Lattices and Orders in Isabelle/HOL

Markus Wenzel
TU München

May 22, 2012

Abstract

We consider abstract structures of orders and lattices. Many fundamental concepts of lattice theory are developed, including dual structures, properties of bounds versus algebraic laws, lattice operations versus set-theoretic ones etc. We also give example instantiations of lattices and orders, such as direct products and function spaces. Well-known properties are demonstrated, like the Knaster-Tarski Theorem for complete lattices.

This formal theory development may serve as an example of applying Isabelle/HOL to the domain of mathematical reasoning about "axiomatic" structures. Apart from the simply-typed classical set-theory of HOL, we employ Isabelle's system of axiomatic type classes for expressing structures and functors in a light-weight manner. Proofs are expressed in the Isar language for readable formal proof, while aiming at its "best-style" of representing formal reasoning.

Contents

1 Orders

```
theory Orders imports Main begin
```

1.1 Ordered structures

We define several classes of ordered structures over some type *'a* with relation ⊑ :: *'a* ⇒ *'a* ⇒ *bool*. For a *quasi-order* that relation is required to be reflexive and transitive, for a *partial order* it also has to be anti-symmetric, while for a *linear order* all elements are required to be related (in either direction).

```
class leq =
  fixes leq :: 'a ⇒ 'a ⇒ bool  (infixl [= 50)

notation (xsymbols)
  leq  (infixl ⊑ 50)

class quasi-order = leq +
  assumes leq-refl [intro?]: x ⊑ x
  assumes leq-trans [trans]: x ⊑ y ⟹ y ⊑ z ⟹ x ⊑ z

class partial-order = quasi-order +
  assumes leq-antisym [trans]: x ⊑ y ⟹ y ⊑ x ⟹ x = y

class linear-order = partial-order +
  assumes leq-linear: x ⊑ y ∨ y ⊑ x

lemma linear-order-cases:
    ((x::'a::linear-order) ⊑ y ⟹ C) ⟹ (y ⊑ x ⟹ C) ⟹ C
  by (insert leq-linear) blast
```

1.2 Duality

The *dual* of an ordered structure is an isomorphic copy of the underlying type, with the ⊑ relation defined as the inverse of the original one.

```
datatype 'a dual = dual 'a

primrec undual :: 'a dual ⇒ 'a where
  undual-dual: undual (dual x) = x

instantiation dual :: (leq) leq
begin

definition
  leq-dual-def: x' ⊑ y' ≡ undual y' ⊑ undual x'

instance ..

end
```

```
lemma undual-leq [iff?]: (undual x′ ⊑ undual y′) = (y′ ⊑ x′)
  by (simp add: leq-dual-def)

lemma dual-leq [iff?]: (dual x ⊑ dual y) = (y ⊑ x)
  by (simp add: leq-dual-def)
```

Functions *dual* and *undual* are inverse to each other; this entails the following fundamental properties.

```
lemma dual-undual [simp]: dual (undual x′) = x′
  by (cases x′) simp

lemma undual-dual-id [simp]: undual o dual = id
  by (rule ext) simp

lemma dual-undual-id [simp]: dual o undual = id
  by (rule ext) simp
```

Since *dual* (and *undual*) are both injective and surjective, the basic logical connectives (equality, quantification etc.) are transferred as follows.

```
lemma undual-equality [iff?]: (undual x′ = undual y′) = (x′ = y′)
  by (cases x′, cases y′) simp

lemma dual-equality [iff?]: (dual x = dual y) = (x = y)
  by simp

lemma dual-ball [iff?]: (∀ x ∈ A. P (dual x)) = (∀ x′ ∈ dual ‘ A. P x′)
proof
  assume a: ∀ x ∈ A. P (dual x)
  show ∀ x′ ∈ dual ‘ A. P x′
  proof
    fix x′ assume x′: x′ ∈ dual ‘ A
    have undual x′ ∈ A
    proof −
      from x′ have undual x′ ∈ undual ‘ dual ‘ A by simp
      thus undual x′ ∈ A by (simp add: image-compose [symmetric])
    qed
    with a have P (dual (undual x′)) ..
    also have … = x′ by simp
    finally show P x′ .
  qed
next
  assume a: ∀ x′ ∈ dual ‘ A. P x′
  show ∀ x ∈ A. P (dual x)
  proof
    fix x assume x ∈ A
    hence dual x ∈ dual ‘ A by simp
    with a show P (dual x) ..
```

  qed
qed

lemma *range-dual* [*simp*]: *surj dual*
proof −
  have $\bigwedge x'.\ \mathit{dual}\ (\mathit{undual}\ x') = x'$ **by** *simp*
  thus *surj dual* **by** (*rule surjI*)
qed

lemma *dual-all* [*iff?*]: $(\forall\, x.\ P\ (\mathit{dual}\ x)) = (\forall\, x'.\ P\ x')$
proof −
  have $(\forall\, x \in \mathit{UNIV}.\ P\ (\mathit{dual}\ x)) = (\forall\, x' \in \mathit{dual}\ \text{`}\ \mathit{UNIV}.\ P\ x')$
    by (*rule dual-ball*)
  thus *?thesis* **by** *simp*
qed

lemma *dual-ex*: $(\exists\, x.\ P\ (\mathit{dual}\ x)) = (\exists\, x'.\ P\ x')$
proof −
  have $(\forall\, x.\ \neg\ P\ (\mathit{dual}\ x)) = (\forall\, x'.\ \neg\ P\ x')$
    by (*rule dual-all*)
  thus *?thesis* **by** *blast*
qed

lemma *dual-Collect*: $\{\mathit{dual}\ x \mid\ x.\ P\ (\mathit{dual}\ x)\} = \{x'.\ P\ x'\}$
proof −
  have $\{\mathit{dual}\ x \mid\ x.\ P\ (\mathit{dual}\ x)\} = \{x'.\ \exists\, x''.\ x' = x'' \wedge P\ x''\}$
    by (*simp only*: *dual-ex* [*symmetric*])
  thus *?thesis* **by** *blast*
qed

1.3 Transforming orders

1.3.1 Duals

The classes of quasi, partial, and linear orders are all closed under formation of dual structures.

instance *dual* :: (*quasi-order*) *quasi-order*
proof
  fix $x'\ y'\ z'$ :: *'a::quasi-order dual*
  have $\mathit{undual}\ x' \sqsubseteq \mathit{undual}\ x'$ **..** **thus** $x' \sqsubseteq x'$ **..**
  assume $y' \sqsubseteq z'$ **hence** $\mathit{undual}\ z' \sqsubseteq \mathit{undual}\ y'$ **..**
  also assume $x' \sqsubseteq y'$ **hence** $\mathit{undual}\ y' \sqsubseteq \mathit{undual}\ x'$ **..**
  finally show $x' \sqsubseteq z'$ **..**
qed

instance *dual* :: (*partial-order*) *partial-order*
proof
  fix $x'\ y'$ :: *'a::partial-order dual*
  assume $y' \sqsubseteq x'$ **hence** $\mathit{undual}\ x' \sqsubseteq \mathit{undual}\ y'$ **..**

```
  also assume x′ ⊑ y′ hence undual y′ ⊑ undual x′ ..
  finally show x′ = y′ ..
qed

instance dual :: (linear-order) linear-order
proof
  fix x′ y′ :: ′a::linear-order dual
  show x′ ⊑ y′ ∨ y′ ⊑ x′
  proof (rule linear-order-cases)
    assume undual y′ ⊑ undual x′
    hence x′ ⊑ y′ .. thus ?thesis ..
  next
    assume undual x′ ⊑ undual y′
    hence y′ ⊑ x′ .. thus ?thesis ..
  qed
qed
```

1.3.2 Binary products

The classes of quasi and partial orders are closed under binary products. Note that the direct product of linear orders need *not* be linear in general.

