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

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 ⇒ ('a ⇒ bool) ⇒ 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 ⇒ bool) ⇒ 'a env-bool ⇒ bool*
where
agree s env = (∀ x b. Mapping.lookup env x = Some b ⟶ s x = b)

lemma *agree-Nil*:
agree s Mapping.empty
by (*simp add*: *agree-def lookup-empty*)

lemma *lookup-update-unfold*:
Mapping.lookup (Mapping.update k v m) k' = (if k = k' then Some v else Mapping.lookup m k')
using *lookup-update lookup-update-neq* **by** *metis*

lemma *agree-Cons*:
x ∉ Mapping.keys env ⟹ agree s (Mapping.update x b env) = ((if b then s x else ¬ s x) ∧ agree s env)
by (*simp add*: *agree-def lookup-update-unfold*; *unfold keys-is-none-rep lookup-update-unfold Option.is-none-def*; *blast*)

lemma *agreeDT*:
agree s env ⟹ Mapping.lookup env x = Some True ⟹ s x
by (*simp add*: *agree-def*)

lemma *agreeDF*:
agree s env ⟹ Mapping.lookup env x = Some False ⟹ ¬s x
by (*auto simp add*: *agree-def*)

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* ⇒ *'a env-bool* ⇒ *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 (*induction t arbitrary*: *env*)
case *Falseif*
have *agree* (λ*x*. *the* (*Mapping.lookup env x*)) *env*
by (*auto simp*: *agree-def*)
thus *?case*
by *auto*
next
case (*IF x t1 t2*)
thus *?case*
proof (*cases Mapping.lookup env x*)
case *None*
with *IF* **show** *?thesis*
by *simp* (*metis is-none-simps*(*1*) *agree-Cons keys-is-none-rep*)
qed (*simp add*: *agree-def*)
qed *simp*

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*)
by (*auto simp*: *taut-test-ifex-def taut-test-rec agree-Nil*)

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-ifex (mkIF x t1 t2) s = val-ifex (IF x t1 t2) s
  by (auto simp: mkIF-def Let-def)

theorem val-reduce:
  agree s env ⟹ val-ifex (reduce env t) s = val-ifex t s
  by (induction t arbitrary: s env)
    (auto simp: map-of-eq-None-iff val-mkIF agree-Cons Let-def keys-is-none-rep
        dest: agreeDT agreeDF split: option.splits)

lemma val-normif:
  agree s env ⟹ val-ifex (normif env t t1 t2) s = val-ifex (if val-ifex t s then t1
else t2) s
  by (induct t arbitrary: t1 t2 s env)
    (auto simp: val-reduce val-mkIF agree-Cons map-of-eq-None-iff keys-is-none-rep
        dest: agreeDT agreeDF split: option.splits)
```

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 t*.

```
fun reduced :: 'a ifex ⇒ 'a set ⇒ bool where
reduced (IF x t1 t2) X =
  (x ∉ X ∧ t1 ≠ t2 ∧ reduced t1 (insert x X) ∧ reduced t2 (insert x X)) |
reduced - - = True

lemma reduced-antimono:
  X ⊆ Y ⟹ reduced t Y ⟹ reduced t X
  by (induction t arbitrary: X Y)
    (auto, (metis insert-mono)+)

lemma reduced-mkIF:
```

$x \notin X \Longrightarrow$ *reduced t1* (*insert x X*) $\Longrightarrow$ *reduced t2* (*insert x X*) $\Longrightarrow$ *reduced* (*mkIF x t1 t2*) *X*
by (*auto simp*: *mkIF-def intro*:*reduced-antimono*)

lemma *reduced-reduce*:
reduced (*reduce env t*) (*Mapping.keys env*)
proof(*induction t arbitrary*: *env*)
case (*IF x t1 t2*)
thus *?case*
using *IF.IH*(*1*) *IF.IH*(*2*)
apply (*auto simp*: *map-of-eq-None-iff image-iff reduced-mkIF split*: *option.split*)

by (*metis is-none-code*(*1*) *keys-is-none-rep keys-update reduced-mkIF*)
qed *auto*

lemma *reduced-normif*:
reduced (*normif env t t1 t2*) (*Mapping.keys env*)
proof(*induction t arbitrary*: *t1 t2 env*)
case (*IF x s1 s2*)
thus *?case* **using** *IF.IH*
apply (*auto simp*: *reduced-mkIF map-of-eq-None-iff split*: *option.split*)
by (*metis is-none-code*(*1*) *keys-is-none-rep keys-update reduced-mkIF*)
qed (*auto simp*: *reduced-reduce*)

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* ⇒ '*b ifex*) ⇒ '*a* ⇒ *bool*
where
taut-test ifex-of b = (*ifex-of b* = *Trueif*)

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*
by (*induction t arbitrary*: *X*) *auto*

lemma *reduced-IF-depends*:
⟦ *reduced t X*; *t* ≠ *Trueif*; *t* ≠ *Falseif* ⟧ ⟹ ∃ *s1 s2*. *val-ifex t s1* ≠ *val-ifex t s2*
proof(*induction t arbitrary*: *X*)
case (*IF x t1 t2*)
let *?t* = *IF x t1 t2*
have *1*: *reduced t1* (*insert x X*) **using** *IF.prems*(*1*) **by** *simp*
have *2*: *reduced t2* (*insert x X*) **using** *IF.prems*(*1*) **by** *simp*
show *?case*
proof(*cases t1*)
case [*simp*]: *Trueif*
show *?thesis*
proof (*cases t2*)
case *Trueif* **thus** *?thesis* **using** *IF.prems*(*1*) **by** *simp*
next
case *Falseif*
hence *val-ifex ?t* (λ-. *True*) ≠ *val-ifex ?t* (λ-. *False*) **by** *simp*
thus *?thesis* **by** *blast*
next
case *IF*
then obtain *s1 s2* **where** *val-ifex t2 s1* ≠ *val-ifex t2 s2*
using *IF.IH*(*2*)[*OF 2*] *IF.prems*(*1*) **by** *auto*
hence *val-ifex ?t* (*s1*(*x*:=*False*)) ≠ *val-ifex ?t* (*s2*(*x*:=*False*))
using *same-val-if-reduced*[*OF 2*, *of s1*(*x*:=*False*) *s1*]
same-val-if-reduced[*OF 2*, *of s2*(*x*:=*False*) *s2*] **by** *simp*
thus *?thesis* **by** *blast*
qed
next
case [*simp*]: *Falseif*
show *?thesis*
proof (*cases t2*)
case *Falseif* **thus** *?thesis* **using** *IF.prems*(*1*) **by** *simp*
next
case *Trueif*
hence *val-ifex ?t* (λ-. *True*) ≠ *val-ifex ?t* (λ-. *False*) **by** *simp*

