An implementation of ROBDDs for Isabelle/HOL

Julius Michaelis and Maximilian Haslbeck and Peter Lammich and Lars Hupel

March 17, 2025

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

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* $i\ t\ e \equiv (\lambda l.\ \mathit{if}\ i\ l\ \mathit{then}\ t\ l\ \mathit{else}\ e\ l)$

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* $\equiv (\lambda l.\ \mathit{True})$
abbreviation *bf-False* $\equiv (\lambda l.\ \mathit{False})$

A quick demonstration:

definition *bf-and* $a\ b \equiv$ *bf-ite* $a\ b$ *bf-False*
lemma $(\mathit{bf\text{-}and}\ a\ b)\ as \longleftrightarrow a\ as \wedge b\ as$ ⟨*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* $v\ t\ e = (\lambda l.\ \mathit{if}\ l\ v\ \mathit{then}\ t\ l\ \mathit{else}\ e\ l)$ ⟨*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* $\Longrightarrow$ *False bf-True = bf-False* $\Longrightarrow$ *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* ⇒ (*'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*)

fun *ifex-vars* :: (*'a* :: *linorder*) *ifex* ⇒ *'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* ≡ *set* (*ifex-vars a*)

fun *ifex-ordered* :: (*'a*::*linorder*) *ifex* ⇒ *bool* **where**
  ifex-ordered (*IF v t e*) = ((∀ *tv* ∈ (*ifex-var-set t* ∪ *ifex-var-set e*). *v* < *tv*)
  ∧ *ifex-ordered t* ∧ *ifex-ordered e*) |
  ifex-ordered Trueif = *True* |
  ifex-ordered Falseif = *True*

fun *ifex-minimal* :: (*'a*::*linorder*) *ifex* ⇒ *bool* **where**
  ifex-minimal (*IF v t e*) ⟷ *t* ≠ *e* ∧ *ifex-minimal t* ∧ *ifex-minimal e* |
  ifex-minimal Trueif = *True* |
  ifex-minimal Falseif = *True*

abbreviation *ro-ifex* **where** *ro-ifex t* ≡ *ifex-ordered t* ∧ *ifex-minimal t*

definition *bf-ifex-rel* **where**
  bf-ifex-rel = {(*a*,*b*). (∀ *ass*. *a ass* ⟷ *val-ifex b ass*) ∧ *ro-ifex b*}

lemma *ifex-var-noinfluence*: *x* ∉ *ifex-var-set b* ⟹ *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* ∉ *ifex-var-set t* **and** *v* ∉ *ifex-var-set e*
⟨*proof*⟩

lemma *roifex-set-var-subtree*:
assumes *ro-ifex b* **and** $b = IF\ v\ t\ e$
shows *val-ifex b* $(ass(v{:=}True)) =$ *val-ifex t ass*
val-ifex b $(ass(v{:=}False)) =$ *val-ifex e ass*
⟨*proof*⟩

lemma *roifex-Trueif-unique*: *ro-ifex b* $\Longrightarrow \forall$ *ass. val-ifex b ass* $\Longrightarrow b = Trueif$
⟨*proof*⟩

lemma *roifex-Falseif-unique*: *ro-ifex b* $\Longrightarrow \forall$ *ass.* $\neg$ *val-ifex b ass* $\Longrightarrow b = Falseif$
⟨*proof*⟩

lemma $(f,\ b) \in$ *bf-ifex-rel* $\Longrightarrow\ b = Trueif \longleftrightarrow f = (\lambda\text{-}.\ True)$
⟨*proof*⟩

lemma $(f,\ b) \in$ *bf-ifex-rel* $\Longrightarrow\ b = Falseif \longleftrightarrow f = (\lambda\text{-}.\ False)$
⟨*proof*⟩

lemma *ifex-ordered-not-part*: *ifex-ordered* $b \Longrightarrow b = IF\ v\ b1\ b2 \Longrightarrow w < v \Longrightarrow w \notin$ *ifex-var-set b*
⟨*proof*⟩

lemma *ro-ifex-unique*: *ro-ifex x* $\Longrightarrow$ *ro-ifex y* $\Longrightarrow (\bigwedge$*ass. val-ifex x ass* $=$ *val-ifex y ass*$) \Longrightarrow x = y$
⟨*proof*⟩

theorem *bf-ifex-rel-single*: *single-valued bf-ifex-rel single-valued* $(\textit{bf-ifex-rel}^{-1})$
⟨*proof*⟩

lemma *bf-ifex-eq*: $(af,\ at) \in$ *bf-ifex-rel* $\Longrightarrow (bf,\ bt) \in$ *bf-ifex-rel* $\Longrightarrow (af = bf) \longleftrightarrow (at = bt)$
⟨*proof*⟩

lemma *nonempty-if-var-set*: *ifex-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-var-set (restrict b var val)*
⟨*proof*⟩

lemma *restrict-assignment*: *val-ifex b* $(ass(var := val)) \longleftrightarrow$ *val-ifex (restrict b var val) ass*
⟨*proof*⟩

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

lemma *restrict-ifex-ordered-invar*: *ifex-ordered b* $\Longrightarrow$ *ifex-ordered* (*restrict b var val*)
⟨*proof*⟩

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

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

fun *ifex-top-var* :: *'a ifex* $\Rightarrow$ *'a option* **where**
ifex-top-var (*IF v t e*) = *Some v* |
ifex-top-var - = *None*

fun *restrict-top* :: (*'a* :: *linorder*) *ifex* $\Rightarrow$ *'a* $\Rightarrow$ *bool* $\Rightarrow$ *'a ifex* **where**
restrict-top (*IF v t e*) *var val* = (*if v* = *var then* (*if val then t else e*) *else* (*IF v t e*)) |
restrict-top i - - = *i*

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

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

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

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

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

lemma *restrict-top-ifex-ordered-invar*: *ifex-ordered b* $\Longrightarrow$ *ifex-ordered* (*restrict-top b var 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* ⇒ (*min u v*) | *None* ⇒ *v*) |
lowest-tops (-#*r*) = *lowest-tops r*

lemma *lowest-tops-NoneD*: *lowest-tops k* = *None* ⟹ (¬(∃ *v t e*. ((*IF v t e*) ∈ *set k*)))
⟨*proof*⟩

lemma *lowest-tops-in*: *lowest-tops k* = *Some l* ⟹ *l* ∈ *set* (*concat* (*map ifex-vars k*))
⟨*proof*⟩

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

function *ifex-ite* :: ′*a ifex* ⇒ ′*a ifex* ⇒ ′*a ifex* ⇒ (′*a* :: *linorder*) *ifex* **where**
ifex-ite i t e = (*case lowest-tops* [*i*, *t*, *e*] *of Some x* ⇒
(*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* ⇒ (*case i of Trueif* ⇒ *t* | *Falseif* ⇒ *e*))
⟨*proof*⟩

lemma *restrict-size-le*: *size* (*restrict-top k var val*) ≤ *size k*
⟨*proof*⟩

lemma *restrict-size-less*: *ifex-top-var k* = *Some var* ⟹ *size* (*restrict-top k var val*) < *size k*
⟨*proof*⟩

lemma *lowest-tops-cases*:
lowest-tops [*i*, *t*, *e*] = *Some var* ⟹ *ifex-top-var i* = *Some var* ∨ *ifex-top-var t* = *Some var* ∨ *ifex-top-var e* = *Some var*
⟨*proof*⟩

lemma *lowest-tops-lowest*: *lowest-tops es* = *Some a* ⟹ *e* ∈ *set es* ⟹ *ifex-ordered e* ⟹ *v* ∈ *ifex-var-set e* ⟹ *a* ≤ *v*
⟨*proof*⟩

lemma *termlemma2*: *lowest-tops* [*i*, *t*, *e*] = *Some xa* ⟹
(*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* ⟹
(*case* (*restrict-top i xa val*, *restrict-top t xa val*, *restrict-top e xa val*) *of*
(*i*, *t*, *e*) ⇒ *size i* + *size t* + *size e*) <

