Stone Algebras

Walter Guttmann

April 13, 2017

Abstract

A range of algebras between lattices and Boolean algebras generalise the notion of a complement. We develop a hierarchy of these pseudo-complemented algebras that includes Stone algebras. Independently of this theory we study filters based on partial orders. Both theories are combined to prove Chen and Grätzer’s construction theorem for Stone algebras. The latter involves extensive reasoning about algebraic structures in addition to reasoning in algebraic structures.

Contents

1 Synopsis and Motivation 2

2 Lattice Basics 3

3 Pseudocomplemented Algebras 11
   3.1 P-Algebras .............................................. 12
      3.1.1 Pseudocomplemented Lattices ...................... 12
      3.1.2 Pseudocomplemented Distributive Lattices ........ 21
   3.2 Stone Algebras ........................................ 22
   3.3 Heyting Algebras ..................................... 27
      3.3.1 Heyting Semilattices .............................. 27
      3.3.2 Heyting Lattices .................................. 30
      3.3.3 Heyting Algebras .................................. 32
      3.3.4 Brouwer Algebras ................................ 34
   3.4 Boolean Algebras ...................................... 35

4 Filters 37
   4.1 Orders ................................................. 37
   4.2 Lattices ................................................ 42
   4.3 Distributive Lattices .................................. 49
5 Stone Construction

5.1 Triples
5.2 The Triple of a Stone Algebra
  5.2.1 Regular Elements
  5.2.2 Dense Elements
  5.2.3 The Structure Map
5.3 Properties of Triples
5.4 The Stone Algebra of a Triple
5.5 The Stone Algebra of the Triple of a Stone Algebra
5.6 Stone Algebra Isomorphism
5.7 Triple Isomorphism
  5.7.1 Boolean Algebra Isomorphism
  5.7.2 Distributive Lattice Isomorphism
  5.7.3 Structure Map Preservation

1 Synopsis and Motivation

This document describes the following four theory files:

* Lattice Basics is a small theory with basic definitions and facts extending Isabelle/HOL’s lattice theory. It is used by the following theories.

* Pseudocomplemented Algebras contains a hierarchy of algebraic structures between lattices and Boolean algebras. Many results of Boolean algebras can be derived from weaker axioms and are useful for more general models. In this theory we develop a number of algebraic structures with such weaker axioms. The theory has four parts. We first extend lattices and distributive lattices with a pseudocomplement operation to obtain (distributive) p-algebras. An additional axiom of the pseudocomplement operation yields Stone algebras. The third part studies a relative pseudocomplement operation which results in Heyting algebras and Brouwer algebras. We finally show that Boolean algebras instantiate all of the above structures.

* Filters contains an order-/lattice-theoretic development of filters. We prove the ultrafilter lemma in a weak setting, several results about the lattice structure of filters and a few further results from the literature. Our selection is due to the requirements of the following theory.

* Construction of Stone Algebras contains the representation of Stone algebras as triples and the corresponding isomorphisms [7, 21]. It is also a case study of reasoning about algebraic structures. Every Stone algebra is isomorphic to a triple comprising a Boolean algebra, a distributive lattice with a greatest element, and a bounded lattice homomorphism from the Boolean algebra to filters of the distributive
lattice. We carry out the involved constructions and explicitly state the functions defining the isomorphisms. A function lifting is used to work around the need for dependent types. We also construct an embedding of Stone algebras to inherit theorems using a technique of universal algebra.

Algebras with pseudocomplements in general, and Stone algebras in particular, appear widely in mathematical literature; for example, see [4, 5, 6, 17]. We apply Stone algebras to verify Prim’s minimum spanning tree algorithm in Isabelle/HOL in [20].

There are at least two Isabelle/HOL theories related to filters. The theory HOL/Algebra/Ideal.thy defines ring-theoretic ideals in locales with a carrier set. In the theory HOL/Filter.thy a filter is defined as a set of sets. Filters based on orders and lattices abstract from the inner set structure; this approach is used in many texts such as [4, 5, 6, 9, 17]. Moreover, it is required for the construction theorem of Stone algebras, whence our theory implements filters this way.

Besides proving the results involved in the construction of Stone algebras, we study how to reason about algebraic structures defined as Isabelle/HOL classes without carrier sets. The Isabelle/HOL theories HOL/Algebra/*.thy use locales with a carrier set, which facilitates reasoning about algebraic structures but requires assumptions involving the carrier set in many places. Extensive libraries of algebraic structures based on classes without carrier sets have been developed and continue to be developed [1, 2, 3, 10, 11, 13, 14, 15, 16, 19, 22, 24, 25, 26]. It is unlikely that these libraries will be converted to carrier-based theories and that carrier-free and carrier-based implementations will be consistently maintained and evolved; certainly this has not happened so far and initial experiments suggest potential drawbacks for proof automation [12]. An improvement of the situation seems to require some form of automation or system support that makes the difference irrelevant.

In the present development, we use classes without carrier sets to reason about algebraic structures. To instantiate results derived in such classes, the algebras must be represented as Isabelle/HOL types. This is possible to a certain extent, but causes a problem if the definition of the underlying set depends on parameters introduced in a locale; this would require dependent types. For the construction theorem of Stone algebras we work around this restriction by a function lifting. If the parameters are known, the functions can be specialised to obtain a simple (non-dependent) type that can instantiate classes. For the construction theorem this specialisation can be done using an embedding. The extent to which this approach can be generalised to other settings remains to be investigated.
2 Lattice Basics

This theory provides notations, basic definitions and facts of lattice-related structures used throughout the subsequent development.

theory Lattice-Basics

imports Main

begin

We use the following notations for the join, meet and complement operations. Changing the precedence of the unary complement allows us to write terms like $-x$ instead of $-(-x)$.

context sup

begin

notation sup (infixl $\sqcup$ 65)

definition additive :: ('a $\Rightarrow$ 'a) $\Rightarrow$ bool

where additive $f$ $\equiv$ $\forall x y. f (x \sqcup y) = f x \sqcup f y$

end

context inf

begin

notation inf (infixl $\sqcap$ 67)

end

context uminus

begin

no-notation uminus ($-$ [81] 80)

notation uminus ($-$ [80] 80)

end

We use the following definition of monotonicity for operations defined in classes. The standard mono places a sort constraint on the target type. We also give basic properties of Galois connections and lift orders to functions.

context ord

begin

definition isotone :: ('a $\Rightarrow$ 'a) $\Rightarrow$ bool

where isotone $f$ $\equiv$ $\forall x y. x \leq y \rightarrow f x \leq f y$

end
definition galois :: ('a ⇒ 'a) ⇒ ('a ⇒ 'a) ⇒ bool
  where galois l u ≡ ∀x y. l x ≤ y ←→ x ≤ u y

definition lifted-less-eq :: ('a ⇒ 'a) ⇒ ('a ⇒ 'a) ⇒ bool ((≤≤ ·) [51, 51] 50)
  where f ≤≤ g ≡ ∀x. f x ≤ g x

end

context order
begin

lemma order-lesseq-imp:
  (∀z. x ≤ z −→ y ≤ z) ←→ y ≤ x
  using order-trans by blast

lemma galois-char:
  galois l u −→ (∀x. x ≤ u (l x)) ∧ (∀x. l (u x) ≤ x) ∧ isotone l ∧ isotone u
  apply (rule iffI)
  apply (metis (full-types) galois-def isotone-def order-refl order-trans)
  using galois-def isotone-def order-trans by blast

lemma galois-closure:
  galois l u =⇒ l x = l (u (l x)) ∧ u x = u (l (u x))
  by (simp add: galois-char isotone-def antisym)

lemma lifted-reflexive:
  f = g =⇒ f ≤≤ g
  by (simp add: lifted-less-eq-def)

lemma lifted-transitive:
  f ≤≤ g =⇒ g ≤≤ h =⇒ f ≤≤ h
  using lifted-less-eq-def order-trans by blast

lemma lifted-antisymmetric:
  f ≤≤ g =⇒ g ≤≤ f =⇒ f = g
  by (metis (full-types) antisym ext lifted-less-eq-def)

end

The following are basic facts in semilattices.

context semilattice-sup
begin

lemma sup-left-isotone:
  x ≤ y =⇒ x ⊔ z ≤ y ⊔ z
  using sup.mono by blast

lemma sup-right-isotone:
  x ≤ y =⇒ z ⊔ x ≤ z ⊔ y
Every bounded semilattice is a commutative monoid. Finite sums
defined in commutative monoids are available via the following sublocale.
\begin{isabelle}
context bounded-semilattice-sup-bot
begin
sublocale sup-monoid: comm-monoid-add where plus = sup and zero = bot
apply unfold-locales
apply (simp add: sup-assoc)
apply (simp add: sup-commute)
by simp
end
\end{isabelle}

The following class requires only the existence of upper bounds, which is
a property common to bounded semilattices and (not necessarily bounded)
lattices. We use it in our development of filters.
\begin{isabelle}
class directed-semilattice-inf = semilattice-inf +
assumes ab: \( \exists \ z . \ x \leq z \land y \leq z \)
\end{isabelle}

We extend the \textit{inf} sublocale, which dualises the order in semilattices, to
bounded semilattices.
context bounded-semilattice-inf-top
begin

subclass directed-semilattice-inf
apply unfold-locales
using top-greatest by blast

sublocale inf: bounded-semilattice-sup-bot where sup = inf and less-eq = greater-eq and less = greater and bot = top
by unfold-locales (simp-all add: less-le-not-le)
end

context lattice
begin

subclass directed-semilattice-inf
apply unfold-locales
using sup-ge1 sup-ge2 by blast

definition dual-additive :: ('a ⇒ 'a) ⇒ bool where dual-additive f ≡ ∀ x y. f (x ⊔ y) = f x ⊓ f y
end

Not every bounded lattice has complements, but two elements might still be complements of each other as captured in the following definition. In this situation we can apply, for example, the shunting property shown below. We introduce most definitions using the abbreviation command.

context bounded-lattice
begin

abbreviation complement x y ≡ x ⊔ y = top ∧ x ⊓ y = bot

lemma complement-symmetric:
complement x y → complement y x
by (simp add: inf.commute sup.commute)

definition conjugate :: ('a ⇒ 'a) ⇒ ('a ⇒ 'a) ⇒ bool where conjugate f g ≡ ∀ x y. f x ∩ y = bot ↔ x ∩ g y = bot
end

class dense-lattice = bounded-lattice +
assumes bot-meet-irreducible: x ∩ y = bot → x = bot ∨ y = bot

context distrib-lattice
begin
lemma relative-equality:
\[ x \sqcup z = y \sqcup z \implies x \sqcap z = y \sqcap z \implies x = y \]
by (metis inf.commute inf-sup-absorb inf-sup-distrib2)

Distributive lattices with a greatest element are widely used in the construction theorem for Stone algebras.

class distrib-lattice-bot = bounded-lattice-bot + distrib-lattice

class distrib-lattice-top = bounded-lattice-top + distrib-lattice

class bounded-distrib-lattice = bounded-lattice + distrib-lattice

begin

subclass distrib-lattice-bot ..

subclass distrib-lattice-top ..

lemma complement-shunting:
assumes complement z w
shows \( z \sqcap x \leq y \iff x \leq w \sqcup y \)
proof
assume 1: \( z \sqcap x \leq y \)

have \( x = (z \sqcup w) \sqcap x \)
  by (simp add: assms)

also have \( \ldots \leq y \sqcup (w \sqcap x) \)
  using 1 sup.commute sup.left-commute inf-sup-distrib2 sup-right-divisibility
by fastforce

also have \( \ldots \leq w \sqcup y \)
  by (simp add: inf.sup-right-isotone) by auto
finally show \( x \leq w \sqcup y \)
.

next

assume \( x \leq w \sqcup y \)

hence \( z \sqcap x \leq z \sqcap (w \sqcup y) \)
  using inf.sup-right-isotone by auto

also have \( \ldots = z \sqcap y \)
  by (simp add: assms inf-sup-distrib1)

also have \( \ldots \leq y \)
  by simp
finally show \( z \sqcap x \leq y \)
.
qed

end

We next consider lattices with a linear order structure. In such lattices, join and meet are selective operations, which give the maximum and the
minimum of two elements, respectively. Moreover, the lattice is automatically distributive.

class bounded-linorder = linorder + order-bot + order-top

class linear-lattice = lattice + linorder
begin

  lemma max-sup:
  max x y = x ⊔ y
  by (metis max.boundedI max.coboundedI max.cobounded2 sup-unique)

  lemma min-inf:
  min x y = x ∩ y
  by (simp add: inf.absorb1 inf.absorb2 min-def)

  lemma sup-inf-selective:
  (x ⊔ y = x ∧ x ∩ y = y) ∨ (x ⊔ y = y ∧ x ∩ y = x)
  by (meson inf.absorb1 inf.absorb2 le-cases sup.absorb1 sup.absorb2)

  lemma sup-selective:
  x ⊔ y = x ∨ x ⊔ y = y
  using sup-inf-selective by blast

  lemma inf-selective:
  x ∩ y = x ∨ x ∩ y = y
  using sup-inf-selective by blast

  subclass distrib-lattice
  apply unfold-locales
  by (metis inf-selective antisym distrib-sup-le inf.commute inf-le2)

  lemma sup-less-eq:
  x ≤ y ⊔ z ⟷ x ≤ y ∧ x ≤ z
  by (metis le-supI1 le-supI2 sup-selective)

  lemma inf-less-eq:
  x ∩ y ≤ z ⟷ x ≤ z ∨ y ≤ z
  by (metis inf.coboundedI1 inf.coboundedI2 inf-selective)

  lemma sup-inf-sup:
  x ⊔ y = (x ⊔ y) ∪ (x ∩ y)
  by (metis sup-commute sup-inf-absorb sup-left-commute)

end

  The following class derives additional properties if the linear order of the lattice has a least and a greatest element.

class linear-bounded-lattice = bounded-lattice + linorder
begin


subclass linear-lattice ..

subclass bounded-linorder ..

subclass bounded-distrib-lattice ..

lemma sup-dense:
\[ x \neq \top \implies y \neq \top \implies x \sqcup y \neq \top \]
by (metis sup-selective)

lemma inf-dense:
\[ x \neq \bot \implies y \neq \bot \implies x \sqcap y \neq \bot \]
by (metis inf-selective)

lemma sup-not-bot:
\[ x \neq \bot \implies x \sqcup y \neq \bot \]
by simp

lemma inf-not-top:
\[ x \neq \top \implies x \sqcap y \neq \top \]
by simp

subclass dense-lattice
  apply unfold-locales
  using inf-dense by blast
end

Every bounded linear order can be expanded to a bounded lattice. Join and meet are maximum and minimum, respectively.

class linorder-lattice-expansion = bounded-linorder + sup + inf +
  assumes sup-def [simp]: \( x \sqcup y = \max x y \)
  assumes inf-def [simp]: \( x \sqcap y = \min x y \)
begin

subclass linear-bounded-lattice
  apply unfold-locales
  by auto
end

Some results, such as the existence of certain filters, require that the algebras are not trivial. This is not an assumption of the order and lattice classes that come with Isabelle/HOL; for example, \( \bot = \top \) may hold in bounded lattices.

class non-trivial =
  assumes consistent: \( \exists x y . \ x \neq y \)
\textbf{class} \textit{non-trivial-order} = \textit{non-trivial} + \textit{order} \\
\textbf{class} \textit{non-trivial-order-bot} = \textit{non-trivial-order} + \textit{order-bot} \\
\textbf{class} \textit{non-trivial-bounded-order} = \textit{non-trivial-order-bot} + \textit{order-top} \\
\textbf{begin} \\
\textbf{lemma} \textit{bot-not-top}: \\
\hspace{1em} \textit{bot} \neq \textit{top} \\
\textbf{proof} - \\
\hspace{1em} \textbf{from} \textit{consistent} \textbf{obtain} \quad \textit{x y \cdot 'a where} \quad \textit{x \neq y} \\
\hspace{2em} \textbf{by} \quad \textbf{auto} \\
\hspace{1em} \textbf{thus} \quad \textbf{?thesis} \\
\hspace{2em} \textbf{by} \quad (\textit{metis bot-less top.extremum-strict}) \\
\textbf{qed} \\
\textbf{end} \\

The following results extend basic Isabelle/HOL facts. \\
\textbf{lemma} \textit{if-distrib-2}: \\
\hspace{1em} \textit{f (if c then x else y) (if c then z else w)} = (\textit{if c then f x z else f y w}) \\
\hspace{1em} \textbf{by} \quad \textbf{simp} \\
\textbf{lemma} \textit{left-invertible-inj}: \\
\hspace{1em} (\forall \textit{x}. \textit{g (f x)} = \textit{x}) \Rightarrow \textit{inj f} \\
\hspace{1em} \textbf{by} \quad (\textit{metis injI}) \\
\textbf{lemma} \textit{invertible-bij}: \\
\hspace{1em} \textbf{assumes} \quad (\forall \textit{x}. \textit{g (f x)} = \textit{x}) \\
\hspace{2em} \textbf{and} \quad (\forall \textit{y}. \textit{f (g y)} = \textit{y}) \\
\hspace{1em} \textbf{shows} \quad \textit{bij f} \\
\hspace{1em} \textbf{by} \quad (\textit{metis assms bijI'}) \\
\textbf{end} \\

\section{Pseudocomplemented Algebras} \\
This theory expands lattices with a pseudocomplement operation. In particular, we consider the following algebraic structures: \\
* pseudocomplemented lattices (p-algebras) \\
* pseudocomplemented distributive lattices (distributive p-algebras) \\
* Stone algebras \\
* Heyting semilattices \\
* Heyting lattices
Heyting algebras

Heyting-Stone algebras

Brouwer algebras

Boolean algebras

Most of these structures and many results in this theory are discussed in [4, 5, 6, 8, 17, 23].

theory P-Algebras

imports Lattice-Basics

begin

3.1 P-Algebras

In this section we add a pseudocomplement operation to lattices and to distributive lattices.

3.1.1 Pseudocomplemented Lattices

The pseudocomplement of an element \( y \) is the greatest element whose meet with \( y \) is the least element of the lattice.

class p-algebra = bounded-lattice + uminus +

assumes pseudo-complement: \( x \sqcap y = \bot \iff x \leq -y \)

begin

Regular elements and dense elements are frequently used in pseudocomplemented algebras.

abbreviation regular \( x \equiv x = -x \)
abbreviation dense \( x \equiv -x = \bot \)
abbreviation complemented \( x \equiv \exists y . x \sqcap y = \bot \land x \sqcup y = \top \)
abbreviation in-p-image \( x \equiv \exists y . x = -y \)
abbreviation selection \( s x \equiv s = -s \sqcap x \)

abbreviation dense-elements \( \equiv \{ x . \text{dense} \ x \} \)
abbreviation regular-elements \( \equiv \{ x . \text{in-p-image} \ x \} \)

lemma p-bot [simp]:

\(-\bot = \top\)

using inf-top.left-neutral pseudo-complement top-unique by blast

lemma p-top [simp]:

\(-\top = \bot\)

by (metis eq-refl inf-top.comm-neutral pseudo-complement)
The pseudocomplement satisfies the following half of the requirements of a complement.

**Lemma inf-p [simp]:**

\[ x \sqcap -x = \bot \]

using inf.commute pseudo-complement by fastforce

**Lemma p-inf [simp]:**

\[ -x \sqcap x = \bot \]

by (simp add: inf.commute)

**Lemma pp-inf-p:**

\[ -x \sqcap -x = \bot \]

by simp

The double complement is a closure operation.

**Lemma pp-increasing:**

\[ x \leq -{-x} \]

using inf-p pseudo-complement by blast

**Lemma ppp [simp]:**

\[ -{-x} = -x \]

by (metis antisym inf.commute order-trans pseudo-complement pp-increasing)

**Lemma pp-idempotent:**

\[ -{-{-x}} = -{-x} \]

by simp

**Lemma regular-in-p-image-iff:**

regular \( x \) \iff in-p-image \( x \)

by auto

**Lemma pseudo-complement-pp:**

\[ x \sqcap y = \bot \iff -{-x} \leq -y \]

by (metis inf-commute pseudo-complement pp-increasing)

**Lemma p-antitone:**

\[ x \leq y \implies -y \leq -x \]

by (metis inf-commute order-trans pseudo-complement pp-increasing)

**Lemma p-antitone-sup:**

\[ -(x \sqcup y) \leq -x \]

by (simp add: p-antitone)

**Lemma p-antitone-inf:**

\[ -x \leq -(x \sqcap y) \]

by (simp add: p-antitone)

**Lemma p-antitone-iff:**

\[ x \leq -y \iff y \leq -x \]
using order-lesseq-imp p-antitone pp-increasing by blast

lemma ppp-isotone:
  \( x \leq y \implies \neg\neg x \leq \neg\neg y \)
  by (simp add: p-antitone)

lemma ppp-isotone-sup:
  \( \neg\neg x \leq \neg\neg (x \sqcup y) \)
  by (simp add: p-antitone)

lemma ppp-isotone-inf:
  \( \neg\neg (x \sqcap y) \leq \neg\neg x \)
  by (simp add: p-antitone)

One of De Morgan’s laws holds in pseudocomplemented lattices.

lemma p-dist-sup [simp]:
  \( -(x \sqcup y) = -x \sqcap -y \)
  apply (rule antisym)
  apply (simp add: p-antitone)
  using inf-le1 inf-le2 le-sup-iff p-antitone-iff by blast

lemma p-supdist-inf:
  \( -x \sqcap -y \leq -(x \sqcap y) \)
  by (simp add: p-antitone)

lemma ppp-dist-pp-sup [simp]:
  \( \neg\neg (- x \sqcup - y) = \neg\neg (x \sqcup y) \)
  by simp

lemma p-sup-p [simp]:
  \( -(x \sqcup -x) = \mathbf{bot} \)
  by simp

lemma p-sup-p [simp]:
  \( - (x \sqcup -x) = \mathbf{top} \)
  by simp

lemma dense-pp:
  dense \( x \leftrightarrow \neg\neg x = \mathbf{top} \)
  by (metis p-bot p-top ppp)

lemma dense-sup-p:
  dense \( x \sqcup -x \)
  by simp

lemma regular-char:
  regular \( x \leftrightarrow (\exists y . x = -y) \)
  by auto
Weak forms of the shunting property hold. Most require a pseudocomplemented element on the right-hand side.

lemma p-shunting-swap:
\[ x \sqcap y \leq -z \iff x \sqcap z \leq -y \]
by (metis inf-assoc inf-commute pseudo-complement)

lemma pp-inf-below-iff:
\[ x \sqcap y \leq -z \iff x \sqcap y \leq -z \]
by (simp add: inf-commute p-shunting-swap)

lemma p-inf-pp [simp]:
\[ -(-x \sqcap -y) = -(x \sqcap y) \]
apply (rule antisym)
apply (simp add: inf.coboundedI2 p-antitone pp-increasing)
using inf-commute p-antitone-iff pp-inf-below-iff by auto

lemma regular-closed-inf:
regular \( x \) \( \Rightarrow \) regular \( y \) \( \Rightarrow \) regular \( x \sqcap y \)
by (metis p-dist-sup ppp)

lemma regular-closed-p:
regular \( -x \)
by simp

lemma regular-closed-pp:
regular \( -x \)
by simp

lemma regular-closed-bot:
regular bot
by simp

lemma regular-closed-top:
regular top
by simp

lemma pp-dist-inf [simp]:
\[ --(x \sqcap y) = --x \sqcap --y \]
by (metis p-dist-sup p-inf-pp-pp ppp)

lemma inf-import-p [simp]:
\( x \cap -(x \cap y) = x \cap -y \)

**apply** (rule antisym)

**using** p-shunting-swap **apply** fastforce

**using** inf.sup-right-isotone p-antitone **by** auto

Pseudocomplements are unique.

**lemma** p-unique:
\[(\forall x . x \cap y = \text{bot} \iff x \leq z) \implies z = -y\]

**using** inf.eq-iff pseudo-complement **by** auto

**lemma** maddux-3-5:
\[x \sqcup x = x \sqcup -(y \sqcup -y)\]

**by** simp

**lemma** shunting-1-pp:
\[x \leq --y \iff x \cap -y = \text{bot}\]

**by** (simp add: pseudo-complement)

**lemma** pp-pp-inf-bot-iff:
\[x \cap y = \text{bot} \iff --x \cap --y = \text{bot}\]

**by** (simp add: pseudo-complement-pp)

**lemma** inf-pp-semi-commute:
\[x \cap --y \leq --(x \cap y)\]

**using** inf.eq-refl p-antitone-iff p-inf-pp **by** presburger

**lemma** inf-pp-commute:
\[--(--x \cap y) = --x \cap --y\]

**by** simp

**lemma** sup-pp-semi-commute:
\[x \cup --y \leq --(x \cup y)\]

**by** (simp add: p-antitone-if)

**lemma** regular-sup:
\[
\text{regular } z \implies (z \leq z \land y \leq z \iff --(x \cup y) \leq z)
\]

**apply** (rule iffI)

**apply** (metis le-supI pp-isotone)

**using** dual-order.trans sup-ge2 pp-increasing pp-isotone-sup **by** blast

**lemma** dense-closed-inf:
\[\text{dense } x \implies \text{dense } y \implies \text{dense } (x \cap y)\]

**by** (simp add: dense-pp)

**lemma** dense-closed-sup:
\[\text{dense } x \implies \text{dense } y \implies \text{dense } (x \cup y)\]

**by** simp

**lemma** dense-closed-pp:

\[16\]
dense \( x \implies \text{dense } (\neg\neg x) \)
by simp

**lemma** dense-closed-top:
dense top
by simp

**lemma** dense-up-closed:
dense \( x \implies x \leq y \implies \text{dense } y \)
using dense-pp top-le pp-isotone by auto

**lemma** regular-dense-top:
regular \( x \implies \text{dense } x \implies x = \top \)
using p-bot by blast

**lemma** selection-char:
selection \( s \ x \Longleftrightarrow (\exists \ y . s = \neg y \cap x) \)
by (metis inf-import-p inf-commute regular-closed-p)

**lemma** selection-closed-inf:
selection \( s \ x \implies \text{selection } t \ x \implies \text{selection } (s \cap t) \ x \)
by (metis inf-assoc inf-commute inf-idem pp-dist-inf)

**lemma** selection-closed-pp:
regular \( x \implies \text{selection } s \ x \implies \text{selection } (\neg\neg s) \ x \)
by (metis pp-dist-inf)

**lemma** selection-closed-bot:
selection bot \( x \)
by simp

**lemma** selection-closed-id:
selection \( x \ x \)
using inf.le-iff-sup pp-increasing by auto

Conjugates are usually studied for Boolean algebras, however, some of their properties generalise to pseudocomplemented algebras.

**lemma** conjugate-unique-p:
assumes conjugate \( f \ g \)
and conjugate \( f \ h \)
shows \( \uminus \circ g = \uminus \circ h \)

**proof** –
have \( \forall x . x \cap g y = \bot \Longleftrightarrow x \cap h y = \bot \)
using assms conjugate-def inf.commute by simp
hence \( \forall x . x \leq (g y) \Longleftrightarrow x \leq (h y) \)
using inf.commute pseudo-complement by simp
hence \( \forall y . (g y) = (h y) \)
using eq-iff by blast
thus ?thesis
by auto

qed

lemma conjugate-symmetric:
conjugate $fg$ $\implies$ conjugate $gf$
by (simp add: conjugate-def inf-commute)

lemma additive-isotone:
additive $f$ $\implies$ isotone $f$
by (metis additive-def isotone-def le-iff-sup)

lemma dual-additive-antitone:
assumes dual-additive $f$
shows isotone $\ominus f$
proof
  have $\forall x y. f (x \sqcup y) \leq f x$
    using assms dual-additive-def by simp
  hence $\forall x y. x \leq y \implies f y \leq f x$
    by (metis sup-absorb2)
  hence $\forall x y. x \leq y \implies -(f x) \leq -(f y)$
    by (simp add: p-antitone)
  thus $?thesis$
    by (simp add: isotone-def)
  qed

lemma conjugate-dual-additive:
assumes conjugate $fg$
shows dual-additive $\ominus f$
proof
  have $1: \forall x y z. -z \leq -(f (x \sqcup y)) \iff -z \leq -(f x) \land -z \leq -(f y)$
  proof (intro allI)
    fix $x y z$
    have $(-z \leq -(f (x \sqcup y))) = (f (x \sqcup y) \sqcap -z = \bot)$
      by (simp add: p-antitone-iff pseudo-complement)
    also have $... = ((x \sqcup y) \sqcap g(-z) = \bot)$
      using assms conjugate-def by auto
    also have $... = (x \sqcup y \leq -(g(-z)))$
      by (simp add: pseudo-complement)
    also have $... = (x \leq -(g(-z)) \land y \leq -(g(-z)))$
      by (simp add: le-sup-iff)
    also have $... = (x \cap g(-z) = \bot \land g(-z) = \bot)$
      by (simp add: pseudo-complement)
    also have $... = (f x \sqcap -z = \bot \land f y \sqcap -z = \bot)$
      using assms conjugate-def by auto
    also have $... = (-z \leq -(f x) \land -z \leq -(f y))$
      by (simp add: p-antitone-iff pseudo-complement)
    finally show $-z \leq -(f (x \sqcup y)) \iff -z \leq -(f x) \land -z \leq -(f y)$
      by simp
  qed
\[
\forall x \, y. \quad -(f (x \sqcup y)) = -(f x) \sqcap -(f y)
\]

proof (intro allI)
  fix \(x\) \(y\)
  have \(-\(f x\) \sqcap -(f y) = --\(-(f x) \sqcap -(f y)\)\)
    by simp
  hence \(-\(f x\) \sqcap -(f y) \leq -(f (x \sqcup y))\)
    using \(f\) by (metis inf-le1 inf-le2)
  thus \(-\(f (x \sqcup y)\) = -(f x) \sqcap -(f y)\)
    using \(f\) antisym by fastforce
qed

thus \(\text{thesis}\)
  using dual-additive-def by simp
qed

lemma conjugate-isotone-pp:
  \(\text{conjugate } f \, g \implies \text{isotone } (\text{uminus} \circ \text{uminus} \circ f)\)
by (simp add: comp-assoc conjugate-dual-additive dual-additive-antitone)

lemma conjugate-char-1-pp:
  \(\text{conjugate } f \, g \iff (\forall x \, y. \quad f(x \sqcap -(g y)) \leq --f x \sqcap y \land g(y \sqcap -(f x)) \leq --g y \sqcap --x)\)
proof (intro allI)
  fix \(x\) \(y\)
  have \(2: f(x \sqcap -(g y)) \leq \neg y\)
    using \(1\) by (simp add: conjugate-def pseudo-complement)
  have \(f(x \sqcap -(g y)) \leq \neg f(x \sqcap -(g y))\)
    by (simp add: pp-increasing)
  also have \(... \leq \neg f x\)
    using \(1\) conjugate-isotone-pp isotone-def by simp
  finally have \(3: f(x \sqcap -(g y)) \leq --f x \sqcap y \land g(y \sqcap -(f x)) \leq --g y \sqcap --x\)
    using \(2\) by simp
  have \(4: \text{isotone } (\text{uminus} \circ \text{uminus} \circ g)\)
    using \(1\) conjugate-isotone-pp conjugate-symmetric by auto
  have \(5: g(y \sqcap -(f x)) \leq --x\)
    using \(f\) by (metis conjugate-def inf.cobounded2 inf-commute)
  have \(g(y \sqcap -(f x)) \leq \neg y (y \sqcap -(f x))\)
    by (simp add: pp-increasing)
  also have \(... \leq \neg g y\)
    using \(4\) isotone-def by auto
  finally have \(g(y \sqcap -(f x)) \leq \neg g y \sqcap --x\)
    using \(5\) by simp
  thus \(f(x \sqcap -(g y)) \leq --f x \sqcap y \land g(y \sqcap -(f x)) \leq --g y \sqcap --x\)
    using \(3\) by simp
qed

next

19
assume 6: \(\forall x \ y. f(x \sqcap -(g \ y)) \leq --f x \sqcap -y \land g(y \sqcap -(f \ x)) \leq --g y \sqcap -x\)
hence 7: \(\forall x \ y. f x \sqcap y = \bot \longrightarrow x \sqcap g y = \bot\)
by (metis inf.le-iff-sup inf.le-sup-iff inf-commute pseudo-complement)

have \(\forall x \ y. x \sqcap g y = \bot \longrightarrow f x \sqcap y = \bot\)
using 6 by (metis inf.le-iff-sup inf.le-sup-iff inf-commute pseudo-complement)
thus conjugate \(f \ g\)
using 7 conjugate-def by auto

lemma conjugate-char-1-isotone:
\(\text{conjugate } f \ g \Longrightarrow \text{isotone } f \ \Longrightarrow \text{isotone } g \ \Longrightarrow f(x \sqcap -(g \ y)) \leq f x \sqcap -y \land g(y \sqcap -(f \ x)) \leq g y \sqcap -x\)
by (simp add: conjugate-char-1-pp ord.isotone-def)

lemma dense-lattice-char-1:
\((\forall x \ y. x \sqcap y = \bot \longrightarrow x = \bot \lor y = \bot) \iff (\forall x. x \neq \bot \longrightarrow \text{dense } x)\)
by (metis inf-top.left-neutral p-bot p-inf pp-inf-bot-iff)

lemma dense-lattice-char-2:
\((\forall x \ y. x \sqcap y = \bot \longrightarrow x = \bot \lor y = \bot) \iff (\forall x. \text{regular } x \longrightarrow x = \bot \lor x = \top)\)
by (metis dense-lattice-char-1 inf-top.left-neutral p-inf regular-closed-p regular-closed-top)

lemma restrict-below-Rep-eq:
\(x \sqcap -(y \leq z) \Longrightarrow x \sqcap y = x \sqcap z \sqcap y\)
by (metis inf.absorb2 inf.commute inf.left-commute pp-increasing)

end

The following class gives equational axioms for the pseudocomplement operation.

