scene-union-assoc 1 2 3 4*)
also have ... = ($x \sqcup_S \bigsqcup_S$ (*removeAll x* (*filter* (λ*xa*. *xa* $\notin$ *set ys* − {*x*}) *xs*))) $\sqcup_S \bigsqcup_S$ (*removeAll x ys*)
by (*metis* (*no-types*, *lifting*) *Cons.prems*(*1*) *filter-is-subset foldr-scene-removeAll pairwise-subset set-subset-Cons*)
also have ... = ($x \sqcup_S \bigsqcup_S$ (*removeAll x* (*filter* (λ*xa*. *xa* $\notin$ *set ys*) *xs*))) $\sqcup_S \bigsqcup_S$ (*removeAll x ys*)
by (*simp only*: *removeAll-overshadow-filter*)
also have ... = ($x \sqcup_S \bigsqcup_S$ (*removeAll x* (*filter* (λ*xa*. *xa* $\notin$ *set ys*) (*x # xs*)))) $\sqcup_S \bigsqcup_S$ (*removeAll x ys*)
by *simp*
also have ... = ($x \sqcup_S \bigsqcup_S$ (*filter* (λ*xa*. *xa* $\notin$ *set ys*) (*x # xs*))) $\sqcup_S \bigsqcup_S$ (*removeAll x ys*)
by (*simp add*: *True*)
also have ... = ($\bigsqcup_S$ (*filter* (λ*xa*. *xa* $\notin$ *set ys*) (*x # xs*)) $\sqcup_S$ *x*) $\sqcup_S \bigsqcup_S$ (*removeAll x ys*)
by (*simp add*: *scene-union-commute*)
also have ... = $\bigsqcup_S$ (*filter* (λ*xa*. *xa* $\notin$ *set ys*) (*x # xs*)) $\sqcup_S$ ($x \sqcup_S \bigsqcup_S$ (*removeAll x ys*))
by (*simp add*: *scene-union-assoc True 2 3 4 scene-compat-sym*)
also have ... = $\bigsqcup_S$ (*filter* (λ*xa*. *xa* $\notin$ *set ys*) (*x # xs*)) $\sqcup_S \bigsqcup_S$ *ys*
by (*metis* (*no-types*, *lifting*) *Cons.prems*(*1*) *Cons.prems*(*2*) *True foldr-scene-union-removeAll pairwise-subset scene-union-commute*)
finally show *?thesis* **.**
next
case *False*
with *Cons*(*2*−*3*) **have** *1*: *set ys* $\subseteq$ *set xs*
by *auto*
have *2*: $x \#\#_S \bigsqcup_S$ (*filter* (λ*x*. *x* $\notin$ *set ys*) *xs*)
by (*metis* (*no-types*, *lifting*) *Cons.prems*(*1*) *filter-is-subset filter-set list.simps*(*15*) *member-filter pairwise-compat-foldr pairwise-insert pairwise-subset scene-compat-refl*)
have *3*: $x \#\#_S \bigsqcup_S$ *ys*
by (*meson Cons.prems*(*1*) *Cons.prems*(*2*) *list.set-intros*(*1*) *pairwise-compat-foldr pairwise-subset scene-compats-members subset-code*(*1*))
from *Cons*(*1*)[*of ys*] *Cons*(*2*−*3*) **have** *4*: $\bigsqcup_S$ (*filter* (λ*x*. *x* $\notin$ *set ys*) *xs*) $\#\#_S \bigsqcup_S$ *ys*
by (*auto simp add*: *pairwise-insert*)
(*metis* (*no-types*, *lifting*) *1 foldr-append foldr-scene-union-eq-sets scene-compat-bot*(*1*) *scene-union-incompat set-append subset-Un-eq*)

with *1 False Cons*(*1*)[*of ys*] *Cons*(*2*−*3*) **show** *?thesis*
by (*auto simp add*: *pairwise-insert scene-union-assoc 2 3 4*)
qed
qed

lemma *foldr-scene-append*:
$\llbracket$ *pairwise* $(\#\#_S)$ (*set* (*xs* @ *ys*)) $\rrbracket \Longrightarrow \bigsqcup_S$ (*xs* @ *ys*) $= \bigsqcup_S$ *xs* $\sqcup_S \bigsqcup_S$ *ys*
by (*simp add*: *foldr-scene-union-add-tail pairwise-compat-foldr pairwise-subset scene-compats-members*)

lemma *foldr-scene-concat*:
$\llbracket$ *pairwise* $(\#\#_S)$ (*set* (*concat xs*)) $\rrbracket \Longrightarrow \bigsqcup_S$ (*concat xs*) $= \bigsqcup_S$ (*map* $\bigsqcup_S$ *xs*)
by (*induct xs*, *simp-all*, *metis foldr-append foldr-scene-append pairwise-subset set-append set-concat sup-ge2*)

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*)
by (*metis local.indep-Vars pairwise-alt scene-indep-compat scene-indeps-def*)

lemma *Vars-ext-lens-indep*: $\llbracket$ $a ;_S x \neq b ;_S x$; $a \in$ *set Vars*; $b \in$ *set Vars* $\rrbracket \Longrightarrow a ;_S x \bowtie_S b ;_S x$
by (*metis indep-Vars pairwiseD scene-comp-indep scene-indeps-def*)

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*]: $\llbracket$ $x \in$ *scene-space*; $y \in$ *scene-space* $\rrbracket \Longrightarrow x \sqcup_S y \in$ *scene-space*

lemma *idem-scene-space*: $a \in$ *scene-space* $\Longrightarrow$ *idem-scene a*
by (*induct rule*: *scene-space.induct*) *auto*

lemma *set-Vars-scene-space* [*simp*]: *set Vars* $\subseteq$ *scene-space*
by *blast*

lemma *pairwise-compat-Vars-subset*: *set xs* $\subseteq$ *set Vars* $\Longrightarrow$ *pairwise* $(\#\#_S)$ (*set xs*)
using *pairwise-subset scene-space-compats* **by** *blast*

lemma *scene-space-foldr*: *set xs* $\subseteq$ *scene-space* $\Longrightarrow \bigsqcup_S$ *xs* $\in$ *scene-space*
by (*induction xs*, *auto*)

lemma *top-scene-eq*: $\top_S = \bigsqcup_S$ *Vars*

using *local.span-Vars scene-span-def* **by** *force*

lemma *top-scene-space*: $\top_S \in$ *scene-space*
proof −
have $\top_S$ = *foldr* ($\sqcup_S$) *Vars* $\bot_S$
using *span-Vars* **by** (*simp add*: *scene-span-def*)
also have ... ∈ *scene-space*
by (*simp add*: *scene-space-foldr*)
finally show *?thesis* **.**
qed

