Free Boolean Algebra

Brian Huffman

March 17, 2025

Abstract

This theory defines a type constructor representing the free Boolean algebra over a set of generators. Values of type (α)*formula* represent propositional formulas with uninterpreted variables from type α, ordered by implication. In addition to all the standard Boolean algebra operations, the library also provides a function for building homomorphisms to any other Boolean algebra type.

1 Free Boolean algebras

theory *Free-Boolean-Algebra*
imports *Main*
begin

1.1 Free boolean algebra as a set

We start by defining the free boolean algebra over type $'a$ as an inductive set. Here $i :: 'a$ represents a variable; $A :: 'a\ set$ represents a valuation, assigning a truth value to each variable; and $S :: 'a\ set\ set$ represents a formula, as the set of valuations that make the formula true. The set *fba* contains representatives of formulas built from finite combinations of variables with negation and conjunction.

inductive-set
  $fba :: 'a\ set\ set\ set$
where
  var: $\{A.\ i \in A\} \in fba$
| *Compl*: $S \in fba \Longrightarrow -\ S \in fba$
| *inter*: $S \in fba \Longrightarrow T \in fba \Longrightarrow S \cap T \in fba$

lemma *fba-Diff*: $S \in fba \Longrightarrow T \in fba \Longrightarrow S - T \in fba$
unfolding *Diff-eq* **by** (*intro fba.inter fba.Compl*)

lemma *fba-union*: $S \in fba \Longrightarrow T \in fba \Longrightarrow S \cup T \in fba$
proof −
  assume $S \in fba$ **and** $T \in fba$

```
  hence − (− S ∩ − T) ∈ fba by (intro fba.intros)
  thus S ∪ T ∈ fba by simp
qed

lemma fba-empty: ({} :: 'a set set) ∈ fba
proof −
  obtain S :: 'a set set where S ∈ fba
    by (fast intro: fba.var)
  hence S ∩ − S ∈ fba
    by (intro fba.intros)
  thus ?thesis by simp
qed

lemma fba-UNIV: (UNIV :: 'a set set) ∈ fba
proof −
  have − {} ∈ fba using fba-empty by (rule fba.Compl)
  thus UNIV ∈ fba by simp
qed
```

1.2 Free boolean algebra as a type

The next step is to use *typedef* to define a type isomorphic to the set *fba*. We also define a constructor *var* that corresponds with the similarly-named introduction rule for *fba*.

```
typedef 'a formula = fba :: 'a set set set
  by (auto intro: fba-empty)

definition var :: 'a ⇒ 'a formula
where var i = Abs-formula {A. i ∈ A}

lemma Rep-formula-var: Rep-formula (var i) = {A. i ∈ A}
unfolding var-def using fba.var by (rule Abs-formula-inverse)
```

Now we make type *'a formula* into a Boolean algebra. This involves defining the various operations (ordering relations, binary infimum and supremum, complement, difference, top and bottom elements) and proving that they satisfy the appropriate laws.

```
instantiation formula :: (type) boolean-algebra
begin

definition
  x ⊓ y = Abs-formula (Rep-formula x ∩ Rep-formula y)

definition
  x ⊔ y = Abs-formula (Rep-formula x ∪ Rep-formula y)

definition
  ⊤ = Abs-formula UNIV
```

definition
$\bot = \textit{Abs-formula}\ \{\}$

definition
$x \le y \longleftrightarrow \textit{Rep-formula}\ x \subseteq \textit{Rep-formula}\ y$

definition
$x < y \longleftrightarrow \textit{Rep-formula}\ x \subset \textit{Rep-formula}\ y$

definition
$-\ x = \textit{Abs-formula}\ (-\ \textit{Rep-formula}\ x)$

definition
$x - y = \textit{Abs-formula}\ (\textit{Rep-formula}\ x - \textit{Rep-formula}\ y)$

lemma *Rep-formula-inf*:
$\textit{Rep-formula}\ (x \sqcap y) = \textit{Rep-formula}\ x \cap \textit{Rep-formula}\ y$
unfolding *inf-formula-def*
by (*intro Abs-formula-inverse fba.inter Rep-formula*)

