Abstract

These components add further fundamental order and lattice-theoretic
concepts and properties to Isabelle’s libraries. They follow by and large
the introductory sections of the Compendium of Continuous Lattices,
covering directed and filtered sets, down-closed and up-closed sets,
ideals and filters, Galois connections, closure and co-closure operators.
Some emphasis is on duality and morphisms between structures—as
in the Compendium. To this end, three ad-hoc approaches to duality
are compared.

Contents

1 Introductory Remarks 2

2 Sup-Lattices and Other Simplifications 4

3 Ad-Hoc Duality for Orderings and Lattices 7

4 Properties of Orderings and Lattices 14
   4.1 Basic Definitions for Orderings and Lattices . . . . . . . . . . 14
   4.2 Properties of Orderings . . . . . . . . . . . . . . . . . . . . . 20
   4.3 Dual Properties of Orderings . . . . . . . . . . . . . . . . . 24
   4.4 Shunting Laws . . . . . . . . . . . . . . . . . . . . . . . . . . 27
   4.5 Properties of Complete Lattices . . . . . . . . . . . . . . . . 28
   4.6 Sup- and Inf-Preservation . . . . . . . . . . . . . . . . . . . . 29
   4.7 Alternative Definitions for Complete Boolean Algebras . . . . 36
   4.8 Atomic Boolean Algebras . . . . . . . . . . . . . . . . . . . . 38

5 Representation Theorems for Orderings and Lattices 40
   5.1 Representation of Posets . . . . . . . . . . . . . . . . . . . . . 40
   5.2 Stone’s Theorem in the Presence of Atoms . . . . . . . . . . . 47
1 Introductory Remarks

Basic order- and lattice-theoretic concepts are well covered in Isabelle’s libraries, and widely used. More advanced components are spread out over various sites (e.g. [11, 9, 8, 1, 4, 2]).

This formalisation takes the initial steps towards a modern structural approach to orderings and lattices, as for instance in denotational semantics of programs, algebraic logic or pointfree topology. Building on the components for orderings and lattices in Isabelle’s main libraries, it follows the classical textbook *A Compendium of Continuous Lattices* [3] and, to a lesser extent, Johnstone’s monograph on *Stone Spaces* [5]. By integrating material from other sources and extending it, a formalisation of undergraduate-level textbook material on orderings and lattices might eventually emerge.

In the textbooks mentioned, concepts such as dualities, isomorphisms between structures and relationships between categories are emphasised. These are essential to modern mathematics beyond orderings and lattices; their formalisation with interactive theorem provers is therefore of wider interest. Nevertheless such notions seem rather underexplored with Isabelle, and I am not aware of a standard way of modelling and using them. The present setting is perhaps the simplest one in which their formalisation can be studied.
These components use Isabelle’s axiomatic approach without carrier sets. This is certainly a limitation, but it can be taken quite far. Yet well-known facts such as Tarski’s theorem—the set of fixpoints of an isotone endofunction on a complete lattice forms a complete lattice—seem hard to formalise with it (at least without using recent experimental extensions [7]).

Firstly, leaner versions of complete lattices are introduced: Sup-lattices (and their dual Inf-lattices), in which only Sups (or Infs) are axiomatised, whereas the remaining operators, which are axiomatised in the standard Isabelle class for complete lattices, are defined explicitly. This not only reduces of proof obligations in instantiation or interpretation proofs, it also helps in constructions where only suprema are represented faithfully (e.g., using morphisms that preserve sups, but not infs, or vice versa). At the moment, Sup-lattices remain rather loosely integrated into Isabelle’s lattice hierarchy; a tighter one seems rather delicate.

Order and lattice duality is modelled, rather ad hoc, within a type class that can be added to those for orderings and lattices. Duality thus becomes a functor that reverses the order and maps Sups to Infs and vice versa, as expected. It also maps order-preserving functions to order-preserving functions, Sup-preserving to Inf-preserving ones and vice versa. This simple approach has not yet been optimised for automatic generation of dual statements (which seems hard to achieve anyway). It works quite well on simple examples.

The class-based approach to duality is contrasted by an implicit, locale-based one (which is quite standard in Isabelle), and Wenzel’s data-type-based one [11]. Wenzel’s approach generates many properties of the duality functor automatically from Isabelle’s data type package. However, duality is not involutive, and this limits the dualisation of theorems quite severely. The local-based approach dualises theorems within the context of a type class or locale highly automatically. But, unlike the present approach, it is limited to such contexts. Yet another approach to duality has been taken in HOL-Algebra [2], but it is essentially based on set theory and therefore beyond the reach of simple axiomatic type classes.

The components presented also cover fundamental concepts such as directed and filtered sets, down-closed and up-closed sets, ideals and filters, notions of sup-closure and inf-closure, sup-preservation and inf-preservation, properties of adjunctions (or Galois connections) between orderings and (complete) lattices, fusion theorems for least and greatest fixpoints, and basic properties of closure and co-closure (kernel) operations, following the Compendium (most of these concepts come as dual pairs!). As in this monograph, emphasis lies on categorical aspects, but no formal category theory is used. In addition, some simple representation theorems have been formalised, including Stone’s theorem for atomic boolean algebras (objects only). The non-atomic case seems possible, but is left for future work. Dealing with opposite maps properly, which is essential for dualities, remains an
issue.

Finally, in Isabelle’s main libraries, complete distributive lattices and complete boolean algebras are currently based on a very strong distributivity law, which makes these structures completely distributive and is basically an Axiom of Choice. While powerset algebras satisfy this law, other applications, for instance in topology require different axiomatisations. Complete boolean algebras, in particular, are usually defined as complete lattices which are also boolean algebras. Hence only a finite distributivity law holds. Weaker distributivity laws are also essential for axiomatising complete Heyting algebras (aka frames or locales), which are relevant for point-free topology [5].

Many questions remain, in particular on tighter integrations of duality and reasoning up to isomorphism with Isabelle and beyond. In its present form, duality is often not picked up in the proofs of more complex statements. Some statements from the Compendium and Johnstone’s book had to be ignored due to the absence of carrier sets in Isabelle’s standard components for orderings and lattices. Whether Kuncar and Popescu’s new types-to-sets translation [7] provides a satisfactory solution remains to be seen.

2 Sup-Lattices and Other Simplifications

theory Sup-Lattice
  imports Main
  HOL-Library.Lattice-Syntax

begin

Some definitions for orderings and lattices in Isabelle could be simpler. The strict order in in ord could be defined instead of being axiomatised. The function mono could have been defined on ord and not on order—even on a general (di)graph it serves as a morphism. In complete lattices, the supremum—and dually the infimum—suffices to define the other operations (in the Isabelle/HOL-definition infimum, binary supremum and infimum, bottom and top element are axiomatised). This not only increases the number of proof obligations in subclass or sublocale statements, instantiations or interpretations, it also complicates situations where suprema are presented faithfully, e.g. mapped onto suprema in some subalgebra, whereas infima in the subalgebra are different from those in the super-structure.

It would be even nicer to use a class less-eq which dispenses with the strict order symbol in ord. Then one would not have to redefine this symbol in all instantiations or interpretations. At least, it does not carry any proof obligations.

context ord
begin
ub-set yields the set of all upper bounds of a set; lb-set the set of all lower bounds.

**definition ub-set :: 'a set ⇒ 'a set where**

\[ \text{ub-set } X = \{ y. \forall x \in X. x \leq y \} \]

**definition lb-set :: 'a set ⇒ 'a set where**

\[ \text{lb-set } X = \{ y. \forall x \in X. y \leq x \} \]

end

**definition ord-pres :: (a::ord ⇒ b::ord) ⇒ bool where**

\[ \text{ord-pres } f = (\forall x y. x \leq y \implies f x \leq f y) \]

**lemma ord-pres-mono:**

| fixes f :: a::order ⇒ b::order |
| shows mono f = ord-pres f |
| by (simp add: mono-def ord-pres-def) |

**class preorder-lean = ord +**

| assumes preorder-refl: x ≤ x |
| and preorder-trans: x ≤ y ⇒ y ≤ z ⇒ x ≤ z |

begin

**definition le :: 'a ⇒ 'a ⇒ bool where**

\[ \text{le } x y = (x \leq y \land \neg (x \geq y)) \]

end

**sublocale preorder-lean ⊆ prel: preorder (≤) le**

| by (unfold-locales, auto simp add: le-def preorder-refl preorder-trans) |

**class order-lean = preorder-lean +**

| assumes order-antisym: x ≤ y ⇒ x ≥ y ⇒ x = y |

**sublocale order-lean ⊆ posl: order (≤) le**

| by (unfold-locales, simp add: order-antisym) |

**class Sup-lattice = order-lean + Sup +**

| assumes Sups-upper: x ∈ X ⇒ x ≤ \bigcup X |
| and Sups-least: (\forall x. x ∈ X ⇒ x ≤ z) ⇒ \bigcup X ≤ z |

begin

**definition Infs :: 'a set ⇒ 'a where**

\[ \text{Infs } X = \bigcup \{ y. \forall x \in X. y \leq x \} \]
definition supers :: 'a ⇒ 'a ⇒ 'a where
  supers x y = \{x, y\}

definition infs :: 'a ⇒ 'a ⇒ 'a where
  infs x y = ⨆\{x, y\}

definition bots :: 'a where
  bots = \{\}

definition tops :: 'a where
  tops = ⨆\{\}

lemma Infs-prop: Infs = Sup ◦ \{\}
  unfolding fun-eq-iff by (simp add: Infs-def prel.ub-set-def)

end

class Inf-lattice = order-lean + Inf +
  assumes Infi-lower: x ∈ X ⇒ \cap X ≤ x
  and Infi-greatest: (\forall x ∈ X. x ≤ z) ⇒ z ≤ \cap X

begin

definition Supi :: 'a set ⇒ 'a where
  Supi X = \{y. \forall x ∈ X. x ≤ y\}

definition supi :: 'a ⇒ 'a ⇒ 'a where
  supi x y = Supi\{x, y\}

definition infi :: 'a ⇒ 'a ⇒ 'a where
  infi x y = \{x, y\}

definition boti :: 'a where
  boti = Supi\{\}

definition topi :: 'a where
  topi = \cap\{\}

lemma Supi-prop: Supi = Inf ◦ \{\}
  unfolding fun-eq-iff by (simp add: Supi-def prel.ub-set-def)

end

sublocale Inf-lattice ⊆ ldual: Sup-lattice Inf (≥)
  rewrites ldual.Infs = Supi
  and ldual.infs = supi
  and ldual.sups = infi
  and ldual.tops = boti
  and ldual.bots = topi
proof –

show class.Sup-lattice Inf (≥)
  by (unfold-locales, simp-all add: Infi-lower Infi-greatest preorder-trans)
then interpret ldual: Sup-lattice Inf (≥).

show a: ldual.Infs = Supi
  unfolding fun-eq-iff by (simp add: ldual.Infs-def Supi-def)
show ldual.infs = supi
  unfolding fun-eq-iff by (simp add: a ldual.infs-def supi-def)
show ldual.sups = infi
  unfolding fun-eq-iff by (simp add: ldual.sups-def infi-def)
show ldual.tops = boti
  by (simp add: a ldual.tops-def boti-def)
show ldual.bots = topi
  by (simp add: ldual.bots-def topi-def)
qed

sublocale Sup-lattice ⊆ supclat: complete-lattice Infs Sup-class Sup infs (≤) le supers bots tops
  apply unfold-locales
  unfolding Infs-def infs-def supers-def bots-def tops-def
  by (simp-all, auto intro: Sups-least, simp-all add: Sups-upper)

sublocale Inf-lattice ⊆ infclat: complete-lattice Inf-class Inf Supi infi (≤) le supi boti topi
end

3 Ad-Hoc Duality for Orderings and Lattices

theory Order-Duality
  imports Sup-Lattice

begin

This component presents an "explicit" formalisation of order and lattice duality. It augments the data type based one used by Wenzel in his lattice components [11], and complements the "implicit" formalisation given by locales. It uses a functor dual, supplied within a type class, which is simply a bijection (isomorphism) between types, with the constraint that the dual of a dual object is the original object. In Wenzel’s formalisation, by contrast, dual is a bijection, but not idempotent or involutive. In the past, Preoteasa has used a similar approach with Isabelle [8].

Duality is such a fundamental concept in order and lattice theory that it probably deserves to be included in the type classes for these objects, as in this section.
class dual =
  fixes dual :: 'a ⇒ 'a (doll)
  assumes inj-dual: inj θ
  and invol-dual [simp]: θ ∘ θ = id

This type class allows one to define a type dual. It is actually a dependent

type for which dual can be instantiated.

typedef (overloaded) 'a dual = range (dual::'a::dual ⇒ 'a)
  by fastforce

setup-lifting type-definition-dual

At the moment I have no use for this type.

context dual
begin

lemma invol-dual-var [simp]: θ (θ x) = x
  by (simp add: pointfree-idE)

lemma surj-dual: surj θ
  unfolding surj-def by (metis invol-dual-var)

lemma bij-dual: bij θ
  by (simp add: bij-def inj-dual surj-dual)

lemma inj-dual-iff: (θ x = θ y) = (x = y)
  by (meson inj-dual injD)

lemma dual-iff: (θ x = y) = (x = θ y)
  by auto

lemma the-inv-dual: the-inv θ = θ
  by (metis comp-apply id-def invol-dual-var inj-dual surj-fun-eq the-inv-f-o-f-id)

end

In boolean algebras, duality is of course De Morgan duality and can be

expressed within the language.

sublocale boolean-algebra ⊆ ba-dual: dual uminus
  by (unfold-locales, simp-all add: inj-def)

definition map-dual:: ('a ⇒ 'b) ⇒ 'a::dual ⇒ 'b::dual (θF) where
  θF f = θ ∘ f ∘ θ

lemma map-dual-func1: θF (f ∘ g) = θF f ∘ θF g
  by (metis (no-types, lifting) comp-assoc comp-id invol-dual map-dual-def)

lemma map-dual-func2 [simp]: θF id = id
by (simp add: map-dual-def)

**lemma** map-dual-nat-iso: $\partial F \circ \partial = \partial \circ \text{id } f$
by (simp add: comp-assoc map-dual-def)

**lemma** map-dual-invol [simp]: $\partial F \circ \partial F = \text{id}$
unfolding map-dual-def comp-def fun-eq-iff by simp

Thus map-dual is naturally isomorphic to the identify functor: The function dual is a natural transformation between map-dual and the identity functor, and, because it has a two-sided inverse — itself, it is a natural isomorphism.

The generic function set-dual provides another natural transformation (see below). Before introducing it, we introduce useful notation for a widely used function.

**abbreviation** $\eta \equiv (\lambda x. \{ x \})$

**lemma** eta-inj: inj $\eta$
by simp

definition set-dual = $\eta \circ \partial$

**lemma** set-dual-prop: set-dual ($\partial x$) = $\{ x \}$
by (metis comp-apply dual-iff set-dual-def)

The next four lemmas show that (functional) image and preimage are functors (on functions). This does not really belong here, but it is useful for what follows. The interaction between duality and (pre)images is needed in applications.

**lemma** image-func1: ($\partial$) ($f \circ g$) = ($\partial$) $f$ ($\partial$) $g$
unfolding fun-eq-iff by (simp add: image-comp)

**lemma** image-func2: ($\partial$) $\text{id}$ = $\text{id}$
by simp

**lemma** vimage-func1: ($\partial$) ($f \circ g$) = ($\partial$) $g$ ($\partial$) ($\partial$) $f$
unfolding fun-eq-iff by (simp add: vimage-comp)

**lemma** vimage-func2: ($\partial$) $\text{id}$ = $\text{id}$
by simp

**lemma** iso-image: mono ($\partial$) $f$
by (simp add: image-mono monoI)

**lemma** iso-preimage: mono ($\partial$) $f$
by (simp add: monoI vimage-mono)

context dual
begin

lemma image-dual [simp]: (′) ∂ o (′) ∂ = id
  by (metis image-func1 image-func2 invol-dual)

lemma vimage-dual [simp]: (−′) ∂ o (−′) ∂ = id
  by (simp add: set.comp)

end

The following natural transformation between the powerset functor (image) and the identity functor is well known.

lemma power-set-func-nat-trans: η ◦ id f = (′) f ◦ η
  unfolding fun-eq-iff comp-def by simp

As an instance, set-dual is a natural transformation with built-in type coercion.

lemma dual-singleton: (′) ∂ o η = η o ∂
  by auto

lemma finite-dual [simp]: finite ◦ (′) ∂ = finite
  unfolding fun-eq-iff comp-def using inj-dual finite-vimageI inj-vimage-image-eq
  by fastforce

lemma finite-dual-var [simp]: finite (′) X = finite X
  by (metis comp-def finite-dual)

lemma subset-dual: (X = (′) Y) = (′) X = Y
  by (metis image-dual pointfree-idE)

lemma subset-dual1: (X ⊆ Y) = (′) X ⊆ (′) Y
  by (simp add: inj-dual inj-image-subset-iff)

lemma dual-empty [simp]: (′) { } = { }
  by simp

lemma dual-UNIV [simp]: (′) UNIV = UNIV
  by (simp add: surj-dual)

lemma fun-dual1: (f = g o ∂) = (f o ∂ = g)
  by (metis comp-assoc comp-id invol-dual)

lemma fun-dual2: (f = ∂ o g) = (∂ o f = g)
  by (metis comp-assoc fun.map-id invol-dual)

lemma fun-dual3: (f = g o (′) ∂) = (f o (′) ∂ = g)
  by (metis comp-id image-dual o-assoc)

lemma fun-dual4: (f = (′) ∂ o g) = ((′) ∂ o f = g)
by (metis comp-assoc id-comp image-dual)

lemma fun-dual5: \( (f = \partial \circ g \circ \partial) = (\partial \circ f \circ \partial = g) \)
by (metis comp-assoc fun-dual1 fun-dual2)

lemma fun-dual6: \( (f = (') \partial \circ g \circ (') \partial) = ((') \partial \circ f \circ (') \partial = g) \)
by (simp add: comp-assoc fun-dual3 fun-dual4)

lemma fun-dual7: \( (f = \partial \circ g \circ (') \partial) = (\partial \circ f \circ (') \partial = g) \)
by (simp add: comp-assoc fun-dual2 fun-dual3)

lemma fun-dual8: \( (f = (') \partial \circ g \circ \partial) = ((') \partial \circ f \circ \partial = g) \)
by (simp add: comp-assoc fun-dual1 fun-dual4)

lemma map-dual-dual: \( (\partial F f = g) = (\partial F g = f) \)
by (metis map-dual-invol pointfree-idE)

The next facts show incrementally that the dual of a complete lattice is a complete lattice.

class ord-with-dual = dual + ord +
  assumes ord-dual: \( x \leq y \Longrightarrow \partial y \leq \partial x \)
begin

lemma dual-dual-ord: \( (\partial x \leq \partial y) = (y \leq x) \)
by (metis dual-iff ord-dual)
end

class preorder-with-dual = ord-with-dual + preorder

begin

lemma ord-pres-dual:
  fixes f :: 'a::ord-with-dual ⇒ 'b::ord-with-dual
  shows ord-pres f ⇒ ord-pres (\partial F f)
  by (simp add: dual-ord-pres-def)

lemma map-dual-anti: \( (f::'a::ord-with-dual ⇒ 'b::ord-with-dual) \leq g \Longrightarrow \partial F g \leq \partial F f \)
by (simp add: le-fun-def map-dual-def)

class preorder-with-dual = ord-with-dual + preorder

begin

lemma less-dual-def-var: \( (\partial y < \partial x) = (x < y) \)
by (simp add: dual-ord-anti le-less)

end

class order-with-dual = preorder-with-dual + order
lemma iso-map-dual:
  fixes f :: 'a::order-with-dual ⇒ 'b::order-with-dual
  shows mono f =⇒ mono (∂F f)
  by (simp add: ord-pres-dual ord-pres-mono)

class lattice-with-dual = lattice + dual +
  assumes sup-dual-def: ∂ (x ⊔ y) = ∂ x ⊓ ∂ y

begin
subclass order-with-dual
  by (unfold-locales, metis inf.absorb-iff sup.absorb1 sup-commute sup-dual-def)

lemma inf-dual: ∂ (x ⊓ y) = ∂ x ⊔ ∂ y
  by (metis invol-dual-var sup-dual-def)

lemma inf-to-sup: x ⊓ y = ∂ (∂ x ⊔ ∂ y)
  using inf-dual dual-iff by fastforce

lemma sup-to-inf: x ⊔ y = ∂ (∂ x ⊓ ∂ y)
  by (simp add: inf-dual)

end

class bounded-lattice-with-dual = lattice-with-dual + bounded-lattice

begin
lemma bot-dual: ∂ ⊥ = ⊤
  by (metis dual-dual-ord dual-iff le-bot top-greatest)

lemma top-dual: ∂ ⊤ = ⊥
  using bot-dual dual-iff by force

end

class boolean-algebra-with-dual = lattice-with-dual + boolean-algebra

sublocale boolean-algebra ⊆ badual: boolean-algebra-with-dual - - - - - - - - uminus
  by unfold-locales simp-all

class Sup-lattice-with-dual = Sup-lattice + dual +
  assumes Sups-dual-def: ∂ o Sup = Infs o (∁) ∂

class Inf-lattice-with-dual = Inf-lattice + dual +
  assumes Sups-dual-def: ∂ o Supi = Inf o (∁) ∂

class complete-lattice-with-dual = complete-lattice + dual +
  assumes Sups-dual-def: ∂ o Sup = Inf o (∁) ∂
sublocale Sup-lattice-with-dual \subseteq sclatd: complete-lattice-with-dual Infs Sup infs \leq le saps bots tops \partial
by (unfold-locales, simp add: Sups-dual-def)

sublocale Inf-lattice-with-dual \subseteq iclatd: complete-lattice-with-dual Inf Supi infi \leq le sapi boti topi \partial
by (unfold-locales, simp add: Sups-dual-def)

class complete-lattice-with-dual

begin

lemma Inf-dual: \partial \circ \Inf = \Sup \circ (\cdot) \partial
by (metis comp-assoc comp-id fun.map-id Sups-dual-def image-dual invol-dual)

lemma Inf-dual-var: \partial (\bigcap X) = \bigcup (\partial \circ X)
using comp-eq-dest Inf-dual by fastforce

lemma Inf-to-Sup: \Inf = \partial \circ \Sup \circ (\cdot) \partial
by (auto simp add: Sups-dual-def image-comp)

lemma Inf-to-Sup-var: \bigcap X = \partial (\bigcap (\partial \circ X))
using Inf-dual-var dual-iff by fastforce

lemma Sup-to-Inf: \Sup = \partial \circ \Inf \circ (\cdot) \partial
by (auto simp add: Inf-dual image-comp)

lemma Sup-to-Inf-var: \bigcup X = \partial (\bigcup (\partial \circ X))
using Sup-to-Inf by force

lemma Sup-dual-def-var: \partial (\bigcup X) = \bigcap (\partial \circ X)
using comp-eq-dest Sups-dual-def by fastforce

lemma bot-dual-def: \partial \top = \bot
by (smt Inf-UNIV Sup-UNIV Sups-dual-def surj-dual o-eq-dest)

lemma top-dual-def: \partial \bot = \top
using bot-dual-def dual-iff by blast

lemma inf-dual2: \partial (x \cap y) = \partial x \cup \partial y
by (smt comp-eq-elim Inf-dual Inf-empty Inf-insert SUP-insert inf-top.right-neutral)

lemma sup-dual: \partial (x \cup y) = \partial x \cap \partial y
by (metis inf-dual2 dual-iff)

subclass lattice-with-dual
by (unfold-locales, auto simp: inf-dual sup-dual)

subclass bounded-lattice-with-dual


4 Properties of Orderings and Lattices

theory Order-Lattice-Props
  imports Order-Duality
begin

4.1 Basic Definitions for Orderings and Lattices

The first definition is for order morphisms — isotone (order-preserving, monotone) functions. An order isomorphism is an order-preserving bijection. This should be defined in the class ord, but mono requires order.

