Properties of Orderings and Lattices

Georg Struth

March 17, 2025

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

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*
begin

unbundle *lattice-syntax*

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* :: $'a\ set \Rightarrow\ 'a\ set$ **where**

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

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

end

definition $ord\text{-}pres$ :: $('a::ord \Rightarrow 'b::ord) \Rightarrow bool$ **where**
$ord\text{-}pres\ f = (\forall x\ y.\ x \leq y \longrightarrow f\ x \leq f\ y)$

lemma $ord\text{-}pres\text{-}mono$:
fixes f :: $'a::order \Rightarrow 'b::order$
shows $mono\ f = ord\text{-}pres\ f$
by ($simp\ add$: $mono\text{-}def\ ord\text{-}pres\text{-}def$)

class $preorder\text{-}lean = ord\ +$
assumes $preorder\text{-}refl$: $x \leq x$
and $preorder\text{-}trans$: $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$

begin

definition le :: $'a \Rightarrow 'a \Rightarrow bool$ **where**
$le\ x\ y = (x \leq y \wedge \neg\ (x \geq y))$

end

sublocale $preorder\text{-}lean \subseteq prel$: $preorder\ (\leq)\ le$
by ($unfold\text{-}locales$, $auto\ simp\ add$: $le\text{-}def\ preorder\text{-}refl\ preorder\text{-}trans$)

class $order\text{-}lean = preorder\text{-}lean\ +$
assumes $order\text{-}antisym$: $x \leq y \Longrightarrow x \geq y \Longrightarrow x = y$

sublocale $order\text{-}lean \subseteq posl$: $order\ (\leq)\ le$
by ($unfold\text{-}locales$, $simp\ add$: $order\text{-}antisym$)

class $Sup\text{-}lattice = order\text{-}lean + Sup\ +$
assumes $Sups\text{-}upper$: $x \in X \Longrightarrow x \leq \bigsqcup X$
and $Sups\text{-}least$: $(\bigwedge x.\ x \in X \Longrightarrow x \leq z) \Longrightarrow \bigsqcup X \leq z$

begin

definition $Infs$ :: $'a\ set \Rightarrow 'a$ **where**
$Infs\ X = \bigsqcup\{y.\ \forall x \in X.\ y \leq x\}$

definition $sups$ :: $'a \Rightarrow 'a \Rightarrow 'a$ **where**
$sups\ x\ y = \bigsqcup\{x,y\}$

definition $infs$ :: $'a \Rightarrow 'a \Rightarrow 'a$ **where**
$infs\ x\ y = Infs\{x,y\}$

definition *bots* :: *'a* **where**
  $bots = \bigsqcup\{\}$

definition *tops* :: *'a* **where**
  $tops = Infs\{\}$

lemma *Infs-prop*: $Infs = Sup \circ lb\text{-}set$
  unfolding *fun-eq-iff* **by** (*simp add*: *Infs-def prel.lb-set-def*)

end

class *Inf-lattice* = *order-lean* + *Inf* +
  assumes *Infi-lower*: $x \in X \Longrightarrow \bigsqcap X \leq x$
  and *Infi-greatest*: $(\bigwedge x.\ x \in X \Longrightarrow z \leq x) \Longrightarrow z \leq \bigsqcap X$

begin

definition *Supi* :: *'a set* ⇒ *'a* **where**
  $Supi\ X = \bigsqcap\{y.\ \forall x \in X.\ x \leq y\}$

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

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

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

definition *topi* :: *'a* **where**
  $topi = \bigsqcap\{\}$

lemma *Supi-prop*: $Supi = Inf \circ ub\text{-}set$
  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 sups bots tops*
apply *unfold-locales*
unfolding *Infs-def infs-def sups-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*
by (*unfold-locales*, *simp-all add*: *ldual.Sups-upper ldual.Sups-least ldual.supclat.Inf-lower ldual.supclat.Inf-greatest*)

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* :: $'a \Rightarrow 'a$ (‹∂›)
assumes *inj-dual*: *inj* ∂
and *invol-dual* [*simp*]: $\partial \circ \partial = id$

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*]: ∂ (∂ *x*) = *x*
by (*simp add*: *pointfree-idE*)

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

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

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

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

lemma *the-inv-dual*: *the-inv* ∂ = ∂
by (*metis comp-apply id-def invol-dual-var inj-dual surj-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* (‹∂_F›) **where**
∂_F *f* = ∂ ∘ *f* ∘ ∂

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

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

lemma *map-dual-nat-iso*: ∂_F *f* ∘ ∂ = ∂ ∘ *id f*
by (*simp add*: *comp-assoc map-dual-def*)

lemma *map-dual-invol* [*simp*]: $\partial_F \circ \partial_F = id$
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* η
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*: $(\text{`})\ (f \circ g) = (\text{`})\ f \circ (\text{`})\ g$
unfolding *fun-eq-iff* **by** (*simp add*: *image-comp*)

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

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

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

lemma *iso-image*: *mono* $((\text{`})\ f)$
by (*simp add*: *image-mono monoI*)

lemma *iso-preimage*: *mono* $((-\text{`})\ f)$
by (*simp add*: *monoI vimage-mono*)

context *dual*
begin

lemma *image-dual* [*simp*]: $(\text{`})\ \partial \circ (\text{`})\ \partial = id$
by (*metis image-func1 image-func2 invol-dual*)

lemma *vimage-dual* [*simp*]: $(-\text{`})\ \partial \circ (-\text{`})\ \partial = id$
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*: $\eta \circ id\ f = (\text{`})\ f \circ \eta$
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*: $(\text{`})\ \partial \circ \eta = \eta \circ \partial$
by *auto*

lemma *finite-dual* [*simp*]: $finite \circ (\text{`})\ \partial = finite$
unfolding *fun-eq-iff comp-def* **using** *inj-dual finite-vimageI inj-vimage-image-eq* **by** *fastforce*

lemma *finite-dual-var* [*simp*]: $finite\ (\partial\ \text{`}\ X) = finite\ X$
by (*metis comp-def finite-dual*)

lemma *subset-dual*: $(X = \partial\ \text{`}\ Y) = (\partial\ \text{`}\ X = Y)$
by (*metis image-dual pointfree-idE*)

lemma *subset-dual1*: $(X \subseteq Y) = (\partial\ \text{`}\ X \subseteq \partial\ \text{`}\ Y)$
by (*simp add*: *inj-dual inj-image-subset-iff*)

lemma *dual-empty* [*simp*]: $\partial\ \text{`}\ \{\} = \{\}$
by *simp*

lemma *dual-UNIV* [*simp*]: $\partial\ \text{`}\ UNIV = UNIV$
by (*simp add*: *surj-dual*)

lemma *fun-dual1*: $(f = g \circ \partial) = (f \circ \partial = g)$
by (*metis comp-assoc comp-id invol-dual*)

lemma *fun-dual2*: $(f = \partial \circ g) = (\partial \circ f = g)$
by (*metis comp-assoc fun.map-id invol-dual*)

lemma *fun-dual3*: $(f = g \circ (\text{`})\ \partial) = (f \circ (\text{`})\ \partial = g)$
by (*metis comp-id image-dual o-assoc*)

lemma *fun-dual4*: $(f = (\text{`})\ \partial \circ g) = ((\text{`})\ \partial \circ f = g)$
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*: $(f = (\text{`})\ \partial \circ g \circ (\text{`})\ \partial) = ((\text{`})\ \partial \circ f \circ (\text{`})\ \partial = g)$
  by (*simp add*: *comp-assoc fun-dual3 fun-dual4*)

lemma *fun-dual7*: $(f = \partial \circ g \circ (\text{`})\ \partial) = (\partial \circ f \circ (\text{`})\ \partial = g)$
  by (*simp add*: *comp-assoc fun-dual2 fun-dual3*)

lemma *fun-dual8*: $(f = (\text{`})\ \partial \circ g \circ \partial) = ((\text{`})\ \partial \circ f \circ \partial = g)$
  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

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-dual-ord map-dual-def 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 ord-dual*)

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

begin

lemma *less-dual-def-var*: $(\partial\ y < \partial\ x) = (x < y)$
  by (*simp add*: *dual-dual-ord less-le-not-le*)

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* $(\partial_F\ f)$
  by (*simp add*: *ord-pres-dual ord-pres-mono*)

class *lattice-with-dual* = *lattice* + *dual* +
assumes *sup-dual-def*: $\partial\ (x \sqcup y) = \partial\ x \sqcap \partial\ y$

begin

subclass *order-with-dual*
by (*unfold-locales*, *metis inf.absorb-iff2 sup-absorb1 sup-commute sup-dual-def*)

lemma *inf-dual*: $\partial\ (x \sqcap y) = \partial\ x \sqcup \partial\ y$
by (*metis invol-dual-var sup-dual-def*)

lemma *inf-to-sup*: $x \sqcap y = \partial\ (\partial\ x \sqcup \partial\ y)$
using *inf-dual dual-iff* **by** *fastforce*

lemma *sup-to-inf*: $x \sqcup y = \partial\ (\partial\ x \sqcap \partial\ y)$
by (*simp add*: *inf-dual*)

end

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

begin

lemma *bot-dual*: $\partial\ \bot = \top$
by (*metis dual-dual-ord dual-iff le-bot top-greatest*)

lemma *top-dual*: $\partial\ \top = \bot$
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*: $\partial \circ Sup = Infs \circ (`)\ \partial$

class *Inf-lattice-with-dual* = *Inf-lattice* + *dual* +
assumes *Sups-dual-def*: $\partial \circ Supi = Inf \circ (`)\ \partial$

class *complete-lattice-with-dual* = *complete-lattice* + *dual* +
assumes *Sups-dual-def*: $\partial \circ Sup = Inf \circ (`)\ \partial$

sublocale *Sup-lattice-with-dual* ⊆ *sclatd*: *complete-lattice-with-dual Infs Sup infs* $(\leq)$ *le sups bots tops* ∂
by (*unfold-locales*, *simp add*: *Sups-dual-def*)

sublocale *Inf-lattice-with-dual* ⊆ *iclatd*: *complete-lattice-with-dual Inf Supi infi* (≤) *le supi boti topi* ∂
by (*unfold-locales*, *simp add*: *Sups-dual-def*)

context *complete-lattice-with-dual*
begin

lemma *Inf-dual*: ∂ ∘ *Inf* = *Sup* ∘ (') ∂
by (*metis comp-assoc comp-id fun.map-id Sups-dual-def image-dual invol-dual*)

lemma *Inf-dual-var*: $\partial\ (\bigsqcap X) = \bigsqcup (\partial\ `\ X)$
using *comp-eq-dest Inf-dual* **by** *fastforce*

lemma *Inf-to-Sup*: *Inf* = ∂ ∘ *Sup* ∘ (') ∂
by (*auto simp add*: *Sups-dual-def image-comp*)

lemma *Inf-to-Sup-var*: $\bigsqcap X = \partial\ (\bigsqcup (\partial\ `\ X))$
using *Inf-dual-var dual-iff* **by** *fastforce*

lemma *Sup-to-Inf*: *Sup* = ∂ ∘ *Inf* ∘ (') ∂
by (*auto simp add*: *Inf-dual image-comp*)

lemma *Sup-to-Inf-var*: $\bigsqcup X = \partial\ (\bigsqcap (\partial\ `\ X))$
using *Sup-to-Inf* **by** *force*

lemma *Sup-dual-def-var*: $\partial\ (\bigsqcup X) = \bigsqcap\ (\partial\ `\ X)$
using *comp-eq-dest Sups-dual-def* **by** *fastforce*

lemma *bot-dual-def*: ∂ ⊤ = ⊥
by (*smt* (*verit*) *Inf-UNIV Sup-UNIV Sups-dual-def surj-dual o-eq-dest*)

lemma *top-dual-def*: ∂ ⊥ = ⊤
using *bot-dual-def dual-iff* **by** *blast*

lemma *inf-dual2*: ∂ (*x* ⊓ *y*) = ∂ *x* ⊔ ∂ *y*
by (*smt* (*verit*) *comp-eq-elim Inf-dual Inf-empty Inf-insert SUP-insert inf-top.right-neutral*)

lemma *sup-dual*: ∂ (*x* ⊔ *y*) = ∂ *x* ⊓ ∂ *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*..

end

end

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* ⟹ *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*::$('a::order \Rightarrow 'b::order)$ *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* :: $('a \Rightarrow 'a) \Rightarrow 'a$ *set* **where**
  $Fix\ f = \{x.\ f\ x = x\}$

lemma *retraction-prop*: $f \circ f = f \Longrightarrow f\ x = x \longleftrightarrow x \in range\ f$
  by (*metis comp-apply f-inv-into-f rangeI*)

lemma *retraction-prop-fix*: $f \circ f = f \Longrightarrow range\ f = Fix\ f$
  unfolding *Fix-def* **using** *retraction-prop* **by** *fastforce*

lemma *Fix-map-dual*: $Fix \circ \partial_F = (`)\ \partial \circ Fix$
  unfolding *Fix-def map-dual-def comp-def fun-eq-iff*
  by (*smt* (*verit*) *Collect-cong invol-dual pointfree-idE setcompr-eq-image*)

lemma *Fix-map-dual-var*: $Fix\ (\partial_F\ f) = \partial\ `\ (Fix\ f)$
  by (*metis Fix-map-dual o-def*)

lemma *gfp-dual*: $(\partial::'a::complete\text{-}lattice\text{-}with\text{-}dual \Rightarrow 'a) \circ gfp = lfp \circ \partial_F$
proof−
  {**fix** f:: $'a \Rightarrow 'a$
  have $\partial\ (gfp\ f) = \partial\ (\bigsqcup\{u.\ u \le f\ u\})$
    by (*simp add*: *gfp-def*)
  also have ... $= \bigsqcap(\partial\ `\ \{u.\ u \le f\ u\})$
    by (*simp add*: *Sup-dual-def-var*)
  also have ... $= \bigsqcap\{\partial\ u\ |u.\ u \le f\ u\}$
    by (*simp add*: *setcompr-eq-image*)
  also have ... $= \bigsqcap\{u\ |u.\ (\partial_F\ f)\ u \le u\}$
    by (*metis* (*no-types*, *opaque-lifting*) *dual-dual-ord dual-iff map-dual-def o-def*)
  finally have $\partial\ (gfp\ f) = lfp\ (\partial_F\ f)$
    by (*metis lfp-def*)}
  thus *?thesis*
    by *auto*
qed

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

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

lemma *gfp-to-lfp-var*:
fixes f :: $'a$::*complete-lattice-with-dual* $\Rightarrow$ $'a$
shows *gfp* $f = \partial$ (*lfp* (∂_F f))
by (*metis gfp-dual-var invol-dual-var*)

lemma *lfp-dual*: (∂::$'a$::*complete-lattice-with-dual* $\Rightarrow$ $'a$) $\circ$ *lfp* = *gfp* $\circ$ ∂_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 ∂ (*lfp* f) = *gfp* (*map-dual* f)
using *comp-eq-dest-lhs lfp-dual* **by** *fastforce*

lemma *lfp-to-gfp*: *lfp* = (∂::$'a$::*complete-lattice-with-dual* $\Rightarrow$ $'a$) $\circ$ *gfp* $\circ$ ∂_F
by (*simp add*: *comp-assoc gfp-dual map-dual-invol*)

lemma *lfp-to-gfp-var*:
fixes f :: $'a$::*complete-lattice-with-dual* $\Rightarrow$ $'a$
shows *lfp* $f = \partial$ (*gfp* (∂_F f))
by (*metis invol-dual-var lfp-dual-var*)

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

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

context *ord*
begin

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

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

end

context *ord-with-dual*
begin

lemma *min-max-set-dual*: $(\text{`})\ \partial \circ \textit{min-set} = \textit{max-set} \circ (\text{`})\ \partial$
  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*: $\partial\ \text{`}\ (\textit{min-set}\ X) = \textit{max-set}\ (\partial\ \text{`}\ X)$
  using *comp-eq-dest min-max-set-dual* **by** *fastforce*

lemma *max-min-set-dual*: $(\text{`})\ \partial \circ \textit{max-set} = \textit{min-set} \circ (\text{`})\ \partial$
  by (*metis* (*no-types*, *opaque-lifting*) *comp-id fun.map-comp id-comp image-dual min-max-set-dual*)

lemma *min-to-max-set*: $\textit{min-set} = (\text{`})\ \partial \circ \textit{max-set} \circ (\text{`})\ \partial$
  by (*metis comp-id image-dual max-min-set-dual o-assoc*)

lemma *max-min-set-dual-var*: $\partial\ \text{`}\ (\textit{max-set}\ X) = \textit{min-set}\ (\partial\ \text{`}\ X)$
  using *comp-eq-dest max-min-set-dual* **by** *fastforce*

lemma *min-to-max-set-var*: $\textit{min-set}\ X = \partial\ \text{`}\ (\textit{max-set}\ (\partial\ \text{`}\ X))$
  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* :: $'a\ set \Rightarrow bool$ **where**
  $\textit{directed}\ X = (\forall Y.\ \textit{finite}\ Y \wedge Y \subseteq X \longrightarrow (\exists x \in X.\ \forall y \in Y.\ y \leq x))$

definition *filtered* :: $'a\ set \Rightarrow bool$ **where**
  $\textit{filtered}\ X = (\forall Y.\ \textit{finite}\ Y \wedge Y \subseteq X \longrightarrow (\exists x \in X.\ \forall y \in Y.\ x \leq y))$

definition *downset-set* :: $'a\ set \Rightarrow 'a\ set$ (‹⇓›) **where**
  $\Downarrow X = \{y.\ \exists x \in X.\ y \leq x\}$

definition *upset-set* :: $'a\ set \Rightarrow 'a\ set$ (‹⇑›) **where**
  $\Uparrow X = \{y.\ \exists x \in X.\ x \leq y\}$

definition *downset* :: $'a \Rightarrow 'a\ set$ (‹↓›) **where**
  $\downarrow\ = \Downarrow \circ\ \eta$

definition *upset* :: $'a \Rightarrow 'a\ set$ (‹↑›) **where**
  $\uparrow\ = \Uparrow \circ\ \eta$

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

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

definition *downclosed-set X* = (*X* ∈ *downsets*)

definition *upclosed-set X* = (*X* ∈ *upsets*)

definition *ideals* :: *'a set set* **where**
ideals = {*X*. *X* ≠ {} ∧ *downclosed-set X* ∧ *directed X*}

definition *filters* :: *'a set set* **where**
filters = {*X*. *X* ≠ {} ∧ *upclosed-set X* ∧ *filtered X*}

abbreviation *idealp X* ≡ *X* ∈ *ideals*

abbreviation *filterp X* ≡ *X* ∈ *filters*

end

These notions are pair-wise dual.