lemma *Rep-formula-sup*:
$\textit{Rep-formula}\ (x \sqcup y) = \textit{Rep-formula}\ x \cup \textit{Rep-formula}\ y$
unfolding *sup-formula-def*
by (*intro Abs-formula-inverse fba-union Rep-formula*)

lemma *Rep-formula-top*: $\textit{Rep-formula}\ \top = \textit{UNIV}$
unfolding *top-formula-def* **by** (*intro Abs-formula-inverse fba-UNIV*)

lemma *Rep-formula-bot*: $\textit{Rep-formula}\ \bot = \{\}$
unfolding *bot-formula-def* **by** (*intro Abs-formula-inverse fba-empty*)

lemma *Rep-formula-compl*: $\textit{Rep-formula}\ (-\ x) = -\ \textit{Rep-formula}\ x$
unfolding *uminus-formula-def*
by (*intro Abs-formula-inverse fba.Compl Rep-formula*)

lemma *Rep-formula-diff*:
$\textit{Rep-formula}\ (x - y) = \textit{Rep-formula}\ x - \textit{Rep-formula}\ y$
unfolding *minus-formula-def*
by (*intro Abs-formula-inverse fba-Diff Rep-formula*)

lemmas *eq-formula-iff* = *Rep-formula-inject* [*symmetric*]

lemmas *Rep-formula-simps* =
less-eq-formula-def less-formula-def eq-formula-iff
Rep-formula-sup Rep-formula-inf Rep-formula-top Rep-formula-bot
Rep-formula-compl Rep-formula-diff Rep-formula-var

instance proof

qed (*unfold Rep-formula-simps*, *auto*)

end

The laws of a Boolean algebra do not require the top and bottom elements to be distinct, so the following rules must be proved separately:

lemma *bot-neq-top-formula* [*simp*]: $(\bot :: 'a\ formula) \neq \top$
unfolding *Rep-formula-simps* **by** *auto*

lemma *top-neq-bot-formula* [*simp*]: $(\top :: 'a\ formula) \neq \bot$
unfolding *Rep-formula-simps* **by** *auto*

Here we prove an essential property of a free Boolean algebra: all generators are independent.

lemma *var-le-var-simps* [*simp*]:
  $var\ i \leq var\ j \longleftrightarrow i = j$
  $\neg\ var\ i \leq -\ var\ j$
  $\neg\ -\ var\ i \leq var\ j$
unfolding *Rep-formula-simps* **by** *fast+*

lemma *var-eq-var-simps* [*simp*]:
  $var\ i = var\ j \longleftrightarrow i = j$
  $var\ i \neq -\ var\ j$
  $-\ var\ i \neq var\ j$
unfolding *Rep-formula-simps set-eq-subset* **by** *fast+*

We conclude this section by proving an induction principle for formulas. It mirrors the definition of the inductive set *fba*, with cases for variables, complements, and conjunction.

lemma *formula-induct* [*case-names var compl inf*, *induct type*: *formula*]:
  fixes $P :: 'a\ formula \Rightarrow bool$
  assumes *1*: $\bigwedge i.\ P\ (var\ i)$
  assumes *2*: $\bigwedge x.\ P\ x \Longrightarrow P\ (-\ x)$
  assumes *3*: $\bigwedge x\ y.\ P\ x \Longrightarrow P\ y \Longrightarrow P\ (x \sqcap y)$
  shows $P\ x$
proof (*induct x rule*: *Abs-formula-induct*)
  fix $y :: 'a\ set\ set$
  assume $y \in fba$ **thus** $P\ (Abs\text{-}formula\ y)$
  proof (*induct rule*: *fba.induct*)
    case (*var i*)
    have $P\ (var\ i)$ **by** (*rule 1*)
    thus *?case* **unfolding** *var-def* **.**
  next
    case (*Compl S*)
    from ‹$P\ (Abs\text{-}formula\ S)$› **have** $P\ (-\ Abs\text{-}formula\ S)$ **by** (*rule 2*)
    with ‹$S \in fba$› **show** *?case*
      unfolding *uminus-formula-def* **by** (*simp add*: *Abs-formula-inverse*)

next
case (*inter S T*)
from ‹*P* (*Abs-formula S*)› **and** ‹*P* (*Abs-formula T*)›
have *P* (*Abs-formula S* ⊓ *Abs-formula T*) **by** (*rule 3*)
with ‹*S* ∈ *fba*› **and** ‹*T* ∈ *fba*› **show** *?case*
unfolding *inf-formula-def* **by** (*simp add*: *Abs-formula-inverse*)
qed
qed