definition ord-homset :: ('a::order ⇒ 'b::order) set where
ord-homset = {f::'a::order ⇒ 'b::order. mono f}

definition ord-embed :: ('a::order ⇒ 'b::order) ⇒ bool where
ord-embed f = (∀x y. f x ≤ f y ←→ x ≤ y)

definition ord-iso :: ('a::order ⇒ 'b::order) ⇒ bool where
ord-iso = bij ∩ mono ∩ (mono ◦ the-inv)

lemma ord-embed-alt: ord-embed f = (mono f ∧ (∀x y. f x ≤ f y → x ≤ y))
  using mono-def ord-embed-def by auto

lemma ord-embed-homset: ord-embed f ∈ ord-homset
  by (simp add: mono-def ord-embed-def ord-homset-def)

lemma ord-embed-inj: ord-embed f ⇒ inj f
  unfolding ord-embed-def inj-def by (simp add: eq-iff)

lemma ord-iso-ord-embed: ord-iso f ⇒ ord-embed f
  unfolding ord-iso-def ord-embed-def bij-def inj-def mono-def
  by (clarsimp, metis inj-def the-inv-f-f)

lemma ord-iso-alt: ord-iso f = (ord-embed f ∧ surj f)
  unfolding ord-iso-def ord-embed-def surj-def bij-def inj-def mono-def
  apply safe
  by simp-all (metis eq-iff inj-def the-inv-f-f)+

lemma ord-iso-the-inv: ord-iso f ⇒ mono (the-inv f)
  by (simp add: ord-iso-def)

lemma ord-iso-inv1: ord-iso f ⇒ (the-inv f) ◦ f = id
using ord-embed-inj ord-iso-ord-embed the-inv-into-f-f by fastforce

lemma ord-iso-inv2: ord-iso f ⇒ f ◦ (the-inv f) = id
  using f-the-inv-into-f ord-embed-inj ord-iso-alt by fastforce

typedef (overloaded) ('a,'b) ord-homset = ord-homset::('a::order ⇒ 'b::order)
  set
  by (force simp: ord-homset-def mono-def)

setup-lifting type-definition-ord-homset

The next definition is for the set of fixpoints of a given function. It is important in the context of orders, for instance for proving Tarski’s fixpoint theorem, but does not really belong here.

definition Fix :: ('a ⇒ 'a) ⇒ 'a set where
  Fix f = {x. f x = x}

lemma retraction-prop: f ◦ f = f ⇒ f x = x ←→ x ∈ range f
  by (metis comp-apply f-inv-into-f rangeI)

lemma retraction-prop-fix: f ◦ f = f ⇒ range f = Fix f
  unfolding Fix-def using retraction-prop by fastforce

lemma Fix-map-dual: Fix ◦ ∂F = ('⁻') ∂ ◦ Fix
  unfolding Fix-def map-dual-def comp-def fun-eq-iff
  by (smt Collect-cong invol-dual pointfree-idE setcompr-eq-image)

lemma Fix-map-dual-var: Fix (∂F f) = ∂ ('⁻') (Fix f)
  by (metis Fix-map-dual o-def)

lemma gfp-dual: (∂::'a::complete-lattice-with-dual ⇒ 'a) ◦ gfp = lfp ◦ ∂F
  proof -
    { fix f:: 'a ⇒ 'a
      have ∂ (gfp f) = ∂ (⨆ {u. u ≤ f u})
        by (simp add: gfp-def)
      also have ... = ⨆ {∂ '⁻' {u. u ≤ f u}}
        by (simp add: Sup-dual-def-var)
      also have ... = ⨆ {∂ u | u. u ≤ f u}
        by (simp add: setcompr-eq-image)
      also have ... = ⨆ {u | u. (∂F f) u ≤ u}
        by (metis (no-types, hide-lams) dual-dual-ord dual-iff map-dual-def o-def)
      finally have ∂ (gfp f) = lfp (∂F f)
        by (metis lfp-def)
      thus ?thesis
        by auto
    } thus qed

lemma gfp-dual-var:
  fixes f :: 'a::complete-lattice-with-dual ⇒ 'a
shows $\partial (\text{gfp} \ f) = \text{lfp} (\partial_F \ f)$
using comp-eq-elim gfp-dual by blast

lemma gfp-to-lfp: $\text{gfp} = (\partial :: 'a::complete-lattice-with-dual \Rightarrow 'a) \circ \text{lfp} \circ \partial_F$
by (simp add: comp-assoc fun-dual2 gfp-dual)

lemma gfp-to-lfp-var:
fixes $f :: 'a::complete-lattice-with-dual \Rightarrow 'a$
shows $\text{gfp} \ f = \partial (\text{lfp} (\partial_F \ f))$
by (metis gfp-dual-var invol-dual-var)

lemma lfp-dual: $(\partial :: 'a::complete-lattice-with-dual \Rightarrow 'a) \circ \text{lfp} = \text{gfp} \circ \partial_F$
by (simp add: comp-assoc gfp-to-lfp map-dual-invol)

lemma lfp-dual-var:
fixes $f :: 'a::complete-lattice-with-dual \Rightarrow 'a$
shows $\partial (\text{lfp} \ f) = \text{gfp} (\text{map-dual} \ f)$
using comp-eq-dest-lhs lfp-dual by fastforce

lemma lfp-to-gfp: $\text{lfp} = (\partial :: 'a::complete-lattice-with-dual \Rightarrow 'a) \circ \text{gfp} \circ \partial_F$
by (simp add: comp-assoc gfp-to-lfp map-dual-invol)

lemma lfp-to-gfp-var:
fixes $f :: 'a::complete-lattice-with-dual \Rightarrow 'a$
shows $\text{lfp} \ f = \partial (\text{gfp} (\partial_F \ f))$
by (metis invol-dual-var lfp-dual-var)

lemma lfp-in-Fix:
fixes $f :: 'a::complete-lattice \Rightarrow 'a$
shows $\text{mono} \ f \Rightarrow \text{lfp} \ f \in \text{Fix} \ f$
by (metis (mono-tags, lifting) Fix-def lfp-unfold mem-Collect-eq)

lemma gfp-in-Fix:
fixes $f :: 'a::complete-lattice \Rightarrow 'a$
shows $\text{mono} \ f \Rightarrow \text{gfp} \ f \in \text{Fix} \ f$
by (metis (mono-tags, lifting) Fix-def gfp-unfold mem-Collect-eq)

lemma nonempty-Fix:
fixes $f :: 'a::complete-lattice \Rightarrow 'a$
shows $\text{mono} \ f \Rightarrow \text{Fix} \ f \neq \{\}$
using lfp-in-Fix by fastforce

Next the minimal and maximal elements of an ordering are defined.

class ord
begin

definition min-set :: 'a set \Rightarrow 'a set where
min-set $X = \{y \in X. \forall x \in X. x \leq y \rightarrow x = y\}$

end
definition max-set :: 'a set ⇒ 'a set where
max-set X = {x ∈ X. ∀ y ∈ X. x ≤ y → x = y}
end

context ord-with-dual
begin

lemma min-max-set-dual: (∧) ∘ min-set = max-set ∘ (∨)
  unfolding max-set-def min-set-def fun-eq-iff comp-def
  apply safe
  using dual-dual-ord inj-dual-iff by auto

lemma min-max-set-dual-var: (′) ∘ (min-set X) = max-set (′ X)
  using comp-eq-dest min-max-set-dual by fastforce

lemma max-min-set-dual: (′) ∘ max-set = min-set ∘ (′)
  by (metis (no-types, hide-lams) comp-id fun.map-comp id-comp image-dual min-max-set-dual)

lemma min-to-max-set: min-set = (′) ∘ max-set ∘ (′)
  by (metis comp-id image-dual max-min-set-dual o-assoc)

lemma max-min-set-dual-var: (′) ∘ (max-set X) = min-set (′ X)
  using comp-eq-dest max-min-set-dual by fastforce

lemma min-to-max-set-var: min-set X = (′) ∘ (max-set (′ X))
  by (simp add: max-min-set-dual-var pointfree-idE)

end

Next, directed and filtered sets, upsets, downsets, filters and ideals in posets
are defined.

context ord
begin

definition directed :: 'a set ⇒ bool where
directed X = (∀ Y. finite Y ∧ Y ⊆ X ⇒ (∃ x ∈ X. ∀ y ∈ Y. y ≤ x))

definition filtered :: 'a set ⇒ bool where
filtered X = (∀ Y. finite Y ∧ Y ⊆ X ⇒ (∃ x ∈ X. ∀ y ∈ Y. x ≤ y))

definition downset-set :: 'a set ⇒ 'a set (⇓) where
⇓X = {y. ∃ x ∈ X. y ≤ x}

definition upset-set :: 'a set ⇒ 'a set (⇑) where
⇑X = {y. ∃ x ∈ X. x ≤ y}

definition downset :: 'a ⇒ 'a set (⇓) where

\[ \downarrow = \downarrow \circ \eta \]

**definition upset :: 'a ⇒ 'a set (↑) where**
\[ ↑ = \uparrow \circ \eta \]

**definition downsets :: 'a set where**
\[ \text{downsets} = \text{Fix } \downarrow \]

**definition upsets :: 'a set where**
\[ \text{upsets} = \text{Fix } \uparrow \]

**definition downclosed-set \(X\) = \((X \in \text{downsets})\)**

**definition upclosed-set \(X\) = \((X \in \text{upsets})\)**

**definition ideals :: 'a set set where**
\[ \text{ideals} = \{X. \ X \neq \{\} \land \text{downclosed-set } X \land \text{directed } X\} \]

**definition filters :: 'a set set where**
\[ \text{filters} = \{X. \ X \neq \{\} \land \text{upclosed-set } X \land \text{filtered } X\} \]

**abbreviation** idealp \(X\) \(\equiv\) \(X \in \text{ideals}\)

**abbreviation** filterp \(X\) \(\equiv\) \(X \in \text{filters}\)

**end**

These notions are pair-wise dual.

Filtered and directed sets are dual.

**context ord-with-dual**

**begin**

**lemma** filtered-directed-dual: \(\text{filtered} \circ (\uparrow) \partial = \text{directed}\)

**unfolding** filtered-def directed-def fun-eq-iff comp-def

**apply** clarsimp

**apply** safe

**apply** (meson finite-imageI imageI image-mono dual-dual-ord)

**by** (smt finite-subset-image imageE ord-dual)

**lemma** directed-filtered-dual: \(\text{directed} \circ (\downarrow) \partial = \text{filtered}\)

**using** filtered-directed-dual** by** (metis comp-id image-dual o-assoc)

**lemma** filtered-to-directed: \(\text{filtered } X = \text{directed } (\partial \ ' X)\)

**by** (metis comp-apply directed-filtered-dual)

Upsets and downsets are dual.

**lemma** downset-set-upset-set-dual: \((\downarrow) \partial \circ \downarrow = \uparrow \circ (\uparrow) \partial\)

**unfolding** downset-set-def upset-set-def fun-eq-iff comp-def
apply safe
apply (meson image-eqI ord-dual)
by (clarsimp, metis (mono-tags, lifting) dual-iff image-iff mem-Collect-eq ord-dual)

lemma upset-set-downset-set-dual: (') \partial \circ \uparrow = \downarrow \circ (') \partial
using downset-set-upset-set-dual by (metis (no-types, hide-lams) comp-id id-comp image-dual o-assoc)

lemma upset-set-to-downset-set: \uparrow = (') \partial \circ \downarrow \circ (') \partial
by (simp add: comp-assoc downset-set-upset-set-dual)

lemma upset-set-to-downset-set2: \uparrow X = (') \partial \circ \downarrow (') \partial \ X)
by (simp add: upset-set-to-downset-set)

lemma downset-upset-dual: (') \partial \circ \down = \uparrow \circ \partial
using downset-def upset-def upset-set-to-downset-set by fastforce

lemma upset-to-downset: (') \partial \circ \up = \down \circ \partial
by (metis comp-assoc id-apply ord downset-def ord upset-def power-set-func-nat-trans upset-set-downset-set-dual)

lemma upset-to-downset2: \up X = \partial \circ \down (') \partial \ X)
by (simp add: upset-to-downset2)

lemma upsets-to-downsets: (X \in upsets) = (') \partial \circ \down (') \partial \ X)
by (simp add: downsets-upsets-dual image-comp)

Filters and ideals are dual.

lemma ideals-filters-dual: (X \in ideals) = (') \partial \ X \in filters
by (smt comp-eq-dest-lhs directed-filtered-dual image-inv-f-f mem-Collect-eq)

lemma idealp-filterp-dual: idealp = filterp \circ (') \partial
unfolding fun-eq-iff by (simp add: ideals-filters-dual)
lemma filters-to-ideals: \((X \in \text{filters}) = ((\partial^+ X) \in \text{ideals})\)
  by (simp add: ideals-filters-dual image-comp)

lemma filterp-idealp-dual: \(\text{filterp} = \text{idealp} \circ (\partial^+)\)
  unfolding fun-eq-iff by (simp add: filters-to-ideals)

end

4.2 Properties of Orderings
context ord
begin

lemma directed-nonempty: directed \(X \implies X \neq \{\}\)
  unfolding directed-def by fastforce

lemma directed-ab: directed \(X \implies (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \land y \leq z)\)
  by (meson empty-subsetI directed-def finite.emptyI finite-insert insert-subset order-refl)

lemma downset-set-prop: \(\downarrow = \text{Union} \circ (\partial) \downarrow\)
  unfolding downset-set-def downset-def fun-eq-iff by fastforce

lemma downset-set-prop-var: \(\downarrow X = (\bigcup x \in X. \downarrow x)\)
  by (simp add: downset-set-prop)

lemma downset-prop2: \(y \leq x \implies y \in \downarrow x\)
  by (simp add: downset-prop)

lemma ideals-downsets: \(X \in \text{ideals} \implies X \in \text{downsets}\)
  by (simp add: downclosed-set-def ideals-def)

lemma ideals-directed: \(X \in \text{ideals} \implies \text{directed} X\)
  by (simp add: ideals-def)

end

context preorder
begin

lemma directed-prop: \(X \neq \{\} \implies (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \land y \leq z) \implies \text{directed} X\)
  proof
  assume h1: \(X \neq \{\}\)
  and h2: \(\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \land y \leq z\)
  {fix \(Y\)
    have \(\text{finite} Y \implies Y \subseteq X \implies (\exists x \in X. \forall y \in Y. y \leq x)\)
  }

end
proof (induct rule: finite-induct)
  case empty then show \(\text{?case}\) using \(h1\) by blast
next
case (insert \(x\) \(F\)) then show \(\text{?case}\) by (metis \(h2\) insert-iff insert-subset order-trans)
qed
thus \(\text{?thesis}\) by (simp add: directed-def)
qed

lemma directed-alt: directed \(X\) = \((X \neq \{\} \land (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \land y \leq z))\)
  by (metis directed-prop directed-nonempty directed-ub)

lemma downset-set-prop-var2: \(x \in \Downarrow X \implies y \leq x \implies y \in \Downarrow X\)
  unfolding downset-set-def using order-trans by blast

lemma downclosed-set-iff: downclosed-set \(X\) = \((\forall x \in X. \forall y. y \leq x \implies y \in X)\)
  unfolding downclosed-set-def downsets-def Fix-def downset-set-def by auto

lemma downclosed-downset-set: downclosed-set \((\Downarrow X)\)
  by (simp add: downclosed-set-iff downset-set-prop-var2 downset-def)

lemma downclosed-downset: downclosed-set \((\downarrow x)\)
  by (simp add: downclosed-downset-set downset-def)

lemma downset-set-ext: \(id \leq \Downarrow\)
  unfolding le-fun-def id-def downset-set-def by auto

lemma downset-set-iso: mono \(\Downarrow\)
  unfolding mono-def downset-set-def by blast

lemma downset-set-idem [simp]: \(\Downarrow \circ \Downarrow = \Downarrow\)
  unfolding fun-eq-iff downset-set-def using order-trans by auto

lemma downset-faithful: \(\downarrow x \subseteq \downarrow y \implies x \leq y\)
  by (simp add: downset-prop subset-eq)

lemma downset-iso-iff: \((\downarrow x \subseteq \downarrow y) = (x \leq y)\)
  using atMost-iff downset-prop order-trans by blast

The following proof uses the Axiom of Choice.

lemma downset-directed-downset-var [simp]: directed \((\Downarrow X)\) = directed \(X\)
proof
  assume \(h1\): directed \(X\)
  { fix \(Y\)

21
assume \( h2 \): finite \( Y \) and \( h3 \): \( Y \subseteq \downarrow X \)

hence \( \forall y. \exists x. y \in Y \rightarrow x \in X \land y \leq x \)
  by (force simp: downset-set-def)

hence \( \exists f. \forall y. y \in Y \rightarrow f y \in X \land y \leq f y \)
  by (rule choice)

hence \( \exists f. \) finite \((f ' Y) \) \( \land f ' Y \subseteq X \land (\forall y \in Y. y \leq f y) \)
  by (metis finite-imageI h2 image-subsetI)

hence \( \exists Z. \) finite \( Z \land Z \subseteq X \land (\forall y \in Y. \exists z \in Z. y \leq z) \)
  by fastforce

hence \( \exists Z. \) finite \( Z \land Z \subseteq X \land (\forall y \in Y. \exists z \in Z. y \leq z) \land (\exists x \in X. \forall z \in Z. z \leq x) \)
  by (metis directed-def h1)

hence \( \exists x \in X. \forall y \in Y. y \leq x \)
  by (meson order-trans)

thus directed \( \downarrow X \)

unfolding directed-def downset-set-def by fastforce

next

assume directed \( \downarrow X \)

thus directed \( X \)

unfolding directed-def downset-set-def

apply clarsimp

by (smt Ball-Collect order-refl order-trans subsetCE)

qed

lemma downset-directed-downset \([simp]\): directed \( \circ \downarrow \) = directed

unfolding fun-eq-iff by simp

lemma directed-downset-ideals: directed \( \downarrow X \) = \( \uparrow X \in \text{ideals} \)

by (metis (mono-tags, lifting) Collect1 Fix-def directed-alt downset-set-idem downclosed-set-def ideals-def o-def ord.ideals-directed)

lemma downclosed-Fix: downclosed-set \( X = (\downarrow X = X) \)

by (metis (mono-tags, lifting) CollectD Fix-def downclosed-downset-set downclosed-set-def downsets-def)

end

lemma downset-iso: mono \( \downarrow 'a::order \Rightarrow 'a::order \)

by (simp add: downset-iso-iff mono-def)

lemma mono-downclosed:

fixes \( f : 'a::order \Rightarrow 'b::order \)

assumes mono \( f \)

shows \( \forall Y. \downarrow 'b::order \rightarrow downclosed-set (f - ' Y) \)

by (simp add: assms downclosed-set-iff monoD)

lemma fixes \( f : 'a::order \Rightarrow 'b::order \)

assumes mono \( f \)
shows \( \forall Y. \text{downclosed-set } X \rightarrow \text{downclosed-set } (f^{-1} X) \)
oops

**lemma downclosed-mono:**

fixes \( f :: 'a::order \Rightarrow 'b::order \)
assumes \( \forall Y. \text{downclosed-set } Y \rightarrow \text{downclosed-set } (f^{-1} Y) \)
shows mono \( f \)

proof
{ fix \( x \) \( y :: 'a::order \)
assume \( h: x \leq y \)
have \( \text{downclosed-set } (\downarrow (f y)) \)
  unfolding \( \text{downclosed-set-def downsets-def Fix-def downset-set-def downset-def} \)
  by auto
hence \( \text{downclosed-set } \{ z. f z \leq f y \} \)
  unfolding \( \text{vimage-def downset-def downset-set-def} \) \by auto
hence \( \forall z w. (f z \leq f y \land w \leq z) \rightarrow f w \leq f y \)
  unfolding \( \text{downclosed-set-def downclosed-set-def downsets-def Fix-def downset-def} \)
  by force
hence \( f x \leq f y \)
  using \( h \) \by blast \}
thus ?thesis..
qed

**lemma mono-downclosed-iff:** mono \( f = (\forall Y. \text{downclosed-set } Y \rightarrow \text{downclosed-set } (f^{-1} Y)) \)
using mono-downclosed downclosed-mono \by auto

context order
begin

**lemma downset-inj:** inj \( \downarrow \)
by (metis injI downset-iso-iff eq-iff)

**lemma** \( (X \subseteq Y) = (\downarrow X \subseteq \downarrow Y) \)
oops

end

context lattice
begin

**lemma lat-ideals:** \( X \in \text{ideals} = (X \neq \{\} \land X \in \text{downsets} \land (\forall x \in X. \forall y \in X. x \sqcup y \in X)) \)
unfolding ideals-def directed-alt downsets-def Fix-def downset-set-def downclosed-set-def by (clarsimp, smt sup.cobounded1 sup.orderE sup.order1 sup.absorb2 sup.left-commute mem-Collect-eq)


context bounded-lattice
begin

lemma bot-ideal: \( X \in \text{ideals} \implies \bot \in X \)
  unfolding ideals-def downclosed-set-def downsets-def Fix-def downset-set-def by fastforce

end

context complete-lattice
begin

lemma Sup-downset-id [simp]: \( \sup \circ \downarrow = \text{id} \)
  using Sup-atMost atMost-def downset-prop by fastforce

lemma downset-Sup-id: \( \text{id} \leq \downarrow \circ \sup \)
  by (simp add: Sup-upper downset-prop le-funI subsetI)

lemma Inf-Sup-var: \( \bigcap \{ x \in X. \downarrow x \} = \downarrow (\bigcap X) \)
  unfolding downset-prop by (simp add: Collect-ball-eq Inf-Sup)

lemma Inf-pres-downset-var: \( \bigcap \{ x \in X. \downarrow x \} = \downarrow (\bigcap X) \)
  unfolding downset-prop by (safe, simp-all add: le-Inf-iff)

end

4.3 Dual Properties of Orderings

context ord-with-dual
begin

lemma filtered-nonempty: \( \text{filtered } X \implies X \neq \{\} \)
  using filtered-to-directed ord.directed-nonempty by auto

lemma filtered-lb: \( \text{filtered } X \implies (\forall x \in X. \forall y \in X. \exists z \in X. z \leq x \land z \leq y) \)
  using filtered-to-directed directed-ub dual-dual-ord by fastforce

lemma upset-set-prop-var: \( \uparrow X = (\bigcup x \in X. \uparrow x) \)
  by (simp add: image-Union downset-set-prop-var upset-set-to-downset-set2 upset-to-downset2)

lemma upset-set-prop: \( \uparrow = \text{Union} \circ (\cdot) \uparrow \)
  unfolding fun-eq-iff by (simp add: upset-set-prop-var)

lemma upset-prop: \( \uparrow x = \{ y. x \leq y \} \)
  unfolding upset-to-downset3 downset-prop image-def using dual-dual-ord by fastforce

