BinarySearchTree

Larry Paulson

March 17, 2025

Contents

1 Isar-style Reasoning for Binary Tree Operations

theory *BinaryTree* **imports** *Main* **begin**

We prove correctness of operations on binary search tree implementing a set.

This document is LGPL.

Author: Viktor Kuncak, MIT CSAIL, November 2003

2 Tree Definition

datatype *'a Tree* = *Tip* | *T 'a Tree 'a 'a Tree*

primrec
  setOf :: *'a Tree* => *'a set*
  — set abstraction of a tree
where
  setOf Tip = {}
| *setOf* (*T t1 x t2*) = (*setOf t1*) *Un* (*setOf t2*) *Un* {*x*}

type-synonym
  — we require index to have an irreflexive total order <
  — apart from that, we do not rely on index being int
  index = *int*

type-synonym — hash function type
  'a hash = *'a* => *index*

definition *eqs* :: *'a hash* => *'a* => *'a set* **where**
  — equivalence class of elements with the same hash code
  eqs h x == {*y*. *h y* = *h x*}

primrec
  sortedTree :: *'a hash* => *'a Tree* => *bool*
  — check if a tree is sorted
where
  sortedTree h Tip = *True*
| *sortedTree h* (*T t1 x t2*) =
    (*sortedTree h t1* ∧
     (∀ *l* ∈ *setOf t1*. *h l* < *h x*) ∧
     (∀ *r* ∈ *setOf t2*. *h x* < *h r*) ∧
     sortedTree h t2)

lemma *sortLemmaL*:
  sortedTree h (*T t1 x t2*) ==> *sortedTree h t1* **by** *simp*

lemma *sortLemmaR*:
  sortedTree h (*T t1 x t2*) ==> *sortedTree h t2* **by** *simp*

3 Tree Lookup

primrec
  $tlookup :: 'a\ hash => index => 'a\ Tree => 'a\ option$
where
  $tlookup\ h\ k\ Tip = None$
| $tlookup\ h\ k\ (T\ t1\ x\ t2) =$
   $(if\ k < h\ x\ then\ tlookup\ h\ k\ t1$
    $else\ if\ h\ x < k\ then\ tlookup\ h\ k\ t2$
    $else\ Some\ x)$

lemma *tlookup-none*:
  $sortedTree\ h\ t \Longrightarrow (tlookup\ h\ k\ t = None) \Longrightarrow x \in setOf\ t \Longrightarrow h\ x \neq k$
  by (*induction t*, *auto*)

lemma *tlookup-some*:
    $sortedTree\ h\ t \Longrightarrow (tlookup\ h\ k\ t = Some\ x) \Longrightarrow x \in setOf\ t \wedge h\ x = k$
proof (*induction t*)
  case *Tip*
  then show *?case* **by** *auto*
next
  case (*T t1 x2 t2*)
  then show *?case*
    apply *simp*
    by (*metis linorder-less-linear option.sel*)
qed

definition *sorted-distinct-pred* :: $'a\ hash => 'a => 'a => 'a\ Tree => bool$ **where**
  — No two elements have the same hash code
  sorted-distinct-pred h a b t
  $\equiv sortedTree\ h\ t \wedge a \in setOf\ t \wedge b \in setOf\ t \wedge h\ a = h\ b \longrightarrow a = b$

declare *sorted-distinct-pred-def* [*simp*]

sorted-distinct-pred holds for out trees:

lemma *sorted-distinct*: *sorted-distinct-pred h a b t* (**is** *?P t*)
  by (*induct t*) *force+*

lemma *tlookup-finds*: — if a node is in the tree, lookup finds it
  assumes $sortedTree\ h\ t\ y \in setOf\ t$
  shows $tlookup\ h\ (h\ y)\ t = Some\ y$
proof (*cases tlookup h* (*h y*) *t*)
  case *None*
  with *assms* **show** *?thesis*
    by (*meson tlookup-none*)
next
  case (*Some z*)
  with *assms* **show** *?thesis*
    by (*metis sorted-distinct sorted-distinct-pred-def tlookup-some*)

qed

3.1 Tree membership as a special case of lookup

definition *memb* :: *'a hash => 'a => 'a Tree => bool* **where**
  memb h x t ==
   (*case* (*tlookup h* (*h x*) *t*) *of*
     None => False
   | *Some z =>* (*x=z*))

lemma *memb-spec*:
  assumes *sortedTree h t* **shows** *memb h x t* = (*x* ∈ *setOf t*)
proof (*cases tlookup h* (*h x*) *t*)
  case *None*
  then show *?thesis*
    by (*metis memb-def option.simps*(*4*) *assms tlookup-none*)
next
  case (*Some z*)
  with *assms tlookup-some* **have** *z* ∈ *setOf t* **by** *fastforce*
  then show *?thesis*
    using *memb-def assms Some tlookup-finds* **by** *force*
qed

declare *sorted-distinct-pred-def* [*simp del*]

