BinarySearchTree

Larry Paulson

December 14, 2021

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* ⟨*proof*⟩
lemma *sortLemmaR*:
  sortedTree h (*T t1 x t2*) ==> *sortedTree h t2* ⟨*proof*⟩

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 & (tlookup h k t = None) −−> (∀ x∈setOf t. h x ~= k)
⟨*proof*⟩

lemma *tlookup-some*:
sortedTree h t & (tlookup h k t = Some x) −−> x:setOf t & h x = k
⟨*proof*⟩

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 == sortedTree h t &
a:setOf t & b:setOf t & h a = h b −−>
a = b

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

— for case analysis on three cases
lemma *cases3*: [| *C1 ==> G*; *C2 ==> G*; *C3 ==> G*;
C1 | C2 | C3 |] ==> G
⟨*proof*⟩

sorted-distinct-pred holds for out trees:

lemma *sorted-distinct*: *sorted-distinct-pred h a b t* (**is** *?P t*)
⟨*proof*⟩

lemma *tlookup-finds*: — if a node is in the tree, lookup finds it
sortedTree h t & y:setOf t −−>
tlookup h (h y) t = Some y
⟨*proof*⟩

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 assumes *s*: *sortedTree h t*

shows *memb-spec*: *memb h x t* = (*x* : *setOf t*)
⟨*proof*⟩

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* = {}
⟨*proof*⟩

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*}
(**is** *?P t*)
⟨*proof*⟩

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

lemma *binsert-sorted*: *sortedTree h t* −−> *sortedTree h* (*binsert h x t*)
⟨*proof*⟩

We summarize the specification of binsert as follows.

corollary *binsert-spec*: *sortedTree h t* −−>
sortedTree h (*binsert h x t*) &
setOf (*binsert h e t*) = (*setOf t*) − (*eqs h e*) *Un* {*e*}
⟨*proof*⟩

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 ~= Tip --> wrmrm h t = (wrm h t, rm h t)
⟨proof⟩

lemma rm-set: t ~= Tip & sortedTree h t --> rm h t : setOf t
⟨proof⟩

lemma wrm-set: t ~= Tip & sortedTree h t -->
                setOf (wrm h t) = setOf t - {rm h t} (is ?P t)
⟨proof⟩

lemma wrm-set1: t ~= Tip & sortedTree h t --> setOf (wrm h t) <= setOf t
⟨proof⟩

lemma wrm-sort: t ~= Tip & sortedTree h t --> sortedTree h (wrm h t) (is ?P
t)
⟨proof⟩
```

lemma *wrm-less-rm*:
  $t \sim= Tip$ & *sortedTree h t* −−>
   (∀ $l \in$ *setOf* (*wrm h t*). *h l* < *h* (*rm h t*)) (**is** *?P t*)
⟨*proof*⟩

lemma *remove-set*: *sortedTree h t* −−>
  setOf (*remove h e t*) = *setOf t* − *eqs h e* (**is** *?P t*)
⟨*proof*⟩

lemma *remove-sort*: *sortedTree h t* −−>
                sortedTree h (*remove h e t*) (**is** *?P t*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mapOf-lookup2*: *tlookup fst i t = Some* (*j*,*a*) *==> mapOf t i = Some a*
⟨*proof*⟩

lemma assumes *h*: *mapOf t i = None*
shows *mapOf-lookup3*: *tlookup fst i t = None*
⟨*proof*⟩

lemma assumes *v*: *valid-tmap t*
assumes *h*: *mapOf t i = Some a*
shows *mapOf-lookup4*: *tlookup fst i t = Some* (*i*,*a*)
⟨*proof*⟩

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

lemma assumes *v*: *valid-tmap t*
shows *mapset-none*: (*mapOf t i = None*) = (∀ *a*. (*i*,*a*) ∉ *setOf t*)
⟨*proof*⟩

lemma assumes *v*: *valid-tmap t*
shows *mapset-some*: (*mapOf t i = Some a*) = ((*i*,*a*) : *setOf t*)
⟨*proof*⟩

9 Empty Map

lemma *mnew-spec-valid*: *valid-tmap Tip*
⟨*proof*⟩

lemma *mtip-spec-empty*: *mapOf Tip k = None*
⟨*proof*⟩

10 Map Update Operation

definition *mupdate* :: *index => 'a => 'a tarray => 'a tarray* **where**

mupdate i a t == binsert fst (i,a) t

lemma assumes *v*: *valid-tmap t*
shows *mupdate-map*: *mapOf (mupdate i a t) = (mapOf t)(i |−> a)*
⟨*proof*⟩

lemma assumes *v*: *valid-tmap t*
shows *mupdate-valid*: *valid-tmap (mupdate i a t)*
⟨*proof*⟩

11 Map Remove Operation

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

lemma assumes *v*: *valid-tmap t*
shows *mremove-valid*: *valid-tmap (mremove i t)*
⟨*proof*⟩

lemma assumes *v*: *valid-tmap t*
shows *mremove-map*: *mapOf (mremove i t) i = None*
⟨*proof*⟩

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 & (∀ x ∈ set-of l. x < y) & (∀ z ∈ 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)*
⟨*proof*⟩

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}*
⟨*proof*⟩

theorem *binsert-sorted*: *sorted t −−> sorted (binsert x t)*
⟨*proof*⟩

corollary *binsert-spec*:
sorted t ==>
  sorted (binsert x t) &
  set-of (binsert x t) = set-of t Un {x}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-of-rem*: *t* ~= *Tip* ==> *set-of t* = *set-of*(*rem t*) *Un* {*rm t*}
⟨*proof*⟩

lemma [*simp*]: [| *t* ~= *Tip*; *sorted t* |] ==> *sorted*(*rem t*)
⟨*proof*⟩

lemma *sorted-rem*: [| *t* ~= *Tip*; *x* ∈ *set-of*(*rem t*); *sorted t* |] ==> *x* < *rm t*
⟨*proof*⟩

theorem *set-of-remove* [*simp*]: *sorted t* ==> *set-of*(*remove x t*) = *set-of t* − {*x*}
⟨*proof*⟩

theorem *remove-sorted*: *sorted t* ==> *sorted*(*remove x t*)
⟨*proof*⟩

corollary *remove-spec*: — summary specification of remove
sorted t ==>
   sorted (*remove x t*) &
   set-of (*remove x t*) = *set-of t* − {*x*}
⟨*proof*⟩

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*)

⟨*proof*⟩

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