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

definition
  leq-prod-def: p ⊑ q ≡ fst p ⊑ fst q ∧ snd p ⊑ snd q

instance ..

end

lemma leq-prodI [intro?]:
    fst p ⊑ fst q ⟹ snd p ⊑ snd q ⟹ p ⊑ q
  by (unfold leq-prod-def) blast

lemma leq-prodE [elim?]:
    p ⊑ q ⟹ (fst p ⊑ fst q ⟹ snd p ⊑ snd q ⟹ C) ⟹ C
  by (unfold leq-prod-def) blast

instance prod :: (quasi-order, quasi-order) quasi-order
proof
  fix p q r :: ′a::quasi-order × ′b::quasi-order
  show p ⊑ p
  proof
    show fst p ⊑ fst p ..
    show snd p ⊑ snd p ..
  qed
  assume pq: p ⊑ q and qr: q ⊑ r
  show p ⊑ r
```

```
  proof
    from pq have fst p ⊑ fst q ..
    also from qr have … ⊑ fst r ..
    finally show fst p ⊑ fst r .
    from pq have snd p ⊑ snd q ..
    also from qr have … ⊑ snd r ..
    finally show snd p ⊑ snd r .
  qed
qed

instance prod :: (partial-order, partial-order) partial-order
proof
  fix p q :: 'a::partial-order × 'b::partial-order
  assume pq: p ⊑ q and qp: q ⊑ p
  show p = q
  proof
    from pq have fst p ⊑ fst q ..
    also from qp have … ⊑ fst p ..
    finally show fst p = fst q .
    from pq have snd p ⊑ snd q ..
    also from qp have … ⊑ snd p ..
    finally show snd p = snd q .
  qed
qed
```

1.3.3 General products

The classes of quasi and partial orders are closed under general products (function spaces). Note that the direct product of linear orders need *not* be linear in general.

```
instantiation fun :: (type, leq) leq
begin

definition
  leq-fun-def: f ⊑ g ≡ ∀ x. f x ⊑ g x

instance ..

end

lemma leq-funI [intro?]: (⋀x. f x ⊑ g x) ⟹ f ⊑ g
  by (unfold leq-fun-def) blast

lemma leq-funD [dest?]: f ⊑ g ⟹ f x ⊑ g x
  by (unfold leq-fun-def) blast

instance fun :: (type, quasi-order) quasi-order
proof
```

```
  fix f g h :: 'a ⇒ 'b::quasi-order
  show f ⊑ f
  proof
    fix x show f x ⊑ f x ..
  qed
  assume fg: f ⊑ g and gh: g ⊑ h
  show f ⊑ h
  proof
    fix x from fg have f x ⊑ g x ..
    also from gh have ... ⊑ h x ..
    finally show f x ⊑ h x .
  qed
qed
```

```
instance fun :: (type, partial-order) partial-order
proof
  fix f g :: 'a ⇒ 'b::partial-order
  assume fg: f ⊑ g and gf: g ⊑ f
  show f = g
  proof
    fix x from fg have f x ⊑ g x ..
    also from gf have ... ⊑ f x ..
    finally show f x = g x .
  qed
qed
```

```
end
```

2 Bounds

```
theory Bounds imports Orders begin
```

```
hide-const (open) inf sup
```

2.1 Infimum and supremum

Given a partial order, we define infimum (greatest lower bound) and supremum (least upper bound) wrt. ⊑ for two and for any number of elements.

```
definition
  is-inf :: 'a::partial-order ⇒ 'a ⇒ 'a ⇒ bool where
  is-inf x y inf = (inf ⊑ x ∧ inf ⊑ y ∧ (∀ z. z ⊑ x ∧ z ⊑ y ⟶ z ⊑ inf))
```

```
definition
  is-sup :: 'a::partial-order ⇒ 'a ⇒ 'a ⇒ bool where
  is-sup x y sup = (x ⊑ sup ∧ y ⊑ sup ∧ (∀ z. x ⊑ z ∧ y ⊑ z ⟶ sup ⊑ z))
```

```
definition
```

$\mathit{is\text{-}Inf} :: {'a}{::}\mathit{partial\text{-}order\ set} \Rightarrow {'a} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}Inf}\ A\ \mathit{inf} = ((\forall x \in A.\ \mathit{inf} \sqsubseteq x) \wedge (\forall z.\ (\forall x \in A.\ z \sqsubseteq x) \longrightarrow z \sqsubseteq \mathit{inf}))$

definition
$\mathit{is\text{-}Sup} :: {'a}{::}\mathit{partial\text{-}order\ set} \Rightarrow {'a} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}Sup}\ A\ \mathit{sup} = ((\forall x \in A.\ x \sqsubseteq \mathit{sup}) \wedge (\forall z.\ (\forall x \in A.\ x \sqsubseteq z) \longrightarrow \mathit{sup} \sqsubseteq z))$

These definitions entail the following basic properties of boundary elements.

lemma $\mathit{is\text{-}infI}$ $[\mathit{intro?}]$: $\mathit{inf} \sqsubseteq x \Longrightarrow \mathit{inf} \sqsubseteq y \Longrightarrow$
$(\bigwedge z.\ z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq \mathit{inf}) \Longrightarrow \mathit{is\text{-}inf}\ x\ y\ \mathit{inf}$
by $(\mathit{unfold\ is\text{-}inf\text{-}def})\ \mathit{blast}$

lemma $\mathit{is\text{-}inf\text{-}greatest}$ $[\mathit{elim?}]$:
$\mathit{is\text{-}inf}\ x\ y\ \mathit{inf} \Longrightarrow z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq \mathit{inf}$
by $(\mathit{unfold\ is\text{-}inf\text{-}def})\ \mathit{blast}$

lemma $\mathit{is\text{-}inf\text{-}lower}$ $[\mathit{elim?}]$:
$\mathit{is\text{-}inf}\ x\ y\ \mathit{inf} \Longrightarrow (\mathit{inf} \sqsubseteq x \Longrightarrow \mathit{inf} \sqsubseteq y \Longrightarrow C) \Longrightarrow C$
by $(\mathit{unfold\ is\text{-}inf\text{-}def})\ \mathit{blast}$

lemma $\mathit{is\text{-}supI}$ $[\mathit{intro?}]$: $x \sqsubseteq \mathit{sup} \Longrightarrow y \sqsubseteq \mathit{sup} \Longrightarrow$
$(\bigwedge z.\ x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow \mathit{sup} \sqsubseteq z) \Longrightarrow \mathit{is\text{-}sup}\ x\ y\ \mathit{sup}$
by $(\mathit{unfold\ is\text{-}sup\text{-}def})\ \mathit{blast}$

lemma $\mathit{is\text{-}sup\text{-}least}$ $[\mathit{elim?}]$:
$\mathit{is\text{-}sup}\ x\ y\ \mathit{sup} \Longrightarrow x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow \mathit{sup} \sqsubseteq z$
by $(\mathit{unfold\ is\text{-}sup\text{-}def})\ \mathit{blast}$

lemma $\mathit{is\text{-}sup\text{-}upper}$ $[\mathit{elim?}]$:
$\mathit{is\text{-}sup}\ x\ y\ \mathit{sup} \Longrightarrow (x \sqsubseteq \mathit{sup} \Longrightarrow y \sqsubseteq \mathit{sup} \Longrightarrow C) \Longrightarrow C$
by $(\mathit{unfold\ is\text{-}sup\text{-}def})\ \mathit{blast}$

lemma $\mathit{is\text{-}InfI}$ $[\mathit{intro?}]$: $(\bigwedge x.\ x \in A \Longrightarrow \mathit{inf} \sqsubseteq x) \Longrightarrow$
$(\bigwedge z.\ (\forall x \in A.\ z \sqsubseteq x) \Longrightarrow z \sqsubseteq \mathit{inf}) \Longrightarrow \mathit{is\text{-}Inf}\ A\ \mathit{inf}$
by $(\mathit{unfold\ is\text{-}Inf\text{-}def})\ \mathit{blast}$

lemma $\mathit{is\text{-}Inf\text{-}greatest}$ $[\mathit{elim?}]$:
$\mathit{is\text{-}Inf}\ A\ \mathit{inf} \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x) \Longrightarrow z \sqsubseteq \mathit{inf}$
by $(\mathit{unfold\ is\text{-}Inf\text{-}def})\ \mathit{blast}$

lemma $\mathit{is\text{-}Inf\text{-}lower}$ $[\mathit{dest?}]$:
$\mathit{is\text{-}Inf}\ A\ \mathit{inf} \Longrightarrow x \in A \Longrightarrow \mathit{inf} \sqsubseteq x$
by $(\mathit{unfold\ is\text{-}Inf\text{-}def})\ \mathit{blast}$

lemma $\mathit{is\text{-}SupI}$ $[\mathit{intro?}]$: $(\bigwedge x.\ x \in A \Longrightarrow x \sqsubseteq \mathit{sup}) \Longrightarrow$
$(\bigwedge z.\ (\forall x \in A.\ x \sqsubseteq z) \Longrightarrow \mathit{sup} \sqsubseteq z) \Longrightarrow \mathit{is\text{-}Sup}\ A\ \mathit{sup}$

```
  by (unfold is-Sup-def) blast

lemma is-Sup-least [elim?]:
    is-Sup A sup ⟹ (⋀x. x ∈ A ⟹ x ⊑ z) ⟹ sup ⊑ z
  by (unfold is-Sup-def) blast

lemma is-Sup-upper [dest?]:
    is-Sup A sup ⟹ x ∈ A ⟹ x ⊑ sup
  by (unfold is-Sup-def) blast
```

2.2 Duality

Infimum and supremum are dual to each other.

