

An implementation of ROBDDs for Isabelle/HOL

Julius Michaelis and Maximilian Haslbeck and Peter Lammich and Lars Hupel

December 17, 2016

Abstract

We present a verified and executable implementation of ROBDDs in Isabelle/HOL. Our implementation relates pointer-based computation in the Heap monad to operations on an abstract definition of boolean functions. Internally, we implemented the if-then-else combinator in a recursive fashion, following the Shannon decomposition of the argument functions. The implementation mixes and adapts known techniques and is built with efficiency in mind.

Contents

1	Preface	2
2	Boolean functions	2
2.1	Shannon decomposition	3
3	Binary Decision Trees	3
4	Option Helpers	13
5	Abstract ITE Implementation	14
6	Pointermap	20
7	Functional interpretation for the abstract implementation	23
8	Array List	26
9	Imparative implementation for Pointermap	28
10	Imparative implementation	29
10.1	A standard library of functions	34
10.2	Printing	35
11	Collapsing the levels	37

1 Preface

This work is not the first to deal with BDDs in Isabelle/HOL. Ortner and Schirmer have formalized BDDs in [4] and proved the correctness of an algorithm that transforms arbitrary BDDs to ROBDDs. However, their specification does not provide efficiently executable algorithms on BDDs. Giorgino and Strecker have presented efficiently executable algorithms for ROBDDs [2] by reducing their arguments to manipulating edges of graphs. However, they have, to the best of our knowledge, not made their theory files available. Thus, no library for efficient computation on (RO)BDDs in Isabelle/HOL existed. Our work is a response to that situation.

The theoretic background of the implementation is mostly based on [1].

2 Boolean functions

```
theory Bool-Func
imports Main
begin
```

The end result of our implementation is verified against these functions:

```
type-synonym 'a boolfunc = ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool
```

if-then-else on boolean functions.

```
definition bf-ite i t e  $\equiv$  ( $\lambda l$ . if i l then t l else e l)
```

if-then-else is interesting because we can, together with constant true and false, represent all binary boolean functions using maximally two applications of it.

```
abbreviation bf-True  $\equiv$  ( $\lambda l$ . True)
```

```
abbreviation bf-False  $\equiv$  ( $\lambda l$ . False)
```

A quick demonstration:

```
definition bf-and a b  $\equiv$  bf-ite a b bf-False
```

```
lemma (bf-and a b) as  $\longleftrightarrow$  a as  $\wedge$  b as <proof>
```

```
definition bf-not b  $\equiv$  bf-ite b bf-False bf-True
```

```
lemma bf-not-alt: bf-not a as  $\longleftrightarrow$   $\neg$ a as <proof>
```

For convenience, we want a few functions more:

```
definition bf-or a b  $\equiv$  bf-ite a bf-True b
```

```
definition bf-lit v  $\equiv$  ( $\lambda l$ . l v)
```

```
definition bf-if v t e  $\equiv$  bf-ite (bf-lit v) t e
```

```
lemma bf-if-alt: bf-if v t e = ( $\lambda l$ . if l v then t l else e l) <proof>
```

definition $bf_nand\ a\ b = bf_not\ (bf_and\ a\ b)$
definition $bf_nor\ a\ b = bf_not\ (bf_or\ a\ b)$
definition $bf_biimp\ a\ b = (bf_ite\ a\ b\ (bf_not\ b))$
lemma $bf_biimp_alt: bf_biimp\ a\ b = (\lambda l. a\ l \longleftrightarrow b\ l)$ *<proof>*
definition $bf_xor\ a\ b = bf_not\ (bf_biimp\ a\ b)$
lemma $bf_xor_alt: bf_xor\ a\ b = (bf_ite\ a\ (bf_not\ b)\ b)$
<proof>

All of these are implemented and had their implementation verified.

definition $bf_imp\ a\ b = bf_ite\ a\ b\ bf_True$
lemma $bf_imp_alt: bf_imp\ a\ b = bf_or\ (bf_not\ a)\ b$ *<proof>*

lemma $[dest!,elim!]: bf_False = bf_True \implies False\ bf_True = bf_False \implies False$
<proof>

lemmas $[simp] = bf_and_def\ bf_or_def\ bf_nand_def\ bf_biimp_def\ bf_xor_alt\ bf_nor_def\ bf_not_def$

2.1 Shannon decomposition

A restriction of a boolean function on a variable is creating the boolean function that evaluates as if that variable was set to a fixed value:

definition $bf_restrict\ (i::'a)\ (val::bool)\ (f::'a\ boolfunc) \equiv (\lambda v. f\ (v(i:=val)))$

Restrictions are useful, because they remove variables from the set of significant variables:

definition $bf_vars\ bf = \{v. \exists as. bf_restrict\ v\ True\ bf\ as \neq bf_restrict\ v\ False\ bf\ as\}$

lemma $var \notin bf_vars\ (bf_restrict\ var\ val\ ex)$
<proof>

We can decompose calculating if-then-else into computing if-then-else of two triples of functions with one variable restricted to true / false. Given that the functions have finite arity, we can use this to construct a recursive definition.

lemma $brace90shannon: bf_ite\ F\ G\ H\ ass =$
 $bf_ite\ (\lambda l. l\ i)$
 $(bf_ite\ (bf_restrict\ i\ True\ F)\ (bf_restrict\ i\ True\ G)\ (bf_restrict\ i\ True\ H))$
 $(bf_ite\ (bf_restrict\ i\ False\ F)\ (bf_restrict\ i\ False\ G)\ (bf_restrict\ i\ False\ H))$
 ass
<proof>

end

3 Binary Decision Trees

theory *BDT*

```
imports Bool-Func
begin
```

We first define all operations and properties on binary decision trees. This has the advantage that we can use a simple, structurally defined type and the disadvantage that we cannot represent sharing.

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

The type is the same as in Boolean Expression Checkers by Nipkow [3]. Internally, Boolean Expression Checkers transforms the boolean expressions to reduced BDTs of this type. Tests like being tautology testing are then trivial.

```
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)
```

```
fun ifex-vars :: ('a :: linorder) ifex  $\Rightarrow$  'a list where
  ifex-vars (IF v t e) = v # ifex-vars t @ ifex-vars e |
  ifex-vars Trueif = [] |
  ifex-vars Falseif = []
```

```
abbreviation ifex-var-set a  $\equiv$  set (ifex-vars a)
```

```
fun ifex-ordered :: ('a::linorder) ifex  $\Rightarrow$  bool where
  ifex-ordered (IF v t e) = (( $\forall$  tv  $\in$  (ifex-var-set t  $\cup$  ifex-var-set e). v < tv)
     $\wedge$  ifex-ordered t  $\wedge$  ifex-ordered e) |
  ifex-ordered Trueif = True |
  ifex-ordered Falseif = True
```

```
fun ifex-minimal :: ('a::linorder) ifex  $\Rightarrow$  bool where
  ifex-minimal (IF v t e)  $\longleftrightarrow$  t  $\neq$  e  $\wedge$  ifex-minimal t  $\wedge$  ifex-minimal e |
  ifex-minimal Trueif = True |
  ifex-minimal Falseif = True
```

```
abbreviation ro-ifex where ro-ifex t  $\equiv$  ifex-ordered t  $\wedge$  ifex-minimal t
```

```
definition bf-ifex-rel where
  bf-ifex-rel = {(a,b). ( $\forall$  ass. a ass  $\longleftrightarrow$  val-ifex b ass)  $\wedge$  ro-ifex b}
```

```
lemma ifex-var-noinfluence: x  $\notin$  ifex-var-set b  $\implies$  val-ifex b (ass(x:=val)) =
  val-ifex b ass
  <proof>
```

```
lemma roifex-var-not-in-subtree:
  assumes ro-ifex b and b = IF v t e
  shows v  $\notin$  ifex-var-set t and v  $\notin$  ifex-var-set e
  <proof>
```

lemma *roifex-set-var-subtree*:

assumes *ro-ifex b* **and** $b = IF\ v\ t\ e$

shows $val\text{-ifex}\ b\ (ass(v:=True)) = val\text{-ifex}\ t\ ass$

$val\text{-ifex}\ b\ (ass(v:=False)) = val\text{-ifex}\ e\ ass$

<proof>

lemma *roifex-Trueif-unique*: $ro\text{-ifex}\ b \implies \forall\ ass.\ val\text{-ifex}\ b\ ass \implies b = Trueif$

<proof>

lemma *roifex-Falseif-unique*: $ro\text{-ifex}\ b \implies \forall\ ass.\ \neg\ val\text{-ifex}\ b\ ass \implies b = Falseif$

<proof>

lemma $(f, b) \in bf\text{-ifex}\text{-rel} \implies b = Trueif \longleftrightarrow f = (\lambda\ \cdot.\ True)$

<proof>

lemma $(f, b) \in bf\text{-ifex}\text{-rel} \implies b = Falseif \longleftrightarrow f = (\lambda\ \cdot.\ False)$

<proof>

lemma *ifex-ordered-not-part*: $ifex\text{-ordered}\ b \implies b = IF\ v\ b1\ b2 \implies w < v \implies$

$w \notin ifex\text{-var}\text{-set}\ b$

<proof>

lemma *ro-ifex-unique*: $ro\text{-ifex}\ x \implies ro\text{-ifex}\ y \implies (\bigwedge\ ass.\ val\text{-ifex}\ x\ ass = val\text{-ifex}\ y\ ass) \implies x = y$

<proof>

theorem *bf-ifex-rel-single*: *single-valued bf-ifex-rel single-valued (bf-ifex-rel⁻¹)*

<proof>

lemma *bf-ifex-eq*: $(af, at) \in bf\text{-ifex}\text{-rel} \implies (bf, bt) \in bf\text{-ifex}\text{-rel} \implies (af = bf) \longleftrightarrow (at = bt)$

<proof>

lemma *nonempty-if-var-set*: $ifex\text{-vars}\ (IF\ v\ t\ e) \neq []$ *<proof>*

fun *restrict where*

$restrict\ (IF\ v\ t\ e)\ var\ val = (let\ rt = restrict\ t\ var\ val;\ re = restrict\ e\ var\ val\ in$

$(if\ v = var\ then\ (if\ val\ then\ rt\ else\ re)\ else\ (IF\ v\ rt\ re)))\ |$

$restrict\ i\ -\ - = i$

declare *Let-def[simp]*

lemma *not-element-restrict*: $var \notin ifex\text{-var}\text{-set}\ (restrict\ b\ var\ val)$

<proof>

lemma *restrict-assignment*: $val\text{-ifex}\ b\ (ass(var := val)) \longleftrightarrow val\text{-ifex}\ (restrict\ b\ var\ val)\ ass$

<proof>

lemma *restrict-variables-subset*: $\text{ifex-var-set } (\text{restrict } b \text{ var } val) \subseteq \text{ifex-var-set } b$
 ⟨proof⟩

lemma *restrict-ifex-ordered-invar*: $\text{ifex-ordered } b \implies \text{ifex-ordered } (\text{restrict } b \text{ var } val)$
 ⟨proof⟩

lemma *restrict-val-invar*: $\forall \text{ ass. } a \text{ ass} = \text{val-ifex } b \text{ ass} \implies$
 $(\text{bf-restrict var } val \ a) \text{ ass} = \text{val-ifex } (\text{restrict } b \text{ var } val) \text{ ass}$
 ⟨proof⟩

lemma *restrict-untouched-id*: $x \notin \text{ifex-var-set } t \implies \text{restrict } t \ x \ val = t$
 ⟨proof⟩

fun *ifex-top-var* :: $'a \text{ ifex} \implies 'a \text{ option}$ **where**
 $\text{ifex-top-var } (\text{IF } v \ t \ e) = \text{Some } v \mid$
 $\text{ifex-top-var } - = \text{None}$

fun *restrict-top* :: $('a :: \text{linorder}) \text{ ifex} \implies 'a \implies \text{bool} \implies 'a \text{ ifex}$ **where**
 $\text{restrict-top } (\text{IF } v \ t \ e) \ \text{var } \text{val} = (\text{if } v = \text{var} \ \text{then } (\text{if } \text{val} \ \text{then } t \ \text{else } e) \ \text{else } (\text{IF } v$
 $t \ e)) \mid$
 $\text{restrict-top } i \ - \ - = i$

lemma *restrict-top-id*: $\text{ifex-ordered } e \implies \text{ifex-top-var } e = \text{Some } v \implies v' < v \implies$
 $\text{restrict-top } e \ v' \ \text{val} = e$
 ⟨proof⟩

lemma *restrict-id*: $\text{ifex-ordered } e \implies \text{ifex-top-var } e = \text{Some } v \implies v' < v \implies$
 $\text{restrict } e \ v' \ \text{val} = e$
 ⟨proof⟩

