

Lattice Properties

Viorel Preoteasa

September 13, 2023

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 lattices and complete lattices in general.

Contents

1	Overview	1
2	Well founded and transitive relations	2
3	Fixpoints and Complete Lattices	4
4	Conjunctive and Disjunctive Functions	7
5	Simplification Lemmas for Lattices	12
6	Modular and Distributive Lattices	13
7	Lattice Orderd Groups	23

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 lattices 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:  $x < y \implies y < z \implies x < z$ 
  and less-eq-def:  $x \leq y \iff x = y \vee x < y$ 
begin
```

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

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

```
lemma order-trans3:
   $x < y \implies y \leq z \implies x < z$ 
  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]:  $(!!x . (!!y . y < x \implies P y) \implies P x) \implies P a$ 
```

```
class well-founded-transitive = transitive + well-founded
```

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

```
definition
```

less-pair-def: $a < b \iff \text{fst } a < \text{fst } b \vee (\text{fst } a = \text{fst } b \wedge \text{snd } a < \text{snd } b)$

definition

less-eq-pair-def: $(a::('a::\text{ord} * 'b::\text{ord})) \leq b \iff a = b \vee a < b$

instance proof qed

end

instantiation *prod*:: (*transitive, transitive*) *transitive*

begin

instance proof

fix $x\ y\ z :: ('a::\text{transitive} * 'b::\text{transitive})$

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 $x \leq y \iff x = y \vee x < y$

by (*simp add: less-eq-pair-def*)

qed

end

instantiation *prod*:: (*well-founded, well-founded*) *well-founded*

begin

instance proof

fix $P::('a * 'b) \Rightarrow \text{bool}$

have $a: \forall P. (\forall x::'a. (\forall y. y < x \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall a. P\ a)$

apply *safe*

apply (*rule less-induct1*)

by *blast*

have $b: \forall P. (\forall x::'b. (\forall y. y < x \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall a. P\ a)$

apply *safe*

apply (*rule less-induct1*)

by *blast*

from a **and** b **have** $c: (\forall x. (\forall y. y < x \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall a. P\ a)$

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  $\implies$  P y)  $\implies$  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  $\Rightarrow$  bool
  fix a
  assume A: ( $\bigwedge$ x . ( $\bigwedge$ y . y < x  $\implies$  P y)  $\implies$  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 monotonic functions to monotonic functions has the least fixpoint monotonic.

context *complete-lattice* **begin**

lemma *inf-Inf*: **assumes** *nonempty*: $A \neq \{\}$
shows $\text{inf } x (\text{Inf } A) = \text{Inf } ((\text{inf } x) \text{ ` } A)$
using *assms* **by** (*auto simp add: INF-inf-const1 nonempty*)

end

definition

$\text{mono-mono } F = (\text{mono } F \wedge (\forall f . \text{mono } f \longrightarrow \text{mono } (F f)))$

theorem *lfp-mono* [*simp*]:

$\text{mono-mono } F \Longrightarrow \text{mono } (\text{lfp } F)$
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 $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *mono*: $\text{mono } f$
and *P-f*: $\forall S. P S \Longrightarrow P (f S)$
and *P-Union*: $\forall M. \forall S \in M. P S \Longrightarrow P (\text{Inf } M)$
shows $P (\text{gfp } f)$

proof –

let $?M = \{S. \text{gfp } f \leq S \wedge P S\}$
have $P (\text{Inf } ?M)$ **using** *P-Union* **by** *simp*
also have $\text{Inf } ?M = \text{gfp } f$
proof (*rule antisym*)
show $\text{gfp } f \leq \text{Inf } ?M$ **by** (*blast intro: Inf-greatest*)
hence $f (\text{gfp } f) \leq f (\text{Inf } ?M)$ **by** (*rule mono [THEN monoD]*)
hence $\text{gfp } f \leq f (\text{Inf } ?M)$ **using** *mono [THEN gfp-unfold]* **by** *simp*
hence $f (\text{Inf } ?M) \in ?M$ **using** *P-f P-Union* **by** *simp*
hence $\text{Inf } ?M \leq f (\text{Inf } ?M)$ **by** (*rule Inf-lower*)
thus $\text{Inf } ?M \leq \text{gfp } f$ **by** (*rule gfp-upperbound*)

qed

finally show *?thesis* .

qed

```

theorem gfp-mono [simp]:
  mono-mono  $F \implies \text{mono } (\text{gfp } F)$ 
  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:

```

```

   $v < w \implies P v \leq \text{Sup-less } P w$ 
  by (simp add: Sup-less-def, rule Sup-upper, blast)

```