```
theorem dual-inf [iff?]:
    is-inf (dual x) (dual y) (dual sup) = is-sup x y sup
  by (simp add: is-inf-def is-sup-def dual-all [symmetric] dual-leq)

theorem dual-sup [iff?]:
    is-sup (dual x) (dual y) (dual inf) = is-inf x y inf
  by (simp add: is-inf-def is-sup-def dual-all [symmetric] dual-leq)

theorem dual-Inf [iff?]:
    is-Inf (dual ` A) (dual sup) = is-Sup A sup
  by (simp add: is-Inf-def is-Sup-def dual-all [symmetric] dual-leq)

theorem dual-Sup [iff?]:
    is-Sup (dual ` A) (dual inf) = is-Inf A inf
  by (simp add: is-Inf-def is-Sup-def dual-all [symmetric] dual-leq)
```

2.3 Uniqueness

Infima and suprema on partial orders are unique; this is mainly due to antisymmetry of the underlying relation.

```
theorem is-inf-uniq: is-inf x y inf ⟹ is-inf x y inf' ⟹ inf = inf'
proof -
  assume inf: is-inf x y inf
  assume inf': is-inf x y inf'
  show ?thesis
  proof (rule leq-antisym)
    from inf' show inf ⊑ inf'
    proof (rule is-inf-greatest)
      from inf show inf ⊑ x ..
      from inf show inf ⊑ y ..
    qed
    from inf show inf' ⊑ inf
    proof (rule is-inf-greatest)
      from inf' show inf' ⊑ x ..
      from inf' show inf' ⊑ y ..
```

```
    qed
  qed
qed

theorem is-sup-uniq: is-sup x y sup ⟹ is-sup x y sup′ ⟹ sup = sup′
proof −
  assume sup: is-sup x y sup and sup′: is-sup x y sup′
  have dual sup = dual sup′
  proof (rule is-inf-uniq)
    from sup show is-inf (dual x) (dual y) (dual sup) ..
    from sup′ show is-inf (dual x) (dual y) (dual sup′) ..
  qed
  then show sup = sup′ ..
qed

theorem is-Inf-uniq: is-Inf A inf ⟹ is-Inf A inf′ ⟹ inf = inf′
proof −
  assume inf: is-Inf A inf
  assume inf′: is-Inf A inf′
  show ?thesis
  proof (rule leq-antisym)
    from inf′ show inf ⊑ inf′
    proof (rule is-Inf-greatest)
      fix x assume x ∈ A
      with inf show inf ⊑ x ..
    qed
    from inf show inf′ ⊑ inf
    proof (rule is-Inf-greatest)
      fix x assume x ∈ A
      with inf′ show inf′ ⊑ x ..
    qed
  qed
qed

theorem is-Sup-uniq: is-Sup A sup ⟹ is-Sup A sup′ ⟹ sup = sup′
proof −
  assume sup: is-Sup A sup and sup′: is-Sup A sup′
  have dual sup = dual sup′
  proof (rule is-Inf-uniq)
    from sup show is-Inf (dual ‘ A) (dual sup) ..
    from sup′ show is-Inf (dual ‘ A) (dual sup′) ..
  qed
  then show sup = sup′ ..
qed
```

2.4 Related elements

The binary bound of related elements is either one of the argument.

```
theorem is-inf-related [elim?]: x ⊑ y ⟹ is-inf x y x
```

```
proof –
  assume x ⊑ y
  show ?thesis
  proof
    show x ⊑ x ..
    show x ⊑ y by fact
    fix z assume z ⊑ x and z ⊑ y show z ⊑ x by fact
  qed
qed

theorem is-sup-related [elim?]: x ⊑ y ⟹ is-sup x y y
proof –
  assume x ⊑ y
  show ?thesis
  proof
    show x ⊑ y by fact
    show y ⊑ y ..
    fix z assume x ⊑ z and y ⊑ z
    show y ⊑ z by fact
  qed
qed
```

2.5 General versus binary bounds

General bounds of two-element sets coincide with binary bounds.

```
theorem is-Inf-binary: is-Inf {x, y} inf = is-inf x y inf
proof –
  let ?A = {x, y}
  show ?thesis
  proof
    assume is-Inf: is-Inf ?A inf
    show is-inf x y inf
    proof
      have x ∈ ?A by simp
      with is-Inf show inf ⊑ x ..
      have y ∈ ?A by simp
      with is-Inf show inf ⊑ y ..
      fix z assume zx: z ⊑ x and zy: z ⊑ y
      from is-Inf show z ⊑ inf
      proof (rule is-Inf-greatest)
        fix a assume a ∈ ?A
        then have a = x ∨ a = y by blast
        then show z ⊑ a
        proof
          assume a = x
          with zx show ?thesis by simp
        next
          assume a = y
          with zy show ?thesis by simp
```

```
        qed
      qed
    qed
  next
    assume is-inf: is-inf x y inf
    show is-Inf {x, y} inf
    proof
      fix a assume a ∈ ?A
      then have a = x ∨ a = y by blast
      then show inf ⊑ a
      proof
        assume a = x
        also from is-inf have inf ⊑ x ..
        finally show ?thesis .
      next
        assume a = y
        also from is-inf have inf ⊑ y ..
        finally show ?thesis .
      qed
    next
      fix z assume z: ∀ a ∈ ?A. z ⊑ a
      from is-inf show z ⊑ inf
      proof (rule is-inf-greatest)
        from z show z ⊑ x by blast
        from z show z ⊑ y by blast
      qed
    qed
  qed
qed
```

```
theorem is-Sup-binary: is-Sup {x, y} sup = is-sup x y sup
proof −
  have is-Sup {x, y} sup = is-Inf (dual ‘ {x, y}) (dual sup)
    by (simp only: dual-Inf)
  also have dual ‘ {x, y} = {dual x, dual y}
    by simp
  also have is-Inf … (dual sup) = is-inf (dual x) (dual y) (dual sup)
    by (rule is-Inf-binary)
  also have … = is-sup x y sup
    by (simp only: dual-inf)
  finally show ?thesis .
qed
```

2.6 Connecting general bounds

Either kind of general bounds is sufficient to express the other. The least upper bound (supremum) is the same as the the greatest lower bound of the set of all upper bounds; the dual statements holds as well; the dual statement holds as well.

```
theorem Inf-Sup: is-Inf {b. ∀ a ∈ A. a ⊑ b} sup ⟹ is-Sup A sup
proof −
  let ?B = {b. ∀ a ∈ A. a ⊑ b}
  assume is-Inf: is-Inf ?B sup
  show is-Sup A sup
  proof
    fix x assume x: x ∈ A
    from is-Inf show x ⊑ sup
    proof (rule is-Inf-greatest)
      fix y assume y ∈ ?B
      then have ∀ a ∈ A. a ⊑ y ..
      from this x show x ⊑ y ..
    qed
  next
    fix z assume ∀ x ∈ A. x ⊑ z
    then have z ∈ ?B ..
    with is-Inf show sup ⊑ z ..
  qed
qed

theorem Sup-Inf: is-Sup {b. ∀ a ∈ A. b ⊑ a} inf ⟹ is-Inf A inf
proof −
  assume is-Sup {b. ∀ a ∈ A. b ⊑ a} inf
  then have is-Inf (dual ‘ {b. ∀ a ∈ A. dual a ⊑ dual b}) (dual inf)
    by (simp only: dual-Inf dual-leq)
  also have dual ‘ {b. ∀ a ∈ A. dual a ⊑ dual b} = {b′. ∀ a′ ∈ dual ‘ A. a′ ⊑ b′}
    by (auto iff: dual-ball dual-Collect simp add: image-Collect)
  finally have is-Inf ... (dual inf) .
  then have is-Sup (dual ‘ A) (dual inf)
    by (rule Inf-Sup)
  then show ?thesis ..
qed

end
```

3 Lattices

```
theory Lattice imports Bounds begin
```

3.1 Lattice operations

A *lattice* is a partial order with infimum and supremum of any two elements (thus any *finite* number of elements have bounds as well).

