Free Boolean Algebra

Brian Huffman

October 13, 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
⟨*proof*⟩

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*: $S \in fba \Longrightarrow T \in fba \Longrightarrow S - T \in fba$
⟨*proof*⟩

lemma *fba-union*: $S \in fba \Longrightarrow T \in fba \Longrightarrow S \cup T \in fba$
⟨*proof*⟩

lemma *fba-empty*: $(\{\} :: {}'a\ set\ set) \in fba$
$\langle proof \rangle$

lemma *fba-UNIV*: $(UNIV :: {}'a\ set\ set) \in fba$
$\langle proof \rangle$

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 $'a\ formula = fba :: {}'a\ set\ set\ set$
$\langle proof \rangle$

definition $var :: {}'a \Rightarrow {}'a\ formula$
where $var\ i = Abs\text{-}formula\ \{A.\ i \in A\}$

lemma *Rep-formula-var*: $Rep\text{-}formula\ (var\ i) = \{A.\ i \in A\}$
$\langle proof \rangle$

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 $formula :: (type)\ boolean\text{-}algebra$
begin

definition
$x \sqcap y = Abs\text{-}formula\ (Rep\text{-}formula\ x \cap Rep\text{-}formula\ y)$

definition
$x \sqcup y = Abs\text{-}formula\ (Rep\text{-}formula\ x \cup Rep\text{-}formula\ y)$

definition
$\top = Abs\text{-}formula\ UNIV$

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

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

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

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

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

lemma *Rep-formula-inf*:
$\textit{Rep-formula}\ (x \sqcap y) = \textit{Rep-formula}\ x \cap \textit{Rep-formula}\ y$
$\langle \textit{proof} \rangle$

lemma *Rep-formula-sup*:
$\textit{Rep-formula}\ (x \sqcup y) = \textit{Rep-formula}\ x \cup \textit{Rep-formula}\ y$
$\langle \textit{proof} \rangle$

lemma *Rep-formula-top*: $\textit{Rep-formula}\ \top = \textit{UNIV}$
$\langle \textit{proof} \rangle$

lemma *Rep-formula-bot*: $\textit{Rep-formula}\ \bot = \{\}$
$\langle \textit{proof} \rangle$

lemma *Rep-formula-compl*: $\textit{Rep-formula}\ (- x) = - \textit{Rep-formula}\ x$
$\langle \textit{proof} \rangle$

lemma *Rep-formula-diff*:
$\textit{Rep-formula}\ (x - y) = \textit{Rep-formula}\ x - \textit{Rep-formula}\ y$
$\langle \textit{proof} \rangle$

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 $\langle \textit{proof} \rangle$

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*]: $(\bot :: \textit{'a formula}) \neq \top$
$\langle \textit{proof} \rangle$

lemma *top-neq-bot-formula* [*simp*]: $(\top :: \textit{'a formula}) \neq \bot$
$\langle \textit{proof} \rangle$

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

lemma *var-le-var-simps* [*simp*]:
$\textit{var}\ i \leq \textit{var}\ j \longleftrightarrow i = j$
$\neg\ \textit{var}\ i \leq -\ \textit{var}\ j$

$\neg - \mathit{var}\ i \leq \mathit{var}\ j$
$\langle \mathit{proof} \rangle$

lemma *var-eq-var-simps* [*simp*]:
$\mathit{var}\ i = \mathit{var}\ j \longleftrightarrow i = j$
$\mathit{var}\ i \neq - \mathit{var}\ j$
$- \mathit{var}\ i \neq \mathit{var}\ j$
$\langle \mathit{proof} \rangle$

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 $P :: {}'a\ \mathit{formula} \Rightarrow \mathit{bool}$
assumes *1*: $\bigwedge i.\ P\ (\mathit{var}\ i)$
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 $P\ x$
$\langle \mathit{proof} \rangle$

1.3 If-then-else for Boolean algebras

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

definition
$\mathit{ifte} :: {}'a{::}\mathit{boolean\text{-}algebra} \Rightarrow {}'a \Rightarrow {}'a \Rightarrow {}'a$
where
$\mathit{ifte}\ a\ x\ y = (a \sqcap x) \sqcup (-\ a \sqcap y)$

lemma *ifte-top* [*simp*]: $\mathit{ifte}\ \top\ x\ y = x$
$\langle \mathit{proof} \rangle$

lemma *ifte-bot* [*simp*]: $\mathit{ifte}\ \bot\ x\ y = y$
$\langle \mathit{proof} \rangle$