lemma *restrict-top-IF-id*: $\text{ifex-ordered } (\text{IF } v \ t \ e) \implies v' < v \implies \text{restrict-top } (\text{IF}$
 $v \ t \ e) \ v' \ \text{val} = (\text{IF } v \ t \ e)$
 ⟨proof⟩

lemma *restrict-IF-id*: **assumes** o : $\text{ifex-ordered } (\text{IF } v \ t \ e)$ **assumes** le : $v' < v$
shows $\text{restrict } (\text{IF } v \ t \ e) \ v' \ \text{val} = (\text{IF } v \ t \ e)$
 ⟨proof⟩

lemma *restrict-top-eq*: $\text{ifex-ordered } (\text{IF } v \ t \ e) \implies \text{restrict } (\text{IF } v \ t \ e) \ v \ \text{val} =$
 $\text{restrict-top } (\text{IF } v \ t \ e) \ v \ \text{val}$
 ⟨proof⟩

lemma *restrict-top-ifex-ordered-invar*: $\text{ifex-ordered } b \implies \text{ifex-ordered } (\text{restrict-top}$
 $b \ \text{var } \text{val})$
 ⟨proof⟩

fun *lowest-tops* :: ('a :: linorder) ifex list \Rightarrow 'a option **where**
lowest-tops [] = None |
lowest-tops ((IF v -)#r) = Some (case *lowest-tops* r of Some u \Rightarrow (min u v) |
None \Rightarrow v) |
lowest-tops (-#r) = *lowest-tops* r

lemma *lowest-tops-NoneD*: *lowest-tops* k = None \implies ($\neg(\exists v t e. ((IF v t e) \in \text{set } k))$)
<proof>

lemma *lowest-tops-in*: *lowest-tops* k = Some l \implies l \in set (concat (map *ifex-vars* k))
<proof>

definition *IFC* v t e \equiv (if t = e then t else IF v t e)

function *ifex-ite* :: 'a ifex \Rightarrow 'a ifex \Rightarrow 'a ifex \Rightarrow ('a :: linorder) ifex **where**
ifex-ite i t e = (case *lowest-tops* [i, t, e] of Some x \Rightarrow
(IFC x (*ifex-ite* (restrict-top i x True) (restrict-top t x True)
(restrict-top e x True))
(*ifex-ite* (restrict-top i x False) (restrict-top t x False)
(restrict-top e x False)))
| None \Rightarrow (case i of Trueif \Rightarrow t | Falseif \Rightarrow e))
<proof>

lemma *restrict-size-le*: size (restrict-top k var val) \leq size k
<proof>

lemma *restrict-size-less*: *ifex-top-var* k = Some var \implies size (restrict-top k var val) < size k
<proof>

lemma *lowest-tops-cases*:
lowest-tops [i, t, e] = Some var \implies *ifex-top-var* i = Some var \vee *ifex-top-var* t
= Some var \vee *ifex-top-var* e = Some var
<proof>

lemma *lowest-tops-lowest*: *lowest-tops* es = Some a \implies e \in set es \implies *ifex-ordered*
e \implies v \in *ifex-var-set* e \implies a \leq v
<proof>

lemma *termlemma2*: *lowest-tops* [i, t, e] = Some xa \implies
(size (restrict-top i xa val) + size (restrict-top t xa val) + size (restrict-top e xa
val)) <
(size i + size t + size e)
<proof>

lemma *termlemma*: *lowest-tops* [i, t, e] = Some xa \implies
(case (restrict-top i xa val, restrict-top t xa val, restrict-top e xa val) of

$(i, t, e) \Rightarrow \text{size } i + \text{size } t + \text{size } e <$
 $(\text{case } (i, t, e) \text{ of } (i, t, e) \Rightarrow \text{size } i + \text{size } t + \text{size } e)$
 $\langle \text{proof} \rangle$

termination *ifex-ite*
 $\langle \text{proof} \rangle$

definition *const x - = x*

declare *const-def[simp]*

lemma *rel-true-false*: $(a, \text{Trueif}) \in \text{bf-ifex-rel} \Rightarrow a = \text{const True}$ $(a, \text{Falseif}) \in$
 $\text{bf-ifex-rel} \Rightarrow a = \text{const False}$
 $\langle \text{proof} \rangle$

lemma *rel-if*: $(a, \text{IF } v \text{ t } e) \in \text{bf-ifex-rel} \Rightarrow (ta, t) \in \text{bf-ifex-rel} \Rightarrow (ea, e) \in$
 $\text{bf-ifex-rel} \Rightarrow a = (\lambda as. \text{if } as \text{ v then } ta \text{ as else } ea \text{ as})$
 $\langle \text{proof} \rangle$

lemma *ifex-ordered-implied*: $(a, b) \in \text{bf-ifex-rel} \Rightarrow \text{ifex-ordered } b$ $\langle \text{proof} \rangle$

lemma *ifex-minimal-implied*: $(a, b) \in \text{bf-ifex-rel} \Rightarrow \text{ifex-minimal } b$ $\langle \text{proof} \rangle$

lemma *ifex-ite-induct2*[*case-names Trueif Falseif IF*]:

$(\bigwedge i \text{ t } e. \text{lowest-tops } [i, t, e] = \text{None} \Rightarrow i = \text{Trueif} \Rightarrow \text{sentence } i \text{ t } e) \Rightarrow$
 $(\bigwedge i \text{ t } e. \text{lowest-tops } [i, t, e] = \text{None} \Rightarrow i = \text{Falseif} \Rightarrow \text{sentence } i \text{ t } e) \Rightarrow$
 $(\bigwedge i \text{ t } e \text{ a. sentence } (\text{restrict-top } i \text{ a True}) (\text{restrict-top } t \text{ a True}) (\text{restrict-top } e \text{ a}$
 $\text{True}) \Rightarrow$
 $\text{sentence } (\text{restrict-top } i \text{ a False}) (\text{restrict-top } t \text{ a False}) (\text{restrict-top } e \text{ a}$
 $\text{False}) \Rightarrow$
 $\text{lowest-tops } [i, t, e] = \text{Some } a \Rightarrow \text{sentence } i \text{ t } e) \Rightarrow \text{sentence } i \text{ t } e$
 $\langle \text{proof} \rangle$

lemma *ifex-ite-induct*[*case-names Trueif Falseif IF*]:

$(\bigwedge i \text{ t } e. \text{lowest-tops } [i, t, e] = \text{None} \Rightarrow i = \text{Trueif} \Rightarrow P \text{ i t } e) \Rightarrow$
 $(\bigwedge i \text{ t } e. \text{lowest-tops } [i, t, e] = \text{None} \Rightarrow i = \text{Falseif} \Rightarrow P \text{ i t } e) \Rightarrow$
 $(\bigwedge i \text{ t } e \text{ a. } (\bigwedge \text{val. } P (\text{restrict-top } i \text{ a val}) (\text{restrict-top } t \text{ a val}) (\text{restrict-top } e \text{ a}$
 $\text{val})) \Rightarrow$
 $\text{lowest-tops } [i, t, e] = \text{Some } a \Rightarrow P \text{ i t } e) \Rightarrow P \text{ i t } e$
 $\langle \text{proof} \rangle$

lemma *restrict-top-subset*: $x \in \text{ifex-var-set } (\text{restrict-top } i \text{ vr vl}) \Rightarrow x \in \text{ifex-var-set}$
 i
 $\langle \text{proof} \rangle$

lemma *ifex-vars-subset*: $x \in \text{ifex-var-set } (\text{ifex-ite } i \text{ t } e) \Rightarrow (x \in \text{ifex-var-set } i) \vee$
 $(x \in \text{ifex-var-set } t) \vee (x \in \text{ifex-var-set } e)$
 $\langle \text{proof} \rangle$

lemma *three-ins*: $i \in \text{set } [i, t, e] \ t \in \text{set } [i, t, e] \ e \in \text{set } [i, t, e] \langle \text{proof} \rangle$

lemma *hlp3*: $\text{lowest-tops } (IF \ v \ uu \ uv \ \# \ r) \neq \text{lowest-tops } r \implies \text{lowest-tops } (IF \ v \ uu \ uv \ \# \ r) = \text{Some } v \langle \text{proof} \rangle$

lemma *hlp2*: $IF \ vi \ vt \ ve \in \text{set } is \implies \text{lowest-tops } is = \text{Some } a \implies a \leq vi \langle \text{proof} \rangle$

lemma *hlp1*: $i \in \text{set } is \implies \text{lowest-tops } is = \text{Some } a \implies \text{ifex-ordered } i \implies a \notin (\text{ifex-var-set } (\text{restrict-top } i \ a \ \text{val})) \langle \text{proof} \rangle$

lemma *order-ifex-ite-invar*: $\text{ifex-ordered } i \implies \text{ifex-ordered } t \implies \text{ifex-ordered } e \implies \text{ifex-ordered } (\text{ifex-ite } i \ t \ e) \langle \text{proof} \rangle$

lemma *ifc-split*: $P \ (IFC \ v \ t \ e) \longleftrightarrow ((t = e) \longrightarrow P \ t) \wedge (t \neq e \longrightarrow P \ (IF \ v \ t \ e)) \langle \text{proof} \rangle$

lemma *restrict-top-ifex-minimal-invar*: $\text{ifex-minimal } i \implies \text{ifex-minimal } (\text{restrict-top } i \ a \ \text{val}) \langle \text{proof} \rangle$

lemma *minimal-ifex-ite-invar*: $\text{ifex-minimal } i \implies \text{ifex-minimal } t \implies \text{ifex-minimal } e \implies \text{ifex-minimal } (\text{ifex-ite } i \ t \ e) \langle \text{proof} \rangle$

lemma *restrict-top-bf*: $i \in \text{set } is \implies \text{lowest-tops } is = \text{Some } vr \implies \text{ifex-ordered } i \implies (\bigwedge \text{ass. } fi \ \text{ass} = \text{val-ifex } i \ \text{ass}) \implies \text{val-ifex } (\text{restrict-top } i \ vr \ vl) \ \text{ass} = \text{bf-restrict } vr \ vl \ fi \ \text{ass} \langle \text{proof} \rangle$

lemma *val-ifex-ite*:
 $(\bigwedge \text{ass. } fi \ \text{ass} = \text{val-ifex } i \ \text{ass}) \implies$
 $(\bigwedge \text{ass. } ft \ \text{ass} = \text{val-ifex } t \ \text{ass}) \implies$
 $(\bigwedge \text{ass. } fe \ \text{ass} = \text{val-ifex } e \ \text{ass}) \implies$
 $\text{ifex-ordered } i \implies \text{ifex-ordered } t \implies \text{ifex-ordered } e \implies$
 $(\text{bf-ite } fi \ ft \ fe) \ \text{ass} = \text{val-ifex } (\text{ifex-ite } i \ t \ e) \ \text{ass} \langle \text{proof} \rangle$

theorem *ifex-ite-rel-bf*:
 $(fi, i) \in \text{bf-ifex-rel} \implies$
 $(ft, t) \in \text{bf-ifex-rel} \implies$
 $(fe, e) \in \text{bf-ifex-rel} \implies$
 $((\text{bf-ite } fi \ ft \ fe), (\text{ifex-ite } i \ t \ e)) \in \text{bf-ifex-rel} \langle \text{proof} \rangle$

definition *param-opt* **where** $\text{param-opt } i \ t \ e =$

(if $i = \text{Trueif}$ then $\text{Some } t$ else
 if $i = \text{Falseif}$ then $\text{Some } e$ else
 if $t = \text{Trueif} \wedge e = \text{Falseif}$ then $\text{Some } i$ else
 if $t = e$ then $\text{Some } t$ else
 if $e = \text{Trueif} \wedge i = t$ then Some Trueif else
 if $t = \text{Falseif} \wedge i = e$ then Some Falseif else
 None)

lemma *param-opt-ifex-ite-eq*: $\text{ro-ifex } i \implies \text{ro-ifex } t \implies \text{ro-ifex } e \implies$
 $\text{param-opt } i \ t \ e = \text{Some } r \implies r = \text{ifex-ite } i \ t \ e$
 <proof>

function *ifex-ite-opt* :: ' a ifex \Rightarrow ' a ifex \Rightarrow ' a ifex \Rightarrow (' a :: linorder) ifex **where**
 $\text{ifex-ite-opt } i \ t \ e = (\text{case } \text{param-opt } i \ t \ e \text{ of } \text{Some } b \Rightarrow b \mid \text{None} \Rightarrow$
 $\text{case } \text{lowest-tops } [i, t, e] \text{ of } \text{Some } x \Rightarrow$
 $(\text{IFC } x \ (\text{ifex-ite-opt } (\text{restrict-top } i \ x \ \text{True})) \ (\text{restrict-top } t \ x$
 $\text{True}))$
 $\text{True}))$
 $(\text{restrict-top } e \ x \ \text{True}))$
 $(\text{ifex-ite-opt } (\text{restrict-top } i \ x \ \text{False}) \ (\text{restrict-top } t \ x \ \text{False}))$
 $(\text{restrict-top } e \ x \ \text{False}))$
 $\mid \text{None} \Rightarrow (\text{case } i \text{ of } \text{Trueif} \Rightarrow t \mid \text{Falseif} \Rightarrow e))$

