

BinarySearchTree

Larry Paulson

March 17, 2025

Contents

1	Isar-style Reasoning for Binary Tree Operations	1
2	Tree Definition	1
3	Tree Lookup	2
3.1	Tree membership as a special case of lookup	3
4	Insertion into a Tree	3
5	Removing an element from a tree	4
6	Mostly Isar-style Reasoning for Binary Tree Operations	5
7	Map implementation and an abstraction function	5
8	Auxiliary Properties of our Implementation	6
8.1	Lemmas <i>mapset-none</i> and <i>mapset-some</i> establish a relation between the set and map abstraction of the tree	6
9	Empty Map	6
10	Map Update Operation	7
11	Map Remove Operation	7
12	Tactic-Style Reasoning for Binary Tree Operations	7
13	Definition of a sorted binary tree	8
14	Tree Membership	8
15	Insertion operation	8
16	Remove operation	9

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 :: $'a \text{Tree} \Rightarrow 'a \text{set}$

— 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 = $'a \Rightarrow \text{index}$

definition *eqs* :: $'a \text{hash} \Rightarrow 'a \Rightarrow 'a \text{set}$ **where**

— equivalence class of elements with the same hash code

eques h x == {y. h y = h x}

primrec

sortedTree :: $'a \text{hash} \Rightarrow 'a \text{Tree} \Rightarrow \text{bool}$

— check if a tree is sorted

where

sortedTree h Tip = *True*

| *sortedTree h (T t1 x t2)* =

(*sortedTree h t1* \wedge

($\forall l \in \text{setOf } t1. h l < h x$) \wedge

($\forall r \in \text{setOf } t2. h x < h r$) \wedge

sortedTree h t2)

lemma *sortLemmaL*:

sortedTree h (T t1 x t2) ==> sortedTree h t1 $\langle \text{proof} \rangle$

lemma *sortLemmaR*:

sortedTree h (T t1 x t2) ==> sortedTree h t2 $\langle \text{proof} \rangle$

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]

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
  assumes sortedTree h t y ∈ setOf t
  shows 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 memb-spec:
  assumes sortedTree h t shows 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}
  ⟨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*: $t \neq \text{Tip} \implies \text{wrmrm } h \ t = (\text{wrm } h \ t, \ \text{rm } h \ t)$
 $\langle \text{proof} \rangle$

lemma *rm-set*: $t \neq \text{Tip} \implies \text{sortedTree } h \ t \implies \text{rm } h \ t \in \text{setOf } t$
 $\langle \text{proof} \rangle$

lemma *wrm-set*: $t \neq \text{Tip} \implies \text{sortedTree } h \ t \implies$
 $\text{setOf } (\text{wrm } h \ t) = \text{setOf } t - \{\text{rm } h \ t\}$
 $\langle \text{proof} \rangle$

lemma *wrm-set1*: $t \neq \text{Tip} \implies \text{sortedTree } h \ t \implies \text{setOf } (\text{wrm } h \ t) \leq \text{setOf } t$
 $\langle \text{proof} \rangle$

lemma *wrm-sort*: $t \neq \text{Tip} \implies \text{sortedTree } h \ t \implies \text{sortedTree } h \ (\text{wrm } h \ t)$
 $\langle \text{proof} \rangle$

lemma *wrm-less-rm*:
 $t \neq \text{Tip} \implies \text{sortedTree } h \ t \implies l \in \text{setOf } (\text{wrm } h \ t) \implies$
 $h \ l < h \ (\text{rm } h \ t)$
 $\langle \text{proof} \rangle$

lemma *remove-set*:
 $\text{sortedTree } h \ t \implies \text{setOf } (\text{remove } h \ e \ t) = \text{setOf } t - \text{eqs } h \ e$
 $\langle \text{proof} \rangle$

```
lemma remove-sort: sortedTree h t  $\implies$  sortedTree h (remove h e t)
⟨proof⟩
```

We summarize the specification of remove as follows.

corollary remove-spec:

sortedTree h t \implies

sortedTree h (remove h e t) \wedge setOf (remove h e t) = setOf t - eqs h e
 ⟨proof⟩

definition test = tlookup id 4 (remove id 3 (bininsert id 4 (bininsert 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 \Rightarrow bool **where**

$$\text{valid-tmap } t \equiv \text{sortedTree fst } t$$

declare valid-tmap-def [simp]

definition mapOf :: 'a tarray \Rightarrow index \Rightarrow 'a option **where**
 — the abstraction function from trees to maps

$$\text{mapOf } t i \equiv$$

$$\begin{cases} \text{case (tlookup fst } i \text{ t) of} \\ \quad \text{None} \Rightarrow \text{None} \\ \quad \text{Some ia} \Rightarrow \text{Some (snd ia)} \end{cases}$$