lemma *Vars-compat-scene-space*: ⟦ *b* ∈ *scene-space*; *x* ∈ *set Vars* ⟧ ⟹ $x \;\#\#_S\; b$
proof (*induct b rule*: *scene-space.induct*)
case *bot-scene-space*
then show *?case*
by (*metis scene-compat-refl scene-union-incompat scene-union-unit*(*1*))
next
case (*Vars-scene-space a*)
then show *?case*
by (*metis local.indep-Vars pairwiseD scene-compat-refl scene-indep-compat scene-indeps-def*)
next
case (*union-scene-space a b*)
then show *?case*
using *scene-union-pres-compat* **by** *blast*
qed

lemma *scene-space-compat*: ⟦ *a* ∈ *scene-space*; *b* ∈ *scene-space* ⟧ ⟹ $a \;\#\#_S\; b$
proof (*induct rule*: *scene-space.induct*)
case *bot-scene-space*
then show *?case*
by *simp*
next
case (*Vars-scene-space x*)
then show *?case*
by (*simp add*: *Vars-compat-scene-space*)
next
case (*union-scene-space x y*)
then show *?case*
using *scene-compat-sym scene-union-pres-compat* **by** *blast*
qed

corollary *scene-space-union-assoc*:
assumes *x* ∈ *scene-space y* ∈ *scene-space z* ∈ *scene-space*
shows $x \sqcup_S (y \sqcup_S z) = (x \sqcup_S y) \sqcup_S z$
by (*simp add*: *assms scene-space-compat scene-union-assoc*)

lemma *scene-space-vars-decomp*: *a* ∈ *scene-space* ⟹ ∃ *xs*. *set xs* ⊆ *set Vars* ∧ *foldr* ($\sqcup_S$) *xs* $\bot_S$ = *a*
proof (*induct rule*: *scene-space.induct*)
case *bot-scene-space*
then show *?case*
by (*simp add*: *exI*[**where** *x*=[]])
next
case (*Vars-scene-space x*)
show *?case*
apply (*rule exI*[**where** *x*=[*x*]])

using *Vars-scene-space* **by** *simp*
next
case (*union-scene-space x y*)
then obtain *xs ys* **where** *xsys*: *set xs* $\subseteq$ *set Vars* $\wedge$ *foldr* $(\sqcup_S)$ *xs* $\perp_S$ = *x*
set ys $\subseteq$ *set Vars* $\wedge$ *foldr* $(\sqcup_S)$ *ys* $\perp_S$ = *y*
by *blast+*
show *?case*
proof (*rule exI*[**where** *x=xs @ ys*])
show *set* (*xs @ ys*) $\subseteq$ *set Vars* $\wedge$ $\bigsqcup_S$ (*xs @ ys*) = *x* $\sqcup_S$ *y*
by (*auto simp*: *xsys*)
(*metis* (*full-types*) *Vars-compat-scene-space foldr-scene-union-add-tail pairwise-subset scene-space-compats subsetD union-scene-space.hyps*(*3*) *xsys*(*1*))
qed
qed

lemma *scene-space-vars-decomp-iff*: *a* $\in$ *scene-space* $\longleftrightarrow$ ($\exists$ *xs. set xs* $\subseteq$ *set Vars* $\wedge$ *a* = *foldr* $(\sqcup_S)$ *xs* $\perp_S$)
apply (*auto simp add*: *scene-space-vars-decomp scene-space.Vars-scene-space scene-space-foldr*)
apply (*simp add*: *scene-space.Vars-scene-space scene-space-foldr subset-eq*)
using *scene-space-vars-decomp* **apply** *auto*[*1*]
by (*meson dual-order.trans scene-space-foldr set-Vars-scene-space*)

lemma *fold* $(\sqcup_S)$ (*map* ($\lambda x.\ x ;_S a$) *Vars*) *b* = $[\![a]\!]_\sim \sqcup_S b$
oops

lemma *Vars-indep-foldr*:
assumes *x* $\in$ *set Vars set xs* $\subseteq$ *set Vars*
shows *x* $\bowtie_S$ $\bigsqcup_S$ (*removeAll x xs*)
proof (*rule foldr-scene-indep*)
show *pairwise* ($\#\#_S$) (*set* (*removeAll x xs*))
by (*simp*, *metis Diff-subset assms*(*2*) *pairwise-mono scene-space-compats*)
from *assms* **show** $\forall$ *b*$\in$*set* (*removeAll x xs*). *x* $\bowtie_S$ *b*
by (*simp*)
(*metis DiffE insertI1 local.indep-Vars pairwiseD scene-indeps-def subset-iff*)
qed

lemma *Vars-indeps-foldr*:
assumes *set xs* $\subseteq$ *set Vars*
shows *foldr* $(\sqcup_S)$ *xs* $\perp_S$ $\bowtie_S$ *foldr* $(\sqcup_S)$ (*filter* ($\lambda x.\ x \notin$ *set xs*) *Vars*) $\perp_S$
apply (*rule foldr-scene-indep*)
apply (*meson filter-is-subset pairwise-subset scene-space-compats*)
apply (*simp*)
apply *auto*
apply (*rule scene-indep-sym*)
apply (*metis* (*no-types*, *lifting*) *assms foldr-scene-indep local.indep-Vars pairwiseD pairwise-mono scene-indeps-def scene-space-compats subset-iff*)
done

lemma *uminus-var-other-vars*:
assumes *x* $\in$ *set Vars*
shows $-$ *x* = *foldr* $(\sqcup_S)$ (*removeAll x Vars*) $\perp_S$
proof (*rule scene-union-indep-uniq*[**where** *Z=x*])
show *idem-scene* (*foldr* $(\sqcup_S)$ (*removeAll x Vars*) $\perp_S$)
by (*metis Diff-subset idem-scene-space order-trans scene-space-foldr set-Vars-scene-space set-removeAll*)
show *idem-scene x idem-scene* ($-x$)

by (*simp-all add*: *assms local.idem-scene-Vars*)
show *foldr* ($\sqcup_S$) (*removeAll x Vars*) $\bot_S \bowtie_S x$
using *Vars-indep-foldr assms scene-indep-sym* **by** *blast*
show $- x \bowtie_S x$
using *scene-indep-self-compl scene-indep-sym* **by** *blast*
show $- x \sqcup_S x =$ *foldr* ($\sqcup_S$) (*removeAll x Vars*) $\bot_S \sqcup_S x$
by (*metis* ‹*idem-scene* (− *x*)› *assms foldr-scene-union-removeAll local.span-Vars scene-space-compats scene-span-def scene-union-compl uminus-scene-twice*)
qed