```

lemma Sup-less-least:

```

```

  ( $\forall v . v < w \implies P v \leq Q$ )  $\implies \text{Sup-less } P w \leq Q$ 
  by (simp add: Sup-less-def, rule Sup-least, blast)

```

```

end

```

```

lemma Sup-less-fun-eq:

```

```

  ( $(\text{Sup-less } P w) i = (\text{Sup-less } (\lambda v . P v i)) w$ )
  apply (simp add: Sup-less-def fun-eq-iff)
  apply (rule arg-cong [of - - Sup])
  apply auto
  done

```

```

theorem fp-wf-induction:

```

```

   $f x = x \implies \text{mono } f \implies (\forall w . (y w) \leq f (\text{Sup-less } y w)) \implies \text{Sup } (\text{range } y) \leq x$ 
  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 . ( $\forall$  y z . times-abc x (inf-b y z) = inf-c (times-abc x y)
(times-abc x z))}
```

lemma *conjunctiveI*:

```
assumes ( $\bigwedge$  b c . times-abc a (inf-b b c) = inf-c (times-abc a b) (times-abc a c))
shows a  $\in$  conjunctive
using assms by (simp add: conjunctive-def)
```

lemma *conjunctiveD*: $x \in \textit{conjunctive} \implies \textit{times-abc } x (\textit{inf-b } y z) = \textit{inf-c } (\textit{times-abc } x y) (\textit{times-abc } x z)$

```
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  $\lambda f . f$ 
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 =  $\lambda$  u . y in spec)
apply (drule-tac x =  $\lambda$  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 ⇒ 'b ⇒ 'c
begin

definition

disjunctive = {x . (∀ y z . *times-abc* x (*sup-b* y z) = *sup-c* (*times-abc* x y) (*times-abc* x z))}

lemma *disjunctiveI*:

assumes (∧ b c. *times-abc* a (*sup-b* b c) = *sup-c* (*times-abc* a b) (*times-abc* a c))
shows a ∈ *disjunctive*
using *assms* **by** (*simp add: disjunctive-def*)

lemma *disjunctiveD*: x ∈ *disjunctive* ⇒ *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* ⇒ 'a ⇒ 'a

sup::'b::semilattice-sup ⇒ 'b ⇒ 'b λ f . f

done

interpretation *Comp*: *disjunctive sup::('a::lattice ⇒ 'a) ⇒ ('a ⇒ 'a) ⇒ ('a ⇒ 'a)*

sup::('a::lattice ⇒ 'a) ⇒ ('a ⇒ 'a) ⇒ ('a ⇒ '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 ⇒ 'b

and *Inf-c* :: 'c set ⇒ 'c

and *times-abc* :: 'a ⇒ 'b ⇒ 'c

begin

definition

Conjunctive = {x . (∀ X . *times-abc* x (*Inf-b* X) = *Inf-c* ((*times-abc* x) ' X))}

lemma *ConjunctiveI*:

assumes ∧ A. *times-abc* a (*Inf-b* A) = *Inf-c* ((*times-abc* a) ' A)