24
lemma upset-prop2: \( x \leq y \implies y \in \uparrow x \)
by (simp add: upset-prop)

lemma filters-upsets: \( X \in \text{filters} \implies X \in \text{upsets} \)
by (simp add: upclosed-set-def filters-def)

lemma filters-filtered: \( X \in \text{filters} \implies \text{filtered} X \)
by (simp add: filters-def)

end

context preorder-with-dual
begin

lemma filtered-prop: \( X \neq \{\} \implies (\forall x \in X. \forall y \in X. \exists z \in X. z \leq x \land z \leq y) \implies \text{filtered} X \)

unfolding filtered-to-directed
by (rule directed-prop, blast, metis (full-types) image-iff ord-dual)

lemma filtered-alt: \( \text{filtered} X = (X \neq \{\} \land (\forall x \in X. \forall y \in X. \exists z \in X. z \leq x \land z \leq y)) \)
by (metis image-empty directed-alt filtered-to-directed filtered-lb filtered-prop)

lemma up-set-prop-var2: \( x \in \uparrow X \implies x \leq y \implies y \in \uparrow X \)
using downset-set-prop-var2 dual-iff ord-dual upset-set-to-downset-set2 by fastforce

lemma upclosed-set-iff: \( \text{upclosed-set} X = (\forall x \in X. \forall y. x \leq y \implies y \in X) \)
unfolding upclosed-set-def upsets-def Fix-def upset-set-def by auto

lemma upclosed-upset-set: \( \text{upclosed-set} (\uparrow X) \)
using up-set-prop-var2 upclosed-set-iff by blast

lemma upclosed-upset: \( \text{upclosed-set} (\uparrow x) \)
by (simp add: upset-def upclosed-upset-set)

lemma upset-set-ext: \( \text{id} \leq \uparrow \)
by (smt comp-def comp-id image-mono le-fun-def downset-set-ext image-dual upset-set-to-downset-set2)

lemma upset-set-anti: \( \text{mono} \uparrow \)
by (metis image-mono downset-set-iso upset-set-to-downset-set2 mono-def)

lemma up-set-idem [simp]: \( \uparrow \circ \uparrow = \uparrow \)

lemma upset-faithful: \( \uparrow x \subseteq \uparrow y \implies y \leq x \)
by (metis inj-image-subset-iff downset-faithful dual-ord inj-dual upset-to-downset3)

lemma upset-anti-iff: \( (\uparrow y \subseteq \uparrow x) = (x \leq y) \)
by (metis downset-iso-iff ord-dual upset-to-downset3 subset-image-iff upset-faithful)

lemma upset-filtered-upset [simp]: filtered ◦⇑ = filtered
  by (metis comp-assoc directed-filtered-dual downset-directed-downset upset-set-downset-set-dual)

lemma filtered-upset-filters: filtered (⇑X) = (⇑X ∈ filters)
  by (metis comp-apply directed-downset-ideals filtered-to-directed filterp-idealp-dual
upset-set-downset-set-dual)

lemma upclosed-Fix: upclosed-set X = (⇑X = X)
  by (simp add: Fix-def upclosed-set-def upsets-def)

end

lemma upset-anti: antimono (⇑::'a::order-with-dual ⇒ 'a set)
  by (simp add: antimono-def upset-anti-iff)

lemma mono-upclosed:
  fixes f :: 'a::order-with-dual ⇒ 'b::order-with-dual
  assumes mono f
  shows ∀ Y. upclosed-set Y −→ upclosed-set (f − ' Y)
  by (simp add: assms monoD upclosed-set-iff)

lemma mono-upclosed:
  fixes f :: 'a::order-with-dual ⇒ 'b::order-with-dual
  assumes mono f
  shows ∀ Y. upclosed-set X −→ upclosed-set (f ' X)
  oops

lemma upclosed-mono:
  fixes f :: 'a::order-with-dual ⇒ 'b::order-with-dual
  assumes ∀ Y. upclosed-set Y −→ upclosed-set (f − ' Y)
  shows mono f
  by (metis (mono-tags, lifting) assms dual-order refl mem-Collect-eq monoI or-
der.trans upclosed-set-iff vimageE vimageI2)

lemma mono-upclosed-iff:
  fixes f :: 'a::order-with-dual ⇒ 'b::order-with-dual
  shows mono f = (∀ Y. upclosed-set Y −→ upclosed-set (f − ' Y))
  using mono-upclosed upclosed-mono by auto

context order-with-dual
begin

lemma upset-inj: inj ↑
  by (metis inj-compose inj-on-imageI2 downset-inj inj-dual upset-to-downset)

lemma (X ⊆ Y) = (⇑Y ⊆ ↑X)
  oops
context lattice-with-dual
begin

lemma lat-filters: \( X \in \text{filters} = (X \neq \{\} \land X \in \text{upsets} \land (\forall x \in X. \forall y \in X. x \cap y \in X)) \)

unfolding filters-to-ideals upsets-to-downsets inf-to-sup lat-ideals
by (smt image-iff image-inv-f-f image-is-empty inj-image-mem-iff inv-unique-comp inj-dual invol-dual)

end

context bounded-lattice-with-dual
begin

lemma top-filter: \( X \in \text{filters} \Rightarrow \top \in X \)
using bot-ideal inj-image-mem-iff inj-dual filters-to-ideals top-dual by fastforce

end

context complete-lattice-with-dual
begin

lemma Inf-upset-id \[\text{simp}]: \text{Inf} \circ \uparrow = \text{id} \]
by (metis comp-assoc comp-id Sup-downset-id Sups-dual-def downset-upset-dual invol-dual)

lemma upset-Inf-id: \( \text{id} \leq \uparrow \circ \text{Inf} \)
by (simp add: Inf-lower le-funI subsetI upset-prop)

lemma Sup-Inf-var: \( \bigcap \{x \in X. \uparrow x\} = \bigcup X \)
unfolding upset-prop by (simp add: Collect-ball-eq Sup-Inf)

lemma Sup-dual-upset-var: \( \bigcap \{x \in X. \uparrow x\} = \uparrow \bigcup X \)
unfolding upset-prop by (safe, simp-all add: Sup-le-iff)

end

4.4 Shunting Laws

The first set of laws supplies so-called shunting laws for boolean algebras. Such laws rather belong into Isabelle Main.

context boolean-algebra
begin

lemma shunt1: \( x \cap y \leq z \Rightarrow (x \leq \neg y \cup z) \)
proof standard

end
\begin{verbatim}
assume \(x \cap y \leq z\)
\hence \(\neg y \sqcup (x \cap y) \leq \neg y \sqcup z\)
  using sup.mono by blast
\hence \(\neg y \sqcup x \leq \neg y \sqcup z\)
  by (simp add: sup-inf-distrib1)
thus \(x \leq \neg y \sqcup z\)
  by simp

next
assume \(x \leq \neg y \sqcup z\)
\hence \(x \cap y \leq (\neg y \sqcup z) \cap y\)
  using inf-mono by auto
thus \(x \cap y \leq z\)
  using inf.boundedE inf-sup-distrib2 by auto
qed

lemma shunt2: \((x \cap \neg y \leq z) = (x \leq y \sqcup z)\)
  by (simp add: shunt1)

lemma meet-shunt: \((x \cap y = \bot) = (x \leq \neg y)\)
  by (simp add: eq-iff shunt1)

lemma join-shunt: \((x \sqcup y = \top) = (\neg x \leq y)\)
  by (metis compl-sup compl-top-eq double-compl meet-shunt)

lemma meet-shunt-var: \((x - y = \bot) = (x \leq y)\)
  by (simp add: diff-eq meet-shunt)

lemma join-shunt-var: \((x \rightarrow y = \top) = (x \leq y)\)
  by simp

end

4.5 Properties of Complete Lattices

\begin{definition} 
Inf-closed-set \(X\) \(= (\forall Y \subseteq X. \cap Y \in X)\)
\end{definition}

\begin{definition} 
Sup-closed-set \(X\) \(= (\forall Y \subseteq X. \sqcup Y \in X)\)
\end{definition}

\begin{definition} 
inf-closed-set \(X\) \(= (\forall x \in X. \forall y \in X. x \cap y \in X)\)
\end{definition}

\begin{definition} 
sup-closed-set \(X\) \(= (\forall x \in X. \forall y \in X. x \sqcup y \in X)\)
\end{definition}

The following facts about complete lattices add to those in the Isabelle libraries.

context complete-lattice
begin

The translation between sup and Sup could be improved. The sup-theorems should be direct consequences of Sup-ones. In addition, duality between sup
and inf is currently not exploited.

**Lemma** \( sup-Sup: x \sqcup y = \bigcup \{x, y\} \)
by simp

**Lemma** \( inf-Inf: x \sqcap y = \bigcap \{x, y\} \)
by simp

The next two lemmas are about Sups and Infs of indexed families. These are interesting for iterations and fixpoints.

**Lemma** \( fsup-unfold: (f :: nat \Rightarrow 'a) \theta \sqcup (\bigcup n. f (Suc n)) = (\bigcup n. f n) \)
apply (intro antisym sup-least)
apply (rule Sup-upper, force)
apply (rule Sup-mono, force)
apply (safe intro: Sup-least)
by (case-tac n, simp-all add: Sup-upper le-supI2)

**Lemma** \( finf-unfold: (f :: nat \Rightarrow 'a) \theta \sqcap (\bigcap n. f (Suc n)) = (\bigcap n. f n) \)
apply (intro antisym inf-greatest)
apply (rule Inf-greatest, safe)
apply (case-tac n)
apply simp-all
using Inf-lower inf.coboundedI2 apply force
apply (simp add: Inf-lower)
apply (auto intro: Inf-mono)
by (simp_all)

**End**

**Lemma** \( Sup-sup-closed: Sup-closed-set (X :: 'a::complete-lattice set) \Rightarrow sup-closed-set X \)
b y (metis Sup-closed-set-def empty-subsetI insert-subset sup-Sup sup-closed-set-def)

**Lemma** \( Inf-inf-closed: Inf-closed-set (X :: 'a::complete-lattice set) \Rightarrow inf-closed-set X \)
b y (metis Inf-closed-set-def empty-subsetI inf-Inf inf-closed-set-def insert-subset)

### 4.6 Sup- and Inf-Preservation

Next, important notation for morphism between posets and lattices is introduced: sup-preservation, inf-preservation and related properties.

**Abbreviation** \( Sup-pres :: ('a::Sup \Rightarrow 'b::Sup) \Rightarrow bool \)

\[ \text{Sup-pres } f \equiv f \circ Sup = Sup \circ (\cdot) \] \( f \)

**Abbreviation** \( Inf-pres :: ('a::Inf \Rightarrow 'b::Inf) \Rightarrow bool \)

\[ \text{Inf-pres } f \equiv f \circ Inf = Inf \circ (\cdot) \] \( f \)

**Abbreviation** \( sup-pres :: ('a::sup \Rightarrow 'b::sup) \Rightarrow bool \)

\[ \text{sup-pres } f \equiv (\forall x y. f (x \sqcup y) = f x \sqcup f y) \]
abbreviation inf-pres :: ('a::inf ⇒ 'b::inf) ⇒ bool where
inf-pres f ≡ (∀ x y. f (x ∩ y) = f x ∩ f y)

abbreviation bot-pres :: ('a::bot ⇒ 'b::bot) ⇒ bool where
bot-pres f ≡ f ⊥ = ⊥

abbreviation top-pres :: ('a::top ⇒ 'b::top) ⇒ bool where
top-pres f ≡ f ⊤ = ⊤

abbreviation Sup-dual :: ('a::Sup ⇒ 'b::Inf) ⇒ bool where
Sup-dual f ≡ f ◦ Sup = Inf ◦ (') f

abbreviation Inf-dual :: ('a::Inf ⇒ 'b::Sup) ⇒ bool where
Inf-dual f ≡ f ◦ Inf = Sup ◦ (') f

abbreviation sup-dual :: ('a::sup ⇒ 'b::inf) ⇒ bool where
sup-dual f ≡ (∀ x y. f (x ⊔ y) = f x ⊓ f y)

abbreviation inf-dual :: ('a::inf ⇒ 'b::sup) ⇒ bool where
inf-dual f ≡ (∀ x y. f (x ⊓ y) = f x ⊔ f y)

abbreviation bot-dual :: ('a::bot ⇒ 'b::top) ⇒ bool where
bot-dual f ≡ f ⊥ = ⊤

abbreviation top-dual :: ('a::top ⇒ 'b::bot) ⇒ bool where
top-dual f ≡ f ⊤ = ⊥

Inf-preservation and sup-preservation relate with duality.

lemma Inf-pres-map-dual-var:
Inf-pres f = Sup-pres (∂F f)
for f :: 'a::complete-lattice-with-dual ⇒ 'b::complete-lattice-with-dual
proof –
{ fix x :: 'a set
  assume ∂ (f (∏ x. (∂ x))) = (∏ y∈x. (∂ (f y))) for x
  then have ∏ (f' x) = f (∏ A) for A
    by (metis no-types Sup-dual-def-var sup-dual-subset image-comp)
  then have ∏ (f' x) = f (∏ A)
    by (metis Sup-dual-def-var subset-dual)
  then show ?thesis
    by (auto simp add: map-dual-def fun-eq_iff Inf-dual-var Sup-dual-var)
} qed

lemma Inf-pres-map-dual: Inf-pres = Sup-pres ∘ (∂F::('a::complete-lattice-with-dual ⇒ 'b::complete-lattice-with-dual) ⇒ 'a ⇒ 'b)
proof –
{ fix f :: 'a ⇒ 'b
  have Inf-pres f = (Sup-pres ∘ ∂F) f
    by (simp add: Inf-pres-map-dual-var)
  thus ?thesis
}
by force

qed

**lemma** Sup-pres-map-dual-var:

**fixes** $f :: 'a::complete-lattice-with-dual \Rightarrow 'b::complete-lattice-with-dual$

**shows** $\operatorname{Sup-pres} f = \operatorname{Inf-pres} (\partial F f)$

by (metis $\operatorname{Inf-pres-map-dual-var}$ fun-dual5 map-dual-def)

**lemma** Sup-pres-map-dual: $\operatorname{Sup-pres} = \operatorname{Inf-pres} \circ (\partial F :: ('a::complete-lattice-with-dual \Rightarrow 'b::complete-lattice-with-dual) \Rightarrow 'a \Rightarrow 'b)$

by (simp add: $\operatorname{Inf-pres-map-dual}$ comp-assoc map-dual-invol)

The following lemmas relate isotonicity of functions between complete lattices with weak (left) preservation properties of sups and infs.

**lemma** fun-isol: mono $f \implies$ mono $(\circ f)$

by (simp add: le-fun-def mono-def)

**lemma** fun-isor: mono $f \implies$ mono $(\lambda x. x \circ f)$

by (simp add: le-fun-def mono-def)

**lemma** Sup-sup-pres:

**fixes** $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$

**shows** $\operatorname{Sup-pres} f = \operatorname{sup-pres} f$

by (metis $\operatorname{Sup-empty}$ $\operatorname{Sup-insert}$ comp-apply image-insert sup-bot

right-neutral)

**lemma** Inf-inf-pres:

**fixes** $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$

**shows** $\operatorname{Inf-pres} f = \operatorname{inf-pres} f$

by (smt $\operatorname{INF-insert}$ Inf-empty Inf-insert comp-eq-elim inf-top

right-neutral)

**lemma** Sup-bot-pres:

**fixes** $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$

**shows** $\operatorname{Sup-pres} f = \operatorname{bot-pres} f$

by (metis $\operatorname{SUP-empty}$ $\operatorname{Sup-insert}$ comp-apply image-insert sup-bot

right-neutral)

**lemma** Inf-top-pres:

**fixes** $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$

**shows** $\operatorname{Inf-pres} f = \operatorname{top-pres} f$

by (metis $\operatorname{INF-empty}$ $\operatorname{Inf-insert}$ comp-eq-elim)

**lemma** Sup-sup-dual:

**fixes** $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$

**shows** $\operatorname{Sup-dual} f = \operatorname{sup-dual} f$

by (smt comp-eq-elim image-empty image-insert inf-inf sup-bot

right-neutral)

**lemma** Inf-inf-dual:

**fixes** $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$

**shows** $\operatorname{Inf-dual} f = \operatorname{inf-dual} f$
by (smt comp-eq-elim image-empty image-insert inf-Inf sup-Sup)

lemma Sup-bot-dual:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows Sup-dual f =⇒ bot-dual f
  by (metis INF-empty Sup-empty comp-eq-elim)

lemma Inf-top-dual:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows Inf-dual f =⇒ top-dual f
  by (metis Inf-empty SUP-empty comp-eq-elim)

However, Inf-preservation does not imply top-preservation and Sup-preservation does not imply bottom-preservation.

lemma
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows Sup-pres f =⇒ top-pres f
  oops

lemma
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows Inf-pres f =⇒ bot-pres f
  oops

context complete-lattice
begin

lemma iso-Inf-subdistl:
  fixes f :: 'a ⇒ 'b::complete-lattice
  shows mono f =⇒ f ◦ Inf ≤ Inf ◦ ('') f
  by (simp add: complete-lattice-class.le-INF-iff le-funI Inf-lower monoD)

lemma iso-Sup-supdistl:
  fixes f :: 'a ⇒ 'b::complete-lattice
  shows mono f =⇒ Sup ◦ ('') f ≤ f ◦ Sup
  by (simp add: complete-lattice-class.Sup-le-funI le-funI Sup-upper monoD)

lemma Inf-subdistl-iso:
  fixes f :: 'a ⇒ 'b::complete-lattice
  shows f ◦ Inf ≤ Inf ◦ ('') f =⇒ mono f
  unfolding mono-def le-fun-def comp-def by (metis complete-lattice-class.le-Inf-iff Inf-atLeast atLeast-iff)

lemma Sup-supdistl-iso:
  fixes f :: 'a ⇒ 'b::complete-lattice
  shows Sup ◦ ('') f ≤ f ◦ Sup =⇒ mono f
  unfolding mono-def le-fun-def comp-def by (metis complete-lattice-class.SUP-le-iff Sup-atMost atMost-iff)

32
lemma supdistl-iso:
fixes f :: 'a ⇒ 'b::complete-lattice
shows (Sup ◦ (′)) f ≤ f ◦ Sup = mono f
using Sup-supdistl-iso iso-Sup-supdistl by force

lemma subdistl-iso:
fixes f :: 'a ⇒ 'b::complete-lattice
shows (f ◦ Inf ≤ Inf ◦ (′)) f = mono f
using Inf-subdistl-iso iso-Inf-subdistl by force

end

lemma ord-iso-Inf-pres:
fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
shows ord-iso f =⇒ Inf ◦ (′) f = f ◦ Inf
proof−
let ?g = the-inv f
assume h: ord-iso f
hence a: mono ?g
  by (simp add: ord-iso-the-inv)
{fix X :: 'a::complete-lattice set
  {fix y :: 'b::complete-lattice
    have (y ≤ f (∏ X)) = (?g y ≤ ∏ X)
      by (metis (mono-tags, lifting) UNIV-I f-the-inv-into-f h monoD ord-embed-alt ord-embed-inj ord-iso-alt)
    also have ... = (∀ x ∈ X. ?g y ≤ x)
      by (simp add: le-Inf-iff)
    also have ... = (∀ x ∈ X. y ≤ f x)
      by (metis (mono-tags, lifting) UNIV-I f-the-inv-into-f h monoD ord-embed-alt ord-embed-inj ord-iso-alt)
    also have ... = (y ≤ ∏ (f ′ X))
      by (simp add: le-INF-iff)
    finally have (y ≤ f (∏ X)) = (y ≤ ∏ (f ′ X)).}
hence f (∏ X) = ∏ (f ′ X)
  by (meson dual-order.antisym order-refl)}
thus ?thesis
  unfolding fun-eq-iff by simp
qed

lemma ord-iso-Sup-pres:
fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
shows ord-iso f =⇒ Sup ◦ (′) f = f ◦ Sup
proof−
let ?g = the-inv f
assume h: ord-iso f
hence a: mono ?g
  by (simp add: ord-iso-the-inv)
{fix X :: 'a::complete-lattice set
  {fix y :: 'b::complete-lattice
}}
have \((\bigsqcup X) \leq y\) = \((\bigsqcup X \leq \exists g y)\)
   by (metis (mono-tags, lifting) UNIV-I f-the-inv-into-f h monoD ord-embed-alt
ord-embed-inj ord-iso-alt)
also have ... = (\(\forall x \in X, x \leq \exists g y\))
   by (simp add: Sup-le-iff)
also have ... = (\(\forall x \in X, f x \leq y\))
   by (metis (mono-tags, lifting) UNIV-I f-the-inv-into-f h monoD ord-embed-alt
ord-embed-inj ord-iso-alt)
finally have \((\bigsqcup X) \leq y\) = \((\bigsqcup (f \cdot X) \leq y)\)
   by (meson dual-order antisym order-refl)

hence \(f (\bigsqcup X) = \bigsqcup (f \cdot X)\)
   by (meson dual-order antisym order-refl)

thus \(?thesis

Right preservation of sups and infs is trivial.

lemma fSup-distr: Sup-pres \((\lambda x. x \circ f)\)
   unfolding fun-eq-iff by (simp add: image-comp)

lemma fSup-distr-var: \((\bigsqcup F \circ g = (\bigsqcup f \in F. f \circ g))\)
   unfolding fun-eq-iff by (simp add: image-comp)

lemma fInf-distr: Inf-pres \((\lambda x. x \circ f)\)
   unfolding fun-eq-iff comp-def
   by (smt INF-apply INF-cong INF-image INF-apply image-comp image-def image-image)

lemma fInf-distr-var: \((\bigsqcup F \circ g = (\bigsqcup f \in F. f \circ g))\)
   unfolding fun-eq-iff comp-def
   by (smt INF-apply INF-cong INF-image INF-apply image-comp image-def image-image)

The next set of lemma revisits the preservation properties in the function space.

lemma fSup-subdistl:
   assumes mono \((f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice)\)
   shows \(Sup \circ (\cdot) (\circ) f \leq (\circ) f \circ Sup\)
   using assms by (simp add: fun-isol supdistl-iso)

lemma fSup-subdistl-var:
   fixes f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice
   shows mono f \(\Rightarrow (\bigsqcup g \in G. f \circ g) \leq f \circ \bigsqcup G\)
   by (simp add: fun-isol mono-Sup)

lemma fInf-subdistl:
   fixes f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice
   shows mono f \(\Rightarrow (\circ) f \circ Inf \leq Inf \circ (\cdot) (\circ) f\)
   by (simp add: fun-isol subdistl-iso)
lemma fInf-subdistl-var:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows mono f ‚ f ∩ G ≤ (∩ g ∈ G. f ‚ g)
  by (simp add: fun-isol mono-Inf)

lemma fSup-distl: Sup-pres f ⇒ Sup-pres ((o) f)
  unfolding fun-eq-iff by (simp add: image-comp)

lemma fSup-distl-var: Sup-pres f ⇒ f ◦ ⨆ G = (∨ g ∈ G. f ◦ g)
  unfolding fun-eq-iff by (simp add: image-comp)

lemma fInf-distl: Inf-pres f ⇒ Inf-pres ((o) f)
  unfolding fun-eq-iff by (simp add: image-comp)

lemma fInf-distl-var: Inf-pres f ⇒ f ◦ ⨅ G = (∨ g ∈ G. f ◦ g)
  unfolding fun-eq-iff by (simp add: image-comp)

Downsets preserve infs whereas upsets preserve sups.

lemma Inf-pres-downset: Inf-pres (↓::'a::complete-lattice-with-dual ⇒ 'a set)
  unfolding downset-prop fun-eq-iff by (safe, simp-all add: le-Inf-iff)

lemma Sup-dual-upset: Sup-dual (↑::'a::complete-lattice-with-dual ⇒ 'a set)
  unfolding upset-prop fun-eq-iff by (safe, simp-all add: Sup-le-iff)

Images of Sup-morphisms are closed under Sups and images of Inf-morphisms
are closed under Infs.

lemma Sup-pres-Sup-closed: Sup-pres f ⇒ Sup-closed-set (range f)
  by (metis (mono-tags, lifting) Sup-closed-set-def comp-eq-elim range-eqI subset-image-iff)

lemma Inf-pres-Inf-closed: Inf-pres f ⇒ Inf-closed-set (range f)
  by (metis (mono-tags, lifting) Inf-closed-set-def comp-eq-elim range-eqI subset-image-iff)

It is well known that functions into complete lattices form complete lattices.
Here, such results are shown for the subclasses of isotone functions, where
additional closure conditions must be respected.

typedef (overloaded) 'a iso = {f::'a::order ⇒ 'a::order. mono f}
  by (metis Abs-ord-homset-cases ord-homset-def)

setup-lifting type-definition-iso

instantiation iso :: (complete-lattice) complete-lattice
begin

lift-definition Inf-iso :: 'a::complete-lattice iso set ⇒ 'a iso is Sup
  by (metis (mono-tags, lifting) SUP-subset-mono Sup-apply mono-def subsetI)

35
Duality has been baked into this result because of its relevance for predicate transformers. A proof where Sups are mapped to Sups and Infs to Infs is certainly possible, but two instantiation of the same type and the same classes are unfortunately impossible. Interpretations could be used instead. A corresponding result for Inf-preserving functions and Sup-lattices, is proved in components on transformers, as more advanced properties about Inf-preserving functions are needed.

4.7 Alternative Definitions for Complete Boolean Algebras

The current definitions of complete boolean algebras deviates from that in most textbooks in that a distributive law with infinite sups and infinite infs is used. There are interesting applications, for instance in topology, where weaker laws are needed — for instance for frames and locales.