lemma *uminus-vars-other-vars*:
assumes *set xs* ⊆ *set Vars*
shows $- \bigsqcup_S xs = \bigsqcup_S$ (*filter* ($\lambda x.\ x \notin$ *set xs*) *Vars*)
proof (*rule scene-union-indep-uniq*[**where** *Z=foldr* ($\sqcup_S$) *xs* $\bot_S$])
show *idem-scene* (− *foldr* ($\sqcup_S$) *xs* $\bot_S$) *idem-scene* (*foldr* ($\sqcup_S$) *xs* $\bot_S$)
using *assms idem-scene-space idem-scene-uminus scene-space-vars-decomp-iff* **by** *blast+*
show *idem-scene* (*foldr* ($\sqcup_S$) (*filter* ($\lambda x.\ x \notin$ *set xs*) *Vars*) $\bot_S$)
by (*meson filter-is-subset idem-scene-space scene-space-vars-decomp-iff*)
show − *foldr* ($\sqcup_S$) *xs* $\bot_S \bowtie_S$ *foldr* ($\sqcup_S$) *xs* $\bot_S$
by (*metis scene-indep-self-compl uminus-scene-twice*)
show *foldr* ($\sqcup_S$) (*filter* ($\lambda x.\ x \notin$ *set xs*) *Vars*) $\bot_S \bowtie_S$ *foldr* ($\sqcup_S$) *xs* $\bot_S$
using *Vars-indeps-foldr assms scene-indep-sym* **by** *blast*
show $- \bigsqcup_S xs \sqcup_S \bigsqcup_S xs = \bigsqcup_S$ (*filter* ($\lambda x.\ x \notin$ *set xs*) *Vars*) $\sqcup_S \bigsqcup_S xs$
using *foldr-scene-union-filter*[*of Vars xs*, *THEN sym*]
by (*simp add*: *assms*)
(*metis* ‹*idem-scene* ($- \bigsqcup_S xs$)› *local.span-Vars scene-span-def scene-union-compl uminus-scene-twice*)
qed

lemma *scene-space-uminus*: ⟦ *a* ∈ *scene-space* ⟧ ⟹ − *a* ∈ *scene-space*
by (*auto simp add*: *scene-space-vars-decomp-iff uminus-vars-other-vars*)
(*metis filter-is-subset*)

lemma *scene-space-inter*: ⟦ *a* ∈ *scene-space*; *b* ∈ *scene-space* ⟧ ⟹ $a \sqcap_S b$ ∈ *scene-space*
by (*simp add*: *inf-scene-def scene-space.union-scene-space scene-space-uminus*)

lemma *scene-union-foldr-remove-element*:
assumes *set xs* ⊆ *set Vars*
shows $a \sqcup_S \bigsqcup_S xs = a \sqcup_S \bigsqcup_S$ (*removeAll a xs*)
using *assms* **proof** (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case* **apply** *auto*
apply (*metis order-trans scene-space.Vars-scene-space scene-space-foldr scene-space-union-assoc scene-union-idem set-Vars-scene-space*)
apply (*smt* (*verit*, *best*) *Diff-subset dual-order.trans removeAll-id scene-space-foldr scene-space-union-assoc scene-union-commute set-Vars-scene-space set-removeAll subset-iff*)
done
qed

lemma *scene-union-foldr-Cons-removeAll*:
assumes *set xs* ⊆ *set Vars a* ∈ *set xs*
shows *foldr* ($\sqcup_S$) *xs* $\bot_S$ = *foldr* ($\sqcup_S$) (*a* # *removeAll a xs*) $\bot_S$
by (*metis assms*(*1*) *assms*(*2*) *foldr-scene-union-eq-sets insert-Diff list.simps*(*15*) *pairwise-subset scene-space-compats*

set-removeAll)

lemma *scene-union-foldr-Cons-removeAll′*:
 assumes *set xs* ⊆ *set Vars a* ∈ *set Vars*
 shows *foldr* (⊔$_S$) (*a* # *xs*) $\bot_S$ = *foldr* ($\sqcup_S$) (*a* # *removeAll a xs*) $\bot_S$
 by (*simp add*: *assms*(*1*) *scene-union-foldr-remove-element*)

lemma *scene-in-foldr*: ⟦ *a* ∈ *set xs*; *set xs* ⊆ *set Vars* ⟧ ⟹ *a* $\subseteq_S$ $\bigsqcup_S$ *xs*
 apply (*induct xs*)
  apply (*simp*)
 apply (*subst scene-union-foldr-Cons-removeAll′*)
   apply *simp*
  apply *simp*
 apply (*auto*)
 apply (*rule scene-union-ub*)
  apply (*metis Diff-subset dual-order.trans idem-scene-space scene-space-vars-decomp-iff set-removeAll*)
 using *Vars-indep-foldr* **apply** *blast*
 apply (*metis Vars-indep-foldr foldr-scene-union-removeAll idem-scene-space local.idem-scene-Vars order.trans pairwise-mono removeAll-id scene-indep-sym scene-space-compats scene-space-foldr scene-union-commute scene-union-ub set-Vars-scene-space subscene-trans*)
 done

lemma *scene-union-foldr-subset*:
 assumes *set xs* ⊆ *set ys set ys* ⊆ *set Vars*
 shows $\bigsqcup_S$ *xs* $\subseteq_S$ $\bigsqcup_S$ *ys*
 using *assms* **proof** (*induct xs arbitrary*: *ys*)
 case *Nil*
 then show *?case*
  by (*simp add*: *scene-bot-least*)
next
 case (*Cons a xs*)
 { assume *a* ∈ *set xs*
  with *Cons* **have** *foldr* ($\sqcup_S$) *xs* $\bot_S$ = *foldr* ($\sqcup_S$) (*a* # *removeAll a xs*) $\bot_S$
   apply (*subst scene-union-foldr-Cons-removeAll*)
    apply (*auto*)
   done
 } note *a-in* = *this*
 { assume *a* ∉ *set xs*
  then have *a* $\sqcup_S$ *foldr* ($\sqcup_S$) *xs* $\bot_S$ = *foldr* ($\sqcup_S$) (*a* # *xs*) $\bot_S$
   by *simp*
 } note *a-out* = *this*
 show *?case* **apply** (*simp*)
  apply (*cases a* ∈ *set xs*)
  using *a-in Cons* **apply** *auto*
   apply (*metis dual-order.trans scene-union-foldr-remove-element*)
  using *a-out Cons* **apply** *auto*
  apply (*rule scene-union-mono*)
  using *scene-in-foldr* **apply** *blast*
    apply *blast*
    apply (*meson Vars-compat-scene-space dual-order.trans scene-space-foldr set-Vars-scene-space subsetD*)
  using *local.idem-scene-Vars* **apply** *blast*
  apply (*meson idem-scene-space scene-space-foldr set-Vars-scene-space subset-trans*)
  done
qed