```
class lattice =
  assumes ex-inf: ∃ inf. is-inf x y inf
  assumes ex-sup: ∃ sup. is-sup x y sup
```

The ⊓ (meet) and ⊔ (join) operations select such infimum and supremum elements.

```
definition
  meet :: 'a::lattice ⇒ 'a ⇒ 'a  (infixl && 70) where
  x && y = (THE inf. is-inf x y inf)
definition
  join :: 'a::lattice ⇒ 'a ⇒ 'a  (infixl || 65) where
  x || y = (THE sup. is-sup x y sup)

notation (xsymbols)
  meet  (infixl ⊓ 70) and
  join  (infixl ⊔ 65)
```

Due to unique existence of bounds, the lattice operations may be exhibited as follows.

```
lemma meet-equality [elim?]: is-inf x y inf ⟹ x ⊓ y = inf
proof (unfold meet-def)
  assume is-inf x y inf
  then show (THE inf. is-inf x y inf) = inf
    by (rule the-equality) (rule is-inf-uniq [OF - ‹is-inf x y inf›])
qed

lemma meetI [intro?]:
    inf ⊑ x ⟹ inf ⊑ y ⟹ (⋀z. z ⊑ x ⟹ z ⊑ y ⟹ z ⊑ inf) ⟹ x ⊓ y = inf
  by (rule meet-equality, rule is-infI) blast+

lemma join-equality [elim?]: is-sup x y sup ⟹ x ⊔ y = sup
proof (unfold join-def)
  assume is-sup x y sup
  then show (THE sup. is-sup x y sup) = sup
    by (rule the-equality) (rule is-sup-uniq [OF - ‹is-sup x y sup›])
qed

lemma joinI [intro?]: x ⊑ sup ⟹ y ⊑ sup ⟹
    (⋀z. x ⊑ z ⟹ y ⊑ z ⟹ sup ⊑ z) ⟹ x ⊔ y = sup
  by (rule join-equality, rule is-supI) blast+
```

The ⊓ and ⊔ operations indeed determine bounds on a lattice structure.

```
lemma is-inf-meet [intro?]: is-inf x y (x ⊓ y)
proof (unfold meet-def)
  from ex-inf obtain inf where is-inf x y inf ..
  then show is-inf x y (THE inf. is-inf x y inf)
    by (rule theI) (rule is-inf-uniq [OF - ‹is-inf x y inf›])
qed

lemma meet-greatest [intro?]: z ⊑ x ⟹ z ⊑ y ⟹ z ⊑ x ⊓ y
  by (rule is-inf-greatest) (rule is-inf-meet)
```

```
lemma meet-lower1 [intro?]: x ⊓ y ⊑ x
  by (rule is-inf-lower) (rule is-inf-meet)

lemma meet-lower2 [intro?]: x ⊓ y ⊑ y
  by (rule is-inf-lower) (rule is-inf-meet)


lemma is-sup-join [intro?]: is-sup x y (x ⊔ y)
proof (unfold join-def)
  from ex-sup obtain sup where is-sup x y sup ..
  then show is-sup x y (THE sup. is-sup x y sup)
    by (rule theI) (rule is-sup-uniq [OF - ‹is-sup x y sup›])
qed

lemma join-least [intro?]: x ⊑ z ⟹ y ⊑ z ⟹ x ⊔ y ⊑ z
  by (rule is-sup-least) (rule is-sup-join)

lemma join-upper1 [intro?]: x ⊑ x ⊔ y
  by (rule is-sup-upper) (rule is-sup-join)

lemma join-upper2 [intro?]: y ⊑ x ⊔ y
  by (rule is-sup-upper) (rule is-sup-join)
```

3.2 Duality

The class of lattices is closed under formation of dual structures. This means that for any theorem of lattice theory, the dualized statement holds as well; this important fact simplifies many proofs of lattice theory.

```
instance dual :: (lattice) lattice
proof
  fix x′ y′ :: ′a::lattice dual
  show ∃ inf′. is-inf x′ y′ inf′
  proof –
    have ∃ sup. is-sup (undual x′) (undual y′) sup by (rule ex-sup)
    then have ∃ sup. is-inf (dual (undual x′)) (dual (undual y′)) (dual sup)
      by (simp only: dual-inf)
    then show ?thesis by (simp add: dual-ex [symmetric])
  qed
  show ∃ sup′. is-sup x′ y′ sup′
  proof –
    have ∃ inf. is-inf (undual x′) (undual y′) inf by (rule ex-inf)
    then have ∃ inf. is-sup (dual (undual x′)) (dual (undual y′)) (dual inf)
      by (simp only: dual-sup)
    then show ?thesis by (simp add: dual-ex [symmetric])
  qed
qed
```

Apparently, the ⊓ and ⊔ operations are dual to each other.

```
theorem dual-meet [intro?]: $dual\ (x \sqcap y) = dual\ x \sqcup dual\ y$
proof -
  from is-inf-meet have $is\text{-}sup\ (dual\ x)\ (dual\ y)\ (dual\ (x \sqcap y))$ ..
  then have $dual\ x \sqcup dual\ y = dual\ (x \sqcap y)$ ..
  then show ?thesis ..
qed

theorem dual-join [intro?]: $dual\ (x \sqcup y) = dual\ x \sqcap dual\ y$
proof -
  from is-sup-join have $is\text{-}inf\ (dual\ x)\ (dual\ y)\ (dual\ (x \sqcup y))$ ..
  then have $dual\ x \sqcap dual\ y = dual\ (x \sqcup y)$ ..
  then show ?thesis ..
qed
```

3.3 Algebraic properties

The $\sqcap$ and $\sqcup$ operations have the following characteristic algebraic properties: associative (A), commutative (C), and absorptive (AB).

```
theorem meet-assoc: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
proof
  show $x \sqcap (y \sqcap z) \sqsubseteq x \sqcap y$
  proof
    show $x \sqcap (y \sqcap z) \sqsubseteq x$ ..
    show $x \sqcap (y \sqcap z) \sqsubseteq y$
    proof -
      have $x \sqcap (y \sqcap z) \sqsubseteq y \sqcap z$ ..
      also have $\ldots \sqsubseteq y$ ..
      finally show ?thesis .
    qed
  qed
  show $x \sqcap (y \sqcap z) \sqsubseteq z$
  proof -
    have $x \sqcap (y \sqcap z) \sqsubseteq y \sqcap z$ ..
    also have $\ldots \sqsubseteq z$ ..
    finally show ?thesis .
  qed
  fix $w$ assume $w \sqsubseteq x \sqcap y$ and $w \sqsubseteq z$
  show $w \sqsubseteq x \sqcap (y \sqcap z)$
  proof
    show $w \sqsubseteq x$
    proof -
      have $w \sqsubseteq x \sqcap y$ by fact
      also have $\ldots \sqsubseteq x$ ..
      finally show ?thesis .
    qed
    show $w \sqsubseteq y \sqcap z$
    proof
      show $w \sqsubseteq y$
      proof -
```

