

BinarySearchTree

Larry Paulson

December 12, 2009

Contents

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

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*}

types

— we require index to have an irreflexive total order ;

— apart from that, we do not rely on index being int

index = *int*

types — hash function type

'a hash = *'a* => *index*

constdefs

eqs :: *'a hash* => *'a* => *'a set*

— 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* &

(*ALL l: setOf t1. h l < h x*) &

(*ALL r: setOf t2. h x < h r*) &

sortedTree h t2)

lemma *sortLemmaL*:

sortedTree h (T t1 x t2) ==> *sortedTree h t1* $\langle proof \rangle$

lemma *sortLemmaR*:

sortedTree h (T t1 x t2) ==> *sortedTree h t2* $\langle proof \rangle$

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) --> (ALL 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>

constdefs

sorted-distinct-pred :: 'a hash => 'a => 'a => 'a Tree => bool

— 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

constdefs

memb :: 'a hash => 'a => 'a Tree => bool

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*: $\text{memb } h \ x \ t = (x : \text{setOf } t)$
 $\langle \text{proof} \rangle$

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

4 Insertion into a Tree

primrec

$\text{binsert} :: 'a \ \text{hash} \Rightarrow 'a \Rightarrow 'a \ \text{Tree} \Rightarrow 'a \ \text{Tree}$

where

$\text{binsert } h \ e \ \text{Tip} = (T \ \text{Tip} \ e \ \text{Tip})$
 $|\ \text{binsert } h \ e \ (T \ t1 \ x \ t2) = (\text{if } h \ e < h \ x \ \text{then}$
 $\quad (T \ (\text{binsert } h \ e \ t1) \ x \ t2)$
 $\quad \text{else}$
 $\quad (\text{if } h \ x < h \ e \ \text{then}$
 $\quad \quad (T \ t1 \ x \ (\text{binsert } h \ e \ t2))$
 $\quad \quad \text{else } (T \ t1 \ e \ t2)))$

A technique for proving disjointness of sets.

lemma *disjCond*: $[[\ ! \ x. [\ x:A; \ x:B \] \ ==> \ \text{False} \] \ ==> \ A \ \text{Int} \ B = \ \{\}$
 $\langle \text{proof} \rangle$

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 $t \ -->$
 $\text{setOf } (\text{binsert } h \ e \ t) = (\text{setOf } t) - (\text{eqs } h \ e) \ \text{Un } \{e\}$
 $(\text{is } ?P \ t)$
 $\langle \text{proof} \rangle$

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

lemma *binsert-sorted*: *sortedTree* h $t \ -->$ *sortedTree* h $(\text{binsert } h \ x \ t)$
 $\langle \text{proof} \rangle$

We summarize the specification of *binsert* as follows.

corollary *binsert-spec*: *sortedTree* h $t \ -->$
 $\text{sortedTree } h \ (\text{binsert } h \ x \ t) \ \&$
 $\text{setOf } (\text{binsert } h \ e \ t) = (\text{setOf } t) - (\text{eqs } h \ e) \ \text{Un } \{e\}$
 $\langle \text{proof} \rangle$

5 Removing an element from a tree

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

t)
<proof>

lemma *wrm-less-rm*:

$t \sim = \text{Tip} \ \& \ \text{sortedTree } h \ t \ \dashrightarrow$
 $(\text{ALL } l:\text{setOf } (\text{wrm } h \ t). \ h \ l < h \ (\text{rm } h \ t)) \ (\text{is } ?P \ t)$
<proof>

lemma *remove-set*: $\text{sortedTree } h \ t \ \dashrightarrow$

$\text{setOf } (\text{remove } h \ e \ t) = \text{setOf } t - \text{eqs } h \ e \ (\text{is } ?P \ t)$
<proof>

lemma *remove-sort*: $\text{sortedTree } h \ t \ \dashrightarrow$

$\text{sortedTree } h \ (\text{remove } h \ e \ t) \ (\text{is } ?P \ t)$
<proof>

We summarize the specification of `remove` as follows.

corollary *remove-spec*: $\text{sortedTree } h \ t \ \dashrightarrow$

$\text{sortedTree } h \ (\text{remove } h \ e \ t) \ \&$
 $\text{setOf } (\text{remove } h \ e \ t) = \text{setOf } t - \text{eqs } h \ e$
<proof>

definition *test* = $\text{tlookup id } 4 \ (\text{remove id } 3 \ (\text{binsert id } 4 \ (\text{binsert id } 3 \ \text{Tip})))$

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

types

$'a \ \text{tarray} = (\text{index} * 'a) \ \text{Tree}$

constdefs
valid-tmap :: 'a tarray => bool
valid-tmap t == *sortedTree fst t*

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

constdefs
mapOf :: 'a tarray => index => 'a option
— 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*: $(\text{mapOf } t \ i = \text{None}) = (\text{ALL } a. (i,a) \sim: \text{setOf } t)$
<*proof*>

lemma *assumes v: valid-tmap t*
shows *mapset-some*: $(\text{mapOf } t \ i = \text{Some } a) = ((i,a) : \text{setOf } t)$
<*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

constdefs

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

constdefs

mremove :: *index* => 'a tarray => 'a tarray
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
 $sorted\ Tip = True$
 $| sorted(Nd\ l\ y\ r) =$
 $(sorted\ l\ \&\ sorted\ r\ \&\ (ALL\ x:set-of\ l.\ x < y)\ \&\ (ALL\ z:set-of\ r.\ y < z))$

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*: $sorted\ t \dashrightarrow memb\ e\ t = (e : set-of\ t)$
 $\langle proof \rangle$

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*]: $set-of\ (binsert\ x\ t) = set-of\ t\ Un\ \{x\}$
 $\langle proof \rangle$

theorem *binsert-sorted*: $sorted\ t \dashrightarrow sorted\ (binsert\ x\ t)$
 $\langle proof \rangle$

corollary *binsert-spec*:

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

16 Remove operation

primrec

$rm :: tree \Rightarrow nat$ — find the rightmost element in the tree

where

$rm(Nd\ l\ x\ r) = (if\ r = Tip\ then\ x\ else\ rm\ r)$

primrec

$rem :: tree \Rightarrow tree$ — 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

$remove :: nat \Rightarrow tree \Rightarrow tree$ — remove a node from sorted tree

where

$remove\ x\ Tip = Tip$
 $| remove\ x\ (Nd\ l\ y\ r) =$
 $\quad (if\ x < y\ then\ Nd\ (remove\ x\ l)\ y\ r\ else$
 $\quad \quad if\ y < x\ then\ Nd\ l\ y\ (remove\ x\ r)\ else$
 $\quad \quad if\ l = Tip\ then\ r$
 $\quad \quad else\ Nd\ (rem\ l)\ (rm\ l)\ r)$

lemma $rm-in-set-of$: $t \sim = Tip \implies rm\ t : set-of\ t$

$\langle proof \rangle$

lemma $set-of-rem$: $t \sim = Tip \implies set-of\ t = set-of\ (rem\ t)\ Un\ \{rm\ t\}$

$\langle proof \rangle$

lemma $[simp]$: $[| t \sim = Tip; sorted\ t |] \implies sorted\ (rem\ t)$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

corollary $remove-spec$: — summary specification of remove

$sorted\ t \implies$

$sorted\ (remove\ x\ t) \ \&$

$set-of\ (remove\ x\ t) = set-of\ t - \{x\}$

$\langle proof \rangle$

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 $t \sim = \text{Tip} \implies \text{remrm } t = (\text{rem } t, \text{rm } t)$

<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