Complete Heyting algebras are also known as frames or locales (they differ with respect to their morphisms).
class complete-boolean-algebra-alt = complete-lattice + boolean-algebra

instance set :: (type) complete-boolean-algebra-alt..

context complete-boolean-algebra-alt
begin

subclass complete-heyting-algebra
proof
fix x Y
{fix t
  have \((x \cap \bigcup Y \leq t) = (\bigcup Y \leq \neg x \cup t)\)
    by (simp add: inf.commute shuntI[symmetric])
  also have \(\ldots = (\forall y \in Y. \ y \leq \neg x \cup t)\)
    using Sup-le-iff by blast
  also have \(\ldots = (\forall y \in Y. x \cap y \leq t)\)
    by (simp add: inf.commute shuntI)
  finally have \((x \cap \bigcup Y \leq t) = ((\bigcup y \in Y. x \cap y) \leq t)\)
    by (simp add: local.SUP-le-iff)
  thus \(x \cap \bigcup Y = (\bigcup y \in Y. x \cap y)\)
    using eq-iff by blast
qed

subclass complete-co-heyting-algebra
  apply unfold-locales
  apply (rule antisym)
  apply (simp add: INF-greatest Inf-lower2)
  by (meson eq-refl le-INF-iff le-Inf-iff shunt2)

lemma de-morgan1: \(- (\bigcup X) = (\bigcap x \in X. \neg x)\)
proof
  {fix y
    have \((y \leq -(\bigcup X)) = (\bigcup X \leq \neg y)\)
      using compl-le-swapI by blast
    also have \(\ldots = (\forall x \in X. \ x \leq \neg y)\)
      by (simp add: Sup-le-iff)
    also have \(\ldots = (\forall y \in X. \ y \leq \neg x)\)
      using compl-le-swapI by blast
    also have \(\ldots = (y \leq (\bigcap x \in X. \neg x))\)
      using le-INF-iff by force
    finally have \((y \leq -(\bigcup X)) = (y \leq (\bigcap x \in X. \neg x))\)
    thus \(?thesis\)
      using antisym by blast
  }
qed

lemma de-morgan2: \(- (\bigcap X) = (\bigcup x \in X. \neg x)\)
  by (metis de-morgan1 ba-dual.dual-iff ba-dual.image-dual pointfree-idE)
end
class complete-boolean-algebra-alt-with-dual = complete-lattice-with-dual + complete-boolean-algebra-alt

instantiation set :: (type) complete-boolean-algebra-alt-with-dual
begin

definition dual-set :: 'a set ⇒ 'a set where
dual-set = uminus

instance
  by intro-classes (simp-all add: ba-dual.inj-dual dual-set-def comp-def uminus-Sup id-def)
end

context complete-boolean-algebra-alt
begin

sublocale cba-dual: complete-boolean-algebra-alt-with-dual - - - - - - uminus - -
by unfold-locales (auto simp: de-morgan2 de-morgan1)
end

4.8 Atomic Boolean Algebras

Next, atomic boolean algebras are defined.

context bounded-lattice
begin

Atoms are covers of bottom.

definition atom x = (x ≠ ⊥ ∧ ¬(∃ y. ⊥ < y ∧ y < x))

definition atom-map x = {y. atom y ∧ y ≤ x}

lemma atom-map-def-var: atom-map x = ↓x ∩ Collect atom
  unfolding atom-map-def downset-def downset-set-def comp-def atom-def by fastforce

lemma atom-map-atoms: (range atom-map) = Collect atom
  unfolding atom-map-def atom-def by auto

end

typedef (overloaded) 'a atoms = range (atom-map::'a::bounded-lattice ⇒ 'a set)
  by blast

setup-lifting type-definition-atoms

definition at-map :: 'a::bounded-lattice ⇒ 'a atoms where
  at-map = Abs-atoms o atom-map
class atomic-boolean-algebra = boolean-algebra +
  assumes atomicity: \( x \neq \perp \implies (\exists y. \text{atom } y \land y \leq x) \)

class complete-atomic-boolean-algebra = complete-lattice + atomic-boolean-algebra

begin
  subclass complete-boolean-algebra-alt ..
end

Here are two equivalent definitions for atoms; first in boolean algebras, and then in complete boolean algebras.

context boolean-algebra
begin
  The following two conditions are taken from Koppelberg’s book [6].

  lemma atom-neg: \( \text{atom } x \implies x \neq \perp \land (\forall y z. \ x \leq y \lor x \leq -y) \)
  by (metis atom-def dual-order.order-iff-strict inf.cobounded1 inf.commute meet-shunt)

  lemma atom-sup: \( (\forall y. \ x \leq y \lor x \leq -y) \implies (\forall y z. \ x \leq y \lor x \leq z) = (x \leq y \lor z) \)
  by (metis inf.orderE le-supI1 shunt2)

  lemma sup-atom: \( x \neq \perp \implies (\forall y z. \ x \leq y \lor x \leq z) = (x \leq y \lor z) \implies \text{atom } x \)
  unfolding atom-def apply clarsimp by (metis bot-less inf.absorb2 less-le-not-le meet-shunt sup-compl-top)

  lemma atom-sup-iff: \( \text{atom } x = (x \neq \perp \land (\forall y z. \ x \leq y \lor x \leq z) = (x \leq y \lor z)) \)
  by (standard, auto simp add: atom-neg atom-sup sup-atom)

  lemma atom-neg-iff: \( \text{atom } x = (x \neq \perp \land (\forall y z. \ x \leq y \lor x \leq -y)) \)
  by (standard, auto simp add: atom-neg atom-sup sup-atom)

  lemma atom-map-bot-pres: atom-map \( \perp = \{\} \)
  using atom-def atom-map-def le-bot by auto

  lemma atom-map-top-pres: atom-map \( \top = \text{Collect atom} \)
  using atom-map-def by auto

end

context complete-boolean-algebra-alt
begin

  lemma atom-Sup: \( \forall Y. \ x \neq \perp \implies (\forall y. \ x \leq y \lor x \leq -y) \implies ((\exists y \in Y. \ x \leq y) = (x \leq \bigsqcup Y)) \)

end
by (metis Sup-least Sup-upper2 compl-swap1 le-iff-inf meet-shunt)

lemma Sup-atom: \( x \neq \bot \implies (\forall Y. (\exists y \in Y. x \leq y) = (x \leq \bigcup Y)) \implies \text{atom } x \)
proof
  assume \( h1: x \neq \bot \)
  and \( h2: \forall Y. (\exists y \in Y. x \leq y) = (x \leq \bigcup Y) \)
  hence \( \forall y z. (x \leq y \lor x \leq z) = (x \leq y \cup z) \)
    by (smt insert-iff sup-Sup sup-bot.right-neutral)
thus \( \text{atom } x \)
  by (simp add: h1 sup-atom)
qed

lemma atom-Sup-iff: \( \text{atom } x = (x \neq \bot \land (\forall Y. (\exists y \in Y. x \leq y) = (x \leq \bigcup Y))) \)
by standard (auto simp: atom-neg atom-Sup Sup-atom)

end

end

5 Representation Theorems for Orderings and Lattices

theory Representations
  imports Order-Lattice-Props
begin

5.1 Representation of Posets

The isomorphism between partial orders and downsets with set inclusion is well known. It forms the basis of Priestley and Stone duality. I show it not only for objects, but also order morphisms, hence establish equivalences and isomorphisms between categories.

typedef (overloaded) '\(a\) downset = range (\downarrow::'a::ord => 'a set)'
  by fastforce

setup-lifting type-definition-downset

The map ds yields the isomorphism between the set and the powerset level if its range is restricted to downsets.

definition ds :: 'a::ord => 'a downset where
  ds = Abs-downset \circ \downarrow

In a complete lattice, its inverse is Sup.

definition SSup :: 'a::complete-lattice downset => 'a where
  SSup = Sup \circ Rep-downset
lemma \( \text{ds-SSup-inv: } ds \circ \text{SSup} = (id::'a::complete-lattice \downset \Rightarrow 'a \downset) \)
unfolding ds-def SSup-def
by (smt Rep-downset Rep-downset-inverse cSup-atMost eq-id-iff imageE o-def ord-class.atMost-def ord-class.downset-prop)

lemma \( \text{SSup-ds-inv: } \text{SSup} \circ ds = (id::'a::complete-lattice \Rightarrow 'a) \)
unfolding ds-def SSup-def fun-eq-iff id-def comp-def by (simp add: Abs-downset-inverse pointfree-idE)

instantiation \( \downset :: (\text{ord}) \text{order} \)
begin

lift-definition \( \text{less-eq-downset :: 'a \downset \Rightarrow 'a \downset \Rightarrow bool} \) is \( (\lambda X Y. \text{Rep-downset} X \subseteq \text{Rep-downset} Y) \).

lift-definition \( \text{less-downset :: 'a \downset \Rightarrow 'a \downset \Rightarrow bool} \) is \( (\lambda X Y. \text{Rep-downset} X \subset \text{Rep-downset} Y) \).

instance
by (intro-classes, transfer, auto simp: Rep-downset-inject less-eq-downset-def)
end

lemma \( \text{ds-iso: mono ds} \)
unfolding mono-def ds-def fun-eq-iff comp-def
by (metis Abs-downset-inverse downset-iso-iff less-eq-downset.rep-eq rangeI)

lemma \( \text{ds-faithful: } ds x \leq ds y \Rightarrow x \leq (y::'a::order) \)
by (simp add: Abs-downset-inverse downset-faithful ds-def less-eq-downset.rep-eq)

lemma \( \text{ds-inj: inj (ds::'a::order \Rightarrow 'a \downset)} \)
by (simp add: ds-faithful dual-order.antisym injI)

lemma \( \text{ds-surj: surj ds} \)
by (metis (no-types, hide-lams) Rep-downset Rep-downset-inverse ds-def image-iff o-apply surj-def)

lemma \( \text{ds-bij: bij (ds::'a::order \Rightarrow 'a \downset)} \)
by (simp add: bijI ds-inj ds-surj)

lemma \( \text{ds-ord-iso: ord-iso ds} \)
unfolding ord-iso-def comp-def inf-bool-def by (smt UNIV-I ds-bij ds-faithful ds-inj ds-iso ds-surj f-the-inv-into-f inf1I mono-def)

The morphisms between orderings and downsets are isotone functions. One can define functors mapping back and forth between these.
definition \( \text{map-ds :: ('a::complete-lattice \Rightarrow 'b::complete-lattice) \Rightarrow ('a \downset \Rightarrow 'b \downset)} \) where
map-ds f = ds ∘ f ∘ SSup

This definition is actually contrived. We have shown that a function f between posets P and Q is isotone if and only if the inverse image of f maps downclosed sets in Q to downclosed sets in P. There is the following duality: ds is a natural transformation between the identity functor and the preimage functor as a contravariant functor from P to Q. Hence orderings with isotone maps and downsets with downset-preserving maps are dual, which is a first step towards Stone duality. I don’t see a way of proving this with Isabelle, as the types of the preimage of f are the wrong way and I don’t see how I could capture opposition with what I have.

lemma map-ds-prop:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows unfolding map-ds-def by (simp add: SSup-ds-inv comp-assoc)

lemma map-ds-prop2:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows unfolding map-ds-def by (simp add: SSup-ds-inv comp-assoc)

This is part of showing that map-ds is naturally isomorphic to the identity functor, ds being the natural isomorphism.

definition map-SSup :: (′a downset ⇒ ′b downset) ⇒ (′a::complete-lattice ⇒ ′b::complete-lattice) where
  map-SSup F = SSup ◦ F ◦ ds

lemma map-SSup-iso-pres:
  fixes F :: 'a::complete-lattice downset ⇒ ′b::complete-lattice downset
  shows unfolding fun-eq-iff mono-def map-ds-def ds-def SSup-def comp-def by (metis Abs-downset-inverse Sup-subset-mono downset-iso-iff less-eq-downset.rep-eq rangeI)

lemma map-SSup-prop:
  fixes F :: 'a::complete-lattice downset ⇒ ′b::complete-lattice downset
  shows unfolding map-SSup-def by (metis ds-SSup-inv fun.map-id0 id-def rewriteL-comp-comp)

lemma map-SSup-prop2:
  fixes F :: 'a::complete-lattice downset ⇒ ′b::complete-lattice downset

42
\begin{verbatim}
shows \(ds \circ map-SSup F = id F \circ ds\)
by (simp add: map-SSup-prop)

lemma map-ds-func1: \(map-ds id = (id :: 'a::complete-lattice downset \Rightarrow 'a downset)\)
by (simp add: ds-SSup-inv map-ds-def)

lemma map-ds-func2:
fixes \(g :: 'a::complete-lattice \Rightarrow 'b::complete-lattice\)
shows \(map-ds (f \circ g) = map-ds f \circ map-ds g\)
unfolding map-ds-def fun-eq-iff comp-def ds-def SSup-def
by (metis Abs-downset-inverse Sup-atMost atMost-def downset-prop rangeI)

lemma map-SSup-func1:
\(map-SSup (id :: 'a::complete-lattice downset \Rightarrow 'a downset) = id\)
by (simp add: SSup-ds-inv map-SSup-def)

lemma map-SSup-func2:
fixes \(F :: 'c::complete-lattice downset \Rightarrow 'a::complete-lattice\)
\(G :: 'a::complete-lattice \Rightarrow 'c downset\)
shows \(map-SSup (F \circ G) = map-SSup F \circ map-SSup G\)
unfolding map-SSup-def fun-eq-iff comp-def id-def ds-def
by (metis comp-apply ds-SSup-inv ds-def id-apply)

lemma map-ds-map-SSup-inv:
fixes \(g :: 'a::complete-lattice \Rightarrow 'b::complete-lattice\)
shows \(map-ds map-SSup g = map-SSup \circ map-ds g\)
by (metis inj-eq inj-map-ds map-ds-map-SSup-inv pointfree-idE)

lemma inj-map-ds: inj (map-ds::('a::complete-lattice \Rightarrow 'b::complete-lattice) \Rightarrow ('a downset \Rightarrow 'b downset))
by (metis inj-on-id inj-on-imageI2 map-ds-map-SSup-inv)

lemma map-ds-map-SSup: inj (map-SSup::('a::complete-lattice downset \Rightarrow 'b::complete-lattice downset) \Rightarrow ('a \Rightarrow 'b))
by (metis inj-on-id inj-on-imageI2 map-ds-map-SSup-inv)

This gives an isomorphism between categories.
\end{verbatim}
lemma surj-map-ds: surj (map-ds::('a::complete-lattice ⇒ 'b::complete-lattice) ⇒ ('a downset ⇒ 'b downset))  
  by (simp add: map-ds-map-SSup-iff surj-def)

lemma surj-map-SSup: surj (map-SSup::('a::complete-lattice-with-dual downset ⇒ 'b::complete-lattice-with-dual downset) ⇒ ('a ⇒ 'b))  
  by (metis map-ds-map-SSup-iff surjI)

There is of course a dual result for upsets with the reverse inclusion ordering.  
Once again, it seems impossible to capture the "real" duality that uses the  
inverse image functor.

typedef (overloaded) 'a upset = range (∪::'a::ord ⇒ 'a set)  
  by fastforce

setup-lifting type-definition-upset

definition us :: 'a::ord ⇒ 'a upset where  
  us = Abs-upset ◦ ↑

definition IInf :: 'a::complete-lattice upset ⇒ 'a where  
  IInf = Inf ◦ Rep-upset

lemma us-ds: us = Abs-upset ◦ (↑) ◦ Rep-downset ◦ ds ◦ (↑::'a::ord-with-dual ⇒ 'a)  
  unfolding us-def ds-def fun-eq-iff comp-def by (simp add: Abs-downset-inverse upset-to-downset2)

lemma IInf-SSup: IInf = (↑) ◦ SSup ◦ Abs-downset ◦ (↑::'a::complete-lattice-with-dual ⇒ 'a) ◦ Rep-upset  
  unfolding IInf-def SSup-def fun-eq-iff comp-def  
  by (metis (no-types, hide-lams) Abs-downset-inverse Rep-upset Sup-dual-def-var image-iff rangeI subset-dual upset-to-downset3)

lemma us-IInf-inv: us ◦ IInf = (id::'a::complete-lattice-with-dual upset ⇒ 'a upset)  
  unfolding us-def IInf-def fun-eq-iff id-def comp-def  
  by (metis Abs-upset-inverse Sup-IInf-var Sup-atLeastAtMost Sup-dual-upset-var order-refl rangeI)

lemma IInf-us-inv: IInf ◦ us = (id::'a::complete-lattice-with-dual ⇒ 'a)  
  unfolding us-def IInf-def fun-eq-iff id-def comp-def  
  by (metis Abs-upset-inverse Sup-IInf-var Sup-atLeastAtMost Sup-dual-upset-var order-refl rangeI)

instantiation upset :: (ord) order

begin

lift-definition less-eq-upset :: 'a upset ⇒ 'a upset ⇒ bool is (λX Y. Rep-upset X ⊇ Rep-upset Y) .

44
lift-definition less-upset :: 'a upset ⇒ 'a upset ⇒ bool is (λX Y. Rep-upset X ⊃ Rep-upset Y).

instance
  by (intro-classes, transfer, simp-all add: less-le-not-le less-eq-upset.rep-eq Rep-upset-inject)
end

lemma us-iso: x ≤ y ⇒ us x ≤ us (y::'a::order-with-dual)
  by (simp add: Abs-upset-inverse less-eq-upset.rep-eq upset-anti-iff)

lemma us-faithful: us x ≤ us y ⇒ x ≤ (y::'a::order-with-dual)
  by (simp add: Abs-upset-inverse)

lemma us-inj: inj (us::'a::order-with-dual ⇒ 'a upset)
  unfolding inj-def by (simp add: us-faithful)

lemma us-surj: surj (us::'a::order-with-dual ⇒ 'a upset)
  unfolding surj-def by (metis Rep-upset us-surj)

lemma us-bij: bij (us::'a::order-with-dual ⇒ 'a upset)
  by (simp add: bij-def us-surj us-inj)

lemma us-ord-iso: ord-iso (us::'a::order-with-dual ⇒ 'a upset)
  unfolding ord-iso-def
  by (simp, metis (no-types, lifting) UNIV-I f-the-inv-into-f monoI us-iso us-bij)

definition map-us :: ('a::complete-lattice ⇒ 'b::complete-lattice) ⇒ ('a upset ⇒ 'b upset) where
  map-us f = us o f o IInf

lemma map-us-prop: map-us f o (us::'a::complete-lattice-with-dual ⇒ 'a upset) = us o id f
  unfolding map-us-def by (simp add: IInf-us-inv)

definition map-IInf :: ('a::complete-lattice ⇒ 'b::complete-lattice) ⇒ ('a upset ⇒ 'b upset) where
  map-IInf F = IInf o F o us

lemma map-IInf-prop: (us::'a::complete-lattice-with-dual ⇒ 'a upset) o map-IInf F = id F o us
  proof
    have us o map-IInf F = (us o IInf) o F o us
      by (simp add: fun.map-comp map-IInf-def)
    thus ?thesis
      by (simp add: us-IInf-inv)
qed

lemma map-us-func1: map us id = (id::{a::complete-lattice-with-dual upset ⇒ 'a upset})
  unfolding map-us-def fun-eq-iff comp-def us-def id-def IInf-def
  by (metis (no-types, lifting) Inf-upset-id Rep-upset Rep-upset-inverse image-iff pointfree-idE)

lemma map-us-func2:
  fixes f :: 'c::complete-lattice-with-dual ⇒ 'b::complete-lattice-with-dual
  and g :: 'a::complete-lattice-with-dual ⇒ 'c
  shows map-us (f ◦ g) = map-us f ◦ map-us g
  unfolding map-us-def fun-eq-iff comp-def us-def IInf-def
  by (metis Abs-upset-inverse Sup-Inf-var Sup-atLeastAtMost Sup-dual-upset-var order-refl rangeI)

lemma map-IInf-func1:
  unfolding map-IInf-def fun-eq-iff comp-def id-def us-def IInf-def
  by (simp add: Abs-upset-inverse pointfree-idE)

lemma map-IInf-func2:
  fixes F :: 'c::complete-lattice-with-dual upset ⇒ 'b::complete-lattice-with-dual
  and G :: 'a::complete-lattice-with-dual upset ⇒ 'c upset
  shows map-IInf (F ◦ G) = map-IInf F ◦ map-IInf G
  unfolding map-IInf-def fun-eq-iff comp-def id-def us-def
  by (metis comp-apply id-apply us-IInf-inv us-def)

lemma inj-map-us:
  inj (map-us::{a::complete-lattice-with-dual ⇒ 'b::complete-lattice-with-dual}
  ⇒ ('a upset ⇒ 'b upset))
  unfolding map-us-def us-def IInf-def inj-def comp-def fun-eq-iff
  by (metis (no-types, lifting) Inf-upset-id Rep-upset Rep-upset-inverse image-iff pointfree-idE)

lemma inj-map-IInf:
  inj (map-IInf::{a::complete-lattice-with-dual upset ⇒ 'b::complete-lattice-with-dual upset}
  ⇒ ('a ⇒ 'b))
  unfolding map-IInf-def fun-eq-iff inj-def comp-def IInf-def us-def
  by (metis (no-types, hide-lams) Abs-upset-inverse Inf-upset-id pointfree-idE rangeI)

46
lemma map-us-map-IInf-iff:
  fixes g :: 'a::complete-lattice-with-dual ⇒ 'b::complete-lattice-with-dual
  shows (f = map-us g) = (map-IInf f = g)
  by (metis inj-eq inj-map-us map-us-map-IInf-inv pointfree-idE)

lemma map-us-mono-pres:
  fixes f :: 'a::complete-lattice-with-dual ⇒ 'b::complete-lattice-with-dual
  shows mono f =⇒ mono (map-us f)
  unfolding mono-def map-us-def comp-def us-def IInf-def
  by (metis Abs-upset-inverse Inf-superset-mono less-eq-upset rep-eq rangeI upset-anti-iff)

lemma map-IInf-mono-pres:
  fixes F :: 'a::complete-lattice-with-dual upset ⇒ 'b::complete-lattice-with-dual
  shows mono F =⇒ mono (map-IInf F)
  unfolding mono-def map-IInf-def comp-def us-def IInf-def
  oops

lemma surj-map-us: surj (map-us::('a::complete-lattice-with-dual ⇒ 'b::complete-lattice-with-dual) ⇒ ('a upset ⇒ 'b upset))
  by (simp add: map-us-map-IInf-iff surj-def)

lemma surj-map-IInf: surj (map-IInf::('a::complete-lattice-with-dual upset ⇒ 'b::complete-lattice-with-dual upset) ⇒ ('a ⇒ 'b))
  by (metis map-us-map-IInf-iff surjI)

Hence we have again an isomorphism — or rather equivalence — between categories. Here, however, duality is not consistently picked up.