<proof>

termination *ifex-ite-opt*
 <proof>

lemma *ifex-ite-opt-eq*:
 $\text{ro-ifex } i \implies \text{ro-ifex } t \implies \text{ro-ifex } e \implies \text{ifex-ite-opt } i \ t \ e = \text{ifex-ite } i \ t \ e$
 <proof>

lemma *ro-ifexI*: $(a, b) \in \text{bf-ifex-rel} \implies \text{ro-ifex } b$ <proof>

theorem *ifex-ite-opt-rel-bf*:
 $(fi, i) \in \text{bf-ifex-rel} \implies$
 $(ft, t) \in \text{bf-ifex-rel} \implies$
 $(fe, e) \in \text{bf-ifex-rel} \implies$
 $((\text{bf-ite } fi \ ft \ fe), (\text{ifex-ite-opt } i \ t \ e)) \in \text{bf-ifex-rel}$
 <proof>

lemma *restrict-top-bf-ifex-rel*:
 $(f, i) \in \text{bf-ifex-rel} \implies \exists f'. (f', \text{restrict-top } i \ \text{var } \text{val}) \in \text{bf-ifex-rel}$
 <proof>

lemma *param-opt-lowest-tops-lem*: $\text{param-opt } i \ t \ e = \text{None} \implies \exists y. \text{lowest-tops}$
 $[i, t, e] = \text{Some } y$
 <proof>

```

fun ifex-sat where
ifex-sat Trueif = Some (const False) |
ifex-sat Falseif = None |
ifex-sat (IF v t e) =
  (case ifex-sat e of
    Some a  $\Rightarrow$  Some (a(v:=False)) |
    None  $\Rightarrow$  (case ifex-sat t of
      Some a  $\Rightarrow$  Some (a(v:=True)) |
      None  $\Rightarrow$  None))

```

lemma ifex-sat-untouched-False: $v \notin \text{ifex-var-set } i \implies \text{ifex-sat } i = \text{Some } a \implies a = \text{False}$
 <proof>

lemma ifex-upd-other: $v \notin \text{ifex-var-set } i \implies \text{val-ifex } i (a(v:=\text{any})) = \text{val-ifex } i a$
 <proof>

```

fun ifex-no-twice where
ifex-no-twice (IF v t e) = (
  v  $\notin$  (ifex-var-set t  $\cup$  ifex-var-set e)  $\wedge$ 
  ifex-no-twice t  $\wedge$  ifex-no-twice e) |
ifex-no-twice - = True

```

lemma ordered-ifex-no-twiceI: $\text{ifex-ordered } i \implies \text{ifex-no-twice } i$
 <proof>

lemma ifex-sat-NoneD: $\text{ifex-sat } i = \text{None} \implies \text{val-ifex } i \text{ ass} = \text{False}$
 <proof>

lemma ifex-sat-SomeD: $\text{ifex-no-twice } i \implies \text{ifex-sat } i = \text{Some } \text{ass} \implies \text{val-ifex } i \text{ ass} = \text{True}$
 <proof>

lemma ifex-sat-NoneI: $\text{ifex-no-twice } i \implies (\bigwedge \text{ass. val-ifex } i \text{ ass} = \text{False}) \implies \text{ifex-sat } i = \text{None}$

<proof>

```

fun ifex-sat-list where
ifex-sat-list Trueif = Some [] |
ifex-sat-list Falseif = None |
ifex-sat-list (IF v t e) =
  (case ifex-sat-list e of
    Some a  $\Rightarrow$  Some ((v,False)#a) |
    None  $\Rightarrow$  (case ifex-sat-list t of
      Some a  $\Rightarrow$  Some ((v,True)#a) |
      None  $\Rightarrow$  None))

```

definition *update-assignment-alt* $u\ as = (\lambda v. \text{case map-of } u\ v\ \text{of } \text{None} \Rightarrow as\ v\ |\ \text{Some } n \Rightarrow n)$

fun *update-assignment* **where**

update-assignment $((v,u)\#us)\ as = (\text{update-assignment } us\ as)(v:=u)\ |$

update-assignment $[]\ as = as$

lemma *update-assignment-notin*: $a \notin \text{fst } 'set\ us \Longrightarrow \text{update-assignment } us\ as\ a = as\ a$
 $\langle proof \rangle$

lemma *update-assignment-alt*: $\text{update-assignment } u\ as = \text{update-assignment-alt } u\ as$
 $\langle proof \rangle$

lemma *update-assignment*: $\text{distinct } (\text{map } \text{fst } ((v,u)\#us)) \Longrightarrow \text{update-assignment } ((v,u)\#us)\ as = \text{update-assignment } us\ (as(v:=u))$
 $\langle proof \rangle$

lemma *ass-upd-same*: $\text{update-assignment } ((v, u) \# a)\ ass\ v = u\ \langle proof \rangle$

lemma *ifex-sat-list-subset*: $\text{ifex-sat-list } t = \text{Some } u \Longrightarrow \text{fst } 'set\ u \subseteq \text{ifex-var-set } t$
 $\langle proof \rangle$

lemma *sat-list-distinct*: $\text{ifex-no-twice } t \Longrightarrow \text{ifex-sat-list } t = \text{Some } u \Longrightarrow \text{distinct } (\text{map } \text{fst } u)$
 $\langle proof \rangle$

lemma *ifex-sat-list-NoneD*: $\text{ifex-sat-list } i = \text{None} \Longrightarrow \text{val-ifex } i\ ass = \text{False}$
 $\langle proof \rangle$

lemma *ifex-sat-list-SomeD*: $\text{ifex-no-twice } i \Longrightarrow \text{ifex-sat-list } i = \text{Some } u \Longrightarrow ass = \text{update-assignment } u\ ass' \Longrightarrow \text{val-ifex } i\ ass = \text{True}$
 $\langle proof \rangle$

fun *sat-list-to-bdt* **where**

sat-list-to-bdt $[] = \text{Trueif}\ |$

sat-list-to-bdt $((v,u)\#us) = (\text{if } u\ \text{then } \text{IF } v\ (\text{sat-list-to-bdt } us)\ \text{Falseif}\ \text{else } \text{IF } v\ \text{Falseif } (\text{sat-list-to-bdt } us))$

lemma *ifex-sat-list* $i = \text{Some } u \Longrightarrow \text{val-ifex } (\text{sat-list-to-bdt } u)\ as \Longrightarrow \text{val-ifex } i\ as$
 $\langle proof \rangle$

lemma *bf-ifex-rel-consts*[*simp,intro!*]:

$(\text{bf-True}, \text{Trueif}) \in \text{bf-ifex-rel}$

$(\text{bf-False}, \text{Falseif}) \in \text{bf-ifex-rel}$

$\langle proof \rangle$

lemma *bf-ifex-rel-lit*[*simp,intro!*]:

$(\text{bf-lit } v, \text{IFC } v\ \text{Trueif}\ \text{Falseif}) \in \text{bf-ifex-rel}$

$\langle proof \rangle$

lemma *bf-ifex-rel-consts-ensured*[simp]:
 $(bf-True, x) \in bf-ifex-rel \longleftrightarrow (x = Trueif)$
 $(bf-False, x) \in bf-ifex-rel \longleftrightarrow (x = Falseif)$
 $\langle proof \rangle$

lemma *bf-ifex-rel-consts-ensured-rev*[simp]:
 $(x, Trueif) \in bf-ifex-rel \longleftrightarrow (x = bf-True)$
 $(x, Falseif) \in bf-ifex-rel \longleftrightarrow (x = bf-False)$
 $\langle proof \rangle$

declare *ifex-ite-opt.simps restrict-top.simps lowest-tops.simps*[simp del]

end

4 Option Helpers

These definitions were contributed by Peter Lammich.

theory *Option-Helpers*
imports *Main* $\sim\sim$ */src/HOL/Library/Monad-Syntax*
begin

primrec *oassert* :: *bool* \Rightarrow *unit option* **where**
oassert True = *Some ()* | *oassert False* = *None*

lemma *oassert-iff*[simp]:
 $oassert \Phi = Some\ x \longleftrightarrow \Phi$
 $oassert \Phi = None \longleftrightarrow \neg\Phi$
 $\langle proof \rangle$

The idea is that we want the result of some computation to be *Some v* and the contents of *v* to satisfy some property *Q*.

primrec *ospec* :: (*'a option*) \Rightarrow (*'a* \Rightarrow *bool*) \Rightarrow *bool* **where**
ospec None = *False*
| *ospec (Some v)* *Q* = *Q v*

named-theorems *ospec-rules*

lemma *oreturn-rule*[*ospec-rules*]: $\llbracket P\ r \rrbracket \Longrightarrow ospec\ (Some\ r)\ P \langle proof \rangle$

lemma *obind-rule*[*ospec-rules*]: $\llbracket ospec\ m\ Q; \bigwedge r. Q\ r \Longrightarrow ospec\ (f\ r)\ P \rrbracket \Longrightarrow ospec\ (m\ \gg\ f)\ P \langle proof \rangle$

lemma *ospec-alt*: $ospec\ m\ P = (case\ m\ of\ None \Rightarrow False\ | Some\ x \Rightarrow P\ x) \langle proof \rangle$

lemma *ospec-bind-simp*: $ospec (m \gg f) P \longleftrightarrow (ospec m (\lambda r. ospec (f r) P))$
 ⟨proof⟩

lemma *ospec-cons*:
assumes $ospec m Q$
assumes $\bigwedge r. Q r \implies P r$
shows $ospec m P$
 ⟨proof⟩

lemma *oreturn-synth*: $ospec (Some x) (\lambda r. r=x)$ ⟨proof⟩

lemma *ospecD*: $ospec x P \implies x = Some y \implies P y$ ⟨proof⟩

lemma *ospecD2*: $ospec x P \implies \exists y. x = Some y \wedge P y$ ⟨proof⟩

end

5 Abstract ITE Implementation

theory *Abstract-Impl*

imports *BDT*

../Automatic-Refinement/Lib/Refine-Lib

Option-Helpers

begin

datatype $('a, 'ni)$ *IFEXD* = *TD* | *FD* | *IFD* $'a$ $'ni$ $'ni$

locale *bdd-impl-pre* =
fixes $R :: 's \Rightarrow ('ni \times ('a :: linorder) ifex)$ *set*
fixes $I :: 's \Rightarrow bool$

begin

definition $les :: 's \Rightarrow 's \Rightarrow bool$ **where**

$les s s' == \forall ni n. (ni, n) \in R s \longrightarrow (ni, n) \in R s'$

end

locale *bdd-impl* = *bdd-impl-pre* **for** $R :: 's \Rightarrow ('ni \times ('a :: linorder) ifex)$ *set* +
fixes $Timpl :: 's \rightarrow ('ni \times 's)$
fixes $Fimpl :: 's \rightarrow ('ni \times 's)$
fixes $IFimpl :: 'a \Rightarrow 'ni \Rightarrow 'ni \Rightarrow 's \rightarrow ('ni \times 's)$
fixes $DESTRimpl :: 'ni \Rightarrow 's \rightarrow ('a, 'ni)$ *IFEXD*

assumes *Timpl-rule*: $I s \implies ospec (Timpl s) (\lambda(ni, s'). (ni, Trueif) \in R s' \wedge I s' \wedge les s s')$

assumes *Fimpl-rule*: $I s \implies ospec (Fimpl s) (\lambda(ni, s'). (ni, Falseif) \in R s' \wedge I s' \wedge les s s')$

assumes *IFimpl-rule*: $\llbracket I s; (ni1, n1) \in R s; (ni2, n2) \in R s \rrbracket$
 $\implies ospec (IFimpl v ni1 ni2 s) (\lambda(ni, s'). (ni, IFC v n1 n2) \in R s' \wedge I s' \wedge les s s')$

assumes *DESTRimpl-rule1*: $I s \implies (ni, Trueif) \in R s \implies ospec (DESTRimpl$