class p-algebra-eq = bounded-lattice + uminus +
assumes p-bot-eq: \(-\bot = \top\)
and p-top-eq: \(-\top = \bot\)
and inf-import-p-eq: \(x \sqcap -(x \sqcap y) = x \sqcap -y\)
begin

lemma inf-p-eq:
\(x \sqcap -x = \bot\)
by (metis inf-bot-right inf-import-p-eq inf-top-right p-top-eq)

subclass p-algebra
apply unfold-locales
apply (rule iffI)
apply (metis inf.orderI inf-import-p-eq inf-top.right-neutral p-bot-eq)
3.1.2 Pseudocomplemented Distributive Lattices

We obtain further properties if we assume that the lattice operations are distributive.

class pd-algebra = p-algebra + bounded-distrib-lattice
begin

lemma p-inf-sup-below:
\(-x \cap (x \sqcup y) \leq y\)
by (simp add: inf-sup-distrib1)

lemma pp-inf-sup-p [simp]:
\(-x \cap (x \sqcup -x) = x\)
using inf.absorb2 inf-sup-distrib1 pp-increasing by auto

lemma complement-p:
\(x \cap y = bot \implies x \sqcup y = top \implies -x = y\)
by (metis pseudo-complement inf.commute inf-top.left-neutral sup.absorb-iff1
sup.commute sup-bot.right-neutral sup-inf-distrib2 p-inf)

lemma complemented-regular:
complemented x \implies regular x
using complement-p inf.commute sup.commute by fastforce

lemma regular-inf-dense:
\(\exists y z . \text{regular } y \land \text{dense } z \land x = y \cap z\)
by (metis pp-inf-sup-p dense-sup-p ppp)

lemma maddux-3-12 [simp]:
\((x \sqcup -y) \cap (x \sqcup y) = x\)
by (metis p-inf sup-bot-right sup-inf-distrib1)

lemma maddux-3-13 [simp]:
\((x \sqcup y) \cap -x = y \cap -x\)
by (simp add: inf-sup-distrib2)

lemma maddux-3-20:
\(((v \sqcap w) \sqcup (-v \sqcap x)) \cap -((v \sqcap y) \sqcup (-v \sqcap z)) = (v \sqcap w \sqcap -y) \sqcup (-v \sqcap x \sqcap -z)\)
proof
  have \(v \sqcap w \sqcap -(v \sqcap y) \sqcap -(-v \sqcap z) = v \sqcap w \sqcap -(v \sqcap y)\)
  by (meson inf.cobounded1 inf-absorb1 le-iff1 p-antitone-iff)
  also have \(\ldots = v \sqcap w \sqcap -y\)
  using inf.sup-relative-same-increasing inf-import-p inf-le1 by blast
finally have 1: \( v \sqcap w \sqcap -(v \sqcap y) \sqcap -(v \sqcap z) = v \sqcap w \sqcap -y \)

have \(-v \sqcap x \sqcap -(v \sqcap y) \sqcap -(v \sqcap z) = -v \sqcap x \sqcap -(v \sqcap z)\)
by (simp add: inf.absorb1 le-infI1 p-antitone-inf)
also have \( ... = -v \sqcap x \sqcap -z \)
by (simp add: inf.assoc inf-left-commute)
finally have 2: \(-v \sqcap x \sqcap -(v \sqcap y) \sqcap -(v \sqcap z) = -v \sqcap x \sqcap -z\)

have \(((v \sqcap w) \sqcup -(v \sqcap x)) \sqcap -((v \sqcap y) \sqcup -(v \sqcap z)) = (v \sqcap w \sqcap -(v \sqcap y) \sqcap -(v \sqcap z)) \sqcup -(v \sqcap x) \sqcap -(v \sqcap y) \sqcap -(v \sqcap z)\)
by (simp add: inf-assoc inf-sup-distrib2)
also have \( ... = (v \sqcap w \sqcap -y) \sqcup -(v \sqcap x \sqcap -z)\)
using 1 2 by simp
finally show \(?thesis\)

qed

lemma order-char-1:
\( x \leq y \iff x \leq y \sqcup -x \)
by (metis inf.sup-left-isotone inf-sup-absorb le-supI1 maddux-3-12 sup-commute)

lemma order-char-2:
\( x \leq y \iff x \sqcup -x \leq y \sqcup -x \)
using order-char-1 by auto

end

3.2 Stone Algebras

A Stone algebra is a distributive lattice with a pseudocomplement that satisfies the following equation. We thus obtain the other half of the requirements of a complement at least for the regular elements.

class stone-algebra = pd-algebra +
assumes stone [simp]: \(-x \sqcup -x = top\)
begins

As a consequence, we obtain both De Morgan’s laws for all elements.

lemma p-dist-inf [simp]:
\(-(x \sqcap y) = -x \sqcup -y\)
proof (rule p-unique[THEN sym], rule allI, rule iffI)
fix \(w\)
assume \(w \sqcap (x \sqcap y) = bot\)

hence \(w \sqcap -x \sqcap y = bot\)
using inf-commute inf-left-commute pseudo-complement by auto

hence 1: \(w \sqcap -x \leq -y\)

by (simp add: pseudo-complement)
have \( w = (w \cap -x) \cup (w \cap -y) \)
using distrib-imp2 sup-inf-distrib1 by auto
thus \( w \leq -x \cup -y \)
using 1 by (metis inf-le2 supmono)

next
fix \( w \)
assume \( w \leq -x \cup -y \)
thus \( w \leq -x \cup -y \)
using distrib-imp2 sup-inf-distrib1 by auto

thus \( w \leq -x \cup -y \)
using order-trans p-supdist-inf pseudo-complement by blast

qed

lemma pp-dist-sup [simp]:
\(-y = -x \cup -y\)
by simp

lemma regular-closed-sup:
regular \( x \Rightarrow regular y \Rightarrow regular (x \cup y) \)
by simp

The regular elements are precisely the ones having a complement.

lemma regular-complemented-iff:
regular \( x \iff \) complemented \( x \)
by (metis inf-p stone complemented-regular)

lemma selection-closed-sup:
selection \( s \cup x \Rightarrow selection \ t \cup x \Rightarrow selection \ (s \cup t) \setminus x \)
by simp add: inf-sup-distrib2

lemma huntington-3-pp [simp]:
\(-y \cup -y \cup -x \cup y = -x\)
by (metis p-dist-inf p-inf sup commut supinf-distrib1)

lemma maddux-3-3 [simp]:
\(-y \cup -y \cup -x = -x\)
by simp add: sup commut supinf-distrib1

lemma maddux-3-11-pp:
\((x \cap -y) \cup (x \cap -y) = x\)
by (metis inf-sup-distrib1 inf-top-right stone)

lemma maddux-3-19-pp:
\((-y \cap -y) \cup (-y \cap -y) = (-y \cap -y) \cup (-y \cap -y)\)
proof
have \((-y \cap -y) \cap (-y \cap -y) = (-y \cap -y) \cup (-y \cap -y)\)
by (simp add: inf commut inf-sup-distrib1 sup assoc)
also have ... = \((-y \cap -y) \cup (-y \cap -y) \cup (-y \cap -y)\)
by simp
also have ... = \((-y \cap -y) \cup (-y \cap -y) \cup (-y \cap -y)\)
using inf-sup-distrib1 sup assoc inf commut inf assoc by presburger

23
also have \( (-x \cap z) \cup (y \cap -x) \cup (y \cap z \cap -x) \)
  by simp
also have \( (-x \cap z) \cup (-x \cap z \cap y) \cup (y \cap -x) \)
  by (simp add: inf-assoc inf-commute sup.left-commute sup-commute)
also have \( (-x \cap z) \cup (y \cap -x) \)
  by simp
finally show \(?thesis \)
  by (simp add: inf-commute sup-commute)
qed

lemma compl-inter-eq-pp:
\( (-x \cap y = (-x \cap z) \cap (-x \cap z \cap y) \cap (y \cap -x) \)
  by (simp add: inf-assoc inf-commute sup.left-commute sup-commute)

lemma maddux-3-21-pp [simp]:
\( (-x \cup (-x \cap y) = (-x \cup y) \)
  by (simp add: sup.commute sup-inf-distrib1)

lemma shunting-2-pp:
\( x \leq -y \iff -x \cup -y = \top \)
  by (metis inf-top-left p-bot p-dist-inf pseudo-complement)

lemma shunting-p:
\( x \cap y \leq -z \iff x \leq -z \cup -y \)
  by (metis inf.assoc p-dist-inf p-shunting-swap pseudo-complement)

The following weak shunting property is interesting as it does not require
the element \( z \) on the right-hand side to be regular.

lemma shunting-var-p:
\( x \cap -y \leq z \iff x \leq z \cup -y \)
proof
  assume \( x \cap -y \leq z \)
  hence \( z \cup -y = -y \cup (z \cup x \cap -y) \)
    by (simp add: sup.absorb1 sup.commute)
  thus \( x \leq z \cup -y \)
    by (metis inf-commute maddux-3-21-pp sup.commute sup.left-commute
        sup-left-divisibility)
next
  assume \( x \leq z \cup -y \)
  thus \( x \cap -y \leq z \)
    by (metis inf.mono maddux-3-12 sup.ge2)
qed

lemma conjugate-char-2-pp:
\( \text{conjugate } f \ g \iff f \ bot = \bot \wedge g \ bot = \bot \wedge (\forall \ x \ y . \ f \ x \cap y \leq -(-(f(\ x \cap
-(-g \ y)\)) \wedge g \ y \cap x \leq -(-(g(y \cap -(-f \ x))))\)
proof
\textbf{assume} 1: \textit{conjugate} \( f \ g \)
\textbf{hence} 2: \textit{dual-additive} (\textit{uminus} \circ \ g)
\textbf{using} \textit{conjugate-symmetric} \textit{conjugate-dual-additive} \textbf{by auto}
\textbf{show} \( f \ bot = bot \land g \ bot = bot \land (\forall x y \cdot f x \sqcap y \leq -(f(x \sqcap -(g y))) \land g \ y \sqcap x \leq -(g(y \sqcap -(f x)))) \)
\textbf{proof} \ (\textit{intro conjI})
\textbf{show} \( f \ bot = bot \)
\textbf{using} \ 1 \ \textbf{by} \ \{ \textit{metis conjugate-def inf-idem inf-bot-left} \}
\textbf{next}
\textbf{show} \( g \ bot = bot \)
\textbf{using} \ 1 \ \textbf{by} \ \{ \textit{metis conjugate-def inf-idem inf-bot-right} \}
\textbf{next}
\textbf{show} \( \forall x y \cdot f x \sqcap y \leq -(f(x \sqcap -(g y))) \land g \ y \sqcap x \leq -(g(y \sqcap -(f x))) \)
\textbf{proof} \ (\textit{intro allI})
\textbf{fix} \( x \ y \)
\textbf{have} \ 3: \( y \leq -(f(x \sqcap -(g y))) \)
\textbf{using} \ 1 \ \textbf{by} \ \{ \textit{simp add: conjugate-def pseudo-complement inf-commute} \}
\textbf{have} \ 4: \( x \leq -(g(y \sqcap -(f x))) \)
\textbf{using} \ 1 \ \textbf{conjugate-def inf-commute pseudo-complement} \ \textbf{by fastforce}
\textbf{have} \( y \sqcap -(f(x \sqcap -(g y))) = y \sqcap -(f(x \sqcap -(g y))) \sqcap -(f(x \sqcap -(g y))) \)
\textbf{using} \ 3 \ \textbf{by} \ \{ \textit{simp add: inf-le-iff sup inf-commute} \}
\textbf{also have} \( \ldots = y \sqcap -(f((x \sqcap -(g y))) \sqcup (x \sqcap -(g y))) \)
\textbf{using} \ 1 \ \textbf{conjugate-dual-additive dual-additive-def inf-assoc} \ \textbf{by auto}
\textbf{also have} \( \ldots = y \sqcap -(f x) \)
\textbf{by} \ \{ \textit{simp add: maddux-3-11-pp} \}
\textbf{also have} \( \ldots \leq -(f x) \)
\textbf{by} \ \{ \textit{simp} \}
\textbf{finally have} \ 5: \( f x \sqcap y \leq -(f(x \sqcap -(g y))) \)
\textbf{by} \ \{ \textit{simp add: inf-commute p-shunting-swap} \}
\textbf{have} \( x \sqcap -(g(y \sqcap -(f x))) = x \sqcap -(g(y \sqcap -(f x))) \sqcap -(g(y \sqcap -(f x))) \)
\textbf{using} \ 4 \ \textbf{by} \ \{ \textit{simp add: inf-le-iff sup inf-commute} \}
\textbf{also have} \( \ldots = x \sqcap -(g((y \sqcap -(f x))) \sqcup (y \sqcap -(f x))) \)
\textbf{using} \ 2 \ \textbf{by} \ \{ \textit{simp add: dual-additive-def inf-assoc} \}
\textbf{also have} \( \ldots = x \sqcap -(g y) \)
\textbf{by} \ \{ \textit{simp add: maddux-3-11-pp} \}
\textbf{also have} \( \ldots \leq -(g y) \)
\textbf{by} \ \{ \textit{simp} \}
\textbf{finally have} \( g y \sqcap x \leq -(g(y \sqcap -(f x))) \)
\textbf{by} \ \{ \textit{simp add: inf-commute p-shunting-swap} \}
\textbf{thus} \( f x \sqcap y \leq -(f(x \sqcap -(g y))) \land g y \sqcap x \leq -(g(y \sqcap -(f x))) \)
\textbf{using} \ 5 \ \textbf{by} \ \{ \textit{simp} \}
\textbf{qed}
\textbf{qed}
\textbf{next}
\textbf{assume} \( f \ bot = bot \land g \ bot = bot \land (\forall x y \cdot f x \sqcap y \leq -(f(x \sqcap -(g y))) \land g \ y \sqcap x \leq -(g(y \sqcap -(f x))) \)
\textbf{thus} \ \textit{conjugate} \( f \ g \)
\textbf{by} \ \{ \textit{unfold conjugate-def , metis inf-commute le-bot pp-inf-bot-iff regular-closed-bot} \}

25
lemma conjugate-char-2-pp-additive:
  assumes conjugate \( f \) \( g \)
  and additive \( f \)
  and additive \( g \)
  shows \( f \ x \cap y \leq f(\ x \cap \neg\neg(g \ y)) \land g \ y \cap x \leq g(\ y \cap \neg\neg(\ f \ x)) \)
proof –
  have \( f \ x \cap y = f \ ((\ x \cap \neg\neg g \ y) \cup (\ x \cap \neg g \ y)) \cap y \)
    by (simp add: sup.commute sup-inf-distrib1)
also have \( \ldots = (f(\ x \cap \neg\neg g \ y) \cap y) \cup (f(\ x \cap \neg g \ y) \cap y) \)
    using assms(2) additive-def inf-sup-distrib2 by auto
also have \( \ldots = f \ (\ x \cap \neg\neg g \ y) \cap y \)
    by (metis assms(1) conjugate-def inf-le2 pseudo-complement sup-bot.right-neutral)
finally have \( 2: f \ x \cap y \leq f \ (\ x \cap \neg\neg g \ y) \)
    by simp
  have \( g \ y \cap x = g \ ((\ y \cap \neg\neg f \ x) \cup (\ y \cap \neg f \ x)) \cap x \)
    by (simp add: sup.commute sup-inf-distrib1)
also have \( \ldots = (g(\ y \cap \neg\neg f \ x) \cap x) \cup (g(\ y \cap \neg f \ x) \cap x) \)
    using assms(3) additive-def inf-sup-distrib2 by auto
also have \( \ldots = g \ (\ y \cap \neg\neg f \ x) \cap x \)
    by (metis assms(1) conjugate-def inf.cobounded2 pseudo-complement sup-bot.right-neutral inf-commute)
finally have \( g \ y \cap x \leq g \ (\ y \cap \neg\neg f \ x) \)
    by simp
thus \( \?thesis \)
  using \( 2 \) by simp
qed

end

Every bounded linear order can be expanded to a Stone algebra. The pseudocomplement takes \( \text{bot} \) to the \( \text{top} \) and every other element to \( \text{bot} \).

class linorder-stone-algebra-expansion = linorder-lattice-expansion + uminus +
  assumes uminus-def \[ simp \]: \( -x = (if \ x = \text{bot} \ then \text{top} \ else \text{bot}) \)
begin

subclass stone-algebra
  apply unfold-locales
  using bot-unique min-def top-le by auto

The regular elements are the least and greatest elements. All elements except the least element are dense.

lemma regular-bot-top:
  regular \( x \iff x = \text{bot} \lor x = \text{top} \)
  by simp
lemma not-bot-dense: 
\[ x \neq \text{bot} \implies \neg \neg x = \text{top} \]
by simp

end

3.3 Heyting Algebras

In this section we add a relative pseudocomplement operation to semilattices and to lattices.

3.3.1 Heyting Semilattices

The pseudocomplement of an element \( y \) relative to an element \( z \) is the least element whose meet with \( y \) is below \( z \). This can be stated as a Galois connection. Specialising \( z = \text{bot} \) gives (non-relative) pseudocomplements. Many properties can already be shown if the underlying structure is just a semilattice.

class implies =
fixes implies :: \( 'a \Rightarrow 'a \Rightarrow 'a \) (infixl \( \Rightarrow \))

class heyting-semilattice = semilattice-inf + implies +
assumes implies-galois: \( x \sqcap y \leq z \iff x \leq y \Rightarrow z \)
begin

lemma implies-below-eq [simp]:
\[ y \sqcap (x \Rightarrow y) = y \]
using implies-galois inf.absorb-iff1 inf.cobounded1 by blast

lemma implies-increasing:
\[ x \leq y \Rightarrow x \]
by (simp add: inf.orderI)

lemma implies-galois-swap:
\[ x \leq y \Rightarrow z \iff y \leq x \Rightarrow z \]
by (metis implies-galois inf-commute)

lemma implies-galois-var:
\[ x \sqcap y \leq z \iff y \leq x \Rightarrow z \]
by (simp add: implies-galois-swap implies-galois)

lemma implies-galois-increasing:
\[ x \leq y \Rightarrow (x \sqcap y) \]
using implies-galois by blast

lemma implies-galois-decreasing:
\[ (y \Rightarrow x) \sqcap y \leq x \]
using implies-galois by blast

lemma implies-mp-below:
  \( x \cap (x \leadsto y) \leq y \)
using implies-galois-decreasing inf-commute by auto

lemma implies-isotone:
  \( x \leq y \implies z \leadsto x \leq z \leadsto y \)
using implies-galois order-trans by blast

lemma implies-antitone:
  \( x \leq y \implies y \leadsto z \leq x \leadsto z \)
by (meson implies-galois-swap order-lesseq-imp)

lemma implies-isotone-inf:
  \( x \leadsto (y \cap z) \leq x \leadsto y \)
by (simp add: implies-isotone)

lemma implies-antitone-inf:
  \( x \leadsto z \leq (x \cap y) \leadsto z \)
by (simp add: implies-antitone)

lemma implies-curry:
  \( x \leadsto (y \leadsto z) = (x \cap y) \leadsto z \)
by (metis implies-galois-decreasing implies-galois inf-assoc antisym)

lemma implies-curry-flip:
  \( x \leadsto (y \leadsto z) = y \leadsto (x \leadsto z) \)
by (simp add: implies-curry inf-commute)

lemma triple-implies [simp]:
  \( (x \leadsto y) \leadsto y = x \leadsto y \)
using implies-antitone implies-galois-swap eq-iff by auto

lemma implies-mp-eq [simp]:
  \( x \cap (x \leadsto y) = x \cap y \)
by (metis implies-below-eq implies-mp-below inf-left-commute inf.absorb2)

lemma implies-dist-implies:
  \( x \leadsto (y \leadsto z) \leq (x \leadsto y) \leadsto (x \leadsto z) \)
using implies-curry implies-curry-flip by auto

lemma implies-import-inf [simp]:
  \( x \cap ((x \cap y) \leadsto (x \leadsto z)) = x \cap (y \leadsto z) \)
by (metis implies-curry implies-mp-eq inf-commute)

lemma implies-dist-inf:
  \( x \leadsto (y \cap z) = (x \leadsto y) \cap (x \leadsto z) \)
proof –

28
have \((x \leadsto y) \cap (x \leadsto z) \cap x \leq y \cap z\)
by (simp add: implies-galois)
hence \((x \leadsto y) \cap (x \leadsto z) \leq x \leadsto (y \cap z)\)
using implies-galois by blast
thus \(\text{thesis}\)
by (simp add: implies-isotone eq-iff)
qede

lemma implies-itself-top:
\(y \leq x \leadsto x\)
by (simp add: implies-galois-swap implies-increasing)

lemma inf-implies-top:
\(z \leq (x \cap y) \leadsto x\)
using implies-galois-var le-infI by blast

lemma inf-inf-implies [simp]:
\(z \cap ((x \cap y) \leadsto x) = z\)
by (simp add: inf-implies-top inf-absorb1)

lemma le-implies-top:
\(x \leq y =\Rightarrow z \leq x \leadsto y\)
using implies-antitone implies-itself-top order.trans by blast

lemma le-iff-le-implies:
\(x \leq y \iff x \leq x \leadsto y\)
using implies-galois inf-idem by force

lemma implies-inf-isotone:
\(x \leadsto y \leq (x \cap z) \leadsto (y \cap z)\)
by (metis implies-curry implies-galois-increasing implies-isotone)

lemma implies-transitive:
\((x \leadsto y) \cap (y \leadsto z) \leq x \leadsto z\)
using implies-dist-implies implies-galois-var implies-increasing order-lesseq-imp
by blast

lemma implies-inf-absorb [simp]:
\(x \leadsto (x \cap y) = x \leadsto y\)
using implies-dist-inf implies-itself-top inf.absorb-iff2 by auto

lemma implies-implies-absorb [simp]:
\(x \leadsto (x \leadsto y) = x \leadsto y\)
by (simp add: implies-curry)

lemma implies-inf-identity:
\((x \leadsto y) \cap y = y\)
by (simp add: inf-commute)
**lemma** implies-itself-same:

\[ x \rightsquigarrow x = y \rightsquigarrow y \]

*by (simp add: le-implies-top eq-iff)*

**end**

The following class gives equational axioms for the relative pseudocomplement operation (inequalities can be written as equations).

**class** heyting-semilattice-eq = semilattice-inf + implies +

**assumes** implies-mp-below: \( x \cap (x \rightsquigarrow y) \leq y \)

*and* implies-galois-increasing: \( x \leq y \rightsquigarrow (x \cap y) \)

*and* implies-isotone-inf: \( x \rightsquigarrow (y \cap z) \leq x \rightsquigarrow y \)

**begin**

**subclass** heyting-semilattice

*apply unfold-locales*

*apply (rule iffI)*

*apply (metis implies-galois-inf le-inf2 top-inf)*

*by (metis implies-mp-below inf-commute order-trans inf-mono order-refl)*

**end**

The following class allows us to explicitly give the pseudocomplement of an element relative to itself.

**class** bounded-heyting-semilattice = bounded-semilattice-inf-top + heyting-semilattice

**begin**

**lemma** implies-itself [simp]:

\[ x \rightsquigarrow x = \top \]

*using* implies-galois inf-le2 top-le *by blast*

**lemma** implies-order:

\[ x \leq y \iff x \rightsquigarrow y = \top \]

*by (metis implies-galois-inf top-left-neutral top-unique)*

**lemma** inf-implies [simp]:

\( (x \cap y) \rightsquigarrow x = \top \)

*using* implies-order inf-le1 *by blast*

**lemma** top-implies [simp]:

\[ \top \rightsquigarrow x = x \]

*by (metis implies-mp-eq inf-top.left-neutral)*

**end**

**3.3.2 Heyting Lattices**
We obtain further properties if the underlying structure is a lattice. In particular, the lattice operations are automatically distributive in this case.

```plaintext
class heyting-lattice = lattice + heyting-semilattice
begin

lemma sup-distrib-inf-le:
  \((x \sqcup y) \sqcap (x \sqcup z) \leq x \sqcup (y \sqcap z)\)
proof
  have \(x \sqcup z \leq y \Rightarrow (x \sqcup (y \sqcap z))\)
  using implies-galois-var implies-increasing sup.bounded-iff sup.cobounded2 by blast
  hence \(x \sqcup y \leq (x \sqcup z) \Rightarrow (x \sqcup (y \sqcap z))\)
  using implies-galois-swap implies-increasing le-sup-iff by blast
  thus \(?\)thesis
  by (simp add: implies-galois)
qed

subclass distrib-lattice
  apply unfold-locales
  using distrib-sup-le eq-iff sup-distrib-inf-le by auto

lemma implies-isotone-sup:
  \(x \Rightarrow y \leq x \Rightarrow (y \sqcup z)\)
by (simp add: implies-isotone)

lemma implies-antitone-sup:
  \((x \sqcup y) \Rightarrow z \leq x \Rightarrow z\)
by (simp add: implies-antitone)

lemma implies-sup:
  \(x \Rightarrow z \leq (y \Rightarrow z) \Rightarrow ((x \sqcup y) \Rightarrow z)\)
proof
  have \((x \Rightarrow z) \sqcap (y \Rightarrow z) \sqcap y \leq z\)
  by (simp add: implies-galois)
  hence \((x \Rightarrow z) \sqcap (y \Rightarrow z) \sqcap (x \sqcup y) \leq z\)
  using implies-galois-swap implies-galois-var by fastforce
  thus \(?\)thesis
  by (simp add: implies-galois)
qed

lemma implies-dist-sup:
  \((x \sqcup y) \Rightarrow z = (x \Rightarrow z) \sqcap (y \Rightarrow z)\)
apply (rule antisym)
apply (simp add: implies-antitone)
by (simp add: implies-sup implies-galois)

lemma implies-antitone-isotone:
  \((x \sqcup y) \Rightarrow (x \sqcap y) \leq x \Rightarrow y\)
by (simp add: implies-antitone-sup implies-dist-inf le-infI2)
```

31
lemma implies-antisymmetry:
  \((x \rightarrow y) \cap (y \rightarrow x) = (x \sqcup y) \rightarrow (x \sqcap y)\)
by (metis implies-dist-sup implies-inf-absorb inf.commute)

lemma sup-inf-implies [simp]:
  \((x \sqcup y) \cap (x \rightarrow y) = y\)
by (simp add: inf-sup-distrib2 sup.absorb2)

lemma implies-subdist-sup:
  \((x \rightarrow y) \sqcup (x \rightarrow z) \leq x \rightarrow (y \sqcup z)\)
by (simp add: implies-isotone)

lemma implies-subdist-inf:
  \((x \rightarrow z) \sqcup (y \rightarrow z) \leq (x \sqcap y) \rightarrow z\)
by (simp add: implies-antitone)

lemma implies-sup-absorb:
  \((x \rightarrow y) \sqcup z \leq (x \sqcup z) \rightarrow (y \sqcup z)\)
by (metis implies-dist-sup implies-isotone-sup implies-increasing inf-inf-implies le-sup-iff sup-inf-implies)

lemma sup-below-implies-implies:
  \(x \sqcup y \leq (x \rightarrow y) \rightarrow y\)
by (simp add: implies-dist-sup implies-galois-swap implies-increasing)

end

class bounded-heyting-lattice = bounded-lattice + heyting-lattice
begin

subclass bounded-heyting-semilattice ..

lemma implies-bot [simp]:
  bot \rightarrow x = top
using implies-galois top-unique by fastforce

end

3.3.3 Heyting Algebras

The pseudocomplement operation can be defined in Heyting algebras, but it is typically not part of their signature. We add the definition as an axiom so that we can use the class hierarchy, for example, to inherit results from the class pd-algebra.

class heyting-algebra = bounded-heyting-lattice + uminus +
  assumes uminus-eq: \(-x = x \rightarrow \text{bot}\)
begin
subclass pd-algebra
  apply unfold-locales
  using bot-unique implies-galois uminus-eq by auto

lemma boolean-implies-below:
$-x \sqcup y \leq x \Rightarrow y$
by (simp add: implies-increasing implies-isotone uminus-eq)

lemma negation-implies:
$-(x \Rightarrow y) = --x \sqcap -y$
proof (rule antisym)
  show $-(x \Rightarrow y) \leq --x \sqcap -y$
    using boolean-implies-below p-antitone by auto
next
  have $x \sqcap -y \sqcap (x \Rightarrow y) = \text{bot}$
    by (metis implies-import-inf inf inf-sup-left-inf-commute inf-left-commute)
  hence $--x \sqcap -y \sqcap (x \Rightarrow y) = \text{bot}$
    using pp-inf-bot-iff inf-assoc by auto
  thus $--x \sqcap -y \leq -(x \Rightarrow y)$
    by (simp add: pseudo-complement)
qed

lemma double-negation-dist-implies:
$--(x \Rightarrow y) = --x \Rightarrow --y$
apply (rule antisym)
apply (metis pp-inf-below-iff implies-galois-decreasing implies-galois
  negation-implies ppp)
by (simp add: p-antitone-iff negation-implies)

end

The following class gives equational axioms for Heyting algebras.

class heyting-algebra-eq = bounded-lattice + implies + uminus +
assumes implies-mp-eq: $x \sqcap (x \Rightarrow y) = x \sqcap y$
  and implies-import-inf: $x \sqcap ((x \sqcap y) \Rightarrow (x \Rightarrow z)) = x \sqcap (y \Rightarrow z)$
  and inf-inf-implies: $z \sqcap ((x \sqcap y) \Rightarrow x) = z$
  and uminus-eq-eq: $-x = x \Rightarrow \text{bot}$
begin
subclass heyting-algebra
  apply unfold-locales
  apply (rule iffI)
  apply (metis implies-import-inf inf-sup-left-divisibility inf-inf-implies le-iff-inf)
  apply (metis implies-mp-eq inf-commute inf-le-sup iff inf-le-sup iff inf-sup-right-isotone)
  by (simp add: uminus-eq-eq)
end
A relative pseudocomplement is not enough to obtain the Stone equation, so we add it in the following class.

```plaintext
class heyting-stone-algebra = heyting-algebra +
   assumes heyting-stone: \(-x \sqcup -x = \top\)
begin

subclass stone-algebra
   by unfold-locales (simp add: heyting-stone)
end
```

3.3.4 Brouwer Algebras

Brouwer algebras are dual to Heyting algebras. The dual pseudocomplement of an element \(y\) relative to an element \(x\) is the least element whose join with \(y\) is above \(x\). We can now use the binary operation provided by Boolean algebras in Isabelle/HOL because it is compatible with dual relative pseudocomplements (not relative pseudocomplements).

```plaintext
class brouwer-algebra = bounded-lattice + minus + uminus +
   assumes minus-galois: \(x \leq y \sqcup z \iff x - y \leq z\)
   and uminus-eq-minus: \(-x = \top - x\)
begin

sublocale brouwer: heyting-algebra where
   inf = sup and less-eq = greater-eq
   and less = greater and sup = inf and bot = top and top = bot and implies =
   \(\lambda x y. y - x\)
   apply unfold-locales
   apply simp
   apply simp
   apply simp
   apply simp
   apply (metis minus-galois sup-commute)
   by (simp add: uminus-eq-minus)

lemma curry-minus:
   \(x - (y \sqcup z) = (x - y) - z\)
   by (simp add: brouwer.implies-curry sup-commute)

lemma minus-subdist-sup:
   \((x - z) \sqcup (y - z) \leq (x \sqcup y) - z\)
   by (simp add: brouwer.implies-dist-inf)

lemma inf-sup-minus:
   \((x \sqcap y) \sqcup (x - y) = x\)
   by (simp add: inf.absorb1 brouwer.inf-sup-distrib2)
```
3.4 Boolean Algebras

This section integrates Boolean algebras in the above hierarchy. In particular, we strengthen several results shown above.

context boolean-algebra
begin

Every Boolean algebra is a Stone algebra, a Heyting algebra and a Brouwer algebra.

subclass stone-algebra
  apply unfold-locales
  apply (rule iffI)
  apply (metis compl-sup-top inf.orderI inf-bot-right inf-sup-distrib1 inf-top-right sup-inf-absorb)
  using inf.commute inf.sup-right-divisibility apply fastforce
  by simp

subclass heyting: heyting-algebra where implies = λx y. ¬x ⊔ y
  apply unfold-locales
  apply (rule iffI)
  using shunting-var-p sup-commute apply fastforce

subclass brouwer-algebra
  apply unfold-locales
  apply (simp add: diff-eq shunting-var-p sup.commute)
  by (simp add: diff-eq)

lemma huntington-3 [simp]:
  ¬(¬x ⊔ ¬y) ⊔ ¬(¬x ⊔ y) = x
  using huntington-3-pp by auto

lemma maddux-3-1:
  x ⊔ ¬x = y ⊔ ¬y
  by simp

lemma maddux-3-4:
  x ⊔ (y ⊔ ¬y) = z ⊔ ¬z
  by simp

lemma maddux-3-11 [simp]:
  (x ⊓ y) ⊔ (x ⊓ ¬y) = x
  using brouwer.maddux-3-12 sup.commute by auto

lemma maddux-3-19:
  (¬x ⊓ y) ⊔ (x ⊓ z) = (x ⊔ y) ⊔ (¬x ⊔ z)
using maddux-3-19-pp by auto

**Lemma** compl-inter-eq:
\[ x \cap y = x \cap z \implies -x \cap y = -x \cap z \implies y = z \]
by (metis inf-commute maddux-3-11)

**Lemma** maddux-3-21 [simp]:
\[ x \cup (-x \cap y) = x \cup y \]
by (simp add: sup-inf-distrib1)

**Lemma** shunting-1:
\[ x \leq y \iff x \cap -y = bot \]
by (simp add: pseudo-complement)

**Lemma** uminus-involutive:
\[ uminus \circ uminus = id \]
by auto

**Lemma** uminus-injective:
\[ uminus \circ f = uminus \circ g \implies f = g \]
by (metis comp-assoc id-o minus-comp-minus)

**Lemma** conjugate-unique:
\[ conjugate f g \implies conjugate f \circ h \implies g = h \]
using conjugate-unique-p uminus-injective by blast

**Lemma** dual-additive-additive:
\[ dual-additive (uminus \circ f) \implies additive f \]
by (metis additive-def compl-eq-compl-iff dual-additive-def p-dist-sup o-def)

**Lemma** conjugate-additive:
\[ conjugate f g \implies additive f \]
by (simp add: conjugate-dual-additive dual-additive-additive)

**Lemma** conjugate-isotone:
\[ conjugate f g \implies isotone f \]
by (simp add: conjugate-additive additive-isotone)

**Lemma** conjugate-char-1:
\[ conjugate f g \iff (\forall x y . f(x \cap -(g y)) \leq f x \cap -y \land g(y \cap -(f x)) \leq g y \cap -x) \]
by (simp add: conjugate-char-1-pp)

**Lemma** conjugate-char-2:
\[ conjugate f g \iff f bot = bot \land g bot = bot \land (\forall x y . f x \cap y \leq f(x \cap g y) \land g y \cap x \leq g(y \cap f x)) \]
by (simp add: conjugate-char-2-pp)

**Lemma** shunting:
\[ x \cap y \leq z \iff x \leq z \cup -y \]
by \((\text{simp add: heyting.implies-galois sup commute})\)

\textbf{lemma} \texttt{shunting-var}:
\[ x \cap -y \leq z \iff x \leq z \cup y \]
by \((\text{simp add: shunting})\)

\textbf{class} \texttt{non-trivial-stone-algebra} = \texttt{non-trivial-bounded-order} + \texttt{stone-algebra}

\textbf{class} \texttt{non-trivial-boolean-algebra} = \texttt{non-trivial-stone-algebra} + \texttt{boolean-algebra}

\textbf{end}

4 Filters

This theory develops filters based on orders, semilattices, lattices and distributive lattices. We prove the ultrafilter lemma for orders with a least element. We show the following structure theorems:

- The set of filters over a directed semilattice forms a lattice with a greatest element.
- The set of filters over a bounded semilattice forms a bounded lattice.
- The set of filters over a distributive lattice with a greatest element forms a bounded distributive lattice.

