Lattice Properties

Viorel Preoteasa

March 17, 2025

Abstract

This formalization introduces and collects some algebraic structures based on lattices and complete lattices for use in other developments. The structures introduced are modular, and lattice ordered groups. In addition to the results proved for the new lattices, this formalization also introduces theorems about latices and complete lattices in general.

Contents

1 Overview

Section 2 introduces well founded and transitive relations. Section 3 introduces some properties about fixpoints of monotonic application which maps monotonic functions to monotonic functions. The most important property is that such a monotonic application has the least fixpoint monotonic. Section 4 introduces conjunctive, disjunctive, universally conjunctive, and universally disjunctive functions. In section 5 some simplification lemmas for alttices are proved. Section 6 introduces modular lattices and proves some properties about them and about distributive lattices. The main result of this section is that a lattice is distributive if and only if it satisfies

$$\forall x\ y\ z : x \sqcap z = y \sqcap z \wedge x \sqcup z = y \sqcup z \longrightarrow x = y$$

Section 7 introduces lattice ordered groups and some of their properties. The most important is that they are distributive lattices, and this property is proved using the results from Section 5.

2 Well founded and transitive relations

theory *WellFoundedTransitive*
imports *Main*
begin

class *transitive* = *ord* +
  assumes *order-trans1*: $x < y \Longrightarrow y < z \Longrightarrow x < z$
  and *less-eq-def*: $x \leq y \longleftrightarrow x = y \vee x < y$
begin

lemma *eq-less-eq* [*simp*]:
  $x = y \Longrightarrow x \leq y$
  by (*simp add*: *less-eq-def*)

lemma *order-trans2* [*simp*]:
  $x \leq y \Longrightarrow y < z \Longrightarrow x < z$
  apply (*simp add*: *less-eq-def*)
  apply *auto*
  apply (*erule less-eq-def order-trans1*)
  by *assumption*

lemma *order-trans3*:
  $x < y \Longrightarrow y \leq z \Longrightarrow x < z$
  apply (*simp add*: *less-eq-def*)
  apply *auto*
  apply (*erule less-eq-def order-trans1*)
  by *assumption*
end

class *well-founded* = *ord* +
  assumes *less-induct1* [*case-names less*]: (!!x . (!!y . $y < x \Longrightarrow P\ y) \Longrightarrow P\ x$) $\Longrightarrow P\ a$

class *well-founded-transitive* = *transitive* + *well-founded*

instantiation *prod*:: (*ord*, *ord*) *ord*
begin

definition

```
  less-pair-def: a < b ⟷ fst a < fst b ∨ (fst a = fst b ∧ snd a < snd b)

definition
  less-eq-pair-def: (a::('a::ord * 'b::ord)) <= b ⟷ a = b ∨ a < b
instance proof qed
end

instantiation prod:: (transitive, transitive) transitive
begin
instance proof
  fix x y z :: ('a::transitive * 'b::transitive)
  assume x < y and y < z then show x < z
    apply (simp add: less-pair-def)
    apply auto
    apply (drule order-trans1)
    apply auto
    apply (drule order-trans1)
    apply auto
    apply (drule order-trans1)
    apply auto
    done
next
  fix x y :: 'a * 'b
  show x ≤ y ⟷ x = y ∨ x < y
    by (simp add: less-eq-pair-def)
qed
end

instantiation prod:: (well-founded, well-founded) well-founded
begin
instance proof
  fix P::('a * 'b) ⇒ bool
  have a: ∀P. (∀x::'a . (∀y. y < x ⟶ P y) ⟶ P x) ⟶ (∀a . P a)
    apply safe
    apply (rule less-induct1)
    by blast
  have b: ∀P. (∀x::'b. (∀y . y < x ⟶ P y) ⟶ P x) ⟶ (∀a . P a)
    apply safe
    apply (rule less-induct1)
    by blast
  from a and b have c: (∀x. (∀y. y < x ⟶ P y) ⟶ P x) ⟶ (∀a. P a)
    apply (unfold less-pair-def)
    apply (rule impI)
    apply (simp (no-asm-use) only: split-paired-All)
    apply (unfold fst-conv snd-conv)
    apply (drule spec)
    apply (erule mp)
    apply (rule allI)
    apply (rule impI)
```

  apply (*drule spec*)
  apply (*erule mp*)
  by *blast*
 assume *A*: (!! *x*. (!! *y*. $y < x \Longrightarrow P\ y) \Longrightarrow P\ x$)
 fix *a*
 from *c A* **show** *P a* **by** *blast*
qed
end

instantiation *prod*:: (*well-founded-transitive*, *well-founded-transitive*) *well-founded-transitive*
begin
instance proof qed
end

instantiation *nat* :: *transitive*
begin

instance proof
 fix *x y z*::*nat*
 assume $x < y$ **and** $y < z$ **then show** $x < z$ **by** *simp*
 next
 fix *x y*::*nat* **show** $(x \leq y) \longleftrightarrow (x = y \vee x < y)$
  apply (*unfold le-less*)
  by *safe*
 qed
end

instantiation *nat*:: *well-founded*
begin
instance proof
 fix *P*::*nat* ⇒ *bool*
 fix *a*
 assume *A*: $(\bigwedge x\ .\ (\bigwedge y\ .\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x)$
 show *P a*
 by (*rule less-induct*, *rule A*, *simp*)
 qed
end

instantiation *nat*:: *well-founded-transitive*
begin
instance proof qed
end

end

3 Fixpoints and Complete Lattices

theory *Complete-Lattice-Prop*
imports *WellFoundedTransitive*

begin

This theory introduces some results about fixpoints of functions on complete lattices. The main result is that a monotonic function mapping momotonic functions to monotonic functions has the least fixpoint monotonic.

```
context complete-lattice begin

lemma inf-Inf: assumes nonempty: A ≠ {}
  shows inf x (Inf A) = Inf ((inf x) ‘ A)
  using assms by (auto simp add: INF-inf-const1 nonempty)

end
```

```
definition
  mono-mono F = (mono F ∧ (∀ f . mono f ⟶ mono (F f)))

theorem lfp-mono [simp]:
  mono-mono F ⟹ mono (lfp F)
  apply (simp add: mono-mono-def)
  apply (rule-tac f=F and P = mono in lfp-ordinal-induct)
  apply (simp-all add: mono-def)
  apply (intro allI impI SUP-least)
  apply (rule-tac y = f y in order-trans)
  apply (auto intro: SUP-upper)
  done

lemma gfp-ordinal-induct:
  fixes f :: 'a::complete-lattice => 'a
  assumes mono: mono f
  and P-f: !!S. P S ==> P (f S)
  and P-Union: !!M. ∀ S∈M. P S ==> P (Inf M)
  shows P (gfp f)
proof −
  let ?M = {S. gfp f ≤ S ∧ P S}
  have P (Inf ?M) using P-Union by simp
  also have Inf ?M = gfp f
  proof (rule antisym)
    show gfp f ≤ Inf ?M by (blast intro: Inf-greatest)
    hence f (gfp f) ≤ f (Inf ?M) by (rule mono [THEN monoD])
    hence gfp f ≤ f (Inf ?M) using mono [THEN gfp-unfold] by simp
    hence f (Inf ?M) ∈ ?M using P-f P-Union by simp
    hence Inf ?M ≤ f (Inf ?M) by (rule Inf-lower)
    thus Inf ?M ≤ gfp f by (rule gfp-upperbound)
  qed
  finally show ?thesis .
qed
```

theorem *gfp-mono* [*simp*]:
mono-mono F $\Longrightarrow$ *mono* (*gfp F*)
apply (*simp add*: *mono-mono-def*)
apply (*rule-tac f=F* **and** *P = mono* **in** *gfp-ordinal-induct*)
apply (*simp-all*, *safe*)
apply (*simp-all add*: *mono-def*)
apply (*intro allI impI INF-greatest*)
apply (*rule-tac y = f x* **in** *order-trans*)
apply (*auto intro*: *INF-lower*)
done