$ni\ s$ ($\lambda r. r = TD$)
assumes *DESTRIimpl-rule2*: $I\ s \implies (ni, Falseif) \in R\ s \implies ospec\ (DESTRIimpl\ ni\ s)$
 $ni\ s$ ($\lambda r. r = FD$)
assumes *DESTRIimpl-rule3*: $I\ s \implies (ni, IF\ v\ n1\ n2) \in R\ s \implies$
 $ospec\ (DESTRIimpl\ ni\ s)$
 $(\lambda r. \exists\ ni1\ ni2. r = (IFD\ v\ ni1\ ni2) \wedge (ni1, n1) \in R\ s$
 $\wedge (ni2, n2) \in R\ s)$
begin

lemma *les-refl[simp,intro!]*: $les\ s\ s$ *<proof>*
lemma *les-trans[trans]*: $les\ s1\ s2 \implies les\ s2\ s3 \implies les\ s1\ s3$ *<proof>*
lemmas *DESTRIimpl-rules* = *DESTRIimpl-rule1 DESTRIimpl-rule2 DESTRIimpl-rule3*

lemma *DESTRIimpl-rule-useless*:
 $I\ s \implies (ni, n) \in R\ s \implies ospec\ (DESTRIimpl\ ni\ s)$ ($\lambda r. (case\ r\ of$
 $TD \Rightarrow (ni, Trueif) \in R\ s \mid$
 $FD \Rightarrow (ni, Falseif) \in R\ s \mid$
 $IFD\ v\ nt\ ne \Rightarrow (\exists\ t\ e. n = IF\ v\ t\ e \wedge (ni, IF\ v\ t\ e) \in R\ s))$)
<proof>

lemma *DESTRIimpl-rule*:
 $I\ s \implies (ni, n) \in R\ s \implies ospec\ (DESTRIimpl\ ni\ s)$ ($\lambda r. (case\ n\ of$
 $Trueif \Rightarrow r = TD \mid$
 $Falseif \Rightarrow r = FD \mid$
 $IF\ v\ t\ e \Rightarrow (\exists\ tn\ en. r = IFD\ v\ tn\ en \wedge (tn, t) \in R\ s \wedge (en, e) \in R\ s))$)
<proof>

definition *case-ifexi fti ffi fii ni s* $\equiv do\ \{$
 $dest \leftarrow DESTRIimpl\ ni\ s;$
 $case\ dest\ of$
 $TD \Rightarrow fti\ s$
 $\mid FD \Rightarrow ffi\ s$
 $\mid IFD\ v\ ti\ ei \Rightarrow fii\ v\ ti\ ei\ s\}$

lemma *case-ifexi-rule*:
assumes *INV*: $I\ s$
assumes *NI*: $(ni, n) \in R\ s$
assumes *FII*: $\llbracket n = Trueif \rrbracket \implies ospec\ (fti\ s)$ ($\lambda(r, s'). (r, ft) \in Q\ s \wedge I'\ s'$)
assumes *FFI*: $\llbracket n = Falseif \rrbracket \implies ospec\ (ffi\ s)$ ($\lambda(r, s'). (r, ff) \in Q\ s \wedge I'\ s'$)
assumes *FII*: $\bigwedge ti\ ei\ v\ t\ e. \llbracket n = IF\ v\ t\ e; (ti, t) \in R\ s; (ei, e) \in R\ s \rrbracket \implies ospec$
 $(fii\ v\ ti\ ei\ s)$ ($\lambda(r, s'). (r, fi\ v\ t\ e) \in Q\ s \wedge I'\ s'$)
shows $ospec\ (case-ifexi\ fti\ ffi\ fii\ ni\ s)$ ($\lambda(r, s'). (r, case-ifex\ ft\ ff\ fi\ n) \in Q\ s \wedge I'$
 s')
<proof>

abbreviation *return x* $\equiv \lambda s. Some\ (x, s)$

primrec *lowest-tops-impl where*
 $lowest-tops-impl\ []\ s = Some\ (None, s) \mid$
 $lowest-tops-impl\ (e\#\ es)\ s =$

```

case-ifexi
  (λs. lowest-tops-impl es s)
  (λs. lowest-tops-impl es s)
  (λv t e s. do {
    (rec,s) ← lowest-tops-impl es s;
    (case rec of
      Some u ⇒ Some ((Some (min u v)), s) |
      None ⇒ Some ((Some v), s))
    }) e s

```

declare *lowest-tops-impl.simps*[simp del]

```

fun lowest-tops-alt where
lowest-tops-alt [] = None |
lowest-tops-alt (e#es) = (
  let rec = lowest-tops-alt es in
  case-ifex
    rec
    rec
    (λv t e. (case rec of
      Some u ⇒ (Some (min u v)) |
      None ⇒ (Some v))
    ) e
)

```

lemma *lowest-tops-alt*: $\text{lowest-tops } l = \text{lowest-tops-alt } l$
 ⟨proof⟩

lemma *lowest-tops-impl-R*:
assumes *list-all2* (*in-rel* (*R s*)) *li l I s*
shows *ospec* (*lowest-tops-impl li s*) ($\lambda(r,s'). r = \text{lowest-tops } l \wedge s'=s$)
 ⟨proof⟩

definition *restrict-top-impl* **where**
restrict-top-impl e vr vl s =
 case-ifexi
 (return e)
 (return e)
 (λv te ee. return (if v = vr then (if vl then te else ee) else e))
 e s

lemma *restrict-top-alt*: $\text{restrict-top } n \text{ var } val = (\text{case } n \text{ of } (IF v t e) \Rightarrow (\text{if } v = \text{var then (if val then t else e) else } (IF v t e)) | - \Rightarrow n)$
 ⟨proof⟩

lemma *restrict-top-impl-spec*: $I s \Longrightarrow (ni,n) \in R s \Longrightarrow \text{ospec } (\text{restrict-top-impl } ni$

$vr\ vl\ s) (\lambda(res,s'). (res, restrict-top\ n\ vr\ vl) \in R\ s \wedge s'=s)$
 ⟨proof⟩

partial-function(*option*) *ite-impl* **where**

ite-impl *i* *t* *e* *s* = do {
 (*lt*,-) ← *lowest-tops-impl* [*i*, *t*, *e*] *s*;
 (case *lt* of
 Some *a* ⇒ do {
 (*ti*,-) ← *restrict-top-impl* *i* *a* *True* *s*;
 (*tt*,-) ← *restrict-top-impl* *t* *a* *True* *s*;
 (*te*,-) ← *restrict-top-impl* *e* *a* *True* *s*;
 (*fi*,-) ← *restrict-top-impl* *i* *a* *False* *s*;
 (*ft*,-) ← *restrict-top-impl* *t* *a* *False* *s*;
 (*fe*,-) ← *restrict-top-impl* *e* *a* *False* *s*;
 (*tb*,*s*) ← *ite-impl* *ti* *tt* *te* *s*;
 (*fb*,*s*) ← *ite-impl* *fi* *ft* *fe* *s*;
 IFimpl *a* *tb* *fb* *s*}
 | None ⇒ *case-ifexi* (λ -.(*Some* (*t*,*s*))) (λ -.(*Some* (*e*,*s*))) (λ - - - . *None*) *i* *s*
)}

lemma *ite-impl-R*: *I* *s*

⇒ *in-rel* (*R* *s*) *ii* *i* ⇒ *in-rel* (*R* *s*) *ti* *t* ⇒ *in-rel* (*R* *s*) *ei* *e*
 ⇒ *ospec* (*ite-impl* *ii* *ti* *ei* *s*) ($\lambda(r, s'). (r, ifex-ite\ i\ t\ e) \in R\ s' \wedge I\ s' \wedge les\ s$

s')

⟨proof⟩

lemma *case-ifexi-mono*[*partial-function-mono*]:

assumes [*partial-function-mono*]:

mono-option ($\lambda F. fti\ F\ s$)

mono-option ($\lambda F. ffi\ F\ s$)

$\wedge x31\ x32\ x33. mono-option\ (\lambda F. fii\ F\ x31\ x32\ x33\ s)$

shows *mono-option* ($\lambda F. case-ifexi\ (fti\ F)\ (ffi\ F)\ (fii\ F)\ ni\ s$)

⟨proof⟩

partial-function(*option*) *val-impl* :: '*ni* ⇒ ('*a* ⇒ *bool*) ⇒ '*s* ⇒ (*bool* × '*s*) *option*

where

val-impl *e* *ass* *s* = *case-ifexi*

($\lambda s. Some\ (True, s)$)

($\lambda s. Some\ (False, s)$)

($\lambda v\ t\ e\ s. val-impl\ (if\ ass\ v\ then\ t\ else\ e)\ ass\ s$)

e *s*

lemma *I* *s* ⇒ (*ni*,*n*) ∈ *R* *s* ⇒ *ospec* (*val-impl* *ni* *ass* *s*) ($\lambda(r,s'). r = (val-ifex\ n\ ass) \wedge s'=s$)

⟨proof⟩

end

locale *bdd-impl-cmp-pre* = *bdd-impl-pre*
begin

definition *map-invar-impl* *m s* =

($\forall ii\ ti\ ei\ ri. m\ (ii,ti,ei) = \text{Some}\ ri \longrightarrow$
 $(\exists i\ t\ e. ((ri,ifex-ite-opt\ i\ t\ e) \in R\ s) \wedge (ii,i) \in R\ s \wedge (ti,t) \in R\ s \wedge (ei,e) \in R\ s))$)

lemma *map-invar-impl-les*: *map-invar-impl* *m s* \implies *les* *s s'* \implies *map-invar-impl* *m s'*
 <proof>

lemma *map-invar-impl-update*: *map-invar-impl* *m s* \implies

$(ii,i) \in R\ s \implies (ti,t) \in R\ s \implies (ei,e) \in R\ s \implies$
 $(ri, ifex-ite-opt\ i\ t\ e) \in R\ s \implies \text{map-invar-impl}\ (m((ii,ti,ei) \mapsto ri))\ s$
 <proof>

end

locale *bdd-impl-cmp* = *bdd-impl* + *bdd-impl-cmp-pre* +

fixes *M* :: 'a \Rightarrow ('b \times 'b \times 'b) \Rightarrow 'b *option*

fixes *U* :: 'a \Rightarrow ('b \times 'b \times 'b) \Rightarrow 'b \Rightarrow 'a

fixes *cmp* :: 'b \Rightarrow 'b \Rightarrow bool

assumes *cmp-rule1*: *I s* \implies (*ni, i*) $\in R\ s \implies$ (*ni', i*) $\in R\ s \implies$ *cmp* *ni ni'*

assumes *cmp-rule2*: *I s* \implies *cmp* *ni ni'* \implies (*ni, i*) $\in R\ s \implies$ (*ni', i'*) $\in R\ s$
 $\implies i = i'$

assumes *map-invar-rule1*: *I s* \implies *map-invar-impl* (*M s*) *s*

assumes *map-invar-rule2*: *I s* \implies (*ii,it*) $\in R\ s \implies$ (*ti,tt*) $\in R\ s \implies$ (*ei,et*) $\in R\ s \implies$

$(ri, ifex-ite-opt\ it\ tt\ et) \in R\ s \implies U\ s\ (ii,ti,ei)\ ri = s' \implies I\ s'$

assumes *map-invar-rule3*: *I s* $\implies R\ (U\ s\ (ii, ti, ei)\ ri) = R\ s$

begin

lemma *cmp-rule-eq*: *I s* \implies (*ni, i*) $\in R\ s \implies$ (*ni', i'*) $\in R\ s \implies$ *cmp* *ni ni'* \longleftrightarrow *i = i'*

<proof>

lemma *DESTRimpl-Some*: *I s* \implies (*ni, i*) $\in R\ s \implies$ *ospec* (*DESTRimpl* *ni s*)

($\lambda r. \text{True}$)

<proof>

fun *param-opt-impl* **where**

param-opt-impl *i t e s* = *do* {

ii \leftarrow *DESTRimpl* *i s*;

ti \leftarrow *DESTRimpl* *t s*;

ei \leftarrow *DESTRimpl* *e s*;

(*tn,s*) \leftarrow *Timpl* *s*;

(*fn,s*) \leftarrow *Fimpl* *s*;

```

Some ((if ii = TD then Some t else
if ii = FD then Some e else
if ti = TD ∧ ei = FD then Some i else
if cmp t e then Some t else
if ei = TD ∧ cmp i t then Some tn else
if ti = FD ∧ cmp i e then Some fn else
None), s)}

```

declare *param-opt-impl.simps*[*simp del*]

lemma *param-opt-impl-lesI*:

```

assumes I s (ii,i) ∈ R s (ti,t) ∈ R s (ei,e) ∈ R s
shows ospec (param-opt-impl ii ti ei s)
          (λ(r,s′). I s′ ∧ les s s′)
⟨proof⟩

```

lemma *param-opt-impl-R*:

```

assumes I s (ii,i) ∈ R s (ti,t) ∈ R s (ei,e) ∈ R s
shows ospec (param-opt-impl ii ti ei s)
          (λ(r,s′). case r of None ⇒ param-opt i t e = None
                    | Some r ⇒ (∃ r′. param-opt i t e = Some r′ ∧ (r, r′)
                    ∈ R s′))
⟨proof⟩