```
        have w ⊑ x ⊓ y by fact
        also have ... ⊑ y ..
        finally show ?thesis .
      qed
      show w ⊑ z by fact
    qed
  qed
qed

theorem join-assoc: (x ⊔ y) ⊔ z = x ⊔ (y ⊔ z)
proof -
  have dual ((x ⊔ y) ⊔ z) = (dual x ⊓ dual y) ⊓ dual z
    by (simp only: dual-join)
  also have ... = dual x ⊓ (dual y ⊓ dual z)
    by (rule meet-assoc)
  also have ... = dual (x ⊔ (y ⊔ z))
    by (simp only: dual-join)
  finally show ?thesis ..
qed

theorem meet-commute: x ⊓ y = y ⊓ x
proof
  show y ⊓ x ⊑ x ..
  show y ⊓ x ⊑ y ..
  fix z assume z ⊑ y and z ⊑ x
  then show z ⊑ y ⊓ x ..
qed

theorem join-commute: x ⊔ y = y ⊔ x
proof -
  have dual (x ⊔ y) = dual x ⊓ dual y ..
  also have ... = dual y ⊓ dual x
    by (rule meet-commute)
  also have ... = dual (y ⊔ x)
    by (simp only: dual-join)
  finally show ?thesis ..
qed

theorem meet-join-absorb: x ⊓ (x ⊔ y) = x
proof
  show x ⊑ x ..
  show x ⊑ x ⊔ y ..
  fix z assume z ⊑ x and z ⊑ x ⊔ y
  show z ⊑ x by fact
qed

theorem join-meet-absorb: x ⊔ (x ⊓ y) = x
proof -
  have dual x ⊓ (dual x ⊔ dual y) = dual x
```

  by (*rule meet-join-absorb*)
 then have $\mathit{dual}\ (x \sqcup (x \sqcap y)) = \mathit{dual}\ x$
  by (*simp only*: *dual-meet dual-join*)
 then show *?thesis* **..**
qed

Some further algebraic properties hold as well. The property idempotent (I) is a basic algebraic consequence of (AB).

theorem *meet-idem*: $x \sqcap x = x$
proof –
 have $x \sqcap (x \sqcup (x \sqcap x)) = x$ **by** (*rule meet-join-absorb*)
 also have $x \sqcup (x \sqcap x) = x$ **by** (*rule join-meet-absorb*)
 finally show *?thesis* **.**
qed

theorem *join-idem*: $x \sqcup x = x$
proof –
 have $\mathit{dual}\ x \sqcap \mathit{dual}\ x = \mathit{dual}\ x$
  by (*rule meet-idem*)
 then have $\mathit{dual}\ (x \sqcup x) = \mathit{dual}\ x$
  by (*simp only*: *dual-join*)
 then show *?thesis* **..**
qed

Meet and join are trivial for related elements.

theorem *meet-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow x \sqcap y = x$
proof
 assume $x \sqsubseteq y$
 show $x \sqsubseteq x$ **..**
 show $x \sqsubseteq y$ **by** *fact*
 fix z **assume** $z \sqsubseteq x$ **and** $z \sqsubseteq y$
 show $z \sqsubseteq x$ **by** *fact*
qed

theorem *join-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow x \sqcup y = y$
proof –
 assume $x \sqsubseteq y$ **then have** $\mathit{dual}\ y \sqsubseteq \mathit{dual}\ x$ **..**
 then have $\mathit{dual}\ y \sqcap \mathit{dual}\ x = \mathit{dual}\ y$ **by** (*rule meet-related*)
 also have $\mathit{dual}\ y \sqcap \mathit{dual}\ x = \mathit{dual}\ (y \sqcup x)$ **by** (*simp only*: *dual-join*)
 also have $y \sqcup x = x \sqcup y$ **by** (*rule join-commute*)
 finally show *?thesis* **..**
qed

3.4 Order versus algebraic structure

The $\sqcap$ and $\sqcup$ operations are connected with the underlying $\sqsubseteq$ relation in a canonical manner.

theorem *meet-connection*: $(x \sqsubseteq y) = (x \sqcap y = x)$

proof
  assume $x \sqsubseteq y$
  then have *is-inf* $x\ y\ x$ **..**
  then show $x \sqcap y = x$ **..**
next
  have $x \sqcap y \sqsubseteq y$ **..**
  also assume $x \sqcap y = x$
  finally show $x \sqsubseteq y$ **.**
qed

theorem *join-connection*: $(x \sqsubseteq y) = (x \sqcup y = y)$
proof
  assume $x \sqsubseteq y$
  then have *is-sup* $x\ y\ y$ **..**
  then show $x \sqcup y = y$ **..**
next
  have $x \sqsubseteq x \sqcup y$ **..**
  also assume $x \sqcup y = y$
  finally show $x \sqsubseteq y$ **.**
qed

The most fundamental result of the meta-theory of lattices is as follows (we do not prove it here).

Given a structure with binary operations $\sqcap$ and $\sqcup$ such that (A), (C), and (AB) hold (cf. §3.3). This structure represents a lattice, if the relation $x \sqsubseteq y$ is defined as $x \sqcap y = x$ (alternatively as $x \sqcup y = y$). Furthermore, infimum and supremum with respect to this ordering coincide with the original $\sqcap$ and $\sqcup$ operations.

3.5 Example instances

3.5.1 Linear orders

Linear orders with *minimum* and *maximum* operations are a (degenerate) example of lattice structures.

definition
  minimum :: *'a::linear-order* $\Rightarrow$ *'a* $\Rightarrow$ *'a* **where**
  minimum $x\ y = ($*if* $x \sqsubseteq y$ *then* x *else* $y)$
definition
  maximum :: *'a::linear-order* $\Rightarrow$ *'a* $\Rightarrow$ *'a* **where**
  maximum $x\ y = ($*if* $x \sqsubseteq y$ *then* y *else* $x)$

lemma *is-inf-minimum*: *is-inf* $x\ y$ (*minimum* $x\ y$)
proof
  let *?min* = *minimum* $x\ y$
  from *leq-linear* **show** *?min* $\sqsubseteq x$ **by** (*auto simp add*: *minimum-def*)
  from *leq-linear* **show** *?min* $\sqsubseteq y$ **by** (*auto simp add*: *minimum-def*)

```
  fix z assume z ⊑ x and z ⊑ y
  with leq-linear show z ⊑ ?min by (auto simp add: minimum-def)
qed

lemma is-sup-maximum: is-sup x y (maximum x y)
proof
  let ?max = maximum x y
  from leq-linear show x ⊑ ?max by (auto simp add: maximum-def)
  from leq-linear show y ⊑ ?max by (auto simp add: maximum-def)
  fix z assume x ⊑ z and y ⊑ z
  with leq-linear show ?max ⊑ z by (auto simp add: maximum-def)
qed

instance linear-order ⊆ lattice
proof
  fix x y :: 'a::linear-order
  from is-inf-minimum show ∃ inf. is-inf x y inf ..
  from is-sup-maximum show ∃ sup. is-sup x y sup ..
qed
```

The lattice operations on linear orders indeed coincide with *minimum* and *maximum*.

```
theorem meet-mimimum: x ⊓ y = minimum x y
  by (rule meet-equality) (rule is-inf-minimum)

theorem meet-maximum: x ⊔ y = maximum x y
  by (rule join-equality) (rule is-sup-maximum)
```

3.5.2 Binary products

The class of lattices is closed under direct binary products (cf. §1.3.2).

```
lemma is-inf-prod: is-inf p q (fst p ⊓ fst q, snd p ⊓ snd q)
proof
  show (fst p ⊓ fst q, snd p ⊓ snd q) ⊑ p
  proof –
    have fst p ⊓ fst q ⊑ fst p ..
    moreover have snd p ⊓ snd q ⊑ snd p ..
    ultimately show ?thesis by (simp add: leq-prod-def)
  qed
  show (fst p ⊓ fst q, snd p ⊓ snd q) ⊑ q
  proof –
    have fst p ⊓ fst q ⊑ fst q ..
    moreover have snd p ⊓ snd q ⊑ snd q ..
    ultimately show ?thesis by (simp add: leq-prod-def)
  qed
  fix r assume rp: r ⊑ p and rq: r ⊑ q
  show r ⊑ (fst p ⊓ fst q, snd p ⊓ snd q)
  proof –
```