Filtered and directed sets are dual.

context *ord-with-dual*
begin

lemma *filtered-directed-dual*: *filtered* ∘ (') ∂ = *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* (*verit*, *ccfv-threshold*) *finite-subset-image imageE ord-dual*)

lemma *directed-filtered-dual*: *directed* ∘ (') ∂ = *filtered*
using *filtered-directed-dual* **by** (*metis comp-id image-dual o-assoc*)

lemma *filtered-to-directed*: *filtered X* = *directed* (∂ ' *X*)
by (*metis comp-apply directed-filtered-dual*)

Upsets and downsets are dual.

lemma *downset-set-upset-set-dual*: (') ∂ ∘ ⇓ = ⇑ ∘ (') ∂
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*: (') ∂ ∘ ⇑ = ⇓ ∘ (') ∂

using *downset-set-upset-set-dual* **by** (*metis* (*no-types*, *opaque-lifting*) *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*: $\Uparrow X = \partial\ `\ (\Downarrow (\partial\ `\ X))$
by (*simp add*: *upset-set-to-downset-set*)

lemma *downset-upset-dual*: $(`)\ \partial \circ \downarrow = \uparrow \circ \partial$
using *downset-def upset-def upset-set-to-downset-set* **by** *fastforce*

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

lemma *upset-to-downset3*: $\uparrow x = \partial\ `\ (\downarrow (\partial\ x))$
by (*simp add*: *upset-to-downset2*)

lemma *downsets-upsets-dual*: ($X \in$ *downsets*) = ($\partial\ `\ X \in$ *upsets*)
unfolding *downsets-def upsets-def Fix-def*
by (*smt* (*verit*) *comp-eq-dest downset-set-upset-set-dual image-inv-f-f inj-dual mem-Collect-eq*)

lemma *downset-setp-upset-setp-dual*: *upclosed-set* $\circ$ $(`)\ \partial$ = *downclosed-set*
unfolding *downclosed-set-def upclosed-set-def* **using** *downsets-upsets-dual* **by** *fastforce*

lemma *upsets-to-downsets*: ($X \in$ *upsets*) = ($\partial\ `\ X \in$ *downsets*)
by (*simp add*: *downsets-upsets-dual image-comp*)

lemma *upset-setp-downset-setp-dual*: *downclosed-set* $\circ$ $(`)\ \partial$ = *upclosed-set*
by (*metis comp-id downset-setp-upset-setp-dual image-dual o-assoc*)

Filters and ideals are dual.

lemma *ideals-filters-dual*: ($X \in$ *ideals*) = (($\partial\ `\ X$) $\in$ *filters*)
by (*smt* (*verit*) *comp-eq-dest-lhs directed-filtered-dual image-inv-f-f image-is-empty inv-unique-comp filters-def ideals-def inj-dual invol-dual mem-Collect-eq upset-setp-downset-setp-dual*)

lemma *idealp-filterp-dual*: *idealp* = *filterp* $\circ$ $(`)\ \partial$
unfolding *fun-eq-iff* **by** (*simp add*: *ideals-filters-dual*)

lemma *filters-to-ideals*: ($X \in$ *filters*) = (($\partial\ `\ X$) $\in$ *ideals*)
by (*simp add*: *ideals-filters-dual image-comp*)

lemma *filterp-idealp-dual*: *filterp* = *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* $X \Longrightarrow X \neq \{\}$
unfolding *directed-def* **by** *fastforce*

lemma *directed-ub*: *directed* $X \Longrightarrow (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \wedge y \leq z)$
by (*meson empty-subsetI directed-def finite.emptyI finite-insert insert-subset order-refl*)

lemma *downset-set-prop*: $\Downarrow = $ *Union* $\circ$ (') $\downarrow$
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-prop*: $\downarrow x = \{y. y \leq x\}$
unfolding *downset-def downset-set-def fun-eq-iff* **by** *fastforce*

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

lemma *ideals-downsets*: $X \in$ *ideals* $\Longrightarrow X \in$ *downsets*
by (*simp add*: *downclosed-set-def ideals-def*)

lemma *ideals-directed*: $X \in$ *ideals* $\Longrightarrow$ *directed* X
by (*simp add*: *ideals-def*)

end

context *preorder*
begin

lemma *directed-prop*: $X \neq \{\} \Longrightarrow (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \wedge y \leq z) \Longrightarrow$ *directed* X
proof−
assume *h1*: $X \neq \{\}$
and *h2*: $\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \wedge y \leq z$
{fix Y
have *finite* $Y \Longrightarrow Y \subseteq X \Longrightarrow (\exists x \in X. \forall y \in Y. y \leq x)$
proof (*induct rule*: *finite-induct*)
case *empty*
then show *?case*

using *h1* **by** *blast*
next
case (*insert x F*)
then show *?case*
by (*metis h2 insert-iff insert-subset order-trans*)
qed}
thus *?thesis*
by (*simp add*: *directed-def*)
qed

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

lemma *downset-set-prop-var2*: $x \in \Downarrow X \Longrightarrow y \leq x \Longrightarrow y \in \Downarrow X$
unfolding *downset-set-def* **using** *order-trans* **by** *blast*

lemma *downclosed-set-iff*: *downclosed-set* $X = (\forall x \in X. \forall y. y \leq x \longrightarrow y \in X)$
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*: $\downarrow x \subseteq \downarrow y \Longrightarrow x \leq y$
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
assume *h2*: *finite* Y **and** *h3*: $Y \subseteq \Downarrow X$
hence $\forall y. \exists x. y \in Y \longrightarrow x \in X \wedge y \leq x$
by (*force simp*: *downset-set-def*)

hence $\exists f.\ \forall y.\ y \in Y \longrightarrow f\ y \in X \wedge y \leq f\ y$
by (*rule choice*)
hence $\exists f.$ *finite* $(f\ `\ Y) \wedge f\ `\ Y \subseteq X \wedge (\forall y \in Y.\ y \leq f\ y)$
by (*metis finite-imageI h2 image-subsetI*)
hence $\exists Z.$ *finite* $Z \wedge Z \subseteq X \wedge (\forall y \in Y.\ \exists\ z \in Z.\ y \leq z)$
by *fastforce*
hence $\exists Z.$ *finite* $Z \wedge Z \subseteq X \wedge (\forall y \in Y.\ \exists\ z \in Z.\ y \leq z) \wedge (\exists x \in X.\ \forall\ z \in Z.\ z \leq x)$
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* (*verit*) *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* $(\Downarrow X) = (\Downarrow X \in$ *ideals*$)$
by (*metis* (*mono-tags*, *lifting*) *CollectI Fix-def directed-alt downset-set-idem downclosed-set-def downsets-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* $(\downarrow::'a::order \Rightarrow 'a\ set)$
by (*simp add*: *downset-iso-iff mono-def*)

lemma *mono-downclosed*:
fixes $f :: 'a::order \Rightarrow 'b::order$
assumes *mono* f
shows $\forall Y.$ *downclosed-set* $Y \longrightarrow$ *downclosed-set* $(f -`\ Y)$
by (*simp add*: *assms downclosed-set-iff monoD*)

lemma
fixes $f :: 'a::order \Rightarrow 'b::order$
assumes *mono* f
shows $\forall Y.$ *downclosed-set* $X \longrightarrow$ *downclosed-set* $(f\ `\ X)$
oops

lemma *downclosed-mono*:
fixes f :: $'a$::*order* $\Rightarrow$ $'b$::*order*
assumes $\forall$ Y. *downclosed-set* Y $\longrightarrow$ *downclosed-set* (f $-$‘ Y)
shows *mono f*
proof−
{**fix** x y :: $'a$::*order*
assume h: $x \leq y$
have *downclosed-set* ($\downarrow$ (f y))
unfolding *downclosed-set-def downsets-def Fix-def downset-set-def downset-def*
by *auto*
hence *downclosed-set* (f $-$‘ ($\downarrow$ (f y)))
by (*simp add*: *assms*)
hence *downclosed-set* $\{z. f\ z \leq f\ y\}$
unfolding *vimage-def downset-def downset-set-def* **by** *auto*
hence $\forall$ z w. ($f\ z \leq f\ y \wedge w \leq z$) $\longrightarrow$ $f\ w \leq f\ y$
unfolding *downclosed-set-def downclosed-set-def downsets-def Fix-def downset-set-def*
by *force*
hence $f\ x \leq f\ y$
using h **by** *blast*}
thus *?thesis*..
qed

lemma *mono-downclosed-iff*: *mono f* = ($\forall$ Y. *downclosed-set* Y $\longrightarrow$ *downclosed-set* (f $-$‘ Y))
using *mono-downclosed downclosed-mono* **by** *auto*

context *order*
begin

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

lemma ($X \subseteq Y$) = ($\Downarrow X \subseteq \Downarrow Y$)
oops

end

context *lattice*
begin

lemma *lat-ideals*: $X \in$ *ideals* = ($X \neq \{\}$ $\wedge$ $X \in$ *downsets* $\wedge$ ($\forall$ $x \in X$. $\forall$ $y \in X$. $x \sqcup y \in X$))
unfolding *ideals-def directed-alt downsets-def Fix-def downset-set-def downclosed-set-def*
using *local.sup.bounded-iff local.sup-ge2* **by** *blast*

end

context *bounded-lattice*

begin

lemma *bot-ideal*: $X \in ideals \Longrightarrow \bot \in X$
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*]: $Sup \circ \downarrow = id$
using *Sup-atMost atMost-def downset-prop* **by** *fastforce*

lemma *downset-Sup-id*: $id \leq \downarrow \circ Sup$
by (*simp add*: *Sup-upper downset-prop le-funI subsetI*)

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

lemma *Inf-pres-downset-var*: $(\bigcap x \in X. \downarrow x) = \downarrow(\bigsqcap X)$
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*: $filtered\ X \Longrightarrow X \neq \{\}$
using *filtered-to-directed ord.directed-nonempty* **by** *auto*

lemma *filtered-lb*: $filtered\ X \Longrightarrow (\forall x \in X. \forall y \in X. \exists z \in X. z \leq x \wedge z \leq y)$
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*: $\Uparrow = Union \circ (\text{`}) \uparrow$
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*

lemma *upset-prop2*: $x \leq y \Longrightarrow y \in \uparrow x$
by (*simp add*: *upset-prop*)

lemma *filters-upsets*: $X \in filters \Longrightarrow X \in upsets$
by (*simp add*: *upclosed-set-def filters-def*)

lemma *filters-filtered*: $X \in filters \Longrightarrow filtered\ X$
by (*simp add*: *filters-def*)

end

context *preorder-with-dual*
begin

lemma *filtered-prop*: $X \neq \{\} \Longrightarrow (\forall x \in X. \forall y \in X. \exists z \in X. z \leq x \wedge z \leq y) \Longrightarrow filtered\ X$
unfolding *filtered-to-directed*
by (*rule directed-prop*, *blast*, *metis* (*full-types*) *image-iff ord-dual*)

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 (*metis image-empty directed-alt filtered-to-directed filtered-lb filtered-prop*)

lemma *up-set-prop-var2*: $x \in \Uparrow X \Longrightarrow x \leq y \Longrightarrow y \in \Uparrow X$
using *downset-set-prop-var2 dual-iff ord-dual upset-set-to-downset-set2* **by** *fast-force*

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

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

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

lemma *upset-set-ext*: $id \leq \Uparrow$
by (*smt* (*verit*) *comp-def comp-id image-mono le-fun-def downset-set-ext image-dual upset-set-to-downset-set2*)

lemma *upset-set-anti*: $mono\ \Uparrow$
by (*metis image-mono downset-set-iso upset-set-to-downset-set2 mono-def*)

lemma *up-set-idem* [*simp*]: $\Uparrow \circ \Uparrow = \Uparrow$
by (*metis comp-assoc downset-set-idem upset-set-downset-set-dual upset-set-to-downset-set*)

lemma *upset-faithful*: $\uparrow x \subseteq \uparrow y \Longrightarrow y \leq x$
by (*metis inj-image-subset-iff downset-faithful dual-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 order.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

end

context *lattice-with-dual*
begin

lemma *lat-filters*: $X \in filters = (X \neq \{\} \wedge X \in upsets \wedge (\forall x \in X. \forall\ y \in X. x \sqcap y \in X))$
  unfolding *filters-to-ideals upsets-to-downsets inf-to-sup lat-ideals*
 by (*smt* (*verit*) *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*: $X \in filters \Longrightarrow \top \in X$
  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* [*simp*]: $Inf \circ \uparrow = id$
  by (*metis comp-assoc comp-id Sup-downset-id Sups-dual-def downset-upset-dual invol-dual*)

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

lemma *Sup-Inf-var*: $\bigsqcap (\bigcap x \in X. \uparrow x) = \bigsqcup X$
  unfolding *upset-prop* **by** (*simp add*: *Collect-ball-eq Sup-eq-Inf*)

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

end

4.4 Properties of Complete Lattices

definition *Inf-closed-set* $X = (\forall\ Y \subseteq X. \bigsqcap Y \in X)$

definition *Sup-closed-set* $X = (\forall\ Y \subseteq X. \bigsqcup Y \in X)$

definition *inf-closed-set* $X = (\forall x \in X. \forall y \in X. x \sqcap y \in X)$

definition *sup-closed-set* $X = (\forall x \in X. \forall y \in X. x \sqcup y \in X)$

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*: $x \sqcup y = \bigsqcup\{x,y\}$
  by *simp*

lemma *inf-Inf*: $x \sqcap y = \bigsqcap\{x,y\}$
  by *simp*

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

lemma *fSup-unfold*: $(f::nat \Rightarrow 'a)\ 0 \sqcup (\bigsqcup n.\ f\ (Suc\ n)) = (\bigsqcup n.\ f\ n)$
  apply (*intro order.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*: $(f::nat \Rightarrow 'a)\ 0 \sqcap (\bigsqcap n.\ f\ (Suc\ n)) = (\bigsqcap n.\ f\ n)$
  apply (*intro order.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*)

end

lemma *Sup-sup-closed*: *Sup-closed-set* $(X::'a::complete\text{-}lattice\ set) \Longrightarrow$ *sup-closed-set* X
 by (*metis Sup-closed-set-def empty-subsetI insert-subsetI sup-Sup sup-closed-set-def*)

lemma *Inf-inf-closed*: *Inf-closed-set* $(X::'a::complete\text{-}lattice\ set) \Longrightarrow$ *inf-closed-set* X
  by (*metis Inf-closed-set-def empty-subsetI inf-Inf inf-closed-set-def insert-subset*)

4.5 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* :: $('a::Sup \Rightarrow 'b::Sup) \Rightarrow bool$ **where**
  Sup-pres $f \equiv f \circ Sup = Sup \circ (`)\ f$

