Stone Algebras

Walter Guttmann

September 6, 2016

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

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** ⊔ *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** ⊓ *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.

context *ord*
begin

definition *isotone* :: $('a \Rightarrow 'a) \Rightarrow bool$
  where $isotone\ f \equiv \forall x\ y\ .\ x \le y \longrightarrow f\ x \le f\ y$

end

context *order*
begin

lemma *order-lesseq-imp*:
$(\forall z . x \le z \longrightarrow y \le z) \longleftrightarrow y \le x$
using *order-trans* **by** *blast*

end

The following are basic facts in semilattices.

context *semilattice-sup*
begin

lemma *sup-left-isotone*:
$x \le y \Longrightarrow x \sqcup z \le y \sqcup z$
using *sup.mono* **by** *blast*

lemma *sup-right-isotone*:
$x \le y \Longrightarrow z \sqcup x \le z \sqcup y$
using *sup.mono* **by** *blast*

lemma *sup-left-divisibility*:
$x \le y \longleftrightarrow (\exists z . x \sqcup z = y)$
using *sup.absorb2 sup.cobounded1* **by** *blast*

lemma *sup-right-divisibility*:
$x \le y \longleftrightarrow (\exists z . z \sqcup x = y)$
by (*metis sup.cobounded2 sup.orderE*)

lemma *sup-same-context*:
$x \le y \sqcup z \Longrightarrow y \le x \sqcup z \Longrightarrow x \sqcup z = y \sqcup z$
by (*simp add*: *le-iff-sup sup-left-commute*)

lemma *sup-relative-same-increasing*:
$x \le y \Longrightarrow x \sqcup z = x \sqcup w \Longrightarrow y \sqcup z = y \sqcup w$
using *sup.assoc sup-right-divisibility* **by** *auto*

end

context *semilattice-inf*
begin

lemma *inf-same-context*:
$x \le y \sqcap z \Longrightarrow y \le x \sqcap z \Longrightarrow x \sqcap z = y \sqcap z$
using *antisym* **by** *auto*

end

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.

class *directed-semilattice-inf* = *semilattice-inf* +
  assumes *ub*: $\exists z . x \leq z \wedge y \leq z$

We extend the *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* :: $('a \Rightarrow 'a) \Rightarrow bool$
  where *dual-additive* $f \equiv \forall x\ y . f\ (x \sqcup y) = f\ x \sqcap f\ y$

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* $x\ y \equiv x \sqcup y = top \wedge x \sqcap y = bot$

lemma *complement-symmetric*:
  complement $x\ y \Longrightarrow$ *complement* $y\ x$
  by (*simp add*: *inf*.*commute sup*.*commute*)

definition *conjugate* :: $('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) \Rightarrow bool$
  where *conjugate* $f\ g \equiv \forall x\ y . f\ x \sqcap y = bot \longleftrightarrow x \sqcap g\ y = bot$

end

context *distrib-lattice*
begin

lemma *relative-equality*:
$x \sqcup z = y \sqcup z \Longrightarrow x \sqcap z = y \sqcap z \Longrightarrow x = y$
by (*metis inf.commute inf-sup-absorb inf-sup-distrib2*)

end

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 $z \sqcap x \leq y \longleftrightarrow x \leq w \sqcup y$
proof
assume *1*: $z \sqcap x \leq y$
have $x = (z \sqcup w) \sqcap x$
by (*simp add*: *assms*)
also have ... $\leq y \sqcup (w \sqcap x)$
using *1 sup.commute sup.left-commute inf-sup-distrib2 sup-right-divisibility*
by *fastforce*
also have ... $\leq w \sqcup y$
by (*simp add*: *inf.coboundedI1*)
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 ... $= z \sqcap y$
by (*simp add*: *assms inf-sup-distrib1*)
also have ... $\leq y$
by *simp*
finally show $z \sqcap x \leq y$
.
qed

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: ∃ x y . x ≠ y

class non-trivial-order = non-trivial + order

class non-trivial-order-bot = non-trivial-order + order-bot

class non-trivial-bounded-order = non-trivial-order-bot + order-top
begin

lemma bot-not-top:
  bot ≠ top
proof −
  from consistent obtain x y :: 'a where x ≠ y
    by auto
  thus ?thesis
    by (metis bot-less top.extremum-strict)
qed
```

The following results extend basic Isabelle/HOL facts.

```
lemma if-distrib-2:
  f (if c then x else y) (if c then z else w) = (if c then f x z else f y w)
  by simp

lemma left-invertible-inj:
  (∀ x . g (f x) = x) ⟹ inj f
  by (metis injI)

lemma invertible-bij:
  assumes ∀ x . g (f x) = x
      and ∀ y . f (g y) = y
    shows bij f
  by (metis assms bijI')

end

end
```

3 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 least element whose meet with y is the least element of the lattice.

class *p-algebra* = *bounded-lattice* + *uminus* +
  assumes *pseudo-complement*: $x \sqcap y = bot \longleftrightarrow x \le -y$
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* $x \equiv \exists\, y\ .\ x \sqcap y = bot \wedge x \sqcup y = top$
abbreviation *in-p-image* $x \equiv \exists\, y\ .\ x = -y$
abbreviation *selection* $s\ x \equiv s = --s \sqcap x$

abbreviation *dense-elements* ≡ { *x* . *dense x* }
abbreviation *regular-elements* ≡ { *x* . *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 ⊓ −*x* = *bot*
 using *inf.commute pseudo-complement* **by** *fastforce*

lemma *p-inf* [*simp*]:
 −*x* ⊓ *x* = *bot*
 by (*simp add*: *inf-commute*)

lemma *pp-inf-p*:
 −−*x* ⊓ −*x* = *bot*
 by *simp*

The double complement is a closure operation.

lemma *pp-increasing*:
 x ≤ −−*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 ⟷ *in-p-image x*
 by *auto*

lemma *pseudo-complement-pp*:
 x ⊓ *y* = *bot* ⟷ −−*x* ≤ −*y*
 by (*metis inf-commute pseudo-complement ppp*)

lemma *p-antitone*:
 x ≤ *y* ⟹ −*y* ≤ −*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*:
$x \leq -y \longleftrightarrow y \leq -x$
using *order-lesseq-imp p-antitone pp-increasing* **by** *blast*

lemma *pp-isotone*:
$x \leq y \Longrightarrow --x \leq --y$
by (*simp add*: *p-antitone*)

lemma *pp-isotone-sup*:
$--x \leq --(x \sqcup y)$
by (*simp add*: *p-antitone*)

lemma *pp-isotone-inf*:
$--(x \sqcap y) \leq --x$
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*:
$-x \sqcup -y \leq -(x \sqcap y)$
by (*simp add*: *p-antitone*)

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

lemma *p-sup-p* [*simp*]:
$-(x \sqcup -x) = bot$
by *simp*

lemma *pp-sup-p* [*simp*]:
$--(x \sqcup -x) = top$
by *simp*

lemma *dense-pp*:
  dense $x \longleftrightarrow --x = top$
  by (*metis p-bot p-top ppp*)

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

lemma *regular-char*:
  regular $x \longleftrightarrow (\exists\, y\ .\ x = -y)$
  by *auto*

lemma *pp-inf-bot-iff*:
  $x \sqcap y = bot \longleftrightarrow --x \sqcap y = bot$
  by (*simp add*: *pseudo-complement-pp*)

Weak forms of the shunting property hold. Most require a pseudocomplemented element on the right-hand side.

lemma *p-shunting-swap*:
  $x \sqcap y \leq -z \longleftrightarrow x \sqcap z \leq -y$
  by (*metis inf-assoc inf-commute pseudo-complement*)

lemma *pp-inf-below-iff*:
  $x \sqcap y \leq -z \longleftrightarrow --x \sqcap y \leq -z$
  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 *p-inf-pp-pp* [*simp*]:
  $-(--x \sqcap --y) = -(x \sqcap y)$
  by (*simp add*: *inf-commute*)

lemma *regular-closed-inf*:
  regular $x \Longrightarrow$ *regular* $y \Longrightarrow$ *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*]:
$x \sqcap -(x \sqcap y) = x \sqcap -y$
apply (*rule antisym*)
using *p-shunting-swap* **apply** *fastforce*
using *inf.sup-right-isotone p-antitone* **by** *auto*

Pseudocomplements are unique.

lemma *p-unique*:
$(\forall x . x \sqcap y = bot \longleftrightarrow x \leq z) \Longrightarrow z = -y$
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*:
$x \leq --y \longleftrightarrow x \sqcap -y = bot$
by (*simp add*: *pseudo-complement*)

lemma *pp-pp-inf-bot-iff*:
$x \sqcap y = bot \longleftrightarrow --x \sqcap --y = bot$
by (*simp add*: *pseudo-complement-pp*)

lemma *inf-pp-semi-commute*:
$x \sqcap --y \leq --(x \sqcap y)$
using *inf.eq-refl p-antitone-iff p-inf-pp* **by** *presburger*

lemma *inf-pp-commute*:
$--(--x \sqcap y) = --x \sqcap --y$
by *simp*

lemma *sup-pp-semi-commute*:
$x \sqcup --y \leq --(x \sqcup y)$
by (*simp add*: *p-antitone-iff*)

lemma *regular-sup*:
$regular\ z \Longrightarrow (x \leq z \wedge y \leq z \longleftrightarrow --(x \sqcup y) \leq z)$
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*:
dense $x \Longrightarrow$ *dense* $y \Longrightarrow$ *dense* $(x \sqcap y)$
by (*simp add*: *dense-pp*)

lemma *dense-closed-sup*:
dense $x \Longrightarrow$ *dense* $y \Longrightarrow$ *dense* $(x \sqcup y)$
by *simp*

lemma *dense-closed-pp*:
dense $x \Longrightarrow$ *dense* $(--x)$
by *simp*

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

lemma *dense-up-closed*:
dense $x \Longrightarrow x \leq y \Longrightarrow$ *dense* y
using *dense-pp top-le pp-isotone* **by** *auto*

lemma *regular-dense-top*:
regular $x \Longrightarrow$ *dense* $x \Longrightarrow x =$ *top*
using *p-bot* **by** *blast*

lemma *selection-char*:
selection $s\ x \longleftrightarrow (\exists y\ .\ s = -y \sqcap x)$
by (*metis inf-import-p inf-commute regular-closed-p*)

lemma *selection-closed-inf*:
selection $s\ x \Longrightarrow$ *selection* $t\ x \Longrightarrow$ *selection* $(s \sqcap t)\ x$
by (*metis inf-assoc inf-commute inf-idem pp-dist-inf*)

lemma *selection-closed-pp*:
regular $x \Longrightarrow$ *selection* $s\ x \Longrightarrow$ *selection* $(--s)\ x$
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 $\forall x\ y\ .\ x \sqcap g\ y = bot \longleftrightarrow x \sqcap h\ y = bot$
    using *assms conjugate-def inf.commute* **by** *simp*
  hence $\forall x\ y\ .\ x \le -(g\ y) \longleftrightarrow x \le -(h\ y)$
    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*:
  $conjugate\ f\ g \Longrightarrow conjugate\ g\ f$
  by (*simp add*: *conjugate-def inf-commute*)

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

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

lemma *conjugate-dual-additive*:
  assumes *conjugate f g*
  shows *dual-additive* ($uminus \circ f$)
proof −
  have *1*: $\forall x\ y\ z\ .\ -z \le -(f\ (x \sqcup y)) \longleftrightarrow -z \le -(f\ x) \wedge -z \le -(f\ y)$
  proof (*intro allI*)
    fix *x y z*
    have $(-z \le -(f\ (x \sqcup y))) = (f\ (x \sqcup y) \sqcap -z = bot)$
      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 ... $= (x \sqcup y \le -(g(-z)))$

by (*simp add*: *pseudo-complement*)
also have ... $= (x \leq -(g(-z)) \wedge y \leq -(g(-z)))$
by (*simp add*: *le-sup-iff*)
also have ... $= (x \sqcap g(-z) = bot \wedge y \sqcap g(-z) = bot)$
by (*simp add*: *pseudo-complement*)
also have ... $= (f\, x \sqcap -z = bot \wedge f\, y \sqcap -z = bot)$
using *assms conjugate-def* **by** *auto*
also have ... $= (-z \leq -(f\, x) \wedge -z \leq -(f\, y))$
by (*simp add*: *p-antitone-iff pseudo-complement*)
finally show $-z \leq -(f\ (x \sqcup y)) \longleftrightarrow -z \leq -(f\, x) \wedge -z \leq -(f\, y)$
by *simp*
qed
have $\forall x\ y\ .\ -(f\ (x \sqcup y)) = -(f\, x) \sqcap -(f\, y)$
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 *1* **by** (*metis inf-le1 inf-le2*)
thus $-(f\ (x \sqcup y)) = -(f\, x) \sqcap -(f\, y)$
using *1 antisym* **by** *fastforce*
qed
thus *?thesis*
using *dual-additive-def* **by** *simp*
qed

lemma *conjugate-isotone-pp*:
conjugate f g $\Longrightarrow$ *isotone* (*uminus* $\circ$ *uminus* $\circ$ *f*)
by (*simp add*: *comp-assoc conjugate-dual-additive dual-additive-antitone*)

lemma *conjugate-char-1-pp*:
conjugate f g $\longleftrightarrow (\forall x\ y\ .\ f(x \sqcap -(g\ y)) \leq --f\ x \sqcap -y \wedge g(y \sqcap -(f\, x)) \leq --g\ y \sqcap -x)$
proof
assume *1*: *conjugate f g*
show $\forall x\ y\ .\ f(x \sqcap -(g\ y)) \leq --f\ x \sqcap -y \wedge g(y \sqcap -(f\, x)) \leq --g\ y \sqcap -x$
proof (*intro allI*)
fix $x\ y$
have *2*: $f(x \sqcap -(g\ y)) \leq -y$
using *1* **by** (*simp add*: *conjugate-def pseudo-complement*)
have $f(x \sqcap -(g\ y)) \leq --f(x \sqcap -(g\ y))$
by (*simp add*: *pp-increasing*)
also have ... $\leq --f\ x$
using *1 conjugate-isotone-pp isotone-def* **by** *simp*
finally have *3*: $f(x \sqcap -(g\ y)) \leq --f\ x \sqcap -y$
using *2* **by** *simp*
have *4*: *isotone* (*uminus* $\circ$ *uminus* $\circ$ *g*)
using *1 conjugate-isotone-pp conjugate-symmetric* **by** *auto*
have *5*: $g(y \sqcap -(f\, x)) \leq -x$

using *1* **by** (*metis conjugate-def inf.cobounded2 inf-commute pseudo-complement*)
have $g(y \sqcap -(f\ x)) \leq --g(y \sqcap -(f\ x))$
by (*simp add*: *pp-increasing*)
also have ... $\leq --g\ y$
using *4 isotone-def* **by** *auto*
finally have $g(y \sqcap -(f\ x)) \leq --g\ y \sqcap -x$
using *5* **by** *simp*
thus $f(x \sqcap -(g\ y)) \leq --f\ x \sqcap -y \wedge g(y \sqcap -(f\ x)) \leq --g\ y \sqcap -x$
using *3* **by** *simp*
qed
next
assume *6*: $\forall x\ y\ .\ f(x \sqcap -(g\ y)) \leq --f\ x \sqcap -y \wedge g(y \sqcap -(f\ x)) \leq --g\ y \sqcap -x$
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*
qed

lemma *conjugate-char-1-isotone*:
conjugate f g $\Longrightarrow$ *isotone f* $\Longrightarrow$ *isotone g* $\Longrightarrow f(x \sqcap -(g\ y)) \leq f\ x \sqcap -y \wedge g(y \sqcap -(f\ x)) \leq g\ y \sqcap -x$
by (*simp add*: *conjugate-char-1-pp ord.isotone-def*)

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*)
by (*metis* (*full-types*) *inf.left-commute inf.orderE inf-bot-right inf-commute inf-p-eq*)

end

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*:
$-x \sqcap (x \sqcup y) \leq y$
by (*simp add*: *inf-sup-distrib1*)

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

lemma *complement-p*:
$x \sqcap y = bot \Longrightarrow x \sqcup y = top \Longrightarrow -x = y$
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 $x \Longrightarrow$ *regular* x
using *complement-p inf.commute sup.commute* **by** *fastforce*

lemma *regular-inf-dense*:
$\exists y\ z\ .\ regular\ y \wedge dense\ z \wedge x = y \sqcap z$
by (*metis pp-inf-sup-p dense-sup-p ppp*)

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

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

lemma *maddux-3-20*:
$((v \sqcap w) \sqcup (-v \sqcap x)) \sqcap -((v \sqcap y) \sqcup (-v \sqcap z)) = (v \sqcap w \sqcap -y) \sqcup (-v \sqcap x \sqcap -z)$
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-infI1 p-antitone-iff*)
also have ... $= v \sqcap w \sqcap -y$
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*:
$x \le y \longleftrightarrow x \le y \sqcup -x$
by (*metis inf.sup-left-isotone inf-sup-absorb le-supI1 maddux-3-12 sup-commute*)

lemma *order-char-2*:
$x \le y \longleftrightarrow x \sqcup -x \le y \sqcup -x$
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*]: $-x \sqcup --x = top$
begin

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*: $w \sqcap --x \le -y$
  by (*simp add*: *pseudo-complement*)
have $w = (w \sqcap -x) \sqcup (w \sqcap --x)$
  using *distrib-imp2 sup-inf-distrib1* **by** *auto*

thus $w \le -x \sqcup -y$
using *1* **by** (*metis inf-le2 sup.mono*)
next
fix w
assume $w \le -x \sqcup -y$
thus $w \sqcap (x \sqcap y) = bot$
using *order-trans p-supdist-inf pseudo-complement* **by** *blast*
qed

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

lemma *regular-closed-sup*:
regular $x \Longrightarrow$ *regular* $y \Longrightarrow$ *regular* $(x \sqcup y)$
by *simp*

The regular elements are precisely the ones having a complement.

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

lemma *selection-closed-sup*:
selection s $x \Longrightarrow$ *selection* t $x \Longrightarrow$ *selection* $(s \sqcup t)$ x
by (*simp add*: *inf-sup-distrib2*)

lemma *huntington-3-pp* [*simp*]:
$-(-x \sqcup -y) \sqcup -(-x \sqcup y) = --x$
by (*metis p-dist-inf p-inf sup.commute sup-bot-left sup-inf-distrib1*)

lemma *maddux-3-3* [*simp*]:
$-(x \sqcup y) \sqcup -(x \sqcup -y) = -x$
by (*simp add*: *sup-commute sup-inf-distrib1*)

lemma *maddux-3-11-pp*:
$(x \sqcap -y) \sqcup (x \sqcap --y) = x$
by (*metis inf-sup-distrib1 inf-top-right stone*)

lemma *maddux-3-19-pp*:
$(-x \sqcap y) \sqcup (--x \sqcap z) = (--x \sqcup y) \sqcap (-x \sqcup z)$
proof –
have $(--x \sqcup y) \sqcap (-x \sqcup z) = (--x \sqcap z) \sqcup (y \sqcap -x) \sqcup (y \sqcap z)$
by (*simp add*: *inf.commute inf-sup-distrib1 sup.assoc*)
also have ... $= (--x \sqcap z) \sqcup (y \sqcap -x) \sqcup (y \sqcap z \sqcap (-x \sqcup --x))$
by *simp*
also have ... $= (--x \sqcap z) \sqcup ((y \sqcap -x) \sqcup (y \sqcap -x \sqcap z)) \sqcup (y \sqcap z \sqcap --x)$
using *inf-sup-distrib1 sup-assoc inf-commute inf-assoc* **by** *presburger*
also have ... $= (--x \sqcap z) \sqcup (y \sqcap -x) \sqcup (y \sqcap z \sqcap --x)$
by *simp*

also have ... = $((--x \sqcap z) \sqcup (--x \sqcap z \sqcap y)) \sqcup (y \sqcap -x)$
by (*simp add*: *inf-assoc inf-commute sup.left-commute sup-commute*)
also have ... = $(--x \sqcap z) \sqcup (y \sqcap -x)$
by *simp*
finally show *?thesis*
by (*simp add*: *inf-commute sup-commute*)
qed

lemma *compl-inter-eq-pp*:
$--x \sqcap y = --x \sqcap z \Longrightarrow -x \sqcap y = -x \sqcap z \Longrightarrow y = z$
by (*metis inf-commute inf-p inf-sup-distrib1 inf-top.right-neutral p-bot p-dist-inf*)

lemma *maddux-3-21-pp* [*simp*]:
$--x \sqcup (-x \sqcap y) = --x \sqcup y$
by (*simp add*: *sup.commute sup-inf-distrib1*)

lemma *shunting-2-pp*:
$x \leq --y \longleftrightarrow -x \sqcup --y = top$
by (*metis inf-top-left p-bot p-dist-inf pseudo-complement*)

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

lemma *conjugate-char-2-pp*:
conjugate f g $\longleftrightarrow$ *f bot* = *bot* $\wedge$ *g bot* = *bot* $\wedge$ $(\forall x\ y\ .\ f\ x \sqcap y \leq --(f(x \sqcap --(g\ y))) \wedge g\ y \sqcap x \leq --(g(y \sqcap --(f\ x))))$
proof
assume *1*: *conjugate f g*
hence *2*: *dual-additive* (*uminus* $\circ$ *g*)

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