$(case\ (i,\ t,\ e)\ of\ (i,\ t,\ e) \Rightarrow size\ i + size\ t + size\ e)$
⟨*proof*⟩

termination *ifex-ite*
⟨*proof*⟩

definition *const x - = x*
declare *const-def*[*simp*]
lemma *rel-true-false*: $(a,\ Trueif) \in bf\text{-}ifex\text{-}rel \Longrightarrow a = const\ True\ (a,\ Falseif) \in bf\text{-}ifex\text{-}rel \Longrightarrow a = const\ False$
⟨*proof*⟩

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

lemma *ifex-ordered-implied*: $(a,\ b) \in bf\text{-}ifex\text{-}rel \Longrightarrow ifex\text{-}ordered\ b$ ⟨*proof*⟩
lemma *ifex-minimal-implied*: $(a,\ b) \in bf\text{-}ifex\text{-}rel \Longrightarrow ifex\text{-}minimal\ b$ ⟨*proof*⟩

lemma *ifex-ite-induct2*[*case-names Trueif Falseif IF*]:
$(\bigwedge i\ t\ e.\ lowest\text{-}tops\ [i,\ t,\ e] = None \Longrightarrow i = Trueif \Longrightarrow sentence\ i\ t\ e) \Longrightarrow$
$(\bigwedge i\ t\ e.\ lowest\text{-}tops\ [i,\ t,\ e] = None \Longrightarrow i = Falseif \Longrightarrow sentence\ i\ t\ e) \Longrightarrow$
$(\bigwedge i\ t\ e\ a.\ sentence\ (restrict\text{-}top\ i\ a\ True)\ (restrict\text{-}top\ t\ a\ True)\ (restrict\text{-}top\ e\ a\ True) \Longrightarrow$
$sentence\ (restrict\text{-}top\ i\ a\ False)\ (restrict\text{-}top\ t\ a\ False)\ (restrict\text{-}top\ e\ a\ False) \Longrightarrow$
$lowest\text{-}tops\ [i,\ t,\ e] = Some\ a \Longrightarrow sentence\ i\ t\ e) \Longrightarrow sentence\ i\ t\ e$
⟨*proof*⟩

lemma *ifex-ite-induct*[*case-names Trueif Falseif IF*]:
$(\bigwedge i\ t\ e.\ lowest\text{-}tops\ [i,\ t,\ e] = None \Longrightarrow i = Trueif \Longrightarrow P\ i\ t\ e) \Longrightarrow$
$(\bigwedge i\ t\ e.\ lowest\text{-}tops\ [i,\ t,\ e] = None \Longrightarrow i = Falseif \Longrightarrow P\ i\ t\ e) \Longrightarrow$
$(\bigwedge i\ t\ e\ a.\ (\bigwedge val.\ P\ (restrict\text{-}top\ i\ a\ val)\ (restrict\text{-}top\ t\ a\ val)\ (restrict\text{-}top\ e\ a\ val)) \Longrightarrow$
$lowest\text{-}tops\ [i,\ t,\ e] = Some\ a \Longrightarrow P\ i\ t\ e) \Longrightarrow P\ i\ t\ e$
⟨*proof*⟩

lemma *restrict-top-subset*: $x \in ifex\text{-}var\text{-}set\ (restrict\text{-}top\ i\ vr\ vl) \Longrightarrow x \in ifex\text{-}var\text{-}set\ i$
⟨*proof*⟩

lemma *ifex-vars-subset*: $x \in ifex\text{-}var\text{-}set\ (ifex\text{-}ite\ i\ t\ e) \Longrightarrow (x \in ifex\text{-}var\text{-}set\ i) \lor (x \in ifex\text{-}var\text{-}set\ t) \lor (x \in ifex\text{-}var\text{-}set\ e)$
⟨*proof*⟩

lemma *three-ins*: $i \in set\ [i,\ t,\ e]\ t \in set\ [i,\ t,\ e]\ e \in set\ [i,\ t,\ e]$ ⟨*proof*⟩

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

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

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

lemma *order-ifex-ite-invar*: $ifex\text{-}ordered\ i \Longrightarrow ifex\text{-}ordered\ t \Longrightarrow ifex\text{-}ordered\ e \Longrightarrow ifex\text{-}ordered\ (ifex\text{-}ite\ i\ t\ e)$
⟨*proof*⟩

lemma *ifc-split*: $P\ (IFC\ v\ t\ e) \longleftrightarrow ((t = e) \longrightarrow P\ t) \wedge (t \neq e \longrightarrow P\ (IF\ v\ t\ e))$
⟨*proof*⟩

lemma *restrict-top-ifex-minimal-invar*: $ifex\text{-}minimal\ i \Longrightarrow ifex\text{-}minimal\ (restrict\text{-}top\ i\ a\ val)$
⟨*proof*⟩

lemma *minimal-ifex-ite-invar*: $ifex\text{-}minimal\ i \Longrightarrow ifex\text{-}minimal\ t \Longrightarrow ifex\text{-}minimal\ e \Longrightarrow ifex\text{-}minimal\ (ifex\text{-}ite\ i\ t\ e)$
⟨*proof*⟩

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

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

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

definition *param-opt* **where** $param\text{-}opt\ i\ t\ e =$
$(if\ i = Trueif\ then\ Some\ t\ else$

if i = Falseif then Some e else
if t = Trueif ∧ *e = Falseif then Some i else*
if t = e then Some t else
if e = Trueif ∧ *i = t then Some Trueif else*
if t = Falseif ∧ *i = e then Some Falseif else*
None)

lemma *param-opt-ifex-ite-eq*: *ro-ifex i* ⟹ *ro-ifex t* ⟹ *ro-ifex e* ⟹
param-opt i t e = Some r ⟹ *r = ifex-ite i t e*
⟨*proof*⟩

function *ifex-ite-opt* :: *′a ifex* ⇒ *′a ifex* ⇒ *′a ifex* ⇒ (*′a* :: *linorder*) *ifex* **where**
ifex-ite-opt i t e = (*case param-opt i t e of Some b* ⇒ *b* | *None* ⇒
(*case lowest-tops* [*i*, *t*, *e*] *of Some x* ⇒
(*IFC x* (*ifex-ite-opt* (*restrict-top i x True*) (*restrict-top t x True*)
(*restrict-top e x True*))
(*ifex-ite-opt* (*restrict-top i x False*) (*restrict-top t x False*)
(*restrict-top e x False*)))
| *None* ⇒ (*case i of Trueif* ⇒ *t* | *Falseif* ⇒ *e*)))
⟨*proof*⟩

termination *ifex-ite-opt*
⟨*proof*⟩

lemma *ifex-ite-opt-eq*:
ro-ifex i ⟹ *ro-ifex t* ⟹ *ro-ifex e* ⟹ *ifex-ite-opt i t e = ifex-ite i t e*
⟨*proof*⟩

lemma *ro-ifexI*: (*a*,*b*) ∈ *bf-ifex-rel* ⟹ *ro-ifex b* ⟨*proof*⟩

theorem *ifex-ite-opt-rel-bf*:
(*fi*,*i*) ∈ *bf-ifex-rel* ⟹
(*ft*,*t*) ∈ *bf-ifex-rel* ⟹
(*fe*,*e*) ∈ *bf-ifex-rel* ⟹
((*bf-ite fi ft fe*), (*ifex-ite-opt i t e*)) ∈ *bf-ifex-rel*
⟨*proof*⟩

lemma *restrict-top-bf-ifex-rel*:
(*f*, *i*) ∈ *bf-ifex-rel* ⟹ ∃ *f′*. (*f′*, *restrict-top i var val*) ∈ *bf-ifex-rel*
⟨*proof*⟩