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* ⊤ = ⊥

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*))) = (⨆ *y*∈*x*. ∂ (*f* (∂ *y*))) **for** *x*
then have ⨅ (*f* ‘ ∂ ‘ *A*) = *f* (∂ (⨆ *A*)) **for** *A*
by (*metis* (*no-types*) *Sup-dual-def-var image-image invol-dual-var subset-dual*)
then have ⨅ (*f* ‘ *x*) = *f* (⨅ *x*)
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-def-var image-comp*)
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* ⇒ ′*b*::*complete-lattice-with-dual*
  shows *Sup-pres f* = *Inf-pres* (∂_F *f*)
  by (*metis Inf-pres-map-dual-var fun-dual5 map-dual-def*)

lemma *Sup-pres-map-dual*: *Sup-pres* = *Inf-pres* ∘ (∂_F::(′*a*::*complete-lattice-with-dual* ⇒ ′*b*::*complete-lattice-with-dual*) ⇒ ′*a* ⇒ ′*b*)
  by (*simp add*: *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* ⟹ *mono* ((∘) *f*)
  by (*simp add*: *le-fun-def mono-def*)

lemma *fun-isor*: *mono f* ⟹ *mono* (λ*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*, *opaque-lifting*) *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* (*verit*) *INF-insert Inf-empty Inf-insert comp-eq-elim 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 Inf-empty comp-eq-elim*)

lemma *Sup-sup-dual*:
  fixes *f* :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
  shows *Sup-dual f* ⟹ *sup-dual f*
  by (*smt* (*verit*) *comp-eq-elim image-empty image-insert inf-Inf sup-Sup*)

lemma *Inf-inf-dual*:
  fixes *f* :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
  shows *Inf-dual f* ⟹ *inf-dual f*
  by (*smt* (*verit*) *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-iff 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 :: $'a \Rightarrow 'b$::*complete-lattice*
shows $Sup \circ (`)\ f \leq f \circ Sup \Longrightarrow mono\ f$
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 :: $'a \Rightarrow 'b$::*complete-lattice*
shows $(Sup \circ (`)\ f \leq f \circ Sup) = mono\ f$
using *Sup-supdistl-iso iso-Sup-supdistl* **by** *force*

lemma *subdistl-iso*:
fixes f :: $'a \Rightarrow 'b$::*complete-lattice*
shows $(f \circ Inf \leq Inf \circ (`)\ f) = mono\ f$
using *Inf-subdistl-iso iso-Inf-subdistl* **by** *force*

end

lemma *ord-iso-Inf-pres*:
fixes f :: $'a$::*complete-lattice* $\Rightarrow$ $'b$::*complete-lattice*
shows $ord\text{-}iso\ f \Longrightarrow Inf \circ (`)\ f = f \circ Inf$
proof−
let $?g = the\text{-}inv\ f$
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 $(y \leq f\ (\bigsqcap X)) = (?g\ y \leq \bigsqcap X)$
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 ... $= (\forall x \in X.\ ?g\ y \leq x)$
by (*simp add*: *le-Inf-iff*)
also have ... $= (\forall x \in X.\ y \leq f\ x)$
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 ... $= (y \leq \bigsqcap\ (f\ `\ X))$
by (*simp add*: *le-INF-iff*)
finally have $(y \leq f\ (\bigsqcap X)) = (y \leq \bigsqcap\ (f\ `\ X))$.}
hence $f\ (\bigsqcap X) = \bigsqcap\ (f\ `\ X)$
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* $\Rightarrow$ $'b$::*complete-lattice*

shows *ord-iso f* $\Longrightarrow$ *Sup* $\circ$ (‘) *f* = *f* $\circ$ *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 (*f* ($\bigsqcup X$) $\leq$ *y*) = ($\bigsqcup X \leq$ *?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 ... = ($\forall x \in X.\ x \leq$ *?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*)
also have ... = ($\bigsqcup$ (*f* ‘ *X*) $\leq$ *y*)
by (*simp add*: *SUP-le-iff*)
finally have (*f* ($\bigsqcup X$) $\leq$ *y*) = ($\bigsqcup$ (*f* ‘ *X*) $\leq$ *y*).}
hence *f* ($\bigsqcup X$) = $\bigsqcup$ (*f* ‘ *X*)
by (*meson dual-order.antisym order-refl*)}
thus *?thesis*
unfolding *fun-eq-iff* **by** *simp*
qed

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* (*verit*) *INF-apply Inf-fun-def Sup.SUP-cong*)

lemma *fInf-distr-var*: $\bigsqcap F \circ g = (\bigsqcap f \in F.\ f \circ g)$
unfolding *fun-eq-iff comp-def*
by (*smt* (*verit*) *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* ⇒ ′*b*::*complete-lattice*)
shows *Sup* $\circ$ (‘) (($\circ$) *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* $f \Longrightarrow (\bigsqcup g \in G.\ f \circ g) \leq f \circ \bigsqcup G$
by (*simp add*: *fun-isol mono-Sup*)

lemma *fInf-subdistl*:
fixes f :: $'a$::*complete-lattice* $\Rightarrow$ $'b$::*complete-lattice*
shows *mono* $f \Longrightarrow (\circ)\ f \circ Inf \leq Inf \circ (`)\ ((\circ)\ f)$
by (*simp add*: *fun-isol subdistl-iso*)

lemma *fInf-subdistl-var*:
fixes f :: $'a$::*complete-lattice* $\Rightarrow$ $'b$::*complete-lattice*
shows *mono* $f \Longrightarrow f \circ \bigsqcap G \leq (\bigsqcap g \in G.\ f \circ g)$
by (*simp add*: *fun-isol mono-Inf*)

lemma *fSup-distl*: *Sup-pres* $f \Longrightarrow$ *Sup-pres* $((\circ)\ f)$
unfolding *fun-eq-iff* **by** (*simp add*: *image-comp*)

lemma *fSup-distl-var*: *Sup-pres* $f \Longrightarrow f \circ \bigsqcup G = (\bigsqcup g \in G.\ f \circ g)$
unfolding *fun-eq-iff* **by** (*simp add*: *image-comp*)

lemma *fInf-distl*: *Inf-pres* $f \Longrightarrow$ *Inf-pres* $((\circ)\ f)$
unfolding *fun-eq-iff* **by** (*simp add*: *image-comp*)

lemma *fInf-distl-var*: *Inf-pres* $f \Longrightarrow f \circ \bigsqcap G = (\bigsqcap g \in G.\ f \circ g)$
unfolding *fun-eq-iff* **by** (*simp add*: *image-comp*)

Downsets preserve infs whereas upsets preserve sups.

lemma *Inf-pres-downset*: *Inf-pres* ($\downarrow$::$'a$::*complete-lattice-with-dual* $\Rightarrow$ $'a$ *set*)
unfolding *downset-prop fun-eq-iff*
by (*safe*, *simp-all add*: *le-Inf-iff*)

lemma *Sup-dual-upset*: *Sup-dual* ($\uparrow$::$'a$::*complete-lattice-with-dual* $\Rightarrow$ $'a$ *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* $f \Longrightarrow$ *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* $f \Longrightarrow$ *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* $\Rightarrow$ $'a$::*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*)

lift-definition *Sup-iso* :: *'a::complete-lattice iso set* ⇒ *'a iso* **is** *Inf*
by (*smt* (*verit*) *INF-lower2 Inf-apply le-INF-iff mono-def*)

lift-definition *bot-iso* :: *'a::complete-lattice iso* **is** ⊤
by (*simp add*: *monoI*)

lift-definition *sup-iso* :: *'a::complete-lattice iso* ⇒ *'a iso* ⇒ *'a iso* **is** *inf*
by (*smt* (*verit*) *inf-apply inf-mono monoD monoI*)

lift-definition *top-iso* :: *'a::complete-lattice iso* **is** ⊥
by (*simp add*: *mono-def*)

lift-definition *inf-iso* :: *'a::complete-lattice iso* ⇒ *'a iso* ⇒ *'a iso* **is** *sup*
by (*smt* (*verit*) *mono-def sup.mono sup-apply*)

lift-definition *less-eq-iso* :: *'a::complete-lattice iso* ⇒ *'a iso* ⇒ *bool* **is** (≥)**.**

lift-definition *less-iso* :: *'a::complete-lattice iso* ⇒ *'a iso* ⇒ *bool* **is** (>)**.**

instance
by (*intro-classes*; *transfer*, *simp-all add*: *less-fun-def Sup-upper Sup-least Inf-lower Inf-greatest*)

end

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-preseving functions and Sup-lattices, is proved in components on transformers, as more advanced properties about Inf-preserving functions are needed.

4.6 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.

class *complete-heyting-algebra* = *complete-lattice* +
  assumes *ch-dist*: $x \sqcap \bigsqcup Y = (\bigsqcup y \in Y.\ x \sqcap y)$

Complete Heyting algebras are also known as frames or locales (they differ with respect to their morphisms).

class *complete-co-heyting-algebra* = *complete-lattice* +
  assumes *co-ch-dist*: $x \sqcup \bigsqcap Y = (\bigsqcap y \in Y.\ x \sqcup y)$

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 $(x \sqcap \bigsqcup Y \leq t) = (\bigsqcup Y \leq -x \sqcup t)$
      by (*simp add*: *inf.commute shunt1*[*symmetric*])
    also have ... $= (\forall y \in Y.\ y \leq -x \sqcup t)$
      using *Sup-le-iff* **by** *blast*
    also have ... $= (\forall y \in Y.\ x \sqcap y \leq t)$
      by (*simp add*: *inf.commute shunt1*)
    finally have $(x \sqcap \bigsqcup Y \leq t) = ((\bigsqcup y{\in}Y.\ x \sqcap y) \leq t)$
      by (*simp add*: *local.SUP-le-iff*)}
  thus $x \sqcap \bigsqcup Y = (\bigsqcup y{\in}Y.\ x \sqcap y)$
    using *order.eq-iff* **by** *blast*
qed

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

lemma *de-morgan1*: $-(\bigsqcup X) = (\bigsqcap x \in X.\ -x)$
proof−
  {**fix** y
  have $(y \leq -(\bigsqcup X)) = (\bigsqcup X \leq -y)$
    using *compl-le-swap1* **by** *blast*
  also have ... $= (\forall x \in X.\ x \leq -y)$
    by (*simp add*: *Sup-le-iff*)
  also have ... $= (\forall x \in X.\ y \leq -x)$
    using *compl-le-swap1* **by** *blast*
  also have ... $= (y \leq (\bigsqcap x \in X.\ -x))$
    using *le-INF-iff* **by** *force*

finally have $(y \leq -(\bigsqcup X)) = (y \leq (\bigsqcap x \in X. -x))$.}
thus *?thesis*
using *order.antisym* **by** *blast*
qed

lemma *de-morgan2*: $-(\bigsqcap X) = (\bigsqcup x \in X. -x)$
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

4.7 Atomic Boolean Algebras

Next, atomic boolean algebras are defined.

context *bounded-lattice*
begin

Atoms are covers of bottom.

definition *atom* $x = (x \neq \bot \wedge \neg(\exists y. \bot < y \wedge y < x))$

definition *atom-map* $x = \{y.$ *atom* $y \wedge y \leq x\}$

lemma *atom-map-def-var*: *atom-map* $x = \downarrow x \cap$ *Collect atom*
unfolding *atom-map-def downset-def downset-set-def comp-def atom-def* **by** *fastforce*

lemma *atom-map-atoms*: $\bigcup$(*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* ∘ *atom-map*

class *atomic-boolean-algebra* = *boolean-algebra* +
assumes *atomicity*: $x \neq \bot \Longrightarrow (\exists\, y.\ atom\ y \wedge y \leq x)$

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*: $atom\ x \Longrightarrow x \neq \bot \wedge (\forall\, y\ z.\ x \leq y \vee x \leq -y)$
by (*auto simp add*: *atom-def*) (*metis local.dual-order.not-eq-order-implies-strict local.inf.cobounded1 local.inf.cobounded2 local.inf-shunt*)

lemma *atom-sup*: $(\forall\, y.\ x \leq y \vee x \leq -y) \Longrightarrow (\forall\, y\ z.\ (x \leq y \vee x \leq z) = (x \leq y \sqcup z))$
by (*metis inf.orderE le-supI1 shunt2*)

lemma *sup-atom*: $x \neq \bot \Longrightarrow (\forall\, y\ z.\ (x \leq y \vee x \leq z) = (x \leq y \sqcup z)) \Longrightarrow atom\ x$
by (*auto simp add*: *atom-def*) (*metis* (*full-types*) *local.inf.boundedI local.inf.cobounded2 local.inf-shunt local.inf-sup-ord*(*4*) *local.le-iff-sup local.shunt1 local.sup.absorb1 local.sup.strict-order-iff*)

lemma *atom-sup-iff*: $atom\ x = (x \neq \bot \wedge (\forall\, y\ z.\ (x \leq y \vee x \leq z) = (x \leq y \sqcup z)))$
by *rule* (*auto simp add*: *atom-neg atom-sup sup-atom*)

lemma *atom-neg-iff*: $atom\ x = (x \neq \bot \wedge (\forall\, y\ z.\ x \leq y \vee x \leq -y))$
by *rule* (*auto simp add*: *atom-neg atom-sup sup-atom*)

lemma *atom-map-bot-pres*: $atom\text{-}map\ \bot = \{\}$

using *atom-def atom-map-def le-bot* **by** *auto*

lemma *atom-map-top-pres*: *atom-map* $\top$ = *Collect atom*
using *atom-map-def* **by** *auto*

end

context *complete-boolean-algebra-alt*
begin

lemma *atom-Sup*: $\bigwedge Y.\ x \neq \bot \Longrightarrow (\forall y.\ x \leq y \vee x \leq -y) \Longrightarrow ((\exists y \in Y.\ x \leq y) = (x \leq \bigsqcup Y))$
by (*metis Sup-least Sup-upper2 compl-le-swap1 le-iff-inf inf-shunt*)

lemma *Sup-atom*: $x \neq \bot \Longrightarrow (\forall Y.\ (\exists y \in Y.\ x \leq y) = (x \leq \bigsqcup Y)) \Longrightarrow$ *atom x*
proof−
assume *h1*: $x \neq \bot$
and *h2*: $\forall Y.\ (\exists y \in Y.\ x \leq y) = (x \leq \bigsqcup Y)$
hence $\forall y\ z.\ (x \leq y \vee x \leq z) = (x \leq y \sqcup z)$
by (*smt* (*verit*) *insert-iff sup-Sup sup-bot.right-neutral*)
thus *atom x*
by (*simp add*: *h1 sup-atom*)
qed

lemma *atom-Sup-iff*: *atom x* = $(x \neq \bot \wedge (\forall Y.\ (\exists y \in Y.\ x \leq y) = (x \leq \bigsqcup Y)))$
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* (↓::*'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* ∘ ↓

In a complete lattice, its inverse is Sup.

definition *SSup* :: *'a*::*complete-lattice downset* ⇒ *'a* **where**
  SSup = *Sup* ∘ *Rep-downset*

lemma *ds-SSup-inv*: *ds* ∘ *SSup* = (*id*::*'a*::*complete-lattice downset* ⇒ *'a downset*)
  unfolding *ds-def SSup-def*
  by (*smt* (*verit*) *Rep-downset Rep-downset-inverse cSup-atMost eq-id-iff imageE o-def ord-class.atMost-def ord-class.downset-prop*)

lemma *SSup-ds-inv*: *SSup* ∘ *ds* = (*id*::*'a*::*complete-lattice* ⇒ *'a*)
 unfolding *ds-def SSup-def fun-eq-iff id-def comp-def* **by** (*simp add*: *Abs-downset-inverse pointfree-idE*)

instantiation *downset* :: (*ord*) *order*
begin

lift-definition *less-eq-downset* :: *'a downset* ⇒ *'a downset* ⇒ *bool* **is** (λ*X Y*. *Rep-downset X* ⊆ *Rep-downset Y*) **.**

lift-definition *less-downset* :: *'a downset* ⇒ *'a downset* ⇒ *bool* **is** (λ*X Y*. *Rep-downset X* ⊂ *Rep-downset Y*) **.**

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

end

lemma *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 *ds-faithful*: *ds x* ≤ *ds y* ⟹ *x* ≤ (*y*::*'a*::*order*)
 by (*simp add*: *Abs-downset-inverse downset-faithful ds-def less-eq-downset.rep-eq*)

lemma *ds-inj*: *inj* (*ds*::*'a*::*order* ⇒ *'a downset*)
  by (*simp add*: *ds-faithful dual-order.antisym injI*)

lemma *ds-surj*: *surj ds*
  by (*metis* (*no-types*, *opaque-lifting*) *Rep-downset Rep-downset-inverse ds-def image-iff o-apply surj-def*)

lemma *ds-bij*: *bij* (*ds*::′*a*::*order* ⇒ ′*a downset*)
 by (*simp add*: *bijI ds-inj ds-surj*)