```
lemma conjugate-char-2-pp-additive:
  assumes conjugate f g
      and additive f
      and additive g
    shows f x ⊓ y ≤ f(x ⊓ −−(g y)) ∧ g y ⊓ x ≤ g(y ⊓ −−(f x))
proof −
  have f x ⊓ y = f ((x ⊓ −−g y) ⊔ (x ⊓ −g y)) ⊓ y
    by (simp add: sup.commute sup-inf-distrib1)
  also have ... = (f (x ⊓ −−g y) ⊓ y) ⊔ (f (x ⊓ −g y) ⊓ y)
    using assms(2) additive-def inf-sup-distrib2 by auto
  also have ... = f (x ⊓ −−g y) ⊓ y
    by (metis assms(1) conjugate-def inf-le2 pseudo-complement
sup-bot.right-neutral)
  finally have 2: f x ⊓ y ≤ f (x ⊓ −−g y)
    by simp
  have g y ⊓ x = g ((y ⊓ −−f x) ⊔ (y ⊓ −f x)) ⊓ x
    by (simp add: sup.commute sup-inf-distrib1)
  also have ... = (g (y ⊓ −−f x) ⊓ x) ⊔ (g (y ⊓ −f x) ⊓ x)
    using assms(3) additive-def inf-sup-distrib2 by auto
  also have ... = g (y ⊓ −−f x) ⊓ x
    by (metis assms(1) conjugate-def inf.cobounded2 pseudo-complement
sup-bot.right-neutral inf-commute)
  finally have g y ⊓ x ≤ g (y ⊓ −−f x)
    by simp
  thus ?thesis
    using 2 by simp
qed


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 $z = bot$ gives (non-relative) pseudocomplements. Many properties can already be shown if the underlying structure is just a semilattice.

```
class implies =
  fixes implies :: 'a ⇒ 'a ⇒ 'a (infixl ⇝ 65)

class heyting-semilattice = semilattice-inf + implies +
```

assumes *implies-galois*: $x \sqcap y \leq z \longleftrightarrow x \leq y \rightsquigarrow z$
begin

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

lemma *implies-increasing*:
$x \leq y \rightsquigarrow x$
by (*simp add*: *inf.orderI*)

lemma *implies-galois-swap*:
$x \leq y \rightsquigarrow z \longleftrightarrow y \leq x \rightsquigarrow z$
by (*metis implies-galois inf-commute*)

lemma *implies-galois-var*:
$x \sqcap y \leq z \longleftrightarrow y \leq x \rightsquigarrow z$
by (*simp add*: *implies-galois-swap implies-galois*)

lemma *implies-galois-increasing*:
$x \leq y \rightsquigarrow (x \sqcap y)$
using *implies-galois* **by** *blast*

lemma *implies-galois-decreasing*:
$(y \rightsquigarrow x) \sqcap y \leq x$
using *implies-galois* **by** *blast*

lemma *implies-mp-below*:
$x \sqcap (x \rightsquigarrow y) \leq y$
using *implies-galois-decreasing inf-commute* **by** *auto*

lemma *implies-isotone*:
$x \leq y \Longrightarrow z \rightsquigarrow x \leq z \rightsquigarrow y$
using *implies-galois order-trans* **by** *blast*

lemma *implies-antitone*:
$x \leq y \Longrightarrow y \rightsquigarrow z \leq x \rightsquigarrow z$
by (*meson implies-galois-swap order-lesseq-imp*)

lemma *implies-isotone-inf*:
$x \rightsquigarrow (y \sqcap z) \leq x \rightsquigarrow y$
by (*simp add*: *implies-isotone*)

lemma *implies-antitone-inf*:
$x \rightsquigarrow z \leq (x \sqcap y) \rightsquigarrow z$
by (*simp add*: *implies-antitone*)

lemma *implies-curry*:
$x \rightsquigarrow (y \rightsquigarrow z) = (x \sqcap y) \rightsquigarrow z$

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*]:
$((x \leadsto y) \leadsto y) \leadsto y = x \leadsto y$
using *implies-antitone implies-galois-swap eq-iff* **by** *auto*

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

lemma *implies-dist-inf*:
$x \leadsto (y \sqcap z) = (x \leadsto y) \sqcap (x \leadsto z)$
proof −
have $(x \leadsto y) \sqcap (x \leadsto z) \sqcap x \leq y \sqcap z$
by (*simp add*: *implies-galois*)
hence $(x \leadsto y) \sqcap (x \leadsto z) \leq x \leadsto (y \sqcap z)$
using *implies-galois* **by** *blast*
thus *?thesis*
by (*simp add*: *implies-isotone eq-iff*)
qed

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

lemma *inf-implies-top*:
$z \leq (x \sqcap y) \leadsto x$
using *implies-galois-var le-infI1* **by** *blast*

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

lemma *le-implies-top*:
$x \leq y \Longrightarrow z \leq x \leadsto y$
using *implies-antitone implies-itself-top order.trans* **by** *blast*

lemma *le-iff-le-implies*:
$x \le y \longleftrightarrow x \le x \rightsquigarrow y$
using *implies-galois inf-idem* **by** *force*

lemma *implies-inf-isotone*:
$x \rightsquigarrow y \le (x \sqcap z) \rightsquigarrow (y \sqcap z)$
by (*metis implies-curry implies-galois-increasing implies-isotone*)

lemma *implies-transitive*:
$(x \rightsquigarrow y) \sqcap (y \rightsquigarrow z) \le x \rightsquigarrow z$
using *implies-dist-implies implies-galois-var implies-increasing order-lesseq-imp*
by *blast*

lemma *implies-inf-absorb* [*simp*]:
$x \rightsquigarrow (x \sqcap y) = x \rightsquigarrow y$
using *implies-dist-inf implies-itself-top inf.absorb-iff2* **by** *auto*

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

lemma *implies-inf-identity*:
$(x \rightsquigarrow y) \sqcap y = y$
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*: $x \sqcap (x \rightsquigarrow y) \le y$
and *implies-galois-increasing*: $x \le y \rightsquigarrow (x \sqcap y)$
and *implies-isotone-inf*: $x \rightsquigarrow (y \sqcap z) \le x \rightsquigarrow y$
begin

subclass *heyting-semilattice*
apply *unfold-locales*
apply (*rule iffI*)
apply (*metis implies-galois-increasing implies-isotone-inf inf.absorb2 order-lesseq-imp*)
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*]:
  $x \leadsto x = top$
  using *implies-galois inf-le2 top-le* **by** *blast*

lemma *implies-order*:
  $x \leq y \longleftrightarrow x \leadsto y = top$
  by (*metis implies-galois inf-top.left-neutral top-unique*)

lemma *inf-implies* [*simp*]:
  $(x \sqcap y) \leadsto x = top$
  using *implies-order inf-le1* **by** *blast*

lemma *top-implies* [*simp*]:
  $top \leadsto x = x$
  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.

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 $x \sqcup z \leq y \leadsto (x \sqcup (y \sqcap z))$
    using *implies-galois-var implies-increasing sup.bounded-iff sup.cobounded2* **by** *blast*
  hence $x \sqcup y \leq (x \sqcup z) \leadsto (x \sqcup (y \sqcap z))$
    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*:
  $x \leadsto y \leq x \leadsto (y \sqcup z)$
  by (*simp add*: *implies-isotone*)

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

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

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

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

lemma *implies-antisymmetry*:
$(x \rightsquigarrow y) \sqcap (y \rightsquigarrow x) = (x \sqcup y) \rightsquigarrow (x \sqcap y)$
by (*metis implies-dist-sup implies-inf-absorb inf.commute*)

lemma *sup-inf-implies* [*simp*]:
$(x \sqcup y) \sqcap (x \rightsquigarrow y) = y$
by (*simp add*: *inf-sup-distrib2 sup.absorb2*)

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

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

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

lemma *sup-below-implies-implies*:
$x \sqcup y \leq (x \rightsquigarrow y) \rightsquigarrow y$
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*]:
  $bot \rightsquigarrow x = top$
  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*: $-x = x \rightsquigarrow bot$
begin

subclass *pd-algebra*
  apply *unfold-locales*
  using *bot-unique implies-galois uminus-eq* **by** *auto*

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

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

lemma *double-negation-dist-implies*:
  $--(x \rightsquigarrow y) = --x \rightsquigarrow --y$
  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*: $x \sqcap (x \leadsto y) = x \sqcap y$
and *implies-import-inf*: $x \sqcap ((x \sqcap y) \leadsto (x \leadsto z)) = x \sqcap (y \leadsto z)$
and *inf-inf-implies*: $z \sqcap ((x \sqcap y) \leadsto x) = z$
and *uminus-eq-eq*: $-x = x \leadsto bot$
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.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.

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).

class *brouwer-algebra* = *bounded-lattice* + *minus* + *uminus* +
assumes *minus-galois*: $x \le y \sqcup z \longleftrightarrow x - y \le z$
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*)

end

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*

sublocale *heyting*: *heyting-algebra* **where** *implies* = $\lambda x\ y\ .\ -x \sqcup y$
  apply *unfold-locales*
  apply (*rule iffI*)
  using *shunting-var-p sup-commute* **apply** *fastforce*
  using *shunting-var-p sup-commute* **apply** *force*

by *simp*

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*]:
$-(-x \sqcup -y) \sqcup -(-x \sqcup y) = x$
using *huntington-3-pp* **by** *auto*

lemma *maddux-3-1*:
$x \sqcup -x = y \sqcup -y$
by *simp*

lemma *maddux-3-4*:
$x \sqcup (y \sqcup -y) = z \sqcup -z$
by *simp*

lemma *maddux-3-11* [*simp*]:
$(x \sqcap y) \sqcup (x \sqcap -y) = x$
using *brouwer.maddux-3-12 sup.commute* **by** *auto*

lemma *maddux-3-19*:
$(-x \sqcap y) \sqcup (x \sqcap z) = (x \sqcup y) \sqcap (-x \sqcup z)$
using *maddux-3-19-pp* **by** *auto*

lemma *compl-inter-eq*:
$x \sqcap y = x \sqcap z \Longrightarrow -x \sqcap y = -x \sqcap z \Longrightarrow y = z$
by (*metis inf-commute maddux-3-11*)

lemma *maddux-3-21* [*simp*]:
$x \sqcup (-x \sqcap y) = x \sqcup y$
by (*simp add*: *sup-inf-distrib1*)

lemma *shunting-1*:
$x \leq y \longleftrightarrow x \sqcap -y = bot$
by (*simp add*: *pseudo-complement*)

lemma *uminus-involutive*:
$uminus \circ uminus = id$
by *auto*

lemma *uminus-injective*:
$uminus \circ f = uminus \circ g \Longrightarrow f = g$
by (*metis comp-assoc id-o minus-comp-minus*)

lemma *conjugate-unique*:
$conjugate\ f\ g \Longrightarrow conjugate\ f\ h \Longrightarrow g = h$

using *conjugate-unique-p uminus-injective* **by** *blast*

lemma *dual-additive-additive*:
dual-additive (*uminus* ∘ *f*) ⟹ *additive f*
by (*metis additive-def compl-eq-compl-iff dual-additive-def p-dist-sup o-def*)

lemma *conjugate-additive*:
conjugate f g ⟹ *additive f*
by (*simp add*: *conjugate-dual-additive dual-additive-additive*)

lemma *conjugate-isotone*:
conjugate f g ⟹ *isotone f*
by (*simp add*: *conjugate-additive additive-isotone*)

lemma *conjugate-char-1*:
$conjugate\ f\ g \longleftrightarrow (\forall x\ y\ .\ f(x \sqcap -(g\ y)) \leq f\ x \sqcap -y \wedge g(y \sqcap -(f\ x)) \leq g\ y \sqcap -x)$
by (*simp add*: *conjugate-char-1-pp*)

lemma *conjugate-char-2*:
$conjugate\ f\ g \longleftrightarrow f\ bot = bot \wedge g\ bot = bot \wedge (\forall x\ y\ .\ f\ x \sqcap y \leq f(x \sqcap g\ y) \wedge g\ y \sqcap x \leq g(y \sqcap f\ x))$
by (*simp add*: *conjugate-char-2-pp*)

lemma *shunting*:
$x \sqcap y \leq z \longleftrightarrow x \leq z \sqcup -y$
by (*simp add*: *heyting.implies-galois sup.commute*)

lemma *shunting-var*:
$x \sqcap -y \leq z \longleftrightarrow x \leq z \sqcup y$
by (*simp add*: *shunting*)

end

class *non-trivial-stone-algebra* = *non-trivial-bounded-order* + *stone-algebra*

class *non-trivial-boolean-algebra* = *non-trivial-stone-algebra* + *boolean-algebra*

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.

theory *Filters*

imports *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.

context *ord*
begin

abbreviation *down* :: *'a* ⇒ *'a set* (↓- [*81*] *80*)
  where $\downarrow x \equiv \{ y . y \leq x \}$

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

abbreviation *is-down-set* :: *'a set* ⇒ *bool*
  where *is-down-set* $X \equiv \forall x{\in}X . \forall y . y \leq x \longrightarrow y{\in}X$

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

abbreviation *up* :: *'a* ⇒ *'a set* (↑- [*81*] *80*)
  where $\uparrow x \equiv \{ y . x \leq y \}$

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

abbreviation *is-up-set* :: *'a set* ⇒ *bool*
  where *is-up-set* $X \equiv \forall x{\in}X . \forall y . x \leq y \longrightarrow y{\in}X$

