

# Free Boolean Algebra

Brian Huffman

September 13, 2023

## Abstract

This theory defines a type constructor representing the free Boolean algebra over a set of generators. Values of type  $(\alpha)formula$  represent propositional formulas with uninterpreted variables from type  $\alpha$ , 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  $fb$  contains representatives of formulas built from finite combinations of variables with negation and conjunction.

#### inductive-set

$fb :: 'a\ set\ set\ set$

#### where

$var: \{A. i \in A\} \in fb$

|  $Compl: S \in fb \implies \neg S \in fb$

|  $inter: S \in fb \implies T \in fb \implies S \cap T \in fb$

**lemma** *fb-Diff*:  $S \in fb \implies T \in fb \implies S - T \in fb$

*<proof>*

**lemma** *fb-union*:  $S \in fb \implies T \in fb \implies S \cup T \in fb$

*<proof>*

**lemma** *fb-empty*:  $(\{\} :: 'a \text{ set set}) \in \text{fba}$   
*<proof>*

**lemma** *fb-UNIV*:  $(\text{UNIV} :: 'a \text{ set set}) \in \text{fba}$   
*<proof>*

## 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*  
*<proof>*

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

**lemma** *Rep-formula-var*: *Rep-formula* (*var i*) =  $\{A. i \in A\}$   
*<proof>*

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

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

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

**definition**  
 $\perp = \text{Abs-formula } \{\}$

**definition**  
 $x \leq y \iff \text{Rep-formula } x \subseteq \text{Rep-formula } y$

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

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

**definition**

$$x - y = \text{Abs-formula } (\text{Rep-formula } x - \text{Rep-formula } y)$$
**lemma** *Rep-formula-inf*:
$$\text{Rep-formula } (x \sqcap y) = \text{Rep-formula } x \cap \text{Rep-formula } y$$

*<proof>*

**lemma** *Rep-formula-sup*:
$$\text{Rep-formula } (x \sqcup y) = \text{Rep-formula } x \cup \text{Rep-formula } y$$

*<proof>*

**lemma** *Rep-formula-top*:  $\text{Rep-formula } \top = \text{UNIV}$ 

*<proof>*

**lemma** *Rep-formula-bot*:  $\text{Rep-formula } \perp = \{\}$ 

*<proof>*

**lemma** *Rep-formula-compl*:  $\text{Rep-formula } (- x) = - \text{Rep-formula } x$ 

*<proof>*

**lemma** *Rep-formula-diff*:
$$\text{Rep-formula } (x - y) = \text{Rep-formula } x - \text{Rep-formula } y$$

*<proof>*

**lemmas** *eq-formula-iff* = *Rep-formula-inject* [*symmetric*]**lemmas** *Rep-formula-simps* =
$$\text{less-eq-formula-def } \text{less-formula-def } \text{eq-formula-iff}$$

$$\text{Rep-formula-sup } \text{Rep-formula-inf } \text{Rep-formula-top } \text{Rep-formula-bot}$$

$$\text{Rep-formula-compl } \text{Rep-formula-diff } \text{Rep-formula-var}$$
**instance** *<proof>***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*]:  $(\perp :: 'a \text{ formula}) \neq \top$ 

*<proof>*

**lemma** *top-neq-bot-formula* [*simp*]:  $(\top :: 'a \text{ formula}) \neq \perp$ 

*<proof>*

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

**lemma** *var-le-var-simps* [*simp*]:
$$\text{var } i \leq \text{var } j \iff i = j$$

$$\neg \text{var } i \leq - \text{var } j$$

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

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

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

### 1.3 If-then-else for Boolean algebras

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

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

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

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

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

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

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

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

## 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

*formulas* :: 'a set  $\Rightarrow$  'a formula set

### where

*formulas*  $S =$   
 $\{x. \forall A B. (\forall i \in S. i \in A \longleftrightarrow i \in B) \longrightarrow$   
 $A \in \text{Rep-formula } x \longleftrightarrow B \in \text{Rep-formula } x\}$

### lemma *formulasI*:

**assumes**  $\bigwedge A B. \forall i \in S. i \in A \longleftrightarrow i \in B$   
 $\implies A \in \text{Rep-formula } x \longleftrightarrow B \in \text{Rep-formula } x$   
**shows**  $x \in \text{formulas } S$   
(*proof*)

### lemma *formulasD*:

**assumes**  $x \in \text{formulas } S$   
**assumes**  $\forall i \in S. i \in A \longleftrightarrow i \in B$   
**shows**  $A \in \text{Rep-formula } x \longleftrightarrow B \in \text{Rep-formula } x$   
(*proof*)

### lemma *formulas-mono*: $S \subseteq T \implies \text{formulas } S \subseteq \text{formulas } T$

(*proof*)

### lemma *formulas-insert*: $x \in \text{formulas } S \implies x \in \text{formulas } (\text{insert } a \ S)$

(*proof*)

### lemma *formulas-var*: $i \in S \implies \text{var } i \in \text{formulas } S$

(*proof*)

### lemma *formulas-var-iff*: $\text{var } i \in \text{formulas } S \longleftrightarrow i \in S$

(*proof*)

### lemma *formulas-bot*: $\perp \in \text{formulas } S$

(*proof*)

### lemma *formulas-top*: $\top \in \text{formulas } S$

(*proof*)

### lemma *formulas-compl*: $x \in \text{formulas } S \implies \neg x \in \text{formulas } S$

(*proof*)

### lemma *formulas-inf*:

$x \in \text{formulas } S \implies y \in \text{formulas } S \implies x \sqcap y \in \text{formulas } S$   
(*proof*)