Another result is that in a distributive lattice ultrafilters are prime filters. We also prove a lemma of Grätzer and Schmidt about principal filters.

We apply these results in proving the construction theorem for Stone algebras (described in a separate theory). See, for example, \[4, 5, 6, 9, 17\] for further results about filters.

\textbf{theory} \texttt{Filters}

\textbf{imports} \texttt{Lattice-Basics}

begin

4.1 Orders

This section gives the basic definitions related to filters in terms of orders. The main result is the ultrafilter lemma.

\textbf{context} \texttt{ord}
begin
abbreviation down :: 'a ⇒ 'a set (\- [81] 80)
where \| x \equiv \{ y . y \leq x \}

abbreviation down-set :: 'a set ⇒ 'a set (\- [81] 80)
where \| X \equiv \{ y . \exists x \in X . y \leq x \}

abbreviation is-down-set :: 'a set ⇒ bool
where is-down-set X \equiv \forall x \in X . \forall y . y \leq x \rightarrow y \in X

abbreviation up :: 'a ⇒ 'a set (\+ [81] 80)
where \| x \equiv \{ y . x \leq y \}

abbreviation up-set :: 'a set ⇒ 'a set (\+ [81] 80)
where \| X \equiv \{ y . \exists x \in X . x \leq y \}

abbreviation is-up-set :: 'a set ⇒ bool
where is-up-set X \equiv \forall x \in X . \forall y . x \leq y \rightarrow y \in X

abbreviation is-principal-down :: 'a set ⇒ bool
where is-principal-down X \equiv \exists x . X = \| x

abbreviation is-principal-up :: 'a set ⇒ bool
where is-principal-up X \equiv \exists x . X = up x

A filter is a non-empty, downward directed, up-closed set.

definition filter :: 'a set ⇒ bool
where filter F \equiv (F \neq \{\}) \land (\forall x \in F . \forall y \in F . \exists z \in F . z \leq x \land z \leq y) \land
is-up-set F

abbreviation proper-filter :: 'a set ⇒ bool
where proper-filter F \equiv filter F \land F \neq UNIV

abbreviation ultra-filter :: 'a set ⇒ bool
where ultra-filter F \equiv proper-filter F \land (\forall G . proper-filter G \land F \subseteq G \rightarrow F = G)

end

context order
begin

lemma self-in-downset [simp]:
x \in down x
by simp

lemma self-in-upset [simp]:
x \in up x
by simp

38
lemma up-filter [simp]:
    filter (∧x)
  using filter-def order-lesseq-imp by auto

lemma up-set-up-set [simp]:
    is-up-set (∣X)
  using order.trans by fastforce

lemma up-injective:
    ∣x = ∣y ⟹ x = y
  using antisym by auto

lemma up-antitone:
    x ≤ y ⟷ ∣y ⊆ ∣x
  by auto

end

context order-bot
begin

lemma bot-in-downset [simp]:
    bot ∈ ↓x
  by simp

lemma down-bot [simp]:
    ↓bot = {bot}
  by (simp add: bot-unique)

lemma up-bot [simp]:
    ↑bot = UNIV
  by simp

  The following result is the ultrafilter lemma, generalised from [9, 10.17] to orders with a least element. Its proof uses Isabelle/HOL’s Zorn-Lemma, which requires closure under union of arbitrary (possibly empty) chains. Actually, the proof does not use any of the underlying order properties except bot-least.

lemma ultra-filter:
  assumes proper-filter F
  shows ∃G . ultra-filter G ∧ F ⊆ G

proof –
  let ?A = { G . (proper-filter G ∧ F ⊆ G) ∨ G = {} } 
  have ∀ C ∈ chains ?A . ∪ C ∈ ?A
  proof
    fix C :: 'a set set 
    let ?D = C − {{}}
    assume 1: C ∈ chains ?A
    hence 2: ∀ x ∈ ∪ ?D . ∃ H ∈ ?D . x ∈ H ∧ proper-filter H

  end
using `chainsD2` by `fastforce`

have 3: \( \bigcup \mathcal{D} = \bigcup \mathcal{C} \)

by `blast`

have \( \bigcup \mathcal{D} \in \mathcal{A} \)

proof (cases \( \mathcal{D} = \{\} \))

assume \( \mathcal{D} = \{\} \)

thus \( \text{thesis} \)

by `auto`

next

assume 4: \( \mathcal{D} \neq \{\} \)

then obtain \( G \) where \( G \in \mathcal{D} \)

by `auto`

hence 5: \( F \subseteq \bigcup \mathcal{D} \)

using 1 `chainsD2` by `blast`

have 6: `is-up-set (\bigcup \mathcal{D})`

proof

fix \( x \)

assume \( x \in \bigcup \mathcal{D} \)

then obtain \( H \) where \( x \in H \land H \in \mathcal{D} \land \text{filter } H \)

using 2 by `auto`

thus \( \forall y . \; x \leq y \rightarrow y \in \bigcup \mathcal{D} \)

using `filter-def UnionI` by `fastforce`

qed

have 7: \( \bigcup \mathcal{D} \neq \text{UNIV} \)

proof (rule `ccontr`

assume \( \neg (\bigcup \mathcal{D} \neq \text{UNIV} \) \)

then obtain \( H \) where \( \text{bot} \in H \land \text{proper-filter } H \)

using 2 by `blast`

thus \( \text{False} \)

by (meson `UNIV-I` `bot-least` `filter-def` `subsetI` `subset-antisym`)

qed

\{ \}

\fix \( x \; y \)

\assume \( x \in \bigcup \mathcal{D} \land y \in \bigcup \mathcal{D} \)

then obtain \( H \; I \) where 8: \( x \in H \land H \in \mathcal{D} \land \text{filter } H \land y \in I \land I \in \mathcal{D} \land \text{filter } I \)

using 2 by `metis`

have \( \exists z \in \bigcup \mathcal{D} . \; z \leq x \land z \leq y \)

proof (cases \( H \subseteq I \))

assume \( H \subseteq I \)

hence \( \exists z \in I . \; z \leq x \land z \leq y \)

using 8 by (metis `subsetCE` `filter-def`

thus \( \text{thesis} \)

using 8 by (metis `UnionI`)

next

assume \( \neg (H \subseteq I) \)

hence \( I \subseteq H \)

using 1 8 by (meson `DiffE` `chainsD`)

hence \( \exists z \in H . \; z \leq x \land z \leq y \)
using 8 by (metis subsetCE filter-def)
thus ?thesis
using 8 by (metis UnionI)
qed
}
thus ?thesis
using 4 5 6 7 filter-def by auto
qed
thus \( \bigcup C \in \mathcal{A} \) using 3 by simp
qed
hence \( \exists M \in \mathcal{A} \wedge (\forall X \in \mathcal{A} . M \subseteq X \longrightarrow X = M) \)
by (rule Zorn-Lemma)
then obtain M where 9: \( M \in \mathcal{A} \wedge (\forall X \in \mathcal{A} . M \subseteq X \longrightarrow X = M) \)
by auto
hence 10: \( M \neq \{\} \)
using assms filter-def by auto
{
fix G
assume 11: proper-filter G \( \wedge M \subseteq G \)
\hence F \subseteq G
using 9 10 by blast
\hence M = G
using 9 11 by auto
}
thus ?thesis
using 9 10 by blast
qed
end

context order-top
begin

lemma down-top [simp]:
\( \downarrow \text{top} = \text{UNIV} \)
by simp

lemma top-in-upset [simp]:
\( \text{top} \in \uparrow x \)
by simp

lemma up-top [simp]:
\( \uparrow \text{top} = \{\text{top}\} \)
by (simp add: top-unique)

lemma filter-top [simp]:
filter \{top\}
using filter-def top-unique by auto
The existence of proper filters and ultrafilters requires that the underlying order contains at least two elements.

context non-trivial-order
begin

lemma proper-filter-exists:
\exists F . proper-filter F
proof -
  from consistent obtain x y :: 'a where x \neq y
  by auto
  hence \uparrow x \neq UNIV \lor \uparrow y \neq UNIV
  using antisym by blast
  hence proper-filter (\uparrow x) \lor proper-filter (\uparrow y)
    by simp
  thus ?thesis
  by blast
qed

end

context non-trivial-order-bot
begin

lemma ultra-filter-exists:
\exists F . ultra-filter F
using ultra-filter proper-filter-exists by blast

end

context non-trivial-bounded-order
begin

lemma proper-filter-top:
  proper-filter \{top\}
  using bot-not-top filter-top by blast

lemma ultra-filter-top:
\exists G . ultra-filter G \land top \in G
using ultra-filter proper-filter-top by fastforce

end
4.2 Lattices

This section develops the lattice structure of filters based on a semilattice structure of the underlying order. The main results are that filters over a directed semilattice form a lattice with a greatest element and that filters over a bounded semilattice form a bounded lattice.

context semilattice-sup
begin

abbreviation prime-filter :: 'a set ⇒ bool
  where prime-filter F ≡ proper-filter F ∧ (∀ x y . x ⊔ y ∈ F → x ∈ F ∨ y ∈ F)

end

context semilattice-inf
begin

lemma filter-inf-closed:
  filter F =⇒ x ∈ F =⇒ y ∈ F =⇒ x ⊓ y ∈ F
  by (meson filter-def inf.bounded1)

lemma filter-univ:
  filter UNIV
  by (meson UNIV-I UNIV-not-empty filter-def inf.cobounded1 inf.cobounded2)

  The operation filter-sup is the join operation in the lattice of filters.

abbreviation filter-sup F G ≡ { z . ∃ x ∈ F . ∃ y ∈ G . x ⊓ y ≤ z }

lemma filter-sup:
  assumes filter F
    and filter G
  shows filter (filter-sup F G)
proof –
  have F ≠ {} ∧ G ≠ {} using assms filter-def by blast
  hence 1: filter-sup F G ≠ {} by blast
  have 2: ∀ x ∈ filter-sup F G . ∀ y ∈ filter-sup F G . ∃ z ∈ filter-sup F G . z ≤ x ∧ z ≤ y
  proof
    fix x
    assume x ∈ filter-sup F G
    then obtain t u where 3: t ∈ F ∧ u ∈ G ∧ t ⊓ u ≤ x
      by auto
    show ∀ y ∈ filter-sup F G . ∃ z ∈ filter-sup F G . z ≤ x ∧ z ≤ y
    proof
      fix y
      assume y ∈ filter-sup F G

43
then obtain \( v, w \) where
\[
4: v \in F \land w \in G \land v \cap w \leq y
\]
by auto

let \(?z = (t \cap v) \cap (u \cap w)\)

have 5: \(?z \leq x \land ?z \leq y\)
using 3 4 by (meson order.trans inf.cobounded1 inf.cobounded2 inf-mono)

have \(?z \in \text{filter-sup} \ F \ G\)
using assms 3 4 filter-inf-closed by blast
thus \( \exists z \in \text{filter-sup} \ F \ G . \ z \leq x \land z \leq y\)
using 5 by blast

qed

lemma filter-sup-left-upper-bound:
assumes filter G
shows \( F \subseteq \text{filter-sup} \ F \ G\)

proof
- from assms obtain \( y \) where \( y \in G\)
  using all-not-in-conv filter-def by auto
thus \(?\thesis\)
  using inf.cobounded1 by blast

qed

lemma filter-sup-symmetric:
\( \text{filter-sup} \ F \ G = \text{filter-sup} \ G \ F\)
using inf.commute by fastforce

lemma filter-sup-right-upper-bound:
\( \text{filter} \ F \Rightarrow \ G \subseteq \text{filter-sup} \ F \ G\)
using filter-sup-symmetric filter-sup-left-upper-bound by simp

lemma filter-sup-least-upper-bound:
assumes filter H
  and \( F \subseteq H\)
  and \( G \subseteq H\)
shows \( \text{filter-sup} \ F \ G \subseteq H\)

proof
fix \( x \)
assume \( x \in \text{filter-sup} \ F \ G\)
then obtain \( y, z \) where \( I: y \in F \land z \in G \land y \cap z \leq x\)
by auto
hence \( y \in H \land z \in H\)
using assms(2–3) by auto
hence \( y \cap z \in H\)
by (simp add: assms(1) filter-inf-closed)
thus $x \in H$
  using 1 assms(1) filter-def by auto
qed

lemma filter-sup-left-isotone:
  $G \subseteq H \Longrightarrow \text{filter-sup } G F \subseteq \text{filter-sup } H F$
  by blast

lemma filter-sup-right-isotone:
  $G \subseteq H \Longrightarrow \text{filter-sup } F G \subseteq \text{filter-sup } F H$
  by blast

lemma filter-sup-right-isotone-var:
  $\text{filter-sup } F (G \cap H) \subseteq \text{filter-sup } F H$
  by blast

lemma up-dist-inf:
  $\uparrow (x \cap y) = \text{filter-sup } (\uparrow x) (\uparrow y)$
proof
  show $\uparrow (x \cap y) \subseteq \text{filter-sup } (\uparrow x) (\uparrow y)$
    by blast
next
  show $\text{filter-sup } (\uparrow x) (\uparrow y) \subseteq \uparrow (x \cap y)$
proof
    fix $z$
    assume $z \in \text{filter-sup } (\uparrow x) (\uparrow y)$
    then obtain $u v$
    where $u \in \uparrow x \land v \in \uparrow y \land u \cap v \leq z$
      by auto
    hence $x \cap y \leq z$
      using order.trans inf-mono by blast
    thus $z \in \uparrow (x \cap y)$
      by blast
  qed
qed

The following result is part of [9, Exercise 2.23].

lemma filter-inf-filter [simp]:
  assumes $\text{filter } F$
  shows $\text{filter } (\exists z \in F . x \cap z = y)$
proof
  let $?G = \exists z \in F . x \cap z = y$
  have $F \neq \{\}$
    using assms filter-def by simp
  hence 1: $?G \neq \{\}$
    by blast
  have 2: is-up-set $?G$
    by simp
  { fix $y z$

assume \( y \in \mathcal{F} \land z \in \mathcal{G} \)
then obtain \( v \) \( w \) where \( v \in \mathcal{F} \land w \in \mathcal{F} \land x \cap v \leq y \land x \cap w \leq z \)
by auto
hence \( v \cap w \in \mathcal{F} \land x \cap (v \cap w) \leq y \cap z \)
by (meson assms filter-inf-closed order.trans inf.boundedI inf.cobounded1 inf.cobounded2)
hence \( \exists u \in \mathcal{G} . u \leq y \land u \leq z \)
by auto
hence \( \forall x \in \mathcal{G} . \forall y \in \mathcal{G} . \exists z \in \mathcal{G} . z \leq x \land z \leq y \)
by auto
thus \( \exists \text{thesis} \)
using 1 2 filter-def by presburger
qed

context directed-semilattice-inf
begin

Set intersection is the meet operation in the lattice of filters.

lemma filter-inf:
assumes filter F
and filter G
shows filter \((F \cap G)\)
proof (unfold filter-def, intro conjI)
from assms obtain \( x \) \( y \) where 1: \( x \in \mathcal{F} \land y \in \mathcal{G} \)
using all-not-in-conv filter-def by auto
from ub obtain \( z \) where \( x \leq z \land y \leq z \)
by auto
hence \( z \in \mathcal{F} \cap \mathcal{G} \)
using 1 by (meson assms Int-iff filter-def)
thus \( F \cap G \neq \{\} \)
by blast
next
show is-up-set \((F \cap G)\)
by (meson assms Int-iff filter-def)
next
show \( \forall x \in \mathcal{F} \cap \mathcal{G} . \forall y \in \mathcal{F} \cap \mathcal{G} . \exists z \in \mathcal{F} \cap \mathcal{G} . z \leq x \land z \leq y \)
by (metis assms Int-iff filter-inf-closed inf.cobounded2 inf.commute)
qed

end

We introduce the following type of filters to instantiate the lattice classes
and thereby inherit the results shown about lattices.

typedef (overloaded) 'a filter = \{ F::'a::order set . filter F \}
by (meson mem-Collect-eq up-filter)
lemma simp-filter [simp]:
  filter (Rep-filter x)
  using Rep-filter by simp

setup-lifting type-definition-filter

  The set of filters over a directed semilattice forms a lattice with a greatest
element.

instantiation filter :: (directed-semilattice-inf) bounded-lattice-top
begin

lift-definition top-filter :: 'a filter is UNIV
  by (simp add: filter-univ)

lift-definition sup-filter :: 'a filter ⇒ 'a filter ⇒ 'a filter is filter-sup
  by (simp add: filter-sup)

lift-definition inf-filter :: 'a filter ⇒ 'a filter ⇒ 'a filter is inter
  by (simp add: filter-inf)

lift-definition less-eq-filter :: 'a filter ⇒ 'a filter ⇒ bool is subset-eq .

lift-definition less-filter :: 'a filter ⇒ 'a filter ⇒ bool is subset .

instance
  apply intro-classes
  apply (simp add: less-eq-filter.rep-eq less-filter.rep-eq inf.less-le-not-le)
  apply (simp add: less-eq-filter.rep-eq)
  apply (simp add: less-eq-filter.rep-eq)
  apply (simp add: Rep-filter-inject less-eq-filter.rep-eq)
  apply (simp add: inf-filter.rep-eq less-eq-filter.rep-eq)
  apply (simp add: inf-filter.rep-eq less-eq-filter.rep-eq)
  apply (simp add: inf-filter.rep-eq less-eq-filter.rep-eq)
  apply (simp add: inf-filter.rep-eq less-eq-filter.rep-eq)
  apply (simp add: less-eq-filter.rep-eq filter-sup-left-upper-bound sup-filter.rep-eq)
  apply (simp add: less-eq-filter.rep-eq filter-sup-right-upper-bound sup-filter.rep-eq)
  apply (simp add: less-eq-filter.rep-eq filter-sup-least-upper-bound sup-filter.rep-eq)
  by (simp add: less-eq-filter.rep-eq top-filter.rep-eq)

end

context bounded-semilattice-inf-top
begin

abbreviation filter-complements F G ≡ filter F ∧ filter G ∧ filter-sup F G = UNIV ∧ F ∩ G = {top}

end

47
The set of filters over a bounded semilattice forms a bounded lattice.

**instantiation** filter :: (bounded-semilattice-inf-top) bounded-lattice
begin

**lift-definition** bot-filter :: 'a filter is \{top\}
  by simp

**instance**
  by intro-classes (simp add: less-eq-filter.rep-eq bot-filter.rep-eq)
end

c**ontext** lattice
begin

**lemma** up-dist-sup:
\[\uparrow(x \sqcup y) = \uparrow x \cap \uparrow y\]
  by auto
end

For convenience, the following function injects principal filters into the filter type. We cannot define it in the order class since the type filter requires the sort constraint order that is not available in the class. The result of the function is a filter by lemma up-filter.

**abbreviation** up-filter :: 'a::order ⇒ 'a filter
  where up-filter x ≡ Abs-filter (\uparrow x)

**lemma** up-filter-dist-inf:
  up-filter ((x::'a::lattice) ∩ y) = up-filter x ∪ up-filter y
  by (simp add: eq-onp-def sup-filter.abs-eq up-dist-inf)

**lemma** up-filter-dist-sup:
  up-filter ((x::'a::lattice) ⊔ y) = up-filter x ∩ up-filter y
  by (metis eq-onp-same-args less-eq-filter.abs-eq up-dist-sup up-filter)

**lemma** up-filter-injective:
  up-filter x = up-filter y \implies x = y
  by (metis Abs-filter-inject mem-Collect-eq up-filter up-injective)

**lemma** up-filter-antitone:
  x ≤ y \iff up-filter y ≤ up-filter x
  by (metis eq-onp-same-args less-eq-filter.abs-eq up-antitone up-filter)

The following definition applies a function to each element of a filter. The subsequent lemma gives conditions under which the result of this application is a filter.

**abbreviation** filter-map :: ('a::order ⇒ 'b::order) ⇒ 'a filter ⇒ 'b filter

48
where \( \text{filter-map } f \ F \equiv \text{Abs-filter} (f \cdot \text{Rep-filter } F) \)

**Lemma filter-map-filter:**
- **Assumes** \( \text{mono } f \) and \( \forall x y . f x \leq y \rightarrow (\exists z . x \leq z \land y = f z) \)
- **Shows** \( \text{filter} (f \cdot \text{Rep-filter } F) \)

**Proof:** (unfold filter-def, intro conjI)
- show \( f \cdot \text{Rep-filter } F \neq \{\} \)
  - by (metis empty-is-image filter-def simp-filter)

**Next**
- show \( \forall x \in f \cdot \text{Rep-filter } F \cdot \forall y \in f \cdot \text{Rep-filter } F \cdot \exists z \in f \cdot \text{Rep-filter } F . z \leq x \land z \leq y \)
  - proof (intro ballI)
    - fix \( x y \)
    - assume \( x \in f \cdot \text{Rep-filter } F \) and \( y \in f \cdot \text{Rep-filter } F \)
    - then obtain \( u v \) where \( 1: x = f u \land u \in \text{Rep-filter } F \land y = f v \land v \in \text{Rep-filter } F \)
  - by auto
    - then obtain \( w \) where \( w \leq u \land w \leq v \land w \in \text{Rep-filter } F \)
      - by (meson filter-def simp-filter)
    - thus \( \exists z \in f \cdot \text{Rep-filter } F . z \leq x \land z \leq y \)
      - using \( 1 \) assms \( 1 \) mono-def rev-image-eqI by blast
  - qed

**Next**
- show \( \text{is-up-set} (f \cdot \text{Rep-filter } F) \)
  - proof
    - fix \( x \)
    - assume \( x \in f \cdot \text{Rep-filter } F \)
    - then obtain \( u \) where \( 1: x = f u \land u \in \text{Rep-filter } F \)
      - by auto
    - show \( \forall y . x \leq y \rightarrow y \in f \cdot \text{Rep-filter } F \)
      - proof (rule allI, rule impI)
        - fix \( y \)
        - assume \( x \leq y \)
        - hence \( f u \leq y \)
          - using \( 1 \) by simp
        - then obtain \( z \) where \( u \leq z \land y = f z \)
          - using assms \( 2 \) by auto
        - thus \( y \in f \cdot \text{Rep-filter } F \)
          - using \( 1 \) by (meson image-iff filter-def simp-filter)
    - qed
    - qed

**4.3 Distributive Lattices**

In this section we additionally assume that the underlying order forms a distributive lattice. Then filters form a bounded distributive lattice if the underlying order has a greatest element. Moreover ultrafilters are prime
filters. We also prove a lemma of Grätzer and Schmidt about principal filters.

case context distrib-lattice begin

lemma filter-sup-left-dist-inf:
  assumes filter F
  and filter G
  and filter H
  shows filter-sup F (G ∩ H) = filter-sup F G ∩ filter-sup F H

proof
  show filter-sup F (G ∩ H) ⊆ filter-sup F G ∩ filter-sup F H
  using filter-sup-right-isotone-var by blast
  next
  show filter-sup F G ∩ filter-sup F H ⊆ filter-sup F (G ∩ H)
  proof
    fix x assume x ∈ filter-sup F G ∩ filter-sup F H
    then obtain t u v w
      where 1: t ∈ F ∧ u ∈ G ∧ v ∈ F ∧ w ∈ H ∧ t ⊓ u ≤ x ∧ v ⊓ w ≤ x
        by auto
    let ?y = t ⊓ v
    let ?z = u ⊔ w
    have 2: ?y ∈ F
      using 1 by (simp add: assms(1) filter-inf-closed)
    have 3: ?z ∈ G ∩ H
      using 1 by (meson assms(2-3) Int-iff filter-def sup-ge1 sup-ge2)
    have ?y ⊓ ?z = (t ⊓ v ∩ u) ∪ (t ⊓ v ⊔ w)
      by (simp add: inf-sup-distrib1)
    also have ... ≤ (t ⊓ u) ∪ (v ⊓ w)
      by (metis inf.cobounded1 inf.cobounded2 inf.left-idem inf.mono sup.mono)
    also have ... ≤ x
      by (simp add: le-supI)
    finally show x ∈ filter-sup F (G ∩ H)
      using 2 3 by blast
  qed
qed

lemma filter-inf-principal-rep:
  F ∩ G = ↑z =⇒ (∃ x∈F . ∃ y∈G . z = x ⊔ y)
  by force

lemma filter-sup-principal-rep:
  assumes filter F
    and filter G
  and filter-sup F G = ↑z
  shows ∃ x∈F . ∃ y∈G . z = x ⊓ y

proof
  from assms(3) obtain x y where 1: x∈F ∧ y∈G ∧ x ⊓ y ≤ z

using order-refl by blast

hence 2: \( x \sqcup z \in F \land y \sqcup z \in G \)
  by (meson assms(1-2) sup-ge1 filter-def)

have \((x \sqcup z) \cap (y \sqcup z) = z\)
  using 1 sup-absorb2 sup-inf-distrib2 by fastforce

thus \(?thesis
  using 2 by force

qed

lemma inf-sup-principal-aux:
  assumes filter F
  and filter G
  and is-principal-up (filter-sup F G)
  and is-principal-up \((F \cap G)\)
  shows is-principal-up F

proof –
  from assms(3-4) obtain x y where 1: filter-sup F G = \( \uparrow x \land F \cap G = \uparrow y \)
  by blast

  from filter-inf-principal-rep obtain t u where 2: \( t \in F \sqcup u \in G \land y = t \sqcup u \)
      using 1 by meson

  from filter-sup-principal-rep obtain v w where 3: \( v \in F \land w \in G \land x = v \sqcap w \)
    using 1 by (meson assms(1-2))

  have \( t \in filter-sup F G \land u \in filter-sup F G \)
      using 2 inf.cobounded1 inf.cobounded2 by blast

  hence x \( \leq t \land x \leq u \)
      using 1 by blast

  hence 4: \( (t \sqcap v) \sqcap (u \sqcap w) = x \)
      using 3 by (simp add: inf.absorb2 inf.assoc inf.left-commute)

  have \( (t \sqcap v) \sqcup (u \sqcap w) \in F \land (t \sqcap v) \sqcup (u \sqcap w) \in G \)
      using 2 3 by (metis (no-types, lifting) assms(1-2) filter-inf-closed inf.cobounded1 inf.cobounded2 filter-def)

  hence y \( \leq (t \sqcap v) \sqcup (u \sqcap w) \)
      using 1 Int-iff by blast

  hence 5: \( (t \sqcap v) \sqcup (u \sqcap w) = y \)
      using 2 by (simp add: antisym inf.cobounded1I)

  have \( F = \uparrow(t \sqcap v) \)

proof

  show \( F \subseteq \univ \)

proof

  fix z

  assume 6: \( z \in F \)

  hence z \( \in filter-sup F G \)
      using 2 inf.cobounded1 by blast

  hence x \( \leq z \)
      using 1 by simp

  hence 7: \( (t \sqcup v \sqcap z) \sqcap (u \sqcap w) = x \)
      using 4 by (metis inf.absorb1 inf.assoc inf.commute)

  have \( z \sqcup u \in F \land z \sqcup u \in G \land z \sqcup w \in F \land z \sqcup w \in G \)
      using 2 3 6 by (meson assms(1-2) filter-def sup-ge1 sup-ge2)
hence \[ y \leq (z \sqcup u) \cap (z \sqcup w) \]
using 1 Int-iff filter-inf-closed by auto
hence 8: \[(t \cap v \cap z) \cup (u \cap w) = y \]
using 5 by (metis inf.absorb1 sup.commute sup-inf-distrib2)
have \[ t \cap v \cap z = t \cap v \]
using 4 5 7 8 relative-equality by blast
thus \[ z \in \uparrow (t \cap v) \]
by (simp add: inf.orderI)
qed
next
show \( \uparrow (t \cap v) \subseteq F \)
proof
fix \[ z \]
have 9: \[ t \cap v \in F \]
using 2 3 by (simp add: assms(1) filter-inf-closed)
assume \[ z \in \uparrow (t \cap v) \]
hence \[ t \cap v \leq z \] by simp
thus \[ z \in F \]
using assms(1) 9 filter-def by auto
qed
qed
thus \( ?\text{thesis} \)
by blast
qed

The following result is [18, Lemma II]. If both join and meet of two filters are principal filters, both filters are principal filters.