lemma *param-opt-lowest-tops-lem*: *param-opt i t e = None* ⟹ ∃ *y*. *lowest-tops*
[*i*,*t*,*e*] = *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 ⇒ *Some* (*a*(*v*:=*False*)) |
    None ⇒ (*case ifex-sat t of*
      Some a ⇒ *Some* (*a*(*v*:=*True*)) |
      None ⇒ *None*))

lemma *ifex-sat-untouched-False*: $v \notin$ *ifex-var-set i* $\Longrightarrow$ *ifex-sat i* = *Some a* $\Longrightarrow$ *a v* = *False*
⟨*proof*⟩

lemma *ifex-upd-other*: $v \notin$ *ifex-var-set i* $\Longrightarrow$ *val-ifex i* (*a*(*v*:=*any*)) = *val-ifex i a*
⟨*proof*⟩

fun *ifex-no-twice* **where**
ifex-no-twice (*IF v t e*) = (
  $v \notin$ (*ifex-var-set t* ∪ *ifex-var-set e*) ∧
  ifex-no-twice t ∧ *ifex-no-twice e*) |
ifex-no-twice - = *True*
lemma *ordered-ifex-no-twiceI*: *ifex-ordered i* $\Longrightarrow$ *ifex-no-twice i*
  ⟨*proof*⟩

lemma *ifex-sat-NoneD*: *ifex-sat i* = *None* $\Longrightarrow$ *val-ifex i ass* = *False*
  ⟨*proof*⟩
lemma *ifex-sat-SomeD*: *ifex-no-twice i* $\Longrightarrow$ *ifex-sat i* = *Some ass* $\Longrightarrow$ *val-ifex i ass* = *True*
⟨*proof*⟩
lemma *ifex-sat-NoneI*: *ifex-no-twice i* $\Longrightarrow$ ($\bigwedge$*ass. val-ifex i ass* = *False*) $\Longrightarrow$ *ifex-sat i* = *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 ⇒ *Some* ((*v*,*False*)#*a*) |
    None ⇒ (*case ifex-sat-list t of*
      Some a ⇒ *Some* ((*v*,*True*)#*a*) |
      None ⇒ *None*))

definition *update-assignment-alt u as* = (λ*v. case map-of u v of None* ⇒ *as v* | *Some n* ⇒ *n*)

fun *update-assignment* **where**
update-assignment $((v,u)\#us)$ *as* = (*update-assignment us as*)$(v:=u)$ |
update-assignment [] *as* = *as*

lemma *update-assignment-notin*: $a \notin$ *fst* ` *set us* $\Longrightarrow$ *update-assignment us as a* = *as a*
⟨*proof*⟩

lemma *update-assignment-alt*: *update-assignment u as* = *update-assignment-alt u as*
⟨*proof*⟩

lemma *update-assignment*: *distinct* (*map fst* $((v,u)\#us)$) $\Longrightarrow$ *update-assignment* $((v,u)\#us)$ *as* = *update-assignment us* (*as*$(v:=u)$)
⟨*proof*⟩

lemma *ass-upd-same*: *update-assignment* $((v, u) \# a)$ *ass* $v = u$ ⟨*proof*⟩

lemma *ifex-sat-list-subset*: *ifex-sat-list t* = *Some u* $\Longrightarrow$ *fst* ` *set u* $\subseteq$ *ifex-var-set t*
⟨*proof*⟩

lemma *sat-list-distinct*: *ifex-no-twice t* $\Longrightarrow$ *ifex-sat-list t* = *Some u* $\Longrightarrow$ *distinct* (*map fst u*)
⟨*proof*⟩

lemma *ifex-sat-list-NoneD*: *ifex-sat-list i* = *None* $\Longrightarrow$ *val-ifex i ass* = *False*
⟨*proof*⟩
lemma *ifex-sat-list-SomeD*: *ifex-no-twice i* $\Longrightarrow$ *ifex-sat-list i* = *Some u* $\Longrightarrow$ *ass* = *update-assignment u ass'* $\Longrightarrow$ *val-ifex i ass* = *True*
⟨*proof*⟩

fun *sat-list-to-bdt* **where**
sat-list-to-bdt [] = *Trueif* |
sat-list-to-bdt $((v,u)\#us)$ = (*if u then IF v* (*sat-list-to-bdt us*) *Falseif else IF v Falseif* (*sat-list-to-bdt us*))

lemma *ifex-sat-list i* = *Some u* $\Longrightarrow$ *val-ifex* (*sat-list-to-bdt u*) *as* $\Longrightarrow$ *val-ifex i as*
⟨*proof*⟩

lemma *bf-ifex-rel-consts*[*simp*,*intro*!]:
(*bf-True*, *Trueif*) $\in$ *bf-ifex-rel*
(*bf-False*, *Falseif*) $\in$ *bf-ifex-rel*
⟨*proof*⟩
lemma *bf-ifex-rel-lit*[*simp*,*intro*!]:
(*bf-lit v*, *IFC v Trueif Falseif*) $\in$ *bf-ifex-rel*
⟨*proof*⟩

lemma *bf-ifex-rel-consts-ensured*[*simp*]:

$(bf\text{-}True,x) \in bf\text{-}ifex\text{-}rel \longleftrightarrow (x = Trueif)$
$(bf\text{-}False,x) \in bf\text{-}ifex\text{-}rel \longleftrightarrow (x = Falseif)$
⟨*proof*⟩

lemma *bf-ifex-rel-consts-ensured-rev*[*simp*]:
$(x,Trueif) \in bf\text{-}ifex\text{-}rel \longleftrightarrow (x = bf\text{-}True)$
$(x,Falseif) \in bf\text{-}ifex\text{-}rel \longleftrightarrow (x = bf\text{-}False)$
⟨*proof*⟩

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 HOL−Library.Monad-Syntax*
begin

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

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

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*) ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
ospec None - = *False*
| *ospec* (*Some v*) *Q* = *Q v*

named-theorems *ospec-rules*

lemma *oreturn-rule*[*ospec-rules*]: $[\![\ P\ r\ ]\!] \Longrightarrow ospec\ (Some\ r)\ P$ ⟨*proof*⟩

lemma *obind-rule*[*ospec-rules*]: $[\![\ ospec\ m\ Q;\ \bigwedge r.\ Q\ r \Longrightarrow ospec\ (f\ r)\ P\ ]\!] \Longrightarrow ospec$ $(m \ggg f)\ P$
⟨*proof*⟩

lemma *ospec-alt*: $ospec\ m\ P = (case\ m\ of\ None \Rightarrow False \mid Some\ x \Rightarrow P\ x)$
⟨*proof*⟩

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

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

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

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

end

5 Abstract ITE Implementation

theory *Abstract-Impl*
imports *BDT*
    Automatic-Refinement.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 R* **for** $R :: 's \Rightarrow ('ni \times ('a :: linorder)\ ifex)\ set$ +
  fixes $Timpl :: 's \rightharpoonup ('ni \times 's)$
  fixes $Fimpl :: 's \rightharpoonup ('ni \times 's)$
  fixes $IFimpl :: 'a \Rightarrow 'ni \Rightarrow 'ni \Rightarrow 's \rightharpoonup ('ni \times 's)$
  fixes $DESTRimpl :: 'ni \Rightarrow 's \rightharpoonup ('a,\ 'ni)\ IFEXD$

  assumes *Timpl-rule*: $I\ s \Longrightarrow ospec\ (Timpl\ s)\ (\lambda(ni,\ s').\ (ni,\ Trueif) \in R\ s' \wedge I\ s' \wedge les\ s\ s')$
  assumes *Fimpl-rule*: $I\ s \Longrightarrow ospec\ (Fimpl\ s)\ (\lambda(ni,\ s').\ (ni,\ Falseif) \in R\ s' \wedge I\ s' \wedge les\ s\ s')$
  assumes *IFimpl-rule*: $[\![I\ s;\ (ni1,n1) \in R\ s;(ni2,n2) \in R\ s]\!]$
    $\Longrightarrow ospec\ (IFimpl\ v\ ni1\ ni2\ s)\ (\lambda(ni,\ s').\ (ni,\ IFC\ v\ n1\ n2) \in R\ s' \wedge I\ s' \wedge les\ s\ s')$

  assumes *DESTRimpl-rule1*: $I\ s \Longrightarrow (ni,\ Trueif) \in R\ s \Longrightarrow ospec\ (DESTRimpl\ ni\ s)\ (\lambda r.\ r = TD)$
  assumes *DESTRimpl-rule2*: $I\ s \Longrightarrow (ni,\ Falseif) \in R\ s \Longrightarrow ospec\ (DESTRimpl$