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

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

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

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

abbreviation *ultra-filter* :: $'a$ *set* ⇒ *bool*
  where *ultra-filter* $F \equiv$ *proper-filter* $F \wedge (\forall G$. *proper-filter* $G \wedge F \subseteq G \longrightarrow F = G)$

end

context *order*
begin

lemma *self-in-downset* [*simp*]:
  $x \in {\downarrow}x$
  by *simp*

lemma *self-in-upset* [*simp*]:
  $x \in {\uparrow}x$
  by *simp*

lemma *up-filter* [*simp*]:
  filter $({\uparrow}x)$
  using *filter-def order-lesseq-imp* **by** *auto*

lemma *up-set-up-set* [*simp*]:
  is-up-set $({\Uparrow}X)$
  using *order.trans* **by** *fastforce*

lemma *up-injective*:
  ${\uparrow}x = {\uparrow}y \Longrightarrow x = y$
  using *antisym* **by** *auto*

lemma *up-antitone*:
  $x \leq y \longleftrightarrow {\uparrow}y \subseteq {\uparrow}x$
  by *auto*

end

context *order-bot*
begin

lemma *bot-in-downset* [*simp*]:
  bot $\in {\downarrow}x$
  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
      using chainsD2 by fastforce
    have 3: ⋃ ?D = ⋃ C
      by blast
    have ⋃ ?D ∈ ?A
    proof (cases ?D = {})
      assume ?D = {}
      thus ?thesis
        by auto
    next
      assume 4: ?D ≠ {}
      then obtain G where G ∈ ?D
        by auto
      hence 5: F ⊆ ⋃ ?D
        using 1 chainsD2 by blast
      have 6: is-up-set (⋃ ?D)
      proof
        fix x
        assume x ∈ ⋃ ?D
        then obtain H where x ∈ H ∧ H ∈ ?D ∧ filter H
          using 2 by auto
        thus ∀ y . x ≤ y ⟶ y∈⋃ ?D
          using filter-def UnionI by fastforce
      qed
      have 7: ⋃ ?D ≠ UNIV
      proof (rule ccontr)
```

assume ¬ ⋃ *?D* ≠ *UNIV*
then obtain *H* **where** *bot* ∈ *H* ∧ *proper-filter H*
using *2* **by** *blast*
thus *False*
by (*meson UNIV-I bot-least filter-def subsetI subset-antisym*)
qed
{
fix *x y*
assume *x*∈⋃ *?D* ∧ *y*∈⋃ *?D*
then obtain *H I* **where** *8*: *x* ∈ *H* ∧ *H* ∈ *?D* ∧ *filter H* ∧ *y* ∈ *I* ∧ *I* ∈ *?D* ∧ *filter I*
using *2* **by** *metis*
have ∃ *z*∈⋃ *?D* . *z* ≤ *x* ∧ *z* ≤ *y*
proof (*cases H* ⊆ *I*)
assume *H* ⊆ *I*
hence ∃ *z*∈*I* . *z* ≤ *x* ∧ *z* ≤ *y*
using *8* **by** (*metis subsetCE filter-def*)
thus *?thesis*
using *8* **by** (*metis UnionI*)
next
assume ¬ (*H* ⊆ *I*)
hence *I* ⊆ *H*
using *1 8* **by** (*meson DiffE chainsD*)
hence ∃ *z*∈*H* . *z* ≤ *x* ∧ *z* ≤ *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 ⋃ *C* ∈ *?A*
using *3* **by** *simp*
qed
hence ∃ *M*∈*?A* . ∀ *X*∈*?A* . *M* ⊆ *X* ⟶ *X* = *M*
by (*rule Zorn-Lemma*)
then obtain *M* **where** *9*: *M* ∈ *?A* ∧ (∀ *X*∈*?A* . *M* ⊆ *X* ⟶ *X* = *M*)
by *auto*
hence *10*: *M* ≠ {}
using *assms filter-def* **by** *auto*
{
fix *G*
assume *11*: *proper-filter G* ∧ *M* ⊆ *G*
hence *F* ⊆ *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*]:
$\downarrow$*top* = *UNIV*
by *simp*

lemma *top-in-upset* [*simp*]:
top $\in$ $\uparrow$*x*
by *simp*

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

lemma *filter-top* [*simp*]:
filter {*top*}
using *filter-def top-unique* **by** *auto*

lemma *top-in-filter* [*simp*]:
filter F $\Longrightarrow$ *top* $\in$ *F*
using *filter-def* **by** *fastforce*

end

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* $\vee$ $\uparrow y \neq$ *UNIV*
using *antisym* **by** *blast*
hence *proper-filter* ($\uparrow$*x*) $\vee$ *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* $G \wedge top \in G$
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* $\Rightarrow$ *bool*
where *prime-filter* $F \equiv$ *proper-filter* $F \wedge (\forall x\ y\ .\ x \sqcup y \in F \longrightarrow x \in F \vee y \in F)$

end

context *semilattice-inf*
begin

lemma *filter-inf-closed*:
filter $F \Longrightarrow x \in F \Longrightarrow y \in F \Longrightarrow x \sqcap y \in F$
by (*meson filter-def inf.boundedI*)

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 . $x \sqcap y \leq z$ }

lemma *filter-sup*:
  assumes *filter F*
    and *filter G*
   shows *filter* (*filter-sup F G*)
proof –
  have $F \neq \{\} \wedge G \neq \{\}$
   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* . $z \leq x \wedge z$
$\leq y$
  proof
   fix x
   assume x∈*filter-sup F G*
   then obtain t u **where** *3*: $t \in F \wedge u \in G \wedge t \sqcap u \leq x$
    by *auto*
   show ∀y∈*filter-sup F G* . ∃z∈*filter-sup F G* . $z \leq x \wedge z \leq y$
   proof
    fix y
    assume y∈*filter-sup F G*
    then obtain v w **where** *4*: $v \in F \wedge w \in G \wedge v \sqcap w \leq y$
     by *auto*
    let *?z* = $(t \sqcap v) \sqcap (u \sqcap w)$
    have *5*: *?z* ≤ x ∧ *?z* ≤ y
     using *3 4* **by** (*meson order.trans inf.cobounded1 inf.cobounded2 inf-mono*)
    have *?z* ∈ *filter-sup F G*
     using *assms 3 4 filter-inf-closed* **by** *blast*
    thus ∃z∈*filter-sup F G* . $z \leq x \wedge z \leq y$
     using *5* **by** *blast*
   qed
  qed
  have ∀x∈*filter-sup F G* . ∀y . $x \leq y \longrightarrow y \in$ *filter-sup F G*
   using *order-trans* **by** *blast*
  thus *?thesis*
   using *1 2 filter-def* **by** *presburger*
qed

lemma *filter-sup-left-upper-bound*:
  assumes *filter G*
   shows $F \subseteq$ *filter-sup F G*
proof –
  from *assms* **obtain** y **where** y∈G
   using *all-not-in-conv filter-def* **by** *auto*
  thus *?thesis*
   using *inf.cobounded1* **by** *blast*
qed

lemma *filter-sup-symmetric*:
  filter-sup F G = *filter-sup G F*
  using *inf.commute* **by** *fastforce*

lemma *filter-sup-right-upper-bound*:
  filter F $\Longrightarrow$ $G \subseteq$ *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 *filter-sup F G* $\subseteq H$
proof
  fix x
  assume $x \in$ *filter-sup F G*
  then obtain y z **where** *1*: $y \in F \wedge z \in G \wedge y \sqcap z \leq x$
    by *auto*
  hence $y \in H \wedge z \in H$
    using *assms*(*2*−*3*) **by** *auto*
  hence $y \sqcap z \in H$
    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*:
  $G \subseteq H \Longrightarrow$ *filter-sup G F* $\subseteq$ *filter-sup H F*
  by *blast*

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

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

lemma *up-dist-inf*:
  $\uparrow(x \sqcap y)$ = *filter-sup* $(\uparrow x)$ $(\uparrow y)$
proof
  show $\uparrow(x \sqcap y) \subseteq$ *filter-sup* $(\uparrow x)$ $(\uparrow y)$
    by *blast*
next
  show *filter-sup* $(\uparrow x)$ $(\uparrow y) \subseteq \uparrow(x \sqcap y)$
  proof
    fix z
    assume $z \in$ *filter-sup* $(\uparrow x)$ $(\uparrow y)$

then obtain u v **where** $u{\in}{\uparrow}x \wedge v{\in}{\uparrow}y \wedge u \sqcap v \leq z$
by *auto*
hence $x \sqcap y \leq z$
using *order.trans inf-mono* **by** *blast*
thus $z \in {\uparrow}(x \sqcap y)$
by *blast*
qed
qed

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

lemma *filter-inf-filter* [*simp*]:
assumes *filter F*
shows *filter* $({\Uparrow}\{ y . \exists z{\in}F . x \sqcap z = y\})$
proof –
let $?G = {\Uparrow}\{ y . \exists z{\in}F . x \sqcap z = y\}$
have $F \neq \{\}$
using *assms filter-def* **by** *simp*
hence *1*: $?G \neq \{\}$
by *blast*
have *2*: *is-up-set ?G*
by *auto*
{
fix y z
assume $y \in ?G \wedge z \in ?G$
then obtain v w **where** $v \in F \wedge w \in F \wedge x \sqcap v \leq y \wedge x \sqcap w \leq z$
by *auto*
hence $v \sqcap w \in F \wedge x \sqcap (v \sqcap w) \leq y \sqcap z$
by (*meson assms filter-inf-closed order.trans inf.boundedI inf.cobounded1 inf.cobounded2*)
hence $\exists u{\in}?G . u \leq y \wedge u \leq z$
by *auto*
}
hence $\forall x{\in}?G . \forall y{\in}?G . \exists z{\in}?G . z \leq x \wedge z \leq y$
by *auto*
thus *?thesis*
using *1 2 filter-def* **by** *presburger*
qed

end

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*: $x{\in}F \wedge y{\in}G$
  using *all-not-in-conv filter-def* **by** *auto*
from *ub* **obtain** *z* **where** $x \le z \wedge y \le z$
  by *auto*
hence $z \in F \cap G$
  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 $\forall x{\in}F \cap G . \forall y{\in}F \cap G . \exists z{\in}F \cap G . z \le x \wedge z \le y$
    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*: *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* $F\ G \equiv filter\ F \wedge filter\ G \wedge filter\text{-}sup\ F\ G = UNIV \wedge F \cap G = \{top\}$

end

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

context *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* (↑*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-def inf-filter.abs-eq up-dist-sup up-filter*)

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

lemma *up-filter-antitone*:
  x ≤ *y* ⟷ *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*
  where *filter-map f F* ≡ *Abs-filter* (*f* ‘ *Rep-filter F*)

lemma *filter-map-filter*:
  assumes *mono f*
    and ∀ *x y* . *f x* ≤ *y* ⟶ (∃ *z* . *x* ≤ *z* ∧ *y* = *f z*)
   shows *filter* (*f* ‘ *Rep-filter F*)
proof (*unfold filter-def*, *intro conjI*)
  show *f* ‘ *Rep-filter F* ≠ {}
   by (*metis empty-is-image filter-def simp-filter*)
next
  show ∀ *x*∈*f* ‘ *Rep-filter F* . ∀ *y*∈*f* ‘ *Rep-filter F* . ∃ *z*∈*f* ‘ *Rep-filter F* . *z* ≤ *x* ∧ *z* ≤ *y*
  proof (*intro ballI*)
   fix *x y*
   assume *x* ∈ *f* ‘ *Rep-filter F* **and** *y* ∈ *f* ‘ *Rep-filter F*
   then obtain *u v* **where** *1*: *x* = *f u* ∧ *u* ∈ *Rep-filter F* ∧ *y* = *f v* ∧ *v* ∈ *Rep-filter F*
    by *auto*
   then obtain *w* **where** *w* ≤ *u* ∧ *w* ≤ *v* ∧ *w* ∈ *Rep-filter F*
    by (*meson filter-def simp-filter*)
   thus ∃ *z*∈*f* ‘ *Rep-filter F* . *z* ≤ *x* ∧ *z* ≤ *y*
    using *1 assms*(*1*) *mono-def rev-image-eqI* **by** *blast*
  qed
next
  show *is-up-set* (*f* ‘ *Rep-filter F*)
  proof

fix x
assume $x \in f$ ‘ *Rep-filter* F
then obtain u **where** *1*: $x = f\ u \wedge u \in$ *Rep-filter* F
by *auto*
show $\forall y . x \leq y \longrightarrow y \in f$ ‘ *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** $u \leq z \wedge y = f\ z$
using *assms*(*2*) **by** *auto*
thus $y \in f$ ‘ *Rep-filter* F
using *1* **by** (*meson image-iff filter-def simp-filter*)
qed
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.

context *distrib-lattice*
begin

lemma *filter-sup-left-dist-inf*:
assumes *filter* F
and *filter* G
and *filter* H
shows *filter-sup* F $(G \cap H) =$ *filter-sup* F $G \cap$ *filter-sup* F H
proof
show *filter-sup* F $(G \cap H) \subseteq$ *filter-sup* F $G \cap$ *filter-sup* F H
using *filter-sup-right-isotone-var* **by** *blast*
next
show *filter-sup* F $G \cap$ *filter-sup* F $H \subseteq$ *filter-sup* F $(G \cap H)$
proof
fix x
assume $x \in$ *filter-sup* F $G \cap$ *filter-sup* F H
then obtain $t\ u\ v\ w$ **where** *1*: $t \in F \wedge u \in G \wedge v \in F \wedge w \in H \wedge t \sqcap u \leq x \wedge v \sqcap w \leq x$
by *auto*
let $?y = t \sqcap v$
let $?z = u \sqcup w$
have *2*: $?y \in F$
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*
using *1* **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* ⊔ *z* ∈ *F* ∧ *y* ⊔ *z* ∈ *G*
by (*meson assms*(*1*−*2*) *sup-ge1 filter-def*)
have (*x* ⊔ *z*) ⊓ (*y* ⊔ *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* ∩ *G*)
shows *is-principal-up F*
proof −
from *assms*(*3*−*4*) **obtain** *x y* **where** *1*: *filter-sup F G* = ↑*x* ∧ *F* ∩ *G* = ↑*y*
by *blast*
from *filter-inf-principal-rep* **obtain** *t u* **where** *2*: *t*∈*F* ∧ *u*∈*G* ∧ *y* = *t* ⊔ *u*
using *1* **by** *meson*
from *filter-sup-principal-rep* **obtain** *v w* **where** *3*: *v*∈*F* ∧ *w*∈*G* ∧ *x* = *v* ⊓ *w*
using *1* **by** (*meson assms*(*1*−*2*))
have *t* ∈ *filter-sup F G* ∧ *u* ∈ *filter-sup F G*
using *2 inf.cobounded1 inf.cobounded2* **by** *blast*
hence *x* ≤ *t* ∧ *x* ≤ *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 $(t \sqcap v) \sqcup (u \sqcap w) \in F \wedge (t \sqcap v) \sqcup (u \sqcap w) \in G$
using *2 3* **by** (*metis* (*no-types*, *lifting*) *assms*(*1*−*2*) *filter-inf-closed sup.cobounded1 sup.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.coboundedI1*)
have $F = \uparrow(t \sqcap v)$
proof
show $F \subseteq \uparrow(t \sqcap v)$
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*: $(t \sqcap v \sqcap z) \sqcap (u \sqcap w) = x$
using *4* **by** (*metis inf.absorb1 inf.assoc inf.commute*)
have $z \sqcup u \in F \wedge z \sqcup u \in G \wedge z \sqcup w \in F \wedge z \sqcup w \in G$
using *2 3 6* **by** (*meson assms*(*1*−*2*) *filter-def sup-ge1 sup-ge2*)
hence $y \leq (z \sqcup u) \sqcap (z \sqcup w)$
using *1 Int-iff filter-inf-closed* **by** *auto*
hence *8*: $(t \sqcap v \sqcap z) \sqcup (u \sqcap w) = y$
using *5* **by** (*metis inf.absorb1 sup.commute sup-inf-distrib2*)
have $t \sqcap v \sqcap z = t \sqcap v$
using *4 5 7 8 relative-equality* **by** *blast*
thus $z \in \uparrow(t \sqcap v)$
by (*simp add*: *inf.orderI*)
qed
next
show $\uparrow(t \sqcap v) \subseteq F$
proof
fix z
have *9*: $t \sqcap v \in F$
using *2 3* **by** (*simp add*: *assms*(*1*) *filter-inf-closed*)
assume $z \in \uparrow(t \sqcap v)$
hence $t \sqcap v \leq z$ **by** *simp*
thus $z \in F$
using *assms*(*1*) *9 filter-def* **by** *auto*
qed
qed
thus *?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* ∩ *G*)
   shows *is-principal-up F* ∧ *is-principal-up G*
proof –
  have *filter G* ∧ *filter F* ∧ *is-principal-up* (*filter-sup G F*) ∧ *is-principal-up* (*G* ∩ *F*)
   by (*simp add*: *assms Int-commute filter-sup-symmetric*)
  thus *?thesis*
   using *assms*(*3*) *inf-sup-principal-aux* **by** *blast*
qed

lemma *filter-sup-absorb-inf*: *filter F* ⟹ *filter G* ⟹ *filter-sup* (*F* ∩ *G*) *G* = *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* ⟹ *filter G* ⟹ *filter-sup F G* ∩ *G* = *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 –
  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 show *?thesis*
   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 –