**Lemma inf-sup-principal:**

assumes filter \( F \)
and filter \( G \)
and is-principal-up (filter-sup \( F \) \( G \))
and is-principal-up (\( F \cap G \))
shows is-principal-up \( F \land G \)

**Proof**

have filter \( G \land F \land \text{is-principal-up (filter-sup G F)} \land \text{is-principal-up (G} \cap \text{G}) \)
by (simp add: assms Int-commute filter-sup-symmetric)
thus \( ?\text{thesis} \)
using assms(3) inf-sup-principal-aux by blast
qed

**Lemma filter-sup-absorb-inf:** filter \( F \Rightarrow filter G \Rightarrow filter-sup (F \cap G) \) \( G \Rightarrow G \)
by (simp add: filter-inf filter-sup-least-upper-bound filter-sup-left-upper-bound filter-sup-symmetric subset-antisym)

**Lemma filter-inf-absorb-sup:** filter \( F \Rightarrow filter G \Rightarrow filter-sup F \cap G \) \( G \Rightarrow G \)
apply (rule subset-antisym)
apply simp
by (simp add: filter-sup-right-upper-bound)
lemma filter-inf-right-dist-sup:
  assumes filter F
  and filter G
  and filter H
  shows filter-sup F G ∩ H = filter-sup (F ∩ H) (G ∩ H)
proof (complete proof)
  have filter-sup (F ∩ H) (G ∩ H) = filter-sup (F ∩ H) G ∩ filter-sup (F ∩ H) H
    by (simp add: assms filter-sup-left-dist-inf filter-inf)
  also have ... = filter-sup (F ∩ H) G ∩ H
    using assms(1,3) filter-sup-absorb-inf by simp
  also have ... = filter-sup F G ∩ filter-sup G H ∩ H
    using assms filter-sup-left-dist-inf filter-sup-symmetric by simp
  also have ... = filter-sup F G ∩ H
    by (simp add: assms(2-3) filter-inf-absorb-sup semilattice-inf-class.inf-assoc)
  finally have ... ∈ F
    by simp
qed

The following result generalises [9, 10.11] to distributive lattices as remarked after that section.

lemma ultra-filter-prime:
  assumes ultra-filter F
  shows prime-filter F
proof (complete proof)
  { fix x y
    assume 1: x ⊔ y ∈ F ∧ x ∉ F
    let ?G = ⇑{ z . ∃ w ∈ F . x ⊓ w = z }
    have 2: filter ?G
      using assms filter-inf-filter by simp
    have x ∈ ?G
      using 1 by auto
    hence 3: F ≠ ?G
      using 1 by auto
    have F ⊆ ?G
      using inf-le2 order-trans by blast
    hence ?G = UNIV
      using 2 3 assms by blast
    then obtain z where 4: z ∈ F ∧ x ∩ z ≤ y
      by blast
    hence y ∩ z = (x ⊔ y) ∩ z
      by (simp add: inf-sup-distrib2 sup-absorb2)
    also have ... ∈ F
      using 1 4 assms filter-inf-closed by auto
    finally have y ∈ F
      using assms by (simp add: filter-def)
  }

53
thus \( \text{thesis} \)
    using \text{assms} by \text{blast}
qed

end

context distrib-lattice-bot
begin

lemma prime-filter:
    \text{proper-filter} F \implies \exists G . \text{prime-filter} G \land F \subseteq G
    \text{by} (\text{metis ultra-filter ultra-filter-prime})

end

context distrib-lattice-top
begin

lemma complemented-filter-inf-principal:
    \text{assumes} \text{filter-complements} F G
    \text{shows} \text{is-principal-up} (F \cap \uparrow x)
proof –
    have 1: filter F \land filter G
        \text{by} (\text{simp add: assms})
    hence 2: filter \( (F \cap \uparrow x) \land filter \ (G \cap \uparrow x) \)
        \text{by} (\text{simp add: filter-inf})
    have \( (F \cap \uparrow x) \cap (G \cap \uparrow x) = \{ \text{top} \} \)
        \text{using assms Int-assoc Int-insert-left-if1 inf-bot-left inf-sup-aci (3) top-in-upset inf.idem by auto}
    hence 3: is-principal-up \( ((F \cap \uparrow x) \cap (G \cap \uparrow x)) \)
        \text{using up-top by blast}
    have filter-sup \( (F \cap \uparrow x) \cap (G \cap \uparrow x) = filter-sup F \ G \cap \uparrow x \)
        \text{using 1 filter-inf-right-dist-sup up-filter by auto}
    also have \( \ldots = \uparrow x \)
        \text{by (simp add: assms)}
    finally have is-principal-up \( (filter-sup \ (F \cap \uparrow x) \ (G \cap \uparrow x)) \)
        \text{by auto}
    thus \( \text{thesis} \)
        \text{using 1 2 3 inf-sup-principal-aux by blast}
qed

end

The set of filters over a distributive lattice with a greatest element forms a bounded distributive lattice.

instantiation filter :: (distrib-lattice-top) bounded-distrib-lattice
begin
instance

\textend{document}
proof
fix \( x \ y \ z \) :: 'a filter
have \( \text{Rep}-\text{filter} (x \sqcup (y \sqcap z)) = \text{filter}-\text{sup} (\text{Rep}-\text{filter} x) (\text{Rep}-\text{filter} (y \sqcap z)) \)
  by (simp add: sup-filter.rep-eq)
also have \( \ldots = \text{filter}-\text{sup} (\text{Rep}-\text{filter} x) (\text{Rep}-\text{filter} y \sqcap \text{Rep}-\text{filter} z) \)
  by (simp add: inf-filter.rep-eq)
also have \( \ldots = \text{filter}-\text{sup} (\text{Rep}-\text{filter} x) (\text{Rep}-\text{filter} y) \cap \text{filter}-\text{sup} (\text{Rep}-\text{filter} x) (\text{Rep}-\text{filter} z) \)
  by (simp add: filter-sup-left-dist-inf)
also have \( \ldots = \text{Rep}-\text{filter} (x \sqcup y) \cap \text{Rep}-\text{filter} (x \sqcup z) \)
  by (simp add: sup-filter.rep-eq)
also have \( \ldots = \text{Rep}-\text{filter} ((x \sqcup y) \cap (x \sqcup z)) \)
  by (simp add: inf-filter.rep-eq)
finally show \( x \sqcup (y \sqcap z) = (x \sqcup y) \cap (x \sqcup z) \)
  by (simp add: Rep-filter-inject)
qed

end

end

5 Stone Construction

This theory proves the uniqueness theorem for the triple representation of Stone algebras and the construction theorem of Stone algebras [7, 21]. Every Stone algebra \( S \) has an associated triple consisting of

* the set of regular elements \( B(S) \) of \( S \),

* the set of dense elements \( D(S) \) of \( S \), and

* the structure map \( \varphi(S) : B(S) \to F(D(S)) \) defined by \( \varphi(x) = \uparrow x \cap D(S). \)

Here \( F(X) \) is the set of filters of a partially ordered set \( X \). We first show that

* \( B(S) \) is a Boolean algebra,

* \( D(S) \) is a distributive lattice with a greatest element, whence \( F(D(S)) \) is a bounded distributive lattice, and

* \( \varphi(S) \) is a bounded lattice homomorphism.

Next, from a triple \( T = (B, D, \varphi) \) such that \( B \) is a Boolean algebra, \( D \) is a distributive lattice with a greatest element and \( \varphi : B \to F(D) \) is a bounded lattice homomorphism, we construct a Stone algebra \( S(T) \). The elements of \( S(T) \) are pairs taken from \( B \times F(D) \) following the construction of [21]. We need to represent \( S(T) \) as a type to be able to instantiate the
Stone algebra class. Because the pairs must satisfy a condition depending on \( \varphi \), this would require dependent types. Since Isabelle/HOL does not have dependent types, we use a function lifting instead. The lifted pairs form a Stone algebra.

Next, we specialise the construction to start with the triple associated with a Stone algebra \( S \), that is, we construct \( S(B(S), D(S), \varphi(S)) \). In this case, we can instantiate the lifted pairs to obtain a type of pairs (that no longer implements a dependent type). To achieve this, we construct an embedding of the type of pairs into the lifted pairs, so that we inherit the Stone algebra axioms (using a technique of universal algebra that works for universally quantified equations and equational implications).

Next, we show that the Stone algebras \( S(B(S), D(S), \varphi(S)) \) and \( S \) are isomorphic. We give explicit mappings in both directions. This implies the uniqueness theorem for the triple representation of Stone algebras.

Finally, we show that the triples \( (B(S(T)), D(S(T)), \varphi(S(T))) \) and \( T \) are isomorphic. This requires an isomorphism of the Boolean algebras \( B \) and \( B(S(T)) \), an isomorphism of the distributive lattices \( D \) and \( D(S(T)) \), and a proof that they preserve the structure maps. We give explicit mappings of the Boolean algebra isomorphism and the distributive lattice isomorphism in both directions. This implies the construction theorem of Stone algebras. Because \( S(T) \) is implemented by lifted pairs, so are \( B(S(T)) \) and \( D(S(T)) \); we therefore also lift \( B \) and \( D \) to establish the isomorphisms.

theory Stone-Construction

imports P-Algebras Filters

begin

5.1 Triples

This section gives definitions of lattice homomorphisms and isomorphisms and basic properties. It concludes with a locale that represents triples as discussed above.

class sup-inf-top-bot-uminus = sup + inf + top + bot + uminus
class sup-inf-top-bot-uminus-ord = sup-inf-top-bot-uminus + ord

context p-algebra

begin

subclass sup-inf-top-bot-uminus-ord .

end

abbreviation sup-homomorphism :: (a::sup \Rightarrow b::sup) \Rightarrow bool
where sup-homomorphism \( f \equiv \forall x \ y \ (f x \sqcup y) = f x \sqcup f y \)
abbreviation inf-homomorphism :: ('a::inf ⇒ 'b::inf) ⇒ bool
  where inf-homomorphism f ≡ ∀x y. f (x ∩ y) = f x ∩ f y

abbreviation sup-inf-homomorphism :: ('a::{sup,inf} ⇒ 'b::{sup}) ⇒ bool
  where sup-inf-homomorphism f ≡ sup-homomorphism f ∧ inf-homomorphism f

abbreviation sup-inf-top-homomorphism :: ('a::{sup,inf,bot} ⇒ 'b::{sup,inf,top}) ⇒ bool
  where sup-inf-top-homomorphism f ≡ sup-inf-homomorphism f ∧ f top = top

abbreviation sup-inf-top-bot-homomorphism :: ('a::{sup,inf,bot} ⇒ 'b::{sup,inf,bot,bot}) ⇒ bool
  where sup-inf-top-bot-homomorphism f ≡ sup-inf-top-homomorphism f ∧ f bot = bot

abbreviation bounded-lattice-homomorphism :: ('a::bounded-lattice ⇒ 'b::bounded-lattice) ⇒ bool
  where bounded-lattice-homomorphism f ≡ sup-inf-top-bot-homomorphism f

abbreviation sup-inf-top-bot-uminus-homomorphism :: ('a::{sup-inf,bot} ⇒ 'b::sup-inf-bot-uminus) ⇒ bool
  where sup-inf-top-bot-uminus-homomorphism f ≡ sup-inf-top-bot-homomorphism f ∧ (∀x. f (−x) = −f x)

abbreviation sup-inf-top-bot-uminus-ord-homomorphism :: ('a::{sup-inf-bot-uminus-ord} ⇒ 'b::{sup-inf}) ⇒ bool
  where sup-inf-top-bot-uminus-ord-homomorphism f ≡ sup-inf-top-bot-uminus-homomorphism f ∧ (∀x y. x ≤ y ⇒ f x ≤ f y)

abbreviation sup-inf-top-isomorphism :: ('a::{sup,inf,top} ⇒ 'b::{sup,inf,top}) ⇒ bool
  where sup-inf-top-isomorphism f ≡ sup-inf-top-homomorphism f ∧ bij f

abbreviation bounded-lattice-top-isomorphism :: ('a::bounded-lattice-top ⇒ 'b::bounded-lattice-top) ⇒ bool
  where bounded-lattice-top-isomorphism f ≡ sup-inf-top-isomorphism f

abbreviation sup-inf-top-bot-uminus-isomorphism :: ('a::sup-inf-top-bot-uminus ⇒ 'b::sup-inf-top-bot-uminus) ⇒ bool
  where sup-inf-top-bot-uminus-isomorphism f ≡ sup-inf-top-bot-uminus-homomorphism f ∧ bij f

abbreviation stone-algebra-isomorphism :: ('a::stone-algebra ⇒ 'b::stone-algebra) ⇒ bool
  where stone-algebra-isomorphism f ≡ sup-inf-top-bot-uminus-isomorphism f

abbreviation boolean-algebra-isomorphism :: ('a::boolean-algebra ⇒ 'b::boolean-algebra) ⇒ bool
  where boolean-algebra-isomorphism f ≡ sup-inf-top-bot-uminus-isomorphism f
lemma sup-homomorphism-mono:
  sup-homomorphism (f::'a::semilattice-sup ⇒ 'b::semilattice-sup) ⇒ mono f
by (metis le-iff-sup monoI)

lemma sup-isomorphism-ord-isomorphism:
  assumes sup-homomorphism (f::'a::semilattice-sup ⇒ 'b::semilattice-sup)
  and bij f
  shows x ≤ y ⟷ f x ≤ f y
proof
  assume x ≤ y
  thus f x ≤ f y
    by (metis assms(1) le-iff-sup)
next
  assume f x ≤ f y
  hence f (x ⊔ y) = f y
    by (simp add: assms(1) le-iff-sup)
  hence x ⊔ y = y
    by (metis injD bij-is-inj assms)
  thus x ≤ y
    by (simp add: le-iff-sup)
qed

A triple consists of a Boolean algebra, a distributive lattice with a greatest element, and a structure map. The Boolean algebra and the distributive lattice are represented as HOL types. Because both occur in the type of the structure map, the triple is determined simply by the structure map and its HOL type. The structure map needs to be a bounded lattice homomorphism.

locale triple =
  fixes phi :: 'a::boolean-algebra ⇒ 'b::distrib-lattice-top filter
  assumes hom: bounded-lattice-homomorphism phi

5.2 The Triple of a Stone Algebra
In this section we construct the triple associated to a Stone algebra.

5.2.1 Regular Elements
The regular elements of a Stone algebra form a Boolean subalgebra.

typedef (overloaded) 'a regular = regular-elements::'a::stone-algebra set
  by auto

lemma simp-regular [simp]:
  ∃ y. Rep-regular x = − y
using Rep-regular by simp

setup-lifting type-definition-regular
instantiation regular :: (stone-algebra) boolean-algebra

begin

lift-definition sup-regular :: 'a regular ⇒ 'a regular ⇒ 'a regular is sup
by (meson regular-in-p-image-iff regular-closed-sup)

lift-definition inf-regular :: 'a regular ⇒ 'a regular ⇒ 'a regular is inf
by (meson regular-in-p-image-iff regular-closed-inf)

lift-definition minus-regular :: 'a regular ⇒ 'a regular ⇒ 'a regular is λx y . x \cap -y
by (meson regular-in-p-image-iff regular-closed-inf)

lift-definition uminus-regular :: 'a regular ⇒ 'a regular is uminus
by auto

lift-definition bot-regular :: 'a regular is bot
by (meson regular-in-p-image-iff regular-closed-bot)

lift-definition top-regular :: 'a regular is top
by (meson regular-in-p-image-iff regular-closed-top)

lift-definition less-eq-regular :: 'a regular ⇒ 'a regular ⇒ bool is less-eq.

instance
apply intro-clauses
apply (simp add: less-eq-regular.rep-eq less-regular.rep-eq inf.le-not-le)
apply (simp add: less-eq-regular.rep-eq)
apply (simp add: less-eq-regular.rep-eq)
apply (simp add: Rep-regular-inject less-eq-regular.rep-eq)
apply (simp add: inf-eq-regular.rep-eq less-eq-regular.rep-eq)
apply (simp add: inf-eq-regular.rep-eq less-eq-regular.rep-eq)
apply (simp add: inf-eq-regular.rep-eq less-eq-regular.rep-eq)
apply (simp add: sup-regular.rep-eq less-eq-regular.rep-eq)
apply (simp add: sup-regular.rep-eq less-eq-regular.rep-eq)
apply (simp add: sup-regular.rep-eq less-eq-regular.rep-eq)
apply (simp add: top-regular.rep-eq less-eq-regular.rep-eq)
apply (metis (mono-tags) Rep-regular-inject inf-regular.rep-eq sup-inf-distrib1
sup-regular.rep-eq)
apply (metis (mono-tags) Rep-regular-inverse bot-regular.abs-eq
inf-regular.rep-eq inf-p uminus-regular.rep-eq)
apply (metis (mono-tags) top-regular.abs-eq Rep-regular-inverse simp-regular
stone sup-regular.rep-eq uminus-regular.rep-eq)
by (metis (mono-tags) Rep-regular-inject inf-regular.rep-eq minus-regular.rep-eq
uminus-regular.rep-eq)
instantiation  \textit{regular} :: (\textit{non-trivial-stone-algebra}) \textit{non-trivial-boolean-algebra}
begin

instance
proof (intro-classes, rule ccontr)
assume \( \neg(\exists \, x \, y::'a \, \text{regular} \cdot x \neq y) \)
hence (bot::'a regular) = top
  by simp
hence (bot::'a) = top
  by (metis bot-regular.rep-eq top-regular.rep-eq)
thus False
  by (simp add: bot-not-top)
qed

end

5.2.2 Dense Elements
The dense elements of a Stone algebra form a distributive lattice with a greatest element.
typedef (overloaded) 'a dense = dense-elements::'a::stone-algebra set
  using dense-closed-top by blast

lemma simp-dense [simp]:
  \text{Rep-dense} \ x = \text{bot}
  using Rep-dense by simp

setup-lifting type-definition-dense

instantiation dense :: (stone-algebra) distrib-lattice-top
begin

lift-definition sup-dense :: 'a dense \Rightarrow 'a dense \Rightarrow 'a dense is sup
  by simp

lift-definition inf-dense :: 'a dense \Rightarrow 'a dense \Rightarrow 'a dense is inf
  by simp

lift-definition top-dense :: 'a dense is top
  by simp

lift-definition less-eq-dense :: 'a dense \Rightarrow 'a dense \Rightarrow bool is less-eq .

lift-definition less-dense :: 'a dense \Rightarrow 'a dense \Rightarrow bool is less .

instance
apply intro-classes
apply (simp add: less-eq-dense.rep-eq less-dense.rep-eq inf.less-le-not-le)
apply (simp add: less-eq-dense.rep-eq)
apply (simp add: less-eq-dense.rep-eq)
apply (simp add: Rep-dense-inject less-eq-dense.rep-eq)
apply (simp add: inf-dense.rep-eq less-eq-dense.rep-eq)
apply (simp add: inf-dense.rep-eq less-eq-dense.rep-eq)
apply (simp add: sup-dense.rep-eq less-eq-dense.rep-eq)
apply (simp add: sup-dense.rep-eq less-eq-dense.rep-eq)
apply (simp add: sup-dense.rep-eq less-eq-dense.rep-eq)
apply (simp add: top-dense.rep-eq less-eq-dense.rep-eq)
by (metis (mono-tags, lifting) Rep-dense-inject sup-inf-distrib1 inf-dense.rep-eq sup-dense.rep-eq)
end

lemma up-filter-dense-antitone-dense:
dense (x ⊔¬ x ⊔ y) ∧ dense (x ⊔¬ x ⊔ y ⊔ z)
by simp

lemma up-filter-dense-antitone:
  up-filter (Abs-dense (x ⊔¬ x ⊔ y ⊔ z)) ≤ up-filter (Abs-dense (x ⊔¬ x ⊔ y))
by (unfold up-filter-antitone[THEN sym]) (simp add: Abs-dense-inverse less-eq-dense.rep-eq)

The filters of dense elements of a Stone algebra form a bounded distributive lattice.

type-synonym 'a dense-filter = 'a dense filter
typedef (overloaded) 'a dense-filter-type = { x::'a dense-filter . True }
  using filter-top by blast
setup-lifting type-definition-dense-filter-type

instantiation dense-filter-type :: (stone-algebra) bounded-distrib-lattice
begin

lift-definition sup-dense-filter-type :: 'a dense-filter-type ⇒ 'a dense-filter-type ⇒ 'a dense-filter-type is sup
  .

lift-definition inf-dense-filter-type :: 'a dense-filter-type ⇒ 'a dense-filter-type ⇒ 'a dense-filter-type is inf
  .

lift-definition bot-dense-filter-type :: 'a dense-filter-type is bot ..

lift-definition top-dense-filter-type :: 'a dense-filter-type is top ..

lift-definition less-eq-dense-filter-type :: 'a dense-filter-type ⇒ 'a dense-filter-type ⇒ 'a
dense-filter-type \Rightarrow \text{bool} \text{ is less-eq}.

\textbf{lift-definition} less-dense-filter-type :: 'a dense-filter-type \Rightarrow 'a dense-filter-type
\Rightarrow \text{bool} \text{ is less}.

\textbf{instance}
\textbf{apply} intro-classes
\textbf{apply} (simp add: less-eq-dense-filter-type.rep-eq less-dense-filter-type.rep-eq
inf.le-not-le)
\textbf{apply} (simp add: less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: less-eq-dense-filter-type.rep-eq
inf.order-lesseq-imp)
\textbf{apply} (simp add: Rep-dense-filter-type-inject less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: inf-dense-filter-type.rep-eq less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: inf-dense-filter-type.rep-eq
less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: less-eq-dense-filter-type.rep-eq
sup-dense-filter-type.rep-eq)
\textbf{apply} (simp add: less-eq-dense-filter-type.rep-eq
sup-dense-filter-type.rep-eq)
\textbf{apply} (simp add: less-eq-dense-filter-type.rep-eq
sup-dense-filter-type.rep-eq)
\textbf{apply} (simp add: less-eq-dense-filter-type.rep-eq
bot-dense-filter-type.rep-eq)
\textbf{apply} (simp add: sup-dense-filter-type.rep-eq
less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: sup-dense-filter-type.rep-eq
less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: sup-dense-filter-type.rep-eq
less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: sup-dense-filter-type.rep-eq
less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: top-dense-filter-type.rep-eq
less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: top-dense-filter-type.rep-eq
less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: top-dense-filter-type.rep-eq
less-eq-dense-filter-type.rep-eq)
\textbf{apply} (simp add: less-eq-dense-filter-type.rep-eq
sup-dense-filter-type.rep-eq)
\textbf{apply} (simp add: inf-dense-filter-type.rep-eq
sup-dense-filter-type.rep-eq)
\textbf{by} (metis (mono-tags, lifting) Rep-dense-filter-type-inject sup-inf-distrib1
inf-dense-filter-type.rep-eq sup-dense-filter-type.rep-eq)

end

5.2.3 The Structure Map

The structure map of a Stone algebra is a bounded lattice homomorphism. It maps a regular element \(x\) to the set of all dense elements above \(-x\). This set is a filter.

\textbf{abbreviation} stone-phi-set :: 'a::stone-algebra regular \Rightarrow 'a dense set
\textbf{where} stone-phi-set \(x\) \(\equiv\) \{ \(y\) . \(-\text{Rep-regular} x \leq \text{Rep-dense} y\} \}

\textbf{lemma} stone-phi-set-filter:
\textbf{filter} (stone-phi-set \(x\))
\textbf{apply} (unfold filter-def, intro conjI)
\textbf{apply} (metis Collect-empty-eq top-dense.rep-eq top-greatest)
\textbf{apply} (metis inf-dense.rep-eq inf-le2 le-inf-iff mem-Collect-eq)
\textbf{using} order-trans less-eq-dense.rep-eq by blast

\textbf{definition} stone-phi :: 'a::stone-algebra regular \Rightarrow 'a dense-filter
\textbf{where} stone-phi \(x\) = Abs-filter (stone-phi-set \(x\))

To show that we obtain a triple, we only need to prove that \(stone-phi\) is a bounded lattice homomorphism. The Boolean algebra and the distributive lattice requirements are taken care of by the type system.

\textbf{interpretation} stone-phi: triple stone-phi
\textbf{proof} (unfold-locales, intro conjI)
have 1: Rep-regular (Abs-regular bot) = bot 
  by (metis bot-regular.rep-eq bot-regular-def)
show stone-phi bot = bot 
  by (meta bot-regular.rep-eq)
apply (unfold stone-phi-def bot-regular-def p-bot bot-filter-def)
  by (metis (mono-tags, lifting) Collect-cong Rep-dense-inject order-refl
  singleton-conv top.extremum-uniqueI top-dense.rep-eq)
next 
  show stone-phi top = top 
    by (meta Collect-cong stone-phi-def UNIV-I bot.extremum dense-closed-top
  top-empty-eq top-filter.abs-eq top-regular.rep-eq top-set-def)
next 
  show ∀ x y::'a regular . stone-phi (x ⊔ y) = stone-phi x ⊔ stone-phi y
  proof (intro allI)
    fix x y :: 'a regular
    have stone-phi-set (x ⊔ y) = filter-sup (stone-phi-set x) (stone-phi-set y)
      proof (rule set-eqI, rule iffI)
        fix z
        assume 2: z ∈ stone-phi-set (x ⊔ y)
        let ?t = ~ Rep-regular x ⊔ Rep-dense z
        let ?u = ~ Rep-regular y ⊔ Rep-dense z
        let ?v = Abs-dense ?t
        let ?w = Abs-dense ?u
        have 3: ?v ∈ stone-phi-set x ∧ ?w ∈ stone-phi-set y
          by (simp add: Abs-dense-inverse)
        have ?v ▽ ?w = Abs-dense (?t ▽ ?u)
          by (simp add: eq-onp-def inf-dense)
        also have ... = Abs-dense (− (x ⊔ y) ⊔ Rep-dense z)
          by (simp add: distrib(1) sup-commute sup-regular.rep-eq)
        also have ... = Abs-dense (Rep-dense z)
          using 2 by (simp add: le_iff-sup)
        also have ... = z
          by (simp add: Rep-dense-inverse)
        finally show z ∈ filter-sup (stone-phi-set x) (stone-phi-set y)
          using 3 mem-Collect-eq order-refl by fastforce
next 
  fix z
  assume z ∈ filter-sup (stone-phi-set x) (stone-phi-set y)
  then obtain v w where 4: v ∈ stone-phi-set x ∧ w ∈ stone-phi-set y ∧ v ▽ w ≤ z
    by auto
  have − Rep-regular (x ⊔ y) = Rep-regular (− (x ⊔ y))
    by (metis uminus-regular.rep-eq)
  also have ... = − Rep-regular x ⊓ − Rep-regular y
    by (simp add: inf-regular.rep-eq uminus-regular.rep-eq)
  also have ... ≤ Rep-dense v ⊓ Rep-dense w
    using 4 inf-mono mem-Collect-eq by blast
  also have ... = Rep-dense (v ⊓ w)
    by (simp add: inf-dense.rep-eq)
  also have ... ≤ Rep-dense z

using 4 by (simp add: less-eq-dense_rep-eq)
finally show \( z \in \text{stone-phi-set} (x \sqcup y) \)
by simp
qed
thus \( \text{stone-phi} (x \sqcup y) = \text{stone-phi} x \sqcup \text{stone-phi} y \)
by (simp add: stone-phi-def_eq-omp_same-args stone-phi-set-filter sup-filter.abs-eq)
qed
next
show \( \forall x y. \exists a \text{ regular} . \text{stone-phi} (x \sqcap y) = \text{stone-phi} x \sqcap \text{stone-phi} y \)
proof (intro allI)
fix \( x y :: \exists a \text{ regular} \)
have \( \forall z. \exists \text{Rep-regular} (x \sqcap y) \leq \text{Rep-dense} z \iff \neg \exists \text{Rep-regular} x \leq \neg \text{Rep-dense} z \)
by (simp add: inf-regular_rep-eq)
thus \( \text{stone-phi-set} (x \sqcap y) = (\text{stone-phi-set} x) \cap (\text{stone-phi-set} y) \)
by auto
thus \( \text{stone-phi} (x \sqcap y) = \text{stone-phi} x \sqcap \text{stone-phi} y \)
by (simp add: stone-phi-def_eq-omp_same-args stone-phi-set-filter inf-filter.abs-eq)
qed
qed

5.3 Properties of Triples

In this section we construct a certain set of pairs from a triple, introduce operations on these pairs and develop their properties. The given set and operations will form a Stone algebra.

context triple
begin

lemma phi-bot:
\( \phi \bot = \text{Abs-filter} \{ \text{top} \} \)
by (metis hom bot-filter-def)

lemma phi-top:
\( \phi \top = \text{Abs-filter} \text{UNIV} \)
by (metis hom top-filter-def)

The occurrence of \( \phi \) in the following definition of the pairs creates a need for dependent types.

definition pairs :: \( \exists a \times b \text{ filter} \text{ set} \)
where \( \text{pairs} = \{ (x,y) . \exists z . y = \phi (-x) \sqcup \text{up-filter} z \} \)

Operations on pairs are defined in the following. They will be used to establish that the pairs form a Stone algebra.

fun pairs-less-eq :: \( \exists a \times b \text{ filter} \Rightarrow (a \times b \text{ filter}) \Rightarrow \text{bool} \)
where \( \text{pairs-less-eq} (x,y) (z,w) = (x \leq z \land w \leq y) \)
fun pairs-less :: ('a × 'b filter) ⇒ ('a × 'b filter) ⇒ bool
where pairs-less (x,y) (z,w) = (pairs-less-eq (x,y) (z,w) ∧ ¬ pairs-less-eq (z,w) (x,y))

fun pairs-sup :: ('a × 'b filter) ⇒ ('a × 'b filter) ⇒ ('a × 'b filter)
where pairs-sup (x,y) (z,w) = (x ⊔ z,y ⊓ w)

fun pairs-inf :: ('a × 'b filter) ⇒ ('a × 'b filter) ⇒ ('a × 'b filter)
where pairs-inf (x,y) (z,w) = (x ⊓ z,y ⊔ w)

fun pairs-uminus :: ('a × 'b filter) ⇒ ('a × 'b filter)
where pairs-uminus (x,y) = (−x,phi x)

abbreviation pairs-bot :: ('a × 'b filter)
where pairs-bot ≡ (bot,Abs-filter UNIV)

abbreviation pairs-top :: ('a × 'b filter)
where pairs-top ≡ (top,Abs-filter {top})

lemma pairs-top-in-set:
(x,y) ∈ pairs ⇒ top ∈ Rep-filter y
by simp

lemma phi-complemented:
complement (phi x) (phi (−x))
by (metis hom inf-compl-bot sup-compl-top)

lemma phi-inf-principal:
∃ z . up-filter z = phi x ∩ up-filter y
proof
let ?F = Rep-filter (phi x)
let ?G = Rep-filter (phi (−x))
have 1: eq-onp filter ?F ?F ∧ eq-onp filter (↑y) (↑y)
  by (simp add: eq-onp-def)
have filter-complements ?F ?G
  apply (intro conjI)
  apply simp
  apply simp
  apply (metis (no-types) phi-complemented sup-filter.rep-eq top-filter.rep-eq)
  apply simp
  apply (metis (no-types) phi-complemented inf-filter.rep-eq bot-filter.rep-eq)
  hence is-principal-up (?F ∩ ↑y)
    using complemented-filter-inf-principal by blast
then obtain z where ↑z = ?F ∩ ↑y
  by auto
hence up-filter z = Abs-filter (?F ∩ ↑y)
  by simp
also have ... = Abs-filter ?F ∩ up-filter y
  using 1 inf-filter.abs-eq by force
Quite a bit of filter theory is involved in showing that the intersection of \( \phi x \) with a principal filter is a principal filter, so the following function can extract its least element.

```plaintext
fun rho :: 'a ⇒ 'b ⇒ 'b
where rho x y = (SOME z. up-filter z = \phi x ⊓ up-filter y)
```

The following results show that the pairs are closed under the given operations.

```plaintext
lemma pairs-sup-closed:
assumes ((x,y) ∈ pairs) and (z,w) ∈ pairs
shows pairs-sup (x,y) (z,w) ∈ pairs
proof
from assms obtain u v where y = \( \phi (-x) ∪\) up-filter u ∧ w = \( \phi (-z) ∪\) up-filter v
by simp
also have ... = (x ⊓,\( \phi (-x) ⊓\) up-filter u) ∩ (\( \phi (-z) ⊓\) up-filter v))
by (simp add: inf-sup-commute inf-sup-distrib1 sup-commute sup-left-commute)
also have ... = (x ⊓ z,\( \phi (-x) ⊓\) up-filter u) ∩ (\( \phi (-z) ⊓\) up-filter v)
using hom by simp
also have ... = (x ⊓ z,\( \phi (-x) ⊓\) up-filter u) ∩ (\( \phi (-z) ⊓\) up-filter v)
by (metis inf-sup-commute inf-sup-distrib1 sup-commute sup-left-commute)
finally show ?thesis
using pairs-def by auto
qed
```
lemma pairs-inf-closed:
  assumes \((x,y) \in \text{pairs}\)
  and \((z,w) \in \text{pairs}\)
  shows \(\text{pairs-inf} \ (x,y) \ (z,w) \in \text{pairs}\)
proof
  from assms obtain \(u\ v\) where \(y = \phi \ (-x) \uplus \text{up-filter} \ u \land w = \phi \ (-z) \uplus \text{up-filter} \ v\)
  using pairs-def by auto
  hence \(\text{pairs-inf} \ (x,y) \ (z,w) = (x \cap z, (\phi \ (-x) \uplus \text{up-filter} \ u) \uplus (\phi \ (-z) \uplus \text{up-filter} \ v))\)
  by simp
  also have \(... = (x \cap z, (\phi \ (-x) \uplus \phi \ (-z)) \uplus (\text{up-filter} \ u \uplus \text{up-filter} \ v))\)
  by (simp add: sup-commute sup-left-commute)
  also have \(... = (x \cap z, \phi \ (-x \cap z) \uplus (\text{up-filter} \ u \uplus \text{up-filter} \ v))\)
  using hom by simp
  also have \(... = (x \cap z, \phi \ (- (x \cap z)) \uplus \text{up-filter} \ (u \cap v))\)
  by (simp add: up-filter-dist-inf)
  finally show \(?\text{thesis}\)
  using pairs-def by auto
qed

lemma pairs-uminus-closed:
  \(\text{pairs-uminus} \ (x,y) \in \text{pairs}\)
proof
  have \(\text{pairs-uminus} \ (x,y) = (-x, \phi \ (-x) \uplus \text{bot})\)
  by simp
  also have \(... = (-x, \phi \ (-x) \uplus \text{up-filter} \ \text{top})\)
  by (simp add: bot-filter.abs-eq)
  finally show \(?\text{thesis}\)
  by (metis (mono-tags, lifting) mem-Collect-eq old.prod.case pairs-def)
qed

lemma pairs-bot-closed:
  \(\text{pairs-bot} \in \text{pairs}\)
using pairs-def phi-top triple.hom triple-axioms by fastforce

lemma pairs-top-closed:
  \(\text{pairs-top} \in \text{pairs}\)
by (metis p-bot pairs-uminus.simps pairs-uminus-closed phi-bot)