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

lemma *les-refl*[*simp*,*intro*!]:*les s s* ⟨*proof*⟩
lemma *les-trans*[*trans*]:*les s1 s2* ⟹ *les s2 s3* ⟹ *les s1 s3* ⟨*proof*⟩
lemmas *DESTRimpl-rules* = *DESTRimpl-rule1 DESTRimpl-rule2 DESTRimpl-rule3*

lemma *DESTRimpl-rule-useless*:
$I\ s \Longrightarrow (ni,\ n) \in R\ s \Longrightarrow ospec\ (DESTRimpl\ ni\ s)\ (\lambda r.\ (case\ r\ of$
$TD \Rightarrow (ni,\ Trueif) \in R\ s\ |$
$FD \Rightarrow (ni,\ Falseif) \in R\ s\ |$
$IFD\ v\ nt\ ne \Rightarrow (\exists\, t\ e.\ n = IF\ v\ t\ e \wedge (ni,\ IF\ v\ t\ e) \in R\ s)))$
⟨*proof*⟩
lemma *DESTRimpl-rule*:
$I\ s \Longrightarrow (ni,\ n) \in R\ s \Longrightarrow ospec\ (DESTRimpl\ ni\ s)\ (\lambda r.\ (case\ n\ of$
$Trueif \Rightarrow r = TD\ |$
$Falseif \Rightarrow r = FD\ |$
$IF\ v\ t\ e \Rightarrow (\exists\, tn\ en.\ r = IFD\ v\ tn\ en \wedge (tn,t) \in R\ s \wedge (en,e) \in R\ s)))$
⟨*proof*⟩

definition *case-ifexi fti ffi fii ni s* ≡ *do* {
dest ← *DESTRimpl ni s*;
case dest of
TD ⇒ *fti s*
| *FD* ⇒ *ffi s*
| *IFD v ti ei* ⇒ *fii v ti ei s*}

lemma *case-ifexi-rule*:
assumes *INV*: *I s*
assumes *NI*: $(ni,n) \in R\ s$
assumes *FTI*: $[\![\ n = Trueif\ ]\!] \Longrightarrow ospec\ (fti\ s)\ (\lambda(r,s').\ (r,ft) \in Q\ s \wedge I'\ s')$
assumes *FFI*: $[\![\ n = Falseif\ ]\!] \Longrightarrow ospec\ (ffi\ s)\ (\lambda(r,s').\ (r,ff) \in Q\ s \wedge I'\ s')$
assumes *FII*: $\bigwedge ti\ ei\ v\ t\ e.\ [\![\ n = IF\ v\ t\ e;\ (ti,t) \in R\ s;\ (ei,e) \in R\ s\ ]\!] \Longrightarrow ospec\ (fii$
$v\ ti\ ei\ s)\ (\lambda(r,s').\ (r,fi\ v\ t\ e) \in Q\ s \wedge I'\ s')$
shows $ospec\ (case\text{-}ifexi\ fti\ ffi\ fii\ ni\ s)\ (\lambda(r,s').\ (r,case\text{-}ifex\ ft\ ff\ fi\ n) \in Q\ s \wedge I'$
$s')$
⟨*proof*⟩

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

primrec *lowest-tops-impl* **where**
lowest-tops-impl [] *s* = *Some* (*None*,*s*) |
lowest-tops-impl (*e*#*es*) *s* =
case-ifexi
($\lambda s.$ *lowest-tops-impl es s*)

($\lambda s.$ *lowest-tops-impl es s*)
($\lambda v\ t\ e\ s.$ *do* {
(*rec*,*s*) $\leftarrow$ *lowest-tops-impl es s*;
  (*case rec of*
    Some u $\Rightarrow$ *Some* ((*Some* (*min u v*)), *s*) |
    None $\Rightarrow$ *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
    ($\lambda v\ t\ e.$ (*case rec of*
      Some u $\Rightarrow$ (*Some* (*min u v*)) |
      None $\Rightarrow$ (*Some v*))
) *e*
)

lemma *lowest-tops-alt*: *lowest-tops l* = *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 = *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*)
    ($\lambda v\ te\ ee.$ *return* (*if v* = *vr then* (*if vl then te else ee*) *else e*))
    e s

lemma *restrict-top-alt*: *restrict-top n var val* = (*case n of*
  (*IF v t e*) $\Rightarrow$ (*if v* = *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* $\Longrightarrow$ *ospec* (*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*) (λ(*r*, *s'*). (*r*, *ifex-ite i t e*) ∈ *R s'* ∧ *I s'* ∧ *les s s'*)
⟨*proof*⟩

lemma *case-ifexi-mono*[*partial-function-mono*]:
  assumes [*partial-function-mono*]:
    mono-option (λ*F*. *fti F s*)
    mono-option (λ*F*. *ffi F s*)
    ⋀*x31 x32 x33*. *mono-option* (λ*F*. *fii F x31 x32 x33 s*)
  shows *mono-option* (λ*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*
  (λ*s*. *Some* (*True*,*s*))
  (λ*s*. *Some* (*False*,*s*))
  (λ*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*) (λ(*r*,*s'*). *r* = (*val-ifex n ass*) ∧ *s'*=*s*)
  ⟨*proof*⟩

end

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

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

lemma *map-invar-impl-les*: $map\text{-}invar\text{-}impl\ m\ s \Longrightarrow les\ s\ s' \Longrightarrow map\text{-}invar\text{-}impl\ m\ s'$
 ⟨*proof*⟩

lemma *map-invar-impl-update*: $map\text{-}invar\text{-}impl\ m\ s \Longrightarrow$
 $(ii,i) \in R\ s \Longrightarrow (ti,t) \in R\ s \Longrightarrow (ei,e) \in R\ s \Longrightarrow$
 $(ri,\ ifex\text{-}ite\text{-}opt\ i\ t\ e) \in R\ s \Longrightarrow map\text{-}invar\text{-}impl\ (m((ii,ti,ei) \mapsto ri))\ s$
⟨*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*: $I\ s \Longrightarrow (ni,\ i) \in R\ s \Longrightarrow (ni',\ i) \in R\ s \Longrightarrow cmp\ ni\ ni'$
 assumes *cmp-rule2*: $I\ s \Longrightarrow cmp\ ni\ ni' \Longrightarrow (ni,\ i) \in R\ s \Longrightarrow (ni',\ i') \in R\ s \Longrightarrow i = i'$
 assumes *map-invar-rule1*: $I\ s \Longrightarrow map\text{-}invar\text{-}impl\ (M\ s)\ s$
 assumes *map-invar-rule2*: $I\ s \Longrightarrow (ii,it) \in R\ s \Longrightarrow (ti,tt) \in R\ s \Longrightarrow (ei,et) \in R\ s \Longrightarrow$
 $(ri,\ ifex\text{-}ite\text{-}opt\ it\ tt\ et) \in R\ s \Longrightarrow U\ s\ (ii,ti,ei)\ ri = s' \Longrightarrow I\ s'$
 assumes *map-invar-rule3*: $I\ s \Longrightarrow R\ (U\ s\ (ii,\ ti,\ ei)\ ri) = R\ s$
begin

