

Boolean Expression Checkers

Tobias Nipkow

April 19, 2020

Abstract

This entry provides executable checkers for the following properties of boolean expressions: satisfiability, tautology and equivalence. Internally, the checkers operate on binary decision trees and are reasonably efficient (for purely functional algorithms).

Contents

1	Tautology (etc) Checking via Binary Decision Trees	2
1.1	Binary Decision Trees	2
1.1.1	Environment	2
1.2	Recursive Tautology Checker	2
1.3	Reduced Binary Decision Trees	3
1.3.1	Normalisation	3
1.3.2	Functional Correctness Proof	4
1.3.3	Reduced If-Expressions	4
1.3.4	Checkers Based on Reduced Binary Decision Trees	4
1.4	Boolean Expressions	6
2	Tweaks for <i>AList-Mapping.Mapping</i>	7
3	Example	8
3.1	Indirect Translation using the Boolean Expression Interface	8
3.2	Direct Translation into Reduced Binary Decision Trees	9
3.3	Test: Pigeonhole Formulas	9

```
theory Boolean-Expression-Checkers
  imports Main HOL-Library.Mapping
begin
```

1 Tautology (etc) Checking via Binary Decision Trees

1.1 Binary Decision Trees

datatype 'a ifex = Trueif | Falseif | IF 'a 'a ifex 'a ifex

fun val-ifex :: 'a ifex \Rightarrow ('a \Rightarrow bool) \Rightarrow bool

where

val-ifex Trueif s = True
| val-ifex Falseif s = False
| val-ifex (IF n t1 t2) s = (if s n then val-ifex t1 s else val-ifex t2 s)

1.1.1 Environment

Environments are substitutions of values for variables:

type-synonym 'a env-bool = ('a, bool) mapping

definition agree :: ('a \Rightarrow bool) \Rightarrow 'a env-bool \Rightarrow bool

where

agree s env = (\forall x b. Mapping.lookup env x = Some b \longrightarrow s x = b)

lemma agree-Nil:

agree s Mapping.empty
<proof>

lemma lookup-update-unfold:

Mapping.lookup (Mapping.update k v m) k' = (if k = k' then Some v else Mapping.lookup m k')
<proof>

lemma agree-Cons:

$x \notin$ Mapping.keys env \Longrightarrow agree s (Mapping.update x b env) = ((if b then s x else \neg s x) \wedge agree s env)
<proof>

lemma agreeDT:

agree s env \Longrightarrow Mapping.lookup env x = Some True \Longrightarrow s x
<proof>

lemma agreeDF:

agree s env \Longrightarrow Mapping.lookup env x = Some False \Longrightarrow \neg s x
<proof>

1.2 Recursive Tautology Checker

Provided for completeness. However, it is recommend to use the checkers based on reduced trees.

fun taut-test-rec :: 'a ifex \Rightarrow 'a env-bool \Rightarrow bool

where

taut-test-rec Trueif env = True
| *taut-test-rec Falseif env = False*
| *taut-test-rec (IF x t1 t2) env = (case Mapping.lookup env x of*
 Some b ⇒ taut-test-rec (if b then t1 else t2) env |
 None ⇒ taut-test-rec t1 (Mapping.update x True env) ∧ taut-test-rec t2 (Mapping.update
x False env))

lemma *taut-test-rec*:

taut-test-rec t env = (∀ s. agree s env → val-ifex t s)
<proof>

definition *taut-test-ifex* :: 'a ifex ⇒ bool

where

taut-test-ifex t = taut-test-rec t Mapping.empty

corollary *taut-test-ifex*:

taut-test-ifex t = (∀ s. val-ifex t s)
<proof>

1.3 Reduced Binary Decision Trees

1.3.1 Normalisation

A normalisation avoiding duplicate variables and collapsing *if x then t else t* to *t*.