We prove enough properties of the pair operations so that we can later show they form a Stone algebra.

lemma pairs-sup-dist-inf:
  \((x,y) \in \text{pairs} \Rightarrow (z,w) \in \text{pairs} \Rightarrow (u,v) \in \text{pairs} \Rightarrow \text{pairs-sup} \ (x,y) \ (z,w) \ (u,v) = \text{pairs-sup} \ (x,y) \ (z,w) \ (u,v))\)
using sup-inf-distrib1 inf-sup-distrib1 by auto

lemma pairs-phi-less-eq:
  \((x,y) \in \text{pairs} \Rightarrow \phi \ (-x) \leq y\)
using pairs-def by auto

lemma pairs-uminus-galois:
  assumes \((x, y) \in \text{pairs}\)
  and \((z, w) \in \text{pairs}\)
  shows \(\text{pairs-inf } (x, y) (z, w) = \text{pairs-bot } \leftrightarrow \text{pairs-less-eq } (x, y) \quad \text{pairs-uminus} \quad (z, w)\)
proof
  have 1: \(x \cap z = \text{bot} \wedge y \cup w = \text{Abs-filter } \text{UNIV} \quad \text{phi} \quad z \leq y\)
    by (metis (no-types, lifting) assms(1) heyting.implies-inf-absorb hom le-supE pairs-phi-less-eq sup-bot-right)
  have 2: \(x \leq -z \wedge \text{phi } z \leq y \quad \text{sup-mono } \text{pairs-phi-less-eq } \text{by auto}\)
proof
  assume 3: \(x \leq -z \wedge \text{phi } z \leq y\)
  have \(\text{Abs-filter } \text{UNIV} = \text{phi } z \sqcup \text{phi } (-z)\)
    using hom phi-complemented phi-top by auto
  also have \(\text{... } \leq y \sqcup w\)
    using 3 assms(2) sup-mono pairs-phi-less-eq by auto
  finally show \(y \sqcup w = \text{Abs-filter } \text{UNIV}\)
    using hom phi-top top.extremum-uniqueI by auto
qed

have \(x \cap z = \text{bot } \leftrightarrow x \leq -z\)
by (simp add: shunting-1)
thus \(?\text{thesis}\)
  using 1 2 Pair-inject pairs-inf.simps pairs-less-eq.simps pairs-uminus.simps simps by auto
qed

lemma pairs-stone:
\[(x, y) \in \text{pairs} \quad \Rightarrow \quad \text{pairs-sup } (\text{pairs-uminus } (x, y)) \quad \text{pairs-uminus } (\text{pairs-uminus} \quad (x, y)) = \text{pairs-top}\]
by (metis hom pairs-sup.simps pairs-uminus.simps phi-bot phi-complemented stone)

The following results show how the regular elements and the dense elements among the pairs look like.

abbreviation dense-pairs \(\equiv \{ \quad (x, y) \quad . \quad (x, y) \in \text{pairs} \wedge \text{pairs-uminus } (x, y) = \text{pairs-bot} \quad \}\)
abbreviation regular-pairs \(\equiv \{ \quad (x, y) \quad . \quad (x, y) \in \text{pairs} \wedge \text{pairs-uminus} \quad (x, y) \quad = \quad (x, y) \quad \}\)
abbreviation is-principal-up-filter \(x \equiv \exists \quad y \quad . \quad x = \text{up-filter } y\)

lemma dense-pairs:
dense-pairs = \(\{ \quad (x, y) \quad . \quad x = \text{top} \wedge \text{is-principal-up-filter } y \quad \}\)
proof
  have dense-pairs = \(\{ \quad (x, y) \quad . \quad (x, y) \in \text{pairs} \wedge x = \text{top} \quad \}\)
    by (metis Pair-inject compl-bot-eq double-compl pairs-uminus.simps phi-top)
  also have \(\text{... } \equiv \{ \quad (x, y) \quad . \quad (\exists z \quad . \quad y = \text{up-filter } z) \wedge x = \text{top} \quad \}\)
    using hom pairs-def by auto

qed
finally show ?thesis
  by auto
qed

lemma regular-pairs:
  regular-pairs = \{ (x,y) . y = phi (-x) \}
using pairs-def pairs-uminus-closed by fastforce

  The following extraction function will be used in defining one direction
of the Stone algebra isomorphism.
fun rho-pair :: 'a × 'b filter ⇒ 'b
  where rho-pair (x,y) = (SOME z . up-filter z = phi x ⊓ y)

lemma get-rho-pair-char:
  assumes (x,y) ∈ pairs
  shows up-filter (rho-pair (x,y)) = phi x ⊓ y
proof –
  from assms obtain w where y = phi (-x) ⊓ up-filter w
  using pairs-def by auto
  hence phi x ⊓ y = phi x ⊓ up-filter w
  by (simp add: inf-sup-distrib phi-complemented)
  thus ?thesis
  using rho-char by auto
qed

lemma sa-iso-pair:
  (−−x,phi (-x) ⊓ up-filter y) ∈ pairs
using pairs-def by auto
end

5.4 The Stone Algebra of a Triple

In this section we prove that the set of pairs constructed in a triple forms a
Stone Algebra. The following type captures the parameter phi on which the
type of triples depends. This parameter is the structure map that occurs
in the definition of the set of pairs. The set of all structure maps is the
set of all bounded lattice homomorphisms (of appropriate type). In order
to make it a HOL type, we need to show that at least one such structure
map exists. To this end we use the ultrafilter lemma: the required bounded
lattice homomorphism is essentially the characteristic map of an ultrafilter,
but the latter must exist. In particular, the underlying Boolean algebra
must contain at least two elements.

typedef (overloaded) ('a,'b) phi = \{ f::'a::non-trivial-boolean-algebra ⇒
'b::distrib-lattice-top filter . bounded-lattice-homomorphism f \}
proof –
  from ultra-filter-exists obtain F :: 'a set where 1: ultra-filter F
  by auto
hence 2: prime-filter F
   using ultra-filter-prime by auto
let \( \mathcal{F} = \lambda x . \text{if } x \in F \text{ then } \text{top else } \text{bot} \) filter
have bounded-lattice-homomorphism \( \mathcal{F} \)
proof (intro conjI)
  show \( \mathcal{F} \text{ bot} = \text{bot} \)
  using 1 by (meson bot.extremum filter-def subset-eq top.extremum-unique)
next
  show \( \mathcal{F} \text{ top} = \text{top} \)
  using 1 by simp
next
  show \( \forall x y . \mathcal{F} \text{ (} x \sqcup y \text{)} = \mathcal{F} \text{ x} \sqcup \mathcal{F} \text{ y} \)
proof (intro allI)
  fix x y
  show \( \mathcal{F} \text{ (} x \sqcup y \text{)} = \mathcal{F} \text{ x} \sqcup \mathcal{F} \text{ y} \)
  apply (cases \( x \in F; \text{ cases } y \in F \))
  using 1 filter-def apply fastforce
  using 1 filter-def apply fastforce
  using 1 filter-def apply fastforce
  using 2 sup-bot-left by auto
qed
next
  show \( \forall x y . \mathcal{F} \text{ (} x \sqcap y \text{)} = \mathcal{F} \text{ x} \sqcap \mathcal{F} \text{ y} \)
proof (intro allI)
  fix x y
  show \( \mathcal{F} \text{ (} x \sqcap y \text{)} = \mathcal{F} \text{ x} \sqcap \mathcal{F} \text{ y} \)
  apply (cases \( x \in F; \text{ cases } y \in F \))
  using 1 apply (simp add: filter-inf-closed)
  using 1 apply (metis (mono-tags, lifting) brouwer.inf-sup-ord(4)
  inf-top-left filter-def)
  using 1 apply (metis (mono-tags, lifting) brouwer.inf-sup-ord(3)
  inf-top-right filter-def)
  using 1 filter-def by force
qed
qed
hence \( \mathcal{F} \in \{ f . \text{bounded-lattice-homomorphism } f \} \)
by simp
thus \( \text{thesis} \)
by meson
qed

lemma simp-phi [simp]:
   bounded-lattice-homomorphism (Rep-phi x)
using Rep-phi by simp

setup-lifting type-definition-phi

The following implements the dependent type of pairs depending on
structure maps. It uses functions from structure maps to pairs with the
requirement that, for each structure map, the corresponding pair is contained
in the set of pairs constructed for a triple with that structure map.

If this type could be defined in the locale triple and instantiated to Stone algebras there, there would be no need for the lifting and we could work with triples directly.

```tangle
typedef (overloaded) ('a', 'b') lifted-pair = { pf:('a::non-trivial-boolean-algebra', 'b::distrib-lattice-top) phi ⇒ 'a × 'b filter . ∀ f . pf f ∈ triple.pairs (Rep-phi f) }
proof
  have ∀ f::('a, 'b) phi . triple.pairs-bot ∈ triple.pairs (Rep-phi f)
  proof
    fix f:: ('a, 'b) phi
    have triple (Rep-phi f)
      by (simp add: triple-def)
    thus triple.pairs-bot ∈ triple.pairs (Rep-phi f)
      using triple.regular-pairs triple.phi-top by fastforce
  qed
  thus ?thesis
    by auto
qed

lemma simp-lifted-pair [simp]:
  ∀ f . Rep-lifted-pair pf f ∈ triple.pairs (Rep-phi f)
  using Rep-lifted-pair by simp

setup-lifting type-definition-lifted-pair

The lifted pairs form a Stone algebra.

instantiation lifted-pair :: (non-trivial-boolean-algebra, distrib-lattice-top)
stone-algebra
begin
  All operations are lifted point-wise.

lift-definition sup-lifted-pair :: ('a', 'b) lifted-pair ⇒ ('a', 'b) lifted-pair ⇒ ('a', 'b)
  lifted-pair is λxf yf f . triple.pairs-sup (xf f) (yf f)
  by (metis (no-types, hide-lams) simp-phi triple-def triple.pairs-sup-closed prod.collapse)

lift-definition inf-lifted-pair :: ('a', 'b) lifted-pair ⇒ ('a', 'b) lifted-pair ⇒ ('a', 'b)
  lifted-pair is λxf yf f . triple.pairs-inf (xf f) (yf f)
  by (metis (no-types, hide-lams) simp-phi triple-def triple.pairs-inf-closed prod.collapse)

lift-definition uminus-lifted-pair :: ('a', 'b) lifted-pair ⇒ ('a', 'b) lifted-pair is λxf f . triple.pairs-uminus (Rep-phi f) (xf f)
  by (metis (no-types, hide-lams) simp-phi triple-def triple.pairs-uminus-closed prod.collapse)

lift-definition bot-lifted-pair :: ('a', 'b) lifted-pair is λf . triple.pairs-bot
  by (metis (no-types, hide-lams) simp-phi triple-def triple.pairs-bot-closed)
```
lift-definition top-lifted-pair :: ('a,'b) lifted-pair is λf . triple.pairs-top
  by (metis (no-types, hide-lams) simp-phi triple-def triple.pairs-top-closed)

lift-definition less-eq-lifted-pair :: ('a,'b) lifted-pair ⇒ ('a,'b) lifted-pair ⇒ bool
  is λxf yf . ∀f . triple.pairs-less-eq (xf f) (yf f) .

lift-definition less-lifted-pair :: ('a,'b) lifted-pair ⇒ ('a,'b) lifted-pair ⇒ bool is
  λxf yf . (∀f . triple.pairs-less-eq (xf f) (yf f)) ∧ ¬(∀f . triple.pairs-less-eq (yf f) (xf f)) .

instance
proof intro-classes
  fix xf yf :: ('a,'b) lifted-pair
  show xf < yf ⟷ xf ≤ yf ∧ ¬yf ≤ xf
    by (simp add: less-lifted-pair.rep-eq less-eq-lifted-pair.rep-eq)
next
  fix xf :: ('a,'b) lifted-pair
  { fix f :: ('a,'b) phi
    have 1: triple (Rep-phi f)
      by (simp add: triple-def)
    let ?x = Rep-lifted-pair xf f
    obtain x1 x2 where (x1,x2) = ?x
      using prod.collapse by blast
    hence triple.pairs-less-eq ?x ?x
      using 1 by (metis triple.pairs-less-eq.simps order-refl)
  } thus xf ≤ xf
    by (simp add: less-eq-lifted-pair.rep-eq)
next
  fix xf yf zf :: ('a,'b) lifted-pair
  assume 1: xf ≤ yf and 2: yf ≤ zf
  { fix f :: ('a,'b) phi
    have 3: triple (Rep-phi f)
      by (simp add: triple-def)
    let ?x = Rep-lifted-pair xf f
    let ?y = Rep-lifted-pair yf f
    let ?z = Rep-lifted-pair zf f
    obtain x1 x2 y1 y2 z1 z2 where 4: (x1,x2) = ?x ∧ (y1,y2) = ?y ∧ (z1,z2) = ?z
      using prod.collapse by blast
    have triple.pairs-less-eq ?x ?y ∧ triple.pairs-less-eq ?y ?z
      using 1 2 3 less-eq-lifted-pair.rep-eq by simp
    hence triple.pairs-less-eq ?x ?z
      using 3 4 by (metis (mono-tags, lifting) triple.pairs-less-eq.simps order-trans)
  }
thus $x \leq z$
by (simp add: less-eq-lifted-pair.rep-eq)

next

fix $x$ $y$ :: ('a,'b) lifted-pair

assume 1: $x \leq y$ and 2: $y \leq x$

\{ 

\begin{align*}
\text{fix } f :: ('a,'b) \phi \\
\text{have 3: triple (Rep-\phi f)} \\
\text{by (simp add: triple-def)} \\
\text{let } ?x = \text{Rep-lifted-pair } x f \\
\text{let } ?y = \text{Rep-lifted-pair } y f \\
\text{obtain } x1 \ x2 \ y1 \ y2 \text{ where } 4: (x1,x2) = (?x \land (y1,y2) = ?y \\
\text{using prod.collapse by blast} \\
\text{have triple.pairs-less-eq } ?x \ ?y \land \text{triple.pairs-less-eq } ?y \ ?x \\
\text{using 1 2 3 less-eq-lifted-pair.rep-eq by simp} \\
\text{hence } ?x = ?y \\
\text{using 3 4 by (metis (mono-tags, lifting) triple.pairs-less-eq.simps antisym))}
\end{align*}

\}

thus $x = y$
by (metis Rep-lifted-pair-inverse ext)

next

fix $x$ $y$ :: ('a,'b) lifted-pair

\{ 

\begin{align*}
\text{fix } f :: ('a,'b) \phi \\
\text{have 1: triple (Rep-\phi f)} \\
\text{by (simp add: triple-def)} \\
\text{let } ?x = \text{Rep-lifted-pair } x f \\
\text{let } ?y = \text{Rep-lifted-pair } y f \\
\text{obtain } x1 \ x2 \ y1 \ y2 \text{ where } (x1,x2) = (?x \land (y1,y2) = ?y \\
\text{using prod.collapse by blast} \\
\text{hence triple.pairs-less-eq (triple.pairs-inf )?x ?y} \\
\text{using 7 by (metis (mono-tags, lifting) inf-sup-ord(2) sup.co\text{bounded2 triple.pairs-inf.simps triple.pairs-less-eq.simps inf-lifted-pair.rep-eq))}
\end{align*}

\}

thus $x \sqcap y \leq y$
by (simp add: less-eq-lifted-pair.rep-eq inf-lifted-pair.rep-eq)

next

fix $x$ $y$ :: ('a,'b) lifted-pair

\{ 

\begin{align*}
\text{fix } f :: ('a,'b) \phi \\
\text{have 1: triple (Rep-\phi f)} \\
\text{by (simp add: triple-def)} \\
\text{let } ?x = \text{Rep-lifted-pair } x f \\
\text{let } ?y = \text{Rep-lifted-pair } y f \\
\text{obtain } x1 \ x2 \ y1 \ y2 \text{ where } (x1,x2) = (?x \land (y1,y2) = ?y \\
\text{using prod.collapse by blast} \\
\text{hence triple.pairs-less-eq (triple.pairs-inf )?x ?y} \\
\text{using 7 by (metis (mono-tags, lifting) inf-sup-ord(1) sup.co\text{bounded1 triple.pairs-inf.simps triple.pairs-less-eq.simps inf-lifted-pair.rep-eq))}
\end{align*}

\}
thus $xf \cap yf \leq xf$
  by (simp add: less-eq-lifted-pair.rep-eq inf-lifted-pair.rep-eq)

next
fix $xf yf zf :: (a',b)\ lifted-pair$
assume $1: xf \leq yf$ and $2: xf \leq zf$
{  
  fix $f :: (a',b)\ phi$
  have $3: \text{triple}(\text{Rep-phi}\ f)$
    by (simp add: triple-def)
  let $?x = \text{Rep-lifted-pair}\ xf f$
  let $?y = \text{Rep-lifted-pair}\ yf f$
  let $?z = \text{Rep-lifted-pair}\ zf f$
  obtain $x1\ x2\ y1\ y2\ z1\ z2$ where $4: (x1,x2) = ?x \land (y1,y2) = ?y \land (z1,z2)$
    $= ?z$
    using prod.collapse by blast
  have triple.pairs-less-eq $?x\ ?y \land \text{triple.pairs-less-eq}\ ?x\ ?z$  
    using $1\ 2\ 3$ less-eq-lifted-pair.rep-eq by simp
  hence triple.pairs-less-eq $?x (\text{triple.pairs-inf}\ ?y \ ?z)$  
    using $3\ 4$ by (metis (mono-tags, lifting) le-inf-iff sup.bounded-iff triple.pairs-inf.simps triple.pairs-less-eq.simps)
}
thus $xf \leq yf \cap zf$
  by (simp add: less-eq-lifted-pair.rep-eq inf-lifted-pair.rep-eq)

next
fix $xf yf :: (a',b)\ lifted-pair$
{  
  fix $f :: (a',b)\ phi$
  have $1: \text{triple}(\text{Rep-phi}\ f)$
    by (simp add: triple-def)
  let $?x = \text{Rep-lifted-pair}\ xf f$
  let $?y = \text{Rep-lifted-pair}\ yf f$
  obtain $x1\ x2\ y1\ y2$ where $2: (x1,x2) = ?x \land (y1,y2) = ?y$
    using prod.collapse by blast
  hence triple.pairs-less-eq $?x (\text{triple.pairs-sup}\ ?x \ ?y)$  
    using $1$ by (metis (no-types, lifting) inf-commute sup.cobounded1 inf.cobounded2 triple.pairs-sup.simps triple.pairs-less-eq.simps triple.pairs-inf.simps sup-lifted-pair.rep-eq)
}
thus $xf \leq xf \sqcup yf$
  by (simp add: less-eq-lifted-pair.rep-eq sup-lifted-pair.rep-eq)

next
fix $xf yf :: (a',b)\ lifted-pair$
{  
  fix $f :: (a',b)\ phi$
  have $1: \text{triple}(\text{Rep-phi}\ f)$
    by (simp add: triple-def)
  let $?x = \text{Rep-lifted-pair}\ xf f$
  let $?y = \text{Rep-lifted-pair}\ yf f$
  hence $\text{triple.pairs-inf}\ ?x \ ?y$  
    using triple.pairs-inf.simps triple.pairs-less-eq.simps
obtain $x_1 \ x_2 \ y_1 \ y_2$ where $(x_1, x_2) = ?x$ and $(y_1, y_2) = ?y$
using prod.collapse by blast
hence triple.pairs-less-eq ?y (triple.pairs-sup ?x ?y)
using f by (metis (no-types, lifting) sup.cobounded2 inf.cobounded2 triple.pairs-sup.simps triple.pairs-less-eq.simps sup-lifted-pair.rep-eq)
}
thus $yf \leq xf \sqcup yf$
by (simp add: less-eq-lifted-pair.rep-eq sup-lifted-pair.rep-eq)

next
fix $xf \ yf \ zf :: ('a, 'b) lifted-pair$
assume 1: $yf \leq xf$ and 2: $zf \leq xf$
{
  fix $f :: ('a, 'b) phi$
  have 3: triple (Rep-phi f)
    by (simp add: triple-def)
  let $?x = Rep-lifted-pair xf f$
  let $?y = Rep-lifted-pair yf f$
  let $?z = Rep-lifted-pair zf f$
  obtain $x_1 \ x_2 \ y_1 \ y_2 \ z_1 \ z_2$ where 4: $(x_1, x_2) = ?x$ and $(y_1, y_2) = ?y$ and $(z_1, z_2) = ?z$
using prod.collapse by blast
  have triple.pairs-less-eq ?y ?z ?x
    using 1 2 3 less-eq-lifted-pair.rep-eq by simp
  hence triple.pairs-less-eq (triple.pairs-sup ?y ?z) ?x
    using 3 4 by (metis (mono-tags, lifting) le-inf-iff sup.bounded-iff triple.pairs-sup.simps triple.pairs-less-eq.simps)
  }
  thus $yf \sqcup zf \leq xf$
  by (simp add: less-eq-lifted-pair.rep-eq sup-lifted-pair.rep-eq)

next
fix $xf :: ('a, 'b) lifted-pair$
{
  fix $f :: ('a, 'b) phi$
  have 1: triple (Rep-phi f)
    by (simp add: triple-def)
  let $?x = Rep-lifted-pair xf f$
  obtain $x_1 \ x_2$ where $(x_1, x_2) = ?x$
    using prod.collapse by blast
  hence triple.pairs-less-eq triple.pairs-bot ?x
    using f by (metis bot.extremum top-greatest top-filter.abs-eq triple.pairs-less-eq.simps)
  }
  thus $bot \leq xf$
  by (simp add: less-eq-lifted-pair.rep-eq bot-lifted-pair.rep-eq)

next
fix $xf :: ('a, 'b) lifted-pair$
{
  fix $f :: ('a, 'b) phi$
  have 1: triple (Rep-phi f)

by (simp add: triple-def)
let \( ?x = \text{Rep-lifted-pair } xf f \)
obtain \( x1 \ x2 \) where \( (x1,x2) = ?x \)
  using prod.collapse by blast
hence triple.pairs-less-eq \( ?x \) triple.pairs-top
  using \( f \) by (metis top.extremum bot.least bot.filter.abs-eq
    triple.pairs-less-eq.simps)
}
thus \( xf \leq \text{top} \)
  by (simp add: less-eq.lifted-pair.rep.eq top.lifted-pair.rep.eq)
next
fix \( xf \ yf \ zf \) :: ('a,'b) lifted-pair
{
  fix \( f \) :: ('a,'b) phi
  have \( 1 : \text{triple } (\text{Rep-}\phi f) \)
    by (simp add: triple-def)
  let \( ?x = \text{Rep-lifted-pair } xf f \)
  let \( ?y = \text{Rep-lifted-pair } yf f \)
  let \( ?z = \text{Rep-lifted-pair } zf f \)
  obtain \( x1 \ x2 \ y1 \ y2 \ z1 \ z2 \) where \( (x1,x2) = ?x \land (y1,y2) = ?y \land (z1,z2) = ?z \)
    using prod.collapse by blast
hence triple.pairs-sup \( ?x \) (triple.pairs-inf \( ?y \) ?z) = triple.pairs-inf
  (triple.pairs-sup \( ?x \ ?y \)) (triple.pairs-sup \( ?x \ ?z \))
    using \( f \) by (metis (no_types) sup-inf-distrib1 inf-sup-distrib1
      triple.pairs-sup.simps triple.pairs-inf.simps)
}
thus \( xf \sqcup (yf \sqcap zf) = (xf \sqcup yf) \sqcap (xf \sqcup zf) \)
  by (metis Rep.lifted-pair.inverse ext sup.lifted-pair.rep.eq inf.lifted-pair.rep.eq)
next
fix \( xf \ yf \) :: ('a,'b) lifted-pair
{
  fix \( f \) :: ('a,'b) phi
  have \( 1 : \text{triple } (\text{Rep-}\phi f) \)
    by (simp add: triple-def)
  let \( ?x = \text{Rep-lifted-pair } xf f \)
  let \( ?y = \text{Rep-lifted-pair } yf f \)
  obtain \( x1 \ x2 \ y1 \ y2 \) where \( 2 : (x1,x2) = ?x \land (y1,y2) = ?y \)
    using prod.collapse by blast
hence \( ?x \in \text{triple.pairs } (\text{Rep-}\phi f) \land ?y \in \text{triple.pairs } (\text{Rep-}\phi f) \)
    by simp
hence (triple.pairs-inf \( ?x \ ?y \) = triple.pairs-bot) \iff triple.pairs-less-eq \( ?x \ ?y \)
  (triple.pairs-uminus (\text{Rep-}\phi f) \ ?y)
    using \( 1 \ 2 \) by (metis triple.pairs-uminus-galois)
}
  hence \( \forall f \ . \ (\text{Rep-lifted-pair } (xf \sqcap yf) f = \text{Rep-lifted-pair } \text{bot } f) \iff \)
    triple.pairs-less-eq (\text{Rep-lifted-pair } xf f) (Rep.lifted-pair (-yf) f)
    using bot.lifted-pair.rep.eq inf.lifted-pair.rep.eq uminus.lifted-pair.rep.eq by simp
  hence (\text{Rep-lifted-pair } (xf \sqcap yf) = \text{Rep-lifted-pair } \text{bot } f) \iff xf \leq -yf
using less-eq-lifted-pair.rep-eq by auto
thus $(xf \cap yf = \bot) \iff (xf \leq -yf)$
by (simp add: Rep-lifted-pair-inject)

next
fix $xf :: ('a,'b)$ lifted-pair
{
  fix $f :: ('a,'b)$ phi
  have 1: triple (Rep-phi $f$)
  by (simp add: triple-def)
  let $?x = Rep-lifted-pair xf f$
  obtain $x1 x2$ where $(x1,x2) = ?x$
  using prod.collapse by blast
  hence triple.pairs-sup (triple.pairs-uminus (Rep-phi $f$) $?x)
  (triple.pairs-uminus (Rep-phi $f$) (triple.pairs-uminus (Rep-phi $f$) $?x)) =
  triple.pairs-top
  using $f$ by (metis simp-lifted-pair triple.pairs-stone)
  }
  hence $Rep-lifted-pair (-xf \sqcup -xf) = Rep-lifted-pair top$
  using sup-lifted-pair.rep-eq uminus-lifted-pair.rep-eq top-lifted-pair.rep-eq by simp
  thus $-xf \sqcup -xf = top$
  by (simp add: Rep-lifted-pair-inject)
qed

5.5 The Stone Algebra of the Triple of a Stone Algebra

In this section we specialise the above construction to a particular structure map, namely the one obtained in the triple of a Stone algebra. For this particular structure map (as well as for any other particular structure map) the resulting type is no longer a dependent type. It is just the set of pairs obtained for the given structure map.

typedef (overloaded) 'a stone-phi-pair = triple.pairs
(stone-phi::'a::stone-algebra regular ⇒ 'a dense-filter)
using stone-phi.pairs-bot-closed by auto

setup-lifting type-definition-stone-phi-pair

instantiation stone-phi-pair :: (stone-algebra) sup-inf-top-bot-uminus-ord
begin

lift-definition sup-stone-phi-pair :: 'a stone-phi-pair ⇒ 'a stone-phi-pair ⇒ 'a
stone-phi-pair is triple.pairs-sup
using stone-phi.pairs-sup-closed by auto

lift-definition inf-stone-phi-pair :: 'a stone-phi-pair ⇒ 'a stone-phi-pair ⇒ 'a
stone-phi-pair is triple.pairs-inf

77
using stone-phi.pairs-inf-closed by auto

lift-definition uminus-stone-phi-pair :: 'a stone-phi-pair ⇒ 'a stone-phi-pair is triple.pairs-uminus stone-phi
  using stone-phi.pairs-uminus-closed by auto

lift-definition bot-stone-phi-pair :: 'a stone-phi-pair is triple.pairs-bot
  by (rule stone-phi.pairs-bot-closed)

lift-definition top-stone-phi-pair :: 'a stone-phi-pair is triple.pairs-top
  by (rule stone-phi.pairs-top-closed)

lift-definition less-eq-stone-phi-pair :: 'a stone-phi-pair ⇒ 'a stone-phi-pair ⇒ bool is triple.pairs-less-eq.

lift-definition less-stone-phi-pair :: 'a stone-phi-pair ⇒ 'a stone-phi-pair ⇒ bool is λxf yf. triple.pairs-less-eq xf yf ∧ ¬ triple.pairs-less-eq yf xf.

instance ..

end

The result is a Stone algebra and could be proved so by repeating and specialising the above proof for lifted pairs. We choose a different approach, namely by embedding the type of pairs into the lifted type. The embedding injects a pair \( x \) into a function as the value at the given structure map; this makes the embedding injective. The value of the function at any other structure map needs to be carefully chosen so that the resulting function is a Stone algebra homomorphism. We use \( \sim x \), which is essentially a projection to the regular element component of \( x \), whence the image has the structure of a Boolean algebra.

fun stone-phi-embed :: 'a::non-trivial-stone-algebra stone-phi-pair ⇒ ('a regular,'a dense) lifted-pair
  where stone-phi-embed x = Abs-lifted-pair (λf. if Rep-phi f = stone-phi then Rep-stone-phi-pair x else triple.pairs-uminus (Rep-phi f) (triple.pairs-uminus (Rep-phi f) (Rep-stone-phi-pair x)))

The following lemma shows that in both cases the value of the function is a valid pair for the given structure map.

lemma stone-phi-embed-triple-pair:
  (if Rep-phi f = stone-phi then Rep-stone-phi-pair x else triple.pairs-uminus (Rep-phi f) (triple.pairs-uminus (Rep-phi f) (Rep-stone-phi-pair x))) ∈ triple.pairs (Rep-phi f)
  by (metis (no-types, hide-lams) Rep-stone-phi-pair simp-phi surj-pair triple.pairs-uminus-closed triple-def)

The following result shows that the embedding preserves the operations of Stone algebras. Of course, it is not (yet) a Stone algebra homomorphism
as we do not know (yet) that the domain of the embedding is a Stone algebra.
To establish the latter is the purpose of the embedding.