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

lemma *DESTRimpl-Some*: $I\ s \Longrightarrow (ni,\ i) \in R\ s \Longrightarrow ospec\ (DESTRimpl\ ni\ s)\ (\lambda r.\ True)$
 ⟨*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 $\wedge$ *ei = FD then Some i else*
if cmp t e then Some t else
if ei = TD $\wedge$ *cmp i t then Some tn else*
if ti = FD $\wedge$ *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*) $\in$ *R s* (*ti*,*t*) $\in$ *R s* (*ei*,*e*) $\in$ *R s*
shows *ospec* (*param-opt-impl ii ti ei s*)
(λ(*r*,*s'*). *I s'* $\wedge$ *les s s'*)
⟨*proof*⟩

lemma *param-opt-impl-R*:
assumes *I s* (*ii*,*i*) $\in$ *R s* (*ti*,*t*) $\in$ *R s* (*ei*,*e*) $\in$ *R s*
shows *ospec* (*param-opt-impl ii ti ei s*)
(λ(*r*,*s'*). *case r of None* $\Rightarrow$ *param-opt i t e = None*
| *Some r* $\Rightarrow$ ($\exists$ *r'*. *param-opt i t e = Some r'* $\wedge$ (*r*, *r'*)
$\in$ *R s'*))
⟨*proof*⟩

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

lemma *ospec-and*: *ospec f P* $\Longrightarrow$ *ospec f Q* $\Longrightarrow$ *ospec f* (λ*x*. *P x* $\wedge$ *Q x*)
⟨*proof*⟩

lemma *ite-impl-opt-R*:
I s
$\Longrightarrow$ *in-rel* (*R s*) *ii i* $\Longrightarrow$ *in-rel* (*R s*) *ti t* $\Longrightarrow$ *in-rel* (*R s*) *ei e*

$\Longrightarrow$ *ospec* (*ite-impl-opt ii ti ei s*) (λ(*r*, *s'*). (*r*, *ifex-ite-opt i t e*) $\in$ *R s'* $\wedge$ *I s'* $\wedge$ *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*
$\Longrightarrow$ (*ii*,*i*) $\in$ *R s* $\Longrightarrow$ (*ti*,*t*) $\in$ *R s* $\Longrightarrow$ (*ei*,*e*) $\in$ *R s*
$\Longrightarrow$ *ospec* (*ite-impl-lu ii ti ei s*)
(λ(*r*, *s'*). (*r*, *ifex-ite-opt i t e*) $\in$ *R s'* $\wedge$ *I s'* $\wedge$ *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 :: $'a \Rightarrow$ *nat option*

definition *pointermap-sane* $m \equiv (distinct\ (entries\ m) \wedge$
$(\forall n \in \{..<length\ (entries\ m)\}.\ getentry\ m\ (entries\ m\ !\ n) = Some\ n) \wedge$
$(\forall p\ i.\ getentry\ m\ p = Some\ i \longrightarrow entries\ m\ !\ i = p \wedge i < length\ (entries\ m)))$

definition *empty-pointermap* $\equiv (\!|entries = [],\ getentry = \lambda p.\ None\ |\!)$
lemma *pointermap-empty-sane*[*simp*, *intro*!]: *pointermap-sane empty-pointermap*
⟨*proof*⟩

definition *pointermap-insert* $a\ m \equiv (\!|entries = (entries\ m)@[a],\ getentry = (getentry\ m)(a \mapsto length\ (entries\ m))|\!)$

definition *pm-pth* $m\ p \equiv entries\ m\ !\ p$

definition *pointermap-p-valid* $p\ m \equiv p < length\ (entries\ m)$

definition *pointermap-getmk* $a\ m \equiv (case\ getentry\ m\ a\ of\ Some\ p \Rightarrow (p,m)\ |\ None \Rightarrow let\ u = pointermap\text{-}insert\ a\ m\ in\ (the\ (getentry\ u\ a),\ u))$

lemma *pointermap-sane-appendD*: $pointermap\text{-}sane\ s \Longrightarrow m \notin set\ (entries\ s) \Longrightarrow pointermap\text{-}sane\ (pointermap\text{-}insert\ m\ s)$
⟨*proof*⟩

lemma *luentries-noneD*: $getentry\ s\ a = None \Longrightarrow pointermap\text{-}sane\ s \Longrightarrow a \notin set\ (entries\ s)$
⟨*proof*⟩

lemma *pm-pth-append*: $pointermap\text{-}p\text{-}valid\ p\ m \Longrightarrow pm\text{-}pth\ (pointermap\text{-}insert\ a\ m)\ p = pm\text{-}pth\ m\ p$
⟨*proof*⟩

lemma *pointermap-insert-in*: $u = (pointermap\text{-}insert\ a\ m) \Longrightarrow pm\text{-}pth\ u\ (the\ (getentry\ u\ a)) = a$
⟨*proof*⟩

lemma *pointermap-insert-p-validI*: $pointermap\text{-}p\text{-}valid\ p\ m \Longrightarrow pointermap\text{-}p\text{-}valid\ p\ (pointermap\text{-}insert\ a\ m)$
⟨*proof*⟩

thm *nth-eq-iff-index-eq*
lemma *pth-eq-iff-index-eq*: $pointermap\text{-}sane\ m \Longrightarrow pointermap\text{-}p\text{-}valid\ p1\ m \Longrightarrow pointermap\text{-}p\text{-}valid\ p2\ m \Longrightarrow (pm\text{-}pth\ m\ p1 = pm\text{-}pth\ m\ p2) \longleftrightarrow (p1 = p2)$

$\langle proof \rangle$

lemma *pointermap-p-valid-updateI*: *pointermap-sane m* $\Longrightarrow$ *getentry m a* $=$ *None* $\Longrightarrow$ *u* $=$ *pointermap-insert a m* $\Longrightarrow$ *p* $=$ *the* (*getentry u a*) $\Longrightarrow$ *pointermap-p-valid p u*
$\langle proof \rangle$

lemma *pointermap-get-validI*: *pointermap-sane m* $\Longrightarrow$ *getentry m a* $=$ *Some p* $\Longrightarrow$ *pointermap-p-valid p m*
$\langle proof \rangle$

lemma *pointermap-sane-getmkD*:
  assumes *sn*: *pointermap-sane m*
  assumes *res*: *pointermap-getmk a m* $= (p,u)$
  shows *pointermap-sane u* $\wedge$ *pointermap-p-valid p u*
$\langle proof \rangle$

lemma *pointermap-update-pthI*:
  assumes *sn*: *pointermap-sane m*
  assumes *res*: *pointermap-getmk a m* $= (p,u)$
  shows *pm-pth u p* $=$ *a*
$\langle proof \rangle$

lemma *pointermap-p-valid-inv*:
  assumes *pointermap-p-valid p m*
  assumes *pointermap-getmk a m* $= (x,u)$
  shows *pointermap-p-valid p u*
$\langle proof \rangle$

lemma *pointermap-p-pth-inv*:
  assumes *pv*: *pointermap-p-valid p m*
  assumes *u*: *pointermap-getmk a m* $= (x,u)$
  shows *pm-pth u p* $=$ *pm-pth m p*
$\langle proof \rangle$