```
  {
    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)
  }
  thus ?thesis
    using assms by blast
qed

end

context distrib-lattice-bot
begin

lemma prime-filter:
  proper-filter F ⟹ ∃ G . prime-filter G ∧ F ⊆ G
  by (metis ultra-filter ultra-filter-prime)

end

context distrib-lattice-top
begin

lemma complemented-filter-inf-principal:
  assumes filter-complements F G
    shows is-principal-up (F ∩ ↑x)
proof −
  have 1: filter F ∧ filter G
    by (simp add: assms)
  hence 2: filter (F ∩ ↑x) ∧ filter (G ∩ ↑x)
    by (simp add: filter-inf)
```

have $(F \cap \uparrow x) \cap (G \cap \uparrow x) = \{top\}$
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* $((F \cap \uparrow x) \cap (G \cap \uparrow x))$
using *up-top* **by** *blast*
have *filter-sup* $(F \cap \uparrow x)$ $(G \cap \uparrow x)$ = *filter-sup* F $G \cap \uparrow x$
using *1 filter-inf-right-dist-sup up-filter* **by** *auto*
also have ... = $\uparrow x$
by (*simp add*: *assms*)
finally have *is-principal-up* (*filter-sup* $(F \cap \uparrow x)$ $(G \cap \uparrow x)$)
by *auto*
thus *?thesis*
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
proof
fix x y z :: *'a filter*
have *Rep-filter* $(x \sqcup (y \sqcap z))$ = *filter-sup* (*Rep-filter* x) (*Rep-filter* $(y \sqcap z)$)
by (*simp add*: *sup-filter.rep-eq*)
also have ... = *filter-sup* (*Rep-filter* x) (*Rep-filter* y $\cap$ *Rep-filter* z)
by (*simp add*: *inf-filter.rep-eq*)
also have ... = *filter-sup* (*Rep-filter* x) (*Rep-filter* y) $\cap$ *filter-sup* (*Rep-filter* x) (*Rep-filter* z)
by (*simp add*: *filter-sup-left-dist-inf*)
also have ... = *Rep-filter* $(x \sqcup y)$ $\cap$ *Rep-filter* $(x \sqcup z)$
by (*simp add*: *sup-filter.rep-eq*)
also have ... = *Rep-filter* $((x \sqcup y) \sqcap (x \sqcup z))$
by (*simp add*: *inf-filter.rep-eq*)
finally show $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
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 φ, 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* ⇒ '*b*::*sup*) ⇒ *bool*
  where *sup-homomorphism f* ≡ ∀ *x y* . *f* (*x* ⊔ *y*) = *f x* ⊔ *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*,*inf*}) ⇒ *bool*
  where *sup-inf-homomorphism f* ≡ *sup-homomorphism f* ∧ *inf-homomorphism f*

abbreviation *sup-inf-top-homomorphism* :: ('*a*::{*sup*,*inf*,*top*} ⇒ '*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*,*top*,*bot*} ⇒ '*b*::{*sup*,*inf*,*top*,*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-top-bot-uminus* ⇒ '*b*::*sup-inf-top-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-top-bot-uminus-ord* ⇒ 'b::*sup-inf-top-bot-uminus-ord*) ⇒ *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 assms 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*(*2*))
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** λ*x y* . *x* ⊓ −*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* **.**

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

instance
  apply *intro-classes*
  apply (*simp add*: *less-eq-regular.rep-eq less-regular.rep-eq inf.less-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-regular.rep-eq less-eq-regular.rep-eq*)
  apply (*simp add*: *inf-regular.rep-eq less-eq-regular.rep-eq*)
  apply (*simp add*: *inf-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*: *bot-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*)

end

instantiation *regular* :: (*non-trivial-stone-algebra*) *non-trivial-boolean-algebra*
begin

instance
proof (*intro-classes*, *rule ccontr*)
  assume ¬(∃ *x y*::'*a regular* . *x* ≠ *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*]:
$-$*Rep-dense* x = *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* ⇒ *'a dense* ⇒ *'a dense* **is** *sup*
by *simp*

lift-definition *inf-dense* :: *'a dense* ⇒ *'a dense* ⇒ *'a dense* **is** *inf*
by *simp*

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

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

lift-definition *less-dense* :: *'a dense* ⇒ *'a dense* ⇒ *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*: *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 $(x \sqcup -x \sqcup y) \wedge$ *dense* $(x \sqcup -x \sqcup y \sqcup z)$
by *simp*

lemma *up-filter-dense-antitone*:
up-filter (*Abs-dense* $(x \sqcup -x \sqcup y \sqcup z)$) $\leq$ *up-filter* (*Abs-dense* $(x \sqcup -x \sqcup y)$)
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* ⇒ *bool* **is** *less-eq* **.**

lift-definition *less-dense-filter-type* :: *'a dense-filter-type* ⇒ *'a dense-filter-type* ⇒ *bool* **is** *less* **.**

instance
  apply *intro-classes*
  apply (*simp add*: *less-eq-dense-filter-type.rep-eq less-dense-filter-type.rep-eq inf.less-le-not-le*)
  apply (*simp add*: *less-eq-dense-filter-type.rep-eq*)
  apply (*simp add*: *less-eq-dense-filter-type.rep-eq inf.order-lesseq-imp*)
  apply (*simp add*: *Rep-dense-filter-type-inject less-eq-dense-filter-type.rep-eq*)
  apply (*simp add*: *inf-dense-filter-type.rep-eq less-eq-dense-filter-type.rep-eq*)
  apply (*simp add*: *inf-dense-filter-type.rep-eq less-eq-dense-filter-type.rep-eq*)
  apply (*simp add*: *inf-dense-filter-type.rep-eq less-eq-dense-filter-type.rep-eq*)
  apply (*simp add*: *less-eq-dense-filter-type.rep-eq sup-dense-filter-type.rep-eq*)
  apply (*simp add*: *less-eq-dense-filter-type.rep-eq sup-dense-filter-type.rep-eq*)
  apply (*simp add*: *less-eq-dense-filter-type.rep-eq sup-dense-filter-type.rep-eq*)
  apply (*simp add*: *less-eq-dense-filter-type.rep-eq bot-dense-filter-type.rep-eq*)
  apply (*simp add*: *top-dense-filter-type.rep-eq less-eq-dense-filter-type.rep-eq*)
  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.

abbreviation *stone-phi-set* :: *'a::stone-algebra regular* ⇒ *'a dense set*
  where *stone-phi-set x* ≡ { *y* . −*Rep-regular x* ≤ *Rep-dense y* }

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

definition *stone-phi* :: *'a::stone-algebra regular* ⇒ *'a dense-filter*
  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.

interpretation *stone-phi*: *triple stone-phi*
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*
    apply (*unfold stone-phi-def bot-regular-def 1 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 (*metis 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.abs-eq*)
      also have ... = *Abs-dense* (−*Rep-regular* (*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 $z \in$ *filter-sup* (*stone-phi-set* x) (*stone-phi-set* y)
using *3 mem-Collect-eq order-refl* **by** *fastforce*
next
fix z
assume $z \in$ *filter-sup* (*stone-phi-set* x) (*stone-phi-set* y)
then obtain v w **where** *4*: $v \in$ *stone-phi-set* $x \wedge w \in$ *stone-phi-set* $y \wedge v \sqcap w \leq z$
by *auto*
have $-$*Rep-regular* $(x \sqcup y)$ = *Rep-regular* $(-(x \sqcup y))$
by (*metis uminus-regular.rep-eq*)
also have ... = $-$*Rep-regular* $x \sqcap -$*Rep-regular* y
by (*simp add*: *inf-regular.rep-eq uminus-regular.rep-eq*)
also have ... $\leq$ *Rep-dense* $v \sqcap$ *Rep-dense* w
using *4 inf-mono mem-Collect-eq* **by** *blast*
also have ... = *Rep-dense* $(v \sqcap w)$
by (*simp add*: *inf-dense.rep-eq*)
also have ... $\leq$ *Rep-dense* z
using *4* **by** (*simp add*: *less-eq-dense.rep-eq*)
finally show $z \in$ *stone-phi-set* $(x \sqcup y)$
by *simp*
qed
thus *stone-phi* $(x \sqcup y)$ = *stone-phi* $x \sqcup$ *stone-phi* y
by (*simp add*: *stone-phi-def eq-onp-same-args stone-phi-set-filter sup-filter.abs-eq*)
qed
next
show $\forall x\ y$::'*a regular* . *stone-phi* $(x \sqcap y)$ = *stone-phi* $x \sqcap$ *stone-phi* y
proof (*intro allI*)
fix x y :: '*a regular*
have $\forall z$. $-$*Rep-regular* $(x \sqcap y) \leq$ *Rep-dense* $z \longleftrightarrow -$*Rep-regular* $x \leq$ *Rep-dense* $z \wedge -$*Rep-regular* $y \leq$ *Rep-dense* z
by (*simp add*: *inf-regular.rep-eq*)
hence *stone-phi-set* $(x \sqcap y)$ = (*stone-phi-set* x) $\cap$ (*stone-phi-set* y)
by *auto*
thus *stone-phi* $(x \sqcap y)$ = *stone-phi* $x \sqcap$ *stone-phi* y
by (*simp add*: *stone-phi-def eq-onp-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 = *Abs-filter* {*top*}
by (*metis hom bot-filter-def*)

lemma *phi-top*:
phi top = *Abs-filter 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* :: (*'a* × *'b filter*) *set*
where *pairs* = { (*x*,*y*) . ∃ *z* . *y* = *phi* (−*x*) ⊔ *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* :: (*'a* × *'b filter*) ⇒ (*'a* × *'b filter*) ⇒ *bool*
where *pairs-less-eq* (*x*,*y*) (*z*,*w*) = (*x* ≤ *z* ∧ *w* ≤ *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)
    by (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
  also have ... = phi x ⊓ up-filter y
    by (simp add: Rep-filter-inverse)
  finally show ?thesis
    by auto
qed
```

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.

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

lemma rho-char:
  up-filter (rho x y) = phi x ⊓ up-filter y
  by (metis (mono-tags) someI-ex rho.simps phi-inf-principal)
```

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

```
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
    using pairs-def by auto
  hence pairs-sup (x,y) (z,w) = (x ⊔ z,(phi (-x) ⊔ up-filter u) ⊓ (phi (-z) ⊔
up-filter v))
    by simp
```

also have ... = $(x \sqcup z, (phi\ (-x) \sqcap phi\ (-z)) \sqcup (phi\ (-x) \sqcap up\text{-}filter\ v) \sqcup (up\text{-}filter\ u \sqcap phi\ (-z)) \sqcup (up\text{-}filter\ u \sqcap up\text{-}filter\ v))$
by (*simp add*: *inf.sup-commute inf-sup-distrib1 sup-commute sup-left-commute*)
also have ... = $(x \sqcup z, phi\ (-(x \sqcup z)) \sqcup (phi\ (-x) \sqcap up\text{-}filter\ v) \sqcup (up\text{-}filter\ u \sqcap phi\ (-z)) \sqcup (up\text{-}filter\ u \sqcap up\text{-}filter\ v))$
using *hom* **by** *simp*
also have ... = $(x \sqcup z, phi\ (-(x \sqcup z)) \sqcup up\text{-}filter\ (rho\ (-x)\ v) \sqcup up\text{-}filter\ (rho\ (-z)\ u) \sqcup (up\text{-}filter\ u \sqcap up\text{-}filter\ v))$
by (*metis inf.sup-commute rho-char*)
also have ... = $(x \sqcup z, phi\ (-(x \sqcup z)) \sqcup up\text{-}filter\ (rho\ (-x)\ v) \sqcup up\text{-}filter\ (rho\ (-z)\ u) \sqcup up\text{-}filter\ (u \sqcup v))$
by (*metis up-filter-dist-sup*)
also have ... = $(x \sqcup z, phi\ (-(x \sqcup z)) \sqcup up\text{-}filter\ (rho\ (-x)\ v \sqcap rho\ (-z)\ u \sqcap (u \sqcup v)))$
by (*simp add*: *sup-commute sup-left-commute up-filter-dist-inf*)
finally show *?thesis*
using *pairs-def* **by** *auto*
qed

lemma *pairs-inf-closed*:
assumes $(x,y) \in pairs$
and $(z,w) \in pairs$
shows $pairs\text{-}inf\ (x,y)\ (z,w) \in pairs$
proof −
from *assms* **obtain** *u v* **where** $y = phi\ (-x) \sqcup up\text{-}filter\ u \wedge w = phi\ (-z) \sqcup up\text{-}filter\ v$
using *pairs-def* **by** *auto*
hence $pairs\text{-}inf\ (x,y)\ (z,w) = (x \sqcap z, (phi\ (-x) \sqcup up\text{-}filter\ u) \sqcup (phi\ (-z) \sqcup up\text{-}filter\ v))$
by *simp*
also have ... = $(x \sqcap z, (phi\ (-x) \sqcup phi\ (-z)) \sqcup (up\text{-}filter\ u \sqcup up\text{-}filter\ v))$
by (*simp add*: *sup-commute sup-left-commute*)
also have ... = $(x \sqcap z, phi\ (-(x \sqcap z)) \sqcup (up\text{-}filter\ u \sqcup up\text{-}filter\ v))$
using *hom* **by** *simp*
also have ... = $(x \sqcap z, phi\ (-(x \sqcap z)) \sqcup up\text{-}filter\ (u \sqcap v))$
by (*simp add*: *up-filter-dist-inf*)
finally show *?thesis*
using *pairs-def* **by** *auto*
qed

lemma *pairs-uminus-closed*:
$pairs\text{-}uminus\ (x,y) \in pairs$
proof −
have $pairs\text{-}uminus\ (x,y) = (-x, phi\ (--x) \sqcup bot)$
by *simp*
also have ... = $(-x, phi\ (--x) \sqcup up\text{-}filter\ top)$
by (*simp add*: *bot-filter.abs-eq*)
finally show *?thesis*

by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq old.prod.case pairs-def*)
qed

lemma *pairs-bot-closed*:
pairs-bot $\in$ *pairs*
using *pairs-def phi-top triple.hom triple-axioms* **by** *fastforce*

lemma *pairs-top-closed*:
pairs-top $\in$ *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*:
$(x,y) \in$ *pairs* $\Longrightarrow (z,w) \in$ *pairs* $\Longrightarrow (u,v) \in$ *pairs* $\Longrightarrow$ *pairs-sup* (x,y) (*pairs-inf* (z,w) (u,v)) = *pairs-inf* (*pairs-sup* (x,y) (z,w)) (*pairs-sup* (x,y) (u,v))
using *sup-inf-distrib1 inf-sup-distrib1* **by** *auto*

lemma *pairs-phi-less-eq*:
$(x,y) \in$ *pairs* $\Longrightarrow$ *phi* $(-x) \leq y$
using *pairs-def* **by** *auto*

lemma *pairs-uminus-galois*:
assumes $(x,y) \in$ *pairs*
and $(z,w) \in$ *pairs*
shows *pairs-inf* (x,y) (z,w) = *pairs-bot* $\longleftrightarrow$ *pairs-less-eq* (x,y) (*pairs-uminus* (z,w))
proof –
have *1*: $x \sqcap z = bot \wedge y \sqcup w =$ *Abs-filter UNIV* $\longrightarrow$ *phi* $z \leq y$
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *heyting.implies-inf-absorb hom le-supE pairs-phi-less-eq sup-bot-right*)
have *2*: $x \leq -z \wedge$ *phi* $z \leq y \longrightarrow y \sqcup w =$ *Abs-filter UNIV*
proof
assume *3*: $x \leq -z \wedge$ *phi* $z \leq y$
have *Abs-filter UNIV* = *phi* $z \sqcup$ *phi* $(-z)$
using *hom phi-complemented phi-top* **by** *auto*
also have ... $\leq y \sqcup w$
using *3 assms*(*2*) *sup-mono pairs-phi-less-eq* **by** *auto*
finally show $y \sqcup w =$ *Abs-filter UNIV*
using *hom phi-top top.extremum-uniqueI* **by** *auto*
qed
have $x \sqcap z = bot \longleftrightarrow x \leq -z$
by (*simp add*: *shunting-1*)
thus *?thesis*
using *1 2 Pair-inject pairs-inf.simps pairs-less-eq.simps pairs-uminus.simps* **by** *auto*
qed

lemma *pairs-stone*:

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

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

lemma *regular-pairs*:
$regular\text{-}pairs = \{\ (x,y)\ .\ y = phi\ (-x)\ \}$
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* :: $'a \times 'b\ filter \Rightarrow 'b$
where $rho\text{-}pair\ (x,y) = (SOME\ z\ .\ up\text{-}filter\ z = phi\ x \sqcap y)$

lemma *get-rho-pair-char*:
assumes $(x,y) \in pairs$
shows $up\text{-}filter\ (rho\text{-}pair\ (x,y)) = phi\ x \sqcap y$
proof −
from *assms* **obtain** w **where** $y = phi\ (-x) \sqcup up\text{-}filter\ w$
using *pairs-def* **by** *auto*
hence $phi\ x \sqcap y = phi\ x \sqcap up\text{-}filter\ w$
by (*simp add*: *inf-sup-distrib1 phi-complemented*)
thus *?thesis*
using *rho-char* **by** *auto*
qed

lemma *sa-iso-pair*:
$(--x,phi\ (-x) \sqcup up\text{-}filter\ y) \in pairs$
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 ?f = λx . if x∈F then top else bot::'b filter
  have bounded-lattice-homomorphism ?f
  proof (intro conjI)
    show ?f bot = bot
      using 1 by (meson bot.extremum filter-def subset-eq top.extremum-unique)
  next
    show ?f top = top
      using 1 by simp
  next
    show ∀x y . ?f (x ⊔ y) = ?f x ⊔ ?f y
    proof (intro allI)
      fix x y
      show ?f (x ⊔ y) = ?f x ⊔ ?f y
        apply (cases x ∈ F; cases y ∈ 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 ∀x y . ?f (x ⊓ y) = ?f x ⊓ ?f y
    proof (intro allI)
      fix x y
      show ?f (x ⊓ y) = ?f x ⊓ ?f y
        apply (cases x ∈ F; cases y ∈ 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 *?f* ∈ {*f* . *bounded-lattice-homomorphism f*}
  by *simp*
 thus *?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.

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** λ*xf 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** λ*xf 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** λ*xf 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** λ*f* . *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** λ*f* . *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** λ*xf 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** λ*xf 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 $xf \leq xf$
by (*simp add*: *less-eq-lifted-pair.rep-eq*)
next
fix *xf yf zf* :: ('*a*,'*b*) *lifted-pair*
assume *1*: $xf \leq yf$ **and** *2*: $yf \leq zf$
{
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 $xf \leq zf$
by (*simp add*: *less-eq-lifted-pair.rep-eq*)
next
fix *xf yf* :: ('*a*,'*b*) *lifted-pair*
assume *1*: $xf \leq yf$ **and** *2*: $yf \leq xf$
{
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*
obtain *x1 x2 y1 y2* **where** *4*: (*x1*,*x2*) = *?x* ∧ (*y1*,*y2*) = *?y*
using *prod.collapse* **by** *blast*
have *triple.pairs-less-eq ?x ?y* ∧ *triple.pairs-less-eq ?y ?x*
using *1 2 3 less-eq-lifted-pair.rep-eq* **by** *simp*
hence *?x* = *?y*
using *3 4* **by** (*metis* (*mono-tags*, *lifting*) *triple.pairs-less-eq.simps antisym*)
}
thus *xf* = *yf*
by (*metis Rep-lifted-pair-inverse ext*)
next
fix *xf yf* :: ('*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*