1.3 If-then-else for Boolean algebras

This is a generic if-then-else operator for arbitrary Boolean algebras.

definition
ifte :: *'a*::*boolean-algebra* ⇒ *'a* ⇒ *'a* ⇒ *'a*
where
ifte a x y = (*a* ⊓ *x*) ⊔ (− *a* ⊓ *y*)

lemma *ifte-top* [*simp*]: *ifte* ⊤ *x y* = *x*
unfolding *ifte-def* **by** *simp*

lemma *ifte-bot* [*simp*]: *ifte* ⊥ *x y* = *y*
unfolding *ifte-def* **by** *simp*

lemma *ifte-same*: *ifte a x x* = *x*
unfolding *ifte-def*
by (*simp add*: *inf-sup-distrib2* [*symmetric*] *sup-compl-top*)

lemma *compl-ifte*: − *ifte a x y* = *ifte a* (− *x*) (− *y*)
unfolding *ifte-def*
apply (*rule order-antisym*)
apply (*simp add*: *inf-sup-distrib1 inf-sup-distrib2 compl-inf-bot*)
apply (*simp add*: *sup-inf-distrib1 sup-inf-distrib2 sup-compl-top*)
apply (*simp add*: *le-infI1 le-infI2 le-supI1 le-supI2*)
apply (*simp add*: *le-infI1 le-infI2 le-supI1 le-supI2*)
done

lemma *inf-ifte-distrib*:
ifte x a b ⊓ *ifte x c d* = *ifte x* (*a* ⊓ *c*) (*b* ⊓ *d*)
unfolding *ifte-def*
apply (*simp add*: *inf-sup-distrib1 inf-sup-distrib2*)
apply (*simp add*: *inf-sup-aci inf-compl-bot*)
done

lemma *ifte-ifte-distrib*:
ifte x (*ifte y a b*) (*ifte y c d*) = *ifte y* (*ifte x a c*) (*ifte x b d*)
unfolding *ifte-def* [*of x*] *sup-conv-inf*
by (*simp only*: *compl-ifte* [*symmetric*] *inf-ifte-distrib* [*symmetric*] *ifte-same*)

1.4 Formulas over a set of generators

The set *formulas S* consists of those formulas that only depend on variables in the set *S*. It is analogous to the *lists* operator for the list datatype.

definition
  $formulas :: 'a\ set \Rightarrow 'a\ formula\ set$
where
  $formulas\ S =$
    $\{x.\ \forall A\ B.\ (\forall i{\in}S.\ i \in A \longleftrightarrow i \in B) \longrightarrow$
      $A \in Rep\text{-}formula\ x \longleftrightarrow B \in Rep\text{-}formula\ x\}$

lemma *formulasI*:
  assumes $\bigwedge A\ B.\ \forall i{\in}S.\ i \in A \longleftrightarrow i \in B$
    $\Longrightarrow A \in Rep\text{-}formula\ x \longleftrightarrow B \in Rep\text{-}formula\ x$
  shows $x \in formulas\ S$
using *assms* **unfolding** *formulas-def* **by** *simp*

lemma *formulasD*:
  assumes $x \in formulas\ S$
  assumes $\forall i{\in}S.\ i \in A \longleftrightarrow i \in B$
  shows $A \in Rep\text{-}formula\ x \longleftrightarrow B \in Rep\text{-}formula\ x$
using *assms* **unfolding** *formulas-def* **by** *simp*

lemma *formulas-mono*: $S \subseteq T \Longrightarrow formulas\ S \subseteq formulas\ T$
by (*fast intro*!: *formulasI elim*!: *formulasD*)