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

The morphishms between orderings and downsets are isotone functions. One can define functors mapping back and forth between these.

definition *map-ds* :: (′*a*::*complete-lattice* ⇒ ′*b*::*complete-lattice*) ⇒ (′*a downset* ⇒ ′*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 *map-ds f* ∘ *ds* = *ds* ∘ *f*
 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 *map-ds f* ∘ *ds* = *ds* ∘ *id f*
 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-ds-iso-pres*:
 fixes *f* :: ′*a*::*complete-lattice* ⇒ ′*b*::*complete-lattice*
 shows *mono f* ⟹ *mono* (*map-ds f*)
 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-iso-pres*:
 fixes *F* :: ′*a*::*complete-lattice downset* ⇒ ′*b*::*complete-lattice downset*

shows *mono F* $\Longrightarrow$ *mono* (*map-SSup F*)
unfolding *fun-eq-iff mono-def map-SSup-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* $\Rightarrow$ *'b*::*complete-lattice downset*
shows *ds* $\circ$ *map-SSup F* = *F* $\circ$ *ds*
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* $\Rightarrow$ *'b*::*complete-lattice downset*
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* $\Rightarrow$ *'b*::*complete-lattice downset*
and *G* :: *'a*::*complete-lattice downset* $\Rightarrow$ *'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-SSup-map-ds-inv*: *map-SSup* $\circ$ *map-ds* = (*id*::(*'a*::*complete-lattice* $\Rightarrow$ *'b*::*complete-lattice*) $\Rightarrow$ (*'a* $\Rightarrow$ *'b*))
by (*metis* (*no-types*, *opaque-lifting*) *SSup-ds-inv comp-def eq-id-iff fun.map-comp fun.map-id0 id-apply map-SSup-prop map-ds-prop*)

lemma *map-ds-map-SSup-inv*: *map-ds* $\circ$ *map-SSup* = (*id*::(*'a*::*complete-lattice downset* $\Rightarrow$ *'b*::*complete-lattice downset*) $\Rightarrow$ (*'a downset* $\Rightarrow$ *'b downset*))
unfolding *map-SSup-def map-ds-def SSup-def ds-def id-def comp-def fun-eq-iff*
by (*metis* (*no-types*, *lifting*) *Rep-downset Rep-downset-inverse Sup-downset-id image-iff pointfree-idE*)

lemma *inj-map-ds*: *inj* (*map-ds*::(*'a*::*complete-lattice* $\Rightarrow$ *'b*::*complete-lattice*) $\Rightarrow$ (*'a downset* $\Rightarrow$ *'b downset*))
by (*metis* (*no-types*, *lifting*) *SSup-ds-inv fun.map-id0 id-comp inj-def map-ds-prop*

rewriteR-comp-comp2)

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

lemma *map-ds-map-SSup-iff*:
fixes *g* :: '*a*::*complete-lattice* ⇒ '*b*::*complete-lattice*
shows (*f* = *map-ds g*) = (*map-SSup f* = *g*)
by (*metis inj-eq inj-map-ds map-ds-map-SSup-inv pointfree-idE*)

This gives an isomorphism between categories.

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*, *opaque-lifting*) *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 up-set*)
unfolding *us-def IInf-def fun-eq-iff id-def comp-def*

by (*metis* (*no-types*, *lifting*) *Inf-upset-id Rep-upset Rep-upset-inverse f-the-inv-into-f pointfree-idE upset-inj*)

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-Inf-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** (λ*X Y*. *Rep-upset X* ⊇ *Rep-upset Y*) **.**

lift-definition *less-upset* :: ′*a upset* ⇒ ′*a upset* ⇒ *bool* **is** (λ*X 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 us-def*)

lemma *us-faithful*: *us x* ≤ *us y* ⟹ *x* ≤ (*y*::′*a*::*order-with-dual*)
by (*simp add*: *Abs-upset-inverse upset-faithful us-def less-eq-upset.rep-eq*)

lemma *us-inj*: *inj* (*us*::′*a*::*order-with-dual* ⇒ ′*a upset*)
unfolding *inj-def* **by** (*simp add*: *us-faithful dual-order.antisym*)

lemma *us-surj*: *surj* (*us*::′*a*::*order-with-dual* ⇒ ′*a upset*)
unfolding *surj-def* **by** (*metis Rep-upset Rep-upset-inverse comp-def image-iff us-def*)

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 us-faithful us-inj us-surj*)

definition *map-us* :: (′*a*::*complete-lattice* ⇒ ′*b*::*complete-lattice*) ⇒ (′*a upset* ⇒ ′*b upset*) **where**
map-us f = *us* ∘ *f* ∘ *IInf*

lemma *map-us-prop*: *map-us f* ∘ (*us*::′*a*::*complete-lattice-with-dual* ⇒ ′*a upset*) = *us* ∘ *id f*

unfolding *map-us-def* **by** (*simp add*: *IInf-us-inv comp-assoc*)

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

lemma *map-IInf-prop*: (*us*::*'a*::*complete-lattice-with-dual* ⇒ *'a upset*) ∘ *map-IInf F* = *id F* ∘ *us*
proof−
have *us* ∘ *map-IInf F* = (*us* ∘ *IInf*) ∘ *F* ∘ *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*: *map-IInf id* = (*id*::*'a*::*complete-lattice-with-dual* ⇒ *'a*)
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 upset*
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 *map-IInf-map-us-inv*: *map-IInf* ∘ *map-us* = (*id*::(*'a*::*complete-lattice-with-dual* ⇒ *'b*::*complete-lattice-with-dual*) ⇒ (*'a* ⇒ *'b*))
unfolding *map-IInf-def map-us-def IInf-def us-def id-def comp-def fun-eq-iff* **by** (*simp add*: *Abs-upset-inverse pointfree-idE*)

lemma *map-us-map-IInf-inv*: *map-us* ∘ *map-IInf* = (*id*::(*'a*::*complete-lattice-with-dual upset* ⇒ *'b*::*complete-lattice-with-dual upset*) ⇒ (*'a upset* ⇒ *'b upset*))
unfolding *map-IInf-def map-us-def IInf-def us-def id-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-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*, *opaque-lifting*) *Abs-upset-inverse Inf-upset-id pointfree-idE rangeI*)

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*, *opaque-lifting*) *Inf-upset-id Rep-upset Rep-upset-inverse image-iff pointfree-idE*)

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 upset*
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 ($y \in$ *atom-map* $(-x)$) = (*atom* $y \wedge y \leq -x$)
    by (*simp add*: *atom-map-def*)
  also have ... = (*atom* $y \wedge \neg(y \leq x)$)
   by (*metis atom-sup-iff inf.orderE inf-shunt sup-compl-top top.ordering-top-axioms ordering-top.extremum*)
   also have ... = ($y \in$ *Collect atom* − *atom-map* x)
     using *atom-map-def* **by** *auto*
   finally have ($y \in$ *atom-map* $(-x)$) = ($y \in$ *Collect atom* − *atom-map* x).}
  thus *?thesis*
    by *blast*
qed

lemma *atom-map-sup-pres*: *atom-map* $(x \sqcup y)$ = *atom-map* x ∪ *atom-map* y
proof−
  {**fix** z
  have ($z \in$ *atom-map* $(x \sqcup y)$) = (*atom* $z \wedge z \leq x \sqcup y$)
    by (*simp add*: *atom-map-def*)
  also have ... = (*atom* $z \wedge (z \leq x \vee z \leq y)$)
    using *atom-sup-iff* **by** *auto*
  also have ... = ($z \in$ *atom-map* x ∪ *atom-map* y)
    using *atom-map-def* **by** *auto*
  finally have ($z \in$ *atom-map* $(x \sqcup y)$) = ($z \in$ *atom-map* x ∪ *atom-map* y)
    by *blast*}
  thus *?thesis*
    by *blast*
qed

lemma *atom-map-inf-pres*: *atom-map* $(x \sqcap y)$ = *atom-map* x ∩ *atom-map* y
  by (*smt* (*verit*) *Diff-Un atom-map-compl-pres atom-map-sup-pres compl-inf double-compl*)

lemma *atom-map-minus-pres*: *atom-map* $(x - y)$ = *atom-map* x − *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* ⇒ $'a$ *atoms* ⇒ $'a$ *atoms* **is** $\lambda x\ y$. *Abs-atoms* (*Rep-atoms* x − *Rep-atoms* y).

lift-definition *uminus-atoms* :: *'a atoms* ⇒ *'a atoms* **is** *λx. Abs-atoms* (*Collect atom* − *Rep-atoms x*).

lift-definition *bot-atoms* :: *'a atoms* **is** *Abs-atoms* {}.

lift-definition *sup-atoms* :: *'a atoms* ⇒ *'a atoms* ⇒ *'a atoms* **is** *λx y. Abs-atoms* (*Rep-atoms x* ∪ *Rep-atoms y*).

lift-definition *top-atoms* :: *'a atoms* **is** *Abs-atoms* (*Collect atom*).

lift-definition *inf-atoms* :: *'a atoms* ⇒ *'a atoms* ⇒ *'a atoms* **is** *λx y. Abs-atoms* (*Rep-atoms x* ∩ *Rep-atoms y*).

lift-definition *less-eq-atoms* :: *'a atoms* ⇒ *'a atoms* ⇒ *bool* **is** (*λx y. Rep-atoms x* ⊆ *Rep-atoms y*).

lift-definition *less-atoms* :: *'a atoms* ⇒ *'a atoms* ⇒ *bool* **is** (*λx y. Rep-atoms x* ⊂ *Rep-atoms y*).

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* (*verit*) *Abs-atoms-inverse Rep-atoms atom-map-inf-pres image-iff inf-le1 rangeI*)
apply (*transfer*, *smt* (*verit*) *Abs-atoms-inverse Rep-atoms atom-map-inf-pres image-iff inf-le2 rangeI*)
apply (*transfer*, *smt* (*verit*) *Abs-atoms-inverse Rep-atoms atom-map-inf-pres image-iff le-iff-sup rangeI sup-inf-distrib1*)
apply (*transfer*, *smt* (*verit*) *Abs-atoms-inverse Rep-atoms atom-map-sup-pres image-iff image-iff inf.orderE inf-sup-aci*(*6*) *le-iff-sup order-refl rangeI rangeI*)
apply (*transfer*, *smt* (*verit*) *Abs-atoms-inverse Rep-atoms atom-map-sup-pres image-iff inf-sup-aci*(*6*) *le-iff-sup rangeI sup.left-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*, *metis Abs-atoms-inverse Rep-atoms atom-map-compl-pres atom-map-top-pres diff-eq double-compl inf-le1 rangeE rangeI*)
apply (*transfer*, *smt* (*verit*, *ccfv-threshold*) *Abs-atoms-inverse Rep-atoms atom-map-inf-pres atom-map-sup-pres image-iff rangeI sup-inf-distrib1*)
apply (*transfer*, *metis* (*no-types*, *opaque-lifting*) *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*)
apply (*transfer*, *smt Abs-atoms-inverse Rep-atoms atom-map-compl-pres atom-map-inf-pres*

atom-map-minus-pres diff-eq rangeE rangeI)
  done

end

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

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*)

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 $x \sqcap -y \neq \bot \vee -x \sqcap y \neq \bot$
    by (*auto simp*: *inf-shunt*)
  hence $\exists z.$ *atom* $z \wedge (z \leq x \sqcap -y \vee z \leq -x \sqcap y)$
    using *atomicity* **by** *blast*
  hence $\exists z.$ *atom* $z \wedge ((z \in$ *atom-map* $x \wedge \neg(z \in$ *atom-map* $y)) \vee (\neg(z \in$ *atom-map* $x) \wedge z \in$ *atom-map* $y))$
    unfolding *atom-def atom-map-def* **by** (*clarsimp*, *metis diff-eq inf.orderE diff-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* $\Longrightarrow$ *atom-map x* = $\{x\}$
 unfolding *atom-def atom-map-def* **by** (*auto simp*: *bot-less dual-order.order-iff-strict*)

lemma *atom-map-atom-pres2*: *atom x* $\Longrightarrow$ *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

end

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-atom-map inj-def rangeI*)

lemma *at-map-atom-pres*: *atom* (*x*::′*a*::*atomic-boolean-algebra*) $\Longrightarrow$ *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*) $\Longrightarrow$ *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 $x \neq \bot$
 hence $\exists\, y.\ x =$ *at-map* $y \wedge x \neq \bot$
  unfolding *at-map-def comp-def* **by** (*metis Abs-atoms-cases rangeE*)
 hence $\exists\, y.\ x =$ *at-map* $y \wedge (\exists\, z.$ *atom* $z \wedge z \leq y)$
  using *at-map-bot-pres atomicity* **by** *force*
 hence $\exists\, y.\ x =$ *at-map* $y \wedge (\exists\, z.$ *atom* (*at-map* z) $\wedge$ *at-map* $z \leq$ *at-map* $y)$
  by (*metis at-map-atom-pres2 at-map-sup-pres sup.orderE sup-ge2*)
 thus $\exists\, y.$ *atom* $y \wedge y \leq x$
  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*: $Y \subseteq$ *Collect atom* $\Longrightarrow Y =$ *atom-map* $(\bigsqcup Y)$
proof
  assume $Y \subseteq$ *Collect atom*
  thus $Y \subseteq$ *atom-map* $(\bigsqcup Y)$
    using *Sup-upper atom-map-def* **by** *force*
next
  assume $Y \subseteq$ *Collect atom*
  hence *a*: $\forall y.\ y \in Y \longrightarrow$ *atom y*
    by *blast*
  {fix *z*
  assume *h*: $z \in$ *Collect atom* $- Y$
  hence $\forall y \in Y.\ y \sqcap z = \bot$
   by (*metis DiffE a h atom-def dual-order.not-eq-order-implies-strict inf.absorb-iff2 inf-le2 inf-shunt mem-Collect-eq*)
  hence $\bigsqcup Y \sqcap z = \bot$
    using *Sup-least inf-shunt* **by** *simp*
  hence $z \notin$ *atom-map* $(\bigsqcup Y)$
    using *atom-map-bot-pres atom-map-def* **by** *force***}**
  thus *atom-map* $(\bigsqcup Y) \subseteq Y$
    using *atom-map-def* **by** *force*
qed

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

lemma *atom-map-Sup-pres*: *atom-map* $(\bigsqcup X) = (\bigcup x \in X.$ *atom-map* $x)$
proof−
  {fix *z*
  have $(z \in$ *atom-map* $(\bigsqcup X)) = ($*atom* $z \wedge z \leq \bigsqcup X)$
    by (*simp add*: *atom-map-def*)
  also have ... $= ($*atom* $z \wedge (\exists x \in X.\ z \leq x))$
    using *atom-Sup-iff* **by** *auto*
  also have ... $= (z \in (\bigcup x \in X.$ *atom-map* $x))$
    using *atom-map-def* **by** *auto*
  finally have $(z \in$ *atom-map* $(\bigsqcup X)) = (z \in (\bigcup x \in X.$ *atom-map* $x))$
    by *blast***}**
  thus *?thesis*
    by *blast*
qed

lemma *atom-map-Sup-pres-var*: *atom-map* $\circ$ *Sup* $=$ *Sup* $\circ$ (') *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*: *atom-map* ($\bigsqcap X$) = *Collect atom* ∩ ($\bigcap x \in X$. *atom-map* x)
proof−
have *atom-map* ($\bigsqcap X$) = *atom-map* ($-(\bigsqcup x \in X. -x)$)
by (*smt Collect-cong SUP-le-iff atom-map-def compl-le-compl-iff compl-le-swap1 le-Inf-iff*)
also have ... = *Collect atom* − *atom-map* ($\bigsqcup x \in X. -x$)
using *atom-map-compl-pres* **by** *blast*
also have ... = *Collect atom* − ($\bigcup x \in X$. *atom-map* ($-x$))
by (*simp add*: *atom-map-Sup-pres*)
also have ... = *Collect atom* − ($\bigcup x \in X$. *Collect atom* − *atom-map* (x))
using *atom-map-compl-pres* **by** *blast*
also have ... = *Collect atom* ∩ ($\bigcap x \in X$. *atom-map* x)
by *blast*
finally show *?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** λ*X*. *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** λ*X*. *Abs-atoms* (*Union* ((‘) *Rep-atoms X*)).

instance
apply (*intro-classes*; *transfer*)
apply (*metis* (*no-types*, *opaque-lifting*) *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* ($\bigsqcup X$) = ($\bigsqcup$(*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* ($\bigsqcap X$) = *Abs-atoms* (*Collect atom* ⊓ ($\bigsqcap x \in X$. (*Rep-atoms* (*at-map* (*x*::′*a*::*complete-boolean-algebra-alt*)))))
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.