context *complete-lattice* **begin**

definition
Sup-less x (*w*::'*b*::*well-founded*) = *Sup* {*y* ::'*a* . $\exists$ *v* < *w* . *y* = *x v*}

lemma *Sup-less-upper*:
v < *w* $\Longrightarrow$ *P v* $\leq$ *Sup-less P w*
by (*simp add*: *Sup-less-def*, *rule Sup-upper*, *blast*)

lemma *Sup-less-least*:
(!! *v* . *v* < *w* $\Longrightarrow$ *P v* $\leq$ *Q*) $\Longrightarrow$ *Sup-less P w* $\leq$ *Q*
by (*simp add*: *Sup-less-def*, *rule Sup-least*, *blast*)

end

lemma *Sup-less-fun-eq*:
((*Sup-less P w*) *i*) = (*Sup-less* (λ *v* . *P v i*)) *w*
apply (*simp add*: *Sup-less-def fun-eq-iff*)
apply (*rule arg-cong* [*of* - - *Sup*])
apply *auto*
done

theorem *fp-wf-induction*:
f x = *x* $\Longrightarrow$ *mono f* $\Longrightarrow$ ($\forall$ *w* . (*y w*) $\leq$ *f* (*Sup-less y w*)) $\Longrightarrow$ *Sup* (*range y*) $\leq$ *x*
apply (*rule Sup-least*)
apply (*simp add*: *image-def*, *safe*, *simp*)
apply (*rule less-induct1*, *simp-all*)
apply (*rule-tac y = f* (*Sup-less y xa*) **in** *order-trans*, *simp*)
apply (*drule-tac x = Sup-less y xa* **and** *y = x* **in** *monoD*)
by (*simp add*: *Sup-less-least*, *auto*)

end

4 Conjunctive and Disjunctive Functions

theory *Conj-Disj*
imports *Main*
begin

This theory introduces the definitions and some properties for conjunctive, disjunctive, universally conjunctive, and universally disjunctive functions.

locale *conjunctive* =
  fixes *inf-b* :: $'b \Rightarrow 'b \Rightarrow 'b$
  and *inf-c* :: $'c \Rightarrow 'c \Rightarrow 'c$
  and *times-abc* :: $'a \Rightarrow 'b \Rightarrow 'c$
begin

definition
  conjunctive = {*x* . (∀ *y z* . *times-abc x* (*inf-b y z*) = *inf-c* (*times-abc x y*) (*times-abc x z*))}

lemma *conjunctiveI*:
  assumes (⋀*b c*. *times-abc a* (*inf-b b c*) = *inf-c* (*times-abc a b*) (*times-abc a c*))
  shows *a* ∈ *conjunctive*
  using *assms* **by** (*simp add*: *conjunctive-def*)

lemma *conjunctiveD*: *x* ∈ *conjunctive* ⟹ *times-abc x* (*inf-b y z*) = *inf-c* (*times-abc x y*) (*times-abc x z*)
  by (*simp add*: *conjunctive-def*)

end

interpretation *Apply*: *conjunctive inf*::$'a$::*semilattice-inf* $\Rightarrow 'a \Rightarrow 'a$
  inf::$'b$::*semilattice-inf* $\Rightarrow 'b \Rightarrow 'b$ λ *f* . *f*
  done

interpretation *Comp*: *conjunctive inf*::($'a$::*lattice* $\Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$
  inf::($'a$::*lattice* $\Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$ (*o*)
  done

lemma *Apply.conjunctive* = *Comp.conjunctive*
  apply (*simp add*: *Apply.conjunctive-def Comp.conjunctive-def*)
  apply *safe*
  apply (*simp-all add*: *fun-eq-iff inf-fun-def*)
  apply (*drule-tac x* = λ *u* . *y* **in** *spec*)
  apply (*drule-tac x* = λ *u* . *z* **in** *spec*)
  by *simp*

locale *disjunctive* =
  fixes *sup-b* :: $'b \Rightarrow 'b \Rightarrow 'b$
  and *sup-c* :: $'c \Rightarrow 'c \Rightarrow 'c$

and *times-abc* :: $'a \Rightarrow 'b \Rightarrow 'c$
begin

definition
disjunctive = $\{x$. $(\forall$ *y z* . *times-abc x* (*sup-b y z*) = *sup-c* (*times-abc x y*) (*times-abc x z*))$\}$

lemma *disjunctiveI*:
assumes ($\bigwedge$*b c*. *times-abc a* (*sup-b b c*) = *sup-c* (*times-abc a b*) (*times-abc a c*))
shows $a \in$ *disjunctive*
using *assms* **by** (*simp add*: *disjunctive-def*)

lemma *disjunctiveD*: $x \in$ *disjunctive* $\Longrightarrow$ *times-abc x* (*sup-b y z*) = *sup-c* (*times-abc x y*) (*times-abc x z*)
by (*simp add*: *disjunctive-def*)

end

interpretation *Apply*: *disjunctive sup*::$'a$::*semilattice-sup* $\Rightarrow 'a \Rightarrow 'a$
sup::$'b$::*semilattice-sup* $\Rightarrow 'b \Rightarrow 'b$ λ *f* . *f*
done

interpretation *Comp*: *disjunctive sup*::($'a$::*lattice* $\Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$
sup::($'a$::*lattice* $\Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$ (o)
done

lemma *apply-comp-disjunctive*: *Apply.disjunctive* = *Comp.disjunctive*
apply (*simp add*: *Apply.disjunctive-def Comp.disjunctive-def*)
apply *safe*
apply (*simp-all add*: *fun-eq-iff sup-fun-def*)
apply (*drule-tac x* = λ *u* . *y* **in** *spec*)
apply (*drule-tac x* = λ *u* . *z* **in** *spec*)
by *simp*

locale *Conjunctive* =
fixes *Inf-b* :: $'b$ *set* $\Rightarrow 'b$
and *Inf-c* :: $'c$ *set* $\Rightarrow 'c$
and *times-abc* :: $'a \Rightarrow 'b \Rightarrow 'c$
begin

definition
Conjunctive = $\{x$. $(\forall$ *X* . *times-abc x* (*Inf-b X*) = *Inf-c* ((*times-abc x*) ‘ *X*))$\}$