**Lemma** `stone-phi-embed-homomorphism`:

\[
\text{sup-inf-top-bot-uminus-ord-homomorphism stone-phi-embed}
\]

**Proof** (intro `conjI`)

let \( ?p = \lambda f \cdot \text{triple.pairs-uminus (Rep-\phi f)} \)

let \( ?pp = \lambda f x . ?p f ( ?p f x ) \)

let \( ?q = \lambda f x . ?pp f ( \text{Rep-stone-\phi-pair x} ) \)

\[ \forall x y : \text{a stone-\phi-pair} . \text{stone-\phi-embed} ( x \sqcup y ) = \text{stone-\phi-embed} x \sqcup \text{stone-\phi-embed} y \]

**Proof** (intro `allI`)

fix \( x y :: \text{a stone-\phi-pair} \)

have \( 1. \forall f . \text{triple.pairs-sup} (?q f x) (?q f y) = ?q f (x \sqcup y) \)

**Proof**

fix \( f :: \text{('a regular,'a dense) \phi} \)

let \( ?r = \text{Rep-\phi f} \)

obtain \( x1 x2 y1 y2 \) where \( 2. (x1,x2) = \text{Rep-stone-\phi-pair x} \land (y1,y2) = \text{Rep-stone-\phi-pair y} \)

using `prod.collapse` by blast

hence `triple.pairs-sup (?q f x) (?q f y) = triple.pairs-sup (?pp f (x1,x2)) (?pp f (y1,y2))`

by `simp`

also have \( ... = \text{triple.pairs-sup} (- - x1, ?r (- - x1)) (- - y1, ?r (- - y1)) \)

by `(simp add: triple.pairs-uminus.simps triple-def)`

also have \( ... = (- - x1 \sqcup - - y1, ?r (- - x1) \sqcup ?r (- - y1)) \)

by `simp`

also have \( ... = (- - (x1 \sqcup y1), ?r (- - (x1 \sqcup y1))) \)

by `simp`

also have \( ... = ?pp f (x1 \sqcup y1,x2 \sqcup y2) \)

by `(simp add: triple.pairs-uminus.simps triple-def)`

also have \( ... = ?pp f (triple.pairs-sup (x1,x2) (y1,y2)) \)

by `simp`

also have \( ... = ?q f (x \sqcup y) \)

using `2` by `(simp add: sup-stone-\phi-pair.rep-eq)`

finally show `triple.pairs-sup (?q f x) (?q f y) = ?q f (x \sqcup y)`

qed

have `stone-\phi-embed x \sqcup \text{stone-\phi-embed y} = \text{Abs-lifted-pair} (\lambda f . \text{if Rep-\phi f = stone-\phi then Rep-stone-\phi-pair x else ?q f x}) \sqcup \text{Abs-lifted-pair} (\lambda f . \text{if Rep-\phi f = stone-\phi then Rep-stone-\phi-pair y else ?q f y})`

by `simp`

also have \( ... = \text{Abs-lifted-pair} (\lambda f . \text{triple.pairs-sup (if Rep-\phi f = stone-\phi then Rep-stone-\phi-pair x else ?q f x) \sqcup Rep-stone-\phi-pair y else ?q f y}) \)

by `(rule sup-lifted-pair.abs-eq) (simp-all add: eq-onp-same-args stone-\phi-embed-triple-pair)`

also have \( ... = \text{Abs-lifted-pair} (\lambda f . \text{if Rep-\phi f = stone-\phi then triple.pairs-sup (Rep-stone-\phi-pair x) (Rep-stone-\phi-pair y) else triple.pairs-sup (?q f x) (?q f y)}) \)

79
by (simp add: if-distrib-2)
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then
  triple.pairs-sup (Rep-stone-phi-pair x) (Rep-stone-phi-pair y) else ?q f (x ⊔ y))
  using f by meson
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then
  Rep-stone-phi-pair (x ⊔ y) else ?q f (x ⊔ y))
  by (metis sup-stone-phi-pair.rep-eq)
also have ... = stone-phi-embed (x ⊔ y)
  by simp
finally show stone-phi-embed (x ⊔ y) = stone-phi-embed x ⊔ stone-phi-embed y
  by simp
qed

next
let ?p = λf. triple.pairs-uminus (Rep-phi f)
let ?pp = λf x. ?p f (?p f x)
let ?q = λf x. ?pp f (Rep-stone-phi-pair x)
show ∀ x y::'a stone-phi-pair. stone-phi-embed (x ⊓ y) = stone-phi-embed x ⊓ stone-phi-embed y
proof (intro allI)
fix x y :: 'a stone-phi-pair
have 1: ∀ f. triple.pairs-inf (?q f x) (?q f y) = ?q f (x ⊓ y)
proof
  fix f :: ('a regular,'a dense) phi
  let ?r = Rep-phi f
  obtain x1 x2 y1 y2 where 2: (x1,x2) = Rep-stone-phi-pair x ∧ (y1,y2) = Rep-stone-phi-pair y
    using prod.collapse by blast
  hence triple.pairs-inf (?q f x) (?q f y) = triple.pairs-inf (?pp f (x1,x2)) (?pp f (y1,y2))
    by simp
  also have ... = triple.pairs-inf (---x1,?r (-x1)) (---y1,?r (-y1))
    by (simp add: triple.pairs-uminus.simps triple-def)
  also have ... = (---x1 ⊓ ---y1,?r (-x1) ⊓ ?r (-y1))
    by simp
  also have ... = (---(x1 ⊓ y1),?r (-x1 ⊓ y1))
    by simp
  also have ... = ?pp f (x1 ⊓ y1,x2 ⊓ y2)
    by (simp add: triple.pairs-uminus.simps triple-def)
  also have ... = ?pp f (triple.pairs-inf (x1,x2) (y1,y2))
    by simp
  also have ... = ?q f (x ⊓ y)
    using 2 by (simp add: inf-stone-phi-pair.rep-eq)
finally show triple.pairs-inf (?q f x) (?q f y) = ?q f (x ⊓ y)
  qed
have stone-phi-embed x ⊓ stone-phi-embed y = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair x else ?q f x) ⊓ Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair y else ?q f y)
by simp
also have ... = Abs-lifted-pair (λf . triple.pairs-inf (if Rep-phi f = stone-phi then Rep-stone-phi-pair x else ?q f x) (if Rep-phi f = stone-phi then Rep-stone-phi-pair y else ?q f y))
  by (rule inf-lifted-pair.abs-eq) (simp-all add: eq-opn-same-args stone-phi-embed-triple-pair)
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then triple.pairs-inf (Rep-stone-phi-pair x) (Rep-stone-phi-pair y) else triple.pairs-inf (?q f x) (?q f y))
  by (simp add: if-distrib-2)
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then triple.pairs-inf (Rep-stone-phi-pair x) (Rep-stone-phi-pair y) else ?q f (x ∩ y))
  using / by meson
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair (x ∩ y) else ?q f (x ∩ y))
  by (metis inf-stone-phi-pair.rep-eq)
also have ... = stone-phi-embed (x ∩ y)
  by simp
finally show stone-phi-embed (x ∩ y) = stone-phi-embed x ∩ stone-phi-embed y
  by simp
qed
next
have stone-phi-embed (top::'a stone-phi-pair) = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair top else triple.pairs-uminus (Rep-phi f) (triple.pairs-uminus (Rep-phi f) (Rep-stone-phi-pair top)))
  by simp
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then (top,bot) else triple.pairs-uminus (Rep-phi f) (triple.pairs-uminus (Rep-phi f) (top,bot)))
  by (metis (no-types, hide-lams) bot-filter.abs-eq top-stone-phi-pair.rep-eq)
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then (top,bot) else triple.pairs-uminus (Rep-phi f) (bot,top))
  by (metis (no-types, hide-lams) dense-closed-top simp-phi triple.pairs-uminus.simps triple-def)
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then (top,bot) else (top,bot))
  by (metis (no-types, hide-lams) p-bot simp-phi triple.pairs-uminus.simps triple-def)
also have ... = Abs-lifted-pair (λf . (top,Abs-filter {top}))
  by (simp add: bot-filter.abs-eq)
also have ... = top
  by (rule top-lifted-pair.abs-eq[THEN sym])
finally show stone-phi-embed (top::'a stone-phi-pair) = top
  .
next
have stone-phi-embed (bot::'a stone-phi-pair) = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair bot else triple.pairs-uminus (Rep-phi f) (triple.pairs-uminus (Rep-phi f) (Rep-stone-phi-pair bot)))
  by simp
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then (bot,top) else triple.pairs-uminus (Rep-phi f)) (triple.pairs-uminus (Rep-phi f) (bot,top))
  by (metis (no-types, hide-lams) top-filter.abs-eq bot-stone-phi-pair rep-eq)
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then (bot,top) else triple.pairs-uminus (Rep-phi f) (top,bot))
  by (metis (no-types, hide-lams) p-bot simp-phi triple.pairs-uminus.simps triple-def)
also have ... = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then (bot,top) else (bot,top))
  by (metis (no-types, hide-lams) p-top simp-phi triple.pairs-uminus.simps triple-def)
also have ... = Abs-lifted-pair (λf . (bot,Abs-filter UNIV))
  by (simp add: top-filter.abs-eq)
also have ... = bot
  by (rule bot-lifted-pair.abs-eq[THEN sgm])
finally show stone-phi-embed (bot::'a stone-phi-pair) = bot
.
next
let ?q = λf . triple.pairs-uminus (Rep-phi f)
let ?pp = λf x . ?p f (?p f x)
let ?q = λf x . ?pp f (Rep-stone-phi-pair x)
show ∀ x::'a stone-phi-pair . stone-phi-embed (−x) = −stone-phi-embed x 
proof (intro allI)
  fix x :: 'a stone-phi-pair
  have 1: ∀ f . triple.pairs-uminus (Rep-phi f) (?q f x) = ?q f (−x)
proof
  fix f :: ('a regular,'a dense) phi
  let ?r = Rep-phi f
  obtain x1 x2 where 2: (x1,x2) = Rep-stone-phi-pair x
    using prod-collapse by blast
  hence triple.pairs-uminus (Rep-phi f) (?q f x) = triple.pairs-uminus (Rep-phi f) (?pp f (x1,x2))
by simp
also have ... = triple.pairs-uminus (Rep-phi f) (−x1,?r (−x1))
  by (simp add: triple.pairs-uminus.simps triple-def)
also have ... = (−−−x1,?r (−−−x1))
  by (simp add: triple.pairs-uminus.simps triple-def)
also have ... = ?pp f (−x1,stone-phi x1)
  by (simp add: triple.pairs-uminus.simps triple-def)
also have ... = ?pp f (triple.pairs-uminus stone-phi (x1,x2))
  by simp
also have ... = ?q f (−x)
  using 2 by (simp add: uminus-stone-phi-pair.rep-eq)
finally show triple.pairs-uminus (Rep-phi f) (?q f x) = ?q f (−x)
  qed
have −stone-phi-embed x = −Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair x else ?q f x)
  by simp

82
also have ... = Abs-lifted-pair (λf. triple.pairs-uminus (Rep-phi f)) (if Rep-phi f = stone-phi then Rep-stone-phi-pair x else ?q f x)) by (rule uminus-lifted-pair.abs-eq) (simp-all add: eq-onp-same-args stone-phi-embed-triple-pair)
also have ... = Abs-lifted-pair (λf. if Rep-phi f = stone-phi then triple.pairs-uminus (Rep-phi f) (Rep-stone-phi-pair x) else triple.pairs-uminus (Rep-phi f) (?q f x))
by (simp add: if-distrib)
also have ... = Abs-lifted-pair (λf. if Rep-phi f = stone-phi then Rep-stone-phi-pair x else ?q f x)
using 1 by meson
also have ... = stone-phi-embed (−x)
by simp
finally show stone-phi-embed (−x) = −stone-phi-embed x by simp
qed
next
let ?p = λf. triple.pairs-uminus (Rep-phi f)
let ?pp = λf x. ?p f (?p f x)
let ?q = λf x. ?pp f (Rep-stone-phi-pair x)
show ∀ x y::′a stone-phi-pair. x ≤ y −→ stone-phi-embed x ≤ stone-phi-embed y
proof (intro allI, rule impI)
fix x y :: ′a stone-phi-pair
assume 1: x ≤ y
have ∀ f. triple.pairs-less-eq (if Rep-phi f = stone-phi then Rep-stone-phi-pair x else ?q f x) (if Rep-phi f = stone-phi then Rep-stone-phi-pair y else ?q f y)
proof
fix f :: (′a regular, ′a dense) phi
let ?r = Rep-phi f
obtain x1 x2 y1 y2 where 2: (x1,x2) = Rep-stone-phi-pair x ∧ (y1,y2) = Rep-stone-phi-pair y
using prod.collapse by blast
have x1 ≤ y1
using 1 2 by (metis less-eq-stone-phi-pair.rep-eq stone-phi.pairs-less-eq.simps)
hence −−x1 ≤ −−y1 ∧ ?r (−y1) ≤ ?r (−x1)
by (metis compl-le-compl-iff le-iff-sup simp-phi)
hence triple.pairs-less-eq (−−x1,?r (−x1)) (−−y1,?r (−y1))
by simp
hence triple.pairs-less-eq (?pp f (x1,x2)) (?pp f (y1,y2))
by (simp add: triple.pairs-uminus.simps triple-def)
hence triple.pairs-less-eq (?q f x) (?q f y)
using 2 by simp
hence if ?r = stone-phi then triple.pairs-less-eq (Rep-stone-phi-pair x) (Rep-stone-phi-pair y) else triple.pairs-less-eq (?q f x) (?q f y)

83
using 1 by (simp add: less-eq-stone-phi-pair.rep-eq)
thus triple.pairs-less-eq (if ?r = stone-phi then Rep-stone-phi-pair x else ?q f x) (if ?r = stone-phi then Rep-stone-phi-pair y else ?q f y)
by (simp add: if-distrib-2)
qed

hence Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair x else ?q f x) ≤ Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair y else ?q f y)
by (subst less-eq-lifted-pair.abs-eq) (simp-all add: eq-onp-same-args)

thus stone-phi-embed x ≤ stone-phi-embed y
by simp
qed

The following lemmas show that the embedding is injective and reflects the order. The latter allows us to easily inherit properties involving inequalities from the target of the embedding, without transforming them to equations.

lemma stone-phi-embed-injective:
inj stone-phi-embed
proof (rule injI)
fix x y :: 'a stone-phi-pair
have 1: Rep-phi (Abs-phi stone-phi) = stone-phi
  by (simp add: Abs-phi-inverse stone-phi.hom)
assume 2: stone-phi-embed x = stone-phi-embed y
have ∀ t : 'a stone-phi-pair . Rep-lifted-pair (stone-phi-embed x) = (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair x else triple.pairs-uminus (Rep-phi f) (triple.pairs-uminus (Rep-phi y)))
  by (simp add: Abs-lifted-pair-inverse stone-phi-embed-triple-pair)
thus Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair x else triple.pairs-uminus (Rep-phi f) (Rep-stone-phi-pair y))) = Abs-lifted-pair (λf . if Rep-phi f = stone-phi then Rep-stone-phi-pair y else triple.pairs-uminus (Rep-phi f) (Rep-stone-phi-pair x)))
using 2 by metis
hence Rep-stone-phi-pair x = Rep-stone-phi-pair y
using 1 by metis
thus x = y
by (simp add: Rep-stone-phi-pair-inject)
qed

lemma stone-phi-embed-order-injective:
assumes stone-phi-embed x ≤ stone-phi-embed y
shows x ≤ y
proof
let ?f = Abs-phi stone-phi
have ∀ f . triple.pairs-less-eq (if Rep-phi f = stone-phi then Rep-stone-phi-pair x else triple.pairs-uminus (Rep-phi f) (triple.pairs-uminus (Rep-phi f))

\[(\text{Rep-stone-phi-pair}\ x))\] (if \(\text{Rep-phi}\ f = \text{stone-phi}\) then \(\text{Rep-stone-phi-pair}\ y\) else \(\text{triple.pairs-uminus}\ (\text{Rep-phi}\ f)\) (\(\text{triple.pairs-uminus}\ (\text{Rep-phi}\ f)\) (\(\text{Rep-stone-phi-pair}\ y\))))

**using** **assms** by (\(\text{subst less-eq-lifted-pair.abs-eq[THEN sym]}\)) (\(\text{simp-all add: eq-onp-same-args stone-phi-embed-triple-pair}\)

**hence** \(\text{triple.pairs-less-eq}\) (if \(\text{Rep-phi}\ ?f = (\text{stone-phi}::'a \text{regular} \Rightarrow 'a \text{dense-filter})\) then \(\text{Rep-stone-phi-pair}\ x\) else \(\text{triple.pairs-uminus}\ (\text{Rep-phi}\ ?f)\) (\(\text{triple.pairs-uminus}\ (\text{Rep-phi}\ ?f)\) (\(\text{Rep-stone-phi-pair}\ x)\))) (if \(\text{Rep-phi}\ ?f = (\text{stone-phi}::'a \text{regular} \Rightarrow 'a \text{dense-filter})\) then \(\text{Rep-stone-phi-pair}\ y\) else \(\text{triple.pairs-uminus}\ (\text{Rep-phi}\ ?f)\) (\(\text{triple.pairs-uminus}\ (\text{Rep-phi}\ ?f)\) (\(\text{Rep-stone-phi-pair}\ y)\)))

**by** \(\text{simp}\)

**hence** \(\text{triple.pairs-less-eq}\) (\(\text{Rep-stone-phi-pair}\ x)\) (\(\text{Rep-stone-phi-pair}\ y)\)

**by** (\(\text{simp add: Abs-phi-inverse stone-phi.hom}\)

**thus** \(x \leq y\)

**by** (\(\text{simp add: less-eq-stone-phi-pair.rep-eq}\)

**qed**

Now all Stone algebra axioms can be inherited using the embedding. This is due to the fact that the axioms are universally quantified equations or conditional equations (or inequalities); this is called a quasivariety in universal algebra. It would be useful to have this construction available for arbitrary quasivarieties.

**instantiation** **stone-phi-pair ::** (\(\text{non-trivial-stone-algebra}\) **stone-algebra**

**begin**

**instance**

**apply** intro-classes

**apply** (\(\text{simp add: less-stone-phi-pair.rep-eq less-eq-stone-phi-pair.rep-eq}\)

**apply** (\(\text{simp add: stone-phi-embed-order-injective}\)

**apply** (\(\text{meson order.trans stone-phi-embed-homomorphism stone-phi-embed-order-injective}\)

**apply** (\(\text{meson stone-phi-embed-homomorphism antisym stone-phi-embed-injective injD}\)

**apply** (\(\text{metis inf.sap-ge1 stone-phi-embed-homomorphism stone-phi-embed-order-injective}\)

**apply** (\(\text{metis inf.sap-ge2 stone-phi-embed-homomorphism stone-phi-embed-order-injective}\)

**apply** (\(\text{metis inf-greatest stone-phi-embed-homomorphism stone-phi-embed-order-injective}\)

**apply** (\(\text{metis stone-phi-embed-homomorphism stone-phi-embed-order-injective sup-ge1}\)

**apply** (\(\text{metis stone-phi-embed-homomorphism stone-phi-embed-order-injective sup.cobounded2}\)

**apply** (\(\text{metis stone-phi-embed-homomorphism stone-phi-embed-order-injective sup.least}\)

**apply** (\(\text{metis bot.extremum stone-phi-embed-homomorphism stone-phi-embed-order-injective}\)

**apply** (\(\text{metis stone-phi-embed-homomorphism stone-phi-embed-order-injective}\)

**85**
5.6 Stone Algebra Isomorphism

In this section we prove that the Stone algebra of the triple of a Stone algebra is isomorphic to the original Stone algebra. The following two definitions give the isomorphism.

abbreviation sa-iso-inv :: 'a::non-trivial-stone-algebra stone-phi-pair ⇒ 'a

abbreviation sa-iso :: 'a::non-trivial-stone-algebra ⇒ 'a stone-phi-pair
  where sa-iso ≡ λx. Abs-stone-phi-pair (Abs-regular (−−x),stone-phi (Abs-regular (−x)) ⊔ up-filter (Abs-dense (x ⊔ −x)))

lemma sa-iso-triple-pair:
  (Abs-regular (−−x),stone-phi (Abs-regular (−x)) ⊔ up-filter (Abs-dense (x ⊔ −x))) ∈ triple.pairs stone-phi
  by (metis (mono-tags, lifting) double-compl eq-onp-same-args stone-phi.sa-iso-pair uminus-regular.abs-eq)

lemma stone-phi-inf-dense:
  stone-phi (Abs-regular (−x)) ⊓ up-filter (Abs-dense (y ⊔ −y)) ≤ up-filter (Abs-dense (y ⊔ −y ⊔ x))
  proof −
  have Rep-filter (stone-phi (Abs-regular (−x)) ⊓ up-filter (Abs-dense (y ⊔ −y))) ≤ ↑{(Abs-dense (y ⊔ −y ⊔ x))}
    proof
    fix z :: 'a dense
    let ?r = Rep-dense z
    assume z ∈ Rep-filter (stone-phi (Abs-regular (−x)) ⊓ up-filter (Abs-dense (y ⊔ −y)))
    also have ... = Rep-filter (stone-phi (Abs-regular (−x)) ⊓ up-filter (Abs-dense (y ⊔ −y))) ⊓ Rep-filter
      (up-filter (Abs-dense (y ⊔ −y)))
      by (simp add: inf-filter.rep-eq)
    also have ... = stone-phi-set (Abs-regular (−x)) ⊓ ↑{(Abs-dense (y ⊔ −y))}
      by (metis Abs-filter-inverse mem-Collect-eq up-filter stone-phi-set-filter stone-phi-def)
    finally have −−x ≤ ?r ∧ Abs-dense (y ⊔ −y) ≤ z
      by (metis (mono-tags, lifting) Abs-regular-inverse Int-Collect mem-Collect-eq)
    hence −−x ≤ ?r ∧ y ⊔ −y ≤ ?r
by (simp add: Abs-dense-inverse less-eq-dense.rep-eq)
hence \( y \sqcup -y \sqcup x \leq \mathfrak{r} \)
using order-trans pp-increasing by auto
hence Abs-dense \((y \sqcup -y \sqcup x) \leq \text{Abs-dense } \mathfrak{r}\)
by (subst less-eq-dense.abs-eq) (simp-all add: eq-onp-same-args)
thus \( z \in \uparrow (\text{Abs-dense } (y \sqcup -y \sqcup x)) \)
by (simp add: Rep-dense-inverse)

qed

hence Abs-filter \((\text{Rep-filter } (\text{stone-phi } (\text{Abs-regular } (-x))) \sqcap \text{up-filter} \text{Abs-dense } (y \sqcup -y \sqcup x))\)
by (simp add: eq-onp-same-args less-eq-filter.abs-eq)
thus \( \mathfrak{p} \text{thesis} \)
by (simp add: Rep-filter-inverse)

qed

lemma stone-phi-complement:
\( \text{complement } (\text{stone-phi } (\text{Abs-regular } (-x))) (\text{stone-phi } (\text{Abs-regular } (-x))) \)
by (metis (mono-tags, lifting) eq-onp-same-args stone-phi.phi-complemented uminus-regular.Abs-eq)

lemma up-dense-stone-phi:
\( \text{up-filter } (\text{Abs-dense } (x \sqcup -x)) \leq \text{stone-phi } (\text{Abs-regular } (-x)) \)
proof -
have \( \uparrow (\text{Abs-dense } (x \sqcup -x)) \leq \text{stone-phi-set } (\text{Abs-regular } (-x)) \)
proof
fix \( z :: \text{a dense} \)
let \( \mathfrak{r} = \text{Rep-dense } z \)
assume \( z \in \uparrow (\text{Abs-dense } (x \sqcup -x)) \)
hence \( -\mathfrak{r} \leq \mathfrak{r} \)
by (simp add: Abs-dense-inverse less-eq-dense.rep-eq)
hence \( -\text{Rep-regular } (\text{Abs-regular } (-x)) \leq \mathfrak{r} \)
by (metis (mono-tags, lifting) Abs-regular-inverse mem-Collect-eq)
thus \( z \in \text{stone-phi-set } (\text{Abs-regular } (-x)) \)
by simp

qed
thus \( \mathfrak{p} \text{thesis} \)
by (unfold stone-phi-def, subst less-eq-filter.abs-eq, simp-all add: eq-onp-same-args stone-phi-set-filter)

qed

The following two results prove that the isomorphisms are mutually inverse.

lemma sa-iso-left-invertible:
sa-iso-inv \((\text{sa-iso } z) = x \)
proof -
have \( \text{up-filter } (\text{triple.rho-pair } \text{stone-phi } (\text{Abs-regular } (-x)), \text{stone-phi} (\text{Abs-regular } (-x)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x))) = \text{stone-phi } (\text{Abs-regular } (-x)) \sqcap (\text{stone-phi } (\text{Abs-regular } (-x)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x))) \)
using sa-iso-triple-pair stone-phi.get-rho-pair-char by blast

87
also have \( \vdash \text{stone-phi} (\text{Abs-regular} (-x)) \cap \text{up-filter} (\text{Abs-dense} (x \sqcup -x)) \)
  by (simp add: inf-sup-commute inf-sup-distrib1 stone-phi-complement)
also have \( \vdash \text{up-filter} (\text{Abs-dense} (x \sqcup -x)) \)
  using up-dense-stone-phi inf.absorb2 by auto
finally have 1: \( \text{triple.rho-pair} \text{stone-phi} (\text{Abs-regular} (-x), \text{stone-phi} (\text{Abs-regular} (-x)) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup -x))) = \text{Abs-dense} (x \sqcup -x) \)
  using up-filter-injective by auto
have \( \text{sa-iso-inv} (\text{sa-iso} x) = (\lambda p. \text{Rep-regular} (\text{fst} p) \sqcap \text{Rep-dense} (\text{triple.rho-pair} \text{stone-phi} p)) (\text{Abs-regular} (-x), \text{stone-phi} (\text{Abs-regular} (-x)) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup -x))) \)
  by (simp add: Abs-stone-phi-pair-inverse sa-iso-triple-pair)
also have \( \vdash \text{Abs-regular} (-x) \)
  using 1 by (subst Abs-regular-inverse) simp-all
also have \( \vdash \text{Abs-dense} (x \sqcup -x) \)
  by (subst Abs-dense-inverse) simp-all
also have \( \vdash x \)
  by simp
finally show ?thesis
  by auto
qed

code

lemma sa-iso-right-invertible:
  \( \text{sa-iso} (\text{sa-iso-inv} p) = p \)
proof –
  obtain \( x \ y \) where 1: \( (x,y) = \text{Rep-stone-phi-pair} p \)
  using prod-collapse by blast
hence 2: \( (x,y) \in \text{triple.pairs} \text{stone-phi} \)
  by (simp add: Rep-stone-phi-pair)
hence 3: \( \text{stone-phi} (-x) \leq y \)
  by (simp add: stone-phi.pairs-phi-less-eq)
have 4: \( \forall z. z \in \text{Rep-filter} (\text{stone-phi} x \sqcap y) \rightarrow \neg \text{Rep-regular} x \leq \text{Rep-dense} z \)
proof (rule allI, rule impI)
  fix \( z :: \ 'a dense \)
  let \( ?r = \text{Rep-dense} z \)
  assume \( z \in \text{Rep-filter} (\text{stone-phi} x \sqcap y) \)
  hence \( z \in \text{Rep-filter} (\text{stone-phi} x) \)
  by (simp add: inf-filter.rep-eq)
  also have \( \vdash \text{stone-phi-set} x \)
  by (simp add: stone-phi-def Abs-filter-inverse stone-phi-set-filter)
  finally show \( \neg \text{Rep-regular} x \leq ?r \)
  by simp
qed

have \( \text{triple.rho-pair} \text{stone-phi} (x,y) \in \uparrow (\text{triple.rho-pair} \text{stone-phi} (x,y)) \)
  by simp
also have \( \vdash \text{Rep-filter} (\uparrow (\text{triple.rho-pair} \text{stone-phi} (x,y))) \)
proof (simp add: Abs-filter-inverse)
also have ... = Rep-filter (stone-phi x ∩ y)
  using 2 stone-phi.get-rho-pair-char by fastforce
finally have triple.rho-pair stone-phi (x,y) ∈ Rep-filter (stone-phi x ∩ y)
  by simp
hence 5: --Rep-regular x ≤ Rep-dense (triple.rho-pair stone-phi (x,y))
  using 4 by simp
have 6: sa-iso-inv p = Rep-regular x ∩ Rep-dense (triple.rho-pair stone-phi (x,y))
  using 1 by (metis fstI)
hence --sa-iso-inv p = --Rep-regular x
  by simp
hence sa-iso (sa-iso-inv p) = Abs-stone-phi-pair (Abs-regular (--Rep-regular x),
stone-phi (Abs-regular (--Rep-regular x)) ⊔ up-filter (Abs-dense ((Rep-regular x ∩ Rep-dense (triple.rho-pair stone-phi (x,y))) ⊔ --Rep-regular x)))
  using 6 by simp
also have ... = Abs-stone-phi-pair (x,stone-phi (--x) ⊔ up-filter (Abs-dense (Rep-dense (triple.rho-pair stone-phi (x,y))) ⊔ --Rep-regular x)))
  by (metis mono-tags, lifting) Rep-regular-inverse double-compl
  aminus-regular.rep-eq
also have ... = Abs-stone-phi-pair (x,stone-phi (--x) ⊔ up-filter (Abs-dense (Rep-dense (triple.rho-pair stone-phi (x,y))) ⊔ --Rep-regular x)))
  by (metis inf-sup-aci(5) maddax-3.21-pp simp-regular)
also have ... = Abs-stone-phi-pair (x,stone-phi (--x) ⊔ up-filter (Abs-dense (Rep-dense (triple.rho-pair stone-phi (x,y)))))
  using 5 by (simp add: sup.absorb1)
also have ... = Abs-stone-phi-pair (x,stone-phi (--x) ⊔ up-filter (triple.rho-pair stone-phi (x,y)))
  by (simp add: Abs-stone-phi-pair inverse)
also have ... = Abs-stone-phi-pair (x,stone-phi (--x) ⊔ (stone-phi x ∩ y))
  using 2 stone-phi.get-rho-pair-char by fastforce
also have ... = Abs-stone-phi-pair (x,stone-phi (--x) ⊔ y)
  by (simp add: stone-phi.phi-complemented sup.commute sup-inf-distrib1)
also have ... = Abs-stone-phi-pair (x,y)
  using 3 by (simp add: le-iiff-sup)
also have ... = p
  using 1 by (simp add: Rep-stone-phi-pair-inverse)
finally show ?thesis
qed

It remains to show the homomorphism properties, which is done in the following result.