lemma *formulas-insert*: $x \in formulas\ S \Longrightarrow x \in formulas\ (insert\ a\ S)$
unfolding *formulas-def* **by** *simp*

lemma *formulas-var*: $i \in S \Longrightarrow var\ i \in formulas\ S$
unfolding *formulas-def* **by** (*simp add*: *Rep-formula-simps*)

lemma *formulas-var-iff*: $var\ i \in formulas\ S \longleftrightarrow i \in S$
unfolding *formulas-def* **by** (*simp add*: *Rep-formula-simps*, *fast*)

lemma *formulas-bot*: $\bot \in formulas\ S$
unfolding *formulas-def* **by** (*simp add*: *Rep-formula-simps*)

lemma *formulas-top*: $\top \in formulas\ S$
unfolding *formulas-def* **by** (*simp add*: *Rep-formula-simps*)

lemma *formulas-compl*: $x \in formulas\ S \Longrightarrow -\ x \in formulas\ S$
unfolding *formulas-def* **by** (*simp add*: *Rep-formula-simps*)

lemma *formulas-inf*:
  $x \in formulas\ S \Longrightarrow y \in formulas\ S \Longrightarrow x \sqcap y \in formulas\ S$
unfolding *formulas-def* **by** (*auto simp add*: *Rep-formula-simps*)

lemma *formulas-sup*:

$x \in \mathit{formulas}\ S \Longrightarrow y \in \mathit{formulas}\ S \Longrightarrow x \sqcup y \in \mathit{formulas}\ S$
unfolding *formulas-def* **by** (*auto simp add*: *Rep-formula-simps*)

lemma *formulas-diff*:
$x \in \mathit{formulas}\ S \Longrightarrow y \in \mathit{formulas}\ S \Longrightarrow x - y \in \mathit{formulas}\ S$
unfolding *formulas-def* **by** (*auto simp add*: *Rep-formula-simps*)

lemma *formulas-ifte*:
$a \in \mathit{formulas}\ S \Longrightarrow x \in \mathit{formulas}\ S \Longrightarrow y \in \mathit{formulas}\ S \Longrightarrow$
$\mathit{ifte}\ a\ x\ y \in \mathit{formulas}\ S$
unfolding *ifte-def*
by (*intro formulas-sup formulas-inf formulas-compl*)

lemmas *formulas-intros* =
formulas-var formulas-bot formulas-top formulas-compl
formulas-inf formulas-sup formulas-diff formulas-ifte

1.5 Injectivity of if-then-else

The if-then-else operator is injective in some limited circumstances: when the scrutinee is a variable that is not mentioned in either branch.

lemma *ifte-inject*:
assumes $\mathit{ifte}\ (\mathit{var}\ i)\ x\ y = \mathit{ifte}\ (\mathit{var}\ i)\ x'\ y'$
assumes $i \notin S$
assumes $x \in \mathit{formulas}\ S$ **and** $x' \in \mathit{formulas}\ S$
assumes $y \in \mathit{formulas}\ S$ **and** $y' \in \mathit{formulas}\ S$
shows $x = x' \wedge y = y'$
proof
have *1*: $\bigwedge A.\ i \in A \Longrightarrow A \in \mathit{Rep\text{-}formula}\ x \longleftrightarrow A \in \mathit{Rep\text{-}formula}\ x'$
using *assms*(*1*)
by (*simp add*: *Rep-formula-simps ifte-def set-eq-iff*, *fast*)
have *2*: $\bigwedge A.\ i \notin A \Longrightarrow A \in \mathit{Rep\text{-}formula}\ y \longleftrightarrow A \in \mathit{Rep\text{-}formula}\ y'$
using *assms*(*1*)
by (*simp add*: *Rep-formula-simps ifte-def set-eq-iff*, *fast*)