lemma *union-scene-space-foldrs*:
assumes *set xs* $\subseteq$ *set Vars set ys* $\subseteq$ *set Vars*
shows (*foldr* ($\sqcup_S$) *xs* $\perp_S$) $\sqcup_S$ (*foldr* ($\sqcup_S$) *ys* $\perp_S$) = *foldr* ($\sqcup_S$) (*xs* @ *ys*) $\perp_S$
using *assms*
apply (*induct ys*)
apply (*simp-all*)
apply (*metis Vars-compat-scene-space foldr-scene-union-add-tail local.indep-Vars pairwise-mono scene-indep-compat scene-indeps-def scene-space.Vars-scene-space scene-space.union-scene-space scene-space-foldr subset-eq*)
done

lemma *scene-space-ub*:
assumes $a \in$ *scene-space* $b \in$ *scene-space*
shows $a \subseteq_S a \sqcup_S b$
using *assms*
apply (*auto simp add*: *scene-space-vars-decomp-iff union-scene-space-foldrs*)
by (*smt* (*verit*, *ccfv-SIG*) *foldr-append scene-union-foldr-subset set-append sup.bounded-iff sup-commute sup-ge2*)

lemma *scene-compl-subset-iff*:
assumes $a \in$ *scene-space* $b \in$ *scene-space*
shows $- a \subseteq_S -b \longleftrightarrow b \subseteq_S a$
by (*metis scene-indep-sym scene-le-iff-indep-inv uminus-scene-twice*)

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* (λ *x*. *x* $\in$ *set xs* $\cap$ *set ys*) *Vars*)
proof −
have $\bigsqcup_S$ *xs* $\sqcap_S$ $\bigsqcup_S$ *ys* = − (− $\bigsqcup_S$ *xs* $\sqcup_S$ − $\bigsqcup_S$ *ys*)
by (*simp add*: *inf-scene-def*)
also have ... = − ($\bigsqcup_S$ (*filter* (λ*x*. *x* $\notin$ *set xs*) *Vars*) $\sqcup_S$ $\bigsqcup_S$ (*filter* (λ*x*. *x* $\notin$ *set ys*) *Vars*))
by (*simp add*: *uminus-vars-other-vars assms*)
also have ... = − $\bigsqcup_S$ (*filter* (λ*x*. *x* $\notin$ *set xs*) *Vars* @ *filter* (λ*x*. *x* $\notin$ *set ys*) *Vars*)
by (*simp add*: *union-scene-space-foldrs assms*)
also have ... = $\bigsqcup_S$ (*filter* (λ*x*. *x* $\notin$ *set* (*filter* (λ*x*. *x* $\notin$ *set xs*) *Vars* @ *filter* (λ*x*. *x* $\notin$ *set ys*) *Vars*)) *Vars*)
by (*subst uminus-vars-other-vars*, *simp-all*)
also have ... = $\bigsqcup_S$ (*filter* (λ *x*. *x* $\in$ *set xs* $\cap$ *set ys*) *Vars*)
proof −
have $\bigwedge$*x*. *x* $\in$ *set Vars* $\Longrightarrow$ ((*x* $\in$ *set Vars* $\longrightarrow$ *x* $\in$ *set xs*) $\wedge$ (*x* $\in$ *set Vars* $\longrightarrow$ *x* $\in$ *set ys*)) = (*x* $\in$ *set xs* $\wedge$ *x* $\in$ *set ys*)
by *auto*
thus *?thesis*
by (*simp cong*: *arg-cong*[**where** *f*=$\bigsqcup_S$] *filter-cong add*: *assms*)
qed
finally show *?thesis* **.**
qed

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*)
using *assms*
apply (*simp only*: *union-scene-space-foldrs inter-scene-space-foldrs*)
apply (*subst union-scene-space-foldrs*)
apply (*simp add*: *assms*)

apply (*simp add*: *assms*)
apply (*subst inter-scene-space-foldrs*)
apply (*simp*)
apply (*simp*)
apply (*rule foldr-scene-union-eq-sets*)
apply (*simp*)
apply (*smt* (*verit*, *ccfv-threshold*) *Un-subset-iff mem-Collect-eq pairwise-subset scene-space-compats subset-iff*)
apply (*auto*)
done

lemma *scene-union-inter-distrib*:
assumes $a \in$ *scene-space* $b \in$ *scene-space* $c \in$ *scene-space*
shows $a \sqcup_S b \sqcap_S c = (a \sqcup_S b) \sqcap_S (a \sqcup_S c)$
using *assms*
by (*auto simp add*: *scene-space-vars-decomp-iff scene-inter-distrib-lemma*)

lemma *finite-distinct-lists-subset*:
assumes *finite A*
shows *finite* $\{xs.$ *distinct* $xs \wedge$ *set* $xs \subseteq A\}$
by (*metis* (*no-types*, *lifting*) *Collect-cong finite-subset-distinct*[*OF assms*])

lemma *foldr-scene-union-remdups*: *set* $xs \subseteq$ *set Vars* $\Longrightarrow \bigsqcup_S$ (*remdups* xs) $= \bigsqcup_S xs$
by (*auto intro*: *foldr-scene-union-eq-sets simp add*: *pairwise-compat-Vars-subset*)

lemma *scene-space-as-lists*:
scene-space $= \{\bigsqcup_S xs \mid xs.$ *distinct* $xs \wedge$ *set* $xs \subseteq$ *set Vars*$\}$
proof (*rule Set.set-eqI*, *rule iffI*)
fix a
assume $a \in$ *scene-space*
then obtain xs **where** xs: *set* $xs \subseteq$ *set Vars* $\bigsqcup_S xs = a$
using *scene-space-vars-decomp-iff* **by** *auto*
thus $a \in \{\bigsqcup_S xs \mid xs.$ *distinct* $xs \wedge$ *set* $xs \subseteq$ *set Vars*$\}$
by *auto* (*metis distinct-remdups foldr-scene-union-remdups set-remdups*)
next
fix a
assume $a \in \{\bigsqcup_S xs \mid xs.$ *distinct* $xs \wedge$ *set* $xs \subseteq$ *set Vars*$\}$
thus $a \in$ *scene-space*
using *scene-space-vars-decomp-iff* **by** *auto*
qed

lemma *finite-scene-space*: *finite scene-space*
proof −
have *scene-space* $= \{\bigsqcup_S xs \mid xs.$ *distinct* $xs \wedge$ *set* $xs \subseteq$ *set Vars*$\}$
by (*simp add*: *scene-space-as-lists*)
also have ... $= \bigsqcup_S$ ‘ $\{xs.$ *distinct* $xs \wedge$ *set* $xs \subseteq$ *set Vars*$\}$
by *auto*
also have *finite* ...
by (*rule finite-imageI*, *simp add*: *finite-distinct-lists-subset*)
finally show *?thesis* **.**
qed

