BinarySearchTree

Larry Paulson

June 18, 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* **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 & (tlookup h k t = None) --> (∀ x∈setOf t. h x ~= k)
by (*induct t*, *auto*)

lemma *tlookup-some*:
    sortedTree h t & (tlookup h k t = Some x) --> x:setOf t & h x = k
apply (*induct t*)
  — Just auto will do it, but very slowly
apply (*simp*)
apply (*clarify*, *auto*)
apply (*simp-all split*: *if-split-asm*)
done

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*
by *auto*

sorted-distinct-pred holds for out trees:

lemma *sorted-distinct*: *sorted-distinct-pred h a b t* (**is** *?P t*)
proof (*induct t*)
  show *?P Tip* **by** *simp*
  fix *t1* :: *'a Tree* **assume** *h1*: *?P t1*
  fix *t2* :: *'a Tree* **assume** *h2*: *?P t2*
  fix *x* :: *'a*
  show *?P (T t1 x t2)*
  proof (*unfold sorted-distinct-pred-def*, *safe*)
    assume *s*: *sortedTree h (T t1 x t2)*
    assume *adef*: *a : setOf (T t1 x t2)*
    assume *bdef*: *b : setOf (T t1 x t2)*

```
assume hahb: h a = h b
from s have s1: sortedTree h t1 by auto
from s have s2: sortedTree h t2 by auto
show a = b
— We consider 9 cases for the position of a and b are in the tree
proof –
— three cases for a
from adef have a : setOf t1 | a = x | a : setOf t2 by auto
moreover { assume adef1: a : setOf t1
  have ?thesis
  proof –
  — three cases for b
  from bdef have b : setOf t1 | b = x | b : setOf t2 by auto
  moreover { assume bdef1: b : setOf t1
    from s1 adef1 bdef1 hahb h1 have ?thesis by simp }
  moreover { assume bdef1: b = x
    from adef1 bdef1 s have h a < h b by auto
    from this hahb have ?thesis by simp }
  moreover { assume bdef1: b : setOf t2
    from adef1 s have o1: h a < h x by auto
    from bdef1 s have o2: h x < h b by auto
    from o1 o2 have h a < h b by simp
    from this hahb have ?thesis by simp } — case impossible
  ultimately show ?thesis by blast
  qed
}
moreover { assume adef1: a = x
  have ?thesis
  proof –
  — three cases for b
  from bdef have b : setOf t1 | b = x | b : setOf t2 by auto
  moreover { assume bdef1: b : setOf t1
    from this s have h b < h x by auto
    from this adef1 have h b < h a by auto
    from hahb this have ?thesis by simp } — case impossible
  moreover { assume bdef1: b = x
    from adef1 bdef1 have ?thesis by simp }
  moreover { assume bdef1: b : setOf t2
    from this s have h x < h b by auto
    from this adef1 have h a < h b by simp
    from hahb this have ?thesis by simp } — case impossible
  ultimately show ?thesis by blast
  qed
}
moreover { assume adef1: a : setOf t2
  have ?thesis
  proof –
  — three cases for b
  from bdef have b : setOf t1 | b = x | b : setOf t2 by auto
```

```
    moreover { assume bdef1: b : setOf t1
      from bdef1 s have o1: h b < h x by auto
      from adef1 s have o2: h x < h a by auto
      from o1 o2 have h b < h a by simp
      from this hahb have ?thesis by simp } — case impossible
    moreover { assume bdef1: b = x
      from adef1 bdef1 s have h b < h a by auto
      from this hahb have ?thesis by simp } — case impossible
    moreover { assume bdef1: b : setOf t2
      from s2 adef1 bdef1 hahb h2 have ?thesis by simp }
    ultimately show ?thesis by blast
    qed
  }
  ultimately show ?thesis by blast
  qed
 qed
qed

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 safe
 assume s: sortedTree h t
 assume yint: y : setOf t
 show tlookup h (h y) t = Some y
 proof (cases tlookup h (h y) t)
 case None note res = this
  from s res have sortedTree h t & (tlookup h (h y) t = None) by simp
  from this have o1: ∀x∈setOf t. h x ~= h y by (simp add: tlookup-none)
  from o1 yint have h y ~= h y by fastforce
  from this show ?thesis by simp
 next case (Some z) note res = this
  have ls: sortedTree h t & (tlookup h (h y) t = Some z) -->
          z:setOf t & h z = h y by (simp add: tlookup-some)
  have sd: sorted-distinct-pred h y z t
  by (insert sorted-distinct [of h y z t], simp)

  from s res ls have o1: z:setOf t & h z = h y by simp
  from s yint o1 sd have y = z by auto
  from this res show tlookup h (h y) t = Some y by simp
 qed
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 assumes s: sortedTree h t
    shows memb-spec: memb h x t = (x : setOf t)
proof (cases tlookup h (h x) t)
case None note tNone = this
  from tNone have res: memb h x t = False by (simp add: memb-def)
  from s tNone tlookup-none have o1: ∀ y∈setOf t. h y ~= h x by fastforce
  have notIn: x ~: setOf t
  proof
    assume h: x : setOf t
    from h o1 have h x ~= h x by fastforce
    from this show False by simp
  qed
  from res notIn show ?thesis by simp
next case (Some z) note tSome = this
  from s tSome tlookup-some have zin: z : setOf t by fastforce