4 Insertion into a Tree

primrec
  binsert :: *'a hash => 'a => 'a Tree => 'a Tree*
where
  binsert h e Tip = (*T Tip e Tip*)
| *binsert h e* (*T t1 x t2*) =
    (*if h e* < *h x then* (*T* (*binsert h e t1*) *x t2*)
          else (*if h x* < *h e then* (*T t1 x* (*binsert h e t2*))
            else (*T t1 e t2*)))

A technique for proving disjointness of sets.

lemma *disjCond*: [| !! *x*. [| *x*:*A*; *x*:*B* |] ==> *False* |] ==> *A Int B* = {}
  by *fastforce*

The following is a proof that insertion correctly implements the set interface. Compared to *BinaryTree-TacticStyle*, the claim is more difficult, and this time we need to assume as a hypothesis that the tree is sorted.

lemma *binsert-set*: *sortedTree h t* ⟹
          setOf (*binsert h e t*) = (*setOf t*) − (*eqs h e*) *Un* {*e*}
  by (*induction t*) (*auto simp*: *eqs-def*)

Using the correctness of set implementation, preserving sortedness is still simple.

lemma *binsert-sorted*: *sortedTree h t* $-->$ *sortedTree h* (*binsert h x t*)
  by (*induct t*) (*auto simp add*: *binsert-set*)

We summarize the specification of binsert as follows.

corollary *binsert-spec*: *sortedTree h t* $-->$
    sortedTree h (*binsert h x t*) $\wedge$
    setOf (*binsert h e t*) = (*setOf t*) $-$ (*eqs h e*) *Un* $\{e\}$
  by (*simp add*: *binsert-set binsert-sorted*)

5 Removing an element from a tree

These proofs are influenced by those in *BinaryTree-Tactic*

```
primrec
  rm :: 'a hash => 'a Tree => 'a
  — rightmost element of a tree
where
rm h (T t1 x t2) =
  (if t2=Tip then x else rm h t2)

primrec
  wrm :: 'a hash => 'a Tree => 'a Tree
  — tree without the rightmost element
where
wrm h (T t1 x t2) =
  (if t2=Tip then t1 else (T t1 x (wrm h t2)))

primrec
  wrmrm :: 'a hash => 'a Tree => 'a Tree * 'a
  — computing rightmost and removal in one pass
where
wrmrm h (T t1 x t2) =
  (if t2=Tip then (t1,x)
   else (T t1 x (fst (wrmrm h t2)),
         snd (wrmrm h t2)))

primrec
  remove :: 'a hash => 'a => 'a Tree => 'a Tree
   — removal of an element from the tree
where
  remove h e Tip = Tip
| remove h e (T t1 x t2) =
    (if h e < h x then (T (remove h e t1) x t2)
     else if h x < h e then (T t1 x (remove h e t2))
     else (if t1=Tip then t2
           else let (t1p,r) = wrmrm h t1
                in (T t1p r t2)))
```

theorem *wrmrm-decomp*: $t \neq Tip \Longrightarrow$ *wrmrm h t* = (*wrm h t*, *rm h t*)

by (*induct t*) *auto*

lemma *rm-set*: $t \neq Tip \Longrightarrow sortedTree\ h\ t \Longrightarrow rm\ h\ t \in setOf\ t$
by (*induct t*) *auto*

lemma *wrm-set*: $t \neq Tip \Longrightarrow sortedTree\ h\ t \Longrightarrow$
$setOf\ (wrm\ h\ t) = setOf\ t - \{rm\ h\ t\}$
proof (*induction t*)
case *Tip*
then show *?case*
by *blast*
next
case (*T t1 x t2*)
show *?case*
proof (*cases t2 = Tip*)
case *True*
with *T* **show** *?thesis*
by *fastforce*
next
case *False*
with *T rm-set* **show** *?thesis*
by *fastforce*
qed
qed