definition *mkIF* :: 'a ⇒ 'a ifex ⇒ 'a ifex ⇒ 'a ifex

where

mkIF x t1 t2 = (if t1=t2 then t1 else IF x t1 t2)

fun *reduce* :: 'a env-bool ⇒ 'a ifex ⇒ 'a ifex

where

reduce env (IF x t1 t2) = (case Mapping.lookup env x of
 None ⇒ mkIF x (reduce (Mapping.update x True env) t1) (reduce (Mapping.update
x False env) t2) |
 Some b ⇒ reduce env (if b then t1 else t2))
| *reduce - t = t*

primrec *normif* :: 'a env-bool ⇒ 'a ifex ⇒ 'a ifex ⇒ 'a ifex ⇒ 'a ifex

where

normif env Trueif t1 t2 = reduce env t1
| *normif env Falseif t1 t2 = reduce env t2*
| *normif env (IF x t1 t2) t3 t4 =*
 (case Mapping.lookup env x of
 None ⇒ mkIF x (normif (Mapping.update x True env) t1 t3 t4) (normif
(Mapping.update x False env) t2 t3 t4) |
 Some b ⇒ if b then normif env t1 t3 t4 else normif env t2 t3 t4)

1.3.2 Functional Correctness Proof

lemma *val-mkIF*:

$val\text{-ifex } (mkIF\ x\ t1\ t2)\ s = val\text{-ifex } (IF\ x\ t1\ t2)\ s$
<proof>

theorem *val-reduce*:

$agree\ s\ env \implies val\text{-ifex } (reduce\ env\ t)\ s = val\text{-ifex } t\ s$
<proof>

lemma *val-normif*:

$agree\ s\ env \implies val\text{-ifex } (normif\ env\ t\ t1\ t2)\ s = val\text{-ifex } (if\ val\text{-ifex } t\ s\ then\ t1\ else\ t2)\ s$
<proof>

1.3.3 Reduced If-Expressions

An expression reduced iff no variable appears twice on any branch and there is no subexpression $IF\ x\ t$.

fun *reduced* :: 'a ifex \Rightarrow 'a set \Rightarrow bool **where**

$reduced\ (IF\ x\ t1\ t2)\ X =$
 $(x \notin X \wedge t1 \neq t2 \wedge reduced\ t1\ (insert\ x\ X) \wedge reduced\ t2\ (insert\ x\ X)) \mid$
 $reduced\ - = True$

lemma *reduced-antimono*:

$X \subseteq Y \implies reduced\ t\ Y \implies reduced\ t\ X$
<proof>

lemma *reduced-mkIF*:

$x \notin X \implies reduced\ t1\ (insert\ x\ X) \implies reduced\ t2\ (insert\ x\ X) \implies reduced\ (mkIF\ x\ t1\ t2)\ X$
<proof>

lemma *reduced-reduce*:

$reduced\ (reduce\ env\ t)\ (Mapping.keys\ env)$
<proof>

lemma *reduced-normif*:

$reduced\ (normif\ env\ t\ t1\ t2)\ (Mapping.keys\ env)$
<proof>

1.3.4 Checkers Based on Reduced Binary Decision Trees

The checkers are parameterized over the translation function to binary decision trees. They rely on the fact that *ifex-of* produces reduced trees

definition *taut-test* :: ('a \Rightarrow 'b ifex) \Rightarrow 'a \Rightarrow bool

where

$taut\text{-test } ifex\text{-of } b = (ifex\text{-of } b = True\text{if})$

definition *sat-test* :: ('a ⇒ 'b ifex) ⇒ 'a ⇒ bool

where

sat-test ifex-of b = (ifex-of b ≠ Falseif)

definition *impl-test* :: ('a ⇒ 'b ifex) ⇒ 'a ⇒ 'a ⇒ bool

where

impl-test ifex-of b1 b2 = (normif Mapping.empty (ifex-of b1) (ifex-of b2) Trueif = Trueif)

definition *equiv-test* :: ('a ⇒ 'b ifex) ⇒ 'a ⇒ 'a ⇒ bool

where

equiv-test ifex-of b1 b2 = (let t1 = ifex-of b1; t2 = ifex-of b2
in Trueif = normif Mapping.empty t1 t2 (normif Mapping.empty t2 Falseif
Trueif))

locale *reduced-bdt-checkers* =

fixes

ifex-of :: 'b ⇒ 'a ifex

fixes

val :: 'b ⇒ ('a ⇒ bool) ⇒ bool

assumes

val-ifex: *val-ifex* (ifex-of b) s = *val b s*

assumes

reduced-ifex: *reduced* (ifex-of b) {}

begin

Proof that reduced if-expressions are *Trueif*, *Falseif* or can evaluate to both *True* and *False*.