5.2 Stone’s Theorem in the Presence of Atoms

Atom-map is a boolean algebra morphism.

context boolean-algebra
begin

lemma atom-map-compl-pres: atom-map (¬x) = Collect atom − atom-map x
  proof
    { fix y have (y ∈ atom-map (¬x)) = (atom y ∧ y ≤ ¬x)
        by (simp add: atom-map-def)
      also have ... = (atom y ∧ ¬(y ≤ x))
        by (metis atom-sup-iff inf.orderE meet-shunt sup-compl-top top.ordering-top-axioms ordering-top.extremum)
      also have ... = (y ∈ Collect atom − atom-map x)
        using atom-map-def by auto
      finally have (y ∈ atom-map (¬x)) = (y ∈ Collect atom − atom-map x).
    }
  thus ?thesis
lemma atom-map-sup-pres: \( \text{atom-map} (x \sqcup y) = \text{atom-map} x \cup \text{atom-map} y \)
proof
  { fix \( z \)
  have \((z \in \text{atom-map} (x \sqcup y)) = (atom z \land z \leq x \sqcup y)\)
    by (simp add: atom-map-def)
  also have \((z \in \text{atom-map} x \cup \text{atom-map} y)\)
    using atom-sup-iff by auto
  also have \((z \in \text{atom-map} x \cup \text{atom-map} y)\)
    using atom-map-def by auto
  finally have \((z \in \text{atom-map} (x \sqcup y)) = (z \in \text{atom-map} x \cup \text{atom-map} y)\)
    by blast
  } 
thus \(?\text{thesis}\)
by blast
qed

lemma atom-map-inf-pres: \( \text{atom-map} (x \sqcap y) = \text{atom-map} x \cap \text{atom-map} y \)
by (smt Diff-Un atom-map-compl-pres atom-map-sup-pres compl-inf double-compl)

lemma atom-map-minus-pres: \( \text{atom-map} (x - y) = \text{atom-map} x - \text{atom-map} y \)
using atom-map-compl-pres atom-map-def diff-eq by auto
end

The homomorphic images of boolean algebras under atom-map are boolean algebras — in fact powerset boolean algebras.

instantiation atoms :: (boolean-algebra) boolean-algebra
begin

lift-definition minus-atoms :: \('a atoms \Rightarrow 'a atoms \Rightarrow 'a atoms\) \is\( \lambda x y. \text{Abs-atoms} (\text{Rep-atoms} x - \text{Rep-atoms} y)\).

lift-definition uminus-atoms :: \('a atoms \Rightarrow 'a atoms\) \is\( \lambda x. \text{Abs-atoms} (\text{Collect atom} - \text{Rep-atoms} x)\).

lift-definition bot-atoms :: \('a atoms\) \is\( \text{Abs-atoms} \{\}\)\).

lift-definition sup-atoms :: \('a atoms \Rightarrow 'a atoms \Rightarrow 'a atoms\) \is\( \lambda x y. \text{Abs-atoms} (\text{Rep-atoms} x \cup \text{Rep-atoms} y)\).

lift-definition top-atoms :: \('a atoms\) \is\( \text{Abs-atoms} (\text{Collect atom})\).

lift-definition inf-atoms :: \('a atoms \Rightarrow 'a atoms \Rightarrow 'a atoms\) \is\( \lambda x y. \text{Abs-atoms} (\text{Rep-atoms} x \cap \text{Rep-atoms} y)\).

lift-definition less-eq-atoms :: \('a atoms \Rightarrow 'a atoms \Rightarrow \text{bool}\) \is\( \lambda x y. \text{Rep-atoms} x \subseteq \text{Rep-atoms} y\).
lift-definition \( \text{less-atoms} :: 'a \text{ atoms} \Rightarrow 'a \text{ atoms} \Rightarrow \text{bool} \) (\( \lambda x y. \text{Rep-atoms} x \subset \text{Rep-atoms} y \)).

\begin{verbatim}
instance
  apply intro-classes
    apply (transfer, simp add: less-le-not-le)
    apply (transfer, simp)
    apply (transfer, blast)
    apply (simp add: Rep-atoms-inject less-eq-atoms.abs-eq)
    apply (transfer, smt Abs-atoms-inverse Rep-atoms atom-map-inf-pres
      image-iff inf-le1 rangeI)
    apply (transfer, smt Abs-atoms-inverse Rep-atoms atom-map-inf-pres
      image-iff inf-le2 rangeI)
    apply (transfer, smt Abs-atoms-inverse Rep-atoms atom-map-inf-pres
      image-iff le-iff-sup range1 sup-inf-distrib1)
    apply (transfer, smt Abs-atoms-inverse Rep-atoms atom-map-sup-pres
      image-iff inf.orderE inf-sup-acI(6) le-iff-sup order-refl rangeI)
    apply (transfer, smt Abs-atoms-inverse Rep-atoms atom-map-sup-pres
      sup.le-commute sup.right-idem)
    apply (transfer, subst Abs-atoms-inverse, metis (no-types, lifting) Rep-atoms
      atom-map-sup-pres image-iff rangeI, simp)
    apply transfer using Abs-atoms-inverse atom-map-bot-pres apply blast
    apply (transfer, smt Abs-atoms-inverse Rep-atoms atom-map-compl-pres
      atom-map-top-pres diff-eq double-compl inf-le1 rangeE rangeI)
    apply (transfer, smt Abs-atoms-inverse Rep-atoms atom-map-inf-pres atom-map-sup-pres
      image-iff range1 sup-inf-distrib1)
    apply (transfer, metis (no-types, hide-lams) Abs-atoms-inverse Diff-disjoint
      Rep-atoms atom-map-compl-pres rangeE rangeI)
    apply (transfer, smt Abs-atoms-inverse uminus-atoms.abs-eq Rep-atoms Un-Diff-cancel
      atom-map-compl-pres atom-map-inf-pres atom-map-minus-pres atom-map-sup-pres
      atom-map-top-pres diff-eq double-compl inf-compl-bot-right rangeE rangeI sup-commute
      sup-compl-top)
    by transfer (smt Abs-atoms-inverse Rep-atoms atom-map-compl-pres atom-map-inf-pres
      atom-map-minus-pres diff-eq rangeE rangeI)

end
\end{verbatim}

The homomorphism atom-map can then be restricted in its output type to
the powerset boolean algebra.

\begin{verbatim}
lemma at-map-bot-pres: at-map \( \bot = \bot \)
  by (simp add: at-map-def atom-map-bot-pres bot-atoms.transfer)

lemma at-map-top-pres: at-map \( \top = \top \)
  by (simp add: at-map-def atom-map-top-pres top-atoms.transfer)

lemma at-map-compl-pres: at-map \circ uminus = uminus \circ at-map
  unfolding fun-eq-iff by (simp add: Abs-atoms-inverse at-map-def atom-map-compl-pres
    uminus-atoms.abs-eq)
\end{verbatim}
lemma at-map-sup-pres: at-map \( (x \sqcup y) = at-map x \sqcup at-map y \)
unfolding at-map-def comp-def by (metis (mono-tags, lifting) Abs-atoms-inverse
atom-map-sup-pres rangeI sup-atoms.transfer)

lemma at-map-inf-pres: at-map \( (x \sqcap y) = at-map x \sqcap at-map y \)
unfolding at-map-def comp-def by (metis (mono-tags, lifting) Abs-atoms-inverse
atom-map-inf-pres inf-atoms.transfer rangeI)

lemma at-map-minus-pres: at-map \( (x - y) = at-map x - at-map y \)
unfolding at-map-def comp-def by (simp add: Abs-atoms-inverse atom-map-minus-pres
minus-atoms.abs-eq)

context atomic-boolean-algebra
begin

In atomic boolean algebras, atom-map is an embedding that maps atoms of
the boolean algebra to those of the powerset boolean algebra. Analogous
properties hold for at-map.

lemma inj-atom-map: inj atom-map
proof−
\{ fix \( x \) \( y \)::'a
  assume \( x \neq y \)
  hence \( x \sqcap -y \neq \bot \lor -x \sqcap y \neq \bot \)
    by (auto simp: meet-shunt)
  hence \( \exists z. \) atom \( z \land (z \leq x \sqcap -y \lor z \leq -x \sqcap y) \)
    using atomicity by blast
  hence \( \exists z. \) atom \( z \land ((z \in \) atom-map \( x \) \land \neg(z \in \) atom-map \( y \)) \lor (\neg(z \in \) atom-map \( x \) \land z \in \) atom-map \( y \)) \)
    unfolding atom-def atom-map-def by (clarsimp, metis diff-eq inf.orderE
meet-shunt-var)
  hence \( \) atom-map \( x \neq \) atom-map \( y \)
    by blast\}
  thus ?thesis
    by (meson injI)
qed

lemma atom-map-atom-pres: atom \( x \implies \) atom-map \( x = \{x\} \)
unfolding atom-map-def atom-map-def by (auto simp: bot-less dual-order.order-iff-strict)

lemma atom-map-atom-pres2: atom \( x \implies \) atom \( (\) atom-map \( x \) \)
proof−
\{ assume \( \) atom \( x \)
  hence \( \) atom-map \( x = \{x\} \)
    by (simp add: atom-map-atom-pres)
  thus \( \) atom \( (\) atom-map \( x \) \)
    using bounded-lattice-class.atom-def by auto
qed
lemma inj-at-map: inj (at-map::'a::atomic-boolean-algebra ⇒ 'a atoms)
  unfolding at-map-def comp-def by (metis (no-types, lifting) Abs-atoms-inverse inj-at-map inj-def rangeI)

lemma at-map-atom-pres: atom (x::'a::atomic-boolean-algebra) ⇒ at-map x = Abs-atoms {x}
  unfolding at-map-def comp-def by (simp add: atom-map-atom-pres)

lemma at-map-atom-pres2: atom (x::'a::atomic-boolean-algebra) ⇒ atom (at-map x)
  unfolding at-map-def comp-def by (metis Abs-atoms-inverse atom-def atom-map-atom-pres2 atom-map-bot-pres bot-atoms.abs-eq less-atoms.abs-eq rangeI)

Homomorphic images of atomic boolean algebras under atom-map are therefore atomic (rather obviously).

instance atoms :: (atomic-boolean-algebra) atomic-boolean-algebra
proof intro-classes
  fix x::'a atoms
  assume x ≠ ⊥
  hence ∃y. x = at-map y ∧ x ≠ ⊥
    unfolding at-map-def comp-def by (metis Abs-atoms-cases rangeE)
  hence ∃y. x = at-map y ∧ (∃z. atom z ∧ z ≤ y)
    using at-map-bot-pres atomicity by force
  hence ∃y. x = at-map y ∧ (∃z. atom (at-map z) ∧ at-map z ≤ at-map y)
    by (metis at-map-atoms-pres2 at-map-sup-pres sup.orderE sup-ge2)
  thus ∃y. atom y ∧ y ≤ x
    by fastforce
qed

context complete-boolean-algebra-alt
begin

In complete boolean algebras, atom-map is surjective; more precisely it is the left inverse of Sup, at least for sets of atoms. Below, this statement is made more explicit for at-map.

lemma surj-atom-map: Y ⊆ Collect atom ⇒ Y = atom-map (∪ Y)
proof
  assume Y ⊆ Collect atom
  thus Y ⊆ atom-map (∪ Y)
    using Sup-upper atom-map-def by force
next
  assume Y ⊆ Collect atom
  hence a: ∀y. y ∈ Y ⇒ atom y
    by blast
  fix z
assume \( h: z \in \text{Collect atom} - Y \)

hence \( \forall y \in Y. y \cap z = \bot \)

by (metis DiffE a h atom-def dual-order_not-eq-order-implies-strict inf.absorb_iff2 inf-le2 meet-shunt mem-Collect_eq)

hence \( \bigsqcup Y \cap z = \bot \)

by (simp add: atom-def dual-order mem-Collect_eq)

hence \( z \not\in \text{atom-map} (\bigsqcup Y) \)

thus \( \text{atom-map} (\bigsqcup Y) \subseteq Y \)

qed

In this setting, atom-map is a complete boolean algebra morphism.

lemma atom-map-Sup-pres: \( \text{atom-map} (\bigsqcup X) = (\bigcup x \in X. \text{atom-map} x) \)

proof -

{ fix \( z \)

have \( (z \in \text{atom-map} (\bigsqcup X)) = (\text{atom } z \wedge z \leq \bigsqcup X) \)

by (simp add: atom-map-def)

also have \( \ldots = (\text{atom } z \wedge (\exists x \in X. z \leq x)) \)

using atom-Sup-iff by auto

also have \( \ldots = (z \in (\bigcup x \in X. \text{atom-map } x)) \)

using atom-map-def by auto

finally have \( (z \in \text{atom-map} (\bigsqcup X)) = (z \in (\bigcup x \in X. \text{atom-map } x)) \)

by blast

thus \( \text{thesis} \)

by blast

qed

lemma atom-map-Sup-pres-var: \( \text{atom-map} \circ \text{Sup} = \text{Sup} \circ (\cdot) \text{ atom-map} \)

unfolding fun-eq_iff comp_def by (simp add: atom-map-Sup-pres)

For Inf-preservation, it is important that Infs are restricted to homomorphic images; hence they need to be pushed into the set of all atoms.

lemma atom-map-Inf-pres: \( \text{atom-map} (\bigcap X) = \text{Collect atom} \cap (\bigcap x \in X. \text{atom-map } x) \)

proof -

have \( \text{atom-map} (\bigcap X) = \text{atom-map} (- (\bigcup x \in X. \neg x)) \)

by (smt Collect-cong SUP-le_iff atom-map_def compl_le_compl_iff compl_le_swap1 le-Inf_iff)

also have \( \ldots = \text{Collect atom} - \text{atom-map} (\bigcup x \in X. \neg x) \)

using atom-map-compl-pres by blast

also have \( \ldots = \text{Collect atom} - (\bigcup x \in X. \text{atom-map } (\neg x)) \)

by (simp add: atom-map-Sup-pres)

also have \( \ldots = \text{Collect atom} - (\bigcup x \in X. \text{Collect atom} - \text{atom-map } x) \)

using atom-map-compl-pres by blast

also have \( \ldots = \text{Collect atom} \cap (\bigcap x \in X. \text{atom-map } x) \)

by blast

finally show \( \text{thesis} \).

qed
end

It follows that homomorphic images of complete boolean algebras under atom-map form complete boolean algebras.

**instantiation** atoms :: (complete-boolean-algebra-alt) complete-boolean-algebra-alt

**begin**

**lift-definition** Inf-atoms :: 'a::complete-boolean-algebra-alt atoms set ⇒ 'a::complete-boolean-algebra-alt atoms is λX. Abs-atoms (Collect atom ∩ Inter (('') Rep-atoms X)).

**lift-definition** Sup-atoms :: 'a::complete-boolean-algebra-alt atoms set ⇒ 'a::complete-boolean-algebra-alt atoms is λX. Abs-atoms (Union (('') Rep-atoms X)).

**instance**

- apply (intro-classes; transfer)
- apply (metis (no-types, hide-lams) Abs-atoms-inverse image-iff inf-le1 le-Inf-iff le-infI2 order-refl rangeI surj-atom-map)
- apply (metis (no-types, lifting) Abs-atoms-inverse Int-subset-iff Rep-atoms Sup-upper atom-map-atoms inf-le1 le-INF-iff rangeI surj-atom-map)
- apply (metis Abs-atoms-inverse Rep-atoms SUP-least SUP-upper Sup-upper atom-map-atoms rangeI surj-atom-map)
- apply (metis Abs-atoms-inverse Rep-atoms SUP-least Sup-upper atom-map-atoms rangeI surj-atom-map)
- by simp-all

**end**

Once more, properties proved above can now be restricted to at-map.

**lemma** surj-at-map-var: at-map ◦ Sup ◦ Rep-atoms = (id::'a::complete-boolean-algebra-alt atoms ⇒ 'a atoms)
- unfolding at-map-def comp-def fun-eq-iff id-def by (metis Rep-atoms Rep-atoms-inverse Sup-upper atom-map-atoms surj-atom-map)

**lemma** surj-at-map: surj (at-map::'a::complete-boolean-algebra-alt ⇒ 'a atoms)
- unfolding surj-def at-map-def comp-def by (metis Rep-atoms Rep-atoms-inverse image-iff)

**lemma** at-map-Sup-pres: at-map ◦ Sup = Sup ◦ (') (at-map::'a::complete-boolean-algebra-alt ⇒ 'a atoms)
- unfolding fun-eq-iff at-map-def comp-def atom-map-Sup-pres by (smt Abs-atoms-inverse Sup.SUP-cong Sup-atoms.transfer UN-extend-simps(10) rangeI)

**lemma** at-map-Sup-pres-var: at-map (⨆ X) = (⨆ (x::'a::complete-boolean-algebra-alt) ∈ X. (at-map x))
- using at-map-Sup-pres comp-eq-elim by blast

**lemma** at-map-Inf-pres: at-map (⨂ X) = Abs-atoms (Collect atom ∩ (⨂ x ∈ X. (Rep-atoms (at-map (x::'a::complete-boolean-algebra-alt)))))

53
unfolding at-map-def comp-def by (metis (no-types, lifting) Abs-atoms-inverse Sup.SUP-cong atom-map-Inf-pres rangeI)

lemma at-map-Inf-pres-var: at-map ∘ Inf = Inf ∘ (') (at-map::'a::complete-boolean-algebra-alt ⇒ 'a atoms)
  unfolding fun-eq-iff comp-def by (metis Inf-atoms.abs-eq at-map-Inf-pres image-image)

Finally, on complete atomic boolean algebras (CABAs), at-map is an isomorphism, that is, a bijection that preserves the complete boolean algebra operations. Thus every CABA is isomorphic to a powerset boolean algebra and every powerset boolean algebra is a CABA. The bijective pair is given by at-map and Sup (defined on the powerset algebra). This theorem is a little version of Stone’s theorem. In the general case, ultrafilters play the role of atoms.

lemma Sup ◦ atom-map = (id::'a::complete-atomic-boolean-algebra ⇒ 'a)
  unfolding fun-eq-iff comp-def id-def by (metis Union-upper atom-map-atoms inj-atom-map inj-def rangeI surj-atom-map)

lemma inj-at-map-var: Sup ◦ Rep-atoms ◦ at-map = (id ::'a::complete-atomic-boolean-algebra ⇒ 'a)
  unfolding at-map-def comp-def fun-eq-iff id-def by (metis Abs-atoms-inverse Union-upper atom-map-atoms inj-atom-map inj-def rangeI surj-atom-map)

lemma bij-at-map: bij (at-map::'a::complete-atomic-boolean-algebra ⇒ 'a atoms)
  unfolding bij-def by (simp add: inj-at-map surj-at-map)

instance atoms :: (complete-atomic-boolean-algebra) complete-atomic-boolean-algebra.. A full consideration of Stone duality is left for future work.

6 Galois Connections

theory Galois-Connections
  imports Order-Lattice-Props
begin

6.1 Definitions and Basic Properties

The approach follows the Compendium of Continuous Lattices [3], without attempting completeness. First, left and right adjoints of a Galois connection are defined.

definition adj :: ('a::ord ⇒ 'b::ord) ⇒ ('b ⇒ 'a) ⇒ bool (infix1 ⊣ 70) where
  (f ⊣ g) = (∀ x y. (f x ≤ y) = (x ≤ g y))
**Definition** \( \text{ladj} \ (g :: 'a :: \text{Inf} \Rightarrow 'b :: \text{ord}) = (\lambda x. \bigcap \{ y. \ x \leq g y \}) \)

**Definition** \( \text{radj} \ (f :: 'a :: \text{Sup} \Rightarrow 'b :: \text{ord}) = (\lambda y. \bigcup \{ x. \ f x \leq y \}) \)

**Lemma** \( \text{ladj-radj-dual} \):

- **Fixes** \( f :: 'a :: \text{complete-lattice-with-dual} \Rightarrow 'b :: \text{ord-with-dual} \)
- **Shows** \( \text{ladj} f x = \partial (\text{radj} (\partial_F f) (\partial x)) \)

**Proof**

- **Have** \( \text{ladj} f x = \partial (\bigcup (\partial \cdot \{ y. \ \partial (f y) \leq \partial x \})) \)
  - **Unfolding** \( \text{ladj-def} \) by (metis \( \text{no-types, lifting} \) \( \text{Collect-cong Inf-dual-var} \)
  - **Dual-dual-ord dual-iff**
  - **Also have** \( \ldots = \partial (\bigcup \{\partial y | y. \ \partial (f y) \leq \partial x \}) \)
  - **By** (simp add: setcompr-eq-image)
  - **Ultimately show** ?thesis
    - **Unfolding** \( \text{ladj-def} \ \text{radj-def} \ \text{map-dual-def} \ \text{comp-def} \)
    - **By** (smt \( \text{Collect-cong} \ \text{invol-dual-var} \))
  - **Qed**

**Lemma** \( \text{radj-ladj-dual} \):

- **Fixes** \( f :: 'a :: \text{complete-lattice-with-dual} \Rightarrow 'b :: \text{ord-with-dual} \)
- **Shows** \( \text{radj} f x = \partial (\text{ladj} (\partial_F f) (\partial x)) \)
  - **By** (metis \( \text{fun-dual5} \ \text{invol-dual-var} \))

**Lemma** \( \text{ladj-prop} \):

- **Fixes** \( g :: 'b :: \text{Inf} \Rightarrow 'a :: \text{ord-with-dual} \)
- **Shows** \( \text{ladj} g = \text{Inf} \circ (-') \ g \circ \uparrow \)
  - **Unfolding** \( \text{ladj-prop} \)

**Lemma** \( \text{radj-prop} \):

- **Fixes** \( f :: 'b :: \text{Sup} \Rightarrow 'a :: \text{ord} \)
- **Shows** \( \text{radj} f = \text{Sup} \circ (-') \ f \circ \downarrow \)
  - **Unfolding** \( \text{radj-prop} \)

The first set of properties holds without any sort assumptions.

**Lemma** \( \text{adj-iso1} \): \( f \dashv g \Rightarrow \text{mono} f \)
  - **Unfolding** \( \text{adj-iso1} \)

**Lemma** \( \text{adj-iso2} \): \( f \dashv g \Rightarrow \text{mono} g \)
  - **Unfolding** \( \text{adj-iso2} \)

**Lemma** \( \text{adj-comp} \): \( f \dashv g \Rightarrow \text{adj} h k \Rightarrow (f \circ h) \dashv (k \circ g) \)
  - **By** (simp add: \( \text{adj-def} \))

**Lemma** \( \text{adj-dual} \):

- **Fixes** \( f :: 'a :: \text{ord-with-dual} \Rightarrow 'b :: \text{ord-with-dual} \)
- **Shows** \( f \dashv g = (\partial_F g) \dashv (\partial_F f) \)
  - **Unfolding** \( \text{adj-dual} \)

55
6.2 Properties for (Pre)Orders

The next set of properties holds in preorders or orders.

**lemma** adj-cancel1:

fixes \( f :: \text{preorder} \Rightarrow \text{ord} \)
shows \( f \vdash g \implies f \circ g \leq \text{id} \)
by (simp add: adj-def le-funI)

**lemma** adj-cancel2:

fixes \( f :: \text{ord} \Rightarrow \text{preorder} \)
shows \( f \vdash g \implies \text{id} \leq g \circ f \)
by (simp add: adj-def eq-iff le-funI)

**lemma** adj-prop:

fixes \( f :: \text{preorder} \Rightarrow \text{preorder} \)
shows \( f \vdash g \implies f \circ g \leq g \circ f \)
using adj-cancel1 adj-cancel2 order-trans by blast

**lemma** adj-cancel-eq1:

fixes \( f :: \text{preorder} \Rightarrow \text{ord} \)
shows \( f \vdash g \implies f \circ g \circ f = f \)
unfolding adj-def comp-def fun-eq-iff by (meson eq-iff order-refl order-trans)

**lemma** adj-cancel-eq2:

fixes \( f :: \text{ord} \Rightarrow \text{preorder} \)
shows \( f \vdash g \implies g \circ f \circ g = g \)
unfolding adj-def comp-def fun-eq-iff by (meson eq-iff order-refl order-trans)

**lemma** adj-idem1:

fixes \( f :: \text{ord} \Rightarrow \text{ord} \)
shows \( f \vdash g \implies (f \circ g) \circ (f \circ g) = f \circ g \)
by (simp add: adj-cancel-eq1 rewriteL-comp-comp)

**lemma** adj-idem2:

fixes \( f :: \text{ord} \Rightarrow \text{ord} \)
shows \( f \vdash g \implies (g \circ f) \circ (g \circ f) = g \circ f \)
by (simp add: adj-cancel-eq2 rewriteL-comp-comp)

**lemma** adj-iso3:

fixes \( f :: \text{ord} \Rightarrow \text{ord} \)
shows \( f \vdash g \implies \text{mono} (f \circ g) \)
by (simp add: adj-iso1 adj-iso2 monoD monoI)

**lemma** adj-iso4:

fixes \( f :: \text{ord} \Rightarrow \text{ord} \)
shows \( f \vdash g \implies \text{mono} (g \circ f) \)
by (simp add: adj-iso1 adj-iso2 monoD monoI)

**lemma** adj-canc1:
fixes \( f :: 'a::order \Rightarrow 'b::ord \)
shows \( f \circ g \equiv ((f \circ g) \ x = (f \circ g) \ y \rightarrow g \ x = g \ y) \)
unfolding \( \text{adj-def comp-def by (metis eq-iff)} \)

lemma \( \text{adj-canc2} \):
fixes \( f :: 'a::ord \Rightarrow 'b::order \)
shows \( f \vdash \ g \equiv ((g \circ f) \ x = (g \circ f) \ y \rightarrow f \ x = f \ y) \)
unfolding \( \text{adj-def comp-def by (metis eq-iff)} \)

lemma \( \text{adj-sur-inv} \):
fixes \( f :: 'a::preorder \Rightarrow 'b::order \)
shows \( f \vdash g \equiv ((\text{surj } f) = (f \circ g = \text{id})) \)
unfolding \( \text{adj-def surj-def comp-def by (metis eq-id-iff eq-iff order-refl order-trans)} \)

lemma \( \text{adj-surj-inj} \):
fixes \( f :: 'a::order \Rightarrow 'b::order \)
shows \( f \vdash \ g \equiv ((\text{inj } f) = (g \circ f = \text{id})) \)
unfolding \( \text{adj-def inj-def surj-def by (metis eq-iff order-trans)} \)

lemma \( \text{adj-inj-inv} \):
fixes \( f :: 'a::preorder \Rightarrow 'b::order \)
shows \( f \vdash \ g \equiv ((\text{inj } f) = (g \circ f = \text{id})) \)
by \( \text{(metis adj-cancel-eq1 eq-id-iff inj-def o-apply)} \)

lemma \( \text{adj-inj-surj} \):
fixes \( f :: 'a::order \Rightarrow 'b::order \)
shows \( f \vdash \ g \equiv ((\text{inj } f) = (\text{surj } g)) \)
unfolding \( \text{adj-def inj-def surj-def by (metis eq-iff order-trans)} \)

lemma \( \text{surj-id-the-inv} \): surj \( f \Rightarrow g \circ f = \text{id} \Rightarrow g = \text{the-inv } f \)
by \( \text{(metis comp-apply id-apply inj-on-id inj-on-imageI2 surj-fun-eq the-inv-f-f)} \)

lemma \( \text{inj-id-the-inv} \): inj \( f \Rightarrow f \circ g = \text{id} \Rightarrow f = \text{the-inv } g \)
proof –
assume \( a1 \colon \text{inj } f \)
assume \( f \circ g = \text{id} \)
hence \( \forall x. \text{the-inv } g \ x = f \ x \)
using \( a1 \) by \( \text{(metis (no-types) comp-apply eq-id-iff inj-on-id inj-on-imageI2 the-inv-f-f)} \)
thus \( \text{thesis} \)
by \( \text{presburger} \)
qed