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

lemma mapOf-lookup4:
  assumes v: valid-tmap t
  assumes h: mapOf t i = Some a
  shows 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 mapset-none:
  assumes valid-tmap t
  shows (mapOf t i = None) = (∀ a. (i,a) ∉ setOf t)
  ⟨proof⟩

lemma mapset-some:
  assumes valid-tmap t
  shows (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 mupdate-map:
  assumes valid-tmap t
  shows mapOf (mupdate i a t) = (mapOf t)(i |-> a)
```

```

⟨proof⟩

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

lemma mremove-map:
  assumes valid-tmap t
  shows 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( $Nd\ l\ y\ r$ ) =
  ( $\text{sorted } l \& \text{sorted } r \& (\forall x \in \text{set-of } l. x < y) \& (\forall z \in \text{set-of } r. y < z)$ )

```

14 Tree Membership

```

primrec  $memb :: nat \Rightarrow tree \Rightarrow \text{bool}$ 
where
   $memb\ e\ Tip = False$ 
|  $memb\ e\ (Nd\ t1\ x\ t2) =$ 
  ( $\text{if } e < x \text{ then } memb\ e\ t1$ 
    $\text{else if } x < e \text{ then } memb\ e\ t2$ 
    $\text{else } True$ )

```

lemma $\text{member-set}: \text{sorted } t \dashrightarrow memb\ e\ t = (e : \text{set-of } t)$
 $\langle proof \rangle$

15 Insertion operation

```

primrec  $binsert :: nat \Rightarrow tree \Rightarrow tree$ 
— Insert a node into sorted tree.
where
   $binsert\ x\ Tip = (Nd\ Tip\ x\ Tip)$ 
|  $binsert\ x\ (Nd\ t1\ y\ t2) =$  ( $\text{if } x < y \text{ then}$ 
  ( $Nd\ (binsert\ x\ t1)\ y\ t2$ )
    $\text{else}$ 
   ( $\text{if } y < x \text{ then}$ 
    ( $Nd\ t1\ y\ (binsert\ x\ t2)$ )
    $\text{else } (Nd\ t1\ y\ t2))$ )

```

theorem $\text{set-of-binsert} [\text{simp}]: \text{set-of } (binsert\ x\ t) = \text{set-of } t \cup \{x\}$
 $\langle proof \rangle$

theorem $\text{binsert-sorted}: \text{sorted } t \dashrightarrow \text{sorted } (binsert\ x\ t)$
 $\langle proof \rangle$

corollary $binsert\ x\ t ==>$
 $\text{sorted } (binsert\ x\ t) \&$
 $\text{set-of } (binsert\ x\ t) = \text{set-of } t \cup \{x\}$
 $\langle proof \rangle$

16 Remove operation

```

primrec  $rm :: tree \Rightarrow nat$  — find the rightmost element in the tree
where
   $rm(Nd\ l\ x\ r) =$  ( $\text{if } r = Tip \text{ then } x \text{ else } rm\ r$ )

```

```

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  $Nd\ (rm\ l)\ (rm\ l)\ r$ )
  lemma rm-in-set-of:  $t \sim= Tip \implies rm\ t : set-of\ t$ 
  ⟨proof⟩

  lemma set-of-rem:  $t \sim= Tip \implies set-of\ t = set-of(rem\ t)$ 
  ⟨proof⟩

  lemma [simp]: [|  $t \sim= Tip$ ; sorted  $t$  |] => sorted(rem  $t$ )
  ⟨proof⟩

  lemma sorted-rem: [|  $t \sim= Tip$ ;  $x \in set-of(rem\ t)$ ; sorted  $t$  |] =>  $x < rm\ t$ 
  ⟨proof⟩

  theorem set-of-remove [simp]: sorted  $t \implies set-of(remove\ x\ t) = set-of\ t - \{x\}$ 
  ⟨proof⟩

  theorem remove-sorted: sorted  $t \implies sorted(remove\ x\ t)$ 
  ⟨proof⟩

  corollary remove-spec: — summary specification of remove
  sorted  $t \implies$ 
  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  $t \sim= Tip \implies remrm\ t = (rem\ t, rm\ t)$ 
  ⟨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
```