```

partial-function(*option*) *ite-impl-opt* **where**

```

ite-impl-opt i t e s = do {
  (ld, s) ← param-opt-impl i t e s;
  (case ld of Some b ⇒ Some (b, s) |
  None ⇒
  do {
    (lt,-) ← lowest-tops-impl [i, t, e] s;
    (case lt of
    Some a ⇒ do {
      (ti,-) ← restrict-top-impl i a True s;
      (tt,-) ← restrict-top-impl t a True s;
      (te,-) ← restrict-top-impl e a True s;
      (fi,-) ← restrict-top-impl i a False s;
      (ft,-) ← restrict-top-impl t a False s;
      (fe,-) ← restrict-top-impl e a False s;
      (tb,s) ← ite-impl-opt ti tt te s;
      (fb,s) ← ite-impl-opt fi ft fe s;
      IFimpl a tb fb s}
    | None ⇒ case-ifexi (λ-.(Some (t,s))) (λ-.(Some (e,s))) (λ- - - . None) i s
    )}})
}

```

lemma *ospec-and*: *ospec f P ⇒ ospec f Q ⇒ ospec f (λx. P x ∧ Q x)*
 ⟨proof⟩

lemma *ite-impl-opt-R*:

```

  I s
  ⇒ in-rel (R s) ii i ⇒ in-rel (R s) ti t ⇒ in-rel (R s) ei e
  ⇒ ospec (ite-impl-opt ii ti ei s) (λ(r, s'). (r, ifex-ite-opt i t e) ∈ R s' ∧ I s' ∧
les s s')
⟨proof⟩

```

```

partial-function(option) ite-impl-lu where
ite-impl-lu i t e s = do {
  (case M s (i,t,e) of Some b ⇒ Some (b,s) | None ⇒ do {
    (ld, s) ← param-opt-impl i t e s;
    (case ld of Some b ⇒ Some (b, s) |
None ⇒
do {
  (lt,-) ← lowest-tops-impl [i, t, e] s;
  (case lt of
Some a ⇒ do {
    (ti,-) ← restrict-top-impl i a True s;
    (tt,-) ← restrict-top-impl t a True s;
    (te,-) ← restrict-top-impl e a True s;
    (fi,-) ← restrict-top-impl i a False s;
    (ft,-) ← restrict-top-impl t a False s;
    (fe,-) ← restrict-top-impl e a False s;
    (tb,s) ← ite-impl-lu ti tt te s;
    (fb,s) ← ite-impl-lu fi ft fe s;
    (r,s) ← IFimpl a tb fb s;
    let s = U s (i,t,e) r;
    Some (r,s)
  } |
None ⇒ None
  )}}})}

```

```

declare ifex-ite-opt.simps[simp del]

```

```

lemma ite-impl-lu-R: I s
  ⇒ (ii,i) ∈ R s ⇒ (ti,t) ∈ R s ⇒ (ei,e) ∈ R s
  ⇒ ospec (ite-impl-lu ii ti ei s)
    (λ(r, s'). (r, ifex-ite-opt i t e) ∈ R s' ∧ I s' ∧ les s s')
⟨proof⟩

```

```

end
end

```

6 Pointermap

```

theory Pointer-Map
imports Main
begin

```

We need a datastructure that supports the following two operations:

- Given an element, it can construct a pointer (i.e., a small representation) of that element. It will always construct the same pointer for equal elements.
- Given a pointer, we can retrieve the element

record *'a pointermap* =
entries :: *'a list*
getentry :: *'a* ⇒ *nat option*

definition *pointermap-sane* *m* ≡ (*distinct* (*entries m*) ∧
(∀ *n* ∈ {..*length* (*entries m*)}. *getentry m* (*entries m* ! *n*) = *Some n*) ∧
(∀ *p i*. *getentry m p* = *Some i* → *entries m* ! *i* = *p* ∧ *i* < *length* (*entries m*)))

definition *empty-pointermap* ≡ (|*entries* = [], *getentry* = λ*p*. *None* |)

lemma *pointermap-empty-sane*[*simp*, *intro!*]: *pointermap-sane empty-pointermap*
⟨*proof*⟩

definition *pointermap-insert* *a m* ≡ (|*entries* = (*entries m*)@[*a*], *getentry* = (*getentry m*)(*a* ↦ *length* (*entries m*)) |)

definition *pm-pth* *m p* ≡ *entries m* ! *p*

definition *pointermap-p-valid* *p m* ≡ *p* < *length* (*entries m*)

definition *pointermap-getmk* *a m* ≡ (*case getentry m a of Some p* ⇒ (*p,m*) | *None*
⇒ *let u = pointermap-insert a m in (the (getentry u a), u)*)

lemma *pointermap-sane-appendD*: *pointermap-sane s* ⇒ *m* ∉ *set (entries s)* ⇒
pointermap-sane (pointermap-insert m s)
⟨*proof*⟩

lemma *lentries-noneD*: *getentry s a = None* ⇒ *pointermap-sane s* ⇒ *a* ∉ *set*
(*entries s*)
⟨*proof*⟩

lemma *pm-pth-append*: *pointermap-p-valid p m* ⇒ *pm-pth (pointermap-insert a m)* *p* = *pm-pth m p*
⟨*proof*⟩

lemma *pointermap-insert-in*: *u = (pointermap-insert a m)* ⇒ *pm-pth u (the*
(*getentry u a*)) = *a*
⟨*proof*⟩

lemma *pointermap-insert-p-validI*: *pointermap-p-valid p m* ⇒ *pointermap-p-valid*
p (pointermap-insert a m)
⟨*proof*⟩

thm *nth-eq-iff-index-eq*

lemma *pth-eq-iff-index-eq*: $\text{pointermap-sane } m \implies \text{pointermap-p-valid } p1 \ m \implies \text{pointermap-p-valid } p2 \ m \implies (\text{pm-pth } m \ p1 = \text{pm-pth } m \ p2) \longleftrightarrow (p1 = p2)$
 ⟨proof⟩

lemma *pointermap-p-valid-updateI*: $\text{pointermap-sane } m \implies \text{getentry } m \ a = \text{None} \implies u = \text{pointermap-insert } a \ m \implies p = \text{the } (\text{getentry } u \ a) \implies \text{pointermap-p-valid } p \ u$
 ⟨proof⟩

lemma *pointermap-get-validI*: $\text{pointermap-sane } m \implies \text{getentry } m \ a = \text{Some } p \implies \text{pointermap-p-valid } p \ m$
 ⟨proof⟩

lemma *pointermap-sane-getmkD*:
 assumes *sn*: $\text{pointermap-sane } m$
 assumes *res*: $\text{pointermap-getmk } a \ m = (p, u)$
 shows $\text{pointermap-sane } u \wedge \text{pointermap-p-valid } p \ u$
 ⟨proof⟩

lemma *pointermap-update-pthI*:
 assumes *sn*: $\text{pointermap-sane } m$
 assumes *res*: $\text{pointermap-getmk } a \ m = (p, u)$
 shows $\text{pm-pth } u \ p = a$
 ⟨proof⟩

lemma *pointermap-p-valid-inv*:
 assumes $\text{pointermap-p-valid } p \ m$
 assumes $\text{pointermap-getmk } a \ m = (x, u)$
 shows $\text{pointermap-p-valid } p \ u$
 ⟨proof⟩

lemma *pointermap-p-pth-inv*:
 assumes *pv*: $\text{pointermap-p-valid } p \ m$
 assumes *u*: $\text{pointermap-getmk } a \ m = (x, u)$
 shows $\text{pm-pth } u \ p = \text{pm-pth } m \ p$
 ⟨proof⟩

lemma *pointermap-backward-valid*:
 assumes *puv*: $\text{pointermap-p-valid } p \ u$
 assumes *u*: $\text{pointermap-getmk } a \ m = (x, u)$
 assumes *ne*: $x \neq p$
 shows $\text{pointermap-p-valid } p \ m$

⟨proof⟩

end

7 Functional interpretation for the abstract implementation

```

theory Middle-Impl
imports Abstract-Impl Pointer-Map
begin

```

For the lack of a better name, the suffix *mi* stands for middle-implementation. This reflects that this “implementation” is neither entirely abstract, nor has it been made fully concrete: the data structures are decided, but not their implementations.

```

record bdd =
  dpm :: (nat × nat × nat) pointermap
  dcl :: ((nat × nat × nat), nat) map

```

definition *emptymi* \equiv $(\lambda dpm = \text{empty-pointermap}, dcl = \text{Map.empty})$

```

fun destrmi :: nat ⇒ bdd ⇒ (nat, nat) IFEXD where
  destrmi 0 bdd = FD |
  destrmi (Suc 0) bdd = TD |
  destrmi (Suc (Suc n)) bdd = (case pm-pth (dpm bdd) n of (v, t, e) ⇒ IFD v t e)
fun tmi where tmi bdd = (1, bdd)
fun fmi where fmi bdd = (0, bdd)
fun ifmi :: nat ⇒ nat ⇒ nat ⇒ bdd ⇒ (nat × bdd) where
  ifmi v t e bdd = (if t = e
    then (t, bdd)
    else (let (r, pm) = pointermap-getmk (v, t, e) (dpm bdd) in
      (Suc (Suc r), dpm-update (const pm) bdd)))

```

```

fun Rmi-g :: nat ⇒ nat ifex ⇒ bdd ⇒ bool where
  Rmi-g 0 Falseif bdd = True |
  Rmi-g (Suc 0) Trueif bdd = True |
  Rmi-g (Suc (Suc n)) (IF v t e) bdd = (pointermap-p-valid n (dpm bdd)
    ∧ (case pm-pth (dpm bdd) n of (nv, nt, ne) ⇒ nv = v ∧ Rmi-g nt t bdd ∧ Rmi-g
      ne e bdd)) |
  Rmi-g - - - = False

```

definition *Rmi s* \equiv $\{(a,b) \mid a \ b. \ Rmi\text{-}g \ a \ b \ s\}$

interpretation *mi-pre*: *bdd-impl-cmp-pre* *Rmi* \langle proof \rangle

definition *bdd-node-valid* *bdd n* \equiv $n \in \text{Domain} (Rmi \ bdd)$

```

lemma [simp]:
  bdd-node-valid bdd 0
  bdd-node-valid bdd (Suc 0)
   $\langle$ proof $\rangle$ 

```

definition *ifexd-valid* *bdd e* \equiv $(\text{case } e \text{ of } IFD \ - \ t \ e \Rightarrow \text{bdd-node-valid } bdd \ t \ \wedge \ \text{bdd-node-valid } bdd \ e \ | \ - \Rightarrow \text{True})$

definition $bdd\text{-sane } bdd \equiv pointermap\text{-sane } (dpm \ bdd) \wedge mi\text{-pre.map-invar-impl } (dcl \ bdd) \ bdd$

lemma $[simp,intro!]: bdd\text{-sane } emptymi$
 $\langle proof \rangle$

lemma $prod\text{-split3}: P \ (case \ p \ of \ (x, \ x_a, \ x_{aa}) \Rightarrow \ f \ x \ x_a \ x_{aa}) = (\forall \ x_1 \ x_2 \ x_3. \ p = (x_1, \ x_2, \ x_3) \longrightarrow P \ (f \ x_1 \ x_2 \ x_3))$
 $\langle proof \rangle$

lemma $IFI: (c \Longrightarrow P \ x) \Longrightarrow (\neg c \Longrightarrow P \ y) \Longrightarrow P \ (if \ c \ then \ x \ else \ y) \langle proof \rangle$

lemma $fstsndI: x = (a, b) \Longrightarrow fst \ x = a \wedge snd \ x = b \langle proof \rangle$

thm $nat.split$

lemma $Rmi\text{-g-2-split}: P \ (Rmi\text{-g} \ n \ x \ m) = ((x = Falseif \longrightarrow P \ (Rmi\text{-g} \ n \ x \ m)) \wedge (x = Trueif \longrightarrow P \ (Rmi\text{-g} \ n \ x \ m)) \wedge (\forall \ vs \ ts \ es. \ x = IF \ vs \ ts \ es \longrightarrow P \ (Rmi\text{-g} \ n \ x \ m)))$
 $\langle proof \rangle$

lemma $rmigeq: Rmi\text{-g} \ ni1 \ n1 \ s \Longrightarrow Rmi\text{-g} \ ni2 \ n2 \ s \Longrightarrow ni1 = ni2 \Longrightarrow n1 = n2$
 $\langle proof \rangle$

lemma $rmigneq: bdd\text{-sane} \ s \Longrightarrow Rmi\text{-g} \ ni1 \ n1 \ s \Longrightarrow Rmi\text{-g} \ ni2 \ n2 \ s \Longrightarrow ni1 \neq ni2 \Longrightarrow n1 \neq n2$
 $\langle proof \rangle$

lemma $ifmi\text{-les-hlp}: pointermap\text{-sane} \ (dpm \ s) \Longrightarrow pointermap\text{-getmk} \ (v, \ ni1, \ ni2) \ (dpm \ s) = (x1, \ dpm \ s') \Longrightarrow Rmi\text{-g} \ nia \ n \ s \Longrightarrow Rmi\text{-g} \ nia \ n \ s'$
 $\langle proof \rangle$

lemma $ifmi\text{-les}$:

assumes $bdd\text{-sane} \ s$
assumes $ifmi \ v \ ni1 \ ni2 \ s = (ni, \ s')$
shows $mi\text{-pre.les} \ s \ s'$