end

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* :: $('a::ord \Rightarrow 'b::ord) \Rightarrow ('b \Rightarrow 'a) \Rightarrow bool$ (**infixl** ‹⊣› *70*) **where**
$(f \dashv g) = (\forall x\ y.\ (f\ x \leq y) = (x \leq g\ y))$

definition *ladj* $(g::'a::Inf \Rightarrow 'b::ord) = (\lambda x.\ \bigsqcap\{y.\ x \leq g\ y\})$

definition *radj* $(f::'a::Sup \Rightarrow 'b::ord) = (\lambda y.\ \bigsqcup\{x.\ f\ x \leq y\})$

lemma *ladj-radj-dual*:
fixes f :: $'a::complete\text{-}lattice\text{-}with\text{-}dual \Rightarrow 'b::ord\text{-}with\text{-}dual$
shows $ladj\ f\ x = \partial\ (radj\ (\partial_F\ f)\ (\partial\ x))$
proof−
have $ladj\ f\ x = \partial\ (\bigsqcup(\partial\ `\ \{y.\ \partial\ (f\ y) \leq \partial\ x\}))$
unfolding *ladj-def* **by** (*metis* (*no-types*, *lifting*) *Collect-cong Inf-dual-var dual-dual-ord dual-iff*)
also have ... = $\partial\ (\bigsqcup\{\partial\ y|y.\ \partial\ (f\ y) \leq \partial\ x\})$
by (*simp add*: *setcompr-eq-image*)
ultimately show *?thesis*
unfolding *ladj-def radj-def map-dual-def comp-def*
by (*smt* (*verit*) *Collect-cong invol-dual-var*)
qed

lemma *radj-ladj-dual*:
fixes f :: $'a::complete\text{-}lattice\text{-}with\text{-}dual \Rightarrow 'b::ord\text{-}with\text{-}dual$
shows $radj\ f\ x = \partial\ (ladj\ (\partial_F\ f)\ (\partial\ x))$
by (*metis fun-dual5 invol-dual-var ladj-radj-dual map-dual-def*)

lemma *ladj-prop*:
fixes g :: $'b::Inf \Rightarrow 'a::ord\text{-}with\text{-}dual$
shows $ladj\ g = Inf \circ (-`)\ g \circ \uparrow$
unfolding *ladj-def vimage-def upset-prop fun-eq-iff comp-def* **by** *simp*

lemma *radj-prop*:
fixes f :: $'b::Sup \Rightarrow 'a::ord$
shows $radj\ f = Sup \circ (-`)\ f \circ \downarrow$

unfolding *radj-def vimage-def downset-prop fun-eq-iff comp-def* **by** *simp*

The first set of properties holds without any sort assumptions.

lemma *adj-iso1*: $f \dashv g \Longrightarrow mono\ f$
unfolding *adj-def mono-def* **by** (*meson dual-order.refl dual-order.trans*)

lemma *adj-iso2*: $f \dashv g \Longrightarrow mono\ g$
unfolding *adj-def mono-def* **by** (*meson dual-order.refl dual-order.trans*)

lemma *adj-comp*: $f \dashv g \Longrightarrow adj\ h\ k \Longrightarrow (f \circ h) \dashv (k \circ g)$
by (*simp add*: *adj-def*)

lemma *adj-dual*:
fixes f :: $'a$::*ord-with-dual* $\Rightarrow$ $'b$::*ord-with-dual*
shows $f \dashv g = (\partial_F\ g) \dashv (\partial_F\ f)$
unfolding *adj-def map-dual-def comp-def* **by** (*metis* (*mono-tags*, *opaque-lifting*) *dual-dual-ord invol-dual-var*)

6.2 Properties for (Pre)Orders

The next set of properties holds in preorders or orders.

lemma *adj-cancel1*:
fixes f :: $'a$::*preorder* $\Rightarrow$ $'b$::*ord*
shows $f \dashv g \Longrightarrow f \circ g \leq id$
by (*simp add*: *adj-def le-funI*)

lemma *adj-cancel2*:
fixes f :: $'a$::*ord* $\Rightarrow$ $'b$::*preorder*
shows $f \dashv g \Longrightarrow id \leq g \circ f$
by (*simp add*: *adj-def eq-iff le-funI*)

lemma *adj-prop*:
fixes f :: $'a$::*preorder* $\Rightarrow$ $'a$
shows $f \dashv g \Longrightarrow f \circ g \leq g \circ f$
using *adj-cancel1 adj-cancel2 order-trans* **by** *blast*

lemma *adj-cancel-eq1*:
fixes f :: $'a$::*preorder* $\Rightarrow$ $'b$::*order*
shows $f \dashv g \Longrightarrow f \circ g \circ f = f$
unfolding *adj-def comp-def fun-eq-iff* **by** (*meson eq-iff order-refl order-trans*)

lemma *adj-cancel-eq2*:
fixes f :: $'a$::*order* $\Rightarrow$ $'b$::*preorder*
shows $f \dashv g \Longrightarrow g \circ f \circ g = g$
unfolding *adj-def comp-def fun-eq-iff* **by** (*meson eq-iff order-refl order-trans*)

lemma *adj-idem1*:
fixes f :: $'a$::*preorder* $\Rightarrow$ $'b$::*order*
shows $f \dashv g \Longrightarrow (f \circ g) \circ (f \circ g) = f \circ g$

by (*simp add*: *adj-cancel-eq1 rewriteL-comp-comp*)

lemma *adj-idem2*:
fixes f :: $'a$::*preorder* ⇒ $'b$::*preorder*
shows $f \dashv g \Longrightarrow (g \circ f) \circ (g \circ f) = g \circ f$
by (*simp add*: *adj-cancel-eq2 rewriteL-comp-comp*)

lemma *adj-iso3*:
fixes f :: $'a$::*order* ⇒ $'b$::*order*
shows $f \dashv g \Longrightarrow$ *mono* $(f \circ g)$
by (*simp add*: *adj-iso1 adj-iso2 monoD monoI*)

lemma *adj-iso4*:
fixes f :: $'a$::*order* ⇒ $'b$::*order*
shows $f \dashv g \Longrightarrow$ *mono* $(g \circ f)$
by (*simp add*: *adj-iso1 adj-iso2 monoD monoI*)

lemma *adj-canc1*:
fixes f :: $'a$::*order* ⇒ $'b$::*ord*
shows $f \dashv g \Longrightarrow ((f \circ g)\ x = (f \circ g)\ y \longrightarrow g\ x = g\ y)$
unfolding *adj-def comp-def* **by** (*metis eq-iff*)

lemma *adj-canc2*:
fixes f :: $'a$::*ord* ⇒ $'b$::*order*
shows $f \dashv g \Longrightarrow ((g \circ f)\ x = (g \circ f)\ y \longrightarrow f\ x = f\ y)$
unfolding *adj-def comp-def* **by** (*metis eq-iff*)

lemma *adj-sur-inv*:
fixes f :: $'a$::*preorder* ⇒ $'b$::*order*
shows $f \dashv g \Longrightarrow ((surj\ f) = (f \circ g = id))$
unfolding *adj-def surj-def comp-def* **by** (*metis eq-id-iff eq-iff order-refl order-trans*)

lemma *adj-surj-inj*:
fixes f :: $'a$::*order* ⇒ $'b$::*order*
shows $f \dashv g \Longrightarrow ((surj\ f) = (inj\ g))$
unfolding *adj-def inj-def surj-def* **by** (*metis eq-iff order-trans*)

lemma *adj-inj-inv*:
fixes f :: $'a$::*preorder* ⇒ $'b$::*order*
shows $f \dashv g \Longrightarrow ((inj\ f) = (g \circ f = id))$
by (*metis adj-cancel-eq1 eq-id-iff inj-def o-apply*)

lemma *adj-inj-surj*:
fixes f :: $'a$::*order* ⇒ $'b$::*order*
shows $f \dashv g \Longrightarrow ((inj\ f) = (surj\ g))$
unfolding *adj-def inj-def surj-def* **by** (*metis eq-iff order-trans*)

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

lemma *inj-id-the-inv*: *inj f* $\Longrightarrow$ *f* $\circ$ *g* = *id* $\Longrightarrow$ *f* = *the-inv g*
proof –
  assume *a1*: *inj f*
  assume *f* $\circ$ *g* = *id*
  hence $\forall$ *x*. *the-inv g x* = *f x*
    using *a1* **by** (*metis* (*no-types*) *comp-apply eq-id-iff inj-on-id inj-on-imageI2 the-inv-f-f*)
  thus *?thesis*
    by *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 *radj-Inf-pres*:
  fixes *g* :: *'b::complete-lattice* $\Rightarrow$ *'a::complete-lattice*
  shows ($\exists$ *f*. *f* $\dashv$ *g*) $\Longrightarrow$ *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* $\dashv$ *g*) $\Longrightarrow$ *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* $\dashv$ *g* $\Longrightarrow$ *g* = (*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* $\dashv$ *g* $\Longrightarrow$ *f* = (*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* :: *'a::complete-lattice* $\Rightarrow$ *'b::complete-lattice*
  shows *Inf-pres g* $\Longrightarrow$ (*ladj g*) $\dashv$ *g*
proof–
  assume *a*: *Inf-pres g*
  {**fix** *x y*
   assume *b*: *ladj g x* $\leq$ *y*
  hence *g* (*ladj g x*) $\leq$ *g y*
    by (*simp add*: *Inf-subdistl-iso a monoD*)

hence $\bigsqcap\{g\ y\ |y.\ x \leq g\ y\} \leq g\ y$
by (*metis a comp-eq-dest-lhs setcompr-eq-image ladj-def*)
hence $x \leq g\ y$
using *dual-order.trans le-Inf-iff* **by** *blast*
hence *ladj g x* $\leq y \longrightarrow x \leq g\ y$
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* ⇒ *'b::complete-lattice-with-dual*
shows *Sup-pres f* ⟹ *f* ⊣ (*radj f*)
by (*metis* (*no-types*, *opaque-lifting*) *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* ⇒ *'a::complete-lattice*
shows *Inf-pres g* ⟹ (∃*f*. *f* ⊣ *g*)
using *Inf-pres-radj-aux* **by** *fastforce*

lemma *Sup-pres-ladj*:
fixes *f* :: *'a::complete-lattice-with-dual* ⇒ *'b::complete-lattice-with-dual*
shows *Sup-pres f* ⟹ (∃*g*. *f* ⊣ *g*)
using *Sup-pres-ladj-aux* **by** *fastforce*

lemma *Inf-pres-upper-adj-eq*:
fixes *g* :: *'b::complete-lattice* ⇒ *'a::complete-lattice*
shows (*Inf-pres g*) = (∃*f*. *f* ⊣ *g*)
using *radj-Inf-pres Inf-pres-radj* **by** *blast*

lemma *Sup-pres-ladj-eq*:
fixes *f* :: *'a::complete-lattice-with-dual* ⇒ *'b::complete-lattice-with-dual*
shows (*Sup-pres f*) = (∃*g*. *f* ⊣ *g*)
using *Sup-pres-ladj ladj-Sup-pres* **by** *blast*

lemma *Sup-downset-adj*: (*Sup::'a::complete-lattice set* ⇒ *'a*) ⊣ ↓
unfolding *adj-def downset-prop Sup-le-iff* **by** *force*

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

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

end

7 Fixpoint Fusion

theory *Fixpoint-Fusion*

imports *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 *lfp-Fix*:
  fixes *f* :: *'a::complete-lattice-with-dual* ⇒ *'a*
  shows *mono f* ⟹ *lfp f* = ⨅(*Fix f*)
  unfolding *lfp-def Fix-def*
  apply (*rule antisym*)
  apply (*simp add*: *Collect-mono Inf-superset-mono*)
  by (*metis* (*mono-tags*) *Inf-lower lfp-def lfp-unfold mem-Collect-eq*)

lemma *gfp-Fix*:
  fixes *f* :: *'a::complete-lattice-with-dual* ⇒ *'a*
  shows *mono f* ⟹ *gfp f* = ⨆(*Fix f*)
  by (*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* ⇒ *'a*
  and *g* :: *'b::complete-lattice* ⇒ *'b*
  assumes *mono f*
  assumes *h* ∘ *f* ≤ *g* ∘ *h*
  shows *h* (*gfp f*) ≤ *gfp g*
proof−
  have *h* (*f* (*gfp f*)) ≤ *g* (*h* (*gfp f*))
    using *assms*(*2*) *le-funD* **by** *fastforce*
  hence *h* (*gfp f*) ≤ *g* (*h* (*gfp f*))
    by (*simp add*: *assms*(*1*) *gfp-fixpoint*)
  thus *h* (*gfp f*) ≤ *gfp g*
    by (*simp add*: *gfp-upperbound*)
qed

lemma *lfp-little-fusion*:
  fixes *f* :: *'a::complete-lattice-with-dual* ⇒ *'a*
  and *g* :: *'b::complete-lattice-with-dual* ⇒ *'b*
  assumes *mono f*
  assumes *g* ∘ *h* ≤ *h* ∘ *f*
  shows *lfp g* ≤ *h* (*lfp f*)
proof−
  have *a*: *mono* (*map-dual f*)
    by (*simp add*: *assms iso-map-dual*)
  have *map-dual h* ∘ *map-dual f* ≤ *map-dual g* ∘ *map-dual h*
    by (*metis assms map-dual-anti map-dual-func1*)
  thus *?thesis*
    by (*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 ⇒ 'a
  and g :: 'b::complete-lattice ⇒ 'b
  assumes ∃f. f ⊣ h
  and mono f
  and mono g
  and h ∘ f = g ∘ h
  shows h (gfp f) = gfp g
proof−
  have a: h (gfp f) ≤ gfp g
    by (simp add: assms(2) assms(4) gfp-little-fusion)
  obtain hl where conn: ∀x y. (hl x ≤ y) ⟷ (x ≤ h y)
    using assms adj-def by blast
  have hl ∘ g ≤ hl ∘ g ∘ h ∘ hl
    by (simp add: le-fun-def, meson conn assms(3) monoE order-refl order-trans)
  also have ... = hl ∘ h ∘ f ∘ hl
    by (simp add: assms(4) comp-assoc)
  finally have hl ∘ g ≤ f ∘ hl
    by (simp add: le-fun-def, metis conn inf.coboundedI2 inf.orderE order-refl)
  hence hl (gfp g) ≤ f (hl (gfp g))
    by (metis comp-eq-dest-lhs gfp-unfold assms(3) le-fun-def)
  hence hl (gfp g) ≤ gfp f
    by (simp add: gfp-upperbound)
  hence gfp g ≤ h (gfp f)
    by (simp add: conn)
  thus ?thesis
    by (simp add: a eq-iff)
qed