lemma *scene-space-inter-assoc*:
assumes $x \in$ *scene-space* $y \in$ *scene-space* $z \in$ *scene-space*
shows $(x \sqcap_S y) \sqcap_S z = x \sqcap_S (y \sqcap_S z)$

proof −
  have $(x \sqcap_S y) \sqcap_S z = - (- x \sqcup_S - y \sqcup_S - z)$
    by (*simp add*: *scene-demorgan1 uminus-scene-twice*)
  also have ... $= - (- x \sqcup_S (- y \sqcup_S - z))$
    by (*simp add*: *assms scene-space-uminus scene-space-union-assoc*)
  also have ... $= x \sqcap_S (y \sqcap_S z)$
    by (*simp add*: *scene-demorgan1 uminus-scene-twice*)
  finally show *?thesis* **.**
qed

lemma *scene-inter-union-distrib*:
  assumes $x \in$ *scene-space* $y \in$ *scene-space* $z \in$ *scene-space*
  shows $x \sqcap_S (y \sqcup_S z) = (x \sqcap_S y) \sqcup_S (x \sqcap_S z)$
proof−
  have $x \sqcap_S (y \sqcup_S z) = (x \sqcap_S (x \sqcup_S z)) \sqcap_S (y \sqcup_S z)$
   by (*metis assms*(*1*) *assms*(*3*) *idem-scene-space local.scene-union-inter-distrib scene-indep-bot scene-inter-commute scene-inter-indep scene-space.simps scene-union-unit*(*1*))
  also have ... $= (y \sqcup_S z) \sqcap_S (x \sqcap_S (x \sqcup_S z))$
   by (*simp add*: *scene-union-inter-distrib assms scene-inter-commute scene-union-assoc union-scene-space scene-space-inter scene-union-commute*)
  also have . . . $= x \sqcap_S ((y \sqcup_S z) \sqcap_S (x \sqcup_S z))$
    by (*metis assms scene-inter-commute scene-space.union-scene-space scene-space-inter-assoc*)
  also have . . . $= x \sqcap_S (z \sqcup_S (x \sqcap_S y))$
    by (*simp add*: *assms scene-union-inter-distrib scene-inter-commute scene-union-commute*)

  also have . . . $= ((x \sqcap_S y) \sqcup_S x) \sqcap_S ((x \sqcap_S y) \sqcup_S z)$
   by (*metis* (*no-types*, *opaque-lifting*) *assms*(*1*) *assms*(*2*) *idem-scene-space local.scene-union-inter-distrib scene-indep-bot scene-inter-commute scene-inter-indep scene-space.bot-scene-space scene-union-commute scene-union-idem scene-union-unit*(*1*))
  also have . . . $= (x \sqcap_S y) \sqcup_S (x \sqcap_S z)$
    by (*simp add*: *assms scene-union-inter-distrib scene-space-inter*)
  finally show *?thesis* **.**
qed

lemma *scene-union-inter-minus*:
  assumes $a \in$ *scene-space* $b \in$ *scene-space*
  shows $a \sqcup_S (b \sqcap_S - a) = a \sqcup_S b$
 by (*metis assms*(*1*) *assms*(*2*) *bot-idem-scene idem-scene-space idem-scene-uminus local.scene-union-inter-distrib scene-demorgan1 scene-space-uminus scene-union-compl scene-union-unit*(*1*) *uminus-scene-twice*)

lemma *scene-union-foldr-minus-element*:
  assumes $a \in$ *scene-space set xs* $\subseteq$ *scene-space*
  shows $a \sqcup_S \bigsqcup_S xs = a \sqcup_S \bigsqcup_S (map\ (\lambda\ x.\ x \sqcap_S - a)\ xs)$
using *assms* **proof** (*induct xs*)
  case *Nil*
  then show *?case* **by** (*simp*)
next
  case (*Cons y ys*)
  have $a \sqcup_S (y \sqcup_S \bigsqcup_S ys) = y \sqcup_S (a \sqcup_S \bigsqcup_S ys)$
   by (*metis Cons.prems*(*2*) *assms*(*1*) *insert-subset list.simps*(*15*) *scene-space-foldr scene-space-union-assoc scene-union-commute*)
  also have ... $= y \sqcup_S (a \sqcup_S \bigsqcup_S (map\ (\lambda x.\ x \sqcap_S - a)\ ys))$
    using *Cons.hyps Cons.prems*(*2*) *assms*(*1*) **by** *auto*
  also have ... $= y \sqcup_S a \sqcup_S \bigsqcup_S (map\ (\lambda x.\ x \sqcap_S - a)\ ys)$
    apply (*subst scene-union-assoc*)

```
    using Cons.prems(2) assms(1) scene-space-compat apply auto[1]
      apply (rule pairwise-compat-foldr)
       apply (simp)
       apply (rule pairwise-imageI)
     apply (meson Cons.prems(2) assms(1) scene-space-compat scene-space-inter scene-space-uminus
set-subset-Cons subsetD)
    apply simp
   apply (meson Cons.prems(2) assms(1) in-mono list.set-intros(1) scene-space-compat scene-space-inter
scene-space-uminus set-subset-Cons)
     apply (rule pairwise-compat-foldr)
    apply (simp)
       apply (rule pairwise-imageI)
   apply (meson Cons.prems(2) assms(1) in-mono scene-space-compat scene-space-inter scene-space-uminus
set-subset-Cons)
    apply (simp)
    apply (meson Cons.prems(2) assms(1) in-mono scene-space-compat scene-space-inter scene-space-uminus
set-subset-Cons)
    apply simp
    done
  also have ... = a ⊔S (y ⊓S − a ⊔S ⨆S (map (λx. x ⊓S − a) ys))
    apply (subst scene-union-assoc)
   using Cons.prems(2) assms(1) scene-space-compat scene-space-inter scene-space-uminus apply force
       apply (metis (no-types, lifting) Cons.hyps Cons.prems(2) assms(1) insert-subset list.simps(15)
scene-compat-sym scene-space-compat scene-space-foldr scene-union-assoc scene-union-idem scene-union-incompat
scene-union-unit(1))
    apply (rule scene-space-compat)
    using Cons.prems(2) assms(1) scene-space-inter scene-space-uminus apply auto[1]
     apply (rule scene-space-foldr)
    apply auto
   apply (meson Cons.prems(2) assms(1) in-mono scene-space-inter scene-space-uminus set-subset-Cons)
   apply (metis Cons.prems(2) assms(1) insert-subset list.simps(15) scene-union-inter-minus scene-union-commute)
    done
  finally show ?case using Cons
    by auto
qed

lemma scene-space-in-foldr: ⟦ a ∈ set xs; set xs ⊆ scene-space ⟧ ⟹ a ⊆S ⨆S xs
proof (induct xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons y ys)
  have ysp: y ⊔S ⨆S ys = y ⊔S ⨆S (map (λ x. x ⊓S − y) ys)
    using Cons.prems(2) scene-union-foldr-minus-element by force
  show ?case
  proof (cases a ⊆S y)
    case False
    with Cons show ?thesis
      by (simp)
        (metis (no-types, lifting) idem-scene-space scene-space-foldr scene-space-ub scene-union-commute
subscene-trans)
  next
    case True
    with Cons show ?thesis
```