lemma *same-val-if-reduced*:

reduced t X ⇒ ∀ x. x ∉ X ⇒ s1 x = s2 x ⇒ *val-ifex t s1* = *val-ifex t s2*
<proof>

lemma *reduced-IF-depends*:

[[*reduced t X*; t ≠ *Trueif*; t ≠ *Falseif*]] ⇒ ∃ s1 s2. *val-ifex t s1* ≠ *val-ifex t s2*
<proof>

corollary *taut-test*:

taut-test ifex-of b = (∀ s. *val b s*)
<proof>

corollary *sat-test*:

sat-test ifex-of b = (∃ s. *val b s*)
<proof>

corollary *impl-test*:

impl-test ifex-of b1 b2 = (∀ s. *val b1 s* ⇒ *val b2 s*)
<proof>

corollary *equiv-test*:

```

    equiv-test ifex-of b1 b2 = ( $\forall s. \text{val } b1 \text{ } s = \text{val } b2 \text{ } s$ )
  <proof>

```

end

1.4 Boolean Expressions

This is the simplified interface to the tautology checker. If you have your own type of Boolean expressions you can either define your own translation to reduced binary decision trees or you can just translate into this type.

```

datatype 'a bool-expr =
  Const-bool-expr bool |
  Atom-bool-expr 'a |
  Neg-bool-expr 'a bool-expr |
  And-bool-expr 'a bool-expr 'a bool-expr |
  Or-bool-expr 'a bool-expr 'a bool-expr |
  Imp-bool-expr 'a bool-expr 'a bool-expr |
  Iff-bool-expr 'a bool-expr 'a bool-expr

```

```

primrec val-bool-expr :: 'a bool-expr  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool where
val-bool-expr (Const-bool-expr b) s = b |
val-bool-expr (Atom-bool-expr x) s = s x |
val-bool-expr (Neg-bool-expr b) s = ( $\neg$  val-bool-expr b s) |
val-bool-expr (And-bool-expr b1 b2) s = (val-bool-expr b1 s  $\wedge$  val-bool-expr b2 s) |
val-bool-expr (Or-bool-expr b1 b2) s = (val-bool-expr b1 s  $\vee$  val-bool-expr b2 s) |
val-bool-expr (Imp-bool-expr b1 b2) s = (val-bool-expr b1 s  $\longrightarrow$  val-bool-expr b2 s)
|
val-bool-expr (Iff-bool-expr b1 b2) s = (val-bool-expr b1 s = val-bool-expr b2 s)

```

```

fun ifex-of :: 'a bool-expr  $\Rightarrow$  'a ifex where
ifex-of (Const-bool-expr b) = (if b then Trueif else Falseif) |
ifex-of (Atom-bool-expr x) = IF x Trueif Falseif |
ifex-of (Neg-bool-expr b) = normif Mapping.empty (ifex-of b) Falseif Trueif |
ifex-of (And-bool-expr b1 b2) = normif Mapping.empty (ifex-of b1) (ifex-of b2)
Falseif |
ifex-of (Or-bool-expr b1 b2) = normif Mapping.empty (ifex-of b1) Trueif (ifex-of
b2) |
ifex-of (Imp-bool-expr b1 b2) = normif Mapping.empty (ifex-of b1) (ifex-of b2)
Trueif |
ifex-of (Iff-bool-expr b1 b2) = (let t1 = ifex-of b1; t2 = ifex-of b2 in
normif Mapping.empty t1 t2 (normif Mapping.empty t2 Falseif Trueif))

```

```

theorem val-ifex:
  val-ifex (ifex-of b) s = val-bool-expr b s
  <proof>