6.3 Properties for Complete Lattices

The next laws state that a function between complete lattices preserves infs if and only if it has a lower adjoint.

lemma \( \text{radj-Inf-pres} \):
fixes \( g :: 'b::complete-lattice \Rightarrow 'a::complete-lattice \)
shows \((\exists f. f \vdash g) \implies \text{Inf-pres } g\)
apply (rule antisym, simp-all add: le-fun-def adj-def, safe)
apply (meson INF-greatest Inf-lower dual-order.refl dual-order.trans)
by (meson Inf-greatest dual-order.refl le-INF-iff)

lemma ladj-Sup-pres:
fixes \(f::'a::complete-lattice-with-dual\)\(\Rightarrow 'b::complete-lattice-with-dual\)
shows \((\exists g. f \vdash g) \implies \text{Sup-pres } f\)
using Sup-pres-map-dual-var adj-dual radj-Inf-pres by blast

lemma radj-adj:
fixes \(f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice\)
shows \(f \vdash g \iff g = (\text{radj } f)\)
unfolding adj-def radj-def by (metis (mono-tags, lifting) cSup-eq-maximum eq-iff mem-Collect-eq)

lemma ladj-adj:
fixes \(g :: 'b::complete-lattice-with-dual \Rightarrow 'a::complete-lattice-with-dual\)
shows \(f \vdash g \iff f = (\text{ladj } g)\)
unfolding adj-def ladj-def by (metis (no-types, lifting) cInf-eq-minimum eq-iff mem-Collect-eq)

lemma Inf-pres-radj-aux:
fixes \(g :: 'b::complete-lattice \Rightarrow 'a::complete-lattice\)
shows \(\text{Inf-pres } g \iff (\text{ladj } g) \vdash g\)
proof
  assume \(a :: \text{Inf-pres } g\)
  {fix \(x y\)
   assume \(b :: \text{ladj } g \leq y\)
   hence \(g \leq y\) by (simp add: Inf-subdistl-iso a monoD)
   hence \(\bigcap \{y. x \leq y\} \leq g\)
   hence \(x \leq g\)
   using dual-order.trans le-Inf-iff by blast
   hence \(\text{ladj } g \leq y \iff x \leq g\)
   by simp}
thus \(?thesis\)
unfolding adj-def ladj-def by (meson CollectI Inf-lower)
qed

lemma Sup-pres-ladj-aux:
fixes \(f :: 'a::complete-lattice-with-dual \Rightarrow 'b::complete-lattice-with-dual\)
shows \(\text{Sup-pres } f \iff f \vdash (\text{radj } f)\)
by (metis (no-types, hide-lams) Inf-pres-radj-aux Sup-pres-map-dual-var adj-dual fun-dual5 map-dual-def radj-adj)

lemma Inf-pres-radj:
fixes \(g :: 'b::complete-lattice \Rightarrow 'a::complete-lattice\)
shows \( \text{Inf-pres } g \implies (\exists f. f \vdash g) \)
using \( \text{Inf-pres-radj-aux} \) by fastforce

lemma \( \text{Sup-pres-ladj} \):
fixes \( f :: 'a::complete-lattice-with-dual \implies 'b::complete-lattice-with-dual \)
shows \( \text{Sup-pres } f \implies (\exists g. f \vdash g) \)
using \( \text{Sup-pres-ladj-aux} \) by fastforce

lemma \( \text{Inf-pres-upper-adj-eq} \):
fixes \( g :: 'b::complete-lattice \implies 'a::complete-lattice \)
shows \( \text{Inf-pres } g = (\exists f. f \vdash g) \)
using \( \text{radj-Inf-pres Inf-pres-radj} \) by blast

lemma \( \text{Sup-pres-ladj-eq} \):
fixes \( f :: 'a::complete-lattice-with-dual \implies 'b::complete-lattice-with-dual \)
shows \( \text{Sup-pres } f = (\exists g. f \vdash g) \)
using \( \text{Sup-pres-ladj ladj-Sup-pres} \) by blast

lemma \( \text{Sup-downset-adj} \): \( \text{Sup} :: 'a::complete-lattice set \implies 'a \)
unfolding adj-def downset-prop Sup-le-iff by force

lemma \( \text{Sup-downset-adj-var} \): \( \text{Sup} (X :: 'a::complete-lattice set) \leq y) = (X \subseteq \downarrow y) \)
using \( \text{Sup-downset-adj adj-def} \) by auto

Once again many statements arise by duality, which Isabelle usually picks up.

end

7 Fixpoint Fusion

theory \( \text{Fixpoint-Fusion} \)
  imports \( \text{Galois-Connections} \)
begin

Least and greatest fixpoint fusion laws for adjoints in a Galois connection, including some variants, are proved in this section. Again, the laws for least and greatest fixpoints are duals.

lemma \( \text{lfp-Fix} \):
fixes \( f :: 'a::complete-lattice-with-dual \implies 'a \)
shows \( \text{mono } f \implies \text{lfp } f = \bigcap (\text{Fix } f) \)
unfolding \( \text{lfp-def Fix-def} \)
apply (rule antisym)
apply (simp add: Collect-mono Inf-superset-mono)
by (metis \( \text{mono-tags} \) \( \text{Inf-lower lfp-def lfp-unfold mem-Coll}\text{ect-eq} \))

lemma \( \text{gfp-Fix} \):
fixes \( f :: 'a::complete-lattice-with-dual \implies 'a \)
shows $\text{mono } f \Rightarrow \text{gfp } f = \bigsqcup (\text{Fix } f)$
by $(\text{simp add: iso-map-dual gfp-to-lfp lfp-Fix Fix-map-dual-var Sup-to-Inf-var})$

**lemma** gfp-little-fusion:

**fixes** $f :: 'a::complete-lattice \Rightarrow 'a$
and $g :: 'b::complete-lattice \Rightarrow 'b$

**assumes** $\text{mono } f$

**assumes** $h \circ f \leq g \circ h$

**shows** $h (\text{gfp } f) \leq \text{gfp } g$

**proof**

- have $h (f (\text{gfp } f)) \leq g (h (\text{gfp } f))$
  using $\text{assms(2) le-funD}$ by fastforce

hence $h (\text{gfp } f) \leq g (h (\text{gfp } f))$
by $(\text{simp add: assms(1) gfp-fixpoint})$

thus $h (\text{gfp } f) \leq \text{gfp } g$
by $(\text{simp add: gfp-upperbound})$

**qed**

**lemma** lfp-little-fusion:

**fixes** $f :: 'a::complete-lattice-with-dual \Rightarrow 'a$
and $g :: 'b::complete-lattice-with-dual \Rightarrow 'b$

**assumes** $\text{mono } f$

**assumes** $g \circ h \leq h \circ f$

**shows** $\text{lfp } g \leq h (\text{lfp } f)$

**proof**

- have $a :: \text{mono } (\text{map-dual } f)$
  by $(\text{simp add: assms iso-map-dual})$

  have $\text{map-dual } h \circ \text{map-dual } f \leq \text{map-dual } g \circ \text{map-dual } h$
  by $(\text{metis assms map-dual-anti map-dual-func1})$

  thus $\text{?thesis}$
  by $(\text{metis a comp-eq-elim dual-dual-ord fun-dual1 gfp-little-fusion lfp-dual-var map-dual-def})$

**qed**

**lemma** gfp-fusion:

**fixes** $f :: 'a::complete-lattice \Rightarrow 'a$
and $g :: 'b::complete-lattice \Rightarrow 'b$

**assumes** $\exists f :: 'a::complete-lattice \Rightarrow 'b$
and $\text{mono } f$
and $\text{mono } g$
and $h \circ f = g \circ h$

**shows** $h (\text{gfp } f) = \text{gfp } g$

**proof**

- have $h (\text{gfp } f) \leq \text{gfp } g$
  by $(\text{simp add: assms(2) assms(4) gfp-little-fusion})$

  obtain $hl$ where $\text{conn: } \forall x y. (hl x \leq y) \iff (x \leq h y)$
  using $\text{assms adj-def}$ by blast

  have $hl \circ g \leq hl \circ g \circ h \circ hl$
  by $(\text{simp add: le-fun-def, meson conn assms(3) monoE order-refl order-trans})$
also have \( \ldots = h \circ \circ f \circ h \)
  by \((\text{simp add: assms(4)} \text{ comp-assoc})\)

finally have \( h \circ g \leq f \circ h \)
  by \((\text{simp add: le-fun-def, metis conn inf.coboundedI2 inf.orderE order-refl})\)

hence \( h \circ (gfp g) \leq f \circ (h \circ (gfp g)) \)
  by \((\text{metis comp-eq-dest-lhs gfp-unfold assms(3) le-fun-def})\)

hence \( h \circ (gfp g) \leq gfp f \)
  by \((\text{simp add: gfp-upperbound})\)

thus \(?thesis\)
  by \((\text{simp add: conn})\)

lemma \(lfp-fusion\):
  \[
  \begin{align*}
  \text{fixes} & \quad f :: 'a::complete-lattice-with-dual \Rightarrow 'a \\
  \text{and} & \quad g :: 'b::complete-lattice-with-dual \Rightarrow 'b \\
  \text{assumes} & \quad \exists f. h \circ f = g \circ h \\
  \text{and} & \quad \text{mono} f \\
  \text{and} & \quad \text{mono} g \\
  \text{and} & \quad h \circ f = g \circ h \\
  \text{shows} & \quad h \circ lfp f = lfp g \\
  \text{proof} & \quad \text{by} \quad (\text{simp add: a eq-iff})
  \end{align*}
  \]

lemma \(gfp-fusion-inf-pres\):
  \[
  \begin{align*}
  \text{fixes} & \quad f :: 'a::complete-lattice \Rightarrow 'a \\
  \text{and} & \quad g :: 'b::complete-lattice \Rightarrow 'b \\
  \text{assumes} & \quad \text{Inf-pres} h \\
  \text{and} & \quad \text{mono} f \\
  \text{and} & \quad \text{mono} g \\
  \text{and} & \quad h \circ f = g \circ h \\
  \text{shows} & \quad h \circ gfp f = gfp g \\
  \text{by} & \quad (\text{simp add: Inf-pres-radj assms gfp-fusion})
  \end{align*}
  \]

lemma \(lfp-fusion-sup-pres\):
  \[
  \begin{align*}
  \text{fixes} & \quad f :: 'a::complete-lattice-with-dual \Rightarrow 'a \\
  \text{and} & \quad g :: 'b::complete-lattice-with-dual \Rightarrow 'b \\
  \text{shows} & \quad h \circ lfp f = lfp g \\
  \text{by} & \quad (\text{simp add: Inf-pres-radj assms gfp-fusion})
  \end{align*}
  \]
assumes $\text{Sup-pres } h$
and $\text{mono } f$
and $\text{mono } g$
and $h \circ f = g \circ h$
shows $h \ (\text{lfp } f) = \text{lfp } g$
by (simp add: Sup-pres-ladj assms lfp-fusion)

The following facts are useful for the semantics of isotone predicate transformers. A dual statement for least fixpoints can be proved, but is not spelled out here.

**lemma** $k$-adju:
fixes $k :: 'a::order \Rightarrow 'b::complete-lattice$
shows $\exists F. \forall x. (F::'b \Rightarrow 'a \Rightarrow 'b) \cdot (\lambda k. k y)$
by (force intro!: fun-eq-iff Inf-pres-radj)

**lemma** $k$-adju-var: $\exists F. \forall x. \forall f::'a::order \Rightarrow 'b::complete-lattice. (F x \leq f) = (x \leq (\lambda k. k y) f)$
using $k$-adju unfolding adj-def by simp

**lemma** gfp-fusion-var:
fixes $F :: (a::order \Rightarrow b::complete-lattice) \Rightarrow 'a \Rightarrow 'b$
and $g :: 'b \Rightarrow 'b$
assumes $\text{mono } F$
and $\text{mono } g$
and $\forall h. F \ h \ x = g \ (h \ x)$
shows $\text{gfp } F \ x = \text{gfp } g$
by (metis (no-types, hide-lams) assms eq-iff gfp-fixpoint gfp-upperbound k-adju-vars monoE order-refl)

This time, Isabelle is picking up dualities rather inconsistently.

end

8 Closure and Co-Closure Operators

theory Closure-Operators
  imports Galois-Connections
begin

8.1 Closure Operators

Closure and coclosure operators in orders and complete lattices are defined in this section, and some basic properties are proved. Isabelle infers the appropriate types. Facts are taken mainly from the Compendium of Continuous Lattices [3] and Rosenthal's book on quantales [10].

definition clop :: ('a::order) \Rightarrow bool where
clop f = (id \leq f \land \text{mono } f \land f \circ f \leq f)
lemma clop-extensive: \( \text{clop } f \Rightarrow \text{id } \leq f \)
by (simp add: clop-def)

lemma clop-extensive-var: \( \text{clop } f \Rightarrow x \leq f x \)
by (simp add: clop-def le-fun-def)

lemma clop-iso: \( \text{clop } f \Rightarrow \text{mono } f \)
by (simp add: clop-def)

lemma clop-iso-var: \( \text{clop } f \Rightarrow x \leq y \Rightarrow f x \leq f y \)
by (simp add: clop-def mono-def)

lemma clop-idem: \( \text{clop } f \Rightarrow f \circ f = f \)
by (simp add: antisym clop-def le-fun-def)

lemma clop-idem-var: \( \text{clop } f \Rightarrow f (f x) = f x \)
by (simp add: clop-idem retraction-prop)

lemma clop-Inf-closed-var:
fixes \( f \) :: 'a::complete-lattice \Rightarrow 'a
shows \( \text{clop } f \Rightarrow f \circ\ Inf \circ (\prime) f = \text{Inf }\circ (\prime) f \)
unfolding clop-def mono-def comp-def le-fun-def
by (metis (mono-tags, lifting) antisym id-apply le-INF-iff order-refl)

lemma clop-top:
fixes \( f \) :: 'a::complete-lattice \Rightarrow 'a
shows \( \text{clop } f \Rightarrow f \top = \top \)
by (simp add: clop-extensive-var top.extremum-uniqueI)

lemma clop (f::'a::complete-lattice \Rightarrow 'a) \( \Rightarrow f (\bigsqcup x \in X. f x) = (\bigsqcup x \in X. f x) \)
oops

lemma clop (f::'a::complete-lattice \Rightarrow 'a) \( \Rightarrow f (f x \sqcup f y) = f x \sqcup f y \)
oops

lemma clop (f::'a::complete-lattice \Rightarrow 'a) \( \Rightarrow f \bot = \bot \)
oops

lemma clop (f::'a set \Rightarrow 'a set) \( \Rightarrow f (\bigsqcup x \in X. f x) = (\bigsqcup x \in X. f x) \)
oops

lemma clop (f::'a set \Rightarrow 'a set) \( \Rightarrow f (f x \sqcup f y) = f x \sqcup f y \)
oops

lemma clop (f::'a set \Rightarrow 'a set) \( \Rightarrow f \bot = \bot \)
lemma clop-closure: \( \text{clop} \ f \implies (x \in \text{range} \ f) = (f \ x = x) \)
by (simp add: clop-idem retraction-prop)

lemma clop-closure-set: \( \text{clop} \ f \implies \text{range} \ f = \text{Fix} \ f \)
by (simp add: clop-Fix-range)

lemma clop-closure-prop: \( (\text{clop} :: ('a::complete-lattice-with-dual \Rightarrow 'a) \Rightarrow \text{bool}) (\text{Inf} \circ \uparrow) \)
by (simp add: clop-def mono-def)

lemma clop-closure-prop-var: \( \text{clop} (\lambda x :: 'a::complete-lattice. d\{y. x \leq y\}) \)
unfolding clop-def comp-def le-fun-def mono-def
by (simp add: Inf-lower le-Inf-iff)

lemma clop-alt: \( (\text{clop} \ f) = (\forall x \ y. x \leq f y \iff f x \leq f y) \)
unfolding clop-def mono-def le-fun-def comp-def id-def
by (meson dual-order refl order-trans)

Finally it is shown that adjoints in a Galois connection yield closure operators.

lemma clop-adj:
fixes \( f :: 'a::order \Rightarrow 'b::order \)
shows \( f \sqdash g = \implies \text{clop} (g \circ f) \)
by (simp add: adj-cancel2 adj-idem2 adj-iso4 clop-def)

Closure operators are monads for posets, and monads arise from adjunctions.
This fact is not formalised at this point. But here is the first step: every function can be decomposed into a surjection followed by an injection.

definition surj-on \( f \ Y = (\forall y \in Y. \exists x. y = f x) \)

lemma surj-surj-on: \( \text{surj} \ f \implies \text{surj-on} \ f \ Y \)
by (simp add: surjD surj-on-def)

lemma fun-surj-inj: \( \exists g h. f = g \circ h \land \text{surj-on} \ h \ (\text{range} \ f) \land \text{inj-on} \ g \ (\text{range} \ f) \)
proof -
  obtain \( h \) where \( a: \forall x. f x = h x \)
  by blast
  then have \( \text{surj-on} \ h \ (\text{range} \ f) \)
  by (metis (mono-tags, lifting) imageE surj-on-def)
  then show \( \text{thesis} \)
    unfolding inj-on-def surj-on-def fun-eq-iff using \( a \)
by auto
qed

Connections between downsets, upsets and closure operators are outlined next.

lemma preorder-clop: \( \text{clop} (\downarrow::'a::preorder \text{ set} \Rightarrow 'a \text{ set}) \)
by (simp add: clop-def downset-set-ext downset-set-iso)
lemma clop-preorder-aux: clop f \Rightarrow (x \in f \{y\} \iff f \{x\} \subseteq f \{y\})
by (simp add: clop-alt)

lemma clop-preorder: clop f \Rightarrow class.preorder (\lambda x y. f \{x\} \subseteq f \{y\}) (\lambda x y. f \{x\} \subset f \{y\})
unfolding clop-def mono-def le-fun-def id-def comp-def by standard (auto simp: subset-not-subset-eq)

lemma preorder-clop-dual: clop (\forall::'a::preorder-with-dual set \Rightarrow 'a set)
by (simp add: clop-def upset-set-anti upset-set-ext)

The closed elements of any closure operator over a complete lattice form an Inf-closed set (a Moore family).

lemma clop-Inf-closed:
fixes f :: 'a::complete-lattice set
shows clop f \Rightarrow Inf-closed-set (Fix f)
unfolding clop-def Inf-closed-set-def mono-def le-fun-def comp-def id-def Fix-def
by (smt Inf-greatest Inf-lower antisym mem-Collect-eq subsetCE)

lemma clop-top-Fix:
fixes f :: 'a::complete-lattice set
shows clop f \Rightarrow \top \in Fix f
by (simp add: clop-Fix-range clop-closure clop-top)

Conversely, every Inf-closed subset of a complete lattice is the set of fixpoints of some closure operator.

lemma Inf-closed-clop:
fixes X :: 'a::complete-lattice set
shows Inf-closed-set X \Rightarrow clop (\lambda y. \bigcap \{x \in X. y \leq x\})
by (smt Collect-mono-iff Inf-superset-mono clop-alt dual-order trans le-Inf-iff mem-Collect-eq)

lemma Inf-closed-clop-var:
fixes X :: 'a::complete-lattice set
shows clop f \Rightarrow \forall x \in X. x \in range f \Rightarrow \bigcap X \in range f
by (metis Inf-closed-set-def clop-Fix-range clop-Inf-closed subsetI)

It is well known that downsets and upsets over an ordering form subalgebras of the complete powerset lattice.

typedef (overloaded) 'a downsets = range (\forall::'a::order set \Rightarrow 'a set)
by fastforce

setup-lifting type-definition-downsets

typedef (overloaded) 'a upsets = range (\forall::'a::order set \Rightarrow 'a set)
by fastforce

65
setup-lifting type-definition-upsets

instantiation downsets :: (order) Inf-lattice
begin

lift-definition Inf-downsets :: 'a downsets set ⇒ 'a downsets is Abs-downsets ◦ Inf ◦ (‘') Rep-downsets.

lift-definition less-eq-downsets :: 'a downsets ⇒ 'a downsets ⇒ bool is λX Y. Rep-downsets X ⊆ Rep-downsets Y.

lift-definition less-downsets :: 'a downsets ⇒ 'a downsets ⇒ bool is λX Y. Rep-downsets X ⊂ Rep-downsets Y.

instance
apply intro-classes
  apply (transfer, simp)
  apply (transfer, blast)
  apply (simp add: Closure-Operators.less-eq-downsets.abs-eq Rep-downsets-inject)
  apply (transfer, smt Abs-downsets-inverse INF-lower Inf-closed-clop-var Rep-downsets image-iff o-def preorder-clop)
  by (transfer (smt comp-def Abs-downsets-inverse Inf-closed-clop-var Rep-downsets image-iff le-INF-iff preorder-clop))
end

instantiation upsets :: (order-with-dual) Inf-lattice
begin

lift-definition Inf-upsets :: 'a upsets set ⇒ 'a upsets is Abs-upsets ◦ Inf ◦ (‘') Rep-upsets.

lift-definition less-eq-upsets :: 'a upsets ⇒ 'a upsets ⇒ bool is λX Y. Rep-upsets X ⊆ Rep-upsets Y.

lift-definition less-upsets :: 'a upsets ⇒ 'a upsets ⇒ bool is λX Y. Rep-upsets X ⊂ Rep-upsets Y.

instance
apply intro-classes
  apply (transfer, simp)
  apply (transfer, blast)
  apply (simp add: Closure-Operators.less-eq-upsets.abs-eq Rep-upsets-inject)
  apply (transfer, smt Abs-upsets-inverse INF-lower Inf-closed-clop-var Rep-upsets comp-apply image-iff preorder-clop-dual)
  by (transfer (smt comp-def Abs-upsets-inverse Inf-closed-clop-var Inter-iff Rep-upsets image-iff preorder-clop-dual subsetCE subsetI))
end
It has already been shown in the section on representations that the map ds, which maps elements of the order to its downset, is an order embedding. However, the duality between the underlying ordering and the lattices of up-and down-closed sets as categories can probably not be expressed, as there is no easy access to contravariant functors.

### 8.2 Co-Closure Operators

Next, the co-closure (or kernel) operation satisfies dual laws.