```
    let ?y = Rep-lifted-pair yf f
    obtain x1 x2 y1 y2 where (x1,x2) = ?x ∧ (y1,y2) = ?y
      using prod.collapse by blast
    hence triple.pairs-less-eq (triple.pairs-inf ?x ?y) ?y
      using 1 by (metis (mono-tags, lifting) inf-sup-ord(2) sup.cobounded2
triple.pairs-inf.simps triple.pairs-less-eq.simps inf-lifted-pair.rep-eq)
  }
  thus xf ⊓ yf ≤ yf
    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: triple (Rep-phi f)
      by (simp add: triple-def)
    let ?x = Rep-lifted-pair xf f
    let ?y = Rep-lifted-pair yf f
    obtain x1 x2 y1 y2 where (x1,x2) = ?x ∧ (y1,y2) = ?y
      using prod.collapse by blast
    hence triple.pairs-less-eq (triple.pairs-inf ?x ?y) ?x
      using 1 by (metis (mono-tags, lifting) inf-sup-ord(1) sup.cobounded1
triple.pairs-inf.simps triple.pairs-less-eq.simps inf-lifted-pair.rep-eq)
  }
  thus xf ⊓ yf ≤ 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 ≤ yf and 2: xf ≤ 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 ?x ?z
      using 1 2 3 less-eq-lifted-pair.rep-eq by simp
    hence triple.pairs-less-eq ?x (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 ≤ yf ⊓ 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*: *triple* (*Rep-phi f*)
by (*simp add*: *triple-def*)
let *?x* = *Rep-lifted-pair xf f*
let *?y* = *Rep-lifted-pair yf f*
obtain *x1 x2 y1 y2* **where** (*x1*,*x2*) = *?x* ∧ (*y1*,*y2*) = *?y*
using *prod.collapse* **by** *blast*
hence *triple.pairs-less-eq ?x* (*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 sup-lifted-pair.rep-eq*)
}
thus *xf* ≤ *xf* ⊔ *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*: *triple* (*Rep-phi f*)
by (*simp add*: *triple-def*)
let *?x* = *Rep-lifted-pair xf f*
let *?y* = *Rep-lifted-pair yf f*
obtain *x1 x2 y1 y2* **where** (*x1*,*x2*) = *?x* ∧ (*y1*,*y2*) = *?y*
using *prod.collapse* **by** *blast*
hence *triple.pairs-less-eq ?y* (*triple.pairs-sup ?x ?y*)
using *1* **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* ≤ *xf* ⊔ *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* ≤ *xf* **and** *2*: *zf* ≤ *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 *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 ?y ?x* ∧ *triple.pairs-less-eq ?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* ⊔ *zf* ≤ *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 *x1 x2* **where** (*x1*,*x2*) = *?x*
using *prod.collapse* **by** *blast*
hence *triple.pairs-less-eq triple.pairs-bot ?x*
using *1* **by** (*metis bot.extremum top-greatest top-filter.abs-eq triple.pairs-less-eq.simps*)
}
thus *bot* ≤ *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* = *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 *1* **by** (*metis top.extremum bot-least bot-filter.abs-eq triple.pairs-less-eq.simps*)
}
thus *xf* ≤ *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*: *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** (*x1*,*x2*) = *?x* ∧ (*y1*,*y2*) = *?y* ∧ (*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 *1* **by** (*metis* (*no-types*) *sup-inf-distrib1 inf-sup-distrib1 triple.pairs-sup.simps triple.pairs-inf.simps*)
}
thus *xf* ⊔ (*yf* ⊓ *zf*) = (*xf* ⊔ *yf*) ⊓ (*xf* ⊔ *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*: *triple* (*Rep-phi f*)
   by (*simp add*: *triple-def*)
  let *?x* = *Rep-lifted-pair xf f*
  let *?y* = *Rep-lifted-pair yf f*
  obtain *x1 x2 y1 y2* **where** *2*: (*x1,x2*) = *?x* ∧ (*y1,y2*) = *?y*
   using *prod.collapse* **by** *blast*
  have *?x* ∈ *triple.pairs* (*Rep-phi f*) ∧ *?y* ∈ *triple.pairs* (*Rep-phi f*)
   by *simp*
  hence (*triple.pairs-inf ?x ?y* = *triple.pairs-bot*) ⟷ *triple.pairs-less-eq ?x* (*triple.pairs-uminus* (*Rep-phi f*) *?y*)
   using *1 2* **by** (*metis triple.pairs-uminus-galois*)
 }
 hence ∀*f* . (*Rep-lifted-pair* (*xf* ⊓ *yf*) *f* = *Rep-lifted-pair bot f*) ⟷ *triple.pairs-less-eq* (*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 (*Rep-lifted-pair* (*xf* ⊓ *yf*) = *Rep-lifted-pair bot*) ⟷ *xf* ≤ −*yf*
  using *less-eq-lifted-pair.rep-eq* **by** *auto*
 thus (*xf* ⊓ *yf* = *bot*) ⟷ (*xf* ≤ −*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 *1* **by** (*metis simp-lifted-pair triple.pairs-stone*)
 }
 hence *Rep-lifted-pair* (−*xf* ⊔ −−*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* ⊔ −−*xf* = *top*
  by (*simp add*: *Rep-lifted-pair-inject*)
qed

end

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*
  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** λ*xf 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 $--x$, 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* (λ*f* . *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*:
  sup-inf-top-bot-uminus-ord-homomorphism stone-phi-embed
proof (*intro conjI*)
  let *?p* = λ*f* . *triple.pairs-uminus* (*Rep-phi f*)
  let *?pp* = λ*f x* . *?p f* (*?p f x*)
  let *?q* = λ*f 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-sup* (*?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-sup* (*?q f x*) (*?q f y*) = *triple.pairs-sup* (*?pp f* (*x1*,*x2*)) (*?pp f* (*y1*,*y2*))
        by *simp*
      also have ... = *triple.pairs-sup* (−−*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-sup* (*x1*,*x2*) (*y1*,*y2*))
by *simp*
also have ... = *?q f* (*x* ⊔ *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* ⊔ *y*)
.
qed
have *stone-phi-embed x* ⊔ *stone-phi-embed y* = *Abs-lifted-pair* (λ*f* . *if Rep-phi f* = *stone-phi then Rep-stone-phi-pair x else ?q f x*) ⊔ *Abs-lifted-pair* (λ*f* . *if Rep-phi f* = *stone-phi then Rep-stone-phi-pair y else ?q f y*)
by *simp*
also have ... = *Abs-lifted-pair* (λ*f* . *triple.pairs-sup* (*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 sup-lifted-pair.abs-eq*) (*simp-all add*: *eq-onp-same-args stone-phi-embed-triple-pair*)
also have ... = *Abs-lifted-pair* (λ*f* . *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*))
by (*simp add*: *if-distrib-2*)
also have ... = *Abs-lifted-pair* (λ*f* . *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 *1* by *meson*
also have ... = *Abs-lifted-pair* (λ*f* . *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* = λ*f* . *triple.pairs-uminus* (*Rep-phi f*)
let *?pp* = λ*f x* . *?p f* (*?p f x*)
let *?q* = λ*f 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* (λ*f* . *if Rep-phi f* = *stone-phi then Rep-stone-phi-pair x else ?q f x*) ⊓ *Abs-lifted-pair* (λ*f* . *if Rep-phi f* = *stone-phi then Rep-stone-phi-pair y else ?q f y*)
by *simp*
also have ... = *Abs-lifted-pair* (λ*f* . *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-onp-same-args stone-phi-embed-triple-pair*)
also have ... = *Abs-lifted-pair* (λ*f* . *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* (λ*f* . *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 *1* **by** *meson*
also have ... = *Abs-lifted-pair* (λ*f* . *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* (λ*f* . *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* (λ*f* . *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* (λ*f* . *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* (λ*f* . *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* (λ*f* . (*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* (λ*f* . *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* (λ*f* . *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* (λ*f* . *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* (λ*f* . *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* (λ*f* . (*bot*,*Abs-filter UNIV*))
by (*simp add*: *top-filter.abs-eq*)
also have ... = *bot*
by (*rule bot-lifted-pair.abs-eq*[*THEN sym*])
finally show *stone-phi-embed* (*bot*::'*a stone-phi-pair*) = *bot*
.
next
let *?p* = λ*f* . *triple.pairs-uminus* (*Rep-phi f*)
let *?pp* = λ*f x* . *?p f* (*?p f x*)
let *?q* = λ*f 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* (λ*f* . *if Rep-phi f* = *stone-phi then Rep-stone-phi-pair x else ?q f x*)
by *simp*
also have ... = *Abs-lifted-pair* (λ*f* . *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* (λ*f* . *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* (λ*f* . *if Rep-phi f* = *stone-phi then triple.pairs-uminus* (*Rep-phi f*) (*Rep-stone-phi-pair x*) *else ?q f* (−*x*))
using *1* **by** *meson*
also have ... = *Abs-lifted-pair* (λ*f* . *if Rep-phi f* = *stone-phi then Rep-stone-phi-pair* (−*x*) *else ?q f* (−*x*))
by (*metis uminus-stone-phi-pair.rep-eq*)
also have ... = *stone-phi-embed* (−*x*)
by *simp*
finally show *stone-phi-embed* (−*x*) = −*stone-phi-embed x*
by *simp*
qed
next
let *?p* = λ*f* . *triple.pairs-uminus* (*Rep-phi f*)
let *?pp* = λ*f x* . *?p f* (*?p f x*)
let *?q* = λ*f 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*: $x \le y$
have $\forall f$. *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*)
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* (λ*f* . *if Rep-phi f* = *stone-phi then Rep-stone-phi-pair x else ?q f x*) ≤ *Abs-lifted-pair* (λ*f* . *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 stone-phi-embed-triple-pair*)
thus *stone-phi-embed x* ≤ *stone-phi-embed y*
by *simp*
qed
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 ∀*x*::*'a stone-phi-pair* . *Rep-lifted-pair* (*stone-phi-embed x*) = (λ*f* . *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*)))
    by (*simp add*: *Abs-lifted-pair-inverse stone-phi-embed-triple-pair*)
  hence (λ*f* . *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*))) = (λ*f* . *if Rep-phi f* = *stone-phi then Rep-stone-phi-pair y else triple.pairs-uminus* (*Rep-phi f*) (*triple.pairs-uminus* (*Rep-phi f*) (*Rep-stone-phi-pair y*)))
    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*) (*Rep-stone-phi-pair x*))) (*if Rep-phi f* = *stone-phi then Rep-stone-phi-pair y else triple.pairs-uminus* (*Rep-phi f*) (*triple.pairs-uminus* (*Rep-phi f*) (*Rep-stone-phi-pair y*)))
    using *assms* **by** (*subst less-eq-lifted-pair.abs-eq*[*THEN sym*]) (*simp-all add*: *eq-onp-same-args stone-phi-embed-triple-pair*)
  hence *triple.pairs-less-eq* (*if Rep-phi ?f* = (*stone-phi*::′*a regular* ⇒ ′*a dense-filter*) *then Rep-stone-phi-pair x else triple.pairs-uminus* (*Rep-phi ?f*) (*triple.pairs-uminus* (*Rep-phi ?f*) (*Rep-stone-phi-pair x*))) (*if Rep-phi ?f* = (*stone-phi*::′*a regular* ⇒ ′*a dense-filter*) *then Rep-stone-phi-pair y else triple.pairs-uminus* (*Rep-phi ?f*) (*triple.pairs-uminus* (*Rep-phi ?f*) (*Rep-stone-phi-pair y*)))
    by *simp*
  hence *triple.pairs-less-eq* (*Rep-stone-phi-pair x*) (*Rep-stone-phi-pair y*)
    by (*simp add*: *Abs-phi-inverse stone-phi.hom*)
  thus *x* ≤ *y*
    by (*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* :: (*non-trivial-stone-algebra*) *stone-algebra*
begin

instance

apply *intro-classes*
apply (*simp add*: *less-stone-phi-pair.rep-eq less-eq-stone-phi-pair.rep-eq*)
apply (*simp add*: *stone-phi-embed-order-injective*)
apply (*meson order.trans stone-phi-embed-homomorphism stone-phi-embed-order-injective*)
apply (*meson stone-phi-embed-homomorphism antisym stone-phi-embed-injective injD*)
apply (*metis inf.sup-ge1 stone-phi-embed-homomorphism stone-phi-embed-order-injective*)
apply (*metis inf.sup-ge2 stone-phi-embed-homomorphism stone-phi-embed-order-injective*)
apply (*metis inf-greatest stone-phi-embed-homomorphism stone-phi-embed-order-injective*)
apply (*metis stone-phi-embed-homomorphism stone-phi-embed-order-injective sup-ge1*)
apply (*metis stone-phi-embed-homomorphism stone-phi-embed-order-injective sup.cobounded2*)
apply (*metis stone-phi-embed-homomorphism stone-phi-embed-order-injective sup-least*)
apply (*metis bot.extremum stone-phi-embed-homomorphism stone-phi-embed-order-injective*)
apply (*metis stone-phi-embed-homomorphism stone-phi-embed-order-injective top-greatest*)
apply (*metis* (*mono-tags*, *lifting*) *stone-phi-embed-homomorphism sup-inf-distrib1 stone-phi-embed-injective injD*)
apply (*metis stone-phi-embed-homomorphism stone-phi-embed-injective injD stone-phi-embed-order-injective pseudo-complement*)
by (*metis injD stone-phi-embed-homomorphism stone-phi-embed-injective stone*)

end

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*
where *sa-iso-inv* ≡ λ*p* . *Rep-regular* (*fst* (*Rep-stone-phi-pair p*)) ⊓ *Rep-dense* (*triple.rho-pair stone-phi* (*Rep-stone-phi-pair p*))

abbreviation *sa-iso* :: *'a::non-trivial-stone-algebra* ⇒ *'a stone-phi-pair*
where *sa-iso* ≡ λ*x* . *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* ($-x$)) $\sqcap$ *up-filter* (*Abs-dense* ($y \sqcup -y$)) $\leq$ *up-filter* (*Abs-dense* ($y \sqcup -y \sqcup x$))
proof −
have *Rep-filter* (*stone-phi* (*Abs-regular* ($-x$)) $\sqcap$ *up-filter* (*Abs-dense* ($y \sqcup -y$))) $\leq$ $\uparrow$(*Abs-dense* ($y \sqcup -y \sqcup x$))
proof
fix z :: *'a dense*
let *?r* = *Rep-dense* z
assume $z \in$ *Rep-filter* (*stone-phi* (*Abs-regular* ($-x$)) $\sqcap$ *up-filter* (*Abs-dense* ($y \sqcup -y$)))
also have ... = *Rep-filter* (*stone-phi* (*Abs-regular* ($-x$))) $\cap$ *Rep-filter* (*up-filter* (*Abs-dense* ($y \sqcup -y$)))
by (*simp add*: *inf-filter.rep-eq*)
also have ... = *stone-phi-set* (*Abs-regular* ($-x$)) $\cap$ $\uparrow$(*Abs-dense* ($y \sqcup -y$))
by (*metis Abs-filter-inverse mem-Collect-eq up-filter stone-phi-set-filter stone-phi-def*)
finally have $--x \leq$ *?r* $\wedge$ *Abs-dense* ($y \sqcup -y$) $\leq z$
by (*metis* (*mono-tags*, *lifting*) *Abs-regular-inverse Int-Collect mem-Collect-eq*)
hence $--x \leq$ *?r* $\wedge$ $y \sqcup -y \leq$ *?r*
by (*simp add*: *Abs-dense-inverse less-eq-dense.rep-eq*)
hence $y \sqcup -y \sqcup x \leq$ *?r*
using *order-trans pp-increasing* **by** *auto*
hence *Abs-dense* ($y \sqcup -y \sqcup x$) $\leq$ *Abs-dense ?r*
by (*subst less-eq-dense.abs-eq*) (*simp-all add*: *eq-onp-same-args*)
thus $z \in$ $\uparrow$(*Abs-dense* ($y \sqcup -y \sqcup x$))
by (*simp add*: *Rep-dense-inverse*)
qed
hence *Abs-filter* (*Rep-filter* (*stone-phi* (*Abs-regular* ($-x$)) $\sqcap$ *up-filter* (*Abs-dense* ($y \sqcup -y$)))) $\leq$ *up-filter* (*Abs-dense* ($y \sqcup -y \sqcup x$))
by (*simp add*: *eq-onp-same-args less-eq-filter.abs-eq*)
thus *?thesis*
by (*simp add*: *Rep-filter-inverse*)
qed

lemma *stone-phi-complement*:
complement (*stone-phi* (*Abs-regular* ($-x$))) (*stone-phi* (*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*:
up-filter (*Abs-dense* ($x \sqcup -x$)) $\leq$ *stone-phi* (*Abs-regular* ($--x$))
proof −
have $\uparrow$(*Abs-dense* ($x \sqcup -x$)) $\leq$ *stone-phi-set* (*Abs-regular* ($--x$))
proof
fix z :: *'a dense*
let *?r* = *Rep-dense* z