show $x = x'$
unfolding *Rep-formula-simps*
proof (*rule set-eqI*)
fix *A*
have $A \in \mathit{Rep\text{-}formula}\ x \longleftrightarrow \mathit{insert}\ i\ A \in \mathit{Rep\text{-}formula}\ x$
using ‹$x \in \mathit{formulas}\ S$› **by** (*rule formulasD*, *force simp add*: ‹$i \notin S$›)
also have $\ldots \longleftrightarrow \mathit{insert}\ i\ A \in \mathit{Rep\text{-}formula}\ x'$
by (*rule 1*, *simp*)
also have $\ldots \longleftrightarrow A \in \mathit{Rep\text{-}formula}\ x'$
using ‹$x' \in \mathit{formulas}\ S$› **by** (*rule formulasD*, *force simp add*: ‹$i \notin S$›)
finally show $A \in \mathit{Rep\text{-}formula}\ x \longleftrightarrow A \in \mathit{Rep\text{-}formula}\ x'$ **.**
qed
show $y = y'$
unfolding *Rep-formula-simps*

proof (*rule set-eqI*)
  fix *A*
  have $A \in$ *Rep-formula y* $\longleftrightarrow A - \{i\} \in$ *Rep-formula y*
    using ‹$y \in$ *formulas S*› **by** (*rule formulasD*, *force simp add*: ‹$i \notin S$›)
  also have ... $\longleftrightarrow A - \{i\} \in$ *Rep-formula y'*
    by (*rule 2*, *simp*)
  also have ... $\longleftrightarrow A \in$ *Rep-formula y'*
    using ‹$y' \in$ *formulas S*› **by** (*rule formulasD*, *force simp add*: ‹$i \notin S$›)
  finally show $A \in$ *Rep-formula y* $\longleftrightarrow A \in$ *Rep-formula y'* **.**
qed
qed

1.6 Specification of homomorphism operator

Our goal is to define a homomorphism operator *hom* such that for any function *f*, *hom f* is the unique Boolean algebra homomorphism satisfying *hom f* (*var i*) = *f i* for all *i*.

Instead of defining *hom* directly, we will follow the approach used to define Isabelle's *fold* operator for finite sets. First we define the graph of the *hom* function as a relation; later we will define the *hom* function itself using definite choice.

The *hom-graph* relation is defined inductively, with introduction rules based on the if-then-else normal form of Boolean formulas. The relation is also indexed by an extra set parameter *S*, to ensure that branches of each if-then-else do not use the same variable again.

inductive
  hom-graph ::
    ($'a \Rightarrow 'b$::*boolean-algebra*) $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *formula* $\Rightarrow$ $'b \Rightarrow$ *bool*
  for *f* :: $'a \Rightarrow 'b$::*boolean-algebra*
where
  bot: *hom-graph f* {} *bot bot*
| *top*: *hom-graph f* {} *top top*
| *ifte*: $i \notin S \Longrightarrow$ *hom-graph f S x a* $\Longrightarrow$ *hom-graph f S y b* $\Longrightarrow$
  hom-graph f (*insert i S*) (*ifte* (*var i*) *x y*) (*ifte* (*f i*) *a b*)

The next two lemmas establish a stronger elimination rule for assumptions of the form *hom-graph f* (*insert i S*) *x a*. Essentially, they say that we can arrange the top-level if-then-else to use the variable of our choice. The proof makes use of the distributive properties of if-then-else.

lemma *hom-graph-dest*:
  hom-graph f S x a $\Longrightarrow k \in S \Longrightarrow \exists\, y\ z\ b\ c.$
    $x =$ *ifte* (*var k*) *y z* $\wedge$ $a =$ *ifte* (*f k*) *b c* $\wedge$
    hom-graph f ($S - \{k\}$) *y b* $\wedge$ *hom-graph f* ($S - \{k\}$) *z c*
proof (*induct set*: *hom-graph*)
  case (*ifte i S x a y b*) **show** *?case*
  proof (*cases* $i = k$)
    assume $i = k$ **with** *ifte*(*1*,*2*,*4*) **show** *?case* **by** *auto*

```
  next
    assume i ≠ k
    with ‹k ∈ insert i S› have k: k ∈ S by simp
    have *: insert i S − {k} = insert i (S − {k})
      using ‹i ≠ k› by (simp add: insert-Diff-if)
    have **: i ∉ S − {k} using ‹i ∉ S› by simp
    from ifte(1) ifte(3) [OF k] ifte(5) [OF k]
    show ?case
      unfolding *
      apply clarify
      apply (simp only: ifte-ifte-distrib [of var i])
      apply (simp only: ifte-ifte-distrib [of f i])
      apply (fast intro: hom-graph.ifte [OF **])
      done
  qed
qed simp-all

lemma hom-graph-insert-elim:
  assumes hom-graph f (insert i S) x a and i ∉ S
  obtains y z b c
  where x = ifte (var i) y z
    and a = ifte (f i) b c
    and hom-graph f S y b
    and hom-graph f S z c
using hom-graph-dest [OF assms(1) insertI1]
by (clarify, simp add: assms(2))
```