lemma lfp-fusion:
  fixes f :: 'a::complete-lattice-with-dual ⇒ 'a
  and g :: 'b::complete-lattice-with-dual ⇒ 'b
  assumes ∃f. h ⊣ f
  and mono f
  and mono g
  and h ∘ f = g ∘ h
  shows h (lfp f) = lfp g
proof−
  have a: ∃f. map-dual f ⊣ map-dual h
    using adj-dual assms(1) by auto
  have b: mono (map-dual f)
    by (simp add: assms(2) iso-map-dual)
  have c: mono (map-dual g)
    by (simp add: assms(3) iso-map-dual)
  have map-dual h ∘ map-dual f = map-dual g ∘ map-dual h
    by (metis assms(4) map-dual-func1)
  thus ?thesis
```

by (*metis a adj-dual b c gfp-fusion ladj-adj ladj-radj-dual lfp-dual-var lfp-to-gfp-var radj-adj*)
qed

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

lemma *lfp-fusion-sup-pres*:
fixes $f :: 'a::complete\text{-}lattice\text{-}with\text{-}dual \Rightarrow 'a$
and $g :: 'b::complete\text{-}lattice\text{-}with\text{-}dual \Rightarrow 'b$
assumes *Sup-pres h*
and *mono f*
and *mono g*
and $h \circ f = g \circ h$
shows $h\ (lfp\ f) = lfp\ g$
by (*simp add*: *Sup-pres-ladj assms lfp-fusion*)

The following facts are usueful 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 $k :: 'a::order \Rightarrow 'b::complete\text{-}lattice$
shows $\exists F.\forall x.\ (F::'b \Rightarrow 'a \Rightarrow 'b) \dashv (\lambda k.\ k\ y)$
by (*force intro*!: *fun-eq-iff Inf-pres-radj*)

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

lemma *gfp-fusion-var*:
fixes $F :: ('a::order \Rightarrow 'b::complete\text{-}lattice) \Rightarrow 'a \Rightarrow 'b$
and $g :: 'b \Rightarrow 'b$
assumes *mono F*
and *mono g*
and $\forall h.\ F\ h\ x = g\ (h\ x)$
shows $gfp\ F\ x = gfp\ g$
by (*metis* (*no-types*, *opaque-lifting*) *assms eq-iff gfp-fixpoint gfp-upperbound k-adju-var 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* ⇒ '*a*) ⇒ *bool* **where**
clop f = (*id* ≤ *f* ∧ *mono f* ∧ *f* ∘ *f* ≤ *f*)

lemma *clop-extensive*: *clop f* ⟹ *id* ≤ *f*
by (*simp add*: *clop-def*)

lemma *clop-extensive-var*: *clop f* ⟹ *x* ≤ *f x*
by (*simp add*: *clop-def le-fun-def*)

lemma *clop-iso*: *clop f* ⟹ *mono f*
by (*simp add*: *clop-def*)

lemma *clop-iso-var*: *clop f* ⟹ *x* ≤ *y* ⟹ *f x* ≤ *f y*
by (*simp add*: *clop-def mono-def*)

lemma *clop-idem*: *clop f* ⟹ *f* ∘ *f* = *f*
by (*simp add*: *antisym clop-def le-fun-def*)

lemma *clop-Fix-range*: *clop f* ⟹ (*Fix f* = *range f*)
by (*simp add*: *clop-idem retraction-prop-fix*)

lemma *clop-idem-var*: *clop f* ⟹ *f* (*f x*) = *f x*
by (*simp add*: *clop-idem retraction-prop*)

lemma *clop-Inf-closed-var*:
fixes *f* :: '*a*::*complete-lattice* ⇒ '*a*
shows *clop f* ⟹ *f* ∘ *Inf* ∘ (‘) *f* = *Inf* ∘ (‘) *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* ⇒ '*a*
shows *clop f* ⟹ *f* ⊤ = ⊤
by (*simp add*: *clop-extensive-var top.extremum-uniqueI*)

lemma *clop* (*f*::'*a*::*complete-lattice* ⇒ '*a*) ⟹ *f* (⨆*x* ∈ *X*. *f x*) = (⨆*x* ∈ *X*. *f x*)

oops

lemma *clop* $(f::'a::complete\text{-}lattice \Rightarrow 'a) \Longrightarrow f\ (f\ x \sqcup f\ y) = f\ x \sqcup f\ y$
oops

lemma *clop* $(f::'a::complete\text{-}lattice \Rightarrow 'a) \Longrightarrow f\ \bot = \bot$
oops

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

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

lemma *clop* $(f::'a\ set \Rightarrow 'a\ set) \Longrightarrow f\ \bot = \bot$
oops

lemma *clop-closure*: *clop f* $\Longrightarrow (x \in range\ f) = (f\ x = x)$
by (*simp add*: *clop-idem retraction-prop*)

lemma *clop-closure-set*: *clop f* $\Longrightarrow$ *range f* = *Fix f*
by (*simp add*: *clop-Fix-range*)

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

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

lemma *clop-alt*: $(clop\ f) = (\forall x\ y.\ x \le f\ y \longleftrightarrow f\ x \le f\ y)$
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 $f \dashv g \Longrightarrow clop\ (g \circ f)$
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*: *surj f* $\Longrightarrow$ *surj-on f Y*
by (*simp add*: *surjD surj-on-def*)

lemma *fun-surj-inj*: $\exists\, g\ h.\ f = g \circ h \wedge$ *surj-on h* (*range f*) $\wedge$ *inj-on g* (*range f*)
proof−
  obtain *h* **where** *a*: $\forall\, x.\ f\ x = h\ x$
    by *blast*
  then have *surj-on h* (*range f*)
    by (*metis* (*mono-tags*, *lifting*) *imageE surj-on-def*)
  then show *?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*: *clop* ($\Downarrow$::$'a$::*preorder set* $\Rightarrow$ $'a$ *set*)
  by (*simp add*: *clop-def downset-set-ext downset-set-iso*)

lemma *clop-preorder-aux*: *clop f* $\Longrightarrow$ ($x \in f\ \{y\} \longleftrightarrow f\ \{x\} \subseteq f\ \{y\}$)
  by (*simp add*: *clop-alt*)

lemma *clop-preorder*: *clop f* $\Longrightarrow$ *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* ($\Uparrow$::$'a$::*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* $\Rightarrow$ $'a$
  shows *clop f* $\Longrightarrow$ *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* (*verit*) *Inf-greatest Inf-lower antisym mem-Collect-eq subsetCE*)

lemma *clop-top-Fix*:
  fixes *f* :: $'a$::*complete-lattice* $\Rightarrow$ $'a$
  shows *clop f* $\Longrightarrow$ $\top \in$ *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* $\Longrightarrow$ *clop* ($\lambda y.\ \bigsqcap\{x \in X.\ y \le x\}$)
 by (*smt* (*verit*) *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* ⟹ ∀ *x* ∈ *X*. *x* ∈ *range f* ⟹ ⨅ *X* ∈ *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* (⇓::*'a::order set* ⇒ *'a set*)
by *fastforce*

setup-lifting *type-definition-downsets*

typedef (**overloaded**) *'a upsets* = *range* (⇑::*'a::order set* ⇒ *'a set*)
by *fastforce*

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** λ*X Y*. *Rep-downsets X* ⊆ *Rep-downsets Y*.

lift-definition *less-downsets* :: *'a downsets* ⇒ *'a downsets* ⇒ *bool* **is** λ*X 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* (*verit*) *Abs-downsets-inverse INF-lower Inf-closed-clop-var Rep-downsets image-iff o-def preorder-clop*)
apply (*transfer*, *smt* (*verit*) *comp-def Abs-downsets-inverse Inf-closed-clop-var Rep-downsets image-iff le-INF-iff preorder-clop*)
done

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** λ*X Y*. *Rep-upsets X* ⊆ *Rep-upsets Y*.

lift-definition *less-upsets* :: *'a upsets* ⇒ *'a upsets* ⇒ *bool* **is** $\lambda X\ Y.$ *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* (*verit*) *Abs-upsets-inverse Inf-closed-clop-var Inf-lower Rep-upsets comp-apply image-iff preorder-clop-dual*)
  apply (*transfer*, *smt* (*verit*) *comp-def Abs-upsets-inverse Inf-closed-clop-var Inter-iff Rep-upsets image-iff preorder-clop-dual subsetCE subsetI*)
  done

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 *coclop* :: (*'a*::*order* ⇒ *'a*::*order*) ⇒ *bool* **where**
  coclop f = ($f \le id \wedge$ *mono f* $\wedge f \le f \circ f$)

lemma *coclop-dual*: (*coclop*::(*'a*::*order-with-dual* ⇒ *'a*) ⇒ *bool*) = *clop* ∘ ∂_F
  unfolding *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 *coclop-dual-var*:
  fixes *f* :: *'a*::*order-with-dual* ⇒ *'a*
  shows *coclop f* = *clop* (∂_F *f*)
  by (*simp add*: *coclop-dual*)

lemma *clop-dual*: (*clop*::(*'a*::*order-with-dual* ⇒ *'a*) ⇒ *bool*) = *coclop* ∘ ∂_F
  by (*simp add*: *coclop-dual comp-assoc map-dual-invol*)

lemma *clop-dual-var*:
  fixes *f* :: *'a*::*order-with-dual* ⇒ *'a*
  shows *clop f* = *coclop* (∂_F *f*)
  by (*simp add*: *clop-dual*)

lemma *coclop-coextensive*: *coclop f* ⟹ $f \le id$

by (*simp add*: *coclop-def*)

lemma *coclop-coextensive-var*: *coclop f* $\Longrightarrow$ *f x* $\leq$ *x*
using *coclop-def le-funD* **by** *fastforce*

lemma *coclop-iso*: *coclop f* $\Longrightarrow$ *mono f*
by (*simp add*: *coclop-def*)

lemma *coclop-iso-var*: *coclop f* $\Longrightarrow$ ($x \leq y \longrightarrow f\,x \leq f\,y$)
by (*simp add*: *coclop-iso monoD*)

lemma *coclop-idem*: *coclop f* $\Longrightarrow$ $f \circ f = f$
by (*simp add*: *antisym coclop-def le-fun-def*)

lemma *coclop-closure*: *coclop f* $\Longrightarrow$ ($x \in$ *range f*) = ($f\,x = x$)
by (*simp add*: *coclop-idem retraction-prop*)

lemma *coclop-Fix-range*: *coclop f* $\Longrightarrow$ (*Fix f* = *range f*)
by (*simp add*: *coclop-idem retraction-prop-fix*)

lemma *coclop-idem-var*: *coclop f* $\Longrightarrow$ $f\ (f\,x) = f\,x$
by (*simp add*: *coclop-idem retraction-prop*)

lemma *coclop-Sup-closed-var*:
fixes *f* :: *'a*::*complete-lattice-with-dual* $\Rightarrow$ *'a*
shows *coclop f* $\Longrightarrow$ $f \circ$ *Sup* $\circ$ (') *f* = *Sup* $\circ$ (') *f*
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* $\Longrightarrow$ $\forall\, x \in X.\ x \in$ *range f* $\Longrightarrow$ $\bigsqcup X \in$ *range f*
by (*smt* (*verit*) *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* $\Rightarrow$ *'a*
shows *coclop f* $\Longrightarrow$ $f\ \bot = \bot$
by (*simp add*: *bot.extremum-uniqueI coclop-coextensive-var*)

lemma *coclop* (*f*::*'a*::*complete-lattice* $\Rightarrow$ *'a*) $\Longrightarrow$ $f\ (\bigsqcap x \in X.\ f\ x) = (\bigsqcap x \in X.\ f\ x)$
oops

lemma *coclop* (*f*::*'a*::*complete-lattice* $\Rightarrow$ *'a*) $\Longrightarrow$ $f\ (f\,x \sqcap f\,y) = f\,x \sqcap f\,y$
oops

lemma *coclop* (*f*::*'a*::*complete-lattice* $\Rightarrow$ *'a*) $\Longrightarrow$ $f\ \top = \top$
oops

lemma *coclop* $(f::'a\ set \Rightarrow 'a\ set) \Longrightarrow f\ (\prod x \in X.\ f\ x) = (\prod x \in X.\ f\ x)$
oops

lemma *coclop* $(f::'a\ set \Rightarrow 'a\ set) \Longrightarrow f\ (f\ x \sqcap f\ y) = f\ x \sqcap f\ y$
oops

lemma *coclop* $(f::'a\ set \Rightarrow 'a\ set) \Longrightarrow f\ \top = \top$
oops

lemma *coclop-coclosure*: *coclop f* $\Longrightarrow f\ x = x \longleftrightarrow x \in$ *range f*
by (*simp add*: *coclop-idem retraction-prop*)

lemma *coclop-coclosure-set*: *coclop f* $\Longrightarrow$ *range f* = *Fix f*
by (*simp add*: *coclop-idem retraction-prop-fix*)

lemma *coclop-coclosure-prop*: (*coclop*::('*a*::*complete-lattice* ⇒ '*a*) ⇒ *bool*) (*Sup* ∘ ↓)
by (*simp add*: *coclop-def mono-def*)

lemma *coclop-coclosure-prop-var*: *coclop* $(\lambda x::'a::complete\text{-}lattice.\ \bigsqcup\{y.\ y \leq x\})$
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*) = $(\forall x\ y.\ f\ x \leq y \longleftrightarrow f\ x \leq f\ y)$
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* ⇒ '*b*::*order*
shows $f \dashv g \Longrightarrow$ *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* ⇒ '*a*
shows *coclop f* ⟹ *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* (*verit*) *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* ⟹ *coclop* $(\lambda y.\ \bigsqcup\{x \in X.\ x \leq y\})$
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* (*verit*) *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 complete 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*: $x \leq y \Longrightarrow$ *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 *clop-cl-op*: *clop cl-op*
  unfolding *clop-def le-fun-def comp-def*
  by (*simp add*: *cl-op-class.clop-ext cl-op-class.clop-iso cl-op-class.clop-wtrans order-class.mono-def*)

lemma *clop-idem* [*simp*]: *cl-op* ∘ *cl-op* = *cl-op*
  using *clop-ext clop-wtrans order.antisym* **by** *auto*

lemma *clop-idem-var* [*simp*]: *cl-op* (*cl-op* x) = *cl-op* x
  by (*simp add*: *order.antisym clop-ext clop-wtrans*)

lemma *clop-range-Fix*: *range cl-op* = *Fix cl-op*
  by (*simp add*: *retraction-prop-fix*)

lemma *Inf-closed-cl-op-var*:
  fixes X :: $'a$ *set*
  shows $\forall x \in X.\ x \in$ *range cl-op* $\Longrightarrow \bigsqcap X \in$ *range cl-op*
proof−
  assume *h*: $\forall x \in X.\ x \in$ *range cl-op*
  hence $\forall x \in X.$ *cl-op* $x = x$
    by (*simp add*: *retraction-prop*)
  hence *cl-op* $(\bigsqcap X) = \bigsqcap X$
    by (*metis Inf-lower clop-ext clop-iso dual-order.antisym le-Inf-iff*)
  thus *?thesis*
    by (*metis rangeI*)
qed

lemma *inf-closed-cl-op-var*: $x \in$ *range cl-op* $\Longrightarrow y \in$ *range cl-op* $\Longrightarrow x \sqcap y \in$ *range cl-op*
  by (*smt* (*verit*) *Inf-closed-cl-op-var UnI1 insert-iff insert-is-Un inf-Inf*)

end

typedef (**overloaded**) $'a$::*clattice-with-clop cl-op-im* = *range* (*cl-op*::$'a \Rightarrow 'a$)

by *force*

setup-lifting *type-definition-cl-op-im*

lemma *cl-op-prop* [*iff*]: (*cl-op* (*x* ⊔ *y*) = *cl-op* *y*) = (*cl-op* (*x*::′*a*::*clattice-with-clop*) ≤ *cl-op* *y*)
by (*smt* (*verit*) *cl-op-class.clop-iso clop-ext clop-wtrans inf-sup-ord*(*4*) *le-iff-sup sup.absorb-iff1 sup-left-commute*)

lemma *cl-op-prop-var* [*iff*]: (*cl-op* (*x* ⊔ *cl-op* *y*) = *cl-op* *y*) = (*cl-op* (*x*::′*a*::*clattice-with-clop*) ≤ *cl-op* *y*)
by (*metis cl-op-prop clattice-with-clop-class.clop-idem-var*)

instantiation *cl-op-im* :: (*clattice-with-clop*) *complete-lattice*
begin

lift-definition *Inf-cl-op-im* :: ′*a cl-op-im set* ⇒ ′*a cl-op-im* **is** *Inf*
by (*simp add*: *Inf-closed-cl-op-var*)

lift-definition *Sup-cl-op-im* :: ′*a cl-op-im set* ⇒ ′*a cl-op-im* **is** $\lambda X.$ *cl-op* ($\bigsqcup X$)
by *simp*

lift-definition *inf-cl-op-im* :: ′*a cl-op-im* ⇒ ′*a cl-op-im* ⇒ ′*a cl-op-im* **is** *inf*
by (*simp add*: *inf-closed-cl-op-var*)

lift-definition *sup-cl-op-im* :: ′*a cl-op-im* ⇒ ′*a cl-op-im* ⇒ ′*a cl-op-im* **is** $\lambda x\ y.$ *cl-op* (*x* ⊔ *y*)
by *simp*

lift-definition *less-eq-cl-op-im* :: ′*a cl-op-im* ⇒ ′*a cl-op-im* ⇒ *bool* **is** (≤)**.**

lift-definition *less-cl-op-im* :: ′*a cl-op-im* ⇒ ′*a cl-op-im* ⇒ *bool* **is** (<)**.**

lift-definition *bot-cl-op-im* :: ′*a cl-op-im* **is** *cl-op* ⊥
by *simp*