```

```

theorem reduced-ifex:
  reduced (ifex-of b) {}
  <proof>

```

definition *bool-taut-test* \equiv *taut-test ifex-of*
definition *bool-sat-test* \equiv *sat-test ifex-of*
definition *bool-impl-test* \equiv *impl-test ifex-of*
definition *bool-equiv-test* \equiv *equiv-test ifex-of*

lemma *bool-tests*:

bool-taut-test $b = (\forall s. \text{val-bool-expr } b \ s) \ (\mathbf{is} \ ?t1)$
bool-sat-test $b = (\exists s. \text{val-bool-expr } b \ s) \ (\mathbf{is} \ ?t2)$
bool-impl-test $b1 \ b2 = (\forall s. \text{val-bool-expr } b1 \ s \longrightarrow \text{val-bool-expr } b2 \ s) \ (\mathbf{is} \ ?t3)$
bool-equiv-test $b1 \ b2 = (\forall s. \text{val-bool-expr } b1 \ s \longleftrightarrow \text{val-bool-expr } b2 \ s) \ (\mathbf{is} \ ?t4)$
 $\langle \text{proof} \rangle$

end

theory *Boolean-Expression-Checkers-AList-Mapping*
imports *Main HOL-Library.ALlist-Mapping Boolean-Expression-Checkers*
begin

2 Tweaks for *AList-Mapping.Mapping*

— If mappings are implemented by *AList-Mapping.Mapping*, the functions *reduce* and *normif* search for *x* twice. The following code equations remove this redundant operation

lemma *AList-Mapping-update*:

map-of $m \ k = \text{None} \implies \text{Mapping.update } k \ v \ (\text{AList-Mapping.Mapping } xs) =$
 $\text{AList-Mapping.Mapping } ((k,v)\#xs)$
 $\langle \text{proof} \rangle$

fun *reduce-alist* $:: ('a * \text{bool}) \ \text{list} \Rightarrow 'a \ \text{ifex} \Rightarrow 'a \ \text{ifex}$

where

reduce-alist $xs \ (\text{IF } x \ t1 \ t2) = (\text{case } \text{map-of } xs \ x \ \text{of}$
 $\text{None} \Rightarrow \text{mkIF } x \ (\text{reduce-alist } ((x, \text{True})\#xs) \ t1) \ (\text{reduce-alist } ((x, \text{False})\#xs)$
 $t2) \ |$
 $\text{Some } b \Rightarrow \text{reduce-alist } xs \ (\text{if } b \ \text{then } t1 \ \text{else } t2))$
 $| \ \text{reduce-alist } - \ t = \ t$

primrec *normif-alist* $:: ('a * \text{bool}) \ \text{list} \Rightarrow 'a \ \text{ifex} \Rightarrow 'a \ \text{ifex} \Rightarrow 'a \ \text{ifex} \Rightarrow 'a \ \text{ifex}$

where

normif-alist $xs \ \text{Trueif } t1 \ t2 = \text{reduce-alist } xs \ t1$
 $| \ \text{normif-alist } xs \ \text{Falseif } t1 \ t2 = \text{reduce-alist } xs \ t2$
 $| \ \text{normif-alist } xs \ (\text{IF } x \ t1 \ t2) \ t3 \ t4 = (\text{case } \text{map-of } xs \ x \ \text{of}$
 $\text{None} \Rightarrow \text{mkIF } x \ (\text{normif-alist } ((x, \text{True})\#xs) \ t1 \ t3 \ t4) \ (\text{normif-alist } ((x,$
 $\text{False})\#xs) \ t2 \ t3 \ t4) \ |$
 $\text{Some } b \Rightarrow \text{if } b \ \text{then } \text{normif-alist } xs \ t1 \ t3 \ t4 \ \text{else } \text{normif-alist } xs \ t2 \ t3 \ t4)$

lemma *reduce-alist-code* [*code*, *code-unfold*]:

```

reduce (AList-Mapping.Mapping xs) t = reduce-alist xs t
⟨proof⟩

```

```

lemma normif-alist-code [code, code-unfold]:
normif (AList-Mapping.Mapping xs) t = normif-alist xs t
⟨proof⟩

```

```

lemmas empty-Mapping [code-unfold]

```

```

end

```

```

theory Boolean-Expression-Example

```

```

imports Boolean-Expression-Checkers Boolean-Expression-Checkers-AList-Mapping
begin