Now we prove the first uniqueness property of the *hom-graph* relation. This version of uniqueness says that for any particular value of S, the relation *hom-graph f S* maps each x to at most one a. The proof uses the injectiveness of if-then-else, which we proved earlier.

```
lemma hom-graph-imp-formulas:
  hom-graph f S x a ⟹ x ∈ formulas S
by (induct set: hom-graph, simp-all add: formulas-intros formulas-insert)

lemma hom-graph-unique:
  hom-graph f S x a ⟹ hom-graph f S x a' ⟹ a = a'
proof (induct arbitrary: a' set: hom-graph)
  case (ifte i S y b z c a')
  from ifte(6,1) obtain y' z' b' c'
    where 1: ifte (var i) y z = ifte (var i) y' z'
      and 2: a' = ifte (f i) b' c'
      and 3: hom-graph f S y' b'
      and 4: hom-graph f S z' c'
    by (rule hom-graph-insert-elim)
  from 1 3 4 ifte(1,2,4) have y = y' ∧ z = z'
    by (intro ifte-inject hom-graph-imp-formulas)
  with 2 3 4 ifte(3,5) show ifte (f i) b c = a'
    by simp
```

```
qed (erule hom-graph.cases, simp-all)+
```

The next few lemmas will help to establish a stronger version of the uniqueness property of *hom-graph*. They show that the *hom-graph* relation is preserved if we replace S with a larger finite set.

```
lemma hom-graph-insert:
  assumes hom-graph f S x a
  shows hom-graph f (insert i S) x a
proof (cases i ∈ S)
  assume i ∈ S with assms show ?thesis by (simp add: insert-absorb)
next
  assume i ∉ S
  hence hom-graph f (insert i S) (ifte (var i) x x) (ifte (f i) a a)
    by (intro hom-graph.ifte assms)
  thus hom-graph f (insert i S) x a
    by (simp only: ifte-same)
qed

lemma hom-graph-finite-superset:
  assumes hom-graph f S x a and finite T and S ⊆ T
  shows hom-graph f T x a
proof −
  from ‹finite T› have hom-graph f (S ∪ T) x a
    by (induct set: finite, simp add: assms, simp add: hom-graph-insert)
  with ‹S ⊆ T› show hom-graph f T x a
    by (simp only: subset-Un-eq)
qed

lemma hom-graph-imp-finite:
  hom-graph f S x a ⟹ finite S
by (induct set: hom-graph) simp-all
```

This stronger uniqueness property says that *hom-graph f* maps each x to at most one a, even for *different* values of the set parameter.

```
lemma hom-graph-unique':
  assumes hom-graph f S x a and hom-graph f T x a'
  shows a = a'
proof (rule hom-graph-unique)
  have fin: finite (S ∪ T)
    using assms by (intro finite-UnI hom-graph-imp-finite)
  show hom-graph f (S ∪ T) x a
    using assms(1) fin Un-upper1 by (rule hom-graph-finite-superset)
  show hom-graph f (S ∪ T) x a'
    using assms(2) fin Un-upper2 by (rule hom-graph-finite-superset)
qed
```

Finally, these last few lemmas establish that the *hom-graph f* relation is total: every x is mapped to some a.