```
    have fst r ⊑ fst p ⊓ fst q
    proof
      from rp show fst r ⊑ fst p by (simp add: leq-prod-def)
      from rq show fst r ⊑ fst q by (simp add: leq-prod-def)
    qed
    moreover have snd r ⊑ snd p ⊓ snd q
    proof
      from rp show snd r ⊑ snd p by (simp add: leq-prod-def)
      from rq show snd r ⊑ snd q by (simp add: leq-prod-def)
    qed
    ultimately show ?thesis by (simp add: leq-prod-def)
  qed
qed

lemma is-sup-prod: is-sup p q (fst p ⊔ fst q, snd p ⊔ snd q)
proof
  show p ⊑ (fst p ⊔ fst q, snd p ⊔ snd q)
  proof −
    have fst p ⊑ fst p ⊔ fst q ..
    moreover have snd p ⊑ snd p ⊔ snd q ..
    ultimately show ?thesis by (simp add: leq-prod-def)
  qed
  show q ⊑ (fst p ⊔ fst q, snd p ⊔ snd q)
  proof −
    have fst q ⊑ fst p ⊔ fst q ..
    moreover have snd q ⊑ snd p ⊔ snd q ..
    ultimately show ?thesis by (simp add: leq-prod-def)
  qed
  fix r assume pr: p ⊑ r and qr: q ⊑ r
  show (fst p ⊔ fst q, snd p ⊔ snd q) ⊑ r
  proof −
    have fst p ⊔ fst q ⊑ fst r
    proof
      from pr show fst p ⊑ fst r by (simp add: leq-prod-def)
      from qr show fst q ⊑ fst r by (simp add: leq-prod-def)
    qed
    moreover have snd p ⊔ snd q ⊑ snd r
    proof
      from pr show snd p ⊑ snd r by (simp add: leq-prod-def)
      from qr show snd q ⊑ snd r by (simp add: leq-prod-def)
    qed
    ultimately show ?thesis by (simp add: leq-prod-def)
  qed
qed

instance prod :: (lattice, lattice) lattice
proof
  fix p q :: 'a::lattice × 'b::lattice
  from is-inf-prod show ∃ inf. is-inf p q inf ..
```

```
  from is-sup-prod show ∃ sup. is-sup p q sup ..
qed
```

The lattice operations on a binary product structure indeed coincide with the products of the original ones.

```
theorem meet-prod: p ⊓ q = (fst p ⊓ fst q, snd p ⊓ snd q)
  by (rule meet-equality) (rule is-inf-prod)

theorem join-prod: p ⊔ q = (fst p ⊔ fst q, snd p ⊔ snd q)
  by (rule join-equality) (rule is-sup-prod)
```

3.5.3 General products

The class of lattices is closed under general products (function spaces) as well (cf. §1.3.3).

```
lemma is-inf-fun: is-inf f g (λx. f x ⊓ g x)
proof
  show (λx. f x ⊓ g x) ⊑ f
  proof
    fix x show f x ⊓ g x ⊑ f x ..
  qed
  show (λx. f x ⊓ g x) ⊑ g
  proof
    fix x show f x ⊓ g x ⊑ g x ..
  qed
  fix h assume hf: h ⊑ f and hg: h ⊑ g
  show h ⊑ (λx. f x ⊓ g x)
  proof
    fix x
    show h x ⊑ f x ⊓ g x
    proof
      from hf show h x ⊑ f x ..
      from hg show h x ⊑ g x ..
    qed
  qed
qed

lemma is-sup-fun: is-sup f g (λx. f x ⊔ g x)
proof
  show f ⊑ (λx. f x ⊔ g x)
  proof
    fix x show f x ⊑ f x ⊔ g x ..
  qed
  show g ⊑ (λx. f x ⊔ g x)
  proof
    fix x show g x ⊑ f x ⊔ g x ..
  qed
  fix h assume fh: f ⊑ h and gh: g ⊑ h
```

```
  show (λx. f x ⊔ g x) ⊑ h
  proof
    fix x
    show f x ⊔ g x ⊑ h x
    proof
      from fh show f x ⊑ h x ..
      from gh show g x ⊑ h x ..
    qed
  qed
qed

instance fun :: (type, lattice) lattice
proof
  fix f g :: 'a ⇒ 'b::lattice
  show ∃ inf. is-inf f g inf by rule (rule is-inf-fun)
  show ∃ sup. is-sup f g sup by rule (rule is-sup-fun)
qed
```

The lattice operations on a general product structure (function space) indeed emerge by point-wise lifting of the original ones.

```
theorem meet-fun: f ⊓ g = (λx. f x ⊓ g x)
  by (rule meet-equality) (rule is-inf-fun)

theorem join-fun: f ⊔ g = (λx. f x ⊔ g x)
  by (rule join-equality) (rule is-sup-fun)
```

3.6 Monotonicity and semi-morphisms

The lattice operations are monotone in both argument positions. In fact, monotonicity of the second position is trivial due to commutativity.

```
theorem meet-mono: x ⊑ z ⟹ y ⊑ w ⟹ x ⊓ y ⊑ z ⊓ w
proof -
  {
    fix a b c :: 'a::lattice
    assume a ⊑ c have a ⊓ b ⊑ c ⊓ b
    proof
      have a ⊓ b ⊑ a ..
      also have ... ⊑ c by fact
      finally show a ⊓ b ⊑ c .
      show a ⊓ b ⊑ b ..
    qed
  } note this [elim?]
  assume x ⊑ z then have x ⊓ y ⊑ z ⊓ y ..
  also have ... = y ⊓ z by (rule meet-commute)
  also assume y ⊑ w then have y ⊓ z ⊑ w ⊓ z ..
  also have ... = z ⊓ w by (rule meet-commute)
  finally show ?thesis .
qed
```

```
theorem join-mono: x ⊑ z ⟹ y ⊑ w ⟹ x ⊔ y ⊑ z ⊔ w
proof −
  assume x ⊑ z then have dual z ⊑ dual x ..
  moreover assume y ⊑ w then have dual w ⊑ dual y ..
  ultimately have dual z ⊓ dual w ⊑ dual x ⊓ dual y
    by (rule meet-mono)
  then have dual (z ⊔ w) ⊑ dual (x ⊔ y)
    by (simp only: dual-join)
  then show ?thesis ..
qed
```

A semi-morphisms is a function f that preserves the lattice operations in the following manner: $f\ (x \sqcap y) \sqsubseteq f\ x \sqcap f\ y$ and $f\ x \sqcup f\ y \sqsubseteq f\ (x \sqcup y)$, respectively. Any of these properties is equivalent with monotonicity.

```
theorem meet-semimorph:
  (⋀x y. f (x ⊓ y) ⊑ f x ⊓ f y) ≡ (⋀x y. x ⊑ y ⟹ f x ⊑ f y)
proof
  assume morph: ⋀x y. f (x ⊓ y) ⊑ f x ⊓ f y
  fix x y :: 'a::lattice
  assume x ⊑ y
  then have x ⊓ y = x ..
  then have x = x ⊓ y ..
  also have f ... ⊑ f x ⊓ f y by (rule morph)
  also have ... ⊑ f y ..
  finally show f x ⊑ f y .
next
  assume mono: ⋀x y. x ⊑ y ⟹ f x ⊑ f y
  show ⋀x y. f (x ⊓ y) ⊑ f x ⊓ f y
  proof −
    fix x y
    show f (x ⊓ y) ⊑ f x ⊓ f y
    proof
      have x ⊓ y ⊑ x .. then show f (x ⊓ y) ⊑ f x by (rule mono)
      have x ⊓ y ⊑ y .. then show f (x ⊓ y) ⊑ f y by (rule mono)
    qed
  qed
qed
```

```
lemma join-semimorph:
  (⋀x y. f x ⊔ f y ⊑ f (x ⊔ y)) ≡ (⋀x y. x ⊑ y ⟹ f x ⊑ f y)
proof
  assume morph: ⋀x y. f x ⊔ f y ⊑ f (x ⊔ y)
  fix x y :: 'a::lattice
  assume x ⊑ y then have x ⊔ y = y ..
  have f x ⊑ f x ⊔ f y ..
  also have ... ⊑ f (x ⊔ y) by (rule morph)
  also from ‹x ⊑ y› have x ⊔ y = y ..
  finally show f x ⊑ f y .
```

```
next
  assume mono: ⋀x y. x ⊑ y ⟹ f x ⊑ f y
  show ⋀x y. f x ⊔ f y ⊑ f (x ⊔ y)
  proof −
    fix x y
    show f x ⊔ f y ⊑ f (x ⊔ y)
    proof
      have x ⊑ x ⊔ y .. then show f x ⊑ f (x ⊔ y) by (rule mono)
      have y ⊑ x ⊔ y .. then show f y ⊑ f (x ⊔ y) by (rule mono)
    qed
  qed
qed

end
```

4 Complete lattices

```
theory CompleteLattice imports Lattice begin
```

4.1 Complete lattice operations

A *complete lattice* is a partial order with general (infinitary) infimum of any set of elements. General supremum exists as well, as a consequence of the connection of infinitary bounds (see §2.6).