lemma *ConjunctiveI*:
assumes $\bigwedge$*A*. *times-abc a* (*Inf-b A*) = *Inf-c* ((*times-abc a*) ‘ *A*)
shows $a \in$ *Conjunctive*
using *assms* **by** (*simp add*: *Conjunctive-def*)

```
lemma ConjunctiveD:
  assumes a ∈ Conjunctive
  shows times-abc a (Inf-b A) = Inf-c ((times-abc a) ‘ A)
  using assms by (simp add: Conjunctive-def)

end

interpretation Apply: Conjunctive Inf Inf λ f . f
  done

interpretation Comp: Conjunctive Inf::(('a::complete-lattice ⇒ 'a) set) ⇒ ('a ⇒
'a)
  Inf::(('a::complete-lattice ⇒ 'a) set) ⇒ ('a ⇒ 'a) (o)
  done

lemma Apply.Conjunctive = Comp.Conjunctive
proof
  show Apply.Conjunctive ⊆ (Comp.Conjunctive :: ('a ⇒ 'a) set)
  proof
    fix f
    assume f ∈ (Apply.Conjunctive :: ('a ⇒ 'a) set)
    then have ∗: f (Inf A) = (INF a∈A. f a) for A
      by (auto dest!: Apply.ConjunctiveD)
    show f ∈ (Comp.Conjunctive :: ('a ⇒ 'a) set)
    proof (rule Comp.ConjunctiveI)
      fix G :: ('a ⇒ 'a) set
      from ∗ have f (INF f∈G. f a) = Inf (f ‘ (λf. f a) ‘ G)
        for a :: 'a .
      then show f ∘ Inf G = Inf (comp f ‘ G)
        by (simp add: fun-eq-iff image-comp)
    qed
  qed
  show Comp.Conjunctive ⊆ (Apply.Conjunctive :: ('a ⇒ 'a) set)
  proof
    fix f
    assume f ∈ (Comp.Conjunctive :: ('a ⇒ 'a) set)
    then have ∗: f ∘ Inf G = (INF g∈G. f ∘ g) for G :: ('a ⇒ 'a) set
      by (auto dest!: Comp.ConjunctiveD)
    show f ∈ (Apply.Conjunctive :: ('a ⇒ 'a) set)
    proof (rule Apply.ConjunctiveI)
      fix A :: 'a set
      from ∗ have f ∘ (INF a∈A. (λb :: 'a. a)) = Inf ((∘) f ‘ (λa b. a) ‘ A) .
      then show f (Inf A) = Inf (f ‘ A)
        by (simp add: fun-eq-iff image-comp)
    qed
  qed
qed

locale Disjunctive =
```

```
  fixes Sup-b :: 'b set ⇒ 'b
  and Sup-c :: 'c set ⇒ 'c
  and times-abc :: 'a ⇒ 'b ⇒ 'c
begin

definition
  Disjunctive = {x . (∀ X . times-abc x (Sup-b X) = Sup-c ((times-abc x) ` X) )}

lemma DisjunctiveI:
  assumes ⋀A. times-abc a (Sup-b A) = Sup-c ((times-abc a) ` A)
  shows a ∈ Disjunctive
  using assms by (simp add: Disjunctive-def)

lemma DisjunctiveD: x ∈ Disjunctive ⟹ times-abc x (Sup-b X) = Sup-c ((times-abc
x) ` X)
  by (simp add: Disjunctive-def)

end

interpretation Apply: Disjunctive Sup Sup λ f . f
  done

interpretation Comp: Disjunctive Sup::(('a::complete-lattice ⇒ 'a) set) ⇒ ('a ⇒
'a)
  Sup::(('a::complete-lattice ⇒ 'a) set) ⇒ ('a ⇒ 'a) (o)
  done

lemma Apply.Disjunctive = Comp.Disjunctive
proof
  show Apply.Disjunctive ⊆ (Comp.Disjunctive :: ('a ⇒ 'a) set)
  proof
    fix f
    assume f ∈ (Apply.Disjunctive :: ('a ⇒ 'a) set)
    then have *: f (Sup A) = (SUP a∈A. f a) for A
      by (auto dest!: Apply.DisjunctiveD)
    show f ∈ (Comp.Disjunctive :: ('a ⇒ 'a) set)
    proof (rule Comp.DisjunctiveI)
      fix G :: ('a ⇒ 'a) set
      from * have f (SUP f∈G. f a) = Sup (f ` (λf. f a) ` G)
        for a :: 'a .
      then show f ∘ Sup G = Sup (comp f ` G)
        by (simp add: fun-eq-iff image-comp)
    qed
  qed
  show Comp.Disjunctive ⊆ (Apply.Disjunctive :: ('a ⇒ 'a) set)
  proof
    fix f
    assume f ∈ (Comp.Disjunctive :: ('a ⇒ 'a) set)
    then have *: f ∘ Sup G = (SUP g∈G. f ∘ g) for G :: ('a ⇒ 'a) set
```

```
      by (auto dest!: Comp.DisjunctiveD)
    show f ∈ (Apply.Disjunctive :: ('a ⇒ 'a) set)
    proof (rule Apply.DisjunctiveI)
      fix A :: 'a set
      from * have f ∘ (SUP a∈A. (λb :: 'a. a)) = Sup ((∘) f ‘ (λa b. a) ‘ A) .
      then show f (Sup A) = Sup (f ‘ A)
        by (simp add: fun-eq-iff image-comp)
    qed
  qed
qed

lemma [simp]: (F::'a::complete-lattice ⇒ 'b::complete-lattice) ∈ Apply.Conjunctive
⟹ F ∈ Apply.conjunctive
  apply (simp add: Apply.Conjunctive-def Apply.conjunctive-def)
  apply safe
  apply (drule-tac x = {y, z} in spec)
  by simp

lemma [simp]: F ∈ Apply.conjunctive ⟹ mono F
  apply (simp add: Apply.conjunctive-def mono-def)
  apply safe
  apply (drule-tac x = x in spec)
  apply (drule-tac x = y in spec)
  apply (subgoal-tac inf x y = x)
  apply simp
  apply (subgoal-tac inf (F x) (F y) ≤ F y)
  apply simp
  apply (rule inf-le2)
  apply (rule antisym)
  by simp-all

lemma [simp]: (F::'a::complete-lattice ⇒ 'b::complete-lattice) ∈ Apply.Conjunctive
⟹ F top = top
  apply (simp add: Apply.Conjunctive-def)
  apply (drule-tac x={} in spec)
  by simp

lemma [simp]: (F::'a::complete-lattice ⇒ 'b::complete-lattice) ∈ Apply.Disjunctive
⟹ F ∈ Apply.disjunctive
  apply (simp add: Apply.Disjunctive-def Apply.disjunctive-def)
  apply safe
  apply (drule-tac x = {y, z} in spec)
  by simp

lemma [simp]: F ∈ Apply.disjunctive ⟹ mono F
  apply (simp add: Apply.disjunctive-def mono-def)
  apply safe
  apply (drule-tac x = x in spec)
  apply (drule-tac x = y in spec)
```

apply (*subgoal-tac sup x y = y*)
apply *simp*
apply (*subgoal-tac F x ≤ sup* (*F x*) (*F y*))
apply *simp*
apply (*rule sup-ge1*)
apply (*rule antisym*)
apply *simp*
by (*rule sup-ge2*)

lemma [*simp*]: (*F*::*'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*) ∈ *Apply.Disjunctive* ⟹ *F bot* = *bot*
apply (*simp add*: *Apply.Disjunctive-def*)
apply (*drule-tac x*={} **in** *spec*)
by *simp*

lemma *weak-fusion*: *h* ∈ *Apply.Disjunctive* ⟹ *mono f* ⟹ *mono g* ⟹
h o f ≤ *g o h* ⟹ *h* (*lfp f*) ≤ *lfp g*
apply (*rule-tac P* = *λ x . h x* ≤ *lfp g* **in** *lfp-ordinal-induct*, *simp-all*)
apply (*rule-tac y* = *g* (*h S*) **in** *order-trans*)
apply (*simp add*: *le-fun-def*)
apply (*rule-tac y* = *g* (*lfp g*) **in** *order-trans*)
apply (*rule-tac f* = *g* **in** *monoD*, *simp-all*)
apply (*simp add*: *lfp-unfold* [*symmetric*])
apply (*simp add*: *Apply.DisjunctiveD*)
by (*rule SUP-least*, *blast*)

lemma *inf-Disj*: (*λ* (*x*::*'a*::*complete-distrib-lattice*) . *inf x y*) ∈ *Apply.Disjunctive*
by (*simp add*: *Apply.Disjunctive-def fun-eq-iff Sup-inf*)

end

5 Simplification Lemmas for Lattices

theory *Lattice-Prop*
imports *Main*
begin

This theory introduces some simplification lemmas for semilattices and lattices

notation
inf (**infixl** ‹⊓› *70*) **and**
sup (**infixl** ‹⊔› *65*)

context *semilattice-inf* **begin**
lemma [*simp*]: (*x* ⊓ *y*) ⊓ *z* ≤ *x*
by (*metis inf-le1 order-trans*)

lemma [*simp*]: *x* ⊓ *y* ⊓ *z* ≤ *y*
by (*rule-tac y* = *x* ⊓ *y* **in** *order-trans*, *rule inf-le1*, *simp*)

lemma [*simp*]: $x \sqcap (y \sqcap z) \leq y$
by (*rule-tac* $y = y \sqcap z$ **in** *order-trans*, *rule inf-le2*, *simp*)