$\langle proof \rangle$

lemma $ifmi\text{-notouch-dcl}: ifmi \ v \ ni1 \ ni2 \ s = (ni, \ s') \Longrightarrow dcl \ s' = dcl \ s$
 $\langle proof \rangle$

lemma $ifmi\text{-saneI}: bdd\text{-sane} \ s \Longrightarrow ifmi \ v \ ni1 \ ni2 \ s = (ni, \ s') \Longrightarrow bdd\text{-sane} \ s'$
 $\langle proof \rangle$

lemma $rmigif: Rmi\text{-g} \ ni \ (IF \ v \ n1 \ n2) \ s \Longrightarrow \exists \ n. \ ni = Suc \ (Suc \ n)$
 $\langle proof \rangle$

lemma $in\text{-lesI}$:

assumes $mi\text{-pre.les} \ s \ s'$
assumes $(ni1, \ n1) \in Rmi \ s$
assumes $(ni2, \ n2) \in Rmi \ s$

shows $(ni1, n1) \in Rmi\ s' \ (ni2, n2) \in Rmi\ s'$
 $\langle proof \rangle$

lemma *ifmi-modification-validI*:

assumes *sane*: *bdd-sane* *s*
assumes *ifm*: *ifmi* *v* *ni1* *ni2* *s* = (ni, s')
assumes *vld*: *bdd-node-valid* *s* *n*
shows *bdd-node-valid* *s'* *n*

$\langle proof \rangle$

definition *tmi'* *s* $\equiv do \{oassert \ (bdd-sane \ s); \ Some \ (tmi \ s)\}$

definition *fmi'* *s* $\equiv do \{oassert \ (bdd-sane \ s); \ Some \ (fmi \ s)\}$

definition *ifmi'* *v* *ni1* *ni2* *s* $\equiv do \{oassert \ (bdd-sane \ s \wedge \ bdd-node-valid \ s \ ni1 \wedge \ bdd-node-valid \ s \ ni2); \ Some \ (ifmi \ v \ ni1 \ ni2 \ s)\}$

lemma *ifmi'-spec*: $\llbracket bdd-sane \ s; \ bdd-node-valid \ s \ ni1; \ bdd-node-valid \ s \ ni2 \rrbracket \implies$
 $ospec \ (ifmi' \ v \ ni1 \ ni2 \ s) \ (\lambda r. \ r = ifmi \ v \ ni1 \ ni2 \ s)$
 $\langle proof \rangle$

lemma *ifmi'-ifmi*: $\llbracket bdd-sane \ s; \ bdd-node-valid \ s \ ni1; \ bdd-node-valid \ s \ ni2 \rrbracket \implies$
 $ifmi' \ v \ ni1 \ ni2 \ s = Some \ (ifmi \ v \ ni1 \ ni2 \ s)$
 $\langle proof \rangle$

definition *destrmi'* *ni* *s* $\equiv do \{oassert \ (bdd-sane \ s \wedge \ bdd-node-valid \ s \ ni); \ Some \ (destrmi \ ni \ s)\}$

lemma *destrmi-someD*: $destrmi' \ e \ bdd = Some \ x \implies bdd-sane \ bdd \wedge \ bdd-node-valid \ bdd \ e$
 $\langle proof \rangle$

lemma *Rmi-sv*:

assumes *bdd-sane* *s* $(ni, n) \in Rmi \ s \ (ni', n') \in Rmi \ s$
shows $ni=ni' \implies n=n'$
and $ni \neq ni' \implies n \neq n'$
 $\langle proof \rangle$

lemma *True-rep[simp]*: $bdd-sane \ s \implies (ni, Trueif) \in Rmi \ s \longleftrightarrow ni = Suc \ 0$
 $\langle proof \rangle$

lemma *False-rep[simp]*: $bdd-sane \ s \implies (ni, Falseif) \in Rmi \ s \longleftrightarrow ni = 0$
 $\langle proof \rangle$

definition *updS* *s* *x* *r* = *dcl-update* $(\lambda m. \ m(x \mapsto r)) \ s$

thm *Rmi-g.induct*

lemma *updS-dpm*: $dpm \ (updS \ s \ x \ r) = dpm \ s$
 $\langle proof \rangle$

lemma *updS-Rmi-g*: $Rmi-g \ n \ i \ (updS \ s \ x \ r) = Rmi-g \ n \ i \ s$

<proof>

lemma *updS-Rmi*: $Rmi (updS s x r) = Rmi s$
<proof>

interpretation *mi*: *bdd-impl-cmp bdd-sane Rmi tmi' fmi' ifmi' destrmi' dcl updS op =*
<proof>

lemma *p-valid-RmiI*: $(Suc (Suc na), b) \in Rmi bdd \implies pointermap-p-valid na (dpm bdd)$
<proof>

lemma *n-valid-RmiI*: $(na, b) \in Rmi bdd \implies bdd-node-valid bdd na$
<proof>

lemma *n-valid-Rmi-alt*: $bdd-node-valid bdd na \longleftrightarrow (\exists b. (na, b) \in Rmi bdd)$
<proof>

lemma *ifmi-result-validI*:

assumes *sane*: *bdd-sane s*

assumes *vld*: *bdd-node-valid s ni1 bdd-node-valid s ni2*

assumes *ifm*: *ifmi v ni1 ni2 s = (ni, s')*

shows *bdd-node-valid s' ni*

<proof>

end

8 Array List

Most of this has been contributed by Peter Lammich.

theory *Array-List*

imports

../Separation-Logic-Imperative-HOL/Examples/Array-Blit

begin

This implements a datastructure that efficiently supports two operations: appending an element and looking up the *n*th element. The implementation is straightforward.

As underlying data structure an array is used. Since changing the length of an array requires copying, we double the size whenever the array needs to be expanded. We use a counter for the current length to track which elements are used and which are spares.

type-synonym *'a array-list* = *'a array* \times *nat*

definition *is-array-list* $l \equiv \lambda(a,n). \exists_A l'. a \mapsto_a l' * \uparrow(n \leq \text{length } l' \wedge l = \text{take } n l' \wedge \text{length } l' > 0)$

definition *initial-capacity* $\equiv 16::nat$

definition *arl-empty* $\equiv do \{$
 $a \leftarrow Array.new\ initial-capacity\ default;$
 $return\ (a,0)$
 $\}$

lemma [*sep-heap-rules*]: $\langle emp \rangle arl-empty \langle is-array-list\ [] \rangle$
 $\langle proof \rangle$

definition *arl-nth* $\equiv \lambda(a,n)\ i.\ do \{$
 $Array.nth\ a\ i$
 $\}$

lemma [*sep-heap-rules*]: $i < length\ l \implies \langle is-array-list\ l\ a \rangle arl-nth\ a\ i < \lambda x.$
 $is-array-list\ l\ a * \uparrow(x = l!i) >$
 $\langle proof \rangle$

definition *arl-append* $\equiv \lambda(a,n)\ x.\ do \{$
 $len \leftarrow Array.len\ a;$

 $if\ n < len\ then\ do \{$
 $a \leftarrow Array.upd\ n\ x\ a;$
 $return\ (a, n+1)$
 $\}$ $else\ do \{$
 $let\ newcap = 2 * len;$
 $a \leftarrow array-grow\ a\ newcap\ default;$
 $a \leftarrow Array.upd\ n\ x\ a;$
 $return\ (a, n+1)$
 $\}$
 $\}$

lemma [*sep-heap-rules*]:
 $\langle is-array-list\ l\ a \rangle$
 $arl-append\ a\ x$
 $\langle \lambda a.\ is-array-list\ (l@[x])\ a \rangle_t$
 $\langle proof \rangle$

lemma *is-array-list-prec*: *precise is-array-list*
 $\langle proof \rangle$

lemma *is-array-list-lengthIA*: $is-array-list\ l\ li \implies_A \uparrow(snd\ li = length\ l) * true$
 $\langle proof \rangle$

find-consts *assn* $\Rightarrow bool$

lemma *is-array-list-lengthI*: $x \models is-array-list\ l\ li \implies snd\ li = length\ l$
 $\langle proof \rangle$

end

9 Imperative implementation for Pointermap

```

theory Pointer-Map-Impl
imports Array-List
  ../Separation-Logic-Imperative-HOL/Sep-Main
  ../Separation-Logic-Imperative-HOL/Examples/Hash-Map-Impl
  Pointer-Map
begin

  record 'a pointermap-impl =
    entriesi :: 'a array-list
    getentryi :: ('a,nat) hashtable
  lemma pointermapieq-exhaust: entries a = entries b  $\implies$  getentry a = getentry b  $\implies a = (b :: 'a pointermap)$  <proof>

  definition is-pointermap-impl :: ('a::{hashable,heap}) pointermap  $\Rightarrow$  'a pointermap-impl
 $\Rightarrow$  assn where
    is-pointermap-impl b bi  $\equiv$ 
      is-array-list (entries b) (entriesi bi)
      * is-hashmap (getentry b) (getentryi bi)

  lemma is-pointermap-impl-prec: precise is-pointermap-impl
    <proof>

  definition pointermap-empty where
    pointermap-empty  $\equiv$  do {
      hm  $\leftarrow$  hm-new;
      arl  $\leftarrow$  arl-empty;
      return (entriesi = arl, getentryi = hm )
    }

  lemma [sep-heap-rules]: < emp > pointermap-empty <is-pointermap-impl empty-pointermap>t
    <proof>

  definition pm-pthi where
    pm-pthi m p  $\equiv$  arl-nth (entriesi m) p

  lemma [sep-heap-rules]: pointermap-sane m  $\implies$  pointermap-p-valid p m  $\implies$ 
    < is-pointermap-impl m mi > pm-pthi mi p < $\lambda$ ai. is-pointermap-impl m mi *
 $\uparrow$ (ai = pm-pth m p)>
    <proof>

  definition pointermap-getmki where
    pointermap-getmki a m  $\equiv$  do {
      lo  $\leftarrow$  ht-lookup a (getentryi m);
      (case lo of
        Some l  $\Rightarrow$  return (l,m) |
        None  $\Rightarrow$  do {
          p  $\leftarrow$  return (snd (entriesi m)));
    }

```

```

    ent ← arl-append (entriesi m) a;
    lut ← hm-update a p (getentryi m);
    u ← return (|entriesi = ent, getentryi = lut|);
    return (p,u)
  }
)
}