lift-definition *top-cl-op-im* :: ′*a cl-op-im* **is** ⊤
by (*simp add*: *clop-cl-op clop-closure clop-top*)

instance
apply (*intro-classes*; *transfer*)
apply (*simp-all add*: *less-le-not-le Inf-lower Inf-greatest*)
apply (*meson clop-cl-op clop-extensive-var dual-order.trans inf-sup-ord*(*3*))
apply (*meson clop-cl-op clop-extensive-var dual-order.trans sup-ge2*)
apply (*metis cl-op-class.clop-iso clop-cl-op clop-closure le-sup-iff*)
apply (*meson Sup-upper clop-cl-op clop-extensive-var dual-order.trans*)
by (*metis Sup-le-iff cl-op-class.clop-iso clop-cl-op clop-closure*)

end

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* :: *'a*::*complete-lattice-with-dual* ⇒ *'b*::*complete-lattice-with-dual*
  assumes *surj f*
  and *Sup-pres f*
  shows *f* ∘ (*radj f*) = *id*
proof−
  have *f* ⊣ (*radj f*)
    using *Sup-pres-ladj assms*(*2*) *radj-adj* **by** *auto*
  thus *?thesis*
    using *adj-sur-inv assms*(*1*) **by** *blast*
qed

lemma *surj-Sup-pres-inj*:
  fixes *f* :: *'a*::*complete-lattice-with-dual* ⇒ *'b*::*complete-lattice-with-dual*
  assumes *surj f*
  and *Sup-pres f*
  shows *inj* (*radj f*)
  by (*metis assms comp-eq-dest-lhs id-apply injI surj-Sup-pres-id*)

lemma *surj-Sup-pres-inj-on*:
  fixes *f* :: *'a*::*complete-lattice-with-dual* ⇒ *'b*::*complete-lattice-with-dual*
  assumes *surj f*
  and *Sup-pres f*
  shows *inj-on f* (*range* (*radj f* ∘ *f*))
  by (*smt* (*verit*) *Sup-pres-ladj-aux adj-idem2 assms*(*2*) *comp-apply inj-on-def retraction-prop*)

lemma *surj-Sup-pres-bij-on*:
  fixes *f* :: *'a*::*complete-lattice-with-dual* ⇒ *'b*::*complete-lattice-with-dual*
  assumes *surj f*
  and *Sup-pres f*
  shows *bij-betw f* (*range* (*radj f* ∘ *f*)) *UNIV*
  unfolding *bij-betw-def*
  apply *safe*

apply (*simp add*: *assms*(*1*) *assms*(*2*) *surj-Sup-pres-inj-on cong del*: *image-cong-simp*)
apply *auto*
apply (*metis* (*mono-tags*) *UNIV-I assms*(*1*) *assms*(*2*) *comp-apply id-apply image-image surj-Sup-pres-id 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 *surj-Sup-pres-iso*:
fixes f :: $'a$::*complete-lattice-with-dual* ⇒ $'b$::*complete-lattice-with-dual*
assumes *surj f*
and *Sup-pres f*
shows f ((*radj f* ∘ *f*) ($\bigsqcup X$)) = ($\bigsqcup x \in X.\ f\ x$)
by (*metis assms*(*1*) *assms*(*2*) *comp-def pointfree-idE surj-Sup-pres-id*)

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* ≡ *lb-set* ∘ *ub-set*

lemma *up-set-prop*: (X::$'a$::*preorder set*) ≠ {} ⟹ *ub-set* $X = \bigcap\{\uparrow x \mid x.\ x \in X\}$
unfolding *ub-set-def upset-def upset-set-def* **by** (*safe*, *simp-all*, *blast*)

lemma *lb-set-prop*: (X::$'a$::*preorder set*) ≠ {} ⟹ *lb-set* $X = \bigcap\{\downarrow x \mid x.\ x \in X\}$
unfolding *lb-set-def downset-def downset-set-def* **by** (*safe*, *simp-all*, *blast*)

lemma *dm-downset-var*: *dm* {x} = ↓(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* ∘ η = (↓::$'a$::*preorder* ⇒ $'a$ *set*)
using *dm-downset-var fun.map-cong* **by** *fastforce*

lemma *dm-inj*: *inj* ((*dm*::$'a$::*order set* ⇒ $'a$ *set*) ∘ η)
by (*simp add*: *dm-downset downset-inj*)

lemma *clop* (*lb-set* ∘ *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*
begin

unbundle *lattice-syntax*

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 $Fix :: ('a \Rightarrow 'a) \Rightarrow 'a\ set$ **where**
$Fix\ f = \{x.\ f\ x = x\}$

context *ord*
begin

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

definition $max\text{-}set :: 'a\ set \Rightarrow 'a\ set$ **where**
$max\text{-}set\ X = \{x \in X.\ \forall y \in X.\ x \leq y \longrightarrow x = y\}$

definition $directed :: 'a\ set \Rightarrow bool$ **where**
$directed\ X = (\forall Y.\ finite\ Y \wedge Y \subseteq X \longrightarrow (\exists x \in X.\ \forall y \in Y.\ y \leq x))$

definition $filtered :: 'a\ set \Rightarrow bool$ **where**
$filtered\ X = (\forall Y.\ finite\ Y \wedge Y \subseteq X \longrightarrow (\exists x \in X.\ \forall y \in Y.\ x \leq y))$

definition $downset\text{-}set :: 'a\ set \Rightarrow 'a\ set$ (‹⇓›) **where**
$\Downarrow X = \{y.\ \exists x \in X.\ y \leq x\}$

definition $upset\text{-}set :: 'a\ set \Rightarrow 'a\ set$ (‹⇑›) **where**
$\Uparrow X = \{y.\ \exists x \in X.\ x \leq y\}$

definition $downset :: 'a \Rightarrow 'a\ set$ (‹↓›) **where**
$\downarrow\ = \Downarrow \circ (\lambda x.\ \{x\})$

definition $upset :: 'a \Rightarrow 'a\ set$ (‹↑›) **where**
$\uparrow\ = \Uparrow \circ (\lambda x.\ \{x\})$

definition $downsets :: 'a\ set\ set$ **where**
$downsets = Fix\ \Downarrow$

definition $upsets :: 'a\ set\ set$ **where**
$upsets = Fix\ \Uparrow$

abbreviation *downset-setp* $X \equiv X \in$ *downsets*

abbreviation *upset-setp* $X \equiv X \in$ *upsets*

definition *ideals* :: *'a set set* **where**
ideals $= \{X.\ X \neq \{\} \wedge$ *downset-setp* $X \wedge$ *directed* $X\}$

definition *filters* :: *'a set set* **where**
filters $= \{X.\ X \neq \{\} \wedge$ *upset-setp* $X \wedge$ *filtered* $X\}$

abbreviation *idealp* $X \equiv X \in$ *ideals*

abbreviation *filterp* $X \equiv X \in$ *filters*

end

abbreviation *Sup-pres* :: $('a::Sup \Rightarrow 'b::Sup) \Rightarrow bool$ **where**
Sup-pres $f \equiv f \circ Sup = Sup \circ (`)\ f$

abbreviation *Inf-pres* :: $('a::Inf \Rightarrow 'b::Inf) \Rightarrow bool$ **where**
Inf-pres $f \equiv f \circ Inf = Inf \circ (`)\ f$

abbreviation *sup-pres* :: $('a::sup \Rightarrow 'b::sup) \Rightarrow bool$ **where**
sup-pres $f \equiv (\forall x\ y.\ f\ (x \sqcup y) = f\ x \sqcup f\ y)$

abbreviation *inf-pres* :: $('a::inf \Rightarrow 'b::inf) \Rightarrow bool$ **where**
inf-pres $f \equiv (\forall x\ y.\ f\ (x \sqcap y) = f\ x \sqcap f\ y)$

abbreviation *bot-pres* :: $('a::bot \Rightarrow 'b::bot) \Rightarrow bool$ **where**
bot-pres $f \equiv f\ \bot = \bot$

abbreviation *top-pres* :: $('a::top \Rightarrow 'b::top) \Rightarrow bool$ **where**
top-pres $f \equiv f\ \top = \top$

abbreviation *Sup-dual* :: $('a::Sup \Rightarrow 'b::Inf) \Rightarrow bool$ **where**
Sup-dual $f \equiv f \circ Sup = Inf \circ (`)\ f$

abbreviation *Inf-dual* :: $('a::Inf \Rightarrow 'b::Sup) \Rightarrow bool$ **where**
Inf-dual $f \equiv f \circ Inf = Sup \circ (`)\ f$

abbreviation *sup-dual* :: $('a::sup \Rightarrow 'b::inf) \Rightarrow bool$ **where**
sup-dual $f \equiv (\forall x\ y.\ f\ (x \sqcup y) = f\ x \sqcap f\ y)$

abbreviation *inf-dual* :: $('a::inf \Rightarrow 'b::sup) \Rightarrow bool$ **where**
inf-dual $f \equiv (\forall x\ y.\ f\ (x \sqcap y) = f\ x \sqcup f\ y)$

abbreviation *bot-dual* :: $('a::bot \Rightarrow 'b::top) \Rightarrow bool$ **where**
bot-dual $f \equiv f\ \bot = \top$

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*
by (*unfold-locales*, *simp-all*)

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

sublocale *boolean-algebra* ⊆ *dual-boolean-algebra*: *boolean-algebra* $\lambda x\ y.\ x \sqcup -y$ *uminus sup* (≥) (>) *inf* ⊤ ⊥
by (*unfold-locales*, *simp-all add*: *inf-sup-distrib1*)

sublocale *complete-lattice* ⊆ *dual-complete-lattice*: *complete-lattice Sup Inf sup* (≥) (>) *inf* ⊤ ⊥
rewrites *dual-gfp*: *gfp* = *dual-complete-lattice.lfp*
proof−
show *class.complete-lattice Sup Inf sup* (≥) (>) *inf* ⊤ ⊥
by (*unfold-locales*, *simp-all add*: *Sup-upper Sup-least Inf-lower Inf-greatest*)

then interpret *dual-complete-lattice*: *complete-lattice Sup Inf sup* $(\geq)$ $(>)$ *inf* $\top$ $\bot$.
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* $X \Longrightarrow X \neq \{\}$
unfolding *directed-def* **by** *fastforce*

lemma *directed-ub*: *directed* $X \Longrightarrow (\forall x \in X.\ \forall y \in X.\ \exists z \in X.\ x \leq z \wedge y \leq z)$
by (*meson empty-subsetI directed-def finite.emptyI finite-insert insert-subset order-refl*)

lemma *downset-set-prop*: $\Downarrow = Union \circ (`)\ \downarrow$
  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-prop*: $\downarrow x = \{y.\ y \le x\}$
  unfolding *downset-def downset-set-def fun-eq-iff comp-def* **by** *fastforce*

end

context *preorder*
begin

lemma *directed-prop*: $X \neq \{\} \Longrightarrow (\forall x \in X.\ \forall y \in X.\ \exists z \in X.\ x \le z \wedge y \le z)$ $\Longrightarrow$ *directed X*
proof−
  assume *h1*: $X \neq \{\}$
  and *h2*: $\forall x \in X.\ \forall y \in X.\ \exists z \in X.\ x \le z \wedge y \le z$
  {fix *Y*
  have *finite Y* $\Longrightarrow Y \subseteq X \Longrightarrow (\exists x \in X.\ \forall y \in Y.\ y \le x)$
  proof (*induct rule*: *finite-induct*)
    case *empty*
    then show *?case*
      using *h1* **by** *blast*
  next
    case (*insert x F*)
    then show *?case*
      by (*metis h2 insert-iff insert-subset order-trans*)
  qed}
  thus *?thesis*
    by (*simp add*: *directed-def*)
qed

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

lemma *downset-set-ext*: *id* $\le \Downarrow$
  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 comp-def* **using** *order-trans* **by** *auto*

lemma *downset-faithful*: $\downarrow x \subseteq \downarrow y \Longrightarrow x \le y$
  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*

lemma *downset-directed-downset-var* [*simp*]: *directed* $(\Downarrow X)$ = *directed* X
proof
assume *h1*: *directed* X
{fix Y
assume *h2*: *finite* Y **and** *h3*: $Y \subseteq \Downarrow X$
hence $\forall y. \exists x. y \in Y \longrightarrow x \in X \wedge y \leq x$
by (*force simp*: *downset-set-def*)
hence $\exists f. \forall y. y \in Y \longrightarrow f\ y \in X \wedge y \leq f\ y$
by (*rule choice*)
hence $\exists f.$ *finite* $(f\ `\ Y) \wedge f\ `\ Y \subseteq X \wedge (\forall y \in Y. y \leq f\ y)$
by (*metis finite-imageI h2 image-subsetI*)
hence $\exists Z.$ *finite* $Z \wedge Z \subseteq X \wedge (\forall y \in Y. \exists z \in Z. y \leq z)$
by *fastforce*
hence $\exists Z.$ *finite* $Z \wedge Z \subseteq X \wedge (\forall y \in Y. \exists z \in Z. y \leq z) \wedge (\exists x \in X. \forall z \in Z. z \leq x)$
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* (*verit*) *Ball-Collect order-refl order-trans subsetCE*)
qed

lemma *downset-directed-downset* [*simp*]: *directed* $\circ \Downarrow$ = *directed*
unfolding *fun-eq-iff comp-def* **by** *simp*

lemma *directed-downset-ideals*: *directed* $(\Downarrow X) = (\Downarrow X \in$ *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* $(\downarrow::'a::order \Rightarrow 'a\ set)$
by (*simp add*: *downset-iso-iff mono-def*)

context *order*
begin

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

end

context *lattice*
begin

lemma *lat-ideals*: $X \in ideals = (X \neq \{\} \wedge X \in downsets \wedge (\forall x \in X.\ \forall\ y \in X.\ x \sqcup y \in X))$
  unfolding *ideals-def directed-alt downsets-def Fix-def downset-set-def*
  using *local.sup.bounded-iff* **by** *blast*

end

context *bounded-lattice*
begin

lemma *bot-ideal*: $X \in ideals \Longrightarrow \bot \in X$
  unfolding *ideals-def downsets-def Fix-def downset-set-def* **by** *fastforce*

end

context *complete-lattice*
begin

lemma *Sup-downset-id* [*simp*]: $Sup \circ \downarrow = id$
  using *Sup-atMost atMost-def downset-prop* **by** *fastforce*

lemma *downset-Sup-id*: $id \leq \downarrow \circ Sup$
  by (*simp add*: *Sup-upper downset-prop le-funI subsetI*)

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

lemma *Inf-pres-downset-var*: $(\bigcap x \in X.\ \downarrow x) = \downarrow(\bigsqcap X)$
  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* $f \Longrightarrow$ *lfp* $f \in$ *Fix* f
  using *Fix-def lfp-unfold* **by** *fastforce*

lemma *gfp-in-Fix*:
  fixes f :: $'a$::*complete-lattice* $\Rightarrow$ $'a$
  shows *mono* $f \Longrightarrow$ *gfp* $f \in$ *Fix* f
  using *Fix-def gfp-unfold* **by** *fastforce*

lemma *nonempty-Fix*:

fixes *f* :: *'a::complete-lattice* ⇒ *'a*
shows *mono f* ⟹ *Fix f* ≠ {}
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* ⟹ *X* ≠ {}
by (*simp add*: *dual-filtered dual-ord.directed-nonempty*)

lemma *filtered-lb*: *filtered X* ⟹ (∀ *x* ∈ *X*. ∀ *y* ∈ *X*. ∃ *z* ∈ *X*. *z* ≤ *x* ∧ *z* ≤ *y*)
by (*simp add*: *dual-filtered dual-ord.directed-ub*)

lemma *upset-set-prop*: ⇑ = *Union* ∘ (') ↑
by (*simp add*: *dual-ord.downset-set-prop dual-upset dual-upset-set*)

lemma *upset-set-prop-var*: ⇑*X* = (⋃ *x* ∈ *X*. ↑*x*)
by (*simp add*: *dual-ord.downset-set-prop-var dual-upset dual-upset-set*)

lemma *upset-prop*: ↑*x* = {*y*. *x* ≤ *y*}
by (*simp add*: *dual-ord.downset-prop dual-upset*)

end

context *preorder*
begin

lemma *filtered-prop*: *X* ≠ {} ⟹ (∀ *x* ∈ *X*. ∀ *y* ∈ *X*. ∃ *z* ∈ *X*. *z* ≤ *x* ∧ *z* ≤ *y*) ⟹ *filtered X*
by (*simp add*: *dual-filtered dual-preorder.directed-prop*)

lemma *filtered-alt*: *filtered X* = (*X* ≠ {} ∧ (∀ *x* ∈ *X*. ∀ *y* ∈ *X*. ∃ *z* ∈ *X*. *z* ≤ *x* ∧ *z* ≤ *y*))
by (*simp add*: *dual-filtered dual-preorder.directed-alt*)

lemma *upset-set-ext*: *id* ≤ ⇑
by (*simp add*: *dual-preorder.downset-set-ext dual-upset-set*)

lemma *upset-set-anti*: *mono* ⇑
by (*simp add*: *dual-preorder.downset-set-iso dual-upset-set*)

lemma *up-set-idem* [*simp*]: ⇑ ∘ ⇑ = ⇑
by (*simp add*: *dual-upset-set*)

lemma *upset-faithful*: ↑*x* ⊆ ↑*y* ⟹ *y* ≤ *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* $(\Uparrow X) = (\Uparrow X \in$ *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* $\uparrow$
by (*simp add*: *dual-order.downset-inj dual-upset*)

end

context *lattice*
begin

lemma *lat-filters*: $X \in$ *filters* $= (X \neq \{\} \wedge X \in$ *upsets* $\wedge (\forall x \in X. \forall y \in X. x \sqcap y \in X))$
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*: $X \in$ *filters* $\Longrightarrow \top \in X$