```
class complete-lattice =
  assumes ex-Inf: ∃ inf. is-Inf A inf

theorem ex-Sup: ∃ sup::'a::complete-lattice. is-Sup A sup
proof −
  from ex-Inf obtain sup where is-Inf {b. ∀ a∈A. a ⊑ b} sup by blast
  then have is-Sup A sup by (rule Inf-Sup)
  then show ?thesis ..
qed
```

The general $\bigsqcap$ (meet) and $\bigsqcup$ (join) operations select such infimum and supremum elements.

```
definition
  Meet :: 'a::complete-lattice set ⇒ 'a where
  Meet A = (THE inf. is-Inf A inf)
definition
  Join :: 'a::complete-lattice set ⇒ 'a where
  Join A = (THE sup. is-Sup A sup)

notation (xsymbols)
  Meet (⨅- [90] 90) and
  Join (⨆- [90] 90)
```

Due to unique existence of bounds, the complete lattice operations may be exhibited as follows.

lemma *Meet-equality* [*elim?*]: $\mathit{is\text{-}Inf}\ A\ \mathit{inf} \Longrightarrow \bigsqcap A = \mathit{inf}$
proof (*unfold Meet-def*)
  assume *is-Inf A inf*
  then show $(\mathit{THE}\ \mathit{inf}.\ \mathit{is\text{-}Inf}\ A\ \mathit{inf}) = \mathit{inf}$
    by (*rule the-equality*) (*rule is-Inf-uniq* [*OF* - ‹*is-Inf A inf*›])
qed

lemma *MeetI* [*intro?*]:
  $(\bigwedge a.\ a \in A \Longrightarrow \mathit{inf} \sqsubseteq a) \Longrightarrow$
    $(\bigwedge b.\ \forall a \in A.\ b \sqsubseteq a \Longrightarrow b \sqsubseteq \mathit{inf}) \Longrightarrow$
    $\bigsqcap A = \mathit{inf}$
  by (*rule Meet-equality*, *rule is-InfI*) *blast+*

lemma *Join-equality* [*elim?*]: $\mathit{is\text{-}Sup}\ A\ \mathit{sup} \Longrightarrow \bigsqcup A = \mathit{sup}$
proof (*unfold Join-def*)
  assume *is-Sup A sup*
  then show $(\mathit{THE}\ \mathit{sup}.\ \mathit{is\text{-}Sup}\ A\ \mathit{sup}) = \mathit{sup}$
    by (*rule the-equality*) (*rule is-Sup-uniq* [*OF* - ‹*is-Sup A sup*›])
qed

lemma *JoinI* [*intro?*]:
  $(\bigwedge a.\ a \in A \Longrightarrow a \sqsubseteq \mathit{sup}) \Longrightarrow$
    $(\bigwedge b.\ \forall a \in A.\ a \sqsubseteq b \Longrightarrow \mathit{sup} \sqsubseteq b) \Longrightarrow$
    $\bigsqcup A = \mathit{sup}$
  by (*rule Join-equality*, *rule is-SupI*) *blast+*

The $\bigsqcap$ and $\bigsqcup$ operations indeed determine bounds on a complete lattice structure.

lemma *is-Inf-Meet* [*intro?*]: $\mathit{is\text{-}Inf}\ A\ (\bigsqcap A)$
proof (*unfold Meet-def*)
  from *ex-Inf* **obtain** *inf* **where** *is-Inf A inf* **..**
  then show $\mathit{is\text{-}Inf}\ A\ (\mathit{THE}\ \mathit{inf}.\ \mathit{is\text{-}Inf}\ A\ \mathit{inf})$
    by (*rule theI*) (*rule is-Inf-uniq* [*OF* - ‹*is-Inf A inf*›])
qed

lemma *Meet-greatest* [*intro?*]: $(\bigwedge a.\ a \in A \Longrightarrow x \sqsubseteq a) \Longrightarrow x \sqsubseteq \bigsqcap A$
  by (*rule is-Inf-greatest*, *rule is-Inf-Meet*) *blast*

lemma *Meet-lower* [*intro?*]: $a \in A \Longrightarrow \bigsqcap A \sqsubseteq a$
  by (*rule is-Inf-lower*) (*rule is-Inf-Meet*)

lemma *is-Sup-Join* [*intro?*]: $\mathit{is\text{-}Sup}\ A\ (\bigsqcup A)$
proof (*unfold Join-def*)
  from *ex-Sup* **obtain** *sup* **where** *is-Sup A sup* **..**
  then show $\mathit{is\text{-}Sup}\ A\ (\mathit{THE}\ \mathit{sup}.\ \mathit{is\text{-}Sup}\ A\ \mathit{sup})$

```
  by (rule theI) (rule is-Sup-uniq [OF - ‹is-Sup A sup›])
qed

lemma Join-least [intro?]: (⋀a. a ∈ A ⟹ a ⊑ x) ⟹ ⨆A ⊑ x
  by (rule is-Sup-least, rule is-Sup-Join) blast
lemma Join-lower [intro?]: a ∈ A ⟹ a ⊑ ⨆A
  by (rule is-Sup-upper) (rule is-Sup-Join)
```

4.2 The Knaster-Tarski Theorem

The Knaster-Tarski Theorem (in its simplest formulation) states that any monotone function on a complete lattice has a least fixed-point (see [2, pages 93–94] for example). This is a consequence of the basic boundary properties of the complete lattice operations.

```
theorem Knaster-Tarski:
  assumes mono: ⋀x y. x ⊑ y ⟹ f x ⊑ f y
  obtains a :: 'a::complete-lattice where
    f a = a and ⋀a'. f a' = a' ⟹ a ⊑ a'
proof
  let ?H = {u. f u ⊑ u}
  let ?a = ⨅?H
  show f ?a = ?a
  proof -
    have ge: f ?a ⊑ ?a
    proof
      fix x assume x: x ∈ ?H
      then have ?a ⊑ x ..
      then have f ?a ⊑ f x by (rule mono)
      also from x have ... ⊑ x ..
      finally show f ?a ⊑ x .
    qed
    also have ?a ⊑ f ?a
    proof
      from ge have f (f ?a) ⊑ f ?a by (rule mono)
      then show f ?a ∈ ?H ..
    qed
    finally show ?thesis .
  qed

  fix a'
  assume f a' = a'
  then have f a' ⊑ a' by (simp only: leq-refl)
  then have a' ∈ ?H ..
  then show ?a ⊑ a' ..
qed

theorem Knaster-Tarski-dual:
  assumes mono: ⋀x y. x ⊑ y ⟹ f x ⊑ f y
```

```
  obtains a :: 'a::complete-lattice where
    f a = a and ⋀a'. f a' = a' ⟹ a' ⊑ a
proof
  let ?H = {u. u ⊑ f u}
  let ?a = ⨆ ?H
  show f ?a = ?a
  proof −
    have le: ?a ⊑ f ?a
    proof
      fix x assume x: x ∈ ?H
      then have x ⊑ f x ..
      also from x have x ⊑ ?a ..
      then have f x ⊑ f ?a by (rule mono)
      finally show x ⊑ f ?a .
    qed
    have f ?a ⊑ ?a
    proof
      from le have f ?a ⊑ f (f ?a) by (rule mono)
      then show f ?a ∈ ?H ..
    qed
    from this and le show ?thesis by (rule leq-antisym)
  qed

  fix a'
  assume f a' = a'
  then have a' ⊑ f a' by (simp only: leq-refl)
  then have a' ∈ ?H ..
  then show a' ⊑ ?a ..
qed
```

4.3 Bottom and top elements

With general bounds available, complete lattices also have least and greatest elements.