lemma *hom-graph-var*: *hom-graph f* $\{i\}$ (*var i*) (*f i*)
proof −
  have *hom-graph f* $\{i\}$ (*ifte* (*var i*) *top bot*) (*ifte* (*f i*) *top bot*)
    by (*simp add*: *hom-graph.intros*)
  thus *hom-graph f* $\{i\}$ (*var i*) (*f i*)
    unfolding *ifte-def* **by** *simp*
qed

lemma *hom-graph-compl*:
  hom-graph f S x a $\Longrightarrow$ *hom-graph f S* (− *x*) (− *a*)
by (*induct set*: *hom-graph*, *simp-all add*: *hom-graph.intros compl-ifte*)

lemma *hom-graph-inf*:
  hom-graph f S x a $\Longrightarrow$ *hom-graph f S y b* $\Longrightarrow$
   hom-graph f S ($x \sqcap y$) ($a \sqcap b$)
 apply (*induct arbitrary*: *y b set*: *hom-graph*)
   apply (*simp add*: *hom-graph.bot*)
  apply *simp*
 apply (*erule* (*1*) *hom-graph-insert-elim*)
 apply (*auto simp add*: *inf-ifte-distrib hom-graph.ifte*)
done

lemma *hom-graph-union-inf*:
  assumes *hom-graph f S x a* **and** *hom-graph f T y b*
  shows *hom-graph f* ($S \cup T$) ($x \sqcap y$) ($a \sqcap b$)
proof (*rule hom-graph-inf*)
  have *fin*: *finite* ($S \cup T$)
    using *assms* **by** (*intro finite-UnI hom-graph-imp-finite*)
  show *hom-graph f* ($S \cup T$) *x a*
    using *assms*(*1*) *fin Un-upper1* **by** (*rule hom-graph-finite-superset*)
  show *hom-graph f* ($S \cup T$) *y b*
    using *assms*(*2*) *fin Un-upper2* **by** (*rule hom-graph-finite-superset*)
qed

lemma *hom-graph-exists*: $\exists$ *a S*. *hom-graph f S x a*
by (*induct x*)
   (*auto intro*: *hom-graph-var hom-graph-compl hom-graph-union-inf*)

1.7 Homomorphisms into other boolean algebras

Now that we have proved the necessary existence and uniqueness properties of *hom-graph*, we can define the function *hom* using definite choice.

definition
  hom :: (*'a* $\Rightarrow$ *'b*::*boolean-algebra*) $\Rightarrow$ *'a formula* $\Rightarrow$ *'b*
where
  hom f x = (*THE a*. $\exists$ *S*. *hom-graph f S x a*)

lemma *hom-graph-hom*: $\exists$ *S*. *hom-graph f S x* (*hom f x*)
unfolding *hom-def*

apply (*rule theI'*)
apply (*rule ex-ex1I*)
apply (*rule hom-graph-exists*)
apply (*fast elim*: *hom-graph-unique'*)
done

lemma *hom-equality*:
  hom-graph f S x a $\Longrightarrow$ *hom f x* = *a*
unfolding *hom-def*
apply (*rule the-equality*)
apply (*erule exI*)
apply (*erule exE*)
apply (*erule* (*1*) *hom-graph-unique'*)
done

The *hom* function correctly implements its specification:

lemma *hom-var* [*simp*]: *hom f* (*var i*) = *f i*
by (*rule hom-equality*, *rule hom-graph-var*)

lemma *hom-bot* [*simp*]: *hom f* $\bot$ = $\bot$
by (*rule hom-equality*, *rule hom-graph.bot*)

lemma *hom-top* [*simp*]: *hom f* $\top$ = $\top$
by (*rule hom-equality*, *rule hom-graph.top*)

lemma *hom-compl* [*simp*]: *hom f* (− *x*) = − *hom f x*
proof −
  obtain *S* **where** *hom-graph f S x* (*hom f x*)
    using *hom-graph-hom* **..**
  hence *hom-graph f S* (− *x*) (− *hom f x*)
    by (*rule hom-graph-compl*)
  thus *hom f* (− *x*) = − *hom f x*
    by (*rule hom-equality*)
qed