```

3 Example

Example usage of checkers. We have our own type of Boolean expressions with its own evaluation function:

```

datatype 'a bexp =
  Const bool |
  Atom 'a |
  Neg 'a bexp |
  And 'a bexp 'a bexp

```

```

fun bval where
bval (Const b) s = b |
bval (Atom a) s = s a |
bval (Neg b) s = (¬ bval b s) |
bval (And b1 b2) s = (bval b1 s ∧ bval b2 s)

```

3.1 Indirect Translation using the Boolean Expression Interface

Now we translate into **datatype** 'a bool-expr = Const-bool-expr bool | Atom-bool-expr 'a | Neg-bool-expr ('a bool-expr) | And-bool-expr ('a bool-expr) ('a bool-expr) | Or-bool-expr ('a bool-expr) ('a bool-expr) | Imp-bool-expr ('a bool-expr) ('a bool-expr) | Iff-bool-expr ('a bool-expr) ('a bool-expr) provided by the checkers interface and show that the semantics remains the same:

```

fun bool-expr-of-bexp :: 'a bexp ⇒ 'a bool-expr
where
  bool-expr-of-bexp (Const b) = Const-bool-expr b
| bool-expr-of-bexp (Atom a) = Atom-bool-expr a
| bool-expr-of-bexp (Neg b) = Neg-bool-expr (bool-expr-of-bexp b)
| bool-expr-of-bexp (And b1 b2) = And-bool-expr (bool-expr-of-bexp b1) (bool-expr-of-bexp b2)

```

```

lemma val-preservation:

```

val-bool-expr (*bool-expr-of-bexp* *b*) *s* = *bval* *b* *s*
 ⟨*proof*⟩

definition *my-taut-test-bool* = *bool-taut-test* *o* *bool-expr-of-bexp*

corollary *my-taut-test*:

my-taut-test-bool *b* = (\forall *s*. *bval* *b* *s*)
 ⟨*proof*⟩

3.2 Direct Translation into Reduced Binary Decision Trees

Now we translate into a reduced binary decision tree, show that the semantics remains the same and the tree is reduced:

fun *ifex-of* :: '*a* *bexp* \Rightarrow '*a* *ifex*

where

ifex-of (*Const* *b*) = (*if* *b* *then* *Trueif* *else* *Falseif*)
 | *ifex-of* (*Atom* *a*) = *IF* *a* *Trueif* *Falseif*
 | *ifex-of* (*Neg* *b*) = *normif* *Mapping.empty* (*ifex-of* *b*) *Falseif* *Trueif*
 | *ifex-of* (*And* *b1* *b2*) = *normif* *Mapping.empty* (*ifex-of* *b1*) (*ifex-of* *b2*) *Falseif*

lemma *val-ifex*:

val-ifex (*ifex-of* *b*) *s* = *bval* *b* *s*
 ⟨*proof*⟩

theorem *reduced-ifex*:

reduced (*ifex-of* *b*) {}
 ⟨*proof*⟩

definition *my-taut-test-ifex* = *taut-test* *ifex-of*

corollary *my-taut-test-ifex*:

my-taut-test-ifex *b* = (\forall *s*. *bval* *b* *s*)
 ⟨*proof*⟩

3.3 Test: Pigeonhole Formulas

definition *Or* *b1* *b2* == *Neg* (*And* (*Neg* *b1*) (*Neg* *b2*))

definition *ors* = *foldl* *Or* (*Const* *False*)

definition *ands* = *foldl* *And* (*Const* *True*)

definition *pc* *n* = *ands*[*ors*[*Atom*(*i*,*j*). *j* <- [1..*n*+1]]. *i* <- [1..*n*+2]]

definition *nc* *n* = *ands*[*Or* (*Neg*(*Atom*(*i*,*k*))) (*Neg*(*Atom*(*j*,*k*))). *k* <- [1..*n*+1],
i <- [1..*n*+1], *j* <- [*i*+1..*n*+2]]

definition *php* *n* = *Neg*(*And* (*pc* *n*) (*nc* *n*))

Takes about 5 secs each; with 7 instead of 6 it takes about 4 mins (2015).

lemma *my-taut-test-bool* (*php* 6)

⟨*proof*⟩

lemma *my-taut-test-ifex* (*php 6*)
 <proof>
end