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

$\langle proof \rangle$

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 relects 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* ≡ (|*dpm* = *empty-pointermap*, *dcl* = *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* ≡ {(*a*,*b*)|*a b*. *Rmi-g a b s*}

interpretation *mi-pre*: *bdd-impl-cmp-pre Rmi* ⟨*proof*⟩

definition *bdd-node-valid bdd n* ≡ *n* ∈ *Domain* (*Rmi bdd*)
lemma [*simp*]:
  bdd-node-valid bdd 0
  bdd-node-valid bdd (*Suc 0*)
  ⟨*proof*⟩

definition *ifexd-valid bdd e* ≡ (*case e of IFD - t e* ⇒ *bdd-node-valid bdd t* ∧ *bdd-node-valid bdd e* | *-* ⇒ *True*)

definition *bdd-sane bdd* ≡ *pointermap-sane* (*dpm bdd*) ∧ *mi-pre.map-invar-impl* (*dcl bdd*) *bdd*

lemma [*simp*,*intro*!]: *bdd-sane emptymi*
⟨*proof*⟩

lemma *prod-split3*: *P* (*case p of* (*x*, *xa*, *xaa*) ⇒ *f x xa xaa*) = (∀ *x1 x2 x3*. *p* = (*x1*, *x2*, *x3*) ⟶ *P* (*f x1 x2 x3*))
⟨*proof*⟩

lemma *IfI*: (*c* ⟹ *P x*) ⟹ (¬*c* ⟹ *P y*) ⟹ *P* (*if c then x else y*) ⟨*proof*⟩
lemma *fstsndI*: *x* = (*a*,*b*) ⟹ *fst x* = *a* ∧ *snd x* = *b* ⟨*proof*⟩
thm *nat.split*
lemma *Rmi-g-2-split*: *P* (*Rmi-g n x m*) =
((*x* = *Falseif* ⟶ *P* (*Rmi-g n x m*)) ∧
(*x* = *Trueif* ⟶ *P* (*Rmi-g n x m*)) ∧
(∀ *vs ts es*. *x* = *IF vs ts es* ⟶ *P* (*Rmi-g n x m*)))
⟨*proof*⟩

lemma *rmigeq*: *Rmi-g ni1 n1 s* ⟹ *Rmi-g ni2 n2 s* ⟹ *ni1* = *ni2* ⟹ *n1* = *n2*
⟨*proof*⟩

lemma *rmigneq*: *bdd-sane s* ⟹ *Rmi-g ni1 n1 s* ⟹ *Rmi-g ni2 n2 s* ⟹ *ni1* ≠ *ni2* ⟹ *n1* ≠ *n2*
⟨*proof*⟩

lemma *ifmi-les-hlp*: *pointermap-sane* (*dpm s*) ⟹ *pointermap-getmk* (*v*, *ni1*, *ni2*) (*dpm s*) = (*x1*, *dpm s'*) ⟹ *Rmi-g nia n s* ⟹ *Rmi-g nia n s'*
⟨*proof*⟩
lemma *ifmi-les*:
assumes *bdd-sane s*
assumes *ifmi v ni1 ni2 s* = (*ni*, *s'*)
shows *mi-pre.les s s'*
⟨*proof*⟩

lemma *ifmi-notouch-dcl*: *ifmi v ni1 ni2 s* = (*ni*, *s'*) ⟹ *dcl s'* = *dcl s*
⟨*proof*⟩

lemma *ifmi-saneI*: *bdd-sane s* ⟹ *ifmi v ni1 ni2 s* = (*ni*, *s'*) ⟹ *bdd-sane s'*
⟨*proof*⟩

lemma *rmigif*: *Rmi-g ni* (*IF v n1 n2*) *s* ⟹ ∃ *n*. *ni* = *Suc* (*Suc n*)
⟨*proof*⟩

lemma *in-lesI*:
assumes *mi-pre.les s s'*
assumes (*ni1*, *n1*) ∈ *Rmi s*
assumes (*ni2*, *n2*) ∈ *Rmi s*

shows $(ni1, n1) \in Rmi\ s'$ $(ni2, n2) \in Rmi\ s'$
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *ifmi′-spec*: $[\![bdd\text{-}sane\ s;\ bdd\text{-}node\text{-}valid\ s\ ni1;\ bdd\text{-}node\text{-}valid\ s\ ni2]\!] \Longrightarrow ospec\ (ifmi'\ v\ ni1\ ni2\ s)\ (\lambda r.\ r = ifmi\ v\ ni1\ ni2\ s)$
⟨*proof*⟩
lemma *ifmi′-ifmi*: $[\![bdd\text{-}sane\ s;\ bdd\text{-}node\text{-}valid\ s\ ni1;\ bdd\text{-}node\text{-}valid\ s\ ni2]\!] \Longrightarrow ifmi'\ v\ ni1\ ni2\ s = Some\ (ifmi\ v\ ni1\ ni2\ s)$
⟨*proof*⟩

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

lemma *destrmi-someD*: $destrmi'\ e\ bdd = Some\ x \Longrightarrow bdd\text{-}sane\ bdd \wedge bdd\text{-}node\text{-}valid\ bdd\ e$
⟨*proof*⟩

lemma *Rmi-sv*:
assumes $bdd\text{-}sane\ s\ (ni,n) \in Rmi\ s\ (ni',n') \in Rmi\ s$
shows $ni=ni' \Longrightarrow n=n'$
and $ni \neq ni' \Longrightarrow n \neq n'$
⟨*proof*⟩

lemma *True-rep*[*simp*]: $bdd\text{-}sane\ s \Longrightarrow (ni,Trueif) \in Rmi\ s \longleftrightarrow ni=Suc\ 0$
⟨*proof*⟩

lemma *False-rep*[*simp*]: $bdd\text{-}sane\ s \Longrightarrow (ni,Falseif) \in Rmi\ s \longleftrightarrow ni=0$
⟨*proof*⟩

definition $updS\ s\ x\ r = dcl\text{-}update\ (\lambda m.\ m(x \mapsto r))\ s$
thm *Rmi-g.induct*

lemma *updS-dpm*: $dpm\ (updS\ s\ x\ r) = dpm\ s$
⟨*proof*⟩

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

$\langle proof \rangle$

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

interpretation *mi*: *bdd-impl-cmp bdd-sane Rmi tmi′ fmi′ ifmi′ destrmi′ dcl updS* $(=)$
$\langle proof \rangle$

lemma *p-valid-RmiI*: $(Suc\ (Suc\ na),\ b) \in Rmi\ bdd \Longrightarrow pointermap\text{-}p\text{-}valid\ na\ (dpm\ bdd)$
$\langle proof \rangle$
lemma *n-valid-RmiI*: $(na,\ b) \in Rmi\ bdd \Longrightarrow bdd\text{-}node\text{-}valid\ bdd\ na$
$\langle proof \rangle$
lemma *n-valid-Rmi-alt*: $bdd\text{-}node\text{-}valid\ bdd\ na \longleftrightarrow (\exists\, b.\ (na,\ b) \in Rmi\ bdd)$
$\langle proof \rangle$

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 $bdd\text{-}node\text{-}valid\ s'\ ni$
$\langle proof \rangle$

end

8 Array List

Most of this has been contributed by Peter Lammich.

theory *Array-List*
imports
Separation-Logic-Imperative-HOL.Array-Blit
begin

This implements a datastructure that efficiently supports two operations: appending an element and looking up the nth 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 $'a\ array\text{-}list = 'a\ array \times nat$

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

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*]: $<$ *emp* $>$ *arl-empty* $<$*is-array-list* []$>$
  $\langle$*proof*$\rangle$

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

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

definition *arl-append* $\equiv$ λ(*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*]:
  $<$ *is-array-list l a* $>$
    arl-append a x
  $<\lambda a$. *is-array-list* (*l*@[*x*]) *a* $>_t$
  $\langle$*proof*$\rangle$

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

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

end

9 Imparative implementation for Pointermap

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

record $'a$ *pointermap-impl* =
entriesi :: $'a$ *array-list*
getentryi :: $('a,nat)$ *hashtable*
lemma *pointermapieq-exhaust*: $entries\ a = entries\ b \Longrightarrow getentry\ a = getentry\ b \Longrightarrow a = (b :: 'a\ pointermap)$ ⟨*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* $\Longrightarrow$ *pointermap-p-valid p m* $\Longrightarrow$
< *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*));