shows a ∈ *Conjunctive*

using *assms* **by** (*simp add: Conjunctive-def*)


```

lemma ConjunctiveD:
  assumes  $a \in \text{Conjunctive}$ 
  shows  $\text{times-abc } a \text{ (Inf-b } A) = \text{Inf-c } ((\text{times-abc } a) \text{ ' } A)$ 
  using assms by (simp add: Conjunctive-def)

end

interpretation Apply: Conjunctive Inf Inf  $\lambda f . f$ 
  done

interpretation Comp: Conjunctive Inf::('a::complete-lattice  $\Rightarrow$  'a) set)  $\Rightarrow$  ( $'a \Rightarrow 'a$ )
  Inf::('a::complete-lattice  $\Rightarrow$  'a) set)  $\Rightarrow$  ( $'a \Rightarrow 'a$ ) (o)
  done

lemma Apply.Conjunctive = Comp.Conjunctive
proof
  show Apply.Conjunctive  $\subseteq$  (Comp.Conjunctive :: ( $'a \Rightarrow 'a$ ) set)
  proof
    fix  $f$ 
    assume  $f \in (\text{Apply.Conjunctive} :: ('a \Rightarrow 'a) \text{ set})$ 
    then have  $*$ :  $f \text{ (Inf } A) = (\text{INF } a \in A. f a)$  for  $A$ 
      by (auto dest!: Apply.ConjunctiveD)
    show  $f \in (\text{Comp.Conjunctive} :: ('a \Rightarrow 'a) \text{ set})$ 
    proof (rule Comp.ConjunctiveI)
      fix  $G :: ('a \Rightarrow 'a) \text{ set}$ 
      from  $*$  have  $f \text{ (INF } f \in G. f a) = \text{Inf } (f \text{ ' } (\lambda f. f a) \text{ ' } G)$ 
        for  $a :: 'a .$ 
      then show  $f \circ \text{Inf } G = \text{Inf } (\text{comp } f \text{ ' } G)$ 
        by (simp add: fun-eq-iff image-comp)
    qed
  qed
  show Comp.Conjunctive  $\subseteq$  (Apply.Conjunctive :: ( $'a \Rightarrow 'a$ ) set)
  proof
    fix  $f$ 
    assume  $f \in (\text{Comp.Conjunctive} :: ('a \Rightarrow 'a) \text{ set})$ 
    then have  $*$ :  $f \circ \text{Inf } G = (\text{INF } g \in G. f \circ g)$  for  $G :: ('a \Rightarrow 'a) \text{ set}$ 
      by (auto dest!: Comp.ConjunctiveD)
    show  $f \in (\text{Apply.Conjunctive} :: ('a \Rightarrow 'a) \text{ set})$ 
    proof (rule Apply.ConjunctiveI)
      fix  $A :: 'a \text{ set}$ 
      from  $*$  have  $f \circ (\text{INF } a \in A. (\lambda b :: 'a. a)) = \text{Inf } ((\circ) f \text{ ' } (\lambda a b. a) \text{ ' } A) .$ 
      then show  $f \text{ (Inf } A) = \text{Inf } (f \text{ ' } A)$ 
        by (simp add: fun-eq-iff image-comp)
    qed
  qed
qed
qed
locale Disjunctive =

```

```

fixes Sup-b :: 'b set  $\Rightarrow$  'b
and Sup-c :: 'c set  $\Rightarrow$  'c
and times-abc :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'c
begin

definition
  Disjunctive = {x . ( $\forall$  X . times-abc x (Sup-b X) = Sup-c ((times-abc x) ' X))}

lemma DisjunctiveI:
  assumes  $\bigwedge A$ . times-abc a (Sup-b A) = Sup-c ((times-abc a) ' A)
  shows a  $\in$  Disjunctive
  using assms by (simp add: Disjunctive-def)

lemma DisjunctiveD: x  $\in$  Disjunctive  $\implies$  times-abc x (Sup-b X) = Sup-c ((times-abc x) ' X)
  by (simp add: Disjunctive-def)

end

interpretation Apply: Disjunctive Sup Sup  $\lambda$  f . f
  done

interpretation Comp: Disjunctive Sup::('a::complete-lattice  $\Rightarrow$  'a) set  $\Rightarrow$  ('a  $\Rightarrow$  'a)
  Sup::('a::complete-lattice  $\Rightarrow$  'a) set  $\Rightarrow$  ('a  $\Rightarrow$  'a) (o)
  done

lemma Apply.Disjunctive = Comp.Disjunctive
proof
  show Apply.Disjunctive  $\subseteq$  (Comp.Disjunctive :: ('a  $\Rightarrow$  'a) set)
  proof
    fix f
    assume f  $\in$  (Apply.Disjunctive :: ('a  $\Rightarrow$  'a) set)
    then have *: f (Sup A) = (SUP a $\in$ A. f a) for A
      by (auto dest!: Apply.DisjunctiveD)
    show f  $\in$  (Comp.Disjunctive :: ('a  $\Rightarrow$  'a) set)
    proof (rule Comp.DisjunctiveI)
      fix G :: ('a  $\Rightarrow$  'a) set
      from * have f (SUP f $\in$ G. f a) = Sup (f ' ( $\lambda$ f. f a) ' G)
        for a :: 'a .
      then show f  $\circ$  Sup G = Sup (comp f ' G)
        by (simp add: fun-eq-iff image-comp)
    qed
  qed
  show Comp.Disjunctive  $\subseteq$  (Apply.Disjunctive :: ('a  $\Rightarrow$  'a) set)
  proof
    fix f
    assume f  $\in$  (Comp.Disjunctive :: ('a  $\Rightarrow$  'a) set)
    then have *: f  $\circ$  Sup G = (SUP g $\in$ G. f  $\circ$  g) for G :: ('a  $\Rightarrow$  'a) set

```

```

    by (auto dest!: Comp.DisjunctiveD)
  show  $f \in (Apply.Disjunctive :: ('a \Rightarrow 'a) set)$ 
  proof (rule Apply.DisjunctiveI)
    fix  $A :: 'a set$ 
    from * have  $f \circ (SUP a \in A. (\lambda b :: 'a. a)) = Sup ((\circ) f ' (\lambda a b. a) ' A)$  .
    then show  $f (Sup A) = Sup (f ' A)$ 
      by (simp add: fun-eq-iff image-comp)
  qed
qed
qed