```
    thus ?thesis by blast
  next
    case IF
    then obtain s1 s2 where val-ifex t2 s1 ≠ val-ifex t2 s2
      using IF.IH(2)[OF 2] IF.prems(1) by auto
    hence val-ifex ?t (s1(x:=False)) ≠ val-ifex ?t (s2(x:=False))
      using same-val-if-reduced[OF 2, of s1(x:=False) s1]
        same-val-if-reduced[OF 2, of s2(x:=False) s2] by simp
    thus ?thesis by blast
  qed
 next
  case IF
  then obtain s1 s2 where val-ifex t1 s1 ≠ val-ifex t1 s2
    using IF.IH(1)[OF 1] IF.prems(1) by auto
  hence val-ifex ?t (s1(x:=True)) ≠ val-ifex ?t (s2(x:=True))
    using same-val-if-reduced[OF 1, of s1(x:=True) s1]
      same-val-if-reduced[OF 1, of s2(x:=True) s2] by simp
  thus ?thesis by blast
 qed
qed auto

corollary taut-test:
  taut-test ifex-of b = (∀ s. val b s)
 by (metis taut-test-def reduced-IF-depends[OF reduced-ifex] val-ifex val-ifex.simps(1,2))

corollary sat-test:
  sat-test ifex-of b = (∃ s. val b s)
 by (metis sat-test-def reduced-IF-depends[OF reduced-ifex] val-ifex val-ifex.simps(1,2))

corollary impl-test:
  impl-test ifex-of b1 b2 = (∀ s. val b1 s ⟶ val b2 s)
proof −
  have impl-test ifex-of b1 b2 = (∀ s. val-ifex (normif Mapping.empty (ifex-of b1)
(ifex-of b2) Trueif) s)
   using reduced-IF-depends[OF reduced-normif] by (fastforce simp: impl-test-def)
  also
   have (∀ s. val-ifex (normif Mapping.empty (ifex-of b1) (ifex-of b2) Trueif) s)
⟷ (∀ s. val b1 s ⟶ val b2 s)
    using reduced-IF-depends[OF reduced-ifex] val-ifex unfolding val-normif[OF
agree-Nil] by simp
  finally
  show ?thesis .
qed