```
definition
  bottom :: 'a::complete-lattice (⊥) where
  ⊥ = ⨅ UNIV

definition
  top :: 'a::complete-lattice (⊤) where
  ⊤ = ⨆ UNIV

lemma bottom-least [intro?]: ⊥ ⊑ x
proof (unfold bottom-def)
  have x ∈ UNIV ..
  then show ⨅ UNIV ⊑ x ..
qed

lemma bottomI [intro?]: (⋀a. x ⊑ a) ⟹ ⊥ = x
```

```
proof (unfold bottom-def)
  assume ⋀a. x ⊑ a
  show ⨅ UNIV = x
  proof
    fix a show x ⊑ a by fact
  next
    fix b :: 'a::complete-lattice
    assume b: ∀ a ∈ UNIV. b ⊑ a
    have x ∈ UNIV ..
    with b show b ⊑ x ..
  qed
qed
```

```
lemma top-greatest [intro?]: x ⊑ ⊤
proof (unfold top-def)
  have x ∈ UNIV ..
  then show x ⊑ ⨆ UNIV ..
qed
```

```
lemma topI [intro?]: (⋀a. a ⊑ x) ⟹ ⊤ = x
proof (unfold top-def)
  assume ⋀a. a ⊑ x
  show ⨆ UNIV = x
  proof
    fix a show a ⊑ x by fact
  next
    fix b :: 'a::complete-lattice
    assume b: ∀ a ∈ UNIV. a ⊑ b
    have x ∈ UNIV ..
    with b show x ⊑ b ..
  qed
qed
```

4.4 Duality

The class of complete lattices is closed under formation of dual structures.

```
instance dual :: (complete-lattice) complete-lattice
proof
  fix A' :: 'a::complete-lattice dual set
  show ∃ inf'. is-Inf A' inf'
  proof -
    have ∃ sup. is-Sup (undual ` A') sup by (rule ex-Sup)
   then have ∃ sup. is-Inf (dual ` undual ` A') (dual sup) by (simp only: dual-Inf)
   then show ?thesis by (simp add: dual-ex [symmetric] image-compose [symmetric])
  qed
qed
```

Apparently, the $\bigsqcap$ and $\bigsqcup$ operations are dual to each other.

```
theorem dual-Meet [intro?]: dual (⨅ A) = ⨆ (dual ` A)
```

```
proof –
  from is-Inf-Meet have is-Sup (dual ‘ A) (dual (⨅A)) ..
  then have ⨆(dual ‘ A) = dual (⨅A) ..
  then show ?thesis ..
qed

theorem dual-Join [intro?]: dual (⨆A) = ⨅(dual ‘ A)
proof –
  from is-Sup-Join have is-Inf (dual ‘ A) (dual (⨆A)) ..
  then have ⨅(dual ‘ A) = dual (⨆A) ..
  then show ?thesis ..
qed
```

Likewise are ⊥ and ⊤ duals of each other.

```
theorem dual-bottom [intro?]: dual ⊥ = ⊤
proof –
  have ⊤ = dual ⊥
  proof
    fix a′ have ⊥ ⊑ undual a′ ..
    then have dual (undual a′) ⊑ dual ⊥ ..
    then show a′ ⊑ dual ⊥ by simp
  qed
  then show ?thesis ..
qed

theorem dual-top [intro?]: dual ⊤ = ⊥
proof –
  have ⊥ = dual ⊤
  proof
    fix a′ have undual a′ ⊑ ⊤ ..
    then have dual ⊤ ⊑ dual (undual a′) ..
    then show dual ⊤ ⊑ a′ by simp
  qed
  then show ?thesis ..
qed
```

4.5 Complete lattices are lattices

Complete lattices (with general bounds available) are indeed plain lattices as well. This holds due to the connection of general versus binary bounds that has been formally established in §2.5.

```
lemma is-inf-binary: is-inf x y (⨅{x, y})
proof –
  have is-Inf {x, y} (⨅{x, y}) ..
  then show ?thesis by (simp only: is-Inf-binary)
qed

lemma is-sup-binary: is-sup x y (⨆{x, y})
```

```
proof −
  have is-Sup {x, y} (⨆{x, y}) ..
  then show ?thesis by (simp only: is-Sup-binary)
qed

instance complete-lattice ⊆ lattice
proof
  fix x y :: 'a::complete-lattice
  from is-inf-binary show ∃ inf. is-inf x y inf ..
  from is-sup-binary show ∃ sup. is-sup x y sup ..
qed

theorem meet-binary: x ⊓ y = ⨅{x, y}
  by (rule meet-equality) (rule is-inf-binary)

theorem join-binary: x ⊔ y = ⨆{x, y}
  by (rule join-equality) (rule is-sup-binary)
```

4.6 Complete lattices and set-theory operations

The complete lattice operations are (anti) monotone wrt. set inclusion.

```
theorem Meet-subset-antimono: A ⊆ B ⟹ ⨅B ⊑ ⨅A
proof (rule Meet-greatest)
  fix a assume a ∈ A
  also assume A ⊆ B
  finally have a ∈ B .
  then show ⨅B ⊑ a ..
qed

theorem Join-subset-mono: A ⊆ B ⟹ ⨆A ⊑ ⨆B
proof −
  assume A ⊆ B
  then have dual ` A ⊆ dual ` B by blast
  then have ⨅(dual ` B) ⊑ ⨅(dual ` A) by (rule Meet-subset-antimono)
  then have dual (⨆B) ⊑ dual (⨆A) by (simp only: dual-Join)
  then show ?thesis by (simp only: dual-leq)
qed
```

Bounds over unions of sets may be obtained separately.

```
theorem Meet-Un: ⨅(A ∪ B) = ⨅A ⊓ ⨅B
proof
  fix a assume a ∈ A ∪ B
  then show ⨅A ⊓ ⨅B ⊑ a
  proof
    assume a: a ∈ A
    have ⨅A ⊓ ⨅B ⊑ ⨅A ..
    also from a have . . . ⊑ a ..
    finally show ?thesis .
```

```
  next
    assume a: a ∈ B
    have ⨅A ⊓ ⨅B ⊑ ⨅B ..
    also from a have ... ⊑ a ..
    finally show ?thesis .
  qed
next
  fix b assume b: ∀ a ∈ A ∪ B. b ⊑ a
  show b ⊑ ⨅A ⊓ ⨅B
  proof
    show b ⊑ ⨅A
    proof
      fix a assume a ∈ A
      then have a ∈  A ∪ B ..
      with b show b ⊑ a ..
    qed
    show b ⊑ ⨅B
    proof
      fix a assume a ∈ B
      then have a ∈  A ∪ B ..
      with b show b ⊑ a ..
    qed
  qed
qed

theorem Join-Un: ⨆(A ∪ B) = ⨆A ⊔ ⨆B
proof −
  have dual (⨆(A ∪ B)) = ⨅(dual ‘ A ∪ dual ‘ B)
    by (simp only: dual-Join image-Un)
  also have ... = ⨅(dual ‘ A) ⊓ ⨅(dual ‘ B)
    by (rule Meet-Un)
  also have ... = dual (⨆A ⊔ ⨆B)
    by (simp only: dual-join dual-Join)
  finally show ?thesis ..
qed
```

Bounds over singleton sets are trivial.

```
theorem Meet-singleton: ⨅{x} = x
proof
  fix a assume a ∈ {x}
  then have a = x by simp
  then show x ⊑ a by (simp only: leq-refl)
next
  fix b assume ∀ a ∈ {x}. b ⊑ a
  then show b ⊑ x by simp
qed

theorem Join-singleton: ⨆{x} = x
proof −
```

```
  have dual (⨆{x}) = ⨅{dual x} by (simp add: dual-Join)
  also have ... = dual x by (rule Meet-singleton)
  finally show ?thesis ..
qed
```

Bounds over the empty and universal set correspond to each other.

```
theorem Meet-empty: ⨅{} = ⨆ UNIV
proof
  fix a :: 'a::complete-lattice
  assume a ∈ {}
  then have False by simp
  then show ⨆ UNIV ⊑ a ..
next
  fix b :: 'a::complete-lattice
  have b ∈ UNIV ..
  then show b ⊑ ⨆ UNIV ..
qed

theorem Join-empty: ⨆{} = ⨅ UNIV
proof –
  have dual (⨆{}) = ⨅{} by (simp add: dual-Join)
  also have ... = ⨆ UNIV by (rule Meet-empty)
  also have ... = dual (⨅ UNIV) by (simp add: dual-Meet)
  finally show ?thesis ..
qed

end
```

References

[1] G. Bauer and M. Wenzel. Computer-assisted mathematics at work — the Hahn-Banach theorem in Isabelle/Isar. In T. Coquand, P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs: TYPES'99*, LNCS, 2000.

[2] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[3] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.

[4] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2000. http://isabelle.in.tum.de/doc/isar-ref.pdf.

[5] M. Wenzel. *Using Axiomatic Type Classes in Isabelle*, 2000. http://isabelle.in.tum.de/doc/axclass.pdf.