```

```

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

```

```

lemma [simp]:  $F \in Apply.conjunctive \Longrightarrow mono F$ 
  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  $(F x) (F y) \leq F y$ )
  apply simp
  apply (rule inf-le2)
  apply (rule antisym)
  by simp-all

```

```

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

```

```

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

```

```

lemma [simp]:  $F \in Apply.disjunctive \Longrightarrow mono F$ 
  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 $(\forall x y z . x \sqcap z = y \sqcap z \wedge x \sqcup z = y \sqcup z \implies x = y)$

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

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

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

context *lattice* **begin**

definition
 $d\text{-aux } a b c = (a \sqcap b) \sqcup (b \sqcap c) \sqcup (c \sqcap a)$

lemma *d-b-c-a*: $d\text{-aux } b c a = d\text{-aux } a b c$
by (*metis d-aux-def sup.assoc sup-commute*)

lemma *d-c-a-b*: $d\text{-aux } c a b = d\text{-aux } a b c$
by (*metis d-aux-def sup.assoc sup-commute*)

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

lemma *e-b-c-a*: $e\text{-aux } b c a = e\text{-aux } a b c$
by (*simp add: e-aux-def ac-simps*)

lemma *e-c-a-b*: $e\text{-aux } c a b = e\text{-aux } a b c$
by (*simp add: e-aux-def ac-simps*)

definition
 $a\text{-aux } a b c = (a \sqcap (e\text{-aux } a b c)) \sqcup (d\text{-aux } a b c)$

definition
 $b\text{-aux } a b c = (b \sqcap (e\text{-aux } a b c)) \sqcup (d\text{-aux } a b c)$

definition
 $c\text{-aux } a b c = (c \sqcap (e\text{-aux } a b c)) \sqcup (d\text{-aux } a b c)$

lemma *b-a*: $b\text{-aux } a b c = a\text{-aux } b c a$
by (*simp add: a-aux-def b-aux-def e-b-c-a d-b-c-a*)

lemma *c-a*: $c\text{-aux } a b c = a\text{-aux } c a b$
by (*simp add: a-aux-def c-aux-def e-c-a-b d-c-a-b*)

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

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

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

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

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

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

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

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

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

lemma b-join-d:  $b \sqcup d\text{-aux } a \ b \ c = b \sqcup (c \sqcap a)$ 
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

```

$incomp\ x\ y = (\neg x \leq y \wedge \neg y \leq x)$

definition

$N5\text{-lattice}\ a\ b\ c = (a \sqcap c = b \sqcap c \wedge a < b \wedge a \sqcup c = b \sqcup c)$

definition

$M5\text{-lattice}\ a\ b\ c = (a \sqcap b = b \sqcap c \wedge c \sqcap a = b \sqcap c \wedge a \sqcup b = b \sqcup c \wedge c \sqcup a = b \sqcup c \wedge a \sqcap b < a \sqcup b)$

lemma $M5\text{-lattice-incomp}$: $M5\text{-lattice}\ a\ b\ c \implies incomp\ a\ b$

apply (*simp add*: $M5\text{-lattice-def}\ incomp\text{-def}$)

apply *safe*

apply (*simp-all add*: $inf\text{-absorb1}\ inf\text{-absorb2}$)

apply (*simp-all add*: $sup\text{-absorb1}\ sup\text{-absorb2}$)

apply (*subgoal-tac* $c \sqcap (b \sqcup c) = c$)

apply *simp*

apply (*subst sup-commute*)

by *simp*

end

context $modular\text{-lattice}$ **begin**

lemma $a\text{-meet-d}$: $a \sqcap (d\text{-aux}\ a\ b\ c) = (a \sqcap b) \sqcup (c \sqcap a)$

proof –

have $a \sqcap (d\text{-aux}\ a\ b\ c) = a \sqcap ((a \sqcap b) \sqcup (b \sqcap c) \sqcup (c \sqcap a))$ **by** (*simp add*: $d\text{-aux-def}$)

also have $\dots = a \sqcap (a \sqcap b \sqcup c \sqcap a \sqcup b \sqcap c)$ **by** (*simp add*: $sup\text{-assoc}$, *simp add*: $sup\text{-commute}$)

also have $\dots = (a \sqcap b \sqcup c \sqcap a) \sqcup (a \sqcap (b \sqcap c))$ **by** (*simp add*: $modular$)