**definition** \( \text{coclop} :: ('a::order ⇒ 'a::order) ⇒ bool \) where
\[
coclop\ f = (f \leq id \land \text{mono } f \land f \leq f \circ f)
\]

**lemma** \( \text{coclop-dual} : (\text{coclop} :: ('a::order-with-dual ⇒ 'a) ⇒ bool) = \text{clop} \circ \partial_F \)

**unfolding** \( \text{coclop-def clop-def id-def mono-def map-dual-def comp-def fun-eq-iff le-fun-def} \)

**by** (metis invol-dual-var ord-dual)

**lemma** \( \text{coclop-dual-var} : \) fixes \( f :: 'a::order-with-dual ⇒ 'a \)
shows \( \text{coclop } f = \text{clop} (\partial_F f) \)
by (simp add: coclop-dual)

**lemma** \( \text{clop-dual} : (\text{clop} :: ('a::order-with-dual ⇒ 'a) ⇒ bool) = \text{coclop} \circ \partial_F \)
by (simp add: coclop-dual comp-assoc map-dual-invol)

**lemma** \( \text{clop-dual-var} : \) fixes \( f :: 'a::order-with-dual ⇒ 'a \)
shows \( \text{clop } f = \text{coclop} (\partial_F f) \)
by (simp add: clop-dual)

**lemma** \( \text{coclop-coextensive} : \text{coclop } f \Rightarrow f \leq id \)
by (simp add: coclop-def)

**lemma** \( \text{coclop-coextensive-var} : \text{coclop } f \Rightarrow f \leq x \)
using \( \text{coclop-def le-funD} \) by fastforce

**lemma** \( \text{coclop-iso} : \text{coclop } f \Rightarrow \text{mono } f \)
by (simp add: coclop-def)

**lemma** \( \text{coclop-iso-var} : \text{coclop } f \Rightarrow (x \leq y \Rightarrow f x \leq f y) \)
by (simp add: coclop-iso monoD)

**lemma** \( \text{coclop-idem} : \text{coclop } f \Rightarrow f \circ f = f \)
by (simp add: antisym coclop-def le-fun-def)

**lemma** \( \text{coclop-closure} : \text{coclop } f \Rightarrow (x \in \text{range } f) = (f x = x) \)
by (simp add: coclop-idem retraction-prop)
lemma coclop-Fix-range: coclop f \Rightarrow (\text{Fix } f = \text{range } f)
by (simp add: coclop-idem retraction-prop-fix)

lemma coclop-idem-var: coclop f \Rightarrow f (f x) = f x
by (simp add: coclop-idem retraction-prop)

lemma coclop-Sup-closed-var:
fixes f :: 'a::complete-lattice-with-dual 
shows unfolding coclop-def mono-def comp-def le-fun-def 
by (metis (mono-tags, lifting) SUP-le-iff antisym id-apply order-refl)

lemma Sup-closed-coclop-var:
fixes X :: 'a::complete-lattice set
shows coclop f \Rightarrow \forall x \in X. x \in \text{range } f \Rightarrow \text{\oplus}_X \in \text{range } f
by (smt Inf.INF-id-eq Sup.SUP-cong antisym coclop-closure coclop-coextensive-var coclop-iso id-apply mono-SUP)

lemma coclop-bot:
fixes f :: 'a::complete-lattice-with-dual 
shows coclop f \Rightarrow f \bot = \bot
by (simp add: bot.extremum-uniqueI coclop-coextensive-var)

lemma coclop-coclosure: coclop f \Rightarrow f x = x \iff x \in \text{range } f
by (simp add: coclop-idem retraction-prop)

lemma coclop-coclosure-set: coclop f \Rightarrow \text{range } f = \text{Fix } f
by (simp add: coclop-idem retraction-prop-fix)

lemma coclop-coclosure-prop: (coclop::('a::complete-lattice \Rightarrow 'a) \Rightarrow \text{bool}) (Sup \circ
lemma coclop-coclosure-prop-var: coclop (\(\lambda x::'a::complete-lattice. \bigcup \{ y. y \leq x \}\))
  by (metis (mono-tags, lifting) Sup-atMost atMost-def coclop-def comp-apply eq-id-iff eq-refl mono-def)

lemma coclop-alt: (coclop f) = (\(\forall x y. f x \leq y \iff f x \leq f y\))
  unfolding coclop-def mono-def le-fun-def comp-def id-def
  by (meson dual-order refl order-trans)

lemma coclop-adj:
  fixes f :: 'a::order \rightarrow 'b::order
  shows f \circ g \Rightarrow coclop (f \circ g)
  by (simp add: adj-cancel1 adj-idem1 adj-iso3 coclop-def)

Finally, a subset of a complete lattice is Sup-closed if and only if it is the
set of fixpoints of some co-closure operator.

lemma coclop-Sup-closed:
  fixes f :: 'a::complete-lattice \Rightarrow 'a
  shows coclop f \Rightarrow Sup-closed-set (Fix f)
  unfolding coclop-def Sup-closed-set-def mono-def le-fun-def comp-def id-def Fix-def
  by (smt Sup-least Sup-upper antisym-conv mem-Collect-eq subsetCE)

lemma Sup-closed-coclop:
  fixes X :: 'a::complete-lattice set
  shows Sup-closed-set X \Rightarrow coclop (\(\lambda y. \bigcup \{ x \in X. x \leq y \}\))
  unfolding Sup-closed-set-def coclop-def mono-def le-fun-def comp-def
  apply safe
  apply (metis (no-types, lifting) Sup-least eq-id-iff mem-Collect-eq)
  apply (smt Collect-mono-iff Sup-subset-mono dual-order.trans)
  by (simp add: Collect-mono-iff Sup-subset-mono Sup-upper)

8.3 Complete Lattices of Closed Elements

The machinery developed allows showing that the closed elements in a com-
plete lattice (with respect to some closure operation) form themselves a
complete lattice.

class cl-op = ord +
  fixes cl-op :: 'a \Rightarrow 'a
  assumes clop-ext: x \leq cl-op x
  and clop-iso: x \leq y \Rightarrow cl-op x \leq cl-op y
  and clop-wtrans: cl-op (cl-op x) \leq cl-op x

class clattice-with-clop = complete-lattice + cl-op

begin
lemma \textit{clop-cl-op}: \textit{clop cl-op}
unfolding clop-def le-fun-def comp-def
by (simp add: cl-op-class clop-ext cl-op-class clop-wtrans order-class mono-def)

lemma \textit{clop-idem} [simp]: \textit{clop \circ clop = clop}
using clop-ext clop-wtrans order.antisym by auto

lemma \textit{clop-idem-var} [simp]: \textit{clop (clop x) = clop x}
by (simp add: antisym clop-ext clop-wtrans)

lemma \textit{clop-range-Fix}: \textit{range \textit{clop} = Fix \textit{clop}}
by (simp add: retraction-prop-fix)

lemma \textit{Inf-closed-cl-op-var}:
fixes \( X :: 'a \) set
shows \( \forall x \in X. \ x \in \text{range \textit{clop}} \Rightarrow \bigcap X \in \text{range \textit{clop}} \)
proof -
assume \( h: \forall x \in X. \ x \in \text{range \textit{clop}} \)
hence \( \forall x \in X. \ \text{clop x} = x \) by (simp add: retraction-prop)
hence \( \text{clop} (\bigcap X) = \bigcap X \)
by (metis Inf-lower clop-ext clop-iso dual-order antisym le-Inf-iff)
thus \( \text{thesis} \)
by (metis rangeI)
qed

lemma \textit{inf-closed-cl-op-var}: \( x \in \text{range \textit{clop}} \Rightarrow y \in \text{range \textit{clop}} \Rightarrow x \cap y \in \text{range \textit{clop}} \)
by (smt Inf-closed-cl-op-var UnI1 insert-iff insert-is-Un inf-Inf)
end

typedef \textit{(overloaded)} 'a::clattice-with-clop \textit{cl-op-im} = \text{range (clop::'a=>'a)}
by force

setup-lifting \textit{type-definition-cl-op-im}

lemma \textit{cl-op-prop} [iff]: \( (\text{clop (x \sqcup y) = clop y}) = (\text{clop (x::'a::clattice-with-clop}) \leq \text{clop y}) \)
by (smt cl-op-class clop-iso clop-ext clop-wtrans inf-sup-ord sup-sup-absorb-iff1 sup-left-commute)

lemma \textit{cl-op-prop-var} [iff]: \( (\text{clop (x \sqcup clop y) = clop y}) = (\text{clop (x::'a::clattice-with-clop}) \leq \text{clop y}) \)
by (metis cl-op-prop clattice-with-clop-cl-op-im clop-idem-var)

instantiation \textit{cl-op-im} :: \textit{(clattice-with-clop complete-lattice)
begin


This statement is perhaps less useful as it might seem, because it is difficult to make it cooperate with concrete closure operators, which one would not generally like to define within a type class. Alternatively, a sublocale statement could perhaps be given. It would also have been nice to prove this statement for Sup-lattices—this would have cut down the number of proof obligations significantly. But this would require a tighter integration of these structures. A similar statement could have been proved for co-closure operators. But this would not lead to new insights.

Next I show that for every surjective Sup-preserving function between complete lattices there is a closure operator such that the set of closed elements is isomorphic to the range of the surjection.

**lemma** **surj-Sup-pres-id:**
fixes $f :: \text{complete-lattice-with-dual} \Rightarrow \text{complete-lattice-with-dual}$
assumes \(\text{surj } f\)
and \(\text{Sup-pres } f\)
shows \(f \circ (\text{radj } f) = \text{id}\)

proof
have $f \dashv (\text{radj } f)$
  using \(\text{Sup-pres-ladj } \text{assms}(2)\) \(\text{radj-adj}\) by auto
thus \(\text{thesis}\)
  using \(\text{adj-sur-inv } \text{assms}(1)\) by blast
qed

lemma \(\text{surj-Sup-pres-inj}\):
fixes $f :: \text{complete-lattice-with-dual} \Rightarrow \text{complete-lattice-with-dual}$
assumes \(\text{surj } f\)
and \(\text{Sup-pres } f\)
shows \(\text{inj } (\text{radj } f)\)
  by (metis \(\text{assms}\) \(\text{comp-eq-dest-lhs}\) \(\text{id-apply}\) \(\text{injI}\) \(\text{surj-Sup-pres-id}\))

lemma \(\text{surj-Sup-pres-inj-on}\):
fixes $f :: \text{complete-lattice-with-dual} \Rightarrow \text{complete-lattice-with-dual}$
assumes \(\text{surj } f\)
and \(\text{Sup-pres } f\)
shows \(\text{inj-on } f\) \(\text{(range } (\text{radj } f \circ f))\)
by (smt \(\text{Sup-pres-ladj-aux}\) \(\text{adj-idem2}\) \(\text{assms}(2)\) \(\text{comp-apply}\) \(\text{inj-on-def}\) \(\text{retraction-prop}\))

lemma \(\text{surj-Sup-pres-bij-on}\):
fixes $f :: \text{complete-lattice-with-dual} \Rightarrow \text{complete-lattice-with-dual}$
assumes \(\text{surj } f\)
and \(\text{Sup-pres } f\)
shows \(\text{bij-betw } f\) \(\text{(range } (\text{radj } f \circ f))\) \(\text{UNIV}\)
unfolding \(\text{bij-betw-def}\)
apply safe
apply (simp \(\text{add}\) : \(\text{assms}(1)\) \(\text{assms}(2)\) \(\text{surj-Sup-pres-inj-on}\) \(\text{cong}\) \(\text{del}\) : \(\text{image-cong-simp}\))
apply auto
apply (metis \(\text{mono-tags}\) \(\text{UNIV-I}\) \(\text{assms}(1)\) \(\text{assms}(2)\) \(\text{comp-apply}\) \(\text{id-apply}\) \(\text{image-image}\) \(\text{surj-Sup-pres-id}\) \(\text{surj-def}\))
done

Thus the restriction of $f$ to the set of closed elements is indeed a bijection.
The final fact shows that it preserves Sups of closed elements, and hence is
an isomorphism of complete lattices.

lemma \(\text{surj-Sup-pres-iso}\):
fixes $f :: \text{complete-lattice-with-dual} \Rightarrow \text{complete-lattice-with-dual}$
assumes \(\text{surj } f\)
and \(\text{Sup-pres } f\)
shows \(\{\text{radj } f \circ f\} (\bigcup X) = \bigcup_{x \in X} f x\)
by (metis \(\text{assms}(1)\) \(\text{assms}(2)\) \(\text{comp-def}\) \(\text{pointfree-idE}\) \(\text{surj-Sup-pres-id}\))

72
8.4 A Quick Example: Dedekind-MacNeille Completions

I only outline the basic construction. Additional facts about join density, and that the completion yields the least complete lattice that contains all Sups and Infs of the underlying posets, are left for future consideration.

abbreviation \( dm \equiv \text{lb-set} \circ \text{ub-set} \)

lemma up-set-prop: \((X::'a::preorder set) \neq \{\} \implies \text{ub-set} X = \bigcap \{ \uparrow x \mid x \in X \}\)
unfolding ub-set-def upset-def upset-set-def by (safe, simp-all, blast)

lemma lb-set-prop: \((X::'a::preorder set) \neq \{\} \implies \text{lb-set} X = \bigcap \{ \downarrow x \mid x \in X \}\)
unfolding lb-set-def downset-def downset-set-def by (safe, simp-all, blast)

lemma dm-downset-var: \(dm \{x\} = \downarrow (x::'a::preorder)\)
unfolding lb-set-def ub-set-def downset-def downset-set-def by (clarsimp, meson order-refl order-trans)

lemma dm-downset: \(dm \circ \eta = (\downarrow ::'a::preorder \Rightarrow 'a set)\)
using dm-downset-var fun.map-cong by fastforce

lemma clop (lb-set o ub-set)
unfolding clop-def lb-set-def ub-set-def
apply safe
unfolding le-fun-def comp-def id-def mono-def
by auto

end

9 Locale-Based Duality

theory Order-Lattice-Props-Loc
imports Main
  HOL-Library.Lattice-Syntax

begin

This section explores order and lattice duality based on locales. Used within the context of a class or locale, this is very effective, though more opaque than the previous approach. Outside of such a context, however, it apparently cannot be used for dualising theorems. Examples are properties of functions between orderings or lattices.

definition Fix :: ('a => 'a) => 'a set where
  Fix f = {x. f x = x}

context ord
begin

definition min-set :: 'a set ⇒ 'a set where
min-set X = \{ y ∈ X. ∀ x ∈ X. x ≤ y → x = y \}

definition max-set :: 'a set ⇒ 'a set where
max-set X = \{ x ∈ X. ∀ y ∈ X. x ≤ y → x = y \}

definition directed :: 'a set ⇒ bool where
directed X = (∀ Y. finite Y ∧ Y ⊆ X → (∃ x ∈ X. ∀ y ∈ Y. y ≤ x))

definition filtered :: 'a set ⇒ bool where
filtered X = (∀ Y. finite Y ∧ Y ⊆ X → (∃ x ∈ X. ∀ y ∈ Y. x ≤ y))

definition downset-set :: 'a set ⇒ 'a set (⇓) where
⇓ X = \{ y. ∃ x ∈ X. y ≤ x \}

definition upset-set :: 'a set ⇒ 'a set (⇑) where
⇑ X = \{ y. ∃ x ∈ X. x ≤ y \}

definition downset :: 'a ⇒ 'a set (⇓) where
downset = ⇓ ◦ (λ x. \{ x \})

definition upset :: 'a ⇒ 'a set (⇑) where
upset = ⇑ ◦ (λ x. \{ x \})

definition downsets :: 'a set set where
downsets = Fix ⇓

definition upsets :: 'a set set where
upsets = Fix ⇑

abbreviation downset-setp X ≡ X ∈ downsets
abbreviation upset-setp X ≡ X ∈ upsets

abbreviation ideals X ≡ X ∈ ideals
abbreviation filters X ≡ X ∈ filters

abbreviation Idealp X ≡ X ∈ ideals
abbreviation Filterp X ≡ X ∈ filters

abbreviation Sup-pres :: ('a::Sup ⇒ 'b::Sup) ⇒ bool where
Sup-pres f ≡ f ∘ Sup = Sup ∘ (‘) f

abbreviation Inf-pres :: ('a::Inf ⇒ 'b::Inf) ⇒ bool where
Inf-pres f ≡ f ∘ Inf = Inf ∘ (‘) f

abbreviation sup-pres :: ('a::sup ⇒ 'b::sup) ⇒ bool where
sup-pres f ≡ (∀ x y. f (x ⊔ y) = f x ⊔ f y)

abbreviation inf-pres :: ('a::inf ⇒ 'b::inf) ⇒ bool where
inf-pres f ≡ (∀ x y. f (x ⊓ y) = f x ⊓ f y)

abbreviation bot-pres :: ('a::bot ⇒ 'b::bot) ⇒ bool where
bot-pres f ≡ f ⊥ = ⊥

abbreviation top-pres :: ('a::top ⇒ 'b::top) ⇒ bool where
top-pres f ≡ f ⊤ = ⊤

abbreviation Sup-dual :: ('a::Sup ⇒ 'b::Inf) ⇒ bool where
Sup-dual f ≡ f ∘ Sup = Inf ∘ (‘) f

abbreviation Inf-dual :: ('a::Inf ⇒ 'b::Sup) ⇒ bool where
Inf-dual f ≡ f ∘ Inf = Sup ∘ (‘) f

abbreviation sup-dual :: ('a::sup ⇒ 'b::inf) ⇒ bool where
sup-dual f ≡ (∀ x y. f (x ⊔ y) = f x ⊓ f y)

abbreviation inf-dual :: ('a::inf ⇒ 'b::sup) ⇒ bool where
inf-dual f ≡ (∀ x y. f (x ⊓ y) = f x ⊔ f y)

abbreviation bot-dual :: ('a::bot ⇒ 'b::top) ⇒ bool where
bot-dual f ≡ f ⊥ = ⊤

abbreviation top-dual :: ('a::top ⇒ 'b::bot) ⇒ bool where
top-dual f ≡ f ⊤ = ⊥

9.1 Duality via Locales

sublocale ord ⊆ dual-ord: ord (≥) (>)
rewrites dual-max-set: max-set = dual-ord.min-set
and dual-filtered: filtered = dual-ord.directed
and dual-upset-set: upset-set = dual-ord.downset-set
and dual-upset: upset = dual-ord.downset
and dual-upsets: upsets = dual-ord.downsets
and dual-filters: filters = dual-ord.ideals
  apply unfold-locales
unfolding max-set-def ord.min-set-def fun-eq-iff apply blast
unfolding filtered-def ord.directed-def apply simp
unfolding upset-set-def ord.downset-set-def apply simp
apply (simp add: ord.downset-def ord.downset-set-def ord.upset-def ord.upset-set-def)
unfolding upsets-def ord.downsets-def apply (metis ord.downset-set-def upset-set-def)
unfolding filters-def ord.ideals-def Fix-def ord.downsets-def upsets-def ord.downset-set-def
upset-set-def ord.directed-def filtered-def
by simp

sublocale preorder ⊆ dual-preorder: preorder (≥) (>)
  apply unfold-locales
  apply (simp add: less-le-not-le)
  apply simp
  using order-trans by blast

sublocale order ⊆ dual-order: order (≥) (>)
  by (unfold-locales, simp)

sublocale lattice ⊆ dual-lattice: lattice sup (≥) (>)
inf T ⊥
  by (unfold-locales, simp-all)

sublocale bounded-lattice ⊆ dual-bounded-lattice: bounded-lattice sup (≥) (>)
inf T ⊥
  by (unfold-locales, simp-all)

sublocale boolean-algebra ⊆ dual-boolean-algebra: boolean-algebra λx y. x ⊔ −y
uminus sup (≥) (>)
inf T ⊥
  by (unfold-locales, simp-all add: inf-sup-distrib1)

sublocale complete-lattice ⊆ dual-complete-lattice: complete-lattice sup (≥) (>)
inf T ⊥
rewrites dual-gfp: gfp = dual-complete-lattice.lfp
proof -
  show class.complete-lattice sup Inf sup (≥) (>)
inf T ⊥
  by (unfold-locales, simp-all add: Sup-upper Sup-least Inf-lower Inf-greatest)
  then interpret dual-complete-lattice: complete-lattice sup Inf sup (≥) (>)
inf T ⊥.
  show gfp = dual-complete-lattice.lfp
  unfolding gfp-def dual-complete-lattice.lfp-def fun-eq-iff by simp
qed

context ord
begin

lemma dual-min-set: min-set = dual-ord.max-set
  by (simp add: dual-ord.dual-max-set)

lemma dual-directed: directed = dual-ord.filtered
  by (simp add:dual-ord.dual-filtered)

lemma dual-downset: downset = dual-ord.upset
  by (simp add: dual-ord.dual-upset)
lemma dual-downset-set: downset-set = dual-ord.upset-set
  by (simp add: dual-ord.dual-upset-set)

lemma dual-downsets: downsets = dual-ord.upsets
  by (simp add: dual-ord.dual-upsets)

lemma dual-ideals: ideals = dual-ord.filters
  by (simp add: dual-ord.dual-filters)

end

context complete-lattice
begin

lemma dual-lfp: lfp = dual-complete-lattice.gfp
  by (simp add: dual-complete-lattice.dual-gfp)

end

9.2 Properties of Orderings, Again

context ord
begin

lemma directed-nonempty: directed X \implies X \neq \{\}
  unfolding directed-def by fastforce

lemma directed-ub: directed X \implies (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \land y \leq z)
  by (meson empty-subsetI directed-def finite.emptyI finite-insert insert-subset order-refl)

lemma downset-set-prop: \Downarrow = \bigcup (x \in X. \Downarrow x)
  unfolding downset-def downset-set-def fun-eq-iff by fastforce

lemma downset-prop: \Downarrow x = \{y. y \leq x\}
  unfolding downset-def downset-set-def fun-eq-iff comp-def by fastforce

end

context preorder
begin

lemma directed-prop: X \neq \{\} \implies (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \land y \leq z) \implies directed X

proof
  assume h1: X \neq \{}
  and h2: \forall x \in X. \forall y \in X. \exists z \in X. x \leq z \land y \leq z

  77
\{ \text{fix } Y \}
\text{have } \text{finite } Y \implies Y \subseteq X \implies (\exists x \in X. \forall y \in Y. y \leq x)
\text{proof (induct rule: finite-induct)}
\begin{itemize}
  \item case empty
  \text{then show } \forall x \in X.
  \exists y \in Y. y \leq x
  \text{using } h1 \text{ by blast}
\end{itemize}
\text{next}
\begin{itemize}
  \item case (insert \( x \) \( F \))
  \text{then show } \forall x \in X.
  \exists y \in Y. y \leq x
  \text{by (metis h2 insert-iff insert-subset order-trans)}
\end{itemize}
\text{qed}
\text{thus } \forall x \in X. \forall y \in Y. y \leq x
\text{by (simp add: directed-def)}
\text{qed}

\text{lemma } \text{directed-alt}: \text{directed } X = (X \neq \{} \land (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \land y \leq z))
\text{by (metis directed-prop directed-nonempty directed-ub)}

\text{lemma } \text{downset-set-ext}: \text{id } \Downarrow\text{id-\text{def}} \text{downset-set-def by auto}

\text{lemma } \text{downset-set-iso}: \text{mono } \Downarrow\text{mono-def downset-set-def by blast}

\text{lemma } \text{downset-set-idem \[simp\]}: \Downarrow \Downarrow = \Downarrow
\text{unfolding fun-eq-iff downset-set-def by auto}

\text{lemma } \text{downset-faithful}: \Downarrow x \subseteq \Downarrow y \implies x \leq y
\text{by (simp add: downset-prop subset-eq)}

\text{lemma } \text{downset-iso-iff}: (\Downarrow x \subseteq \Downarrow y) = (x \leq y)
\text{using atMost-iff downset-prop order-trans by blast}