```

lemmas *pointermap-getmki-defs* = *pointermap-getmki-def* *pointermap-getmk-def*
pointermap-insert-def *is-pointermap-impl-def*

lemma [*sep-heap-rules*]: *pointermap-sane* $m \implies$ *pointermap-getmk* $a\ m = (p,u)$
 \implies
 \langle *is-pointermap-impl* $m\ mi \rangle$
pointermap-getmki $a\ mi$
 $\langle \lambda(pi,ui). is_pointermap_impl\ u\ ui * \uparrow(pi = p) \rangle_t$
 $\langle proof \rangle$

end

10 Imperative implementation

theory *Conc-Impl*

imports *Pointer-Map-Impl* *Middle-Impl*

begin

record *bddi* =

dpmi :: (*nat* × *nat* × *nat*) *pointermap-impl*

dcli :: ((*nat* × *nat* × *nat*), *nat*) *hashtable*

lemma *bdd-exhaust*: *dpm* $a = dpm\ b \implies dcl\ a = dcl\ b \implies a = (b :: bdd) \langle proof \rangle$

instantiation *prod* :: (*default*, *default*) *default*

begin

definition *default-prod* :: ('*a* × '*b*) ≡ (*default*, *default*)

instance $\langle proof \rangle$

end

instantiation *nat* :: *default*

begin

definition *default-nat* ≡ 0 :: *nat*

instance $\langle proof \rangle$

end

definition *is-bdd-impl* (*bdd*::*bdd*) (*bddi*::*bddi*) = *is-pointermap-impl* (*dpm* *bdd*)
(*dpmi* *bddi*) * *is-hashmap* (*dcl* *bdd*) (*dcli* *bddi*)

lemma *is-bdd-impl-prec*: *precise* *is-bdd-impl*

$\langle proof \rangle$

definition *emptyci* :: *bddi* *Heap* ≡ *do* { *ep* ← *pointermap-empty*; *ehm* ← *hm-new*;

return ($\langle dpmi=ep, dcli=ehm \rangle$) }

definition *tci* *bdd* \equiv *return* ($1::nat, bdd::bddi$)

definition *fci* *bdd* \equiv *return* ($0::nat, bdd::bddi$)

definition *ifci* *v t e bdd* \equiv (if $t = e$ then *return* (t, bdd) else do {
 $(p, u) \leftarrow \text{pointermap-getmki } (v, t, e) (dpmi \text{ bdd});$
return (*Suc* (*Suc* p), *dpmi-update* (*const* u) *bdd*)
 })

definition *destrci* $:: nat \Rightarrow bddi \Rightarrow (nat, nat)$ *IFEXD Heap* **where**

destrci $n \text{ bdd} \equiv$ (case n of

$0 \Rightarrow$ *return* *FD* |

Suc $0 \Rightarrow$ *return* *TD* |

Suc (*Suc* p) \Rightarrow *pm-pthi* (*dpmi bdd*) $p \gg= (\lambda(v,t,e). \text{return } (IFD \ v \ t \ e))$)

term *mi.les*

lemma *emptyci-rule*[*sep-heap-rules*]: $\langle emp \rangle \text{emptyci } \langle is\text{-bdd-impl } \text{emptymi} \rangle_t$
 $\langle proof \rangle$

lemma [*sep-heap-rules*]: *tmi'* *bdd* = *Some* (p, bdd')
 $\Rightarrow \langle is\text{-bdd-impl } bdd \ bddi \rangle$
 $\quad tci \ bddi$
 $\quad \langle \lambda(pi, bddi'). is\text{-bdd-impl } bdd' \ bddi' * \uparrow(pi = p) \rangle$
 $\langle proof \rangle$

lemma [*sep-heap-rules*]: *fmi'* *bdd* = *Some* (p, bdd')
 $\Rightarrow \langle is\text{-bdd-impl } bdd \ bddi \rangle$
 $\quad fci \ bddi$
 $\quad \langle \lambda(pi, bddi'). is\text{-bdd-impl } bdd' \ bddi' * \uparrow(pi = p) \rangle$
 $\langle proof \rangle$

lemma [*sep-heap-rules*]: *ifmi'* *v t e bdd* = *Some* (p, bdd') \Rightarrow
 $\langle is\text{-bdd-impl } bdd \ bddi \rangle \text{ifci } v \ t \ e \ bddi$
 $\langle \lambda(pi, bddi'). is\text{-bdd-impl } bdd' \ bddi' * \uparrow(pi = p) \rangle_t$
 $\langle proof \rangle$

lemma *destrci-rule*[*sep-heap-rules*]:
 $\text{destrmi}' \ n \ bdd = \text{Some } r \Rightarrow$
 $\langle is\text{-bdd-impl } bdd \ bddi \rangle \text{destrci } n \ bddi$
 $\langle \lambda r'. is\text{-bdd-impl } bdd \ bddi * \uparrow(r' = r) \rangle$
 $\langle proof \rangle$

term *mi.restrict-top-impl*

thm *mi.case-ifexi-def*

definition *case-ifexici* *fti ffi fii ni bddi* \equiv do {
 $\text{dest} \leftarrow \text{destrci } ni \ bddi;$
 case *dest* of *TD* \Rightarrow *fti* | *FD* \Rightarrow *ffi* | *IFD* $v \ ti \ ei \Rightarrow$ *fii* $v \ ti \ ei$
 }

lemma [sep-decon-rules]:
assumes S : $mi.case-ifexi\ fti\ ffi\ fui\ ni\ bdd = Some\ r$
assumes [sep-heap-rules]:
 $destrmi'\ ni\ bdd = Some\ TD \implies fti\ bdd = Some\ r \implies \langle is-bdd-impl\ bdd\ bddi \rangle$
 $ftci\ \langle Q \rangle$
 $destrmi'\ ni\ bdd = Some\ FD \implies ffi\ bdd = Some\ r \implies \langle is-bdd-impl\ bdd\ bddi \rangle$
 $ffci\ \langle Q \rangle$
 $\bigwedge v\ t\ e.\ destrmi'\ ni\ bdd = Some\ (IFD\ v\ t\ e) \implies fui\ v\ t\ e\ bdd = Some\ r$
 $\implies \langle is-bdd-impl\ bdd\ bddi \rangle\ fici\ v\ t\ e\ \langle Q \rangle$
shows $\langle is-bdd-impl\ bdd\ bddi \rangle\ case-ifexi\ fti\ ffi\ fui\ ni\ bddi\ \langle Q \rangle$
 $\langle proof \rangle$

definition $restrict-topci\ p\ vr\ vl\ bdd =$
 $case-ifexi$
 $(return\ p)$
 $(return\ p)$
 $(\lambda v\ te\ ee.\ return\ (if\ v = vr\ then\ (if\ vl\ then\ te\ else\ ee)\ else\ p))$
 $p\ bdd$

lemma [sep-heap-rules]:
assumes $mi.restrict-top-impl\ p\ var\ val\ bdd = Some\ (r, bdd')$
shows $\langle is-bdd-impl\ bdd\ bddi \rangle\ restrict-topci\ p\ var\ val\ bddi$
 $\langle \lambda ri.\ is-bdd-impl\ bdd\ bddi * \uparrow(ri = r) \rangle$
 $\langle proof \rangle$

fun $lowest-topsci\ where$
 $lowest-topsci\ []\ s = return\ None\ |$
 $lowest-topsci\ (e\#es)\ s =$
 $case-ifexi$
 $(lowest-topsci\ es\ s)$
 $(lowest-topsci\ es\ s)$
 $(\lambda v\ t\ e.\ do\ \{$
 $(rec) \leftarrow lowest-topsci\ es\ s;$
 $(case\ rec\ of$
 $Some\ u \Rightarrow return\ ((Some\ (min\ u\ v)))\ |$
 $None \Rightarrow return\ ((Some\ v)))$
 $\})\ e\ s$

declare $lowest-topsci.simps[simp\ del]$

lemma [sep-heap-rules]:
assumes $mi.lowest-tops-impl\ es\ bdd = Some\ (r, bdd')$
shows $\langle is-bdd-impl\ bdd\ bddi \rangle\ lowest-topsci\ es\ bddi$
 $\langle \lambda(ri).\ is-bdd-impl\ bdd\ bddi * \uparrow(ri = r \wedge bdd' = bdd) \rangle$
 $\langle proof \rangle$

partial-function($heap$) $iteci\ where$

```

iteci i t e s = do {
  (lt) ← lowest-topsci [i, t, e] s;
  case lt of
    Some a ⇒ do {
      ti ← restrict-topci i a True s;
      tt ← restrict-topci t a True s;
      te ← restrict-topci e a True s;
      fi ← restrict-topci i a False s;
      ft ← restrict-topci t a False s;
      fe ← restrict-topci e a False s;
      (tb,s') ← iteci ti tt te s;
      (fb,s'') ← iteci fi ft fe s';
      (ifci a tb fb s'')
    }
  | None ⇒ do {
    case-ifexici (return (t,s)) (return (e,s)) (λ- - -. raise "Cannot happen") i s
  }
}
declare iteci.simps[code]

```

lemma iteci-rule:

```

(mi.ite-impl i t e bdd = Some (p,bdd')) →
<is-bdd-impl bdd bddi>
  iteci i t e bddi
<λ(pi,bddi'). is-bdd-impl bdd' bddi' * ↑(pi=p )>_t
<proof>

```

declare iteci-rule[THEN mp, sep-heap-rules]

definition param-optci **where**

```

param-optci i t e bdd = do {
  (tr, bdd) ← tci bdd;
  (fl, bdd) ← fci bdd;
  id ← destrci i bdd;
  td ← destrci t bdd;
  ed ← destrci e bdd;
  return (
    if id = TD then Some t else
    if id = FD then Some e else
    if td = TD ∧ ed = FD then Some i else
    if t = e then Some t else
    if ed = TD ∧ i = t then Some tr else
    if td = FD ∧ i = e then Some fl else
    None, bdd)
}

```

lemma param-optci-rule:

```

(mi.param-opt-impl i t e bdd = Some (p,bdd')) ⇒
<is-bdd-impl bdd bddi>

```


$\text{param-optci } i \ t \ e \ \text{bddi}$
 $\langle \lambda(pi, \text{bddi}'). \text{is-bdd-impl } \text{bdd}' \ \text{bddi}' * \uparrow(pi=p) \rangle_t$
 $\langle \text{proof} \rangle$

lemma $\text{bdd-hm-lookup-rule}$:
 $(\text{dcl } \text{bdd } (i, t, e) = p) \implies$
 $\langle \text{is-bdd-impl } \text{bdd } \text{bddi} \rangle$
 $\text{hm-lookup } (i, t, e) \ (\text{dcli } \text{bddi})$
 $\langle \lambda(pi). \text{is-bdd-impl } \text{bdd } \text{bddi} * \uparrow(pi = p) \rangle_t$
 $\langle \text{proof} \rangle$

lemma $\text{bdd-hm-update-rule}^{\text{[sep-heap-rules]}}$:
 $\langle \text{is-bdd-impl } \text{bdd } \text{bddi} \rangle$
 $\text{hm-update } k \ v \ (\text{dcli } \text{bddi})$
 $\langle \lambda r. \text{is-bdd-impl } (\text{updS } \text{bdd } k \ v) \ (\text{dcli-update } (\text{const } r) \ \text{bddi}) * \text{true} \rangle$
 $\langle \text{proof} \rangle$

partial-function(heap) iteci-lu **where**
 $\text{iteci-lu } i \ t \ e \ s = \text{do } \{$
 $\text{lu} \leftarrow \text{ht-lookup } (i, t, e) \ (\text{dcli } s);$
 $(\text{case } \text{lu} \ \text{of } \text{Some } b \Rightarrow \text{return } (b, s)$
 $\mid \text{None} \Rightarrow \text{do } \{$
 $(po, s) \leftarrow \text{param-optci } i \ t \ e \ s;$
 $(\text{case } po \ \text{of } \text{Some } b \Rightarrow \text{do } \{$
 $\text{return } (b, s) \}$
 $\mid \text{None} \Rightarrow \text{do } \{$
 $(lt) \leftarrow \text{lowest-topsci } [i, t, e] \ s;$
 $(\text{case } lt \ \text{of } \text{Some } a \Rightarrow \text{do } \{$
 $ti \leftarrow \text{restrict-topci } i \ a \ \text{True } s;$
 $tt \leftarrow \text{restrict-topci } t \ a \ \text{True } s;$
 $te \leftarrow \text{restrict-topci } e \ a \ \text{True } s;$
 $fi \leftarrow \text{restrict-topci } i \ a \ \text{False } s;$
 $ft \leftarrow \text{restrict-topci } t \ a \ \text{False } s;$
 $fe \leftarrow \text{restrict-topci } e \ a \ \text{False } s;$
 $(tb, s) \leftarrow \text{iteci-lu } ti \ tt \ te \ s;$
 $(fb, s) \leftarrow \text{iteci-lu } fi \ ft \ fe \ s;$
 $(r, s) \leftarrow \text{ifci } a \ tb \ fb \ s;$
 $cl \leftarrow \text{hm-update } (i, t, e) \ r \ (\text{dcli } s);$
 $\text{return } (r, \text{dcli-update } (\text{const } cl) \ s)$
 $\}$
 $\mid \text{None} \Rightarrow \text{raise } \text{"Cannot happen"} \ \}) \}$
 $\}} \}$

term ht-lookup

declare $\text{iteci-lu.simps}[\text{code}]$

thm $\text{iteci-lu.simps}[\text{unfolded } \text{restrict-topci-def } \text{case-ifexici-def } \text{param-optci-def } \text{lowest-topsci.simps}]$

partial-function(heap) iteci-lu-code **where** $\text{iteci-lu-code } i \ t \ e \ s = \text{do } \{$

$\text{lu} \leftarrow \text{hm-lookup } (i, t, e) \ (\text{dcli } s);$

$\text{case } \text{lu} \ \text{of } \text{None} \Rightarrow \text{let } po = \text{if } i = 1 \ \text{then } \text{Some } t$

```

else if i = 0 then Some e else if t = 1 ∧ e = 0 then Some
i else if t = e then Some t else if e = 1 ∧ i = t then Some 1 else if t = 0 ∧ i =
e then Some 0 else None
  in case po of None ⇒ do {
    id ← destrci i s;
    td ← destrci t s;
    ed ← destrci e s;
    let a = (case id of IFD v t e ⇒ v);
    let a = (case td of IFD v t e ⇒ min a v | - ⇒ a);
    let a = (case ed of IFD v t e ⇒ min a v | - ⇒ a);
    let ti = (case id of IFD v ti ei ⇒ if v = a then ti
else i | - ⇒ i);
    let tt = (case td of IFD v ti ei ⇒ if v = a then ti
else t | - ⇒ t);
    let te = (case ed of IFD v ti ei ⇒ if v = a then ti
else e | - ⇒ e);
    let fi = (case id of IFD v ti ei ⇒ if v = a then ei
else i | - ⇒ i);
    let ft = (case td of IFD v ti ei ⇒ if v = a then ei
else t | - ⇒ t);
    let fe = (case ed of IFD v ti ei ⇒ if v = a then ei
else e | - ⇒ e);
    (tb, s) ← iteci-lu-code ti tt te s;
    (fb, s) ← iteci-lu-code fi ft fe s;
    (r, s) ← ifci a tb fb s;
    cl ← hm-update (i, t, e) r (dcli s);
    return (r, dcli-update (const cl) s)
  }
| Some b ⇒ return (b, s)
}