lemma *ifte-same*: $\mathit{ifte}\ a\ x\ x = x$
$\langle \mathit{proof} \rangle$

lemma *compl-ifte*: $-\ \mathit{ifte}\ a\ x\ y = \mathit{ifte}\ a\ (-\ x)\ (-\ y)$
$\langle \mathit{proof} \rangle$

lemma *inf-ifte-distrib*:
$\mathit{ifte}\ x\ a\ b \sqcap \mathit{ifte}\ x\ c\ d = \mathit{ifte}\ x\ (a \sqcap c)\ (b \sqcap d)$
$\langle \mathit{proof} \rangle$

lemma *ifte-ifte-distrib*:
$\mathit{ifte}\ x\ (\mathit{ifte}\ y\ a\ b)\ (\mathit{ifte}\ y\ c\ d) = \mathit{ifte}\ y\ (\mathit{ifte}\ x\ a\ c)\ (\mathit{ifte}\ x\ b\ d)$
$\langle \mathit{proof} \rangle$

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

lemma *formulasI*:
  assumes $\bigwedge A\ B.\ \forall i{\in}S.\ i \in A \longleftrightarrow i \in B$
    $\Longrightarrow A \in \mathit{Rep\text{-}formula}\ x \longleftrightarrow B \in \mathit{Rep\text{-}formula}\ x$
  shows $x \in \mathit{formulas}\ S$
⟨*proof*⟩

lemma *formulasD*:
  assumes $x \in \mathit{formulas}\ S$
  assumes $\forall i{\in}S.\ i \in A \longleftrightarrow i \in B$
  shows $A \in \mathit{Rep\text{-}formula}\ x \longleftrightarrow B \in \mathit{Rep\text{-}formula}\ x$
⟨*proof*⟩

lemma *formulas-mono*: $S \subseteq T \Longrightarrow \mathit{formulas}\ S \subseteq \mathit{formulas}\ T$
⟨*proof*⟩

lemma *formulas-insert*: $x \in \mathit{formulas}\ S \Longrightarrow x \in \mathit{formulas}\ (\mathit{insert}\ a\ S)$
⟨*proof*⟩

lemma *formulas-var*: $i \in S \Longrightarrow \mathit{var}\ i \in \mathit{formulas}\ S$
⟨*proof*⟩

lemma *formulas-var-iff*: $\mathit{var}\ i \in \mathit{formulas}\ S \longleftrightarrow i \in S$
⟨*proof*⟩

lemma *formulas-bot*: $\bot \in \mathit{formulas}\ S$
⟨*proof*⟩

lemma *formulas-top*: $\top \in \mathit{formulas}\ S$
⟨*proof*⟩

lemma *formulas-compl*: $x \in \mathit{formulas}\ S \Longrightarrow -\ x \in \mathit{formulas}\ S$
⟨*proof*⟩

lemma *formulas-inf*:
  $x \in \mathit{formulas}\ S \Longrightarrow y \in \mathit{formulas}\ S \Longrightarrow x \sqcap y \in \mathit{formulas}\ S$
⟨*proof*⟩

lemma *formulas-sup*:

$x \in formulas\ S \Longrightarrow y \in formulas\ S \Longrightarrow x \sqcup y \in formulas\ S$
⟨*proof*⟩

lemma *formulas-diff*:
$x \in formulas\ S \Longrightarrow y \in formulas\ S \Longrightarrow x - y \in formulas\ S$
⟨*proof*⟩

lemma *formulas-ifte*:
$a \in formulas\ S \Longrightarrow x \in formulas\ S \Longrightarrow y \in formulas\ S \Longrightarrow$
$ifte\ a\ x\ y \in formulas\ S$
⟨*proof*⟩

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 $ifte\ (var\ i)\ x\ y = ifte\ (var\ i)\ x'\ y'$
assumes $i \notin S$
assumes $x \in formulas\ S$ **and** $x' \in formulas\ S$
assumes $y \in formulas\ S$ **and** $y' \in formulas\ S$
shows $x = x' \wedge y = y'$
⟨*proof*⟩

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 ::
$('a \Rightarrow 'b{::}boolean\text{-}algebra) \Rightarrow 'a\ set \Rightarrow 'a\ formula \Rightarrow 'b \Rightarrow bool$
for $f :: 'a \Rightarrow 'b{::}boolean\text{-}algebra$
where
bot: $hom\text{-}graph\ f\ \{\}\ bot\ bot$

| *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*⟩

lemma *hom-graph-insert-elim*:
assumes *hom-graph* f (*insert* $i\ S$) $x\ a$ **and** $i \notin S$
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* $f\ S\ z\ c$
⟨*proof*⟩