$ent \leftarrow arl\text{-}append\ (entriesi\ m)\ a;$
$lut \leftarrow hm\text{-}update\ a\ p\ (getentryi\ m);$
$u \leftarrow return\ (\!|entriesi = ent,\ getentryi = lut|\!);$
$return\ (p,u)$
}
)
}

lemmas *pointermap-getmki-defs* = *pointermap-getmki-def pointermap-getmk-def pointermap-insert-def is-pointermap-impl-def*
lemma [*sep-heap-rules*]: $pointermap\text{-}sane\ m \Longrightarrow pointermap\text{-}getmk\ a\ m = (p,u) \Longrightarrow$
$<$ *is-pointermap-impl m mi* $>$
pointermap-getmki a mi
$<\lambda(pi,ui).\ is\text{-}pointermap\text{-}impl\ u\ ui * \uparrow(pi = p)>_t$
⟨*proof*⟩

end

10 Imparative implementation

theory *Conc-Impl*
imports *Pointer-Map-Impl Middle-Impl*
begin

record *bddi* =
dpmi :: $(nat \times nat \times nat)$ *pointermap-impl*
dcli :: $((nat \times nat \times nat),nat)$ *hashtable*
lemma *bdd-exhaust*: $dpm\ a = dpm\ b \Longrightarrow dcl\ a = dcl\ b \Longrightarrow a = (b :: bdd)$ ⟨*proof*⟩

instantiation *prod* :: (*default*, *default*) *default*
begin
definition *default-prod* :: $('a \times 'b) \equiv (default,\ default)$
instance ⟨*proof*⟩
end

instantiation *nat* :: *default*
begin
definition *default-nat* $\equiv 0 :: nat$
instance ⟨*proof*⟩
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*
⟨*proof*⟩

definition *emptyci* :: *bddi Heap* $\equiv$ *do* { $ep \leftarrow pointermap\text{-}empty;\ ehm \leftarrow hm\text{-}new;$

return (|*dpmi*=*ep*, *dcli*=*ehm*|) }
definition *tci bdd* ≡ *return* (*1*::*nat*,*bdd*::*bddi*)
definition *fci bdd* ≡ *return* (*0*::*nat*,*bdd*::*bddi*)
definition *ifci v t e bdd* ≡ (*if t* = *e then return* (*t*, *bdd*) *else do* {
    (*p*,*u*) ← *pointermap-getmki* (*v*, *t*, *e*) (*dpmi bdd*);
    return (*Suc* (*Suc p*), *dpmi-update* (*const u*) *bdd*)
})
definition *destrci* :: *nat* ⇒ *bddi* ⇒ (*nat*, *nat*) *IFEXD Heap* **where**
destrci n bdd ≡ (*case n of*
  0 ⇒ *return FD* |
  Suc 0 ⇒ *return TD* |
  Suc (*Suc p*) ⇒ *pm-pthi* (*dpmi bdd*) *p* ≫= (λ(*v*,*t*,*e*). *return* (*IFD v t e*)))

term *mi.les*

lemma *emptyci-rule*[*sep-heap-rules*]: <*emp*> *emptyci* <*is-bdd-impl emptymi*>$_t$
  ⟨*proof*⟩

lemma [*sep-heap-rules*]: *tmi*′ *bdd* = *Some* (*p*,*bdd*′)
  ⟹ <*is-bdd-impl bdd bddi*>
      tci bddi
    <λ(*pi*,*bddi*′). *is-bdd-impl bdd*′ *bddi*′ ∗ ↑(*pi* = *p*)>
  ⟨*proof*⟩

lemma [*sep-heap-rules*]: *fmi*′ *bdd* = *Some* (*p*,*bdd*′)
  ⟹ <*is-bdd-impl bdd bddi*>
      fci bddi
    <λ(*pi*,*bddi*′). *is-bdd-impl bdd*′ *bddi*′ ∗ ↑(*pi* = *p*)>
⟨*proof*⟩

lemma [*sep-heap-rules*]: *ifmi*′ *v t e bdd* = *Some* (*p*, *bdd*′) ⟹
  <*is-bdd-impl bdd bddi*> *ifci v t e bddi*
  <λ(*pi*,*bddi*′). *is-bdd-impl bdd*′ *bddi*′ ∗ ↑(*pi* = *p*)>$_t$
  ⟨*proof*⟩

lemma *destrci-rule*[*sep-heap-rules*]:
  destrmi′ *n bdd* = *Some r* ⟹
  <*is-bdd-impl bdd bddi*> *destrci n bddi*
  <λ*r*′. *is-bdd-impl bdd bddi* ∗ ↑(*r*′ = *r*)>
  ⟨*proof*⟩

term *mi.restrict-top-impl*

thm *mi.case-ifexi-def*

definition *case-ifexici fti ffi fii ni bddi* ≡ *do* {
  dest ← *destrci ni bddi*;
  case dest of TD ⇒ *fti* | *FD* ⇒ *ffi* | *IFD v ti ei* ⇒ *fii v ti ei*
}

lemma [*sep-decon-rules*]:
  assumes *S*: *mi.case-ifexi fti ffi fii ni bdd = Some r*
  assumes [*sep-heap-rules*]:
    destrmi′ ni bdd = Some TD ⟹ *fti bdd = Some r* ⟹ <*is-bdd-impl bdd bddi*> *ftci* <*Q*>
    destrmi′ ni bdd = Some FD ⟹ *ffi bdd = Some r* ⟹ <*is-bdd-impl bdd bddi*> *ffci* <*Q*>
    ⋀*v t e. destrmi′ ni bdd = Some (IFD v t e)* ⟹ *fii v t e bdd = Some r*
      ⟹ <*is-bdd-impl bdd bddi*> *fici v t e* <*Q*>
  shows <*is-bdd-impl bdd bddi*> *case-ifexici ftci ffci fici ni bddi* <*Q*>
  ⟨*proof*⟩

definition *restrict-topci p vr vl bdd* =
  case-ifexici
    (*return p*)
    (*return p*)
    (λ*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 <*is-bdd-impl bdd bddi*> *restrict-topci p var val bddi*
        <λ*ri. is-bdd-impl bdd bddi* ∗ ↑(*ri = r*)>
  ⟨*proof*⟩

fun *lowest-topsci* **where**
lowest-topsci [] *s = return None* |
lowest-topsci (*e#es*) *s* =
    case-ifexici
      (*lowest-topsci es s*)
      (*lowest-topsci es s*)
      (λ*v t e. do* {
      (*rec*) ← *lowest-topsci es s*;
        (*case rec of*
          Some u ⇒ *return* ((*Some* (*min u v*))) |
          None ⇒ *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 <*is-bdd-impl bdd bddi*> *lowest-topsci es bddi*
  <λ(*ri*). *is-bdd-impl bdd bddi* ∗ ↑(*ri = r* ∧ *bdd′=bdd*)>
⟨*proof*⟩

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 STR ″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>
    param-optci i t e bddi
  <λ(pi,bddi′). is-bdd-impl bdd′ bddi′ ∗ ↑(pi=p)>t
⟨proof⟩
```

lemma *bdd-hm-lookup-rule*:

```
  (dcl bdd (i,t,e) = p) =⇒
  <is-bdd-impl bdd bddi>
    hm-lookup (i, t, e) (dcli bddi)
  <λ(pi). is-bdd-impl bdd bddi ∗ ↑(pi = p)>t
⟨proof⟩
```

lemma *bdd-hm-update-rule′*[*sep-heap-rules*]:

```
  <is-bdd-impl bdd bddi>
    hm-update k v (dcli bddi)
  <λr. is-bdd-impl (updS bdd k v) (dcli-update (const r) bddi) ∗ true>
⟨proof⟩
```

partial-function(*heap*) *iteci-lu* **where**