corollary equiv-test:
  equiv-test ifex-of b1 b2 = (∀ s. val b1 s = val b2 s)
proof −
  have equiv-test ifex-of b1 b2 = (∀ s. val-ifex (let t1 = ifex-of b1; t2 = ifex-of b2
in normif Mapping.empty t1 t2 (normif Mapping.empty t2 Falseif Trueif)) s)
```

by (*simp add*: *equiv-test-def Let-def*; *insert reduced-IF-depends*[*OF reduced-normif*]; *force*)
moreover
{
fix *s*
have *val-ifex* (*let t1* = *ifex-of b1*; *t2* = *ifex-of b2 in normif Mapping.empty t1 t2* (*normif Mapping.empty t2 Falseif Trueif*)) *s*
= (*val b1 s* = *val b2 s*)
using *val-ifex* **by** (*simp add*: *Let-def val-normif*[*OF agree-Nil*])
}
ultimately
show *?thesis*
by *blast*
qed

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* ⇒ (*'a* ⇒ *bool*) ⇒ *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* = (¬ *val-bool-expr b s*) |
val-bool-expr (*And-bool-expr b1 b2*) *s* = (*val-bool-expr b1 s* ∧ *val-bool-expr b2 s*) |
val-bool-expr (*Or-bool-expr b1 b2*) *s* = (*val-bool-expr b1 s* ∨ *val-bool-expr b2 s*) |
val-bool-expr (*Imp-bool-expr b1 b2*) *s* = (*val-bool-expr b1 s* ⟶ *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* ⇒ *'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*
by (*induct-tac b*) (*auto simp*: *val-normif agree-Nil Let-def*)

theorem *reduced-ifex*:
reduced (*ifex-of b*) {}
by (*induction b*) (*simp add*: *Let-def*; *metis keys-empty reduced-normif*)+

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

lemma *bool-tests*:
bool-taut-test b = (∀ *s*. *val-bool-expr b s*) (**is** *?t1*)
bool-sat-test b = (∃ *s*. *val-bool-expr b s*) (**is** *?t2*)
bool-impl-test b1 b2 = (∀ *s*. *val-bool-expr b1 s* ⟶ *val-bool-expr b2 s*) (**is** *?t3*)
bool-equiv-test b1 b2 = (∀ *s*. *val-bool-expr b1 s* ⟷ *val-bool-expr b2 s*) (**is** *?t4*)
proof −
interpret *reduced-bdt-checkers ifex-of val-bool-expr*
by (*unfold-locales*; *insert val-ifex reduced-ifex*; *blast*)
show *?t1 ?t2 ?t3 ?t4*
by (*simp-all add*: *bool-taut-test-def bool-sat-test-def bool-impl-test-def bool-equiv-test-def taut-test sat-test impl-test equiv-test*)
qed

end

theory *Boolean-Expression-Checkers-AList-Mapping*
imports *Main HOL−Library.AList-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 = *None* ⟹ *Mapping.update k v* (*AList-Mapping.Mapping xs*) = *AList-Mapping.Mapping* ((*k*,*v*)#*xs*)
by (*metis Mapping.abs-eq map-of.simps*(*2*) *prod.sel*(*1*) *prod.sel*(*2*) *update-Mapping update-conv'*)

fun *reduce-alist* :: (*'a* ∗ *bool*) *list* ⇒ *'a ifex* ⇒ *'a ifex*
where
reduce-alist xs (*IF x t1 t2*) = (*case map-of xs x of*
None ⇒ *mkIF x* (*reduce-alist* ((*x*, *True*)#*xs*) *t1*) (*reduce-alist* ((*x*, *False*)#*xs*) *t2*) |
Some b ⇒ *reduce-alist xs* (*if b then t1 else t2*))
| *reduce-alist - t* = *t*

primrec *normif-alist* :: (*'a* ∗ *bool*) *list* ⇒ *'a ifex* ⇒ *'a ifex* ⇒ *'a ifex* ⇒ *'a ifex*
where
normif-alist xs Trueif t1 t2 = *reduce-alist xs t1*
| *normif-alist xs Falseif t1 t2* = *reduce-alist xs t2*
| *normif-alist xs* (*IF x t1 t2*) *t3 t4* = (*case map-of xs x of*
None ⇒ *mkIF x* (*normif-alist* ((*x*, *True*)#*xs*) *t1 t3 t4*) (*normif-alist* ((*x*, *False*)#*xs*) *t2 t3 t4*) |
Some b ⇒ *if b then normif-alist xs t1 t3 t4 else normif-alist xs t2 t3 t4*)

lemma *reduce-alist-code* [*code*, *code-unfold*]:
reduce (*AList-Mapping.Mapping xs*) *t* = *reduce-alist xs t*
by (*induction t arbitrary*: *xs*)
(*auto simp*: *AList-Mapping-update split*: *option.split*)

lemma *normif-alist-code* [*code*, *code-unfold*]:
normif (*AList-Mapping.Mapping xs*) *t* = *normif-alist xs t*
by (*induction t arbitrary*: *xs*)
(*fastforce simp*: *AList-Mapping-update reduce-alist-code split*: *option.split*)+

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*
by (*induction b*) *auto*

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

corollary *my-taut-test*:
my-taut-test-bool b = (∀ *s*. *bval b s*)
by (*simp add*: *my-taut-test-bool-def val-preservation bool-tests*)

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* ⇒ *'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*
by (*induction b*) (*simp-all add*: *agree-Nil val-normif*)

theorem *reduced-ifex*:
reduced (*ifex-of b*) {}
by (*induction b*) (*simp*; *metis keys-empty reduced-normif*)+

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

corollary *my-taut-test-ifex*:
my-taut-test-ifex b = (∀ *s*. *bval b s*)
proof −
interpret *reduced-bdt-checkers ifex-of bval*
by (*unfold-locales*; *insert val-ifex reduced-ifex*; *blast*)
show *?thesis*
by (*simp add*: *my-taut-test-ifex-def taut-test*)
qed

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*)
by *eval*

lemma *my-taut-test-ifex* (*php 6*)
by *eval*

end