lemma *hom-inf* [*simp*]: *hom f* (*x* $\sqcap$ *y*) = *hom f x* $\sqcap$ *hom f y*
proof −
  obtain *S* **where** *S*: *hom-graph f S x* (*hom f x*)
    using *hom-graph-hom* **..**
  obtain *T* **where** *T*: *hom-graph f T y* (*hom f y*)
    using *hom-graph-hom* **..**
  have *hom-graph f* (*S* $\cup$ *T*) (*x* $\sqcap$ *y*) (*hom f x* $\sqcap$ *hom f y*)
    using *S T* **by** (*rule hom-graph-union-inf*)
  thus *?thesis* **by** (*rule hom-equality*)
qed

lemma *hom-sup* [*simp*]: *hom f* (*x* $\sqcup$ *y*) = *hom f x* $\sqcup$ *hom f y*
unfolding *sup-conv-inf* **by** (*simp only*: *hom-compl hom-inf*)

lemma *hom-diff* [*simp*]: *hom f* (*x* − *y*) = *hom f x* − *hom f y*
unfolding *diff-eq* **by** (*simp only*: *hom-compl hom-inf*)

lemma *hom-ifte* [*simp*]:
hom f (*ifte x y z*) = *ifte* (*hom f x*) (*hom f y*) (*hom f z*)
unfolding *ifte-def* **by** (*simp only*: *hom-compl hom-inf hom-sup*)

lemmas *hom-simps* =
hom-var hom-bot hom-top hom-compl
hom-inf hom-sup hom-diff hom-ifte

The type *'a formula* can be viewed as a monad, with *var* as the unit, and *hom* as the bind operator. We can prove the standard monad laws with simple proofs by induction.

lemma *hom-var-eq-id*: *hom var x* = *x*
by (*induct x*) *simp-all*

lemma *hom-hom*: *hom f* (*hom g x*) = *hom* (λ*i*. *hom f* (*g i*)) *x*
by (*induct x*) *simp-all*

1.8 Map operation on Boolean formulas

We can define a map functional in terms of *hom* and *var*. The properties of *fmap* follow directly from the lemmas we have already proved about *hom*.

definition
fmap :: (*'a* ⇒ *'b*) ⇒ *'a formula* ⇒ *'b formula*
where
fmap f = *hom* (λ*i*. *var* (*f i*))

lemma *fmap-var* [*simp*]: *fmap f* (*var i*) = *var* (*f i*)
unfolding *fmap-def* **by** *simp*

lemma *fmap-bot* [*simp*]: *fmap f* ⊥ = ⊥
unfolding *fmap-def* **by** *simp*

lemma *fmap-top* [*simp*]: *fmap f* ⊤ = ⊤
unfolding *fmap-def* **by** *simp*

lemma *fmap-compl* [*simp*]: *fmap f* (− *x*) = − *fmap f x*
unfolding *fmap-def* **by** *simp*

lemma *fmap-inf* [*simp*]: *fmap f* (*x* ⊓ *y*) = *fmap f x* ⊓ *fmap f y*
unfolding *fmap-def* **by** *simp*

lemma *fmap-sup* [*simp*]: *fmap f* (*x* ⊔ *y*) = *fmap f x* ⊔ *fmap f y*
unfolding *fmap-def* **by** *simp*

lemma *fmap-diff* [*simp*]: *fmap f* (*x* − *y*) = *fmap f x* − *fmap f y*

unfolding *fmap-def* **by** *simp*

lemma *fmap-ifte* [*simp*]:
fmap f (*ifte x y z*) = *ifte* (*fmap f x*) (*fmap f y*) (*fmap f z*)
unfolding *fmap-def* **by** *simp*

lemmas *fmap-simps* =
fmap-var fmap-bot fmap-top fmap-compl
fmap-inf fmap-sup fmap-diff fmap-ifte

The map functional satisfies the functor laws: it preserves identity and function composition.

lemma *fmap-ident*: *fmap* ($\lambda i.\ i$) $x = x$
by (*induct x*) *simp-all*

lemma *fmap-fmap*: *fmap f* (*fmap g x*) = *fmap* ($f \circ g$) *x*
by (*induct x*) *simp-all*

1.9 Hiding lattice syntax

The following command hides the lattice syntax, to avoid potential conflicts with other theories that import this one. To re-enable the syntax, users should unbundle *lattice-syntax*.

unbundle *no lattice-syntax*

end