by (*simp*)
(*meson idem-scene-space scene-space-foldr scene-space-ub subscene-trans*)
qed
qed

lemma *scene-space-foldr-lb*:
$[\![a \in scene\text{-}space; set\ xs \subseteq scene\text{-}space; \forall\ b \in set\ xs.\ b \leq a]\!] \Longrightarrow \bigsqcup_S xs \subseteq_S a$
proof (*induct xs arbitrary*: *a*)
case *Nil*
then show *?case*
by (*simp add*: *scene-bot-least*)
next
case (*Cons x xs*)
then show *?case*
by (*simp add*: *scene-space-compat scene-space-foldr scene-union-lb*)
qed

lemma *var-le-union-choice*:
$[\![x \in set\ Vars; a \in scene\text{-}space; b \in scene\text{-}space; x \leq a \sqcup_S b]\!] \Longrightarrow (x \leq a \vee x \leq b)$
by (*auto simp add*: *scene-space-vars-decomp-iff*)
(*metis Vars-indep-foldr bot-idem-scene idem-scene-space removeAll-id scene-bot-least scene-indep-pres-compat scene-le-iff-indep-inv scene-space.union-scene-space scene-space-foldr scene-space-in-foldr scene-union-compl set-Vars-scene-space subscene-trans subset-trans uminus-scene-twice uminus-top-scene*)

lemma *var-le-union-iff*:
$[\![x \in set\ Vars; a \in scene\text{-}space; b \in scene\text{-}space]\!] \Longrightarrow x \leq a \sqcup_S b \longleftrightarrow (x \leq a \vee x \leq b)$
apply (*rule iffI*, *simp add*: *var-le-union-choice*)
apply (*auto*)
apply (*meson idem-scene-space scene-space-ub subscene-trans*)
apply (*metis idem-scene-space scene-space-ub scene-union-commute subscene-trans*)
done

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

lemma *le-vars-then-equal*: $[\![x \in set\ Vars; y \in set\ Vars; x \leq y; x \neq \bot_S]\!] \Longrightarrow x = y$
by (*metis bot-idem-scene foldr-scene-removeAll local.idem-scene-Vars local.indep-Vars local.span-Vars pairwiseD scene-bot-least scene-indep-pres-compat scene-indeps-def scene-le-iff-indep-inv scene-space-compats scene-span-def scene-union-annhil subscene-antisym uminus-scene-twice uminus-top-scene uminus-var-other-vars*)

end

lemma *foldr-scene-union-eq-scene-space*:
$[\![set\ xs \subseteq scene\text{-}space; set\ xs = set\ ys]\!] \Longrightarrow \bigsqcup_S xs = \bigsqcup_S ys$
by (*metis foldr-scene-union-eq-sets pairwise-def pairwise-subset scene-space-compat*)

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*:
$[\![pairwise\ (\#\#_S)\ (set\ xs); vwb\text{-}lens\ a]\!] \Longrightarrow \bigsqcup_S (map\text{-}lcomp\ xs\ a) = \bigsqcup_S xs\ ;_S\ a$
by (*simp add*: *foldr-compat-dist map-lcomp-def*)

lemma *map-lcomp-Vars-is-lens* [*simp*]: $vwb\text{-}lens\ a \Longrightarrow \bigsqcup_S (map\text{-}lcomp\ Vars\ a) = [\![a]\!]_\sim$
by (*metis map-lcomp-dist scene-comp-top-scene scene-space-compats top-scene-eq*)

lemma *set-map-lcomp* [*simp*]: *set* (*map-lcomp xs a*) = ($\lambda x.\ x\ ;_S\ a$) ‘ *set xs*
by (*simp add*: *map-lcomp-def*)

8.5 Instances

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

definition *Vars-unit* :: *unit scene list* **where** [*simp*]: *Vars-unit* = []

instance
by (*intro-classes*, *simp-all add*: *scene-indeps-def scene-span-def unit-scene-top-eq-bot*)

end

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

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

instance proof
have *pw*: *pairwise* ($\bowtie_S$) (*set* (*map-lcomp Vars* fst_L @ *map-lcomp Vars* snd_L))
by (*auto simp add*: *pairwise-def Vars-ext-lens-indep scene-comp-pres-indep scene-indep-sym*)
show $\bigwedge x$:: ($'a \times 'b$) *scene*. $x \in$ *set Vars* $\Longrightarrow$ *idem-scene x*
by (*auto simp add*: *Vars-prod-def*)
from *pw* **show** *scene-indeps* (*set* (*Vars* :: ($'a \times 'b$) *scene list*))
by (*simp add*: *Vars-prod-def scene-indeps-def*)
show *scene-span* (*Vars* :: ($'a \times 'b$) *scene list*)
by (*simp only*: *scene-span-def Vars-prod-def foldr-scene-append pw pairwise-indep-then-compat map-lcomp-Vars-is-lens fst-vwb-lens snd-vwb-lens*)
(*metis fst-vwb-lens lens-plus-scene lens-scene-top-iff-bij-lens plus-mwb-lens scene-union-commute snd-fst-lens-indep snd-vwb-lens swap-bij-lens vwb-lens-mwb*)
qed

end

8.6 Scene space and basis lenses

locale *var-lens* = *vwb-lens* +
assumes *lens-in-scene-space*: $[\![x]\!]_\sim \in$ *scene-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*
using *lens-in-basis var-lens-axioms-def var-lens-def vwb-lens-axioms* **by** *blast*

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* $\Longrightarrow$ $'s$::*scene-space*) $\Rightarrow$ *bool*
where *evar-lens* ≡ *var-lens*

abbreviation (*input*) *ebasis-lens* :: ($'a$::*two* $\Longrightarrow$ $'s$::*scene-space*) $\Rightarrow$ *bool*
  where *ebasis-lens* $\equiv$ *basis-lens*

lemma *basis-then-var* [*simp*]: *basis-lens x* $\Longrightarrow$ *var-lens x*
  using *basis-lens.lens-in-basis basis-lens-def var-lens-axioms-def var-lens-def* **by** *blast*

lemma *basis-lens-intro*: $[\![$ *vwb-lens x*; $[\![x]\!]_\sim \in$ *set Vars* $]\!] \Longrightarrow$ *basis-lens x*
  using *basis-lens.intro basis-lens-axioms.intro* **by** *blast*