### lemma *formulas-sup*:

$x \in \text{formulas } S \implies y \in \text{formulas } S \implies x \sqcup y \in \text{formulas } S$   
 ⟨proof⟩

**lemma** *formulas-diff*:

$x \in \text{formulas } S \implies y \in \text{formulas } S \implies x - y \in \text{formulas } S$   
 ⟨proof⟩

**lemma** *formulas-ifte*:

$a \in \text{formulas } S \implies x \in \text{formulas } S \implies y \in \text{formulas } S \implies$   
 $\text{ifte } a \ x \ y \in \text{formulas } S$   
 ⟨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**  $\text{ifte } (\text{var } i) \ x \ y = \text{ifte } (\text{var } i) \ x' \ y'$   
**assumes**  $i \notin S$   
**assumes**  $x \in \text{formulas } S$  **and**  $x' \in \text{formulas } S$   
**assumes**  $y \in \text{formulas } S$  **and**  $y' \in \text{formulas } S$   
**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  $\text{hom } f \ (\text{var } i) = f \ i$  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* ::  
 ( $'a \Rightarrow 'b::\text{boolean-algebra}$ )  $\Rightarrow 'a \ \text{set} \Rightarrow 'a \ \text{formula} \Rightarrow 'b \Rightarrow \text{bool}$   
**for**  $f :: 'a \Rightarrow 'b::\text{boolean-algebra}$

**where**

*bot*:  $\text{hom-graph } f \ \{\} \ \text{bot } \text{bot}$

| *top*:  $\text{hom-graph } f \{ \} \text{ top top}$   
| *ifte*:  $i \notin S \implies \text{hom-graph } f S x a \implies \text{hom-graph } f S y b \implies$   
 $\text{hom-graph } f (\text{insert } i S) (\text{ifte } (\text{var } i) x y) (\text{ifte } (f i) a b)$

The next two lemmas establish a stronger elimination rule for assumptions of the form  $\text{hom-graph } f (\text{insert } i S) x a$ . 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*:

$\text{hom-graph } f S x a \implies k \in S \implies \exists y z b c.$   
 $x = \text{ifte } (\text{var } k) y z \wedge a = \text{ifte } (f k) b c \wedge$   
 $\text{hom-graph } f (S - \{k\}) y b \wedge \text{hom-graph } f (S - \{k\}) z c$   
⟨*proof*⟩

**lemma** *hom-graph-insert-elim*:

**assumes**  $\text{hom-graph } f (\text{insert } i S) x a$  **and**  $i \notin S$   
**obtains**  $y z b c$   
**where**  $x = \text{ifte } (\text{var } i) y z$   
**and**  $a = \text{ifte } (f i) b c$   
**and**  $\text{hom-graph } f S y b$   
**and**  $\text{hom-graph } f S z c$   
⟨*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  $\text{hom-graph } f S$  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*:

$\text{hom-graph } f S x a \implies x \in \text{formulas } S$   
⟨*proof*⟩

**lemma** *hom-graph-unique*:

$\text{hom-graph } f S x a \implies \text{hom-graph } f S x a' \implies a = a'$   
⟨*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**  $\text{hom-graph } f S x a$   
**shows**  $\text{hom-graph } f (\text{insert } i S) x a$   
⟨*proof*⟩

**lemma** *hom-graph-finite-superset*:

**assumes**  $\text{hom-graph } f S x a$  **and** *finite*  $T$  **and**  $S \subseteq T$   
**shows**  $\text{hom-graph } f T x a$   
⟨*proof*⟩

**lemma** *hom-graph-imp-finite*:  
 $hom\text{-}graph\ f\ S\ x\ a \implies finite\ S$   
 ⟨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**  $hom\text{-}graph\ f\ S\ x\ a$  **and**  $hom\text{-}graph\ f\ T\ x\ a'$   
**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*:  $hom\text{-}graph\ f\ \{i\}\ (var\ i)\ (f\ i)$   
 ⟨proof⟩

**lemma** *hom-graph-compl*:  
 $hom\text{-}graph\ f\ S\ x\ a \implies hom\text{-}graph\ f\ S\ (-\ x)\ (-\ a)$   
 ⟨proof⟩

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

**lemma** *hom-graph-union-inf*:  
**assumes**  $hom\text{-}graph\ f\ S\ x\ a$  **and**  $hom\text{-}graph\ f\ T\ y\ b$   
**shows**  $hom\text{-}graph\ f\ (S\ \cup\ T)\ (x\ \sqcap\ y)\ (a\ \sqcap\ b)$   
 ⟨proof⟩

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

**lemma** *hom-graph-hom*:  $\exists S.\ hom\text{-}graph\ f\ S\ x\ (hom\ f\ x)$   
 ⟨proof⟩

**lemma** *hom-equality*:  
 $hom\text{-}graph\ f\ S\ x\ a \implies hom\ f\ x = a$



$\langle proof \rangle$

The *hom* function correctly implements its specification:

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

**lemma** *hom-bot* [*simp*]:  $hom\ f\ \perp = \perp$   
 $\langle proof \rangle$

**lemma** *hom-top* [*simp*]:  $hom\ f\ \top = \top$   
 $\langle proof \rangle$

**lemma** *hom-compl* [*simp*]:  $hom\ f\ (\neg\ x) = \neg\ hom\ f\ x$   
 $\langle proof \rangle$

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

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

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

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

**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*:  $hom\ var\ x = x$   
 $\langle proof \rangle$

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

## 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*]:  $fmap\ f\ \perp = \perp$   
*<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**