lemma [*simp*]: $x \sqcap (y \sqcap z) \leq z$
by (*rule-tac* $y = y \sqcap z$ **in** *order-trans*, *rule inf-le2*, *simp*)
end

context *semilattice-sup* **begin**

lemma [*simp*]: $x \leq x \sqcup y \sqcup z$
by (*rule-tac* $y = x \sqcup y$ **in** *order-trans*, *simp-all*)

lemma [*simp*]: $y \leq x \sqcup y \sqcup z$
by (*rule-tac* $y = x \sqcup y$ **in** *order-trans*, *simp-all*)

lemma [*simp*]: $y \leq x \sqcup (y \sqcup z)$
by (*rule-tac* $y = y \sqcup z$ **in** *order-trans*, *simp-all*)

lemma [*simp*]: $z \leq x \sqcup (y \sqcup z)$
by (*rule-tac* $y = y \sqcup z$ **in** *order-trans*, *simp-all*)
end

context *lattice* **begin**

lemma [*simp*]: $x \sqcap y \leq x \sqcup z$
by (*rule-tac* $y = x$ **in** *order-trans*, *simp-all*)

lemma [*simp*]: $y \sqcap x \leq x \sqcup z$
by (*rule-tac* $y = x$ **in** *order-trans*, *simp-all*)

lemma [*simp*]: $x \sqcap y \leq z \sqcup x$
by (*rule-tac* $y = x$ **in** *order-trans*, *simp-all*)

lemma [*simp*]: $y \sqcap x \leq z \sqcup x$
by (*rule-tac* $y = x$ **in** *order-trans*, *simp-all*)

end

end

6 Modular and Distributive Lattices

theory *Modular-Distrib-Lattice*
imports *Lattice-Prop*
begin

The main result of this theory is the fact that a lattice is distributive if and only if it satisfies the following property:

term $(\forall\ x\ y\ z\ .\ x \sqcap z = y \sqcap z \wedge x \sqcup z = y \sqcup z \Longrightarrow x = y)$

This result was proved by Bergmann in [1]. The formalization presented here is based on [3, 4].

class *modular-lattice* = *lattice* +
  assumes *modular*: $x \leq y \Longrightarrow\ x \sqcup (y \sqcap z) = y \sqcap (x \sqcup z)$

context *distrib-lattice* **begin**
subclass *modular-lattice*
  apply *unfold-locales*
  by (*simp add*: *inf-sup-distrib inf-absorb2*)
end

context *lattice* **begin**
definition
  d-aux a b c = $(a \sqcap b) \sqcup (b \sqcap c) \sqcup (c \sqcap a)$

lemma *d-b-c-a*: *d-aux b c a* = *d-aux a b c*
  by (*metis d-aux-def sup.assoc sup-commute*)

lemma *d-c-a-b*: *d-aux c a b* = *d-aux a b c*
  by (*metis d-aux-def sup.assoc sup-commute*)

definition
  e-aux a b c = $(a \sqcup b) \sqcap (b \sqcup c) \sqcap (c \sqcup a)$

lemma *e-b-c-a*: *e-aux b c a* = *e-aux a b c*
  by (*simp add*: *e-aux-def ac-simps*)

lemma *e-c-a-b*: *e-aux c a b* = *e-aux a b c*
  by (*simp add*: *e-aux-def ac-simps*)

definition
  a-aux a b c = $(a \sqcap$ (*e-aux a b c*)) $\sqcup$ (*d-aux a b c*)

definition
  b-aux a b c = $(b \sqcap$ (*e-aux a b c*)) $\sqcup$ (*d-aux a b c*)

definition
  c-aux a b c = $(c \sqcap$ (*e-aux a b c*)) $\sqcup$ (*d-aux a b c*)

lemma *b-a*: *b-aux a b c* = *a-aux b c a*
  by (*simp add*: *a-aux-def b-aux-def e-b-c-a d-b-c-a*)

lemma *c-a*: *c-aux a b c* = *a-aux c a b*
  by (*simp add*: *a-aux-def c-aux-def e-c-a-b d-c-a-b*)

lemma [*simp*]: *a-aux a b c* $\leq$ *e-aux a b c*
  apply (*simp add*: *a-aux-def e-aux-def d-aux-def*)

apply (*rule-tac* $y = (a \sqcup b) \sqcap (b \sqcup c) \sqcap (c \sqcup a)$ **in** *order-trans*)
apply (*rule inf-le2*)
by *simp*

lemma [*simp*]: *b-aux a b c* $\leq$ *e-aux a b c*
apply (*unfold b-a*)
apply (*subst e-b-c-a* [*THEN sym*])
by *simp*

lemma [*simp*]: *c-aux a b c* $\leq$ *e-aux a b c*
apply (*unfold c-a*)
apply (*subst e-c-a-b* [*THEN sym*])
by *simp*

lemma [*simp*]: *d-aux a b c* $\leq$ *a-aux a b c*
by (*simp add*: *a-aux-def e-aux-def d-aux-def*)

lemma [*simp*]: *d-aux a b c* $\leq$ *b-aux a b c*
apply (*unfold b-a*)
apply (*subst d-b-c-a* [*THEN sym*])
by *simp*

lemma [*simp*]: *d-aux a b c* $\leq$ *c-aux a b c*
apply (*unfold c-a*)
apply (*subst d-c-a-b* [*THEN sym*])
by *simp*

lemma *a-meet-e*: $a \sqcap$ (*e-aux a b c*) $= a \sqcap (b \sqcup c)$
by (*rule order.antisym*) (*simp-all add*: *e-aux-def le-infI2*)

lemma *b-meet-e*: $b \sqcap$ (*e-aux a b c*) $= b \sqcap (c \sqcup a)$
by (*simp add*: *a-meet-e* [*THEN sym*] *e-b-c-a*)

lemma *c-meet-e*: $c \sqcap$ (*e-aux a b c*) $= c \sqcap (a \sqcup b)$
by (*simp add*: *a-meet-e* [*THEN sym*] *e-c-a-b*)

lemma *a-join-d*: $a \sqcup$ *d-aux a b c* $= a \sqcup (b \sqcap c)$
by (*rule order.antisym*) (*simp-all add*: *d-aux-def le-supI2*)

lemma *b-join-d*: $b \sqcup$ *d-aux a b c* $= b \sqcup (c \sqcap a)$
by (*simp add*: *a-join-d* [*THEN sym*] *d-b-c-a*)

end

context *lattice* **begin**
definition
no-distrib a b c $= (a \sqcap b \sqcup c \sqcap a < a \sqcap (b \sqcup c))$

definition

incomp x y = (¬ *x* ≤ *y* ∧ ¬ *y* ≤ *x*)

definition
N5-lattice a b c = (*a* ⊓ *c* = *b* ⊓ *c* ∧ *a* < *b* ∧ *a* ⊔ *c* = *b* ⊔ *c*)

definition
M5-lattice a b c = (*a* ⊓ *b* = *b* ⊓ *c* ∧ *c* ⊓ *a* = *b* ⊓ *c* ∧ *a* ⊔ *b* = *b* ⊔ *c* ∧ *c* ⊔ *a* = *b* ⊔ *c* ∧ *a* ⊓ *b* < *a* ⊔ *b*)

lemma *M5-lattice-incomp*: *M5-lattice a b c* ⟹ *incomp a b*
apply (*simp add*: *M5-lattice-def incomp-def*)
apply *safe*
apply (*simp-all add*: *inf-absorb1 inf-absorb2*)
apply (*simp-all add*: *sup-absorb1 sup-absorb2*)
apply (*subgoal-tac c* ⊓ (*b* ⊔ *c*) = *c*)
apply *simp*
apply (*subst sup-commute*)
by *simp*
end