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 $f\ S\ x\ a \Longrightarrow x \in$ *formulas* S
⟨*proof*⟩

lemma *hom-graph-unique*:
hom-graph $f\ S\ x\ a \Longrightarrow$ *hom-graph* $f\ S\ x\ a' \Longrightarrow a = a'$
⟨*proof*⟩

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*⟩

lemma *hom-graph-finite-superset*:
assumes *hom-graph* $f\ S\ x\ a$ **and** *finite* T **and** $S \subseteq T$
shows *hom-graph* $f\ T\ x\ a$
⟨*proof*⟩

lemma *hom-graph-imp-finite*:
  hom-graph f S x a $\Longrightarrow$ *finite S*
⟨*proof*⟩

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*⟩

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*⟩

lemma *hom-graph-compl*:
  hom-graph f S x a $\Longrightarrow$ *hom-graph f S* $(-\ x)$ $(-\ a)$
⟨*proof*⟩

lemma *hom-graph-inf*:
  hom-graph f S x a $\Longrightarrow$ *hom-graph f S y b* $\Longrightarrow$
   hom-graph f S $(x \sqcap y)$ $(a \sqcap b)$
 ⟨*proof*⟩

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*⟩

lemma *hom-graph-exists*: $\exists\, a\ S.$ *hom-graph f S x a*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *hom-equality*:
  hom-graph f S x a $\Longrightarrow$ *hom f x* $= a$

⟨*proof*⟩

The *hom* function correctly implements its specification:

lemma *hom-var* [*simp*]: $hom\ f\ (var\ i) = f\ i$
⟨*proof*⟩

lemma *hom-bot* [*simp*]: $hom\ f\ \bot = \bot$
⟨*proof*⟩

lemma *hom-top* [*simp*]: $hom\ f\ \top = \top$
⟨*proof*⟩

lemma *hom-compl* [*simp*]: $hom\ f\ (-\ x) = -\ hom\ f\ x$
⟨*proof*⟩

lemma *hom-inf* [*simp*]: $hom\ f\ (x \sqcap y) = hom\ f\ x \sqcap hom\ f\ y$
⟨*proof*⟩

lemma *hom-sup* [*simp*]: $hom\ f\ (x \sqcup y) = hom\ f\ x \sqcup hom\ f\ y$
⟨*proof*⟩

lemma *hom-diff* [*simp*]: $hom\ f\ (x - y) = hom\ f\ x - hom\ f\ y$
⟨*proof*⟩

lemma *hom-ifte* [*simp*]:
$hom\ f\ (ifte\ x\ y\ z) = ifte\ (hom\ f\ x)\ (hom\ f\ y)\ (hom\ f\ z)$
⟨*proof*⟩

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*: $hom\ var\ x = x$
⟨*proof*⟩

lemma *hom-hom*: $hom\ f\ (hom\ g\ x) = hom\ (\lambda i.\ hom\ f\ (g\ i))\ x$
⟨*proof*⟩

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
$fmap :: ('a \Rightarrow 'b) \Rightarrow 'a\ formula \Rightarrow 'b\ formula$
where

$fmap\ f = hom\ (\lambda i.\ var\ (f\ i))$

lemma *fmap-var* [*simp*]: $fmap\ f\ (var\ i) = var\ (f\ i)$
⟨*proof*⟩

lemma *fmap-bot* [*simp*]: $fmap\ f\ \bot = \bot$
⟨*proof*⟩

lemma *fmap-top* [*simp*]: $fmap\ f\ \top = \top$
⟨*proof*⟩

lemma *fmap-compl* [*simp*]: $fmap\ f\ (-\ x) = -\ fmap\ f\ x$
⟨*proof*⟩

lemma *fmap-inf* [*simp*]: $fmap\ f\ (x \sqcap y) = fmap\ f\ x \sqcap fmap\ f\ y$
⟨*proof*⟩

lemma *fmap-sup* [*simp*]: $fmap\ f\ (x \sqcup y) = fmap\ f\ x \sqcup fmap\ f\ y$
⟨*proof*⟩

lemma *fmap-diff* [*simp*]: $fmap\ f\ (x - y) = fmap\ f\ x - fmap\ f\ y$
⟨*proof*⟩

lemma *fmap-ifte* [*simp*]:
$fmap\ f\ (ifte\ x\ y\ z) = ifte\ (fmap\ f\ x)\ (fmap\ f\ y)\ (fmap\ f\ z)$
⟨*proof*⟩

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*: $fmap\ (\lambda i.\ i)\ x = x$
⟨*proof*⟩

lemma *fmap-fmap*: $fmap\ f\ (fmap\ g\ x) = fmap\ (f \circ g)\ x$
⟨*proof*⟩

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