lemma *wrm-set1*: $t \neq Tip \Longrightarrow sortedTree\ h\ t \Longrightarrow setOf\ (wrm\ h\ t) <= setOf\ t$
by (*auto simp add*: *wrm-set*)

lemma *wrm-sort*: $t \neq Tip \Longrightarrow sortedTree\ h\ t \Longrightarrow sortedTree\ h\ (wrm\ h\ t)$
by (*induction t*) (*auto simp*: *wrm-set*)

lemma *wrm-less-rm*:
$t \neq Tip \Longrightarrow sortedTree\ h\ t \Longrightarrow l \in setOf\ (wrm\ h\ t) \Longrightarrow$
$h\ l < h\ (rm\ h\ t)$
by (*induction t arbitrary*: *l*) (*use rm-set* **in** *fastforce*)+

lemma *remove-set*:
$sortedTree\ h\ t \Longrightarrow setOf\ (remove\ h\ e\ t) = setOf\ t - eqs\ h\ e$
proof (*induction t*)
case *Tip*
then show *?case* **by** *auto*
next
case (*T t1 x t2*)
show *?case*
proof (*cases rule*: *linorder-cases* [*of h e h x*])
case *less*
with *T* **show** *?thesis*
by (*auto simp*: *eqs-def*)
next

```
    case equal
    then have ∗: (setOf t2) ∩ (eqs h e) = {}
      using T.prems sup.strict-order-iff by (fastforce simp: eqs-def)
    show ?thesis
    proof (cases t1 = Tip)
      case True
      with equal ∗ show ?thesis
        by (fastforce simp: eqs-def)
    next
      case False
      with equal show ?thesis
        using T.prems rm-set wrm-set wrmrm-decomp by (fastforce simp: eqs-def)
    qed
  next
    case greater
    with T show ?thesis
      by (auto simp: eqs-def)
  qed
qed

lemma remove-sort: sortedTree h t ⟹ sortedTree h (remove h e t)
proof (induction t)
  case Tip
  then show ?case
    by simp
next
  case (T t1 x t2)
  show ?case
  proof (cases h e < h x)
    case True
    then show ?thesis
      using T remove-set by force
  next
    case False
    with T show ?thesis
      using DiffD1 remove-set rm-set wrm-less-rm wrm-sort wrmrm-decomp
      by fastforce
  qed
qed
```

We summarize the specification of remove as follows.

```
corollary remove-spec:
  sortedTree h t ⟹
    sortedTree h (remove h e t) ∧ setOf (remove h e t) = setOf t − eqs h e
  by (simp add: remove-sort remove-set)

definition test = tlookup id 4 (remove id 3 (binsert id 4 (binsert id 3 Tip)))

export-code test
```

in *SML* **module-name** *BinaryTree-Code* **file** ‹*BinaryTree-Code.ML*›

end

6 Mostly Isar-style Reasoning for Binary Tree Operations

theory *BinaryTree-Map* **imports** *BinaryTree* **begin**

We prove correctness of map operations implemented using binary search trees from BinaryTree.

This document is LGPL.

Author: Viktor Kuncak, MIT CSAIL, November 2003

7 Map implementation and an abstraction function

type-synonym
  'a tarray = (*index* ∗ *'a*) *Tree*

definition *valid-tmap* :: *'a tarray* => *bool* **where**
  valid-tmap t ≡ *sortedTree fst t*

declare *valid-tmap-def* [*simp*]

definition *mapOf* :: *'a tarray* => *index* => *'a option* **where**
  — the abstraction function from trees to maps
  mapOf t i ≡
   (*case* (*tlookup fst i t*) *of*
     None => *None*
   | *Some ia* => *Some* (*snd ia*))

8 Auxiliary Properties of our Implementation

lemma *mapOf-lookup1*: *tlookup fst i t* = *None* ==> *mapOf t i* = *None*
  by (*simp add*: *mapOf-def*)

lemma *mapOf-lookup2*: *tlookup fst i t* = *Some* (*j*,*a*) ==> *mapOf t i* = *Some a*
  by (*simp add*: *mapOf-def*)

lemma *mapOf-lookup3*:
   assumes *h*: *mapOf t i* = *None*
   shows *tlookup fst i t* = *None*
proof (*cases tlookup fst i t*)
  case *None*
  then show *?thesis* **by** *assumption*

next
  case (*Some ia*)
  then show *?thesis*
    by (*metis h mapOf-def option.discI option.simps*(*5*))
qed

lemma *mapOf-lookup4*:
  assumes *v*: *valid-tmap t*
  assumes *h*: *mapOf t i* = *Some a*
  shows *tlookup fst i t* = *Some* (*i*,*a*)
proof (*cases tlookup fst i t*)
  case *None*
  then show *?thesis*
    by (*metis h mapOf-lookup1 option.discI*)
next
  case (*Some ia*)
  then show *?thesis*
    by (*metis h mapOf-def option.simps*(*5*) *prod.exhaust-sel tlookup-some v valid-tmap-def*)
qed