context *modular-lattice* **begin**

lemma *a-meet-d*: *a* ⊓ (*d-aux a b c*) = (*a* ⊓ *b*) ⊔ (*c* ⊓ *a*)
proof −
have *a* ⊓ (*d-aux a b c*) = *a* ⊓ ((*a* ⊓ *b*) ⊔ (*b* ⊓ *c*) ⊔ (*c* ⊓ *a*)) **by** (*simp add*: *d-aux-def*)
also have ... = *a* ⊓ (*a* ⊓ *b* ⊔ *c* ⊓ *a* ⊔ *b* ⊓ *c*) **by** (*simp add*: *sup-assoc*, *simp add*: *sup-commute*)
also have ... = (*a* ⊓ *b* ⊔ *c* ⊓ *a*) ⊔ (*a* ⊓ (*b* ⊓ *c*)) **by** (*simp add*: *modular*)
also have ... = (*a* ⊓ *b*) ⊔ (*c* ⊓ *a*) **by** (*rule order.antisym*, *simp-all*, *rule-tac y* = *a* ⊓ *b* **in** *order-trans*, *simp-all*)
finally show *?thesis* **by** *simp*
qed

lemma *b-meet-d*: *b* ⊓ (*d-aux a b c*) = (*b* ⊓ *c*) ⊔ (*a* ⊓ *b*)
by (*simp add*: *a-meet-d* [*THEN sym*] *d-b-c-a*)

lemma *c-meet-d*: *c* ⊓ (*d-aux a b c*) = (*c* ⊓ *a*) ⊔ (*b* ⊓ *c*)
by (*simp add*: *a-meet-d* [*THEN sym*] *d-c-a-b*)

lemma *d-less-e*: *no-distrib a b c* ⟹ *d-aux a b c* < *e-aux a b c*
apply (*subst less-le*)
apply(*case-tac d-aux a b c* = *e-aux a b c*)
apply *simp-all*
apply (*simp add*: *no-distrib-def a-meet-e* [*THEN sym*] *a-meet-d* [*THEN sym*])
apply (*rule-tac y* = *a-aux a b c* **in** *order-trans*)
by *simp-all*

lemma *a-meet-b-eq-d*: *d-aux a b c* $\leq$ *e-aux a b c* $\Longrightarrow$ *a-aux a b c* $\sqcap$ *b-aux a b c* = *d-aux a b c*
proof −
assume *d-less-e*: *d-aux a b c* $\leq$ *e-aux a b c*
have (*a* $\sqcap$ *e-aux a b c* $\sqcup$ *d-aux a b c*) $\sqcap$ (*b* $\sqcap$ *e-aux a b c* $\sqcup$ *d-aux a b c*) = (*b* $\sqcap$ *e-aux a b c* $\sqcup$ *d-aux a b c*) $\sqcap$ (*d-aux a b c* $\sqcup$ *a* $\sqcap$ *e-aux a b c*)
by (*simp add*: *inf-commute sup-commute*)
also have ... = *d-aux a b c* $\sqcup$ ((*b* $\sqcap$ *e-aux a b c* $\sqcup$ *d-aux a b c*) $\sqcap$ (*a* $\sqcap$ *e-aux a b c*))
by (*simp add*: *modular*)
also have ... = *d-aux a b c* $\sqcup$ (*d-aux a b c* $\sqcup$ *e-aux a b c* $\sqcap$ *b*) $\sqcap$ (*a* $\sqcap$ *e-aux a b c*)
by (*simp add*: *inf-commute sup-commute*)
also have ... = *d-aux a b c* $\sqcup$ (*e-aux a b c* $\sqcap$ (*d-aux a b c* $\sqcup$ *b*)) $\sqcap$ (*a* $\sqcap$ *e-aux a b c*)
by (*cut-tac d-less-e*, *simp add*: *modular* [*THEN sym*] *less-le*)
also have ... = *d-aux a b c* $\sqcup$ ((*a* $\sqcap$ *e-aux a b c*) $\sqcap$ (*e-aux a b c* $\sqcap$ (*b* $\sqcup$ *d-aux a b c*)))
by (*simp add*: *inf-commute sup-commute*)
also have ... = *d-aux a b c* $\sqcup$ (*a* $\sqcap$ *e-aux a b c* $\sqcap$ (*b* $\sqcup$ *d-aux a b c*)) **by** (*simp add*: *inf-assoc*)
also have ... = *d-aux a b c* $\sqcup$ (*a* $\sqcap$ *e-aux a b c* $\sqcap$ (*b* $\sqcup$ (*c* $\sqcap$ *a*))) **by** (*simp add*: *b-join-d*)
also have ... = *d-aux a b c* $\sqcup$ (*a* $\sqcap$ (*b* $\sqcup$ *c*) $\sqcap$ (*b* $\sqcup$ (*c* $\sqcap$ *a*))) **by** (*simp add*: *a-meet-e*)
also have ... = *d-aux a b c* $\sqcup$ (*a* $\sqcap$ ((*b* $\sqcup$ *c*) $\sqcap$ (*b* $\sqcup$ (*c* $\sqcap$ *a*)))) **by** (*simp add*: *inf-assoc*)
also have ... = *d-aux a b c* $\sqcup$ (*a* $\sqcap$ (*b* $\sqcup$ ((*b* $\sqcup$ *c*) $\sqcap$ (*c* $\sqcap$ *a*)))) **by** (*simp add*: *modular*)
also have ... = *d-aux a b c* $\sqcup$ (*a* $\sqcap$ (*b* $\sqcup$ (*c* $\sqcap$ *a*))) **by** (*simp add*: *inf-absorb2*)
also have ... = *d-aux a b c* $\sqcup$ (*a* $\sqcap$ ((*c* $\sqcap$ *a*) $\sqcup$ *b*)) **by** (*simp add*: *sup-commute inf-commute*)
also have ... = *d-aux a b c* $\sqcup$ ((*c* $\sqcap$ *a*) $\sqcup$ (*a* $\sqcap$ *b*)) **by** (*simp add*: *modular*)
also have ... = *d-aux a b c*
by (*rule order.antisym*, *simp-all add*: *d-aux-def*)
finally show *?thesis* **by** (*simp add*: *a-aux-def b-aux-def*)
qed

lemma *b-meet-c-eq-d*: *d-aux a b c* $\leq$ *e-aux a b c* $\Longrightarrow$ *b-aux a b c* $\sqcap$ *c-aux a b c* = *d-aux a b c*
apply (*subst b-a*)
apply (*subgoal-tac c-aux a b c* = *b-aux b c a*)
apply *simp*
apply (*subst a-meet-b-eq-d*)
by (*simp-all add*: *c-aux-def b-aux-def d-b-c-a e-b-c-a*)

lemma *c-meet-a-eq-d*: *d-aux a b c* $\leq$ *e-aux a b c* $\Longrightarrow$ *c-aux a b c* $\sqcap$ *a-aux a b c* = *d-aux a b c*
apply (*subst c-a*)

apply (*subgoal-tac a-aux a b c* = *b-aux c a b*)
apply *simp*
apply (*subst a-meet-b-eq-d*)
by (*simp-all add*: *a-aux-def b-aux-def d-b-c-a e-b-c-a*)

lemma *a-def-equiv*: *d-aux a b c* $\leq$ *e-aux a b c* $\Longrightarrow$ *a-aux a b c* = (*a* $\sqcup$ *d-aux a b c*) $\sqcap$ *e-aux a b c*
apply (*simp add*: *a-aux-def*)
apply (*subst inf-commute*)
apply (*subst sup-commute*)
apply (*simp add*: *modular*)
by (*simp add*: *inf-commute sup-commute*)

lemma *b-def-equiv*: *d-aux a b c* $\leq$ *e-aux a b c* $\Longrightarrow$ *b-aux a b c* = (*b* $\sqcup$ *d-aux a b c*) $\sqcap$ *e-aux a b c*
apply (*cut-tac a* = *b* **and** *b* = *c* **and** *c* = *a* **in** *a-def-equiv*)
by (*simp-all add*: *d-b-c-a e-b-c-a b-a*)