also have $\dots = (a \sqcap b) \sqcup (c \sqcap a)$ **by** (*rule order.antisym*, *simp-all*, *rule-tac* $y = a \sqcap b$ **in** $order\text{-trans}$, *simp-all*)

finally show $?thesis$ **by** *simp*

qed

lemma $b\text{-meet-d}$: $b \sqcap (d\text{-aux}\ a\ b\ c) = (b \sqcap c) \sqcup (a \sqcap b)$

by (*simp add*: $a\text{-meet-d}$ [*THEN sym*] $d\text{-b-c-a}$)

lemma $c\text{-meet-d}$: $c \sqcap (d\text{-aux}\ a\ b\ c) = (c \sqcap a) \sqcup (b \sqcap c)$

by (*simp add*: $a\text{-meet-d}$ [*THEN sym*] $d\text{-c-a-b}$)

lemma $d\text{-less-e}$: $no\text{-distrib}\ a\ b\ c \implies d\text{-aux}\ a\ b\ c < e\text{-aux}\ a\ b\ c$

apply (*subst less-le*)

apply(*case-tac* $d\text{-aux}\ a\ b\ c = e\text{-aux}\ a\ b\ c$)

apply *simp-all*

apply (*simp add*: $no\text{-distrib-def}\ a\text{-meet-e}$ [*THEN sym*] $a\text{-meet-d}$ [*THEN sym*])

apply (*rule-tac* $y = a\text{-aux}\ a\ b\ c$ **in** $order\text{-trans}$)

by *simp-all*

lemma *a-meet-b-eq-d*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c \implies a\text{-aux } a \ b \ c \sqcap b\text{-aux } a \ b \ c = d\text{-aux } a \ b \ c$

proof –

assume *d-less-e*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c$

have $(a \sqcap e\text{-aux } a \ b \ c \sqcup d\text{-aux } a \ b \ c) \sqcap (b \sqcap e\text{-aux } a \ b \ c \sqcup d\text{-aux } a \ b \ c) = (b \sqcap e\text{-aux } a \ b \ c \sqcup d\text{-aux } a \ b \ c) \sqcap (d\text{-aux } a \ b \ c \sqcup a \sqcap e\text{-aux } a \ b \ c)$

by (*simp add: inf-commute sup-commute*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup ((b \sqcap e\text{-aux } a \ b \ c \sqcup d\text{-aux } a \ b \ c) \sqcap (a \sqcap e\text{-aux } a \ b \ c))$

by (*simp add: modular*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup (d\text{-aux } a \ b \ c \sqcup e\text{-aux } a \ b \ c \sqcap b) \sqcap (a \sqcap e\text{-aux } a \ b \ c)$

by (*simp add: inf-commute sup-commute*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup (e\text{-aux } a \ b \ c \sqcap (d\text{-aux } a \ b \ c \sqcup b)) \sqcap (a \sqcap e\text{-aux } a \ b \ c)$

by (*cut-tac d-less-e, simp add: modular [THEN sym] less-le*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup ((a \sqcap e\text{-aux } a \ b \ c) \sqcap (e\text{-aux } a \ b \ c \sqcap (b \sqcup d\text{-aux } a \ b \ c)))$

by (*simp add: inf-commute sup-commute*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup (a \sqcap e\text{-aux } a \ b \ c \sqcap (b \sqcup d\text{-aux } a \ b \ c))$ **by** (*simp add: inf-assoc*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup (a \sqcap e\text{-aux } a \ b \ c \sqcap (b \sqcup (c \sqcap a)))$ **by** (*simp add: b-join-d*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup (a \sqcap (b \sqcup c) \sqcap (b \sqcup (c \sqcap a)))$ **by** (*simp add: a-meet-e*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup (a \sqcap ((b \sqcup c) \sqcap (b \sqcup (c \sqcap a))))$ **by** (*simp add: inf-assoc*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup (a \sqcap (b \sqcup ((b \sqcup c) \sqcap (c \sqcap a))))$ **by** (*simp add: modular*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup (a \sqcap (b \sqcup (c \sqcap a)))$ **by** (*simp add: inf-absorb2*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup (a \sqcap ((c \sqcap a) \sqcup b))$ **by** (*simp add: sup-commute inf-commute*)

also have $\dots = d\text{-aux } a \ b \ c \sqcup ((c \sqcap a) \sqcup (a \sqcap b))$ **by** (*simp add: modular*)

also have $\dots = d\text{-aux } a \ b \ c$

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*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c \implies b\text{-aux } a \ b \ c \sqcap c\text{-aux } a \ b \ c = d\text{-aux } a \ b \ c$