assume $z \in \uparrow(Abs\text{-}dense\ (x \sqcup -x))$
hence $---x \leq ?r$
by (*simp add*: *Abs-dense-inverse less-eq-dense.rep-eq*)
hence $-Rep\text{-}regular\ (Abs\text{-}regular\ (--x)) \leq ?r$
by (*metis* (*mono-tags*, *lifting*) *Abs-regular-inverse mem-Collect-eq*)
thus $z \in stone\text{-}phi\text{-}set\ (Abs\text{-}regular\ (--x))$
by *simp*
qed
thus *?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*:
$sa\text{-}iso\text{-}inv\ (sa\text{-}iso\ x) = x$
proof –
have $up\text{-}filter\ (triple.rho\text{-}pair\ stone\text{-}phi\ (Abs\text{-}regular\ (--x),stone\text{-}phi\ (Abs\text{-}regular\ (-x)) \sqcup up\text{-}filter\ (Abs\text{-}dense\ (x \sqcup -x)))) = stone\text{-}phi\ (Abs\text{-}regular\ (--x)) \sqcap (stone\text{-}phi\ (Abs\text{-}regular\ (-x)) \sqcup up\text{-}filter\ (Abs\text{-}dense\ (x \sqcup -x)))$
using *sa-iso-triple-pair stone-phi.get-rho-pair-char* **by** *blast*
also have ... $= stone\text{-}phi\ (Abs\text{-}regular\ (--x)) \sqcap up\text{-}filter\ (Abs\text{-}dense\ (x \sqcup -x))$
by (*simp add*: *inf.sup-commute inf-sup-distrib1 stone-phi-complement*)
also have ... $= up\text{-}filter\ (Abs\text{-}dense\ (x \sqcup -x))$
using *up-dense-stone-phi inf.absorb2* **by** *auto*
finally have *1*: $triple.rho\text{-}pair\ stone\text{-}phi\ (Abs\text{-}regular\ (--x),stone\text{-}phi\ (Abs\text{-}regular\ (-x)) \sqcup up\text{-}filter\ (Abs\text{-}dense\ (x \sqcup -x))) = Abs\text{-}dense\ (x \sqcup -x)$
using *up-filter-injective* **by** *auto*
have $sa\text{-}iso\text{-}inv\ (sa\text{-}iso\ x) = (\lambda p\ .\ Rep\text{-}regular\ (fst\ p) \sqcap Rep\text{-}dense\ (triple.rho\text{-}pair\ stone\text{-}phi\ p))\ (Abs\text{-}regular\ (--x),stone\text{-}phi\ (Abs\text{-}regular\ (-x)) \sqcup up\text{-}filter\ (Abs\text{-}dense\ (x \sqcup -x)))$
by (*simp add*: *Abs-stone-phi-pair-inverse sa-iso-triple-pair*)
also have ... $= Rep\text{-}regular\ (Abs\text{-}regular\ (--x)) \sqcap Rep\text{-}dense\ (triple.rho\text{-}pair\ stone\text{-}phi\ (Abs\text{-}regular\ (--x),stone\text{-}phi\ (Abs\text{-}regular\ (-x)) \sqcup up\text{-}filter\ (Abs\text{-}dense\ (x \sqcup -x))))$
by *simp*
also have ... $= --x \sqcap Rep\text{-}dense\ (Abs\text{-}dense\ (x \sqcup -x))$
using *1* **by** (*subst Abs-regular-inverse*) *auto*
also have ... $= --x \sqcap (x \sqcup -x)$
by (*subst Abs-dense-inverse*) *simp-all*
also have ... $= x$
by *simp*
finally show *?thesis*
by *auto*
qed