```

declare *iteci-lu-code.simps*[code]

lemma *iteci-lu-code*[code-unfold]: *iteci-lu i t e s = iteci-lu-code i t e s*
⟨proof⟩

lemma *iteci-lu-rule*:

```

( mi.ite-impl-lu i t e bdd = Some (p,bdd') ) →
<is-bdd-impl bdd bddi>
  iteci-lu i t e bddi
<λ(pi,bddi). is-bdd-impl bdd' bddi' * ↑(pi=p)>_t
⟨proof⟩

```

10.1 A standard library of functions

declare *iteci-rule*[THEN mp, sep-heap-rules]

```

definition notci e s ≡ do {
  (f,s) ← fci s;
  (t,s) ← tci s;
  iteci-lu e f t s
}
definition orci e1 e2 s ≡ do {
  (t,s) ← tci s;
  iteci-lu e1 t e2 s
}
definition andci e1 e2 s ≡ do {
  (f,s) ← fci s;
  iteci-lu e1 e2 f s
}
definition norci e1 e2 s ≡ do {
  (r,s) ← orci e1 e2 s;
  notci r s
}
definition nandci e1 e2 s ≡ do {
  (r,s) ← andci e1 e2 s;
  notci r s
}
definition biimpci a b s ≡ do {
  (nb,s) ← notci b s;
  iteci-lu a b nb s
}
definition xorci a b s ≡ do {
  (nb,s) ← notci b s;
  iteci-lu a nb b s
}
definition litci v bdd ≡ do {
  (t,bdd) ← tci bdd;
  (f,bdd) ← fci bdd;
  ifci v t f bdd
}
definition tautci v bdd ≡ do {
  d ← destrci v bdd;
  return (d = TD)
}

```

10.2 Printing

The following functions are exported unverified. They are intended for BDD debugging purposes.

partial-function(*heap*) *serializeci* :: *nat* ⇒ *bddi* ⇒ ((*nat* × *nat*) × *nat*) *list Heap*

where

```

serializeci p s = do {
  d ← destrci p s;
  (case d of

```

```

    IFD v t e ⇒ do {
      r ← serializeci t s;
      l ← serializeci e s;
      return (remdups (((p,t),1),((p,e),0)] @ r @ l)
    } |
  - ⇒ return []
)
}
declare serializeci.simps[code]

fun mapM where
  mapM f [] = return [] |
  mapM f (a#as) = do {
    r ← f a;
    rs ← mapM f as;
    return (r#rs)
  }
definition liftM f ma = do { a ← ma; return (f a) }
definition sequence = mapM id
term liftM (map f)
lemma liftM (map f) (sequence l) = sequence (map (liftM f) l)
  ⟨proof⟩

```

```

fun string-of-nat :: nat ⇒ string where
  string-of-nat n = (if n < 10 then [char-of-nat (48 + n)]
    else string-of-nat (n div 10) @ [char-of-nat (48 + (n mod
  10))])

```

```

definition labelci :: bddi ⇒ nat ⇒ (string × string × string) Heap where
  labelci s n = do {
    d ← destrci n s;
    let son = string-of-nat n;
    let label = (case d of
      TD ⇒ "T" |
      FD ⇒ "F" |
      (IFD v -) ⇒ string-of-nat v);
    return (label, son, son @ "[label=" @ label @ "];
  ")
}

```

```

definition graphifyci1 bdd a ≡ do {
  let ((f,t),y) = a;
  let c = (string-of-nat f @ " -> " @ string-of-nat t);
  return (c @ (case y of 0 ⇒ "[style=dotted]" | Suc - ⇒ "")) @ "
  ")
}

```

```

definition trd = snd ∘ snd

```


lemma *bdd-relator-absorb-true*[simp]: $\text{bdd-relator } p \ s * \text{ true} = \text{bdd-relator } p \ s$
 ⟨proof⟩

thm *bdd-relator-def*[unfolded *bddmi-rel-def*, simplified]

lemma *join-hlp1*: $\text{is-bdd-impl } a \ s * \text{is-bdd-impl } b \ s \implies_A \text{is-bdd-impl } a \ s * \text{is-bdd-impl } b \ s * \uparrow(a = b)$
 ⟨proof⟩

lemma *join-hlp*: $\text{is-bdd-impl } a \ s * \text{is-bdd-impl } b \ s = \text{is-bdd-impl } b \ s * \text{is-bdd-impl } a \ s * \uparrow(a = b)$
 ⟨proof⟩

lemma *add-true-asm*:

assumes $\langle b * \text{true} \rangle \ p \ \langle a \rangle_t$

shows $\langle b \rangle \ p \ \langle a \rangle_t$

⟨proof⟩

lemma *add-anything*:

assumes $\langle b \rangle \ p \ \langle a \rangle$

shows $\langle b * x \rangle \ p \ \langle \lambda r. a \ r * x \rangle_t$

⟨proof⟩

lemma *add-true*:

assumes $\langle b \rangle \ p \ \langle a \rangle_t$

shows $\langle b * \text{true} \rangle \ p \ \langle a \rangle_t$

⟨proof⟩

definition *node-relator* **where** $\text{node-relator } x \ y \longleftrightarrow x \in y$

sep-auto behaves sub-optimal when having $(bf, bdd) \in \text{computed-pointer-relation}$ as assumption in our cases. Using *node-relator* instead fixes this behavior with a custom solver for *simp*.

lemma *node-relatorI*: $x \in y \implies \text{node-relator } x \ y$ ⟨proof⟩

lemma *node-relatorD*: $\text{node-relator } x \ y \implies x \in y$ ⟨proof⟩

⟨ML⟩

This is the general form one wants to work with: if a function on the bdd is called with a set of already existing and valid pointers, the arguments to the function have to be in that set. The result is that one more pointer is the set of existing and valid pointers.

thm *iteci-rule*[THEN *mp*] *mi.ite-impl-R ifex-ite-rel-bf*

lemma *iteci-rule*[sep-heap-rules]:

$\llbracket \text{node-relator } (ib, ic) \ rp; \text{node-relator } (tb, tc) \ rp; \text{node-relator } (eb, ec) \ rp \rrbracket \implies$

$\langle \text{bdd-relator } rp \ s \rangle$

iteci-lu ic tc ec s

$\langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-ite } ib \text{ } tb \text{ } eb,r) \text{ } rp) \text{ } s' \rangle$
 $\langle \text{proof} \rangle$

lemma *tci-rule*[*sep-heap-rules*]:

$\langle \text{bdd-relator } rp \text{ } s \rangle$
tci s
 $\langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-True},r) \text{ } rp) \text{ } s' \rangle$
 $\langle \text{proof} \rangle$

lemma *fci-rule*[*sep-heap-rules*]:

$\langle \text{bdd-relator } rp \text{ } s \rangle$
fci s
 $\langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-False},r) \text{ } rp) \text{ } s' \rangle$
 $\langle \text{proof} \rangle$

IFC/ifmi/ifci require that the variable order is ensured by the user. Instead of using ifci, a combination of litci and iteci has to be used.

lemma [*sep-heap-rules*]:

$\llbracket (tb, tc) \in rp; (eb, ec) \in rp \rrbracket \implies$
 $\langle \text{bdd-relator } rp \text{ } s \rangle$
ifci v tc ec s
 $\langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-if } v \text{ } tb \text{ } eb,r) \text{ } rp) \text{ } s' \rangle$

This probably doesn't hold.

$\langle \text{proof} \rangle$

lemma *notci-rule*[*sep-heap-rules*]:

assumes *node-relator* (*tb, tc*) *rp*
shows $\langle \text{bdd-relator } rp \text{ } s \rangle \text{notci } tc \text{ } s \langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-not } tb,r) \text{ } rp) \text{ } s' \rangle$
 $\langle \text{proof} \rangle$

lemma *cirules1*[*sep-heap-rules*]:

assumes *node-relator* (*tb, tc*) *rp* *node-relator* (*eb, ec*) *rp*
shows
 $\langle \text{bdd-relator } rp \text{ } s \rangle \text{andci } tc \text{ } ec \text{ } s \langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-and } tb \text{ } eb,r) \text{ } rp) \text{ } s' \rangle$
 $\langle \text{bdd-relator } rp \text{ } s \rangle \text{orci } tc \text{ } ec \text{ } s \langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-or } tb \text{ } eb,r) \text{ } rp) \text{ } s' \rangle$
 $\langle \text{bdd-relator } rp \text{ } s \rangle \text{biimpci } tc \text{ } ec \text{ } s \langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-biimp } tb \text{ } eb,r) \text{ } rp) \text{ } s' \rangle$
 $\langle \text{bdd-relator } rp \text{ } s \rangle \text{xorci } tc \text{ } ec \text{ } s \langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-xor } tb \text{ } eb,r) \text{ } rp) \text{ } s' \rangle$

$\langle \text{proof} \rangle$

lemma *cirules2*[*sep-heap-rules*]:

assumes *node-relator* (*tb, tc*) *rp* *node-relator* (*eb, ec*) *rp*
shows

```

    <bdd-relator rp s> nandci tc ec s < $\lambda(r,s')$ . bdd-relator (insert (bf-nand tb eb,r)
rp) s'>
    <bdd-relator rp s> norci tc ec s < $\lambda(r,s')$ . bdd-relator (insert (bf-nor tb eb,r)
rp) s'>
    <proof>

```

lemma *litci-rule*[*sep-heap-rules*]:

```

    <bdd-relator rp s> litci v s < $\lambda(r,s')$ . bdd-relator (insert (bf-lit v,r) rp) s'>
    <proof>

```

lemma *tautci-rule*[*sep-heap-rules*]:

```

    shows node-relator (tb, tc) rp  $\implies$  <bdd-relator rp s> tautci tc s < $\lambda r$ . bdd-relator
rp s *  $\uparrow(r \longleftrightarrow tb = \text{bf-True})$ >
    <proof>

```

lemma *emptyci-rule*[*sep-heap-rules*]:

```

    shows <emp> emptyci < $\lambda r$ . bdd-relator {} r>
    <proof>

```

lemmas [*simp*] = *bf-ite-def*

Efficient comparison of two nodes.

definition *eqci* *a b* \equiv *return (a = b)*

lemma *iteeq-rule*[*sep-heap-rules*]:

```

[[node-relator (xb, xc) rp; node-relator (yb, yc) rp]]  $\implies$ 
<bdd-relator rp s>
    eqci xc yc
< $\lambda r$ .  $\uparrow(r \longleftrightarrow xb = yb)$ >t
    <proof>

```

end

12 Tests and examples

theory *BDD-Examples*

imports *Level-Collapse*

begin

Just two simple examples:

```

lemma <emp> do {
    s  $\leftarrow$  emptyci;
    (t,s)  $\leftarrow$  tci s;
    tautci t s
} < $\lambda r$ .  $\uparrow(r = \text{True})$ >t
    <proof>

```



```

lemma <emp> do {
  s ← emptyci;
  (a,s) ← litci 0 s;
  (b,s) ← litci 1 s;
  (c,s) ← litci 2 s;
  (t1i,s) ← orci a b s;
  (t1,s) ← andci t1i c s;
  (t2i1,s) ← andci a c s;
  (t2i2,s) ← andci b c s;
  (t2,s) ← orci t2i1 t2i2 s;
  eqci t1 t2
} <↑>_t
<proof>

```

end

References

- [1] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45. ACM, 1991.
- [2] M. Giorgino and M. Strecker. Correctness of pointer manipulating algorithms illustrated by a verified BDD construction. In *FM 2012: Formal Methods*, pages 202–216. Springer, 2012.
- [3] T. Nipkow. Boolean Expression Checkers. *Archive of Formal Proofs*, June 2014. http://isa-afp.org/entries/Boolean_Expression_Checkers.shtml, Formal proof development.
- [4] V. Ortner and N. Schirmer. BDD Normalisation. *Archive of Formal Proofs*, Feb. 2008. <http://isa-afp.org/entries/BDD.shtml>, Formal proof development.