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*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c \implies c\text{-aux } a \ b \ c \sqcap a\text{-aux } a \ b \ c = d\text{-aux } a \ b \ c$

apply (*subst c-a*)

apply (*subgoal-tac* $a\text{-aux } a \ b \ c = b\text{-aux } c \ a \ b$)
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*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c \implies a\text{-aux } a \ b \ c = (a \sqcup d\text{-aux } a \ b \ c) \sqcap e\text{-aux } a \ b \ c$
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*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c \implies b\text{-aux } a \ b \ c = (b \sqcup d\text{-aux } a \ b \ c) \sqcap e\text{-aux } a \ b \ c$
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*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c \implies c\text{-aux } a \ b \ c = (c \sqcup d\text{-aux } a \ b \ c) \sqcap e\text{-aux } a \ b \ c$
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*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c \implies a\text{-aux } a \ b \ c \sqcup b\text{-aux } a \ b \ c = e\text{-aux } a \ b \ c$

proof –

assume *d-less-e*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c$
have $((a \sqcup d\text{-aux } a \ b \ c) \sqcap e\text{-aux } a \ b \ c) \sqcup ((b \sqcup d\text{-aux } a \ b \ c) \sqcap e\text{-aux } a \ b \ c) = ((b \sqcup d\text{-aux } a \ b \ c) \sqcap e\text{-aux } a \ b \ c) \sqcup (e\text{-aux } a \ b \ c \sqcap (a \sqcup d\text{-aux } a \ b \ c))$
by (*simp* *add: inf-commute sup-commute*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap (((b \sqcup d\text{-aux } a \ b \ c) \sqcap e\text{-aux } a \ b \ c) \sqcup (a \sqcup d\text{-aux } a \ b \ c))$
by (*simp* *add: modular*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap ((e\text{-aux } a \ b \ c \sqcap (d\text{-aux } a \ b \ c \sqcup b)) \sqcup (a \sqcup d\text{-aux } a \ b \ c))$
by (*simp* *add: inf-commute sup-commute*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap ((d\text{-aux } a \ b \ c \sqcup (e\text{-aux } a \ b \ c \sqcap b)) \sqcup (a \sqcup d\text{-aux } a \ b \ c))$
by (*cut-tac* *d-less-e*, *simp* *add: modular*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap ((a \sqcup d\text{-aux } a \ b \ c) \sqcup (d\text{-aux } a \ b \ c \sqcup (b \sqcap e\text{-aux } a \ b \ c)))$
by (*simp* *add: inf-commute sup-commute*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap (a \sqcup d\text{-aux } a \ b \ c \sqcup (b \sqcap e\text{-aux } a \ b \ c))$ **by** (*simp* *add: sup-assoc*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap (a \sqcup d\text{-aux } a \ b \ c \sqcup (b \sqcap (c \sqcup a)))$ **by** (*simp* *add: b-meet-e*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap (a \sqcup (b \sqcap c) \sqcup (b \sqcap (c \sqcup a)))$ **by** (*simp* *add: a-join-d*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap (a \sqcup ((b \sqcap c) \sqcup (b \sqcap (c \sqcup a))))$ **by** (*simp* *add:*

sup-assoc)
also have $\dots = e\text{-aux } a \ b \ c \sqcap (a \sqcup (b \sqcap ((b \sqcap c) \sqcup (c \sqcup a))))$ **by** (*simp add: modular*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap (a \sqcup (b \sqcap (c \sqcup a)))$ **by** (*simp add: sup-absorb2*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap (a \sqcup ((c \sqcup a) \sqcap b))$ **by** (*simp add: sup-commute inf-commute*)
also have $\dots = e\text{-aux } a \ b \ c \sqcap ((c \sqcup a) \sqcap (a \sqcup b))$ **by** (*simp add: modular*)
also have $\dots = e\text{-aux } a \ b \ c$
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*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c \implies b\text{-aux } a \ b \ c \sqcup c\text{-aux } a \ b \ c = e\text{-aux } a \ b \ c$
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*: $d\text{-aux } a \ b \ c \leq e\text{-aux } a \ b \ c \implies c\text{-aux } a \ b \ c \sqcup a\text{-aux } a \ b \ c = e\text{-aux } a \ b \ c$
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* $\implies \text{incomp } a \ b$
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* $\implies M5\text{-lattice } (a\text{-aux } a \ b \ c) (b\text{-aux } a \ b \ c) (c\text{-aux } a \ b \ c)$
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*: $M5\text{-lattice } a \ b \ c = (a \sqcap b = b \sqcap c \wedge c \sqcap a = b \sqcap c \wedge a \sqcup b = b \sqcup c \wedge c \sqcup a = b \sqcup c \wedge a \sqcap b < a \sqcup b)$
by (*simp add: M5-lattice-def*)