lemma *sa-iso-right-invertible*:
$sa\text{-}iso\ (sa\text{-}iso\text{-}inv\ p) = p$

```
proof −
  obtain x y where 1: (x,y) = Rep-stone-phi-pair p
    using prod.collapse by blast
  hence 2: (x,y) ∈ triple.pairs stone-phi
    by (simp add: Rep-stone-phi-pair)
  hence 3: stone-phi (−x) ≤ y
    by (simp add: stone-phi.pairs-phi-less-eq)
  have 4: ∀ z . z ∈ Rep-filter (stone-phi x ⊓ y) ⟶ −Rep-regular x ≤ Rep-dense z
  proof (rule allI, rule impI)
    fix z :: 'a dense
    let ?r = Rep-dense z
    assume z ∈ Rep-filter (stone-phi x ⊓ y)
    hence z ∈ Rep-filter (stone-phi x)
      by (simp add: inf-filter.rep-eq)
    also have ... = stone-phi-set x
      by (simp add: stone-phi-def Abs-filter-inverse stone-phi-set-filter)
    finally show −Rep-regular x ≤ ?r
      by simp
  qed
  have triple.rho-pair stone-phi (x,y) ∈ ↑(triple.rho-pair stone-phi (x,y))
    by simp
  also have ... = Rep-filter (Abs-filter (↑(triple.rho-pair stone-phi (x,y))))
    by (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-regular x ⊓ Rep-dense (triple.rho-pair stone-phi (x,y))) ⊔ −Rep-regular
x)))
    by (metis (mono-tags, lifting) Rep-regular-inverse double-compl
uminus-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) maddux-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*: *Rep-dense-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-iff-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* $(x \sqcup -x \sqcup y)) \leq$ *Rep-filter* (*stone-phi* (*Abs-regular* $(-y))$
$\sqcup$ *up-filter ?v*)
proof
fix z
assume $z \in$ ↑(*Abs-dense* $(x \sqcup -x \sqcup y))$
hence *Abs-dense* $(x \sqcup -x \sqcup y) \leq z$
by *simp*
hence *3*: $x \sqcup -x \sqcup y \leq$ *Rep-dense* z
by (*simp add*: *Abs-dense-inverse less-eq-dense.rep-eq*)
have $y \leq x \sqcup -x \sqcup --y$
by (*simp add*: *le-supI2 pp-increasing*)
hence $(x \sqcup -x \sqcup --y) \sqcap (y \sqcup -y) = y \sqcup ((x \sqcup -x \sqcup --y) \sqcap -y)$
by (*simp add*: *le-iff-sup sup-inf-distrib1*)
also have ... $= y \sqcup ((x \sqcup -x) \sqcap -y)$
by (*simp add*: *inf-commute inf-sup-distrib1*)
also have ... $\leq$ *Rep-dense* z
using *3* **by** (*meson le-infI1 sup.bounded-iff*)
finally have *Abs-dense* $((x \sqcup -x \sqcup --y) \sqcap (y \sqcup -y)) \leq z$
by (*simp add*: *Abs-dense-inverse less-eq-dense.rep-eq*)
hence *4*: *?u* $\sqcap$ *?v* $\leq z$
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 ... $\leq$ *Rep-dense ?u*
by (*simp add*: *Abs-dense-inverse*)
finally have *?u* $\in$ *stone-phi-set* (*Abs-regular* $(-y))$
by *simp*
hence *5*: *?u* $\in$ *Rep-filter* (*stone-phi* (*Abs-regular* $(-y)))$
by (*metis mem-Collect-eq stone-phi-def stone-phi-set-filter*
Abs-filter-inverse)
have *?v* $\in$ ↑*?v*
by *simp*
hence *?v* $\in$ *Rep-filter* (*up-filter ?v*)
by (*metis Abs-filter-inverse mem-Collect-eq up-filter*)
thus $z \in$ *Rep-filter* (*stone-phi* (*Abs-regular* $(-y)) \sqcup$ *up-filter ?v*)
using *4 5 sup-filter.rep-eq* **by** *blast*
qed
hence *up-filter* (*Abs-dense* $(x \sqcup -x \sqcup y)) \leq$ *Abs-filter* (*Rep-filter* (*stone-phi*
(*Abs-regular* $(-y)) \sqcup$ *up-filter ?v*))
by (*simp add*: *eq-onp-same-args less-eq-filter.abs-eq*)
also have ... = *stone-phi* (*Abs-regular* $(-y)) \sqcup$ *up-filter ?v*
by (*simp add*: *Rep-filter-inverse*)
finally show *up-filter* (*Abs-dense* $(x \sqcup -x \sqcup y)) \leq$ (*stone-phi* (*Abs-regular*
$(-x)) \sqcup$ *up-filter* (*Abs-dense* $(x \sqcup -x))) \sqcap$ (*stone-phi* (*Abs-regular* $(-y)) \sqcup$
up-filter (*Abs-dense* $(y \sqcup -y)))$
by (*metis le-infI le-supI2 sup-bot.right-neutral up-filter-dense-antitone*)
qed
have *6*: $\forall x::'a$. *in-p-image* $(-x)$
by *auto*

show $\forall x\ y{::}'a$. *sa-iso* $(x \sqcup y) =$ *sa-iso* $x \sqcup$ *sa-iso* y
proof (*intro allI*)
fix $x\ y :: 'a$
have *7*: *up-filter* (*Abs-dense* $(x \sqcup -x)$) $\sqcap$ *up-filter* (*Abs-dense* $(y \sqcup -y)$) $\leq$ *up-filter* (*Abs-dense* $(y \sqcup -y \sqcup x)$)
proof −
have *up-filter* (*Abs-dense* $(x \sqcup -x)$) $\sqcap$ *up-filter* (*Abs-dense* $(y \sqcup -y)$) $=$ *up-filter* (*Abs-dense* $(x \sqcup -x)$ $\sqcup$ *Abs-dense* $(y \sqcup -y)$)
by (*metis up-filter-dist-sup*)
also have ... $=$ *up-filter* (*Abs-dense* $(x \sqcup -x \sqcup (y \sqcup -y))$)
by (*subst sup-dense.abs-eq*) (*simp-all add*: *eq-onp-same-args*)
also have ... $=$ *up-filter* (*Abs-dense* $(y \sqcup -y \sqcup x \sqcup -x)$)
by (*simp add*: *sup-commute sup-left-commute*)
also have ... $\leq$ *up-filter* (*Abs-dense* $(y \sqcup -y \sqcup x)$)
using *up-filter-dense-antitone* **by** *auto*
finally show *?thesis*
.
qed
have *Abs-dense* $(x \sqcup y \sqcup -(x \sqcup y)) =$ *Abs-dense* $((x \sqcup -x \sqcup y) \sqcap (y \sqcup -y \sqcup x))$
by (*simp add*: *sup-commute sup-inf-distrib1 sup-left-commute*)
also have ... $=$ *Abs-dense* $(x \sqcup -x \sqcup y)$ $\sqcap$ *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 *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 ... $\leq$ (*stone-phi* (*Abs-regular* $(-x)$) $\sqcup$ *up-filter* (*Abs-dense* $(x \sqcup -x)$)) $\sqcap$ (*stone-phi* (*Abs-regular* $(-y)$) $\sqcup$ *up-filter* (*Abs-dense* $(y \sqcup -y)$))
using *2* **by** (*simp add*: *inf.sup-commute le-sup-iff*)
finally have *9*: (*stone-phi* (*Abs-regular* $(-x)$) $\sqcap$ *stone-phi* (*Abs-regular* $(-y)$)) $\sqcup$ *up-filter* (*Abs-dense* $(x \sqcup y \sqcup -(x \sqcup y))$) $\leq$ ...
by (*simp add*: *le-supI1*)
have ... $=$ (*stone-phi* (*Abs-regular* $(-x)$) $\sqcap$ *stone-phi* (*Abs-regular* $(-y)$)) $\sqcup$ (*stone-phi* (*Abs-regular* $(-x)$) $\sqcap$ *up-filter* (*Abs-dense* $(y \sqcup -y)$)) $\sqcup$ ((*up-filter* (*Abs-dense* $(x \sqcup -x)$) $\sqcap$ *stone-phi* (*Abs-regular* $(-y)$)) $\sqcup$ (*up-filter* (*Abs-dense* $(x \sqcup -x)$) $\sqcap$ *up-filter* (*Abs-dense* $(y \sqcup -y)$)))
by (*metis* (*no-types*) *inf-sup-distrib1 inf-sup-distrib2*)
also have ... $\leq$ (*stone-phi* (*Abs-regular* $(-x)$) $\sqcap$ *stone-phi* (*Abs-regular* $(-y)$)) $\sqcup$ *up-filter* (*Abs-dense* $(y \sqcup -y \sqcup x)$) $\sqcup$ ((*up-filter* (*Abs-dense* $(x \sqcup -x)$) $\sqcap$ *stone-phi* (*Abs-regular* $(-y)$)) $\sqcup$ (*up-filter* (*Abs-dense* $(x \sqcup -x)$) $\sqcap$ *up-filter* (*Abs-dense* $(y \sqcup -y)$)))
by (*meson sup-left-isotone sup-right-isotone stone-phi-inf-dense*)
also have ... $\leq$ (*stone-phi* (*Abs-regular* $(-x)$) $\sqcap$ *stone-phi* (*Abs-regular* $(-y)$)) $\sqcup$ *up-filter* (*Abs-dense* $(y \sqcup -y \sqcup x)$) $\sqcup$ (*up-filter* (*Abs-dense* $(x \sqcup -x \sqcup y)$) $\sqcup$ (*up-filter* (*Abs-dense* $(x \sqcup -x)$) $\sqcap$ *up-filter* (*Abs-dense* $(y \sqcup -y)$)))
by (*metis inf.commute sup-left-isotone sup-right-isotone stone-phi-inf-dense*)
also have ... $\leq$ (*stone-phi* (*Abs-regular* $(-x)$) $\sqcap$ *stone-phi* (*Abs-regular* $(-y)$)) $\sqcup$ *up-filter* (*Abs-dense* $(y \sqcup -y \sqcup x)$) $\sqcup$ *up-filter* (*Abs-dense* $(x \sqcup -x \sqcup y)$)

using *7* **by** (*simp add*: *sup.absorb1 sup-commute sup-left-commute*)
also have ... = (*stone-phi* (*Abs-regular* (−*x*)) ⊓ *stone-phi* (*Abs-regular* (−*y*))) ⊔ *up-filter* (*Abs-dense* (*x* ⊔ *y* ⊔ −(*x* ⊔ *y*)))
using *8* **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*))) = ...
using *9* **using** *antisym* **by** *blast*
also have ... = *stone-phi* (*Abs-regular* (−*x*) ⊓ *Abs-regular* (−*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 *6* **by** (*subst inf-regular.abs-eq*) (*simp-all add*: *eq-onp-same-args*)
finally have *10*: *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*) ⊔ *Abs-regular* (−−*y*)
using *6* **by** (*subst sup-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* (*triple.pairs-sup* (*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 *10* **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 sup-stone-phi-pair.abs-eq*[*THEN sym*]) (*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
have *1*: ∀ *x y*::′*a* . *dense* (*x* ⊔ −*x* ⊔ *y*)
by *simp*
have *2*: ∀ *x*::′*a* . *in-p-image* (−*x*)
by *auto*
have *3*: ∀ *x y*::′*a* . *stone-phi* (*Abs-regular* (−*y*)) ⊔ *up-filter* (*Abs-dense* (*x* ⊔ −*x*)) = *stone-phi* (*Abs-regular* (−*y*)) ⊔ *up-filter* (*Abs-dense* (*x* ⊔ −*x* ⊔ −*y*))
proof (*intro allI*)
fix *x y* :: ′*a*
have *4*: *up-filter* (*Abs-dense* (*x* ⊔ −*x*)) ≤ *stone-phi* (*Abs-regular* (−*y*)) ⊔ *up-filter* (*Abs-dense* (*x* ⊔ −*x* ⊔ −*y*))
by (*metis* (*no-types*, *lifting*) *complement-shunting stone-phi-inf-dense stone-phi-complement complement-symmetric*)
have *up-filter* (*Abs-dense* (*x* ⊔ −*x* ⊔ −*y*)) ≤ *up-filter* (*Abs-dense* (*x* ⊔ −*x*))
by (*metis sup-idem up-filter-dense-antitone*)
thus *stone-phi* (*Abs-regular* (−*y*)) ⊔ *up-filter* (*Abs-dense* (*x* ⊔ −*x*)) =

stone-phi (*Abs-regular* (−*y*)) ⊔ *up-filter* (*Abs-dense* (*x* ⊔ −*x* ⊔ −*y*))
    using *4* **by** (*simp add*: *le-iff-sup sup-commute sup-left-commute*)
  qed
  show ∀ *x y*::′*a* . *sa-iso* (*x* ⊓ *y*) = *sa-iso x* ⊓ *sa-iso y*
  proof (*intro allI*)
    fix *x y* :: ′*a*
    have *Abs-dense* ((*x* ⊓ *y*) ⊔ −(*x* ⊓ *y*)) = *Abs-dense* ((*x* ⊔ −*x* ⊔ −*y*) ⊓ (*y* ⊔ −*y* ⊔ −*x*))
      by (*simp add*: *sup-commute sup-inf-distrib1 sup-left-commute*)
    also have ... = *Abs-dense* (*x* ⊔ −*x* ⊔ −*y*) ⊓ *Abs-dense* (*y* ⊔ −*y* ⊔ −*x*)
      using *1* **by** (*metis* (*mono-tags*, *lifting*) *Abs-dense-inverse Rep-dense-inverse inf-dense.rep-eq mem-Collect-eq*)
    finally have *5*: *up-filter* (*Abs-dense* ((*x* ⊓ *y*) ⊔ −(*x* ⊓ *y*))) = *up-filter* (*Abs-dense* (*x* ⊔ −*x* ⊔ −*y*)) ⊔ *up-filter* (*Abs-dense* (*y* ⊔ −*y* ⊔ −*x*))
      by (*simp add*: *up-filter-dist-inf*)
    have (*stone-phi* (*Abs-regular* (−*x*)) ⊔ *up-filter* (*Abs-dense* (*x* ⊔ −*x*))) ⊔ (*stone-phi* (*Abs-regular* (−*y*)) ⊔ *up-filter* (*Abs-dense* (*y* ⊔ −*y*))) = (*stone-phi* (*Abs-regular* (−*y*)) ⊔ *up-filter* (*Abs-dense* (*x* ⊔ −*x*))) ⊔ (*stone-phi* (*Abs-regular* (−*x*)) ⊔ *up-filter* (*Abs-dense* (*y* ⊔ −*y*)))
      by (*simp add*: *inf-sup-aci*(*6*) *sup-left-commute*)
    also have ... = (*stone-phi* (*Abs-regular* (−*y*)) ⊔ *up-filter* (*Abs-dense* (*x* ⊔ −*x* ⊔ −*y*))) ⊔ (*stone-phi* (*Abs-regular* (−*x*)) ⊔ *up-filter* (*Abs-dense* (*y* ⊔ −*y* ⊔ −*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*) ⊔ *Abs-regular* (−*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*) ⊓ *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* (*triple.pairs-inf* (*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 sym*]) (*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 sym*], *unfold eq-onp-same-args*) *auto*
also have ... = −*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 isomorphism 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* ⇒ ′*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* ⇒ ('*a*,'*b*) *lifted-boolean-algebra* ⇒ ('*a*,'*b*) *lifted-boolean-algebra* **is** λ*xf yf f* . *sup* (*xf f*) (*yf f*) **.**

lift-definition *inf-lifted-boolean-algebra* :: ('*a*,'*b*) *lifted-boolean-algebra* ⇒ ('*a*,'*b*) *lifted-boolean-algebra* ⇒ ('*a*,'*b*) *lifted-boolean-algebra* **is** λ*xf yf f* . *inf* (*xf f*) (*yf f*) **.**

lift-definition *minus-lifted-boolean-algebra* :: ('*a*,'*b*) *lifted-boolean-algebra* ⇒ ('*a*,'*b*) *lifted-boolean-algebra* ⇒ ('*a*,'*b*) *lifted-boolean-algebra* **is** λ*xf yf f* . *minus* (*xf f*) (*yf f*) **.**

lift-definition *uminus-lifted-boolean-algebra* :: ('*a*,'*b*) *lifted-boolean-algebra* ⇒ ('*a*,'*b*) *lifted-boolean-algebra* **is** λ*xf f* . *uminus* (*xf f*) **.**

lift-definition *bot-lifted-boolean-algebra* :: ('*a*,'*b*) *lifted-boolean-algebra* **is** λ*f* . *bot* **..**

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

lift-definition *less-eq-lifted-boolean-algebra* :: ('*a*,'*b*) *lifted-boolean-algebra* ⇒ ('*a*,'*b*) *lifted-boolean-algebra* ⇒ *bool* **is** λ*xf 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** λ*xf 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*)
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-eq-lifted-boolean-algebra.rep-eq*)
apply (*simp add*: *inf-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*)
apply (*simp add*: *less-eq-lifted-boolean-algebra.rep-eq sup-lifted-boolean-algebra.rep-eq*)
apply (*simp add*: *less-eq-lifted-boolean-algebra.rep-eq sup-lifted-boolean-algebra.rep-eq*)
apply (*simp add*: *bot-lifted-boolean-algebra.rep-eq less-eq-lifted-boolean-algebra.rep-eq*)
apply (*simp add*: *less-eq-lifted-boolean-algebra.rep-eq top-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*)
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*)

end

The following two definitions give the Boolean algebra isomorphism.

abbreviation *ba-iso-inv* :: (*'a*::*non-trivial-boolean-algebra*,*'b*::*distrib-lattice-top*) *lifted-boolean-algebra* ⇒ (*'a*,*'b*) *lifted-pair regular*
where *ba-iso-inv* ≡ λ*xf* . *Abs-regular* (*Abs-lifted-pair* (λ*f* . (*Rep-lifted-boolean-algebra xf f*,*Rep-phi f* (−*Rep-lifted-boolean-algebra xf f*))))

abbreviation *ba-iso* :: (*'a*::*non-trivial-boolean-algebra*,*'b*::*distrib-lattice-top*) *lifted-pair regular* ⇒ (*'a*,*'b*) *lifted-boolean-algebra*
where *ba-iso* ≡ λ*pf* . *Abs-lifted-boolean-algebra* (λ*f* . *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*))

lemma *ba-iso-inv-lifted-pair*:
(*Rep-lifted-boolean-algebra xf f*,*Rep-phi f* (−*Rep-lifted-boolean-algebra xf f*)) ∈ *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* (λ*f* . (*Rep-lifted-boolean-algebra xf f*,*Rep-phi f* (−*Rep-lifted-boolean-algebra xf f*))))
proof −
have ∀*f* . (*Rep-lifted-boolean-algebra xf f*,*Rep-phi f* (−*Rep-lifted-boolean-algebra xf f*)) = *triple.pairs-uminus* (*Rep-phi f*) (*triple.pairs-uminus* (*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* (*λf* . (*Rep-lifted-boolean-algebra xf f*,*Rep-phi f* (−*Rep-lifted-boolean-algebra xf f*))) = −−*Abs-lifted-pair* (*λf* . (*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 *1*: ∀*f* . *snd* (*Rep-lifted-pair* (*Rep-regular pf*) *f*) = *Rep-phi f* (−*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*))
proof
fix *f* :: (*'a*,*'b*) *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 uminus-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 Rep-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* (λ*f* . (*?rf f*,*Rep-phi f* (−*?rf f*))))
proof −
have −−*Abs-lifted-pair* (λ*f* . (*?rf f*,*Rep-phi f* (−*?rf f*))) = −*Abs-lifted-pair* (λ*f* . *triple.pairs-uminus* (*Rep-phi f*) (*?rf f*,*Rep-phi f* (−*?rf f*)))
using *1* **by** (*simp add*: *eq-onp-same-args uminus-lifted-pair.abs-eq*)
also have ... = −*Abs-lifted-pair* (λ*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* (λ*f* . *triple.pairs-uminus* (*Rep-phi f*) (−*?rf f*,*Rep-phi f* (*?rf f*)))
using *1* **by** (*simp add*: *eq-onp-same-args uminus-lifted-pair.abs-eq*)
also have ... = *Abs-lifted-pair* (λ*f* . (*?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* (λ*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 algebra homomorphism properties.

lemma *ba-iso*:
boolean-algebra-isomorphism ba-iso
proof (*intro conjI*)
show *Abs-lifted-boolean-algebra* (λ*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* (λ*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 ∀ *pf qf* . *Abs-lifted-boolean-algebra* (λ*f*::(′*a*,′*b*) *phi* . *fst* (*Rep-lifted-pair*

(*Rep-regular* (*pf* ⊔ *qf*)) *f*)) = *Abs-lifted-boolean-algebra* (*λf* . *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)) ⊔ *Abs-lifted-boolean-algebra* (*λf* . *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* ∧ (*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*) ⊔ *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*) ⊔ *fst* (*Rep-lifted-pair* (*Rep-regular qf*) *f*)
        using *1* **by** *simp*
      hence *fst* (*Rep-lifted-pair* (*Rep-regular* (*pf* ⊔ *qf*)) *f*) = *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*) ⊔ *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* (*λf* . *fst* (*Rep-lifted-pair* (*Rep-regular* (*pf* ⊔ *qf*)) *f*)) = *Abs-lifted-boolean-algebra* (*λf* . *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)) ⊔ *Abs-lifted-boolean-algebra* (*λf* . *fst* (*Rep-lifted-pair* (*Rep-regular qf*) *f*))
      by (*simp add*: *eq-onp-same-args sup-lifted-boolean-algebra.abs-eq sup-regular.rep-eq sup-lifted-boolean-algebra.rep-eq*)
  qed
next
  show ∀ *pf qf* . *Abs-lifted-boolean-algebra* (*λf*::(*'a*,*'b*) *phi* . *fst* (*Rep-lifted-pair* (*Rep-regular* (*pf* ⊓ *qf*)) *f*)) = *Abs-lifted-boolean-algebra* (*λf* . *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)) ⊓ *Abs-lifted-boolean-algebra* (*λf* . *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* ∧ (*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-inf* (*x*,*y*) (*z*,*w*)) = *fst* (*x*,*y*) ⊓ *fst* (*z*,*w*)
        using *triple.pairs-inf.simps* **by** *force*
      hence *fst* (*triple.pairs-inf* (*Rep-lifted-pair* (*Rep-regular pf*) *f*) (*Rep-lifted-pair* (*Rep-regular qf*) *f*)) = *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*) ⊓ *fst* (*Rep-lifted-pair* (*Rep-regular qf*) *f*)
        using *1* **by** *simp*
      hence *fst* (*Rep-lifted-pair* (*Rep-regular* (*pf* ⊓ *qf*)) *f*) = *fst* (*Rep-lifted-pair*

(*Rep-regular pf*) *f*) ⊓ *fst* (*Rep-lifted-pair* (*Rep-regular qf*) *f*)
by (*unfold inf-regular.rep-eq inf-lifted-pair.rep-eq*) *simp*
}
thus *Abs-lifted-boolean-algebra* (λ*f* . *fst* (*Rep-lifted-pair* (*Rep-regular* (*pf* ⊓ *qf*)) *f*)) = *Abs-lifted-boolean-algebra* (λ*f* . *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)) ⊓ *Abs-lifted-boolean-algebra* (λ*f* . *fst* (*Rep-lifted-pair* (*Rep-regular qf*) *f*))
by (*simp add*: *eq-onp-same-args inf-lifted-boolean-algebra.abs-eq inf-regular.rep-eq inf-lifted-boolean-algebra.rep-eq*)
qed
next
show ∀ *pf* . *Abs-lifted-boolean-algebra* (λ*f*::('*a*,'*b*) *phi* . *fst* (*Rep-lifted-pair* (*Rep-regular* (−*pf*)) *f*)) = −*Abs-lifted-boolean-algebra* (λ*f* . *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*))
proof
fix *pf* :: ('*a*,'*b*) *lifted-pair regular*
{
fix *f*
obtain *x y* **where** *1*: (*x*,*y*) = *Rep-lifted-pair* (*Rep-regular pf*) *f*
using *prod.collapse* **by** *blast*
have *triple* (*Rep-phi f*)
by (*simp add*: *triple-def*)
hence *fst* (*triple.pairs-uminus* (*Rep-phi f*) (*x*,*y*)) = −*fst* (*x*,*y*)
using *triple.pairs-uminus.simps* **by** *force*
hence *fst* (*triple.pairs-uminus* (*Rep-phi f*) (*Rep-lifted-pair* (*Rep-regular pf*) *f*)) = −*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)
using *1* **by** *simp*
hence *fst* (*Rep-lifted-pair* (*Rep-regular* (−*pf*)) *f*) = −*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)
by (*unfold uminus-regular.rep-eq uminus-lifted-pair.rep-eq*) *simp*
}
thus *Abs-lifted-boolean-algebra* (λ*f* . *fst* (*Rep-lifted-pair* (*Rep-regular* (−*pf*)) *f*)) = −*Abs-lifted-boolean-algebra* (λ*f* . *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*))
by (*simp add*: *eq-onp-same-args uminus-lifted-boolean-algebra.abs-eq uminus-regular.rep-eq uminus-lifted-boolean-algebra.rep-eq*)
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** λ*xf yf f* . *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** λ*xf yf f* . *inf* (*xf f*) (*yf f*) **.**

lift-definition *top-lifted-distrib-lattice-top* :: ('*a*,'*b*) *lifted-distrib-lattice-top* **is** λ*f* . *top* **..**