lemma *c-def-equiv*: *d-aux a b c* $\leq$ *e-aux a b c* $\Longrightarrow$ *c-aux a b c* = (*c* $\sqcup$ *d-aux a b c*) $\sqcap$ *e-aux a b c*
apply (*cut-tac a* = *c* **and** *b* = *a* **and** *c* = *b* **in** *a-def-equiv*)
by (*simp-all add*: *d-c-a-b e-c-a-b c-a*)

lemma *a-join-b-eq-e*: *d-aux a b c* $\leq$ *e-aux a b c* $\Longrightarrow$ *a-aux a b c* $\sqcup$ *b-aux a b c* = *e-aux a b c*
proof −
assume *d-less-e*: *d-aux a b c* $\leq$ *e-aux a b c*
have ((*a* $\sqcup$ *d-aux a b c*) $\sqcap$ *e-aux a b c*) $\sqcup$ ((*b* $\sqcup$ *d-aux a b c*) $\sqcap$ *e-aux a b c*) = ((*b* $\sqcup$ *d-aux a b c*) $\sqcap$ *e-aux a b c*) $\sqcup$ (*e-aux a b c* $\sqcap$ (*a* $\sqcup$ *d-aux a b c*))
by (*simp add*: *inf-commute sup-commute*)
also have ... = *e-aux a b c* $\sqcap$ (((*b* $\sqcup$ *d-aux a b c*) $\sqcap$ *e-aux a b c*) $\sqcup$ (*a* $\sqcup$ *d-aux a b c*))
by (*simp add*: *modular*)
also have ... = *e-aux a b c* $\sqcap$ ((*e-aux a b c* $\sqcap$ (*d-aux a b c* $\sqcup$ *b*)) $\sqcup$ (*a* $\sqcup$ *d-aux a b c*))
by (*simp add*: *inf-commute sup-commute*)
also have ... = *e-aux a b c* $\sqcap$ ((*d-aux a b c* $\sqcup$ (*e-aux a b c* $\sqcap$ *b*)) $\sqcup$ (*a* $\sqcup$ *d-aux a b c*))
by (*cut-tac d-less-e*, *simp add*: *modular*)
also have ... = *e-aux a b c* $\sqcap$ ((*a* $\sqcup$ *d-aux a b c*) $\sqcup$ (*d-aux a b c* $\sqcup$ (*b* $\sqcap$ *e-aux a b c*)))
by (*simp add*: *inf-commute sup-commute*)
also have ... = *e-aux a b c* $\sqcap$ (*a* $\sqcup$ *d-aux a b c* $\sqcup$ (*b* $\sqcap$ *e-aux a b c*)) **by** (*simp add*: *sup-assoc*)
also have ... = *e-aux a b c* $\sqcap$ (*a* $\sqcup$ *d-aux a b c* $\sqcup$ (*b* $\sqcap$ (*c* $\sqcup$ *a*))) **by** (*simp add*: *b-meet-e*)
also have ... = *e-aux a b c* $\sqcap$ (*a* $\sqcup$ (*b* $\sqcap$ *c*) $\sqcup$ (*b* $\sqcap$ (*c* $\sqcup$ *a*))) **by** (*simp add*: *a-join-d*)
also have ... = *e-aux a b c* $\sqcap$ (*a* $\sqcup$ ((*b* $\sqcap$ *c*) $\sqcup$ (*b* $\sqcap$ (*c* $\sqcup$ *a*)))) **by** (*simp add*:

sup-assoc)
  also have ... = *e-aux a b c* ⊓ (*a* ⊔ (*b* ⊓ ((*b* ⊓ *c*) ⊔ (*c* ⊔ *a*)))) **by** (*simp add*: *modular*)
  also have ... = *e-aux a b c* ⊓ (*a* ⊔ (*b* ⊓ (*c* ⊔ *a*))) **by** (*simp add*: *sup-absorb2*)
  also have ... = *e-aux a b c* ⊓ (*a* ⊔ ((*c* ⊔ *a*) ⊓ *b*)) **by** (*simp add*: *sup-commute inf-commute*)
  also have ... = *e-aux a b c* ⊓ ((*c* ⊔ *a*) ⊓ (*a* ⊔ *b*)) **by** (*simp add*: *modular*)
  also have ... = *e-aux a b c*
    by (*rule order.antisym*, *simp-all*, *simp-all add*: *e-aux-def*)
  finally show *?thesis* **by** (*cut-tac d-less-e*, *simp add*: *a-def-equiv b-def-equiv*)
qed

lemma *b-join-c-eq-e*: *d-aux a b c* <= *e-aux a b c* ⟹ *b-aux a b c* ⊔ *c-aux a b c* = *e-aux a b c*
  apply (*subst b-a*)
  apply (*subgoal-tac c-aux a b c* = *b-aux b c a*)
  apply *simp*
  apply (*subst a-join-b-eq-e*)
  by (*simp-all add*: *c-aux-def b-aux-def d-b-c-a e-b-c-a*)

lemma *c-join-a-eq-e*: *d-aux a b c* <= *e-aux a b c* ⟹ *c-aux a b c* ⊔ *a-aux a b c* = *e-aux a b c*
  apply (*subst c-a*)
  apply (*subgoal-tac a-aux a b c* = *b-aux c a b*)
  apply *simp*
  apply (*subst a-join-b-eq-e*)
  by (*simp-all add*: *a-aux-def b-aux-def d-b-c-a e-b-c-a*)

lemma *no-distrib a b c* ⟹ *incomp a b*
  apply (*simp add*: *no-distrib-def incomp-def ac-simps*)
  using *order.strict-iff-not inf.absorb-iff2 inf.commute modular*
  apply *fastforce*
  done

lemma *M5-modular*: *no-distrib a b c* ⟹ *M5-lattice* (*a-aux a b c*) (*b-aux a b c*) (*c-aux a b c*)
  apply (*frule d-less-e*)
  by (*simp add*: *M5-lattice-def a-meet-b-eq-d b-meet-c-eq-d c-meet-a-eq-d a-join-b-eq-e b-join-c-eq-e c-join-a-eq-e*)

lemma *M5-modular-def*: *M5-lattice a b c* = (*a* ⊓ *b* = *b* ⊓ *c* ∧ *c* ⊓ *a* = *b* ⊓ *c* ∧ *a* ⊔ *b* = *b* ⊔ *c* ∧ *c* ⊔ *a* = *b* ⊔ *c* ∧ *a* ⊓ *b* < *a* ⊔ *b*)
  by (*simp add*: *M5-lattice-def*)

end

context *lattice* **begin**

lemma *not-modular-N5*: (¬ *class.modular-lattice inf* ((≤)::′*a* ⇒ ′*a* ⇒ *bool*) (<) *sup*) =
  (∃ *a b c*::′*a* . *N5-lattice a b c*)
 apply (*subgoal-tac class.lattice* (⊓) ((≤)::′*a* ⇒ ′*a* ⇒ *bool*) (<) *sup*)
 apply (*unfold N5-lattice-def class.modular-lattice-def class.modular-lattice-axioms-def*)
 apply *simp-all*
 apply *safe*
 apply (*subgoal-tac x* ⊔ *y* ⊓ *z* < *y* ⊓ (*x* ⊔ *z*))
 apply (*rule-tac x* = *x* ⊔ *y* ⊓ *z* **in** *exI*)
 apply (*rule-tac x* = *y* ⊓ (*x* ⊔ *z*) **in** *exI*)
 apply (*rule-tac x* = *z* **in** *exI*)
 apply *safe*
 apply (*rule order.antisym*)
 apply *simp*
 apply (*rule-tac y* = *x* ⊔ *y* ⊓ *z* **in** *order-trans*)
 apply *simp-all*
 apply (*rule-tac y* = *y* ⊓ *z* **in** *order-trans*)
 apply *simp-all*
 apply (*rule order.antisym*)
 apply *simp-all*
 apply (*rule-tac y* = *y* ⊓ (*x* ⊔ *z*) **in** *order-trans*)
 apply *simp-all*
 apply (*rule-tac y* = *x* ⊔ *z* **in** *order-trans*)
 apply *simp-all*
 apply (*rule neq-le-trans*)
 apply *simp*
 apply *simp*
 apply (*rule-tac x* = *a* **in** *exI*)
 apply (*rule-tac x* = *b* **in** *exI*)
 apply *safe*
 apply (*simp add*: *less-le*)
 apply (*rule-tac x* = *c* **in** *exI*)
 apply *simp*
 apply (*simp add*: *less-le*)
 apply *safe*
 apply (*subgoal-tac a* ⊔ *a* ⊓ *c* = *b*)
 apply (*unfold sup-inf-absorb*) [*1*]
 apply *simp*
 apply *simp*
 proof qed