lemma sa-iso:
  stone-algebra-isomorphism sa-iso
proof (intro conjI)
  have Abs-stone-phi-pair (Abs-regular (--bot),stone-phi (Abs-regular (--bot)) ⊔ up-filter (Abs-dense (bot ∪ --bot)) = Abs-stone-phi-pair (bot,stone-phi top ⊔
up-filter top)  
  by (simp add: bot-regular.abs-eq top-regular.abs-eq top-dense.abs-eq)  
also have ... = Abs-stone-phi-pair (bot,stone-phi top)  
  by (simp add: stone-phi.hom)  
also have ... = bot  
  by (simp add: bot-stone-phi-pair-def stone-phi,phi-top)  
finally show sa-iso bot = bot  
.
next  
have Abs-stone-phi-pair (Abs-regular (−−top),stone-phi (Abs-regular (−top))) ⊔  
up-filter (Abs-dense (top ⊔ −top))) = Abs-stone-phi-pair (top,stone-phi bot ⊔  
up-filter top)  
  by (simp add: bot-regular.abs-eq top-regular.abs-eq top-dense.abs-eq)  
also have ... = top  
  by (simp add: stone-phi,phi-bot top-stone-phi-pair-def)  
finally show sa-iso top = top  
.
next  
have 1: ∀x y: 'a . dense (x ⊔ −x ⊔ y)  
  by simp  

have 2: ∀x y: 'a . up-filter (Abs-dense (x ⊔ −x ⊔ y)) ≤ (stone-phi  
(Abs-regular (−x)) ⊔ up-filter (Abs-dense (x ⊔ −x))) ∩ (stone-phi (Abs-regular  
(−y)) ⊔ up-filter (Abs-dense (y ⊔ −y)))  
proof (intro allI)  
  fix x y :: 'a  
  let ?u = Abs-dense (x ⊔ −x ⊔ −−y)  
  let ?v = Abs-dense (y ⊔ −y)  
  have {(Abs-dense (x ⊔ −x ⊔ y))} ≤ Rep-filter (stone-phi (Abs-regular (−y))  
⊔ up-filter ?v)  
  proof  
  fix z  
  assume z ∈ ?{Abs-dense (x ⊔ −x ⊔ y)}  
  hence Abs-dense (x ⊔ −x ⊔ y) ≤ z  
    by simp  
  hence 3: x ⊔ −x ⊔ y ≤ Rep-dense z  
    by (simp add: Abs-dense-inverse less-eq-dense.rep-eq)  
  have y ≤ x ⊔ −x ⊔ −−y  
    by (simp add: le-supI2 pp-increasing)  
  hence (x ⊔ −x ⊔ −−y) ∩ (y ⊔ −y) = y ⊔ ((x ⊔ −x ⊔ −−y) ∩ −y)  
    by (simp add: le-iff-sup sup-inf-distrib1)  
  also have ... = y ⊔ ((x ⊔ −x) ∩ −y)  
    by (simp add: inf-commute inf-sup-distrib1)  
  also have ... ≤ Rep-dense z  
    using 3 by (meson le-infI1 sup.bounded-iff)  
  finally have Abs-dense ((x ⊔ −x ⊔ −−y) ∩ (y ⊔ −y)) ≤ z  
    by (simp add: Abs-dense-inverse less-eq-dense.rep-eq)  
  hence 4: ?u ∩ ?v ≤ z  
    by (simp add: eq-onp-same-args inf-dense.abs-eq)  
  have −Rep-regular (Abs-regular (−y)) = −−y
by (metis (mono-tags, lifting) mem-Collect-eq Abs-regular-inverse)
also have ... ≤ Rep-dense ?u
  by (simp add: Abs-dense-inverse)
finally have ?u ∈ stone-phi-set (Abs-regular (−y))
  by simp
hence 5: ?u ∈ Rep-filter (stone-phi (Abs-regular (−y)))
  by (metis mem-Collect-eq stone-phi-def stone-phi-set-filter)
Abs-filter-inverse
have ?v ∈ ∨ ?v
  by simp
hence ?v ∈ Rep-filter (up-filter ?v)
  by (metis Abs-filter-inverse mem-Collect-eq up-filter)
thus z ∈ Rep-filter (stone-phi (Abs-regular (−y)) ⊔ up-filter ?v)
  using 4 5 sup-filter.rep-eq by blast
hence up-filter (Abs-dense (x ⊔ −x ⊔ y)) ≤ Abs-filter (Rep-filter (stone-phi (Abs-regular (−y)) ⊔ up-filter ?v))
  by (simp add: eq-onp-same-args less-eq-filter.abs-eq)
also have ... = stone-phi (Abs-regular (−y)) ⊔ up-filter ?v
  by (simp add: Rep-filter-inverse)
finally show up-filter (Abs-dense (x ⊔ −x ⊔ y)) ≤ (stone-phi (Abs-regular (−x)) ⊔ up-filter (Abs-dense (x ⊔ −x))) ∩ (stone-phi (Abs-regular (−y)) ⊔ up-filter (Abs-dense (y ⊔ −y)))
  by (metis le-infl le-supI2 sup-bot.right-neutral up-filter-dense-antitone)
qed
have 6: ∀ x::'a . in-p-image (−x)
  by auto
show ∀ x y::'a . sa-iso (x ⊔ y) = sa-iso x ⊔ sa-iso y
proof (intro allI)
  fix x y :: 'a
have 7: up-filter (Abs-dense (x ⊔ −x)) ∩ up-filter (Abs-dense (y ⊔ −y)) ≤ up-filter (Abs-dense (y ⊔ −y ⊔ x))
  proof –
  have up-filter (Abs-dense (x ⊔ −x)) ∩ up-filter (Abs-dense (y ⊔ −y)) = up-filter (Abs-dense (x ⊔ −x) ∩ Abs-dense (y ⊔ −y))
    by (metis up-filter-dist-sup)
  also have ... = up-filter (Abs-dense (x ⊔ −x ∪ (y ⊔ −y)))
    by (subst sup-dense.abs-eq) simp-add: eq-onp-same-args)
  also have ... = up-filter (Abs-dense (y ⊔ −y ∪ x ⊔ −x))
    by (simp-add: sup-commute sup-left-commute)
  also have ... ≤ up-filter (Abs-dense (y ⊔ −y ⊔ x))
    using up-filter-dense-antitone by auto
finally show ?thesis
  qed
have Abs-dense (x ⊔ y ⊔ −(x ⊔ y)) = Abs-dense ((x ⊔ −x ⊔ y) ∩ (y ⊔ −y ⊔ x))
  by (simp add: sup-commute sup-inf-distri1 sup-left-commute)
also have ... = Abs-dense (x ⊔ −x ⊔ y) ⊔ Abs-dense (y ⊔ −y ⊔ x)
using I by (metis (mono-tags, lifting) Abs-dense-inverse Rep-dense-inverse inf-dense_rep-eq mem-Collect-eq)

finally have 8: up-filter (Abs-dense \((x \sqcup y \sqcup -(x \sqcup y))\)) = up-filter (Abs-dense \((x \sqcup -x \sqcup y)\)) \sqcup up-filter (Abs-dense \((y \sqcup -y \sqcup x)\))
by (simp add: up-filter-dist-inf)

also have ... \(\leq (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup -x))) \sqcap (\text{stone-phi} (\text{Abs-regular} (-y)) \sqcup \text{up-filter} (\text{Abs-dense} (y \sqcup -y)))\)

using 2 by (simp add: inf-sup-commute le-supI)

finally have 9: (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcap \text{stone-phi} (\text{Abs-regular} (-y))) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup -x \sqcup y)) \leq ...
by (simp add: le-supI)

have ... = (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcap \text{stone-phi} (\text{Abs-regular} (-y))) \sqcup (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcap \text{up-filter} (\text{Abs-dense} (y \sqcup -y))) \sqcup (\text{up-filter} (\text{Abs-dense} (x \sqcup -x)) \sqcap \text{up-filter} (\text{Abs-dense} (y \sqcup -y)))
by (metis (no-types) inf-sup-distrib1 inf-sup-distrib2)

also have ... \(\leq (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcap \text{stone-phi} (\text{Abs-regular} (-y))) \sqcup \text{up-filter} (\text{Abs-dense} (y \sqcup -y \sqcup x)) \sqcup ((\text{up-filter} (\text{Abs-dense} (x \sqcup -x)) \sqcap \text{stone-phi} (\text{Abs-regular} (-y))) \sqcup (\text{up-filter} (\text{Abs-dense} (x \sqcup -x)) \sqcap \text{up-filter} (\text{Abs-dense} (y \sqcup -y))))\)

by (meson sup-left-isotone sup-right-isotone stone-phi-inf-dense)

also have ... \(\leq (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcap \text{stone-phi} (\text{Abs-regular} (-y))) \sqcup \text{up-filter} (\text{Abs-dense} (y \sqcup -y \sqcup x)) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup -x \sqcup y)) \sqcup (\text{up-filter} (\text{Abs-dense} (x \sqcup -x \sqcup y)) \sqcap \text{up-filter} (\text{Abs-dense} (y \sqcup -y)))\)
by (metis inf-commute sup-left-isotone sup-right-isotone stone-phi-inf-dense)

also have ... = (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcap \text{stone-phi} (\text{Abs-regular} (-y))) \sqcup \text{up-filter} (\text{Abs-dense} (y \sqcup -y \sqcup x)) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup -x \sqcup y))
using 7 by (simp add: sup.absorb1 sup-commute sup-left-commute)

also have ... = (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcap \text{stone-phi} (\text{Abs-regular} (-y))) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup -x \sqcup y)) \sqcup \text{up-filter} (\text{Abs-dense} (y \sqcup -y \sqcup x))
using 8 by (simp add: sup.commute sup.left-commute)

finally have (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup -x))) \sqcap (\text{stone-phi} (\text{Abs-regular} (-y)) \sqcup \text{up-filter} (\text{Abs-dense} (y \sqcup -y))) = ...
using 9 using antisym by blast

also have ... = \text{stone-phi} (\text{Abs-regular} (\neg x) \sqcap \text{Abs-regular} (-y)) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup y \sqcup -(x \sqcup y)))
by (simp add: stone-phi.hom)

also have ... = \text{stone-phi} (\text{Abs-regular} (-x \sqcup y)) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup y - (x \sqcup y)))
using 6 by (subst inf-regular.abs-eq) (simp add: eq-onp-same-args)

finally have 10: \text{stone-phi} (\text{Abs-regular} (-(x \sqcup y))) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup y - (x \sqcup y))) = (\text{stone-phi} (\text{Abs-regular} (\neg x)) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup -x))) \sqcap (\text{stone-phi} (\text{Abs-regular} (-y)) \sqcup \text{up-filter} (\text{Abs-dense} (y \sqcup -y)))
by simp

have Abs-regular (-(x \sqcup y)) = Abs-regular (-(x \sqcup y) \sqcup \text{Abs-regular} (-y))
using 6 by (subst sup-regular.abs-eq) (simp add: eq-onp-same-args)

hence Abs-stone-phi-pair (\text{Abs-regular} (-(x \sqcup y)), \text{stone-phi} (\text{Abs-regular} (-y))) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup y - (x \sqcup y))) = \text{Abs-stone-phi-pair} (\text{triple.pairs-sup} (\text{Abs-regular} (\neg x), \text{stone-phi} (\text{Abs-regular} (\neg x))) \sqcup \text{up-filter} (\text{Abs-dense} (x \sqcup y - (x \sqcup y))))
\[(\text{Abs-dense } (x \sqcup -x)) (\text{Abs-regular } (-y), \text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (y \sqcup -y)))\]

using 10 by auto

also have \(\ldots = \text{Abs-stone-phi-pair } (\text{Abs-regular } (-x), \text{stone-phi } (\text{Abs-regular } (-x)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x)) \sqcup \text{Abs-stone-phi-pair } (\text{Abs-regular } (-y), \text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (y \sqcup -y)))\)

by (rule sup-stone-phi-pair.abs-eq[THEN sym]) (simp-all add: eq-opn-same-args sa-iso-triple-pair)

finally show \(\text{sa-iso } (x \sqcup y) = \text{sa-iso } x \sqcup \text{sa-iso } y\)

qed

next

have 1: \(\forall x y . \ldotp \text{a} . \text{dense } (x \sqcup -x \sqcup y)\)

by simp

have 2: \(\forall x . \ldotp \text{a} . \text{in-p-image } (-x)\)

by auto

have 3: \(\forall x y . \ldotp \text{a} . \text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x)) = \text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x \sqcup -y))\)

proof (intro allI)

fix \(x y :: \ldotp \text{a}\)

have 4: \(\text{up-filter } (\text{Abs-dense } (x \sqcup -x)) \leq \text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x \sqcup -y))\)

by (metis (no-types, lifting) complement-shunting stone-phi-inf-dense stone-phi-complement-complement-symmetric)

have up-filter \((\text{Abs-dense } (x \sqcup -x \sqcup -y)) \leq \text{up-filter } (\text{Abs-dense } (x \sqcup -x))\)

by (metis sup-idem up-filter-dense-antitone)

thus \(\text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x)) = \text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x \sqcup -y))\)

using 4 by (simp add: le-iff-sup sup-commute sup-left-commute)

qed

show \(\forall x y . \ldotp \text{a} . \text{sa-iso } (x \sqcap y) = \text{sa-iso } x \sqcap \text{sa-iso } y\)

proof (intro allI)

fix \(x y :: \ldotp \text{a}\)

have Abs-dense \((x \sqcap y) \sqcup -(x \sqcap y)) = Abs-dense \((x \sqcup -x \sqcup -y) \sqcup (y \sqcup -y \sqcup -x))\)

by (simp add: sup-commute sup-inf-distrib sup-left-commute)

also have \(\ldots = \text{Abs-dense } (x \sqcup -x \sqcup -y) \sqcap \text{Abs-dense } (y \sqcup -y \sqcup -x)\)

using 1 by (metis (mono-tags, lifting) Abs-dense-inverse Rep-dense-inverse inf-dense.rep-eq mem-Collect-eq)

finally have 5: \(\text{up-filter } (\text{Abs-dense } (x \sqcap y) \sqcup -(x \sqcap y)) = \text{up-filter } (\text{Abs-dense } (x \sqcup -x \sqcup -y)) \sqcup \text{up-filter } (\text{Abs-dense } (y \sqcup -y \sqcup -x))\)

by (simp add: up-filter-dist-inf)

have \((\text{stone-phi } (\text{Abs-regular } (-x)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x))) \sqcup (\text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (y \sqcup -y))) = (\text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x))) \sqcup (\text{stone-phi } (\text{Abs-regular } (-x)) \sqcup \text{up-filter } (\text{Abs-dense } (y \sqcup -y)))\)

by (simp add: inf-sup-aci(6) sup-left-commute)

also have \(\ldots = (\text{stone-phi } (\text{Abs-regular } (-y)) \sqcup \text{up-filter } (\text{Abs-dense } (x \sqcup -x \sqcup -y))) \sqcup (\text{stone-phi } (\text{Abs-regular } (-x)) \sqcup \text{up-filter } (\text{Abs-dense } (y \sqcup -y \sqcup -x)))\)
using 3 by simp
also have ... = (stone-phi (Abs-regular (−x)) ∪ stone-phi (Abs-regular (−y)))
∪ (up-filter (Abs-dense (x ∩ −x ∩ −y)) ∪ up-filter (Abs-dense (y ∪ −y ∪ −x)))
by (simp add: inf-sup-aci(6) sup-left-commute)
also have ... = (stone-phi (Abs-regular (−x)) ∪ stone-phi (Abs-regular (−y)))
∪ up-filter (Abs-dense ((x ∩ y) ∪ −(x ∩ y)))
using 5 by (simp add: sup.commute sup.left-commute)
finally have (stone-phi (Abs-regular (−x) ∪ up-filter (Abs-dense (x ∩ −x)))
∪ (stone-phi (Abs-regular (−y)) ∪ up-filter (Abs-dense (y ∪ −y))) = ...
by simp
also have ... = stone-phi (Abs-regular (−(x ∩ y))) ∪ up-filter (Abs-dense ((x ∩ y) ∪ −(x ∩ y)))
by (simp add: stone-phi.hom)
also have ... = stone-phi (Abs-regular (−(x ∩ y))) ∪ up-filter (Abs-dense ((x ∩ y) ∪ −(x ∩ y)))
using 2 by (subst sup-regular.abs-eq) (simp-all add: eq-onp-same-args)
finally have 6: stone-phi (Abs-regular (−(x ∩ y))) ∪ up-filter (Abs-dense ((x ∩ y) ∪ −(x ∩ y))) = stone-phi (Abs-regular (−x)) ∪ up-filter (Abs-dense (x ∩ −x))) ∪ (stone-phi (Abs-regular (−y)) ∪ up-filter (Abs-dense (y ∪ −y)))
by simp
have Abs-regular (−(x ∩ y)) = Abs-regular (−(x ∩ y) ∩ Abs-regular (−y))
using 2 by (subst inf-regular.abs-eq) (simp-all add: eq-onp-same-args)

hence Abs-stone-phi-pair (Abs-regular (−(x ∩ y)),stone-phi (Abs-regular (−(x ∩ y)) ∪ up-filter (Abs-dense ((x ∩ y) ∪ −(x ∩ y))))) = Abs-stone-phi-pair (Abs-regular (−x),stone-phi (Abs-regular (−x)) ∪ up-filter (Abs-dense (x ∩ −x))) (Abs-regular (−y),stone-phi (Abs-regular (−y)) ∪ up-filter (Abs-dense (y ∪ −y))))
using 6 by auto
also have ... = Abs-stone-phi-pair (Abs-regular (−x),stone-phi (Abs-regular (−x)) ∪ up-filter (Abs-dense (x ∩ −x))) ∩ Abs-stone-phi-pair (Abs-regular (−y),stone-phi (Abs-regular (−y)) ∪ up-filter (Abs-dense (y ∪ −y))))
by (rule inf-stone-phi-pair.abs-eq[THEN sgm]) (simp-all add: eq-onp-same-args sa-iso-triple-pair)
finally show sa-iso (x ∩ y) = sa-iso x ∩ sa-iso y
.

qed
next
show ∀ x::'a . sa-iso (−x) = −sa-iso x
proof
fix x :: 'a
have sa-iso (−x) = Abs-stone-phi-pair (Abs-regular (−−x),stone-phi (Abs-regular (−−x))) ∪ up-filter top)
by (simp add: top-dense-def)
also have ... = Abs-stone-phi-pair (Abs-regular (−−x),stone-phi (Abs-regular (−−x))
by (metis bot-filter.abs-eq sup-bot.right-neutral up-top)
also have ... = Abs-stone-phi-pair (triple.pairs-uminus stone-phi (Abs-regular (−x),stone-phi (Abs-regular (−x)) ∪ up-filter (Abs-dense (x ∩ −x))))
by (subst uminus-regular.abs-eq[THEN sgm], unfold eq-onp-same-args) auto

94
also have \( \cdots = -sa-iso x \)
by (simp add: eq-onp-def sa-iso-triple-pair uminus-stone-phi-pair.abs-eq)
finally show \( sa-iso \ (-x) = -sa-iso \ x \)
by simp
qed

next
show bij sa-iso
by (metis (mono-tags, lifting) sa-iso-left-invertible sa-iso-right-invertible
invertible-bij[where \( g=sa-iso-inv \)])
qed

5.7 Triple Isomorphism
In this section we prove that the triple of the Stone algebra of a triple is
isomorphic to the original triple. The notion of isomorphism for triples is
described in [7]. It amounts to an isomorphism of Boolean algebras, an iso-
morphism of distributive lattices with a greatest element, and a commuting
diagram involving the structure maps.

5.7.1 Boolean Algebra Isomorphism
We first define and prove the isomorphism of Boolean algebras. Because
the Stone algebra of a triple is implemented as a lifted pair, we also lift the
Boolean algebra.

typedef (overloaded) \( ('a', 'b') \) lifted-boolean-algebra =
\{ xf::('a::non-trivial-boolean-algebra,'b::distrib-lattice-top) phi \Rightarrow 'a . True \}
by simp

setup-lifting type-definition-lifted-boolean-algebra

instantiation lifted-boolean-algebra ::
(non-trivial-boolean-algebra,distrib-lattice-top) boolean-algebra
begin

lift-definition sup-lifted-boolean-algebra :: ('a', 'b') lifted-boolean-algebra \Rightarrow ('a', 'b')
lifted-boolean-algebra is \( \lambda xf yf . sup (xf f) (yf f) \).

lift-definition inf-lifted-boolean-algebra :: ('a', 'b') lifted-boolean-algebra \Rightarrow ('a', 'b')
lifted-boolean-algebra is \( \lambda xf yf . inf (xf f) (yf f) \).

lift-definition minus-lifted-boolean-algebra :: ('a', 'b') lifted-boolean-algebra \Rightarrow
('a', 'b') lifted-boolean-algebra is \( \lambda xf yf . minus (xf f) (yf f) \).

lift-definition uminus-lifted-boolean-algebra :: ('a', 'b') lifted-boolean-algebra \Rightarrow
('a', 'b') lifted-boolean-algebra is \( \lambda xf f . uminus (xf f) \).
lift-definition bot-lifted-boolean-algebra :: ('a,'b) lifted-boolean-algebra is λf . bot

lift-definition top-lifted-boolean-algebra :: ('a,'b) lifted-boolean-algebra is λf . top

lift-definition less-eq-lifted-boolean-algebra :: ('a,'b) lifted-boolean-algebra ⇒ ('a,'b) lifted-boolean-algebra ⇒ bool is λxf yf . ∀f . less-eq (xf f) (yf f).

lift-definition less-lifted-boolean-algebra :: ('a,'b) lifted-boolean-algebra ⇒ ('a,'b) lifted-boolean-algebra ⇒ bool is λxf yf . (∀f . less-eq (xf f) (yf f)) ∧ ¬(∀f . less-eq (yf f) (xf f)).

instance
apply intro-classes
apply (simp add: less-eq-lifted-boolean-algebra.rep-eq
less-lifted-boolean-algebra.rep-eq)
apply (simp add: less-eq-lifted-boolean-algebra.rep-eq
less-lifted-boolean-algebra.rep-eq)
using less-eq-lifted-boolean-algebra.rep-eq order-trans apply fastforce
apply (metis less-eq-lifted-boolean-algebra.rep-eq antisym ext
Rep-lifted-boolean-algebra-inject)
apply (simp add: inf-lifted-boolean-algebra.rep-eq
less-eq-lifted-boolean-algebra.rep-eq)
apply (simp add: inf-lifted-boolean-algebra.rep-eq
less-lifted-boolean-algebra.rep-eq)
apply (simp add: inf-lifted-boolean-algebra.rep-eq
less-lifted-boolean-algebra.rep-eq)
apply (simp add: sup-lifted-boolean-algebra.rep-eq
less-eq-lifted-boolean-algebra.rep-eq
less-eq-lifted-boolean-algebra.rep-eq)
apply (simp add: sup-lifted-boolean-algebra.rep-eq
less-eq-lifted-boolean-algebra.rep-eq
less-lifted-boolean-algebra.rep-eq)
apply (simp add: less-eq-lifted-boolean-algebra.rep-eq
less-eq-lifted-boolean-algebra.rep-eq
less-eq-lifted-boolean-algebra.rep-eq
less-lifted-boolean-algebra.rep-eq)
apply (simp add: less-eq-lifted-boolean-algebra.rep-eq
less-eq-lifted-boolean-algebra.rep-eq
less-lifted-boolean-algebra.rep-eq
less-lifted-boolean-algebra.rep-eq)
apply [unfold Rep-lifted-boolean-algebra-inject[THEN sym]
sup-lifted-boolean-algebra.rep-eq inf-lifted-boolean-algebra.rep-eq, simp add: sup-inf-distrib1]
apply [unfold Rep-lifted-boolean-algebra-inject[THEN sym]
inf-lifted-boolean-algebra.rep-eq uminus-lifted-boolean-algebra.rep-eq
bot-lifted-boolean-algebra.rep-eq, simp]
apply [unfold Rep-lifted-boolean-algebra-inject[THEN sym]
sup-lifted-boolean-algebra.rep-eq uminus-lifted-boolean-algebra.rep-eq
top-lifted-boolean-algebra.rep-eq, simp]
apply by (unfold Rep-lifted-boolean-algebra-inject[THEN sym]
inf-lifted-boolean-algebra.rep-eq uminus-lifted-boolean-algebra.rep-eq
minus-lifted-boolean-algebra.rep-eq, simp add: diff-eq)
The following two definitions give the Boolean algebra isomorphism.

**abbreviation** ba-iso-inv :: \(\langle a::non\text{-}trivial\text{-}boolean\text{-}algebra, b::distrib\text{-}lattice\text{-}top\rangle\)

**lifted-boolean-algebra** \(\Rightarrow\) \(\langle a, b\rangle\)

where ba-iso-inv \(\equiv\) \(\lambda f\). Abs-regular (Abs-lifted-pair \(\lambda f\). (Rep-lifted-boolean-algebra \(xf\ f\), Rep-\(\phi\) \(-\) Rep-lifted-boolean-algebra \(xf\ f\)))

**abbreviation** ba-iso :: \(\langle a::non\text{-}trivial\text{-}boolean\text{-}algebra, b::distrib\text{-}lattice\text{-}top\rangle\)

**lifted-pair** regular \(\Rightarrow\) \(\langle a, b\rangle\) lifted-boolean-algebra

where ba-iso \(\equiv\) \(\lambda pf\). Abs-lifted-boolean-algebra (Abs-lifted-pair (\(\lambda f\). (Rep-regular \(pf\) \(f\)) (Rep-lifted-boolean-algebra \(xf\ f\), Rep-\(\phi\) \(-\) Rep-lifted-boolean-algebra \(xf\ f\))))

**lemma** ba-iso-inv-lifted-pair:

\((\text{Rep-lifted-boolean-algebra } xf f, \text{Rep-\(\phi\)} f (\text{Rep-lifted-boolean-algebra } xf f))\in \text{triple.pairs (Rep-\(\phi\) f)})

by (metis (no-types, hide-lams) double-compl simp-\(\phi\) triple.pairs-uminus.simps triple-def triple.pairs-uminus-closed)

**lemma** ba-iso-inv-regular:

regular (Abs-lifted-pair (\(\lambda f\). (Rep-lifted-boolean-algebra \(xf\ f\), Rep-\(\phi\) f (\(-\) Rep-lifted-boolean-algebra \(xf\ f\)))))

proof –

have \(\forall f\). (\text{Rep-lifted-boolean-algebra } xf f, \text{Rep-\(\phi\)} f (\text{Rep-lifted-boolean-algebra } xf f)) = \text{triple.pairs-uminus} (\text{Rep-\(\phi\)} f) (\text{triple.pairs-uminus} (\text{Rep-\(\phi\)} f))

(Rep-lifted-boolean-algebra \(xf\ f\), Rep-\(\phi\) f (\(-\) Rep-lifted-boolean-algebra \(xf\ f\)))

by (simp add: triple.pairs-uminus.simps triple-def)

hence Abs-lifted-pair (\(\lambda f\). (Rep-lifted-boolean-algebra \(xf\ f\), Rep-\(\phi\) f (\(-\) Rep-lifted-boolean-algebra \(xf\ f\)))) = \(-\) Abs-lifted-pair (\(\lambda f\). (Rep-lifted-boolean-algebra \(xf\ f\), Rep-\(\phi\) f (\(-\) Rep-lifted-boolean-algebra \(xf\ f\))))

by (simp add: triple.pairs-uminus-closed triple-def eq-onp-def)

uminus-lifted-pair, abs-eq ba-iso-inv-lifted-pair)

thus \(?\)thesis

by simp

qed

The following two results prove that the isomorphisms are mutually inverse.

**lemma** ba-iso-left-invertible:

ba-iso-inv (ba-iso \(pf\)) = \(pf\)

proof –

have \(\forall f\). \text{snd} (\text{Rep-lifted-pair} (\text{Rep-regular} \(pf\) \(f\)) = \text{Rep-\(\phi\)} f (\(-\text{fst}\) (Rep-lifted-pair (\text{Rep-regular} \(pf\) \(f\))))

proof

fix \(f\) :: \(\langle a, b\rangle\) \(\phi\)

let \(?r\) = Rep-\(\phi\) \(f\)

have triple \(?r\)

by (simp add: triple-def)
hence 2: ∀p. triple.pairs-uminus ?r p = (¬fst p, ?r (fst p))
by (metis prod.collapse triple.pairs-uminus.simps)

have 3: Rep-regular pf = ¬Rep-regular pf
by (simp add: regular-in-p-image-iff)

show snd (Rep-lifted-pair (Rep-regular pf) f) = ?r (¬fst (Rep-lifted-pair (Rep-regular pf) f))
using 2 3 by (metis fstI sndI aminus.lifted-pair.rep-eq)

qed

have ba-iso-inv (ba-iso pf) = Abs-regular (Abs-lifted-pair (λf. (fst (Rep-lifted-pair (Rep-regular pf) f), Rep-phi f (¬fst (Rep-lifted-pair (Rep-regular pf) f)))))
by (simp add: Abs-lifted-boolean-algebra-inverse)
also have ... = Abs-regular (Abs-lifted-pair (Rep-lifted-pair (Rep-regular pf)))
using 1 by (metis prod.collapse)
also have ... = pf
by (simp add: Rep-regular-inverse Abs-lifted-pair-inverse)
finally show thesis
.

qed

lemma ba-iso-right-invertible:
ba-iso (ba-iso-inv xf) = xf

proof —
let ?rf = Rep-lifted-boolean-algebra xf

have 1: ∀f. (¬?rf f, Rep-phi f (?rf f)) ∈ triple.pairs (Rep-phi f) ∧ (?rf f, Rep-phi f (¬?rf f)) ∈ triple.pairs (Rep-phi f)

proof
  fix f
  have up-filter top = bot
    by (simp add: bot-filter.abs-eq)
  hence (∃z. Rep-phi f (?rf f) = Rep-phi f (?rf f) ⊔ up-filter z) ∧ (∃z. Rep-phi f (¬?rf f) = Rep-phi f (¬?rf f) ⊔ up-filter z)
    by (metis sup-bot-right)
  thus (¬?rf f, Rep-phi f (?rf f)) ∈ triple.pairs (Rep-phi f) ∧ (?rf f, Rep-phi f (¬?rf f)) ∈ triple.pairs (Rep-phi f)
    by (simp add: triple-def triple.pairs-def)

qed

have regular (Abs-lifted-pair (λf. (?rf f, Rep-phi f (¬?rf f))))

proof —
  have ¬Abs-lifted-pair (λf. (?rf f, Rep-phi f (¬?rf f))) = ¬Abs-lifted-pair (λf. triple.pairs-uminus (Rep-phi f) (?rf f, Rep-phi f (¬?rf f)))
    using f by (simp add: eq-ontp.same-args aminus.lifted-pair.abs-eq)
  also have ... = ¬Abs-lifted-pair (λf. triple.pairs-uminus (Rep-phi f) (?rf f, Rep-phi f (¬?rf f)))
    by (metis (no-types, lifting) simp-phi triple-def triple.pairs-uminus.simps)
  also have ... = Abs-lifted-pair (λf. triple.pairs-uminus (Rep-phi f) (¬?rf f, Rep-phi f (?rf f)))
    using f by (simp add: eq-ontp.same-args aminus.lifted-pair.abs-eq)
  also have ... = Abs-lifted-pair (λf. (¬?rf f, Rep-phi f (¬?rf f)))
    by (metis (no-types, lifting) simp-phi triple-def triple.pairs-uminus.simps)
double-compl

finally show ?thesis
by simp

qed

hence in-p-image (Abs-lifted-pair (\lambda f. (?rf f, Rep-phi f (- ?rf f))))
by blast

thus ?thesis
using 1 by (simp add: Rep-lifted-boolean-algebra-inverse
Abs-lifted-pair-inverse Abs-regular-inverse)

qed

The isomorphism is established by proving the remaining Boolean alge-
bra homomorphism properties.

lemma ba-iso:
boolean-algebra-isomorphism ba-iso

proof (intro conjI)

  show Abs-lifted-boolean-algebra (\lambda f. fst (Rep-lifted-pair (Rep-regular bot) f)) = bot
  by (simp add: bot-lifted-boolean-algebra-def bot-regular rep-eq bot-lifted-pair rep-eq)

next

  show Abs-lifted-boolean-algebra (\lambda f. fst (Rep-lifted-pair (Rep-regular top) f)) = top
  by (simp add: top-lifted-boolean-algebra-def top-regular rep-eq top-lifted-pair rep-eq)

next

  show \forall pf qf. Abs-lifted-boolean-algebra (\lambda f::('a,'b) phi. fst (Rep-lifted-pair (Rep-regular (pf \sqcup qf) f)) = Abs-lifted-boolean-algebra (\lambda f. fst (Rep-lifted-pair (Rep-regular pf) f)) \sqcup Abs-lifted-boolean-algebra (\lambda f. fst (Rep-lifted-pair (Rep-regular qf) f)))
  proof (intro allI)

    fix pf qf :: ('a,'b) lifted-pair regular
    
    fix f

    obtain x y z w where 1: (x,y) = Rep-lifted-pair (Rep-regular pf) f \land (z,w)
    = Rep-lifted-pair (Rep-regular qf) f
      using prod.collapse by blast

    have triple (Rep-phi f)
      by (simp add: triple-def)

    hence fst (triple.pairs-sup (x,y) (z,w)) = fst (x,y) \sqcup fst (z,w)
      using triple.pairs-sup.simps by force

    hence fst (triple.pairs-sup (Rep-lifted-pair (Rep-regular pf) f) (Rep-lifted-pair (Rep-regular qf) f)) = fst (Rep-lifted-pair (Rep-regular pf) f) \sqcup fst (Rep-lifted-pair (Rep-regular qf) f)
      using 1 by simp

    hence fst (Rep-lifted-pair (Rep-regular (pf \sqcup qf)) f) = fst (Rep-lifted-pair (Rep-regular pf) f) \sqcup fst (Rep-lifted-pair (Rep-regular qf) f)
      by (unfold sup-regular.rep-eq sup-lifted-pair.rep-eq simp)
thus Abs-lifted-boolean-algebra \((\lambda f . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} (pf \sqcap qf)) f))\) = Abs-lifted-boolean-algebra \((\lambda f . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} pf) f))\)
\sqcap Abs-lifted-boolean-algebra \((\lambda f . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} qf) f))\)
by (simp add: eq-opn-same-args sup-lifted-boolean-algebra.abs-eq sup-regular.rep-eq sup-lifted-boolean-algebra.rep-eq)
qed

next
show \(\forall pf qf . \text{Abs-lifted-boolean-algebra} (\lambda f::('a,'b) \text{phi} . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} (pf \sqcap qf)) f)) = \text{Abs-lifted-boolean-algebra} (\lambda f . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} pf) f)) \sqcap \text{Abs-lifted-boolean-algebra} (\lambda f . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} qf) f))\)
proof (intro allI)
fix pf qf :: ('a,'b) lifted-pair regular
{
fix f
obtain x y z w where 1: \((x,y) = \text{Rep-lifted-pair} (\text{Rep-regular} pf) f \land (z,w) = \text{Rep-lifted-pair} (\text{Rep-regular} qf) f\)
using prod.collapse by blast
have triple (Rep-phi f)
by (simp add: triple-def)
  hence \text{fst} (\text{triples-pairs-inf} (x,y) (z,w)) = \text{fst} (x,y) \sqcap \text{fst} (z,w)
using triple.pairs-inf.simps by force
  hence \text{fst} (\text{triples-pairs-inf} (\text{Rep-lifted-pair} (\text{Rep-regular} pf) f)) (\text{Rep-lifted-pair} (\text{Rep-regular} qf) f)) = \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} pf) f) \sqcap \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} qf) f)
  using 1 by simp
  hence \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} (pf \sqcap qf)) f) = \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} pf) f) \sqcap \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} qf) f)
  by (unfold inf-regular.rep-eq inf-lifted-pair.rep-eq simp)
}
thus Abs-lifted-boolean-algebra \((\lambda f . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} (pf \sqcap qf)) f))\) = Abs-lifted-boolean-algebra \((\lambda f . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} pf) f))\)
\sqcap Abs-lifted-boolean-algebra \((\lambda f . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} qf) f))\)
qed

next
show \(\forall pf . \text{Abs-lifted-boolean-algebra} (\lambda f::('a,'b) \text{phi} . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} (\neg pf)) f)) = -\text{Abs-lifted-boolean-algebra} (\lambda f . \text{fst} (\text{Rep-lifted-pair} (\text{Rep-regular} pf) f))\)
proof
fix pf :: ('a,'b) lifted-pair regular
{
fix f
obtain x y where 1: \((x,y) = \text{Rep-lifted-pair} (\text{Rep-regular} pf) f\)
using prod.collapse by blast
have triple (Rep-phi f)
by (simp add: triple-def)
  hence \text{fst} (\text{triples-pairs-uminus} (\text{Rep-phi} f) (x,y)) = -\text{fst} (x,y)
using triple.pairs-uminus.simps by force

hence \( \text{fst} (\text{triple.pairs-uminus} (\text{Rep-phi} f) (\text{Rep.lifted-pair} (\text{Rep-regular} pf) f)) = -\text{fst} (\text{Rep.lifted-pair} (\text{Rep-regular} pf) f) \)

using 1 by simp

hence \( \text{fst} (\text{Rep.lifted-pair} (\text{Rep-regular} (-pf)) f) = -\text{fst} (\text{Rep.lifted-pair} (\text{Rep-regular} pf) f) \)

by (unfold uminus-regular.rep-eq uminus.lifted-pair.rep-eq simp )