```
iteci-lu i t e s = do {
  lu ← ht-lookup (i,t,e) (dcli s);
  (case lu of Some b ⇒ return (b,s)
    | None ⇒ do {
      (po,s) ← param-optci i t e s;
      (case po of Some b ⇒ do {
        return (b,s)}
      | None ⇒ 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-lu ti tt te s;
        (fb,s) ← iteci-lu 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)
       }
         | None ⇒ raise STR ′′Cannot happen′′ )})
  })}
```

term *ht-lookup*
declare *iteci-lu.simps*[*code*]
thm *iteci-lu.simps*[*unfolded restrict-topci-def case-ifexici-def param-optci-def lowest-topsci.simps*]
partial-function(*heap*) *iteci-lu-code* **where** *iteci-lu-code i t e s = do {*

```
  lu ← hm-lookup (i, t, e) (dcli s);
  case lu of None ⇒ let po = if i = 1 then 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)
  | 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
definition fstp = apsnd fst

definition the-thing-By f l = (let
  nub = remdups (map fst l) in
  map (λe. (e, map snd (filter (λg. (f e (fst g))) l))) nub)
definition the-thing = the-thing-By (=)


definition graphifyci :: string ⇒ nat ⇒ bddi ⇒ string Heap where
graphifyci name ep bdd ≡ do {
  s ← serializeci ep bdd;
  let e = map fst s;
  l ← mapM (labelci bdd) (rev (remdups (map fst e @ map snd e)));
  let grp = (map (λl. foldr (λa t. t @ a @ '';'') (snd l) ''{rank=same;'' @ ''}
'') (the-thing (map fstp l)));
  e ← mapM (graphifyci1 bdd) s;
  let emptyhlp = (case ep of 0 ⇒ ''F;
'' | Suc 0 ⇒ ''T;
'' | - ⇒ '''');
  return (''digraph '' @ name @ '' {
'' @ concat (map trd l) @ concat grp @ concat e @ emptyhlp @ ''}'')
}

end
```

11 Collapsing the levels

theory *Level-Collapse*
imports *Conc-Impl*
begin

The theory up to this point is implemented in a way that separated the different aspects into different levels. This is highly beneficial for us, since it allows us to tackle the difficulties arising in small chunks. However, exporting this to the user would be highly impractical. Thus, this theory collapses all the different levels (i.e. refinement steps) and relates the computations in the heap monad to *boolfunc*.

```
definition bddmi-rel cs ≡ {(a,c)|a b c. (a,b) ∈ bf-ifex-rel ∧ (c,b) ∈ Rmi cs}
definition bdd-relator :: (nat boolfunc × nat) set ⇒ bddi ⇒ assn where
bdd-relator p s ≡ ∃A cs. is-bdd-impl cs s * ↑(p ⊆ (bddmi-rel cs) ∧ bdd-sane cs) *
true
```

The *assn* predicate *bdd-relator* is the interface that is exposed to the user. (The contents of the definition are not exposed.)

lemma *bdd-relator-mono*[*intro*!]: $q \subseteq p \Longrightarrow$ *bdd-relator* $p\ s \Longrightarrow_A$ *bdd-relator* $q\ s$
⟨*proof*⟩

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

thm *bdd-relator-def*[*unfolded bddmi-rel-def*, *simplified*]
lemma *join-hlp1*: *is-bdd-impl* $a\ s *$ *is-bdd-impl* $b\ s \Longrightarrow_A$ *is-bdd-impl* $a\ s *$ *is-bdd-impl* $b\ s * \uparrow(a = b)$
⟨*proof*⟩

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

lemma *add-true-asm*:
assumes $<b * true>\ p\ <a>_t$
shows $<b>\ p\ <a>_t$
⟨*proof*⟩

lemma *add-anything*:
assumes $<b>\ p\ <a>$
shows $<b * x>\ p\ <\lambda r.\ a\ r * x>_t$
⟨*proof*⟩

lemma *add-true*:
assumes $<b>\ p\ <a>_t$
shows $<b * true>\ p\ <a>_t$
⟨*proof*⟩

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

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

lemma *node-relatorI*: $x \in y \Longrightarrow$ *node-relator* $x\ y$ ⟨*proof*⟩
lemma *node-relatorD*: *node-relator* $x\ y \Longrightarrow x \in y$ ⟨*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*]:
⟦*node-relator* (ib, ic) *rp*; *node-relator* (tb, tc) *rp*; *node-relator* (eb, ec) *rp*⟧ $\Longrightarrow$

<*bdd-relator rp s*>
  iteci-lu ic tc ec s
<λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-ite ib tb eb*,*r*) *rp*) *s*′>
  ⟨*proof*⟩

lemma *tci-rule*[*sep-heap-rules*]:
<*bdd-relator rp s*>
  tci s
<λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-True*,*r*) *rp*) *s*′>
  ⟨*proof*⟩

lemma *fci-rule*[*sep-heap-rules*]:
<*bdd-relator rp s*>
  fci s
<λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-False*,*r*) *rp*) *s*′>
  ⟨*proof*⟩

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*]:
⟦(*tb*, *tc*) ∈ *rp*; (*eb*, *ec*) ∈ *rp*⟧ ⟹
<*bdd-relator rp s*>
  ifci v tc ec s
<λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-if v tb eb*,*r*) *rp*) *s*′>

This probably doesn't hold.

⟨*proof*⟩

lemma *notci-rule*[*sep-heap-rules*]:
  assumes *node-relator* (*tb*, *tc*) *rp*
  shows <*bdd-relator rp s*> *notci tc s* <λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-not tb*,*r*) *rp*) *s*′>
⟨*proof*⟩

lemma *cirules1*[*sep-heap-rules*]:
  assumes *node-relator* (*tb*, *tc*) *rp node-relator* (*eb*, *ec*) *rp*
  shows
    <*bdd-relator rp s*> *andci tc ec s* <λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-and tb eb*,*r*) *rp*) *s*′>
    <*bdd-relator rp s*> *orci tc ec s* <λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-or tb eb*,*r*) *rp*) *s*′>
    <*bdd-relator rp s*> *biimpci tc ec s* <λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-biimp tb eb*,*r*) *rp*) *s*′>
    <*bdd-relator rp s*> *xorci tc ec s* <λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-xor tb eb*,*r*) *rp*) *s*′>

⟨*proof*⟩

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* <λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-nand tb eb*,*r*) *rp*) *s*′>
<*bdd-relator rp s*> *norci tc ec s* <λ(*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* <λ(*r*,*s*′). *bdd-relator* (*insert* (*bf-lit v*,*r*) *rp*) *s*′>
⟨*proof*⟩

lemma *tautci-rule*[*sep-heap-rules*]:
shows *node-relator* (*tb*, *tc*) *rp* ⟹ <*bdd-relator rp s*> *tautci tc s* <λ*r*. *bdd-relator rp s* ∗ ↑(*r* ⟷ *tb* = *bf-True*)>
⟨*proof*⟩

lemma *emptyci-rule*[*sep-heap-rules*]:
shows <*emp*> *emptyci* <λ*r*. *bdd-relator* {} *r*>
⟨*proof*⟩

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

Efficient comparison of two nodes.

definition *eqci a b* ≡ *return* (*a* = *b*)

lemma *iteeq-rule*[*sep-heap-rules*]:
⟦*node-relator* (*xb*, *xc*) *rp*; *node-relator* (*yb*, *yc*) *rp*⟧ ⟹
<*bdd-relator rp s*>
eqci xc yc
<λ*r*. ↑(*r* ⟷ *xb* = *yb*)>$_t$
⟨*proof*⟩

end

12 Tests and examples

theory *BDD-Examples*
imports *Level-Collapse*
begin

Just two simple examples:

lemma <*emp*> *do* {
s ← *emptyci*;
(*t*,*s*) ← *tci s*;
tautci t s

$\} <\lambda r.\ \uparrow(r = \mathit{True})>_t$
⟨*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
$\} <\uparrow>_t$
⟨*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.