lemma *not-distrib-N5-M5*: (¬ *class.distrib-lattice* (⊓) ((≤)::′*a* ⇒ ′*a* ⇒ *bool*) (<) (⊔)) =
  ((∃ *a b c*::′*a* . *N5-lattice a b c*) ∨ (∃ *a b c*::′*a* . *M5-lattice a b c*))
 apply (*unfold not-modular-N5* [*THEN sym*])
 proof
  assume *A*: ¬ *class.distrib-lattice* (⊓) ((≤)::′*a* ⇒ ′*a* ⇒ *bool*) (<) (⊔)
  have *B*: ∃ *a b c*:: ′*a* . (*a* ⊓ *b*) ⊔ (*a* ⊓ *c*) < *a* ⊓ (*b* ⊔ *c*)
   apply (*cut-tac A*)

```
      apply (unfold  class.distrib-lattice-def)
      apply safe
      apply simp-all
      proof
        fix x y z::'a
        assume A: ∀(a::'a) (b::'a) c::'a. ¬ a ⊓ b ⊔ a ⊓ c < a ⊓ (b ⊔ c)
        show x ⊔ y ⊓ z = (x ⊔ y) ⊓ (x ⊔ z)
          apply (cut-tac A)
          apply (rule distrib-imp1)
          by (simp add: less-le)
      qed
    from B show ¬ class.modular-lattice (⊓) ((≤)::'a ⇒ 'a ⇒ bool) (<) (⊔) ∨ (∃ a
b c::'a. M5-lattice a b c)
      proof (unfold disj-not1, safe)
        fix a b c::'a
        assume A: a ⊓ b ⊔ a ⊓ c < a ⊓ (b ⊔ c)
        assume B: class.modular-lattice (⊓) ((≤)::'a ⇒ 'a ⇒ bool) (<) (⊔)
        interpret modular: modular-lattice (⊓) ((≤)::'a ⇒ 'a ⇒ bool) (<) (⊔)
          by (fact B)

        have H: M5-lattice (a-aux a b c) (b-aux a b c) (c-aux a b c)
          apply (cut-tac a = a and b = b and c = c in  modular.M5-modular)
          apply (unfold no-distrib-def)
          by (simp-all add: A inf-commute)
        from H show ∃ a b c::'a. M5-lattice a b c by blast
     qed
   next
      assume A: ¬ class.modular-lattice (⊓) ((≤)::'a ⇒ 'a ⇒ bool) (<) (⊔) ∨
(∃ (a::'a) (b::'a) c::'a. M5-lattice a b c)
     show ¬ class.distrib-lattice (⊓) ((≤)::'a ⇒ 'a ⇒ bool) (<) (⊔)
       apply (cut-tac A)
       apply safe
       apply (erule notE)
       apply unfold-locales
       apply (unfold class.distrib-lattice-def)
       apply (unfold class.distrib-lattice-axioms-def)
       apply safe
       apply (simp add: sup-absorb2)
       apply (frule M5-lattice-incomp)
       apply (unfold M5-lattice-def)
       apply (drule-tac x = a in spec)
       apply (drule-tac x = b in spec)
       apply (drule-tac x = c in spec)
       apply safe
       proof −
         fix a b c:: 'a
         assume A:a ⊔ b ⊓ c = (a ⊔ b) ⊓ (a ⊔ c)
         assume B: a ⊓ b = b ⊓ c
         assume D: a ⊔ b = b ⊔ c
```

```
        assume E: c ⊔ a = b ⊔ c
        assume G: incomp a b
        have H: a ⊔ b ⊓ c = a by (simp add: B [THEN sym] sup-absorb1)
        have I: (a ⊔ b) ⊓ (a ⊔ c) = a ⊔ b by (cut-tac E, simp add: sup-commute
D)
        have J: a = a ⊔ b by (cut-tac A, simp add: H I)
        show False
          apply (cut-tac G J)
          apply (subgoal-tac b ≤ a)
          apply (simp add: incomp-def)
          apply (rule-tac y = a ⊔ b in order-trans)
          apply (rule sup-ge2)
          by simp
      qed
    qed

lemma distrib-not-N5-M5: (class.distrib-lattice (⊓) ((≤)::'a ⇒ 'a ⇒ bool) (<) (⊔))
=
   ((∀ a b c::'a . ¬ N5-lattice a b c) ∧ (∀ a b c::'a . ¬ M5-lattice a b c))
  apply (cut-tac not-distrib-N5-M5)
  by auto

lemma distrib-inf-sup-eq:
  (class.distrib-lattice (⊓) ((≤)::'a ⇒ 'a ⇒ bool) (<) (⊔)) =
    (∀ x y z::'a . x ⊓ z = y ⊓ z ∧ x ⊔ z = y ⊔ z ⟶ x = y)
  apply safe
  proof −
    fix x y z:: 'a
    assume A: class.distrib-lattice (⊓) ((≤) ::'a ⇒ 'a ⇒ bool) (<) (⊔)
    interpret distrib: distrib-lattice (⊓) (≤) :: 'a ⇒ 'a ⇒ bool (<) (⊔)
      by (fact A)
    assume B: x ⊓ z = y ⊓ z
    assume C: x ⊔ z = y ⊔ z
    have x = x ⊓ (x ⊔ z) by simp
    also have ... = x ⊓ (y ⊔ z) by (simp add: C)
    also have ... = (x ⊓ y) ⊔ (x ⊓ z) by (simp add: distrib.inf-sup-distrib)
    also have ... = (y ⊓ x) ⊔ (y ⊓ z) by (simp add: B inf-commute)
    also have ... = y ⊓ (x ⊔ z) by (simp add: distrib.inf-sup-distrib)
    also have ... = y by (simp add: C)
    finally show x = y .
  next
    assume A: (∀ x y z:: 'a. x ⊓ z = y ⊓ z ∧ x ⊔ z = y ⊔ z ⟶ x = y)
    have B: !! x y z :: 'a. x ⊓ z = y ⊓ z ∧ x ⊔ z = y ⊔ z ⟹ x = y
      by (cut-tac A, blast)
    show class.distrib-lattice (⊓) ((≤)::'a ⇒ 'a ⇒ bool) (<) (⊔)
      apply (unfold distrib-not-N5-M5)
      apply safe
      apply (unfold N5-lattice-def)
      apply (cut-tac x = a and y = b and z = c in B)
```

apply (*simp-all*)
apply (*unfold M5-lattice-def*)
apply (*cut-tac* $x = a$ **and** $y = b$ **and** $z = c$ **in** B)
by (*simp-all add*: *inf-commute sup-commute*)
qed
end

class *inf-sup-eq-lattice* = *lattice* +
assumes *inf-sup-eq*: $x \sqcap z = y \sqcap z \Longrightarrow x \sqcup z = y \sqcup z \Longrightarrow x = y$
begin
subclass *distrib-lattice*
by (*metis distrib-inf-sup-eq inf-sup-eq*)
end

end

7 Lattice Orderd Groups

theory *Lattice-Ordered-Group*
imports *Modular-Distrib-Lattice*
begin

This theory introduces lattice ordered groups [2] and proves some results about them. The most important result is that a lattice ordered group is also a distributive lattice.