end

context *lattice* **begin**

lemma *not-modular-N5*: $(\neg \text{class.modular-lattice inf } ((\leq)::'a \Rightarrow 'a \Rightarrow \text{bool}) (<) \text{ sup}) =$
 $(\exists a b c::'a . N5\text{-lattice } a b c)$
apply (*subgoal-tac class.lattice* $(\sqcap) ((\leq)::'a \Rightarrow 'a \Rightarrow \text{bool}) (<) \text{ sup}$)
apply (*unfold N5-lattice-def class.modular-lattice-def class.modular-lattice-axioms-def*)
apply *simp-all*
apply *safe*
apply (*subgoal-tac* $x \sqcup y \sqcap z < y \sqcap (x \sqcup z)$)
apply (*rule-tac* $x = x \sqcup y \sqcap z$ **in** *exI*)
apply (*rule-tac* $x = y \sqcap (x \sqcup z)$ **in** *exI*)
apply (*rule-tac* $x = z$ **in** *exI*)
apply *safe*
apply (*rule order.antisym*)
apply *simp*
apply (*rule-tac* $y = x \sqcup y \sqcap z$ **in** *order-trans*)
apply *simp-all*
apply (*rule-tac* $y = y \sqcap z$ **in** *order-trans*)
apply *simp-all*
apply (*rule order.antisym*)
apply *simp-all*
apply (*rule-tac* $y = y \sqcap (x \sqcup z)$ **in** *order-trans*)
apply *simp-all*
apply (*rule-tac* $y = x \sqcup z$ **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* $a \sqcup a \sqcap c = b$)
apply (*unfold sup-inf-absorb*) [1]
apply *simp*
apply *simp*
proof qed

lemma *not-distrib-N5-M5*: $(\neg \text{class.distrib-lattice } (\sqcap) ((\leq)::'a \Rightarrow 'a \Rightarrow \text{bool}) (<) (\sqcup)) =$
 $(\exists a b c::'a . N5\text{-lattice } a b c) \vee (\exists a b c::'a . M5\text{-lattice } a b c)$
apply (*unfold not-modular-N5* [*THEN sym*])
proof
assume *A*: $\neg \text{class.distrib-lattice } (\sqcap) ((\leq)::'a \Rightarrow 'a \Rightarrow \text{bool}) (<) (\sqcup)$
have *B*: $\exists a b c::'a . (a \sqcap b) \sqcup (a \sqcap c) < a \sqcap (b \sqcup c)$
apply (*cut-tac A*)

```

apply (unfold class.distrib-lattice-def)
apply safe
apply simp-all
proof
  fix x y z::'a
  assume A:  $\forall (a::'a) (b::'a) c::'a. \neg a \sqcap b \sqcup a \sqcap c < a \sqcap (b \sqcup c)$ 
  show  $x \sqcup y \sqcap z = (x \sqcup y) \sqcap (x \sqcup z)$ 
    apply (cut-tac A)
    apply (rule distrib-imp1)
    by (simp add: less-le)
qed
from B show  $\neg$  class.modular-lattice ( $\sqcap$ ) (( $\leq$ )::'a  $\Rightarrow$  'a  $\Rightarrow$  bool) (<) ( $\sqcup$ )  $\vee$  ( $\exists$  a
b c::'a. M5-lattice a b c)
proof (unfold disj-not1, safe)
  fix a b c::'a
  assume A:  $a \sqcap b \sqcup a \sqcap c < a \sqcap (b \sqcup c)$ 
  assume B: class.modular-lattice ( $\sqcap$ ) (( $\leq$ )::'a  $\Rightarrow$  'a  $\Rightarrow$  bool) (<) ( $\sqcup$ )
  interpret modular: modular-lattice ( $\sqcap$ ) (( $\leq$ )::'a  $\Rightarrow$  'a  $\Rightarrow$  bool) (<) ( $\sqcup$ )
    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  $\exists$  a b c::'a. M5-lattice a b c by blast
qed
next
  assume A:  $\neg$  class.modular-lattice ( $\sqcap$ ) (( $\leq$ )::'a  $\Rightarrow$  'a  $\Rightarrow$  bool) (<) ( $\sqcup$ )  $\vee$ 
( $\exists$  (a::'a) (b::'a) c::'a. M5-lattice a b c)
  show  $\neg$  class.distrib-lattice ( $\sqcap$ ) (( $\leq$ )::'a  $\Rightarrow$  'a  $\Rightarrow$  bool) (<) ( $\sqcup$ )
    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:  $a \sqcup b \sqcap c = (a \sqcup b) \sqcap (a \sqcup c)$ 
    assume B:  $a \sqcap b = b \sqcap c$ 
    assume D:  $a \sqcup b = b \sqcup c$ 