8.7 Composite lenses

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

lemma *Vars-closed-comp*: $a \in$ *set Vars* $\Longrightarrow a\ ;_S\ x \in$ *set Vars*
  using *comp-in-Vars* **by** *blast*

lemma *scene-space-closed-comp*:
  assumes $a \in$ *scene-space*
  shows $a\ ;_S\ x \in$ *scene-space*
proof −
  obtain *xs* **where** *xs*: $a = \bigsqcup_S$ *xs set xs* $\subseteq$ *set Vars*
    using *assms scene-space-vars-decomp* **by** *blast*
  have $(\bigsqcup_S xs)\ ;_S\ x = \bigsqcup_S$ (*map* $(\lambda\ a.\ a\ ;_S\ x)$ *xs*)
    by (*metis foldr-compat-dist pairwise-subset scene-space-compats xs*(*2*))
  also have ... $\in$ *scene-space*
    by (*auto simp add*: *scene-space-vars-decomp-iff*)
      (*metis comp-in-Vars image-Un le-iff-sup le-supE list.set-map xs*(*2*))
  finally show *?thesis*
    by (*simp add*: *xs*)
qed

sublocale *var-lens*
proof
  show $[\![x]\!]_\sim \in$ *scene-space*
    by (*metis scene-comp-top-scene scene-space-closed-comp top-scene-space vwb-lens-axioms*)
qed

end

lemma *composite-implies-var-lens* [*simp*]:
  composite-lens x $\Longrightarrow$ *var-lens x*
  by (*metis composite-lens.axioms*(*1*) *composite-lens.scene-space-closed-comp scene-comp-top-scene top-scene-space var-lens-axioms.intro var-lens-def*)

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* $]\!] \Longrightarrow$ *var-lens* $(x\ ;_L\ a)$
  by (*metis comp-vwb-lens composite-lens.scene-space-closed-comp composite-lens-def lens-scene-comp var-lens-axioms-def var-lens-def*)

lemma *comp-composite-lens* [*simp*]:
  $[\![$ *composite-lens a*; *composite-lens x* $]\!] \Longrightarrow$ *composite-lens* $(x\ ;_L\ a)$
  by (*auto intro*!: *composite-lens.intro simp add*: *composite-lens-axioms-def*)

(*metis composite-lens.Vars-closed-comp composite-lens.axioms*(*1*) *scene-comp-assoc*)

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* ⟧ ⟹ *basis-lens* ($x ;_L a$)
by (*metis basis-lens.lens-in-basis basis-lens-def basis-lens-intro comp-vwb-lens composite-lens.Vars-closed-comp composite-lens-def lens-scene-comp*)

lemma *id-composite-lens*: *composite-lens* 1_L
by (*force intro*: *composite-lens.intro composite-lens-axioms.intro*)

lemma *fst-composite-lens*: *composite-lens* fst_L
by (*rule composite-lens.intro*, *simp add*: *fst-vwb-lens*, *rule composite-lens-axioms.intro*, *simp add*: *Vars-prod-def*)

lemma *snd-composite-lens*: *composite-lens* snd_L
by (*rule composite-lens.intro*, *simp add*: *snd-vwb-lens*, *rule composite-lens-axioms.intro*, *simp add*: *Vars-prod-def*)

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* ⇒ (*'b*::*two* ⟹ (*'a* ⇒ *'b*)) **where**
[*lens-defs*]: *fun-lens x* = ⦇ *lens-get* = (λ *f*. *f x*), *lens-put* = (λ *f u*. *f*(*x* := *u*)) ⦈

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

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

lemma *fun-lens-indep*:
fun-lens x ⋈ *fun-lens y* ⟷ *x* ≠ *y*
proof −
obtain *u v* :: *'a*::*two* **where** *u* ≠ *v*
using *two-diff* **by** *auto*
thus *?thesis*
by (*auto simp add*: *fun-lens-def lens-indep-def*)
qed

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

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

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

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

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* $\mapsto$ *u*)) |)

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

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

lemma *pget-map-lens*: $pget_{map\text{-}lens\ k}\ f = f\ k$
by (*auto simp add*: *lens-partial-get-def source-map-lens*)
(*auto simp add*: *map-lens-def*, *metis not-Some-eq*)

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

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

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

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

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

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

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

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

We can also prove some laws about our new operators.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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-augment xs k v = (*if* (*k* < *length xs*) *then list-update xs k v else list-update* (*list-pad-out xs k*) *k v*)
  by (*simp add*:*list-pad-out-def list-augment-def*)

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

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

abbreviation *hd-lens* (hd_L) **where** *hd-lens* $\equiv$ *list-lens 0*

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

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

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

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

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

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

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

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

lemma *hd-tl-lens-indep* [*simp*]:
  $hd_L \bowtie tl_L$
  apply (*rule lens-indepI*)
    apply (*simp-all add*: *list-lens-def tl-lens-def*)
    apply (*metis hd-conv-nth hd-def length-greater-0-conv list.case*(*1*) *nth'-def nth'-list-augment*)

apply (*metis* (*full-types*) *hd-conv-nth hd-def length-greater-0-conv list.case*(*1*) *nth'-def*)
apply (*metis One-nat-def diff-Suc-Suc diff-zero length-0-conv length-list-augment-1 length-tl linorder-not-less list.exhaust list.sel*(*2*) *list.sel*(*3*) *list-augment-0 not-less-zero*)
done

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

9.5 Stream Lenses

primrec *stream-update* :: *'a stream* ⇒ *nat* ⇒ *'a* ⇒ *'a stream* **where**
stream-update xs 0 a = *a##*(*stl xs*) |
stream-update xs (*Suc n*) *a* = *shd xs ##* (*stream-update* (*stl xs*) *n a*)

lemma *stream-update-snth*: (*stream-update xs n a*) !! *n* = *a*
proof (*induction n arbitrary*: *xs a*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case* **by** *simp*
qed

lemma *stream-update-unchanged*: *i* ≠ *j* ⟹ (*stream-update xs i a*) !! *j* = *xs* !! *j*
using *gr0-conv-Suc* **by** (*induct i j arbitrary*: *xs rule*: *diff-induct*; *fastforce*)

lemma *stream-update-override*: *stream-update* (*stream-update xs n a*) *n b* = *stream-update xs n b*
by (*induction n arbitrary*: *xs a*; *simp*)

lemma *stream-update-nth*: *stream-update* σ *i* (σ !! *i*) = σ
by (*metis stream.map-cong stream-smap-nats stream-update-snth stream-update-unchanged*)