class *lgroup* = *group-add* + *lattice* +
assumes *add-order-preserving*: $a \leq b \Longrightarrow u + a + v \leq u + b + v$
begin

lemma *add-order-preserving-left*: $a \leq b \Longrightarrow u + a \leq u + b$
apply (*cut-tac* $a = a$ **and** $b = b$ **and** $u = u$ **and** $v = 0$ **in** *add-order-preserving*)
by *simp-all*

lemma *add-order-preserving-right*: $a \leq b \Longrightarrow a + v \leq b + v$
apply (*cut-tac* $a = a$ **and** $b = b$ **and** $u = 0$ **and** $v = v$ **in** *add-order-preserving*)
by *simp-all*

lemma *minus-order*: $-a \leq -b \Longrightarrow b \leq a$
apply (*cut-tac* $a = -a$ **and** $b = -b$ **and** $u = a$ **and** $v = b$ **in** *add-order-preserving*)
by *simp-all*

lemma *right-move-to-left*: $a + - c \leq b \Longrightarrow a \leq b + c$
apply (*drule-tac* $v = c$ **in** *add-order-preserving-right*)
by (*simp add*: *add.assoc*)

lemma *right-move-to-right*: $a \leq b + -c \Longrightarrow a + c \leq b$

apply (*drule-tac* $v = c$ **in** *add-order-preserving-right*)
by (*simp add*: *add.assoc*)

lemma [*simp*]: $(a \sqcap b) + c = (a + c) \sqcap (b + c)$
apply (*rule order.antisym*)
apply *simp*
apply *safe*
apply (*rule add-order-preserving-right*)
apply *simp*
apply (*rule add-order-preserving-right*)
apply *simp*
apply (*rule right-move-to-left*)
apply *simp*
apply *safe*
apply (*simp-all only*: *diff-conv-add-uminus*)
apply (*rule right-move-to-right*)
apply *simp*
apply (*rule right-move-to-right*)
by *simp*

lemma [*simp*]: $(a \sqcap b) - c = (a - c) \sqcap (b - c)$
by (*simp add*: *diff-conv-add-uminus del*: *add-uminus-conv-diff*)

lemma *left-move-to-left*: $-c + a \le b \Longrightarrow a \le c + b$
apply (*drule-tac* $u = c$ **in** *add-order-preserving-left*)
by (*simp add*: *add.assoc* [*THEN sym*])

lemma *left-move-to-right*: $a \le - c + b \Longrightarrow c + a \le b$
apply (*drule-tac* $u = c$ **in** *add-order-preserving-left*)
by (*simp add*: *add.assoc* [*THEN sym*])

lemma [*simp*]: $c + (a \sqcap b) = (c + a) \sqcap (c + b)$
apply (*rule order.antisym*)
apply *simp*
apply *safe*
apply (*rule add-order-preserving-left*)
apply *simp*
apply (*rule add-order-preserving-left*)
apply *simp*
apply (*rule left-move-to-left*)
apply *simp*
apply *safe*
apply (*rule left-move-to-right*)
apply *simp*
apply (*rule left-move-to-right*)
by *simp*

lemma [*simp*]: $- (a \sqcap b) = (- a) \sqcup (- b)$
apply (*rule order.antisym*)
apply (*rule minus-order*)
apply *simp*
apply *safe*
apply (*rule minus-order*)
apply *simp*
apply (*rule minus-order*)
apply *simp*
apply *simp*
apply *safe*
apply (*rule minus-order*)
apply *simp*
apply (*rule minus-order*)
by *simp*

lemma [*simp*]: $(a \sqcup b) + c = (a + c) \sqcup (b + c)$
apply (*rule order.antisym*)
apply (*rule right-move-to-right*)
apply *simp*
apply *safe*
apply (*simp-all only*: *diff-conv-add-uminus*)
apply (*rule right-move-to-left*)
apply *simp*
apply (*rule right-move-to-left*)
apply *simp*
apply *simp*
apply *safe*
apply (*rule add-order-preserving-right*)
apply *simp*
apply (*rule add-order-preserving-right*)
by *simp*

lemma [*simp*]: $c + (a \sqcup b) = (c + a) \sqcup (c + b)$
apply (*rule order.antisym*)
apply (*rule left-move-to-right*)
apply *simp*
apply *safe*
apply (*rule left-move-to-left*)
apply *simp*
apply (*rule left-move-to-left*)
apply *simp*
apply *simp*
apply *safe*
apply (*rule add-order-preserving-left*)
apply *simp*
apply (*rule add-order-preserving-left*)
by *simp*

lemma [*simp*]: $c - (a \sqcap b) = (c - a) \sqcup (c - b)$
by (*simp add*: *diff-conv-add-uminus del*: *add-uminus-conv-diff*)

lemma [*simp*]: $(a \sqcup b) - c = (a - c) \sqcup (b - c)$
by (*simp add*: *diff-conv-add-uminus del*: *add-uminus-conv-diff*)

lemma [*simp*]: $- (a \sqcup b) = (- a) \sqcap (- b)$
apply (*rule order.antisym*)
apply *simp*
apply *safe*
apply (*rule minus-order*)
apply *simp*
apply (*rule minus-order*)
apply *simp*
apply (*rule minus-order*)
by *simp*

lemma [*simp*]: $c - (a \sqcup b) = (c - a) \sqcap (c - b)$
by (*simp add*: *diff-conv-add-uminus del*: *add-uminus-conv-diff*)

lemma *add-pos*: $0 \leq a \Longrightarrow b \leq b + a$
apply (*cut-tac* $a = 0$ **and** $b = a$ **and** $u = b$ **and** $v = 0$ **in** *add-order-preserving*)
by *simp-all*

lemma *add-pos-left*: $0 \leq a \Longrightarrow b \leq a + b$
apply (*rule right-move-to-left*)
by *simp*

lemma *inf-sup*: $a - (a \sqcap b) + b = a \sqcup b$
by (*simp add*: *add.assoc sup-commute*)

lemma *inf-sup-2*: $b = (a \sqcap b) - a + (a \sqcup b)$
apply (*unfold inf-sup* [*THEN sym*])
proof −
fix *a b*:: *'a*
have $b = (a \sqcap b) + (- a + a) + - (a \sqcap b) + b$ **by** (*simp only*: *right-minus left-minus add-0-right add-0-left*)
also have $\ldots = (a \sqcap b) + - a + (a + - (a \sqcap b) + b)$ **by** (*unfold add.assoc*, *simp*)
also have $\ldots = (a \sqcap b) - a + (a - (a \sqcap b) + b)$ **by** *simp*
finally show $b = (a \sqcap b) - a + (a - (a \sqcap b) + b)$ **.**
qed

subclass *inf-sup-eq-lattice*
proof
fix *x y z*:: *'a*
assume *A*: $x \sqcap z = y \sqcap z$
assume *B*: $x \sqcup z = y \sqcup z$

have $x = (z \sqcap x) - z + (z \sqcup x)$ **by** (*rule inf-sup-2*)
also have $\ldots = (z \sqcap y) - z + (z \sqcup y)$ **by** (*simp add*: *sup-commute inf-commute A B*)
also have $\ldots = y$ **by** (*simp only*: *inf-sup-2* [*THEN sym*])
finally show $x = y$ **.**
qed
end

end

References

[1] G. Bergmann. Zur axiomatik der elementargeometrie. *Monatshefte für Mathematik*, 36:269–284, 1929. 10.1007/BF02307616.

[2] G. Birkhoff. Lattice, ordered groups. *Ann. of Math. (2)*, 43:298–331, 1942.

[3] G. Birkhoff. *Lattice theory*. Third edition. American Mathematical Society Colloquium Publications, Vol. XXV. American Mathematical Society, Providence, R.I., 1967.

[4] S. Burris and H. P. Sankappanavar. *A course in universal algebra*, volume 78 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1981.