```

```

assume  $E: c \sqcup a = b \sqcup c$ 
assume  $G: \text{incomp } a \ b$ 
have  $H: a \sqcup b \sqcap c = a$  by (simp add: B [THEN sym] sup-absorb1)
have  $I: (a \sqcup b) \sqcap (a \sqcup c) = a \sqcup b$  by (cut-tac E, simp add: sup-commute)
D)
have  $J: a = a \sqcup b$  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 ( $\sqcap$ ) ( $((\leq)::'a \Rightarrow 'a \Rightarrow \text{bool})$ ) ( $<$ ) ( $\sqcup$ ))
=
  ( $(\forall a \ b \ c::'a . \neg \text{N5-lattice } a \ b \ c) \wedge (\forall a \ b \ c::'a . \neg \text{M5-lattice } a \ b \ c)$ )
apply (cut-tac not-distrib-N5-M5)
by auto

```

```

lemma distrib-inf-sup-eq:
(class.distrib-lattice ( $\sqcap$ ) ( $((\leq)::'a \Rightarrow 'a \Rightarrow \text{bool})$ ) ( $<$ ) ( $\sqcup$ )) =
  ( $(\forall x \ y \ z::'a . x \sqcap z = y \sqcap z \wedge x \sqcup z = y \sqcup z \longrightarrow x = y)$ )
apply safe
proof –
  fix  $x \ y \ z::'a$ 
  assume  $A: \text{class.distrib-lattice } (\sqcap) ((\leq)::'a \Rightarrow 'a \Rightarrow \text{bool}) (<) (\sqcup)$ 
  interpret distrib: distrib-lattice ( $\sqcap$ ) ( $\leq$ )  $:: 'a \Rightarrow 'a \Rightarrow \text{bool} (<) (\sqcup)$ 
  by (fact A)
  assume  $B: x \sqcap z = y \sqcap z$ 
  assume  $C: x \sqcup z = y \sqcup z$ 
  have  $x = x \sqcap (x \sqcup z)$  by simp
  also have  $\dots = x \sqcap (y \sqcup z)$  by (simp add: C)
  also have  $\dots = (x \sqcap y) \sqcup (x \sqcap z)$  by (simp add: distrib.inf-sup-distrib)
  also have  $\dots = (y \sqcap x) \sqcup (y \sqcap z)$  by (simp add: B inf-commute)
  also have  $\dots = y \sqcap (x \sqcup z)$  by (simp add: distrib.inf-sup-distrib)
  also have  $\dots = y$  by (simp add: C)
  finally show  $x = y$  .
next
  assume  $A: (\forall x \ y \ z::'a . x \sqcap z = y \sqcap z \wedge x \sqcup z = y \sqcup z \longrightarrow x = y)$ 
  have  $B: !! x \ y \ z::'a . x \sqcap z = y \sqcap z \wedge x \sqcup z = y \sqcup z \Longrightarrow x = y$ 
  by (cut-tac A, blast)
  show class.distrib-lattice ( $\sqcap$ ) ( $((\leq)::'a \Rightarrow 'a \Rightarrow \text{bool})$ ) ( $<$ ) ( $\sqcup$ )
  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:  $x \sqcap z = y \sqcap z \implies x \sqcup z = y \sqcup z \implies x = y$ 
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:  $a \leq b \implies u + a + v \leq u + b + v$ 
begin

```

```

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

```

```

lemma minus-order:  $-a \leq -b \implies b \leq a$ 
  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:  $a + -c \leq b \implies a \leq b + c$ 
  apply (drule-tac v = c in add-order-preserving-right)
  by (simp add: add.assoc)

```

```

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

```

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

lemma *left-move-to-right*: $a \leq -c + b \implies c + a \leq b$
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*: $0 \leq a \implies b \leq b + a$
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*: $0 \leq a \implies b \leq a + b$
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 $\dots = (a \sqcap b) + -a + (a + -(a \sqcap b) + b)$ **by** (*unfold add.assoc, simp*)
also have $\dots = (a \sqcap b) - a + (a - (a \sqcap b) + b)$ **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 $\dots = (z \sqcap y) - z + (z \sqcup y)$ **by** (*simp add: sup-commute inf-commute*
A B)
also have $\dots = y$ **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.