8.1 Lemmas *mapset-none* and *mapset-some* establish a relation between the set and map abstraction of the tree

lemma *mapset-none*:
  assumes *valid-tmap t*
  shows (*mapOf t i* = *None*) = (∀ *a*. (*i*,*a*) ∉ *setOf t*)
  using *assms*
  unfolding *valid-tmap-def*
  by (*metis mapOf-lookup1 mapOf-lookup3 not-None-eq split-pairs tlookup-none tlookup-some*)

lemma *mapset-some*:
  assumes *valid-tmap t*
  shows (*mapOf t i* = *Some a*) = ((*i*,*a*) ∈ *setOf t*)
  unfolding *valid-tmap-def*
  using *assms mapOf-lookup2 mapOf-lookup4 tlookup-finds tlookup-some*
  by *fastforce*

9 Empty Map

lemma *mnew-spec-valid*: *valid-tmap Tip*
  by (*simp add*: *mapOf-def*)

lemma *mtip-spec-empty*: *mapOf Tip k* = *None*
  by (*simp add*: *mapOf-def*)

10 Map Update Operation

```
definition mupdate :: index => 'a => 'a tarray => 'a tarray where
  mupdate i a t ≡ binsert fst (i,a) t

lemma mupdate-map:
  assumes valid-tmap t
  shows mapOf (mupdate i a t) = (mapOf t)(i |-> a)
proof
  fix j
  let ?tr = binsert fst (i,a) t
  have upres: mupdate i a t = ?tr by (simp add: mupdate-def)
  from assms binsert-set
  have setSpec: setOf ?tr = setOf t − eqs fst (i,a) Un {(i,a)}
    by fastforce
  from assms binsert-sorted have vr: valid-tmap ?tr
    by fastforce
  show mapOf (mupdate i a t) j = ((mapOf t)(i |-> a)) j
  proof (cases i = j)
    case True
    then show ?thesis
      using mapset-some setSpec upres vr by fastforce
  next
    case False note i2nei = this
    from i2nei have rhs-res: ((mapOf t)(i |-> a)) j = mapOf t j by auto
    have lhs-res: mapOf (mupdate i a t) j = mapOf t j
    proof (cases mapOf t j)
      case None
      then show ?thesis
        by (metis DiffD1 Un-empty-right Un-insert-right i2nei insertE mapset-none
            prod.inject setSpec upres assms vr)
    next
      case (Some z)
      then have mapSome: mapOf t j = Some z
        by simp
      then have (j,z) ∈ setOf t
        by (meson mapset-some assms)
      with setSpec i2nei mapset-some vr have mapOf ?tr j = Some z
        by (fastforce simp: eqs-def)
      then show ?thesis
        by (simp add: mapSome upres)
    qed
    from lhs-res rhs-res show ?thesis by simp
  qed
qed

lemma mupdate-valid:
  assumes valid-tmap t shows valid-tmap (mupdate i a t)
  by (metis binsert-sorted mupdate-def assms valid-tmap-def)
```

11 Map Remove Operation

definition *mremove* :: *index* => *'a tarray* => *'a tarray* **where**
mremove i t ≡ *remove fst* (*i*, *undefined*) *t*

lemma *mremove-valid*:
assumes *valid-tmap t*
shows *valid-tmap* (*mremove i t*)
by (*metis mremove-def remove-sort assms valid-tmap-def*)

lemma *mremove-map*:
assumes *valid-tmap t*
shows *mapOf* (*mremove i t*) *i* = *None*
using *assms*
by (*auto simp*: *mremove-def eqs-def mapset-none remove-set remove-sort*)

end

12 Tactic-Style Reasoning for Binary Tree Operations

theory *BinaryTree-TacticStyle* **imports** *Main* **begin**

This example theory illustrates automated proofs of correctness for binary tree operations using tactic-style reasoning. The current proofs for remove operation are by Tobias Nipkow, some modifications and the remaining tree operations are by Viktor Kuncak.