lift-definition *less-eq-lifted-distrib-lattice-top* :: ('*a*,'*b*) *lifted-distrib-lattice-top* ⇒ ('*a*,'*b*) *lifted-distrib-lattice-top* ⇒ *bool* **is** λ*xf yf* . ∀*f* . *less-eq* (*xf f*) (*yf f*) **.**

lift-definition *less-lifted-distrib-lattice-top* :: ('*a*,'*b*) *lifted-distrib-lattice-top* ⇒ ('*a*,'*b*) *lifted-distrib-lattice-top* ⇒ *bool* **is** λ*xf 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-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*: *inf-lifted-distrib-lattice-top.rep-eq* *less-eq-lifted-distrib-lattice-top.rep-eq*)
  apply (*simp add*: *inf-lifted-distrib-lattice-top.rep-eq* *less-eq-lifted-distrib-lattice-top.rep-eq*)
  apply (*simp add*: *inf-lifted-distrib-lattice-top.rep-eq* *less-eq-lifted-distrib-lattice-top.rep-eq*)
  apply (*simp add*: *sup-lifted-distrib-lattice-top.rep-eq* *less-eq-lifted-distrib-lattice-top.rep-eq*)
  apply (*simp add*: *less-eq-lifted-distrib-lattice-top.rep-eq* *sup-lifted-distrib-lattice-top.rep-eq*)
  apply (*simp add*: *less-eq-lifted-distrib-lattice-top.rep-eq* *sup-lifted-distrib-lattice-top.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* :: ('*a*::*non-trivial-boolean-algebra*,'*b*::*distrib-lattice-top*) *lifted-pair dense* ⇒ ('*a*,'*b*) *phi* ⇒ '*b*
  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* ∧ (*x*,*y*) ∈ *triple.pairs* (*Rep-phi f*) ∧ *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*)
    by (*simp add*: *triple-def*)
  hence ∃*z*. *y* = *Rep-phi f* (−*x*) ⊔ *up-filter z*
    using *1 triple.pairs-def* **by** *blast*
  then obtain *z* **where** *y* = *up-filter z*
    using *2* **by** *auto*
  hence *Rep-lifted-pair* (*Rep-dense pf*) *f* = (*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* ⇒ ('*a*,'*b*) *lifted-pair dense*
  where *dl-iso-inv* ≡ λ*xf* . *Abs-dense* (*Abs-lifted-pair* (λ*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* ⇒ ('*a*,'*b*) *lifted-distrib-lattice-top*
  where *dl-iso* ≡ λ*pf* . *Abs-lifted-distrib-lattice-top* (*get-dense pf*)

lemma *dl-iso-inv-lifted-pair*:
  (*top*,*up-filter* (*Rep-lifted-distrib-lattice-top xf f*)) ∈ *triple.pairs* (*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*:

dense (*Abs-lifted-pair* (*λf* . (*top*,*up-filter* (*Rep-lifted-distrib-lattice-top xf f*))))
proof −
have ∀*f* . *triple.pairs-uminus* (*Rep-phi f*) (*top*,*up-filter* (*Rep-lifted-distrib-lattice-top xf f*)) = *triple.pairs-bot*
by (*simp add*: *top-filter.abs-eq triple.pairs-uminus.simps triple-def*)
hence *bot* = −*Abs-lifted-pair* (*λf* . (*top*,*up-filter* (*Rep-lifted-distrib-lattice-top xf f*)))
by (*simp add*: *eq-onp-def uminus-lifted-pair.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*:
dl-iso-inv (*dl-iso pf*) = *pf*
proof −
have *dl-iso-inv* (*dl-iso pf*) = *Abs-dense* (*Abs-lifted-pair* (*λf* . (*top*,*up-filter* (*get-dense pf f*))))
by (*metis Abs-lifted-distrib-lattice-top-inverse UNIV-I UNIV-def*)
also have ... = *Abs-dense* (*Abs-lifted-pair* (*Rep-lifted-pair* (*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* (*λf* . (*top*,*up-filter* (*?rf f*))))
have *1*: ∀*f* . (*top*,*up-filter* (*?rf f*)) ∈ *triple.pairs* (*Rep-phi f*)
proof
fix *f* :: (*'a*,*'b*) *phi*
have *triple* (*Rep-phi f*)
by (*simp add*: *triple-def*)
thus (*top*,*up-filter* (*?rf f*)) ∈ *triple.pairs* (*Rep-phi f*)
using *triple.pairs-def* **by** *force*
qed
have *2*: *dense* (*Abs-lifted-pair* (*λf* . (*top*,*up-filter* (*?rf f*))))
proof −
have −*Abs-lifted-pair* (*λf* . (*top*,*up-filter* (*?rf f*))) = *Abs-lifted-pair* (*λf* . *triple.pairs-uminus* (*Rep-phi f*) (*top*,*up-filter* (*?rf f*)))
using *1* **by** (*simp add*: *eq-onp-same-args uminus-lifted-pair.abs-eq*)
also have ... = *Abs-lifted-pair* (*λf* . (*bot*,*Rep-phi f top*))
by (*simp add*: *triple.pairs-uminus.simps triple-def*)

also have ... = *Abs-lifted-pair* (λ*f* . *triple.pairs-bot*)
  by (*metis* (*no-types*, *hide-lams*) *simp-phi triple.phi-top triple-def*)
also have ... = *bot*
  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 ... = *Rep-lifted-pair* (*Abs-lifted-pair* (λ*f* . (*top*,*up-filter* (*?rf f*)))) *f*
    using *Abs-dense-inverse 2* **by** *force*
  also have ... = (*top*,*up-filter* (*?rf f*))
    using *1* **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

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 *get-dense top* = (λ*f*::(′*a*,′*b*) *phi* . *top*)
  proof
    fix *f* :: (′*a*,′*b*) *phi*
    have *Rep-lifted-pair* (*Rep-dense top*) *f* = (*top*,*Abs-filter* {*top*})
      by (*simp add*: *top-dense.rep-eq top-lifted-pair.rep-eq*)
    hence *up-filter* (*get-dense top f*) = *Abs-filter* {*top*}
      by (*metis prod.inject get-dense-char*)
    hence *Rep-filter* (*up-filter* (*get-dense top f*)) = {*top*}
      by (*metis bot-filter.abs-eq bot-filter.rep-eq*)
    thus *get-dense top f* = *top*
      by (*metis self-in-upset singletonD Abs-filter-inverse mem-Collect-eq up-filter*)
  qed
  thus *Abs-lifted-distrib-lattice-top* (*get-dense top*::(′*a*,′*b*) *phi* ⇒ ′*b*) = *top*
    by (*metis top-lifted-distrib-lattice-top-def*)
next
  show ∀ *pf qf* :: (′*a*,′*b*) *lifted-pair dense* . *Abs-lifted-distrib-lattice-top* (*get-dense* (*pf* ⊔ *qf*)) = *Abs-lifted-distrib-lattice-top* (*get-dense pf*) ⊔ *Abs-lifted-distrib-lattice-top* (*get-dense qf*)
  proof (*intro allI*)
    fix *pf qf* :: (′*a*,′*b*) *lifted-pair dense*
    have *1*: *Abs-lifted-distrib-lattice-top* (*get-dense pf*) ⊔

Abs-lifted-distrib-lattice-top (*get-dense qf*) = *Abs-lifted-distrib-lattice-top* (λ*f* . *get-dense pf f* ⊔ *get-dense qf f*)
      by (*simp add*: *eq-onp-same-args sup-lifted-distrib-lattice-top.abs-eq*)
    have (λ*f* . *get-dense* (*pf* ⊔ *qf*) *f*) = (λ*f* . *get-dense pf f* ⊔ *get-dense qf f*)
    proof
      fix *f*
      have (*top*,*up-filter* (*get-dense* (*pf* ⊔ *qf*) *f*)) = *Rep-lifted-pair* (*Rep-dense* (*pf* ⊔ *qf*)) *f*
        by (*metis get-dense-char*)
      also have ... = *triple.pairs-sup* (*Rep-lifted-pair* (*Rep-dense pf*) *f*) (*Rep-lifted-pair* (*Rep-dense qf*) *f*)
        by (*simp add*: *sup-lifted-pair.rep-eq sup-dense.rep-eq*)
      also have ... = *triple.pairs-sup* (*top*,*up-filter* (*get-dense pf f*)) (*top*,*up-filter* (*get-dense qf f*))
        by (*metis get-dense-char*)
      also have ... = (*top*,*up-filter* (*get-dense pf f*) ⊓ *up-filter* (*get-dense qf f*))
        by (*metis* (*no-types*, *lifting*) *calculation prod.simps*(*1*) *simp-phi triple.pairs-sup.simps triple-def*)
      also have ... = (*top*,*up-filter* (*get-dense pf f* ⊔ *get-dense qf f*))
        by (*metis up-filter-dist-sup*)
      finally show *get-dense* (*pf* ⊔ *qf*) *f* = *get-dense pf f* ⊔ *get-dense qf f*
        using *up-filter-injective* **by** *blast*
    qed
    thus *Abs-lifted-distrib-lattice-top* (*get-dense* (*pf* ⊔ *qf*)) = *Abs-lifted-distrib-lattice-top* (*get-dense pf*) ⊔ *Abs-lifted-distrib-lattice-top* (*get-dense qf*)
      using *1* **by** *metis*
  qed
next
  show ∀ *pf qf* :: ('*a*,'*b*) *lifted-pair dense* . *Abs-lifted-distrib-lattice-top* (*get-dense* (*pf* ⊓ *qf*)) = *Abs-lifted-distrib-lattice-top* (*get-dense pf*) ⊓ *Abs-lifted-distrib-lattice-top* (*get-dense qf*)
  proof (*intro allI*)
    fix *pf qf* :: ('*a*,'*b*) *lifted-pair dense*
    have *1*: *Abs-lifted-distrib-lattice-top* (*get-dense pf*) ⊓ *Abs-lifted-distrib-lattice-top* (*get-dense qf*) = *Abs-lifted-distrib-lattice-top* (λ*f* . *get-dense pf f* ⊓ *get-dense qf f*)
      by (*simp add*: *eq-onp-same-args inf-lifted-distrib-lattice-top.abs-eq*)
    have (λ*f* . *get-dense* (*pf* ⊓ *qf*) *f*) = (λ*f* . *get-dense pf f* ⊓ *get-dense qf f*)
    proof
      fix *f*
      have (*top*,*up-filter* (*get-dense* (*pf* ⊓ *qf*) *f*)) = *Rep-lifted-pair* (*Rep-dense* (*pf* ⊓ *qf*)) *f*
        by (*metis get-dense-char*)
      also have ... = *triple.pairs-inf* (*Rep-lifted-pair* (*Rep-dense pf*) *f*) (*Rep-lifted-pair* (*Rep-dense qf*) *f*)
        by (*simp add*: *inf-lifted-pair.rep-eq inf-dense.rep-eq*)
      also have ... = *triple.pairs-inf* (*top*,*up-filter* (*get-dense pf f*)) (*top*,*up-filter* (*get-dense qf f*))

by (*metis get-dense-char*)
also have ... = (*top*,*up-filter* (*get-dense pf f*) ⊔ *up-filter* (*get-dense qf f*))
by (*metis* (*no-types*, *lifting*) *calculation prod.simps*(*1*) *simp-phi triple.pairs-inf.simps triple-def*)
also have ... = (*top*,*up-filter* (*get-dense pf f* ⊓ *get-dense qf f*))
by (*metis up-filter-dist-inf*)
finally show *get-dense* (*pf* ⊓ *qf*) *f* = *get-dense pf f* ⊓ *get-dense qf f*
using *up-filter-injective* **by** *blast*
qed
thus *Abs-lifted-distrib-lattice-top* (*get-dense* (*pf* ⊓ *qf*)) = *Abs-lifted-distrib-lattice-top* (*get-dense pf*) ⊓ *Abs-lifted-distrib-lattice-top* (*get-dense qf*)
using *1* **by** *metis*
qed
next
show *bij dl-iso*
by (*rule invertible-bij*[**where** *g*=*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 ((λ*qf*::('*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* (λ*qf*::('*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* ∧ *y* = *Rep-lifted-distrib-lattice-top* (*dl-iso rf*) *f*)
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* (λ*g* . *if g* = *f then* (*top*,*up-filter y*) *else Rep-lifted-pair* (*Rep-dense qf*) *g*))
have *2*: ∀ *g* . (*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* (λ*g* . *if g* = *f then* (*top*,*up-filter y*) *else Rep-lifted-pair* (*Rep-dense qf*) *g*) = *Abs-lifted-pair* (λ*g* . *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* (λ*g* . *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* (λ*g* . *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* (λ*g* . *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*)
finally have *3*: *Abs-lifted-pair* (λ*g* . *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* (λ*g* . *if* *g* = *f* *then* (*top*,*up-filter* *y*) *else* *Rep-lifted-pair* (*Rep-dense* *qf*) *g*))) *f*)) = *Rep-lifted-pair* (*Rep-dense* (*Abs-dense* (*Abs-lifted-pair* (λ*g* . *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* (λ*g* . *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* (λ*g* . *if* *g* = *f* *then* (*top*,*up-filter* *y*) *else* *Rep-lifted-pair* (*Rep-dense* *qf*) *g*))) *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* (λ*g* . *if* *g* = *f* *then* (*top*,*up-filter* *y*) *else* *Rep-lifted-pair* (*Rep-dense* *qf*) *g*)))) *g*
by (*metis* (*mono-tags*, *lifting*) *get-dense-char*)
also have ... = *Rep-lifted-pair* (*Abs-lifted-pair* (λ*g* . *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 *2 5* **by** (*simp add*: *Abs-lifted-pair-inverse*)
also have ... = (*top*,*up-filter* (*get-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
}
hence *Rep-lifted-distrib-lattice-top* (*dl-iso qf*) ≤ *Rep-lifted-distrib-lattice-top* (*dl-iso ?rf*)
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
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 structure map *f* to obtain a filter of dense elements; or,
* apply the structure map *stone-phi* to the pair; then apply the distributive lattice isomorphism lifted to the resulting filter.

lemma *phi-iso*:
Rep-phi f (*Rep-lifted-boolean-algebra* (*ba-iso pf*) *f*) = *filter-map* (λ*qf*::('*a*::*non-trivial-boolean-algebra*,'*b*::*distrib-lattice-top*) *lifted-pair dense* . *Rep-lifted-distrib-lattice-top* (*dl-iso qf*) *f*) (*stone-phi pf*)
proof −
let *?r* = *Rep-phi f*
let *?ppf* = λ*g* . *triple.pairs-uminus* (*Rep-phi g*) (*Rep-lifted-pair* (*Rep-regular pf*) *g*)
have *1*: *triple ?r*
by (*simp add*: *triple-def*)
have *2*: *Rep-filter* (*?r* (*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*))) ⊆ { *z* . ∃ *qf* . −*Rep-regular pf* ≤ *Rep-dense qf* ∧ *z* = *get-dense qf f* }
proof
fix *z*
obtain *x* **where** *3*: *x* = *fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)
by *simp*
assume *z* ∈ *Rep-filter* (*?r* (*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)))

hence ↑*z* ⊆ *Rep-filter* (*?r x*)
using *3 filter-def* **by** *fastforce*
hence *4*: *up-filter z* ≤ *?r x*
by (*metis Rep-filter-cases Rep-filter-inverse less-eq-filter.rep-eq mem-Collect-eq up-filter*)
have *5*: ∀ *g* . *?ppf g* ∈ *triple.pairs* (*Rep-phi g*)
by (*metis* (*no-types*) *simp-lifted-pair uminus-lifted-pair.rep-eq*)
let *?zf* = λ*g* . *if g* = *f then* (*top*,*up-filter z*) *else triple.pairs-top*
have *6*: ∀ *g* . *?zf g* ∈ *triple.pairs* (*Rep-phi g*)
proof
fix *g* :: ('*a*,'*b*) *phi*
have *triple* (*Rep-phi g*)
by (*simp add*: *triple-def*)
hence (*top*,*up-filter z*) ∈ *triple.pairs* (*Rep-phi g*)
using *triple.pairs-def* **by** *force*
thus *?zf g* ∈ *triple.pairs* (*Rep-phi g*)
by (*metis simp-lifted-pair top-lifted-pair.rep-eq*)
qed
hence −*Abs-lifted-pair ?zf* = *Abs-lifted-pair* (λ*g* . *triple.pairs-uminus* (*Rep-phi g*) (*?zf g*))
by (*subst uminus-lifted-pair.abs-eq*) (*simp-all add*: *eq-onp-same-args*)
also have ... = *Abs-lifted-pair* (λ*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* (λ*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* ∈ *dense-elements*
by (*simp add*: *bot-lifted-pair.abs-eq*)
let *?qf* = *Abs-dense* (*Abs-lifted-pair ?zf*)
have ∀ *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* ≠ *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
qed
hence *Abs-lifted-pair ?ppf* ≤ *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* ≤ *Rep-dense ?qf*
using *7* **by** (*simp add*: *uminus-lifted-pair-def Abs-dense-inverse*)
have (*top*,*up-filter* (*get-dense ?qf f*)) = *Rep-lifted-pair* (*Rep-dense ?qf*) *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 ?qf f*
using *up-filter-injective* **by** *force*
thus *z* ∈ { *z* . ∃ *qf* . −*Rep-regular pf* ≤ *Rep-dense qf* ∧ *z* = *get-dense qf f* }
using *11* **by** *auto*
qed
have *12*: *Rep-filter* (*?r* (*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*))) ⊇ { *z* . ∃ *qf* . −*Rep-regular pf* ≤ *Rep-dense qf* ∧ *z* = *get-dense qf f* }
proof
fix *z*
assume *z* ∈ { *z* . ∃ *qf* . −*Rep-regular pf* ≤ *Rep-dense qf* ∧ *z* = *get-dense qf f* }
hence ∃ *qf* . −*Rep-regular pf* ≤ *Rep-dense qf* ∧ *z* = *get-dense qf f*
by *auto*
hence *triple.pairs-less-eq* (*Rep-lifted-pair* (−*Rep-regular pf*) *f*) (*top*,*up-filter z*)
by (*metis less-eq-lifted-pair.rep-eq get-dense-char*)
hence *up-filter z* ≤ *snd* (*Rep-lifted-pair* (−*Rep-regular pf*) *f*)
using *1* **by** (*metis* (*no-types*, *hide-lams*) *prod.collapse triple.pairs-less-eq.simps*)
also have ... = *snd* (*?ppf f*)
by (*metis uminus-lifted-pair.rep-eq*)
also have ... = *?r* (*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*))
using *1* **by** (*metis* (*no-types*) *prod.collapse prod.inject triple.pairs-uminus.simps*)
finally have *Rep-filter* (*up-filter z*) ⊆ *Rep-filter* (*?r* (*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)))
by (*simp add*: *less-eq-filter.rep-eq*)
hence ↑*z* ⊆ *Rep-filter* (*?r* (*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)))
by (*metis Abs-filter-inverse mem-Collect-eq up-filter*)
thus *z* ∈ *Rep-filter* (*?r* (*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)))
by *blast*
qed
have *13*: ∀ *qf*∈*Rep-filter* (*stone-phi pf*) . *Rep-lifted-distrib-lattice-top* (*Abs-lifted-distrib-lattice-top* (*get-dense qf*)) *f* = *get-dense qf f*
by (*metis Abs-lifted-distrib-lattice-top-inverse UNIV-I UNIV-def*)
have *Rep-filter* (*?r* (*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*))) = { *z* . ∃ *qf*∈*stone-phi-set pf* . *z* = *get-dense qf f* }
using *2 12* **by** *simp*
hence *?r* (*fst* (*Rep-lifted-pair* (*Rep-regular pf*) *f*)) = *Abs-filter* { *z* .

∃ *qf*∈*stone-phi-set pf* . *z* = *get-dense qf f* }
  by (*metis Rep-filter-inverse*)
 hence *?r* (*Rep-lifted-boolean-algebra* (*ba-iso pf*) *f*) = *Abs-filter* { *z* .
∃ *qf*∈*Rep-filter* (*stone-phi pf*) . *z* = *Rep-lifted-distrib-lattice-top* (*dl-iso qf*) *f* }
  using *13* **by** (*simp add*: *Abs-filter-inverse stone-phi-set-filter stone-phi-def Abs-lifted-boolean-algebra-inverse*)
 thus *?thesis*
  by (*simp add*: *image-def*)
qed

end

References

[1] A. Armstrong, S. Foster, G. Struth, and T. Weber. Relation algebra. *Archive of Formal Proofs*, 2016, first version 2014.

[2] A. Armstrong, V. B. F. Gomes, and G. Struth. Kleene algebra with tests and demonic refinement algebras. *Archive of Formal Proofs*, 2016, first version 2014.

[3] A. Armstrong, V. B. F. Gomes, G. Struth, and T. Weber. Kleene algebra. *Archive of Formal Proofs*, 2016, first version 2013.

[4] R. Balbes and P. Dwinger. *Distributive Lattices*. University of Missouri Press, 1974.

[5] G. Birkhoff. *Lattice Theory*, volume XXV of *Colloquium Publications*. American Mathematical Society, third edition, 1967.

[6] T. S. Blyth. *Lattices and Ordered Algebraic Structures*. Springer, 2005.

[7] C. C. Chen and G. Grätzer. Stone lattices. I: Construction theorems. *Canadian Journal of Mathematics*, 21:884–894, 1969.

[8] H. B. Curry. *Foundations of Mathematical Logic*. Dover Publications, 1977.

[9] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[10] J. Divasón and J. Aransay. Echelon form. *Archive of Formal Proofs*, 2016, first version 2015.

[11] S. Foster and G. Struth. Regular algebras. *Archive of Formal Proofs*, 2016, first version 2014.

[12] S. Foster, G. Struth, and T. Weber. Automated engineering of relational and algebraic methods in Isabelle/HOL. In H. de Swart, editor, *Relational and Algebraic Methods in Computer Science*, volume 6663 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2011.

[13] H. Furusawa and G. Struth. Binary multirelations. *Archive of Formal Proofs*, 2016, first version 2015.

[14] G. Georgescu, L. Leustean, and V. Preoteasa. Pseudo-hoops. *Archive of Formal Proofs*, 2016, first version 2011.

[15] V. B. F. Gomes, W. Guttmann, P. Höfner, G. Struth, and T. Weber. Kleene algebras with domain. *Archive of Formal Proofs*, 2016.

[16] V. B. F. Gomes and G. Struth. Residuated lattices. *Archive of Formal Proofs*, 2016, first version 2015.

[17] G. Grätzer. *Lattice Theory: First Concepts and Distributive Lattices.* W. H. Freeman and Co., 1971.

[18] G. Grätzer and E. T. Schmidt. On ideal theory for lattices. *Acta Scientiarium Mathematicarum*, 19(1–2):82–92, 1958.

[19] W. Guttmann. Isabelle/HOL theories of algebras for iteration, infinite executions and correctness of sequential computations. Technical Report TR-COSC 02/15, University of Canterbury, 2015.

[20] W. Guttmann. Relation-algebraic verification of Prim's minimum spanning tree algorithm. In A. Sampaio and F. Wang, editors, *International Colloquium on Theoretical Aspects of Computing*, volume 9965 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2016.

[21] T. Katriňák. A new proof of the construction theorem for Stone algebras. *Proceedings of the American Mathematical Society*, 40(1):75–78, 1973.

[22] G. Klein, R. Kolanski, and A. Boyton. Separation algebra. *Archive of Formal Proofs*, 2016, first version 2012.

[23] R. D. Maddux. Relation-algebraic semantics. *Theoretical Comput. Sci.*, 160(1–2):1–85, 1996.

[24] V. Preoteasa. Algebra of monotonic Boolean transformers. *Archive of Formal Proofs*, 2016, first version 2011.

[25] V. Preoteasa. Lattice properties. *Archive of Formal Proofs*, 2016, first version 2011.

[26] M. Wampler-Doty. A complete proof of the Robbins conjecture. *Archive of Formal Proofs*, 2016, first version 2010.