\text{lemma } \text{downset-directed-downset-var \[simp\]}: \text{directed } (\Downarrow X) = \text{directed } X
\text{proof}
\text{assume } h1: \text{directed } X
\text{\{fix } Y \}
\text{assume } h2: \text{finite } Y \text{ and } h3: Y \subseteq \Downarrow X
\text{hence } \forall y. \exists x. y \in Y \implies x \in X \land y \leq x
\text{by (force simp: downset-set-def)}
\text{hence } \exists f. \forall y. y \in Y \implies f y \in X \land y \leq f y
\text{by (rule choice)}
\text{hence } \exists f. \text{finite } (f' Y) \land f' Y \subseteq X \land (\forall y \in Y. y \leq f y)
\text{by (metis finite-imageI h2 image-subsetI)}
\text{hence } \exists Z. \text{finite } Z \land Z \subseteq X \land (\forall y \in Y. \exists z \in Z. y \leq z)
\text{by fastforce}
\text{hence } \exists Z. \text{finite } Z \land Z \subseteq X \land (\forall y \in Y. \exists z \in Z. y \leq z) \land (\exists x \in X. \forall z \in Z. z \leq x)
by (metis directed-def h1)

hence \( \exists x \in X. \forall y \in Y. y \leq x \)

by \{meson order-trans\}

thus directed \((\Downarrow X)\)

unfolding directed-def downset-set-def by fastforce

next

assume directed \((\Downarrow X)\)

thus directed \(X\)

unfolding directed-def downset-set-def

apply clarsimp

by \{smt Ball-Collect order-refl order-trans subsetCE\}

qed

lemma downset-directed-downset [simp]: directed \(\circ \Downarrow = \text{directed} \)

unfolding fun-eq-iff comp-def by simp

lemma directed-downset-ideals: directed \((\Downarrow X) = (\Downarrow Y \in \text{ideals})\)

by (metis (mono-tags, lifting) Fix-def comp-apply directed-alt downset-set-idem downsets-def ideals-def mem-Collect-eq)

end

lemma downset-iso: mono \((\Downarrow \cdot \cdot \cdot \cdot \cdot \text{order} \Rightarrow \cdot \cdot \cdot \cdot \cdot \text{set})\)

by (simp add: downset-iso-iff mono-def)

context order

begin

lemma downset-inj: inj \(\Downarrow\)

by (metis injI downset-iso-iff eq-iff)

end

context lattice

begin

lemma lat-ideals: \(X \in \text{ideals} = (X \neq \{\} \land X \in \text{downsets} \land (\forall x \in X. \forall y \in X. x \sqcup y \in X))\)

unfolding ideals-def directed-alt downsets-def Fix-def downset-set-def

by (clarsimp, smt sup.cobounded1 sup.orderE sup.orderI sup-absorb2 sup-left-commute mem-Collect-eq)

end

context bounded-lattice

begin

lemma bot-ideal: \(X \in \text{ideals} \Rightarrow \bot \in X\)

unfolding ideals-def downsets-def Fix-def downset-set-def by fastforce
end

context complete-lattice
begin

lemma Sup-downset-id [simp]: $\text{Sup} \circ \downarrow = \text{id}$
using Sup-atMost atMost-def downset-prop by fastforce

lemma downset-Sup-id: $\text{id} \leq \downarrow \circ \text{Sup}$
by (simp add: Sup-upper downset-prop le-funI subsetI)

lemma Inf-Sup-var: $\bigcup (\bigcap x \in X. \downarrow x) = \bigcap X$
unfolding downset-prop by (simp add: Collect-ball-eq Inf-Sup)

lemma Inf-pres-downset-var: $(\bigcap x \in X. \downarrow x) = \downarrow (\bigcap X)$
unfolding downset-prop by (safe, simp-all add: le-Inf-iff)

end

lemma lfp-in-Fix:
fixes $f :: 'a::complete-lattice \Rightarrow 'a$
shows mono $f \Rightarrow \text{lfp} f \in \text{Fix} f$
using Fix-def lfp-unfold by fastforce

lemma gfp-in-Fix:
fixes $f :: 'a::complete-lattice \Rightarrow 'a$
shows mono $f \Rightarrow \text{gfp} f \in \text{Fix} f$
using Fix-def gfp-unfold by fastforce

lemma nonempty-Fix:
fixes $f :: 'a::complete-lattice \Rightarrow 'a$
shows mono $f \Rightarrow \text{Fix} f \neq \{\}$
using lfp-in-Fix by fastforce

9.3 Dual Properties of Orderings from Locales
These properties can be proved very smoothly overall. But only within the
context of a class or locale!

context ord
begin

lemma filtered-nonempty: filtered $X \Rightarrow X \neq \{\}$
by (simp add: dual-filtered dual-ord.directed-nonempty)

lemma filtered-lb: filtered $X \Rightarrow (\forall x \in X. \forall y \in X. \exists z \in X. z \leq x \land z \leq y)$
by (simp add: dual-filtered dual-ord.directed-lb)

lemma upset-set-prop: $\uparrow = \text{Union} \circ (\cdot) \uparrow$

80
by (simp add: dual-ord.downset-set-prop dual-upset dual-upset-set)

lemma upset-set-prop-var: \( X = (\bigcup z \in X. \uparrow x) \)
  by (simp add: dual-ord.downset-set-prop-prop-var dual-upset dual-upset-set)

lemma upset-prop: \( \uparrow x = \{ y. x \leq y \} \)
  by (simp add: dual-ord.downset-prop dual-upset)

end

context preorder begin

lemma filtered-prop: \( X \neq \{\} \Rightarrow (\forall x \in X. \forall y \in X. \exists z \in X. z \leq x \wedge z \leq y) \)
  \( \Rightarrow \) filtered \( X \)
  by (simp add: dual-filtered dual-preorder.directed-prop)

lemma filtered-alt: filtered \( X = (X \neq \{\} \wedge (\forall x \in X. \forall y \in X. \exists z \in X. z \leq x \wedge z \leq y)) \)
  by (simp add: dual-filtered dual-preorder.directed-alt)

lemma upset-set-ext: id \( \leq \uparrow X \)
  by (simp add: dual-preorder.downset-set-ext dual-upset-set)

lemma upset-set-anti: mono \( \uparrow \)
  by (simp add: dual-preorder.downset-set-iso dual-upset)

lemma up-set-idem [simp]: \( \uparrow \circ \uparrow = \uparrow \)
  by (simp add: dual-upset-set)

lemma upset-faithful: \( \uparrow x \subseteq \uparrow y \Rightarrow y \leq x \)
  by (metis dual-preorder.downset-faithful dual-upset)

lemma upset-anti-iff: \( (\uparrow y \subseteq \uparrow x) = (x \leq y) \)
  by (simp add: dual-preorder.downset-iso-iff dual-upset)

lemma upset-filtered-upset [simp]: filtered \( \circ \uparrow = \) filtered
  by (simp add: dual-filtered dual-upset-set)

lemma filtered-upset-filters: filtered \( (\uparrow X) = (\uparrow X \in \text{filters}) \)
  using dual-filtered dual-preorder.directed-downset-ideals dual-upset-set ord.dual-filters
  by fastforce

end

context order begin

lemma upset-inj: inj \( \uparrow \)

end
by (simp add: dual-order.downset-inj dual-upset)

end

context lattice
begin

lemma lat-filters: \( X \in \text{filters} = (X \neq \{\} \land X \in \text{upsets} \land (\forall x \in X. \forall y \in X. x \cap y \in X)) \)
  by (simp add: dual-filters dual-lattice.lat-ideals dual-ord.downsets-def dual-upset-set upsets-def)

end

context bounded-lattice
begin

lemma top-filter: \( X \in \text{filters} \Rightarrow \top \in X \)
  by (simp add: dual-bounded-lattice.bot-ideal dual-filters)

end

context complete-lattice
begin

lemma Inf-upset-id [simp]: \( \inf \circ \uparrow = \text{id} \)
  by (simp add: dual-complete-lattice)

lemma upset-Inf-id: \( \text{id} \leq \uparrow \circ \inf \)
  by (simp add: dual-complete-lattice.Inf-pres-downset-var dual-upset)

end

9.4 Examples that Do Not Dualise

lemma upset-anti: antimono (\( \uparrow :: \text{a::order} \Rightarrow \text{a set} \))
  by (simp add: antimono-def upset-anti-iff)

context complete-lattice
begin

lemma fSup-unfold: \( \text{f} :: \text{nat} \Rightarrow \text{'}a :: \text{order} \Rightarrow \text{'}a \text{ set} \)
  apply (intro antisym sup-least)

end

82
apply (rule Sup-upper, force)
apply (rule Sup-mono, force)
apply (safe intro!: Sup-least)
by (case-tac n, simp-all add: Sup-upper le-supI2)

lemma fInf-unfold: (f::nat ⇒ 'a) 0 ▷ (⨅ n. f (Suc n)) = (⨅ n. f n)
apply (intro antisym inf-greatest)
apply (rule Inf-greatest, safe)
apply (case-tac n)
apply simp-all
using Inf-lower inf.coboundedI2 apply force
apply (simp add: Inf-lower)
by (auto intro: Inf-mono)

dropend

lemma fun-isol: mono f ⇒ mono (\x. x ∘ f)
by (simp add: le-fun-def mono-def)

lemma fun-isor: mono f ⇒ mono (\x. x ∘ f)
by (simp add: le-fun-def mono-def)

lemma Sup-sup-pres:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows Sup-pres f ⇒ sup-pres f
  by (metis (no-types, hide-lams) Sup-empty Sup-insert comp-apply image-insert sup-bot.right-neutral)

lemma Inf-inf-pres:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows Inf-pres f ⇒ inf-pres f
  by (smt INF-insert comp-eq-elim dual-complete-lattice Sup-empty dual-complete-lattice Sup-insert inf-top.right-neutral)

lemma Sup-bot-pres:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows Sup-pres f ⇒ bot-pres f
  by (metis SUP-empty Sup-empty comp-eq-elim)

lemma Inf-top-pres:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows Inf-pres f ⇒ top-pres f
  by (metis INF-empty comp-eq-elim dual-complete-lattice.Sup-empty)

context complete-lattice
begin

lemma iso-Inf-subdistl:
assumes mono (f::'a ⇒ 'b::complete-lattice)
shows f o Inf ≤ Inf o (') f
by (simp add: assms complete-lattice-class.le-Inf-iff le-funI Inf-lower monoD)

lemma iso-Sup-supdistl:
  assumes mono (f::'a ⇒ 'b::complete-lattice)
  shows Sup o (') f ≤ f o Sup
by (simp add: assms complete-lattice-class.SUP-le-iff le-funI dual-complete-lattice.Inf-lower monoD)

lemma Inf-subdistl-iso:
  fixes f :: 'a ⇒ 'b::complete-lattice
  shows f o Inf ≤ Inf o (') f =⇒ mono f
  unfolding mono-def le-fun-def comp-def by (metis complete-lattice-class.le-INF-iff Inf-atLeast atLeast-iff)

lemma Sup-supdistl-iso:
  fixes f :: 'a ⇒ 'b::complete-lattice
  shows Sup o (') f ≤ f o Sup =⇒ mono f
  unfolding mono-def le-fun-def comp-def by (metis complete-lattice-class.SUP-le-iff Sup-atMost atMost-iff)

lemma supdistl-iso:
  fixes f :: 'a ⇒ 'b::complete-lattice
  shows (Sup o (') f) ≤ f o Sup =⇒ mono f
  using Sup-supdistl-iso iso-Sup-supdistl by force

lemma subdistl-iso:
  fixes f :: 'a ⇒ 'b::complete-lattice
  shows (f o Inf) ≤ Inf o (') f =⇒ mono f
  using Inf-subdistl-iso iso-Inf-subdistl by force

end

lemma fSup-distr: Sup-pres (λx. x o f)
  unfolding fun-eq-iff comp-def
  by (smt INF-apply Inf INF-cong Sup-SUP-apply Sup-apply)

lemma fSup-distr-var: ⨆ F o g = (⨆ f ∈ F. f o g)
  unfolding fun-eq-iff comp-def
  by (smt INF-apply Inf INF-cong Sup-SUP-apply Sup-apply)

lemma fInf-distr: Inf-pres (λx. x o f)
  unfolding fun-eq-iff comp-def
  by (smt INF-apply Inf INF-cong Inf-apply)

lemma fInf-distr-var: ⨍ F o g = (⨍ f ∈ F. f o g)
  unfolding fun-eq-iff comp-def
  by (smt INF-apply Inf INF-cong Inf-apply)
lemma fSup-subdistl:
  assumes mono (f::'a::complete-lattice ⇒ 'b::complete-lattice)
  shows Sup ◦ (′) ((@) f) ≤ (@) f ◦ Sup
  using assms by (simp add: SUP-least Sup-upper le-fun-def monoD image-comp)

lemma fSup-subdistl-var:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows mono f =⇒ (∪ g ∈ G. f ◦ g) ≤ f ◦ G
  by (simp add: SUP-least Sup-upper le-fun-def monoD image-comp)

lemma fInf-subdistl:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows mono f =⇒ (∩ g ∈ G. f ◦ g) ≤ f ◦ G
  by (simp add: INF-greatest Inf-lower le-fun-def monoD image-comp)

lemma fInf-subdistl-var:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows mono f =⇒ f ◦ (d G) ≤ (d g ∈ G. f ◦ g)
  by (simp add: INF-greatest Inf-lower le-fun-def monoD image-comp)

lemma Inf-pres-downset: Inf-pres (↓ :: 'a::complete-lattice ⇒ 'a set)
  unfolding downset-prop fun-eq-iff comp-def
  by (safe, simp-all add: le-iff)

lemma Sup-dual-upset: Sup-dual (↑ :: 'a::complete-lattice ⇒ 'a set)
  unfolding upset-prop fun-eq-iff comp-def
  by (safe, simp-all add: Sup-le-iff)

This approach could probably be combined with the explicit functor-based one. This may be good for proofs, but seems conceptually rather ugly.

end

10 Duality Based on a Data Type

theory Order-Lattice-Props-Wenzel
  imports Main
  HOL-Library.Lattice-Syntax

begin

10.1 Wenzel’s Approach Revisited

This approach is similar to, but inferior to the explicit class-based one. The main caveat is that duality is not involutive with this approach, and this allows dualising less theorems.

I copy Wenzel’s development [11] in this subsection and extend it with ad-
ditional properties. I show only the most important properties.

data type 'a dual = dual (un-dual: 'a) (\partial)

notation un-dual (\partial^-)

lemma dual-inj: inj \partial
  using injI by fastforce

lemma dual-surj: surj \partial
  using dual.exhaust-sel by blast

lemma dual-bij: bij \partial
  by (simp add: bijI dual-inj dual-surj)

Dual is not idempotent, and I see no way of imposing this condition. Yet at
least an inverse exists — namely un-dual..

lemma dual-inv1 [simp]: \partial^- \circ \partial = id
  by fastforce

lemma dual-inv2 [simp]: \partial \circ \partial^- = id
  by fastforce

lemma dual-inv-inj: inj \partial^-
  by (simp add: bij-def dual-inv-inj dual-surj)

lemma dual-inv-surj: surj \partial^-
  by (metis dual.sel surj-def)

lemma dual-inv-bij: bij \partial^-
  by (simp add: bij-def dual-inv-inj dual-inv-surj)

lemma dual-iff: (\partial x = y) \leftrightarrow (x = \partial^- y)
  by fastforce

Isabelle data types come with a number of generic functions.

The functor map-dual lifts functions to dual types. Isabelle’s generic defini-
tion is not straightforward to understand and use. Yet conceptually it can
be explained as follows.

lemma map-dual-def-var [simp]: (map-dual::('a => 'b) => 'a dual => 'b dual) f =
  \partial \circ f \circ \partial^-
  unfolding fun-eq-iff comp-def by (metis dual.map-set dual-iff)

lemma map-dual-def-var2: \partial^- \circ map-dual f = f \circ \partial^-
  by (simp add: rewriteL-comp-comp)

lemma map-dual-funct: map-dual (f \circ g) = map-dual f \circ map-dual g
  unfolding fun-eq-iff comp-def by (metis dual.exhaust dual.map)
lemma map-dual-func2 : map-dual id = id
  by simp

The functor map-dual has an inverse functor as well.

definition map-dual-inv :: ('a dual ⇒ 'b dual) ⇒ ('a ⇒ 'b) where
  map-dual-inv f = ∂⁻ o f o ∂

lemma map-dual-inv-func1 : map-dual-inv id = id
  by (simp add: map-dual-inv-def)

lemma map-dual-inv-func2 : map-dual-inv (f o g) = map-dual-inv f o map-dual-inv g
  unfolding fun-eq_iff comp-def map-dual-inv-def by (metis dual-iff)

lemma map-dual-inv1 : map-dual o map-dual-inv = id
  unfolding fun-eq_iff map-dual-def-var map-dual-inv-def comp-def id-def
  by (metis dual-iff)

lemma map-dual-inv2 : map-dual-inv o map-dual = id
  unfolding fun-eq_iff map-dual-def-var map-dual-inv-def comp-def id-def
  by (metis dual-iff)

Hence dual is an isomorphism between categories.

lemma subset-dual : (∀ Y. (∀ X. X = Y) ↔ (∀ X. X = ∂⁻ X = Y))
  by (metis dual-inj image-comp image-inv-f-f inv-o-cancel dual-inv2)

lemma subset-dual1 : (∀ X. X ≤ Y) ↔ (∀ X. X ≤ ∂⁻ X ≤ Y)
  by (simp add: dual-inj inj-image-subset-iff)

lemma dual-ball : (∀ x. P (∂ x)) ↔ (∀ y. P y)
  by simp

lemma dual-inv-ball : (∀ x. P (∂⁻ x)) ↔ (∀ y. P y)
  by simp

lemma dual-all : (∀ x. P (∂ x)) ↔ (∀ y. P y)
  by (metis dual-collapse)

lemma dual-inv-all : (∀ x. P (∂⁻ x)) ↔ (∀ y. P y)
  by (metis dual-inv-surj surj-def)

lemma dual-ex : (∃ x. P (∂ x)) ↔ (∃ y. P y)
  by (metis UNIV-I bex-imageD dual-surj)

lemma dual-inv-ex : (∃ x. P (∂⁻ x)) ↔ (∃ y. P y)
  by (metis dual.sel)

lemma dual-Collect : {∂ x | x. P (∂ x)} = {y. P y}
by (metis dual.exhaust)

lemma dual-inv-Collect: \{ \partial^\sim x \mid x. \ P(\partial^\sim x) \} = \{ y. \ P y \}
  by (metis dual.collapse dual.inject)

lemma fun-dual1: (f \circ \partial = g) \iff (f = g \circ \partial^\sim)
  by auto

lemma fun-dual2: (\partial \circ f = g) \iff (f = \partial^\sim \circ g)
  by auto

lemma fun-dual3: (f \circ (\partial^\sim) = g)
  unfolding fun-eq-iff comp-def by (metis subset-dual)

lemma fun-dual4: (f = \partial^\sim \circ g \circ (\partial^\sim))
  by (metis fun-dual2 fun-dual3 o-assoc)

The next facts show incrementally that the dual of a complete lattice is a complete lattice. This follows once again Wenzel.

instantiation dual :: (ord) ord
begin

definition less-eq-dual-def: (\leq) = rel-dual (\geq)

definition less-dual-def: (<) = rel-dual (>)

instance..
end

lemma less-eq-dual-def-var: (x \leq y) = (\partial^\sim y \leq \partial^\sim x)
  apply (rule antisym)
  apply (simp add: dual.rel-sel less-eq-dual-def)
  by (simp add: dual.rel-sel less-eq-dual-def)

lemma less-dual-def-var: (x < y) = (\partial^\sim y < \partial^\sim x)
  by (simp add: dual.rel-sel less-dual-def)

instance dual :: (preorder) preorder
  apply standard
  apply (simp add: less-dual-def-var less-eq-dual-def-var less-le-not-le)
  apply (simp add: less-eq-dual-def-var)
  by (meson less-eq-dual-def-var order-trans)

instance dual :: (order) order
  by (standard, simp add: dual.expand less-eq-dual-def-var)

lemma dual-anti: x \leq y \Rightarrow \partial y \leq \partial x
  by (simp add: dual-inj less-eq-dual-def the-inv-f-f)
lemma dual-anti-iff: \((x \leq y) = (\partial y \leq \partial x)\)
by \((\text{simp add: dual-inj less-eq-dual-def the-inv-f-f})\)

map-dual does not map isotone functions to antitone ones. It simply lifts the type!

lemma mono \(f \implies\) mono \((\text{map-dual} f)\)
unfolding map-dual-def-var mono-def by \((\text{metis comp-apply dual-anti less-eq-dual-def-var})\)

instantiation dual :: \((\text{lattice})\) lattice
begin

definition inf-dual-def: \(x \sqcap y = \partial (\partial^- x \sqcup \partial^- y)\)
definition sup-dual-def: \(x \sqcup y = \partial (\partial^- x \sqcap \partial^- y)\)

instance
by \((\text{standard, simp-all add: dual-inj inf-dual-def sup-dual-def less-eq-dual-def-var the-inv-f-f})\)
end

instantiation dual :: \((\text{complete-lattice})\) complete-lattice
begin

definition Inf-dual-def: \(\inf = \partial \circ \sup \circ (\cdot) \partial^-\)
definition Sup-dual-def: \(\sup = \partial \circ \inf \circ (\cdot) \partial^-\)
definition bot-dual-def: \(\bot = \partial \top\)
definition top-dual-def: \(\top = \partial \bot\)

instance
by \((\text{standard, simp-all add: Inf-dual-def top-dual-def Sup-dual-def bot-dual-def dual-inj le-INF-iff SUP-le-iff INF-lower SUP-upper less-eq-dual-def-var the-inv-f-f})\)
end

Next, directed and filtered sets, upsets, downsets, filters and ideals in posets are defined.

context ord
begin

definition directed :: \('a set \Rightarrow bool\) where
directed \(X = (\forall Y. \text{finite } Y \land Y \subseteq X \implies (\exists x \in X. \forall y \in Y. y \leq x))\)

definition filtered :: \('a set \Rightarrow bool\) where
filtered \(X = (\forall Y. \text{finite } Y \land Y \subseteq X \implies (\exists x \in X. \forall y \in Y. x \leq y))\)
definition downset-set :: 'a set ⇒ 'a set (⇓) where
\[ \downarrow X = \{ y. \exists x \in X. y \leq x \} \]

definition upset-set :: 'a set ⇒ 'a set (⇑) where
\[ \uparrow X = \{ y. \exists x \in X. x \leq y \} \]

end

10.2 Examples that Do Not Dualise

Filtered and directed sets are dual.

Proofs could be simplified if dual was idempotent.

lemma filtered-directed-dual: filtered ◦ (′) ∇ = directed
proof
  { fix X :: 'a set have (filtered ◦ (′) ∇) X = (∀ Y. finite (∇− ′ Y) ∧ ∇− ′ Y ⊆ X → (∃ x ∈ X.∀ y ∈ (∇− ′ Y). ∇ x ≤ ∇ y)) unfolding filtered-def comp-def by (simp, metis dual-iff finite-subset-image subset-dual subset-dual1) also have ... = (∀ Y. finite Y ∧ Y ⊆ X → (∃ x ∈ X.∀ y ∈ Y. y ≤ x)) by (metis dual-anti-iff dual-inv-surj finite-subset-image top.extremum) finally have (filtered ◦ (′) ∇) X = directed X using directed-def by auto } thus ?thesis unfolding fun-eq-iff by simp qed

lemma directed-filtered-dual: directed ◦ (′) ∇ = filtered
proof
  { fix X :: 'a set have (directed ◦ (′) ∇) X = (∀ Y. finite (∇− ′ Y) ∧ ∇− ′ Y ⊆ X → (∃ x ∈ X.∀ y ∈ (∇− ′ Y). ∇ x ≤ ∇ y)) unfolding directed-def comp-def by (simp, metis dual-iff finite-subset-image subset-dual subset-dual1) also have ... = (∀ Y. finite Y ∧ Y ⊆ X → (∃ x ∈ X.∀ y ∈ Y. x ≤ y)) unfolding dual-anti-iff[symmetric] by (metis dual-inv-surj finite-subset-image top-greatest) finally have (directed ◦ (′) ∇) X = filtered X using filtered-def by auto } thus ?thesis unfolding fun-eq-iff by simp qed

This example illustrates the deficiency of the approach. In the class-based approach the second proof is trivial.

The next example shows that this is a systematic problem.
lemma \(\downarrow\downarrow \circ \uparrow\uparrow = \uparrow\downarrow \circ \downarrow\downarrow\):

proof

\{ fix \(X :: 'a set \)
have \((\downarrow\downarrow \circ \downarrow\downarrow) X = \{ \downarrow y \mid y, \exists x \in X. y \leq x \}\)
  by (simp add: downset-set-def setcompr-eq-image)
also have \(\ldots = \{ \downarrow y \mid y, \exists x \in X. \downarrow x \leq \downarrow y \}\)
  by (meson dual-anti-iff)
also have \(\ldots = \{ y. \exists x \in \downarrow\downarrow X. x \leq y \}\)
  by (metis (mono-tags, hide-lams) dual.exhaust image-iff)
finally have \((\downarrow\downarrow \circ \downarrow\downarrow) X = (\uparrow\uparrow \circ \downarrow\downarrow) X\)
  by (simp add: upset-set-def)
thus \(?thesis\)
unfolding fun-eq_iff by simp
qed

lemma \(\uparrow\uparrow \circ \downarrow\downarrow \circ \uparrow\uparrow = \downarrow\downarrow \circ \uparrow\uparrow \circ \downarrow\downarrow\):

unfolding downset-set-def upset-set-def fun-eq_iff comp_def
apply (safe, force simp: dual-anti)
by (metis (mono-tags, lifting) dual.exhaust dual-anti_iff mem-Collect_eq rev-image-eqI)

end

References