by (*simp add*: *dual-bounded-lattice.bot-ideal dual-filters*)

end

context *complete-lattice*
begin

lemma *Inf-upset-id* [*simp*]: *Inf* $\circ \uparrow =$ *id*
by (*simp add*: *dual-upset*)

lemma *upset-Inf-id*: *id* $\leq \uparrow \circ$ *Inf*
by (*simp add*: *dual-complete-lattice.downset-Sup-id dual-upset*)

lemma *Sup-Inf-var*: $\bigsqcap(\bigcap x \in X. \uparrow x) = \bigsqcup X$
by (*simp add*: *dual-complete-lattice.Inf-Sup-var dual-upset*)

lemma *Sup-dual-upset-var*: $(\bigcap x \in X. \uparrow x) = \uparrow(\bigsqcup X)$
by (*simp add*: *dual-complete-lattice.Inf-pres-downset-var dual-upset*)

end

9.4 Examples that Do Not Dualise

lemma *upset-anti*: *antimono* ($\uparrow$::$'a$::*order* $\Rightarrow$ $'a$ *set*)
by (*simp add*: *antimono-def upset-anti-iff*)

context *complete-lattice*
begin

lemma *fSup-unfold*: (f::*nat* $\Rightarrow$ $'a$) $0 \sqcup (\bigsqcup n.\ f\ (Suc\ n)) = (\bigsqcup n.\ f\ n)$
apply (*intro order.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* $\Rightarrow$ $'a$) $0 \sqcap (\bigsqcap n.\ f\ (Suc\ n)) = (\bigsqcap n.\ f\ n)$
apply (*intro order.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*)

end

lemma *fun-isol*: *mono f* $\Longrightarrow$ *mono* (($\circ$) *f*)
by (*simp add*: *le-fun-def mono-def*)

lemma *fun-isor*: *mono f* $\Longrightarrow$ *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 *Sup-pres f* $\Longrightarrow$ *sup-pres f*
by (*metis* (*no-types*, *opaque-lifting*) *Sup-empty Sup-insert comp-apply image-insert sup-bot.right-neutral*)

lemma *Inf-inf-pres*:
fixes *f* :: $'a$::*complete-lattice* $\Rightarrow$ $'b$::*complete-lattice*

shows *Inf-pres f* ⟹ *inf-pres f*
by (*smt* (*verit*) *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* ∘ *Inf* ≤ *Inf* ∘ (*`*) *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* ∘ (*`*) *f* ≤ *f* ∘ *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* ∘ *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*)

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 *fSup-distr*: *Sup-pres* $(\lambda x.\ x \circ f)$
  unfolding *fun-eq-iff comp-def*
  by (*smt* (*verit*) *Inf.INF-cong SUP-apply Sup-apply*)

lemma *fSup-distr-var*: $\bigsqcup F \circ g = (\bigsqcup f \in F.\ f \circ g)$
  unfolding *fun-eq-iff comp-def*
  by (*smt* (*verit*) *Inf.INF-cong SUP-apply Sup-apply*)

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

lemma *fInf-distr-var*: $\bigsqcap F \circ g = (\bigsqcap f \in F.\ f \circ g)$
  unfolding *fun-eq-iff comp-def*
  by (*smt* (*verit*) *INF-apply Inf.INF-cong Inf-apply*)

lemma *fSup-subdistl*:
  assumes *mono* $(f::'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice)$
  shows $Sup \circ (`)\ ((\circ)\ f) \leq (\circ)\ f \circ Sup$
  using *assms* **by** (*simp add*: *SUP-least Sup-upper le-fun-def monoD image-comp*)

lemma *fSup-subdistl-var*:
  fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
  shows $mono\ f \Longrightarrow (\bigsqcup g \in G.\ f \circ g) \leq f \circ \bigsqcup G$
  by (*simp add*: *SUP-least Sup-upper le-fun-def monoD image-comp*)

lemma *fInf-subdistl*:
  fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
  shows $mono\ f \Longrightarrow (\circ)\ f \circ Inf \leq Inf \circ (`)\ ((\circ)\ f)$
  by (*simp add*: *INF-greatest Inf-lower le-fun-def monoD image-comp*)

lemma *fInf-subdistl-var*:
  fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
  shows $mono\ f \Longrightarrow f \circ \bigsqcap G \leq (\bigsqcap g \in G.\ f \circ g)$
  by (*simp add*: *INF-greatest Inf-lower le-fun-def monoD image-comp*)

lemma *Inf-pres-downset*: *Inf-pres* $(\downarrow::'a::complete\text{-}lattice \Rightarrow 'a\ set)$
  unfolding *downset-prop fun-eq-iff comp-def*
  by (*safe*, *simp-all add*: *le-Inf-iff*)

lemma *Sup-dual-upset*: *Sup-dual* $(\uparrow::'a::complete\text{-}lattice \Rightarrow 'a\ set)$
  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*
begin

unbundle *lattice-syntax*

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 additional properties. I show only the most important properties.

datatype *'a dual* = *dual* (*un-dual*: *'a*) (‹∂›)

notation *un-dual* (‹∂^-›)

lemma *dual-inj*: *inj* ∂
  using *injI* **by** *fastforce*

lemma *dual-surj*: *surj* ∂
  using *dual.exhaust-sel* **by** *blast*

lemma *dual-bij*: *bij* ∂
  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* ∂^-
  by (*simp add*: *dual.expand injI*)

lemma *dual-inv-surj*: *surj* ∂^-
  by (*metis dual.sel surj-def*)

lemma *dual-inv-bij*: *bij* ∂^-
  by (*simp add*: *bij-def dual-inv-inj dual-inv-surj*)

lemma *dual-iff*: $(\partial\ x = y) \longleftrightarrow (x = \partial^-\ y)$
by *fastforce*

Isabelle data types come with a number of generic functions.

The functor map-dual lifts functions to dual types. Isabelle's generic definition is not straightforward to understand and use. Yet conceptually it can be explained as follows.

lemma *map-dual-def-var* [*simp*]: $(\textit{map-dual}::('a \Rightarrow 'b) \Rightarrow 'a\ \textit{dual} \Rightarrow 'b\ \textit{dual})\ f = \partial \circ f \circ \partial^-$
unfolding *fun-eq-iff comp-def* **by** (*metis dual.map-sel dual-iff*)

lemma *map-dual-def-var2*: $\partial^- \circ \textit{map-dual}\ f = f \circ \partial^-$
by (*simp add*: *rewriteL-comp-comp*)

lemma *map-dual-func1*: $\textit{map-dual}\ (f \circ g) = \textit{map-dual}\ f \circ \textit{map-dual}\ g$
unfolding *fun-eq-iff comp-def* **by** (*metis dual.exhaust dual.map*)

lemma *map-dual-func2* : $\textit{map-dual}\ id = id$
by *simp*

The functor map-dual has an inverse functor as well.

definition *map-dual-inv* :: $('a\ \textit{dual} \Rightarrow 'b\ \textit{dual})$ => $('a$ => $'b)$ **where**
$\textit{map-dual-inv}\ f = \partial^- \circ f \circ \partial$

lemma *map-dual-inv-func1*: $\textit{map-dual-inv}\ id = id$
by (*simp add*: *map-dual-inv-def*)

lemma *map-dual-inv-func2*: $\textit{map-dual-inv}\ (f \circ g) = \textit{map-dual-inv}\ f \circ \textit{map-dual-inv}\ g$
unfolding *fun-eq-iff comp-def map-dual-inv-def* **by** (*metis dual-iff*)

lemma *map-dual-inv1*: $\textit{map-dual} \circ \textit{map-dual-inv} = id$
unfolding *fun-eq-iff map-dual-def-var map-dual-inv-def comp-def id-def*
by (*metis dual-iff*)

lemma *map-dual-inv2*: $\textit{map-dual-inv} \circ \textit{map-dual} = id$
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*: $(\partial\ `\ X = Y) \longleftrightarrow (X = \partial^-\ `\ Y)$
by (*metis dual-inj image-comp image-inv-f-f inv-o-cancel dual-inv2*)

lemma *subset-dual1*: $(X \subseteq Y) \longleftrightarrow (\partial\ `\ X \subseteq \partial\ `\ Y)$
by (*simp add*: *dual-inj inj-image-subset-iff*)

lemma *dual-ball*: $(\forall x \in X.\ P\ (\partial\ x)) \longleftrightarrow (\forall y \in \partial\ `\ X.\ P\ y)$
by *simp*

lemma *dual-inv-ball*: $(\forall x \in X.\ P\ (\partial^- \ x)) \longleftrightarrow (\forall y \in \partial^- \ `\ X.\ P\ y)$
by *simp*

lemma *dual-all*: $(\forall x.\ P\ (\partial\ x)) \longleftrightarrow (\forall y.\ P\ y)$
by (*metis dual.collapse*)

lemma *dual-inv-all*: $(\forall x.\ P\ (\partial^- \ x)) \longleftrightarrow (\forall y.\ P\ y)$
by (*metis dual-inv-surj surj-def*)

lemma *dual-ex*: $(\exists x.\ P\ (\partial\ x)) \longleftrightarrow (\exists y.\ P\ y)$
by (*metis UNIV-I bex-imageD dual-surj*)

lemma *dual-inv-ex*: $(\exists x.\ P\ (\partial^- \ x)) \longleftrightarrow (\exists y.\ P\ y)$
by (*metis dual.sel*)

lemma *dual-Collect*: $\{\partial\ x\ |x.\ P\ (\partial\ x)\} = \{y.\ P\ y\}$
by (*metis dual.exhaust*)

lemma *dual-inv-Collect*: $\{\partial^- \ x\ |x.\ P\ (\partial^- \ x)\} = \{y.\ P\ y\}$
by (*metis dual.collapse dual.inject*)

lemma *fun-dual1*: $(f \circ \partial = g) \longleftrightarrow (f = g \circ \partial^-)$
by *auto*

lemma *fun-dual2*: $(\partial \circ f = g) \longleftrightarrow (f = \partial^- \circ g)$
by *auto*

lemma *fun-dual3*: $(f \circ (`)\ \partial = g) \longleftrightarrow (f = g \circ (`)\ \partial^-)$
unfolding *fun-eq-iff comp-def* **by** (*metis subset-dual*)

lemma *fun-dual4*: $(f = \partial^- \circ g \circ (`)\ \partial) \longleftrightarrow (\partial \circ f \circ (`)\ \partial^- = g)$
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*: $(\leq) = \textit{rel-dual}\ (\geq)$

definition *less-dual-def*: $(<) = \textit{rel-dual}\ (>)$

instance..

end

lemma *less-eq-dual-def-var*: $(x \leq y) = (\partial^- \ y \leq \partial^- \ x)$
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*: $(x < y) = (\partial^- \ y < \partial^- \ x)$
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*: $x \leq y \Longrightarrow \partial \ y \leq \partial \ x$
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 (*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* $\Longrightarrow$ *mono* (*map-dual f*)
unfolding *map-dual-def-var mono-def* **by** (*metis comp-apply dual-anti less-eq-dual-def-var*)

instantiation *dual* :: (*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 (*standard*, *simp-all add*: *dual-inj inf-dual-def sup-dual-def less-eq-dual-def-var the-inv-f-f*)

end

instantiation *dual* :: (*complete-lattice*) *complete-lattice*
begin

definition *Inf-dual-def*: $Inf = \partial \circ Sup \circ (`) \ \partial^-$

definition *Sup-dual-def*: $Sup = \partial \circ Inf \circ (`) \ \partial^-$

definition *bot-dual-def*: $\bot = \partial \ \top$

definition *top-dual-def*: $\top = \partial \ \bot$

instance
  by (*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**
$directed\ X = (\forall Y.\ finite\ Y \wedge Y \subseteq X \longrightarrow (\exists x \in X.\ \forall y \in Y.\ y \leq x))$

definition *filtered* :: $'a\ set \Rightarrow bool$ **where**
$filtered\ X = (\forall Y.\ finite\ Y \wedge Y \subseteq X \longrightarrow (\exists x \in X.\ \forall y \in Y.\ x \leq y))$

definition *downset-set* :: $'a\ set \Rightarrow 'a\ set$ (‹⇓›) **where**
$\Downarrow X = \{y.\ \exists x \in X.\ y \leq x\}$

definition *upset-set* :: $'a\ set \Rightarrow 'a\ set$ (‹⇑›) **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*: $filtered \circ (`)\ \partial = directed$
proof−
 {**fix** X::$'a\ set$
  have $(filtered \circ (`)\ \partial)\ X = (\forall Y.\ finite\ (\partial^-\ `\ Y) \wedge \partial^-\ `\ Y \subseteq X \longrightarrow (\exists x \in X. \forall y \in (\partial^-\ `\ Y).\ \partial\ x \leq \partial\ y))$
   unfolding *filtered-def comp-def* **by** (*simp*, *metis dual-iff finite-subset-image subset-dual subset-dual1*)
  also have ... $= (\forall Y.\ finite\ Y \wedge Y \subseteq X \longrightarrow (\exists x \in X. \forall y \in Y.\ y \leq x))$
   by (*metis dual-anti-iff dual-inv-surj finite-subset-image top.extremum*)
  finally have $(filtered \circ (`)\ \partial)\ X = directed\ X$
   using *directed-def* **by** *auto*}
 thus *?thesis*
  unfolding *fun-eq-iff* **by** *simp*
qed

lemma *directed-filtered-dual*: $directed \circ (`)\ \partial = filtered$
proof−
 {**fix** X::$'a\ set$

have $(directed \circ (`)\ \partial)\ X = (\forall\, Y.\ finite\ (\partial^{-}\ `\ Y) \wedge \partial^{-}\ `\ Y \subseteq X \longrightarrow (\exists\, x \in X. \forall\, y \in (\partial^{-}\ `\ Y).\ \partial\ y \leq \partial\ x))$
unfolding *directed-def comp-def* **by** (*simp*, *metis dual-iff finite-subset-image subset-dual subset-dual1*)
also have ... $= (\forall\, Y.\ finite\ Y \wedge Y \subseteq X \longrightarrow (\exists\, x \in X. \forall\, y \in Y.\ x \leq y))$
unfolding *dual-anti-iff*[*symmetric*] **by** (*metis dual-inv-surj finite-subset-image top-greatest*)
finally have $(directed \circ (`)\ \partial)\ X = filtered\ X$
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 *downset-set-upset-set-dual*: $(`)\ \partial \circ \Downarrow\ = \Uparrow \circ\ (`)\ \partial$
proof−
{**fix** $X{::}'a\ set$
have $((`)\ \partial \circ \Downarrow)\ X = \{\partial\ y\ |y.\ \exists\, x \in X.\ y \leq x\}$
by (*simp add*: *downset-set-def setcompr-eq-image*)
also have ... $= \{\partial\ y\ |y.\ \exists\, x \in X.\ \partial\ x \leq \partial\ y\}$
by (*meson dual-anti-iff*)
also have ... $= \{y.\ \exists\, x \in \partial\ `\ X.\ x \leq y\}$
by (*metis* (*mono-tags*, *opaque-lifting*) *dual.exhaust image-iff*)
finally have $((`)\ \partial \circ \Downarrow)\ X = (\Uparrow \circ\ (`)\ \partial)\ X$
by (*simp add*: *upset-set-def*)}
thus *?thesis*
unfolding *fun-eq-iff* **by** *simp*
qed

lemma *upset-set-downset-set-dual*: $(`)\ \partial \circ \Uparrow\ = \Downarrow \circ\ (`)\ \partial$
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

[1] A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In *RAMiCS 2012*, volume 7560 of *LNCS*, pages 66–81. Springer, 2012.

[2] C. Ballarin. The Isabelle/HOL algebra library. https://isabelle.in.tum.de/dist/library/HOL/HOL-Algebra/index.html.

[3] G. Gierz, K. H. Hofmann, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer, 1980.

[4] V. B. F. Gomes and G. Struth. Residuated lattices. *Archive of Formal Proofs*, 2015.

[5] P. T. Johnstone. *Stone Spaces*. Cambridge University Press, 1982.

[6] S. Koppelberg. *Handbook of Boolean Algebras*. North-Holland, 1989.

[7] O. Kuncar and A. Popescu. From types to sets by local type definitions in higher-order logic. In *ITP 2016*, volume 9807 of *LNCS*, pages 200–218. Springer, 2016.

[8] V. Preoteasa. Algebra of monotonic boolean transformers. *Archive of Formal Proofs*, 2011.

[9] V. Preoteasa. Lattice properties. *Archive of Formal Proofs*, 2011.

[10] K. I. Rosenthal. *Quantales and their Applications*. Longman Scientific & Technical, 1990.

[11] M. Wenzel. Session HOL-Lattice. https://isabelle.in.tum.de/dist/library/HOL/HOL-Lattice/index.html.