13 Definition of a sorted binary tree

datatype *tree* = *Tip* | *Nd tree nat tree*

primrec *set-of* :: *tree* => *nat set*
— The set of nodes stored in a tree.
where
set-of Tip = {}
| *set-of*(*Nd l x r*) = *set-of l Un set-of r Un* {*x*}

primrec *sorted* :: *tree* => *bool*
— Tree is sorted
where
sorted Tip = *True*
| *sorted*(*Nd l y r*) =
(*sorted l* & *sorted r* & ($\forall x \in$ *set-of l*. $x < y$) & ($\forall z \in$ *set-of r*. $y < z$))

14 Tree Membership

```
primrec
  memb :: nat => tree => bool
where
  memb e Tip = False
| memb e (Nd t1 x t2) =
   (if e < x then memb e t1
    else if x < e then memb e t2
    else True)

lemma member-set: sorted t --> memb e t = (e : set-of t)
by (induct t) auto
```

15 Insertion operation

```
primrec binsert :: nat => tree => tree
— Insert a node into sorted tree.
where
  binsert x Tip = (Nd Tip x Tip)
| binsert x (Nd t1 y t2) = (if x < y then
                                (Nd (binsert x t1) y t2)
                            else
                              (if y < x then
                                (Nd t1 y (binsert x t2))
                               else (Nd t1 y t2)))

theorem set-of-binsert [simp]: set-of (binsert x t) = set-of t Un {x}
by (induct t) auto

theorem binsert-sorted: sorted t --> sorted (binsert x t)
by (induct t) (auto simp add: set-of-binsert)

corollary binsert-spec:
sorted t ==>
   sorted (binsert x t) &
   set-of (binsert x t) = set-of t Un {x}
by (simp add: binsert-sorted)
```

16 Remove operation

```
primrec
  rm :: tree => nat — find the rightmost element in the tree
where
  rm(Nd l x r) = (if r = Tip then x else rm r)
primrec
  rem :: tree => tree — find the tree without the rightmost element
where
```

```
  rem(Nd l x r) = (if r=Tip then l else Nd l x (rem r))

primrec
  remove:: nat => tree => tree — remove a node from sorted tree
where
  remove x Tip = Tip
| remove x (Nd l y r) =
    (if x < y then Nd (remove x l) y r else
     if y < x then Nd l y (remove x r) else
     if l = Tip then r
     else Nd (rem l) (rm l) r)

lemma rm-in-set-of: t ~= Tip ==> rm t : set-of t
  by (induct t) auto

lemma set-of-rem: t ~= Tip ==> set-of t = set-of(rem t) Un {rm t}
  by (induct t) auto

lemma [simp]: [| t ~= Tip; sorted t  |] ==> sorted(rem t)
  by (induct t) (auto simp add:set-of-rem)

lemma sorted-rem: [| t ~= Tip; x ∈ set-of(rem t); sorted t |] ==> x < rm t
  by (induct t) (auto simp add:set-of-rem split:if-splits)

theorem set-of-remove [simp]: sorted t ==> set-of(remove x t) = set-of t − {x}
apply(induct t)
 apply simp
apply simp
apply(rule conjI)
 apply fastforce
apply(rule impI)
apply(rule conjI)
 apply fastforce
apply(fastforce simp:set-of-rem)
done

theorem remove-sorted: sorted t ==> sorted(remove x t)
by (induct t) (auto intro: less-trans rm-in-set-of sorted-rem)

corollary remove-spec: — summary specification of remove
sorted t ==>
   sorted (remove x t) &
   set-of (remove x t) = set-of t − {x}
by (simp add: remove-sorted)
```

Finally, note that rem and rm can be computed using a single tree traversal given by remrm.

```
primrec remrm :: tree => tree * nat
where
```

remrm(*Nd l x r*) = (*if r*=*Tip then* (*l*,*x*) *else*
let (*r'*,*y*) = *remrm r in* (*Nd l x r'*,*y*))

lemma *t* ~= *Tip* ==> *remrm t* = (*rem t*, *rm t*)
by (*induct t*) (*auto simp*:*Let-def*)

We can test this implementation by generating code.

definition *test* = *memb 4* (*remove* (*3*::*nat*) (*binsert 4* (*binsert 3 Tip*)))

export-code *test*
in *SML* **module-name** *BinaryTree-TacticStyle-Code* **file** ‹*BinaryTree-TacticStyle-Code.ML*›

end