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*)
apply (*unfold-locales*; *simp add*: *stream-lens-def*)
apply (*rule stream-update-snth*)
apply (*rule stream-update-nth*)
apply (*rule stream-update-override*)
done

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

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

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

```
named-theorems lens

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

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

```
named-theorems alpha-splits
```

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-file ‹Lens-Statespace.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* $typedef_L$
  by (*unfold-locales*, *simp-all add*: *lens-defs Rep-inverse*)

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

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

end

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

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

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

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 :: $'v \Rightarrow 's$ (*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*
using *build-match match-build* **by** *blast*

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

lemma *inj-build*: *inj build*
by (*metis injI match-build option.inject*)

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* $build_X \cap$ *range* $build_Y = \{\}$)

lemma *prism-diff-intro*:
$(\bigwedge s_1\ s_2.\ build_X\ s_1 = build_Y\ s_2 \Longrightarrow False) \Longrightarrow X \nabla Y$
by (*auto simp add*: *prism-diff-def*)

lemma *prism-diff-irrefl*: $\neg X \nabla X$
by (*simp add*: *prism-diff-def*)

lemma *prism-diff-sym*: $X \nabla Y \Longrightarrow Y \nabla X$
by (*auto simp add*: *prism-diff-def*)

lemma *prism-diff-build*: $X \nabla Y \Longrightarrow build_X\ u \neq build_Y\ v$
by (*simp add*: *disjoint-iff-not-equal prism-diff-def*)

lemma *prism-diff-build-match*: $[\![$ *wb-prism X*; $X \nabla Y$ $]\!] \Longrightarrow match_X\ (build_Y\ v) = None$
using *UNIV-I wb-prism.range-build* **by** (*fastforce simp add*: *prism-diff-def*)

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* $1_\triangle$
unfolding *prism-id-def wb-prism-def* **by** *simp*

lemma *prism-id-never-diff*: $\neg\ 1_\triangle\ \nabla\ X$
  by (*simp add*: *prism-diff-def prism-id-def*)

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* = (λ *s*. *case* ($match_X$ *s*, $match_Y$ *s*) *of*
    (*Some u*, -) ⇒ *Some* (*Inl u*) |
    (*None*, *Some v*) ⇒ *Some* (*Inr v*) |
    (*None*, *None*) ⇒ *None*),
  prism-build = (λ *v*. *case v of Inl x* ⇒ $build_X$ *x* | *Inr y* ⇒ $build_Y$ *y*) $|\!)$

lemma *prism-plus-wb* [*simp*]: ⟦ *wb-prism X*; *wb-prism Y*; $X\ \nabla\ Y$ ⟧ ⟹ *wb-prism* $(X +_\triangle Y)$
  apply (*unfold-locales*)
   apply (*auto simp add*: *prism-plus-def sum.case-eq-if option.case-eq-if prism-diff-build-match*)
  apply (*metis map-option-case map-option-eq-Some option.exhaust option.sel sum.disc*(*2*) *sum.sel*(*1*) *wb-prism.build-match-iff*)
 apply (*metis* (*no-types*, *lifting*) *isl-def not-None-eq option.case-eq-if option.sel sum.sel*(*2*) *wb-prism.build-match*)
  done

lemma *build-plus-Inl* [*simp*]: $build_{c\ +_\triangle\ d}$ (*Inl x*) = $build_c$ *x*
  by (*simp add*: *prism-plus-def*)

lemma *build-plus-Inr* [*simp*]: $build_{c\ +_\triangle\ d}$ (*Inr y*) = $build_d$ *y*
  by (*simp add*: *prism-plus-def*)

lemma *prism-diff-preserved-1* [*simp*]: ⟦ $X\ \nabla\ Y$; $X\ \nabla\ Z$ ⟧ ⟹ $X\ \nabla\ Y +_\triangle Z$
  by (*auto simp add*: *lens-defs sum.case-eq-if*)

lemma *prism-diff-preserved-2* [*simp*]: ⟦ $X\ \nabla\ Z$; $Y\ \nabla\ Z$ ⟧ ⟹ $X +_\triangle Y\ \nabla\ Z$
  by (*meson prism-diff-preserved-1 prism-diff-sym*)

The following two lemmas are useful for reasoning about prism sums

lemma *Bex-Sum-iff*: (∃ *x*∈*A*<+>*B*. *P x*) ⟷ (∃ *x*∈*A*. *P* (*Inl x*)) ∨ (∃ *y*∈*B*. *P* (*Inr y*))
  by (*auto*)

lemma *Ball-Sum-iff*: (∀ *x*∈*A*<+>*B*. *P x*) ⟷ (∀ *x*∈*A*. *P* (*Inl x*)) ∧ (∀ *y*∈*B*. *P* (*Inr y*))
  by (*auto*)

11.5 Instances

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

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

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

done

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

lemma *prism-suml-indep-sumr* [*simp*]: $Inl_\triangle \nabla Inr_\triangle$
by (*auto simp add*: *lens-defs*)

lemma *prism-sum-plus*: $Inl_\triangle +_\triangle Inr_\triangle = 1_\triangle$
unfolding *lens-defs prism-plus-def* **by** (*auto simp add*: *Inr-Inl-False sum.case-eq-if*)

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*)
by (*simp add*: *mwb-lens-axioms-def mwb-lens-def weak-lens-def prism-lens-def*)

lemma *get-prism-lens*: $get_{prism\text{-}lens\ X} = the \circ match_X$
by (*simp add*: *prism-lens-def fun-eq-iff*)

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

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

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

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

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

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

lemma *pbij-lens-implies-wb-prism*:

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

ML-file ‹*Prism-Lib.ML*›

end

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* ⇒ *'d*) ⇒ (*'d* ⇒ *bool*) ⇒ (*'d* ⇒ *'a*) ⇒ (*'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*)
by (*unfold-locales*, *simp-all add*: *lens-defs assms*)
(*metis assms*(*3*) *option.distinct*(*1*) *option.sel*)

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

ML-file *Channel-Type.ML*

end

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

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

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

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

context begin

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

end

bundle *lens-state-notation* **begin**
notation *zoom* (**infixr** ▷ *80*)

```
  notation modify (infix %= 80)
  notation assign (infix .= 80)
  notation Lens-State.add (infix += 80)
  notation Lens-State.sub (infix −= 80)
  notation Lens-State.mul (infix ∗= 80)
  notation Lens-State.inc (- ++)
  notation Lens-State.dec (- −−)
end

context includes lens-state-notation begin

lemma zoom-comp1: l1 ▷ l2 ▷ s = (l2 ;L l1) ▷ s
unfolding zoom-def lens-comp-def
by (auto split: prod.splits)

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

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

end

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

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

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

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

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

end
```

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

References

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

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

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

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

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

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

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

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

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