\}\ thus \( \text{Abs.lifted-boolean-algebra} (\lambda \phi \Rightarrow \text{fst} (\text{Rep.lifted-pair} (\text{Rep-regular} (-pf)) f)) = -\text{Abs.lifted-boolean-algebra} (\lambda \phi \Rightarrow \text{fst} (\text{Rep.lifted-pair} (\text{Rep-regular} pf) f)) \)


qed

next

show bij ba-iso

by (rule invertible-bij|where g=ba-iso.inv) (simp-all add:

ba.iso-left-invertible ba.iso-right-invertible)

qed

5.7.2 Distributive Lattice Isomorphism

We carry out a similar development for the isomorphism of distributive lattices. Again, the original distributive lattice with a greatest element needs to be lifted to match the lifted pairs.

typedef (overloaded) ('a,'b) lifted-distrib-lattice-top = {

xf::('a::non-trivial-boolean-algebra,'b::distrib-lattice-top) phi => 'b . True }

by simp

setup-lifting type-definition.lifted-distrib-lattice-top

instantiation lifted-distrib-lattice-top :: (non-trivial-boolean-algebra,distrib-lattice-top) distrib-lattice-top

begin

lift-definition sup.lifted-distrib-lattice-top :: ('a,'b) lifted-distrib-lattice-top => ('a,'b) lifted-distrib-lattice-top => ('a,'b) lifted-distrib-lattice-top is \( \lambda xf yf f \ . \ \text{sup} (xf f) (yf f) \).

lift-definition inf.lifted-distrib-lattice-top :: ('a,'b) lifted-distrib-lattice-top => ('a,'b) lifted-distrib-lattice-top => ('a,'b) lifted-distrib-lattice-top is \( \lambda xf yf f \ . \ \text{inf} (xf f) (yf f) \).

lift-definition top.lifted-distrib-lattice-top :: ('a,'b) lifted-distrib-lattice-top is \( \lambda f \ . \ \text{top} \).

lift-definition less-eq.lifted-distrib-lattice-top :: ('a,'b) lifted-distrib-lattice-top => ('a,'b) lifted-distrib-lattice-top => bool is \( \lambda xf yf f \ . \ \forall f . \ \text{less-eq} (xf f) (yf f) \).

lift-definition less.lifted-distrib-lattice-top :: ('a,'b) lifted-distrib-lattice-top =>

101
\( (\alpha, \beta) \text{ lifted-distrib-lattice-top } \Rightarrow \text{ bool is } \lambda x f y f \cdot (\forall f . \text{ less-eq } (x f) (y f)) \land \neg (\forall f . \text{ less-eq } (y f) (x f)) \).

instance
apply intro-classes
apply (simp add: less-eq-lifted-distrib-lattice-top.rep-eq
less-lifted-distrib-lattice-top.rep-eq)
apply (simp add: less-eq-lifted-distrib-lattice-top.rep-eq)
using less-eq-lifted-distrib-lattice-top.rep-eq order-trans apply fastforce
apply (metis less-eq-lifted-distrib-lattice-top.rep-eq antisym ext
Rep-lifted-distrib-lattice-top-inject)
apply (simp add: less-eq-lifted-distrib-lattice-top.rep-eq
less-lifted-distrib-lattice-top.rep-eq)
apply (simp add: less-eq-lifted-distrib-lattice-top.rep-eq
inf-lifted-distrib-lattice-top.rep-eq
rep-eq order-trans apply fastforce
apply (simp add: less-eq-lifted-distrib-lattice-top.rep-eq
less-lifted-distrib-lattice-top.rep-eq)
apply (simp add: less-eq-lifted-distrib-lattice-top.rep-eq
sup-lifted-distrib-lattice-top.rep-eq
rep-eq)
apply (simp add: less-eq-lifted-distrib-lattice-top.rep-eq
top-lifted-distrib-lattice-top.rep-eq)
by (unfold Rep-lifted-distrib-lattice-top-inject[THEN sym]
sup-lifted-distrib-lattice-top.rep-eq inf-lifted-distrib-lattice-top.rep-eq, simp add: sup-inf-distrib1)

end

The following function extracts the least element of the filter of a dense pair, which turns out to be a principal filter. It is used to define one of the isomorphisms below.

fun get-dense :: \(\alpha::\text{non-trivial-boolean-algebra}, \beta::\text{distrib-lattice-top}\) lifted-pair
dense \Rightarrow (\alpha, \beta) phi \Rightarrow \beta
where get-dense pf f = (SOME z . Rep-lifted-pair (Rep-dense pf) f = (top, up-filter z))

lemma get-dense-char:
Rep-lifted-pair (Rep-dense pf) f = (top, up-filter (get-dense pf f))
proof –
obtain x y where 1: (x, y) = Rep-lifted-pair (Rep-dense pf) f \land (x, y) \in triple.pairs (Rep-phi f) \land triple.pairs-uminus (Rep-phi f) (x, y) = triple.pairs-bot
by (metis bot-lifted-pair.rep-eq prod.collapse simp-dense simp-lifted-pair
uminus-lifted-pair.rep-eq)
hence 2: x = top
by (simp add: triple.intro triple.pairs-uminus.simps dense-pp)
have triple (Rep-phi f)

102
by (simp add: triple-def)

hence \exists z. y = \text{Rep-phi } f \ (-x) \sqcup \text{up-filter } z

using 1 triple.pairs-def by blast

then obtain z where y = \text{up-filter } z

using 2 by auto

hence \text{Rep-lifted-pair} (\text{Rep-dense } pf) f = (\text{top, up-filter } z)

using 1 2 by simp

thus \ ?thesis

by (metis (mono-tags, lifting) tfl-some get-dense.simps)

qed

The following two definitions give the distributive lattice isomorphism.

abbreviation dl-iso-inv :: ('a::non-trivial-boolean-algebra, 'b::distrib-lattice-top)
      lifted-distrib-lattice-top \Rightarrow ('a,'b) lifted-pair dense

where dl-iso-inv \equiv \lambda xf. \text{Abs-dense} (\text{Abs-lifted-pair} (\lambda f. (top, up-filter (Rep-lifted-distrib-lattice-top xf f))))

abbreviation dl-iso :: ('a::non-trivial-boolean-algebra, 'b::distrib-lattice-top)
      lifted-pair dense \Rightarrow ('a,'b) lifted-distrib-lattice-top

where dl-iso \equiv \lambda pf. \text{Abs-lifted-distrib-lattice-top} (get-dense pf)

lemma dl-iso-inv-lifted-pair:
\[(\text{top, up-filter (Rep-lifted-distrib-lattice-top xf f)}) \in \text{triple.pairs} (\text{Rep-phi } f)\]

by (metis (no-types, hide-lams) compl-bot-eq double-compl simp-phi sup-bot.
    left-neutral triple.sa-iso-pair triple-def)

lemma dl-iso-inv-dense:
\[\text{dense} (\text{Abs-lifted-pair} (\lambda f. (top, up-filter (Rep-lifted-distrib-lattice-top xf f))))\]

proof

have \(\forall f. \text{triple.pairs-uminus} (\text{Rep-phi } f) (\text{top, up-filter (Rep-lifted-distrib-lattice-top xf f)}) = \text{triple.pairs-bot}\)

by (simp add: top-filter.\text{abs-eq} triple.pairs-uminus.\text{simps} triple-def)

hence bot = \(-\text{Abs-lifted-pair} (\lambda f. (top, up-filter (Rep-lifted-distrib-lattice-top xf f)))\)

by (simp add: \text{eq-onp-def} \text{uminus-lifted-pair.\text{abs-eq} dl-iso-inv-lifted-pair bot-lifted-pair-def})

thus \ ?thesis

by simp

qed

The following two results prove that the isomorphisms are mutually inverse.

lemma dl-iso-left-invertible:
\[\text{dl-iso-inv} (\text{dl-iso} \ pf) = pf\]

proof

have \(\text{dl-iso-inv} (\text{dl-iso} \ pf) = \text{Abs-dense} (\text{Abs-lifted-pair} (\lambda f. (top, up-filter (get-dense pf f))))\)

by (metis \text{Abs-lifted-distrib-lattice-top-inverse} \text{UNIV-I} \text{UNIV-def})

also have ... = \text{Abs-dense} (\text{Abs-lifted-pair} (\text{Rep-lifted-pair} (\text{Rep-dense \ pf})))
by (metis get-dense-char)
also have ... = pf
  by (simp add: Rep-dense-inverse Rep-lifted-pair-inverse)
finally show \(?thesis
.
qed

lemma dl-iso-right-invertible:
  dl-iso (dl-iso-inv xf) = xf
proof
  let \(?rf = Rep-lifted-distrib-lattice-top xf
  let \(?pf = Abs-dense (Abs-lifted-pair (\lambda f . (top,up-filter (?)f)))
  have 1: \(\forall f . (top,up-filter (?)f) \in triplepairs (Rep-phi f)
    proof
      fix f :: ('a,'b) phi
      have triple (Rep-phi f)
        by (simp add: triple-def)
      thus (top,up-filter (?)f) \in triplepairs (Rep-phi f)
        using triplepairs-def by force
    qed
    have 2: dense (Abs-lifted-pair (\lambda f . (top,up-filter (?)f)))
      proof
        have \(-Abs-lifted-pair (\lambda f . (top,up-filter (?)f)) = Abs-lifted-pair (\lambda f .
          triplepairs-uminus (Rep-phi f) (top,up-filter (?)f))
          using f by (simp add: eq-opn-same-args uminus-lifted-pair.abs-eq)
        also have ...
          by simp add: triplepairs-uminus.simps triple-def)
        also have ...
          by (metis (no-types, hide-lams) simp-phi triplephi-top triple-def)
        also have ...
          by simp add: bot-lifted-pair-def)
        finally show \(?thesis
          by simp
        qed
      have get-dense \(?pf = \(?rf
        proof
          fix f
          have (top,up-filter (get-dense \(?pf f) = Rep-lifted-pair (Rep-dense \(?pf) f
            by (metis get-dense-char)
          also have ...
            by (simp add: Rep-dense-inverse)
          also have ...
            by (simp add: Abs-lifted-pair-inverse)
          finally show get-dense \(?pf f = \(?rf f
            using up-filter-injective by auto
          qed
        thus \(?thesis
          by (simp add: Rep-lifted-distrib-lattice-top-inverse)
        qed

104
To obtain the isomorphism, it remains to show the homomorphism properties of lattices with a greatest element.

**lemma** dl-iso:

*bounded-lattice-top-isomorphism dl-iso*

**proof** (intro conjI)

have \( \text{get-dense } \text{top} = (\lambda f::(a',b) \text{ phi } . \text{top}) \)

**proof**

fix \( f::(a',b) \text{ phi} \)

have \( \text{Rep-lifted-pair} (\text{Rep-dense } \text{top} ) = (\text{top} . \text{Abs-filter} \{ \text{top} \}) \)

by (simp add: top-dense rep-eq top-lifted-pair rep-eq)

hence \( \text{up-filter} (\text{get-dense } \text{top} f) = \text{Abs-filter} \{ \text{top} \} \)

by (metis prod.inject get-dense-char)

hence \( \text{Rep-filter} (\text{up-filter} (\text{get-dense } \text{top} f)) = \{ \text{top} \} \)

by (metis bot-filter.abs-eq bot-filter.rep-eq)

thus \( \text{get-dense } \text{top} f = \text{top} \)

by (metis self-in-upset singletonD Abs-filter.inverse mem-Collect-eq up-filter)

qed

thus \( \text{Abs-lifted-distrib-lattice-top} (\text{get-dense } \text{top}::(a',b) \text{ phi } \Rightarrow b') = \text{top} \)

by (metis top-lifted-distrib-lattice-top-def)

next

show \( \forall pf qf::(a',b) \text{ lifted-pair dense} . \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} (pf \sqcup qf)) = \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} pf) \sqcup \)

\( \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} qf) \)

**proof** (intro allI)

fix \( pf qf::(a',b) \text{ lifted-pair dense} \)

have \( 1: \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} pf) \sqcup \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} qf) \)

by (simp add: eq-onp-same-args sup-lifted-distrib-lattice-top.abs-eq)

have \( (\lambda f . \text{get-dense} (pf \sqcup qf) f) = (\lambda f . \text{get-dense} pf f \sqcup \text{get-dense} qf f) \)

**proof**

fix \( f \)

have \( (\text{top} . \text{up-filter} (\text{get-dense} (pf \sqcup qf) f)) = \text{Rep-lifted-pair} (\text{Rep-dense} (pf \sqcup qf)) f \)

by (metis get-dense-char)

also have \( \ldots = \text{triple.pairs-sup} (\text{Rep-lifted-pair} (\text{Rep-dense} pf) f) \)

\( (\text{Rep-lifted-pair} (\text{Rep-dense} qf) f) \)

by (simp add: sup-lifted-pair.rep-eq sup-dense.rep-eq)

also have \( \ldots = \text{triple.pairs-sup} (\text{top} . \text{up-filter} (\text{get-dense} pf f)) (\text{top} . \text{up-filter} (\text{get-dense} qf f)) \)

by (metis get-dense-char)

also have \( \ldots = (\text{top} . \text{up-filter} (\text{get-dense} pf f) \sqcup \text{up-filter} (\text{get-dense} qf f)) \)

by (metis (no-types, lifting) calculation prod.simps(1) simp-phi triple.pairs-sup.simps triple-def)

also have \( \ldots = (\text{top} . \text{up-filter} (\text{get-dense} pf f \sqcup \text{get-dense} qf f)) \)

by (metis up-filter-dist-sup)

finally show \( \text{get-dense} (pf \sqcup qf) f = \text{get-dense} pf f \sqcup \text{get-dense} qf f \)

using up-filter-injective by blast

qed
thus \( \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} (pf \sqcup qf)) = \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} pf) \sqcup \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} qf) \)

using \( f \) by \text{metis}

qed

next

show \( \forall \text{pf qf} :: (a',b') \text{ lifted-pair dense} \). \( \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} (pf \sqcap qf)) = \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} pf) \sqcap \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} qf) \)

proof (intro allI)

fix \( \text{pf qf} :: (a',b') \text{ lifted-pair dense} \)

have 1: \( \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} pf) \sqcap \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} qf) = (\lambda f . \text{get-dense} pf f \sqcap \text{get-dense} qf f) \)

by (simp add: eq-onp-same-args inf-lifted-distrib-lattice-top.abs-eq)

have \((\lambda f . \text{get-dense} (pf \sqcap qf) f) = (\lambda f . \text{get-dense} pf f \sqcap \text{get-dense} qf f)\)

proof

fix \( f \)

have \((\text{top,up-filter} (\text{get-dense} (pf \sqcap qf) f)) = \text{Rep-lifted-pair} (\text{Rep-dense} (pf \sqcap qf)) f\)

by (metis get-dense-char)

also have \( \ldots = \text{triple.pairs-inf} (\text{Rep-lifted-pair} (\text{Rep-dense} pf) f)\)

((\text{Rep-lifted-pair} (\text{Rep-dense} qf) f))

by (simp add: inf-lifted-pair.rep-eq inf-dense.rep-eq)

also have \( \ldots = \text{triple.pairs-inf} (\text{top,up-filter} (\text{get-dense} pf f)) (\text{top,up-filter} (\text{get-dense} qf f))\)

by (metis get-dense-char)

also have \( \ldots = (\text{top,up-filter} (\text{get-dense} pf f) \sqcup \text{up-filter} (\text{get-dense} qf f))\)

by (metis (no-types, lifting) calculation prod.simps(1) simp-phi triple.pairs-inf.simps triple-def)

also have \( \ldots = (\text{top,up-filter} (\text{get-dense} pf f \sqcap \text{get-dense} qf f))\)

by (metis up-filter-dist-inf)

finally show \( \text{get-dense} (pf \sqcap qf) f = \text{get-dense} pf f \sqcap \text{get-dense} qf f\)

using up-filter-injective by \text{blast}

qed

thus \( \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} (pf \sqcap qf)) = \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} pf) \sqcap \text{Abs-lifted-distrib-lattice-top} (\text{get-dense} qf) \)

using \( f \) by \text{metis}

qed

next

show bij \text{dl-iso}

by (rule invertible-bij[where \( g=\text{dl-iso-inv} \)] (simp-all add: dl-iso-left-invertible dl-iso-right-invertible)

qed

5.7.3 Structure Map Preservation
We finally show that the isomorphisms are compatible with the structure maps. This involves lifting the distributive lattice isomorphism to filters of distributive lattices (as these are the targets of the structure maps). To this end, we first show that the lifted isomorphism preserves filters.

**Lemma** phi-iso-filter:
- **filter** ((λg::('a::non-trivial-boolean-algebra,'b::distrib-lattice-top) lifted-pair dense . Rep-lifted-distrib-lattice-top (dl-iso qf) f) · Rep-filter (stone-phi pf))
- **proof** (rule filter-map-filter)
- **show** mono (λqf::('a::non-trivial-boolean-algebra,'b::distrib-lattice-top)
lifted-pair dense . Rep-lifted-distrib-lattice-top (dl-iso qf) f)
- **by** (metis (no-types, lifting) mono_def dl-iso le-iff-sup sup-lifted-distrib-lattice-top.rep-eq)
- **next**
- **show** ∀ qf y . Rep-lifted-distrib-lattice-top (dl-iso qf) f ≤ y → (∃ rf . qf ≤ rf)
- **proof** (intro allI, rule impI)
- **fix** qf :: ('a,'b) lifted-pair dense
- **fix** y :: 'b
- **assume** 1: Rep-lifted-distrib-lattice-top (dl-iso qf) f ≤ y
- **let** ?rf = Abs-dense (Abs-lifted-pair (λg . if g = f then (top,up-filter y) else Rep-lifted-pair (Rep-dense qf) g))
- **have** 2: ∀ qf . (if g = f then (top,up-filter y) else Rep-lifted-pair (Rep-dense qf) g) ∈ triple.pairs (Rep-phi g)
- **by** (metis Abs-lifted-distrib-lattice-top-inverse dl-iso-inv-lifted-pair mem-Collect-eq simp-lifted-pair
- **hence** ¬ Abs-lifted-pair (λg . if g = f then (top,up-filter y) else Rep-lifted-pair (Rep-dense qf) g) = Abs-lifted-pair (λg . triple.pairs-uminus (Rep-phi g) (if g = f then (top,up-filter y) else Rep-lifted-pair (Rep-dense qf) g))
- **by** (simp add: eq-onp-def uminus-lifted-pair.abs-eq)
- **also have** ... = Abs-lifted-pair (λg . if g = f then triple.pairs-uminus (Rep-phi g) (top,up-filter y) else triple.pairs-uminus (Rep-phi g) (Rep-lifted-pair (Rep-dense qf) g))
- **by** (simp add: if-distrib)
- **also have** ... = Abs-lifted-pair (λg . if g = f then (bot,top) else triple.pairs-uminus (Rep-phi g) (Rep-lifted-pair (Rep-dense qf) g))
- **by** (subst triple.pairs-uminus.simps, simp add: triple-def, metis compl-top-eq simp-phi)
- **also have** ... = Abs-lifted-pair (λg . if g = f then (bot,top) else (bot,top))
- **by** (metis bot-lifted-pair.rep-eq simp-dense top-filter.abs-eq
uminus-lifted-pair.rep-eq)
- **also have** ... = bot
- **by** (simp add: bot-lifted-pair.abs-eq top-filter.abs-eq
uminus-lifted-pair.rep-eq)
- **finally have** 3: Abs-lifted-pair (λg . if g = f then (top,up-filter y) else Rep-lifted-pair (Rep-dense qf) g) ∈ dense-elements
- **by** blast
- **hence** (top,up-filter (get-dense (Abs-dense (Abs-lifted-pair (λg . if g = f then (top,up-filter y) else Rep-lifted-pair (Rep-dense qf) g))) f)) = Rep-lifted-pair (Rep-dense (Abs-lifted-pair (λg . if g = f then (top,up-filter y) else Rep-lifted-pair (Rep-dense qf) g))) f
by (metis (mono-tags, lifting) get-dense-char)
also have ... = Rep-lifted-pair (Abs-lifted-pair (λg . if g = f then (top, up-filter y) else Rep-lifted-pair (Rep-dense qf g)) f)
  using 3 by (simp add: Abs-dense-inverse)
also have ... = (top, up-filter y)
  using 2 by (simp add: Abs-lifted-pair-inverse)
finally have get-dense (Abs-dense (Abs-lifted-pair (λg . if g = f then (top, up-filter y) else Rep-lifted-pair (Rep-dense qf g)) f)) f = y
  using up-filter-injective by blast
hence 4: Rep-lifted-distrib-lattice-top (dl-iso ?rf) f = y
by (simp add: Abs-lifted-distrib-lattice-top-inverse)

\{ 
  fix g 
  have Rep-lifted-distrib-lattice-top (dl-iso qf) g ≤ Rep-lifted-distrib-lattice-top (dl-iso ?rf) g
  proof (cases g = f)
    assume g = f 
    thus ?thesis
      using 1 4 by simp
  next 
    assume 5: g ≠ f 
    have (top, up-filter (get-dense ?rf g)) = Rep-lifted-pair (Rep-dense (Abs-dense (Abs-lifted-pair (λg . if g = f then (top, up-filter y) else Rep-lifted-pair (Rep-dense qf g)) f)) g)
      by (metis (mono-tags, lifting) get-dense-char)
    also have ... = Rep-lifted-pair (Abs-lifted-pair (λg . if g = f then (top, up-filter y) else Rep-lifted-pair (Rep-dense qf g)) g)
      using 3 by (simp add: Abs-dense-inverse)
    also have ... = Rep-lifted-pair (Rep-dense qf g)
      using get-dense-char by auto
    finally have get-dense ?rf g = get-dense qf g
      using up-filter-injective by blast
    thus Rep-lifted-distrib-lattice-top (dl-iso qf) g ≤ Rep-lifted-distrib-lattice-top (dl-iso ?rf) g
      by (simp add: Abs-lifted-distrib-lattice-top-inverse)
  qed 
\}


by (simp add: le-funI)

hence 6: dl-iso qf ≤ dl-iso ?rf 
  by (simp add: le-funD less-eq-lifted-distrib-lattice-top.rep-eq)

hence qf ≤ ?rf 
  by (metis (no-types, lifting) dl-iso sup-isomorphism-ord-isomorphism)

thus ∃ rf . qf ≤ rf ∧ y = Rep-lifted-distrib-lattice-top (dl-iso rf) f
  using 4 by auto

qed
The commutativity property states that the same result is obtained in
two ways by starting with a regular lifted pair \( pf \):

* apply the Boolean algebra isomorphism to the pair; then apply a struc-
ture map \( f \) to obtain a filter of dense elements; or,

* apply the structure map \( \text{stone-phi} \) to the pair; then apply the distribu-
tive lattice isomorphism lifted to the resulting filter.

**Lemma phi-iso:**

\[
\text{Rep-phi } f \ (\text{Rep-lifted-boolean-algebra} \ (\text{ba-iso } pf) \ f) = \text{filter-map} \]
\[
(\lambda qf.::(\text{a::non-trivial-boolean-algebra},\text{b::distrib-lattice-top}) \text{lifted-pair dense} . \text{Rep-lifted-distrib-lattice-top} \ (\text{dl-iso } qf) \ f) \ (\text{stone-phi } pf)
\]

**Proof**

\[
\begin{align*}
\text{let } \ ?r &= \text{Rep-phi } f \\
\text{let } \ ?ppf &= \lambda g . \text{triple.pairs-uminus} \ (\text{Rep-phi } g) \ (\text{Rep-lifted-pair} \ (\text{Rep-regular} \ pf) \ g)
\end{align*}
\]

**Have 1:** triple \( ?r \)

by (simp add: triple-def)

**Have 2:** Rep-filter \( (?r \ (\text{fst} \ (\text{Rep-lifted-pair} \ (\text{Rep-regular} \ pf) \ f)))) \subseteq \{ z . \exists qf . \text{Rep-regular} \ pf \leq \text{Rep-dense} \ qf \land z = \text{get-dense} \ qf \ f \}

**Proof**

\[
\begin{align*}
\text{fix } z \\
\text{obtain } x \text{ where } 3: x &= \text{fst} \ (\text{Rep-lifted-pair} \ (\text{Rep-regular} \ pf) \ f) \\
& \text{by simp} \\
\text{assume } z \in \text{Rep-filter} \ (\ ?r \ (\text{fst} \ (\text{Rep-lifted-pair} \ (\text{Rep-regular} \ pf) \ f)))) \\
\text{hence } \uparrow z \subseteq \text{Rep-filter} \ (\ ?r \ x) \\
& \text{using 3 filter-def by fastforce} \\
\text{hence 4: up-filter } z \leq \ ?r \ x \\
& \text{by (metis Rep-filter-cases Rep-filter-inverse less-eq-filter.rep-eq mem-Collect-eq up-filter)}
\end{align*}
\]

**Have 5:** \( \forall g . \ ?ppf \ g \in \text{triple.pairs} \ (\text{Rep-phi } g) \)

by (metis (no-types) simp-lifted-pair uminus-lifted-pair.rep-eq)

**Let** \( ?zf = \lambda g . \text{if } g = f \text{ then } (\text{top,up-filter } z) \text{ else } \text{triple.pairs-top} \)

**Have 6:** \( \forall g . \ ?zf \ g \in \text{triple.pairs} \ (\text{Rep-phi } g) \)

**Proof**

\[
\begin{align*}
\text{fix } g :: (\text{'a,'b}) \phi \\
\text{have triple } (\text{Rep-phi } g) \\
& \text{by (simp add: triple-def)} \\
\text{hence } (\text{top,up-filter } z) \in \text{triple.pairs} \ (\text{Rep-phi } g) \\
& \text{using triple.pairs-def by force} \\
\text{thus } ?zf \ g \in \text{triple.pairs} \ (\text{Rep-phi } g) \\
& \text{by (metis simp-lifted-pair top-lifted-pair.rep-eq)} \\
\text{qed}
\end{align*}
\]

\[
\text{hence } -\text{Abs-lifted-pair} \ ?zf = \text{Abs-lifted-pair} \ (\lambda g . \text{triple.pairs-uminus} \ (\text{Rep-phi } g) \ (\ ?zf \ g))
\]

by (subst uminus-lifted-pair.abs-eq) (simp-all add: eq-onp-same-args)
also have ... = Abs-lifted-pair (\lambda g . if g = f then triple.pairs-uminus (Rep-phi g) (top, up-filter z) else triple.pairs-uminus (Rep-phi g) triple.pairs-top)
   by (rule arg-cong [where f = Abs-lifted-pair]) auto
also have ... = Abs-lifted-pair (\lambda g . triple.pairs-bot)
   using 1 by (metis bot-lifted-pair.rep-eq dense-closed-top top-lifted-pair.rep-eq triple.pairs-uminus.simps uminus-lifted-pair.rep-eq)
finally have 7: Abs-lifted-pair \?zf \in dense-elements
   by (simp add: bot-lifted-pair.abs-eq)
let \?gf = Abs-dense (Abs-lifted-pair \?zf)
have \forall g . triple.pairs-less-eq (?ppf g) (?zf g)
proof
fix g
show triple.pairs-less-eq (?ppf g) (?zf g)
proof (cases g = f)
   assume 8: g = f
   hence 9: ?ppf g = (\-x, \?r x)
      using 1 3 by (metis prod.collapse triple.pairs-uminus.simps)
   have triple.pairs-less-eq (\-x, \?r x) (top, up-filter z)
      using 1 4 by (meson inf.bot-least triple.pairs-less-eq.simps)
   thus ?thesis
      using 8 9 by simp
next
   assume 10: g \neq f
   have triple.pairs-less-eq (?ppf g) triple.pairs-top
      using 1 by (metis (no-types, hide-lams) bot.extremum top-greatest prod.collapse triple-def triple.pairs-less-eq.simps triple.phi-bot)
   thus ?thesis
      using 10 by simp
qed
hence Abs-lifted-pair ?ppf \leq Abs-lifted-pair ?zf
   using 5 6 by (subst less-eq-lifted-pair.abs-eq) (simp-all add: eq-onp-same-args)
   hence 11: \- Rep-regular pf \leq Rep-dense \?gf
      using 7 by (simp add: uminus-lifted-pair-def Abs-dense-inverse)
   have (top, up-filter (get-dense ?gf f)) = Rep-lifted-pair (Rep-dense ?gf f)
      by (metis get-dense-char)
   also have ... = (top, up-filter z)
      using 6 7 Abs-dense-inverse Abs-lifted-pair-inverse by force
finally have z = get-dense ?gf f
   using up-filter-injective by force
   thus z \in { z . \exists \?gf . \- Rep-regular pf \leq Rep-dense qf \land z = get-dense qf f }
   using 11 by auto
qed
have 12: Rep-filter (?r (fst (Rep-lifted-pair (Rep-regular pf f)))) \supseteq { z . \exists \?gf . 
   \- Rep-regular pf \leq Rep-dense qf \land z = get-dense qf f }
   proof
   fix z
   assume z \in { z . \exists \?gf . 
   \- Rep-regular pf \leq Rep-dense qf \land z = get-dense qf f

110
hence \( \exists qf . -\text{Rep-regular } pf \leq \text{Rep-dense } qf \land z = \text{get-dense } qf f \)

by auto

hence \( \text{triple.pairs-less-eq } (\text{Rep-lifted-pair } (-\text{Rep-regular } pf) f) (\top, \text{up-filter } z) \)

by (metis less-eq-lifted-pair.rep-eq get-dense-char)

hence \( \text{up-filter } z \leq \text{snd } (\text{Rep-lifted-pair } (-\text{Rep-regular } pf) f) \)

using 1 by (metis (no-types, hide-lams) prod.collapse triple.pairs-less-eq.simps)

also have \( ... = \text{snd } (\text{?ppf } f) \)

by (metis uminus-lifted-pair.rep-eq)

also have \( ... = ?r (\text{fst } (\text{Rep-lifted-pair } (\text{Rep-regular } pf) f)) \)

using 1 by (metis (no-types) prod.collapse prod.inject triple.pairs-uminus.simps)

finally have \( \text{Rep-filter } (\text{up-filter } z) \subseteq \text{Rep-filter } (?r (\text{fst } (\text{Rep-lifted-pair } (\text{Rep-regular } pf) f))) \)

by (simp add: less-eq-filter.rep-eq)

hence \( \exists z \subseteq \text{Rep-filter } (?r (\text{fst } (\text{Rep-lifted-pair } (\text{Rep-regular } pf) f))) \)

by (metis Abs-filter-inverse mem-Collect-eq up-filter)

thus \( z \in \text{Rep-filter } (?r (\text{fst } (\text{Rep-lifted-pair } (\text{Rep-regular } pf) f))) \)

by blast

qed

have 13: \( \forall qf \in \text{Rep-filter } (\text{stone-phi } pf) . \text{Rep-lifted-distrib-lattice-top } (\text{Abs-lifted-distrib-lattice-top } (\text{get-dense } qf)) f = \text{get-dense } qf f \)

by (metis Abs-lifted-distrib-lattice-top-inverse UNIV-I UNIV-def)

have \( \text{Rep-filter } (?r (\text{fst } (\text{Rep-lifted-pair } (\text{Rep-regular } pf) f))) = \{ z . \exists qf \in \text{stone-phi-set } pf . z = \text{get-dense } qf f \} \)

using 2 12 by simp

hence \( ?r (\text{fst } (\text{Rep-lifted-pair } (\text{Rep-regular } pf) f)) = \text{Abs-filter } \{ z . \exists qf \in \text{stone-phi-set } pf . z = \text{get-dense } qf f \} \)

by (metis Rep-filter-inverse)

hence \( ?r (\text{Rep-lifted-boolean-algebra } (\text{ba-iso } pf) f) = \text{Abs-filter } \{ z . \exists qf \in \text{Rep-filter } (\text{stone-phi } pf) . z = \text{Rep-lifted-distrib-lattice-top } (\text{dl-iso } qf) f \} \)


thus \( ?\text{thesis} \)

by (simp add: image-def)

qed

end

References