  show ?thesis
  proof (cases x=z)
  case True note xez = this
    from tSome xez have res: memb h x t by (simp add: memb-def)
    from res zin xez show ?thesis by simp
  next case False note xnez = this
    from tSome xnez have res: ~ memb h x t by (simp add: memb-def)
    have x ~: setOf t
    proof
      assume xin: x : setOf t
      from s tSome tlookup-some have hzhx: h x = h z by fastforce
      have o1: sorted-distinct-pred h x z t
      by (insert sorted-distinct [of h x z t], simp)
      from s xin zin hzhx o1 have x = z by fastforce
      from this xnez show False by simp
    qed
    from this res show ?thesis by simp
  qed
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}
   (is ?P t)
proof (induct t)
 — base case
 show ?P Tip by (simp add: eqs-def)
 — inductition step
 fix t1 :: 'a Tree assume h1: ?P t1
 fix t2 :: 'a Tree assume h2: ?P t2
 fix x :: 'a
 show ?P (T t1 x t2)
 proof
  assume s: sortedTree h (T t1 x t2)
  from s have s1: sortedTree h t1 by (rule sortLemmaL)
  from s1 and h1 have c1: setOf (binsert h e t1) = setOf t1 − eqs h e Un {e}
by simp
  from s have s2: sortedTree h t2 by (rule sortLemmaR)
  from s2 and h2 have c2: setOf (binsert h e t2) = setOf t2 − eqs h e Un {e}
by simp
  show setOf (binsert h e (T t1 x t2)) =
     setOf (T t1 x t2) − eqs h e Un {e}
  proof (cases h e < h x)
   case True note eLess = this
    from eLess have res: binsert h e (T t1 x t2) = (T (binsert h e t1) x t2) by
simp
    show setOf (binsert h e (T t1 x t2)) =
       setOf (T t1 x t2) − eqs h e Un {e}
    proof (simp add: res eLess c1)
     show insert x (insert e (setOf t1 − eqs h e Un setOf t2)) =
        insert e (insert x (setOf t1 Un setOf t2) − eqs h e)
     proof −
      have eqsLessX: ∀ el ∈ eqs h e. h el < h x by (simp add: eqs-def eLess)
      from this have eqsDisjX: ∀ el ∈ eqs h e. h el ~= h x by fastforce
      from s have xLessT2: ∀ r ∈ setOf t2. h x < h r by auto
      have eqsLessT2: ∀ el ∈ eqs h e. ∀ r ∈ setOf t2. h el < h r
      proof safe
       fix el assume hel: el : eqs h e
       from hel eqs-def have o1: h el = h e by fastforce
       fix r assume hr: r : setOf t2
```

```
          from xLessT2 hr o1 eLess show h el < h r by auto
        qed
        from eqsLessT2 have eqsDisjT2: ∀ el ∈ eqs h e. ∀ r ∈ setOf t2. h el ~=
h r
        by fastforce
        from eqsDisjX eqsDisjT2 show ?thesis by fastforce
      qed
    qed
  next case False note eNotLess = this
  show setOf (binsert h e (T t1 x t2)) = setOf (T t1 x t2) − eqs h e Un {e}
  proof (cases h x < h e)
    case True note xLess = this
    from xLess have res: binsert h e (T t1 x t2) = (T t1 x (binsert h e t2)) by
simp
    show setOf (binsert h e (T t1 x t2)) =
          setOf (T t1 x t2) − eqs h e Un {e}
    proof (simp add: res xLess eNotLess c2)
      show insert x (insert e (setOf t1 Un (setOf t2 − eqs h e))) =
            insert e (insert x (setOf t1 Un setOf t2) − eqs h e)
      proof −
        have XLessEqs: ∀ el ∈ eqs h e. h x < h el by (simp add: eqs-def xLess)
        from this have eqsDisjX: ∀ el ∈ eqs h e. h el ~= h x by auto
        from s have t1LessX: ∀ l ∈ setOf t1. h l < h x by auto
        have T1lessEqs: ∀ el ∈ eqs h e. ∀ l ∈ setOf t1. h l < h el
        proof safe
          fix el assume hel: el : eqs h e
          fix l assume hl: l : setOf t1
          from hel eqs-def have o1: h el = h e by fastforce
          from t1LessX hl o1 xLess show h l < h el by auto
        qed
        from T1lessEqs have T1disjEqs: ∀ el ∈ eqs h e. ∀ l ∈ setOf t1. h el ~=
h l
        by fastforce
        from eqsDisjX T1lessEqs show ?thesis by auto
      qed
    qed
  next case False note xNotLess = this
    from xNotLess eNotLess have xeqe: h x = h e by simp
    from xeqe have res: binsert h e (T t1 x t2) = (T t1 e t2) by simp
    show setOf (binsert h e (T t1 x t2)) =
          setOf (T t1 x t2) − eqs h e Un {e}
    proof (simp add: res eNotLess xeqe)
      show insert e (setOf t1 Un setOf t2) =
            insert e (insert x (setOf t1 Un setOf t2) − eqs h e)
      proof −
        have insert x (setOf t1 Un setOf t2) − eqs h e =
              setOf t1 Un setOf t2
        proof −
          have x : eqs h e by (simp add: eqs-def xeqe)
```

```
          moreover have (setOf t1) Int (eqs h e) = {}
          proof (rule disjCond)
            fix w
            assume whSet: w : setOf t1
            assume whEq: w : eqs h e
            from whSet s have o1: h w < h x by simp
            from whEq eqs-def have o2: h w = h e by fastforce
            from o2 xeqe have o3: ~ h w < h x by simp
            from o1 o3 show False by contradiction
          qed
          moreover have (setOf t2) Int (eqs h e) = {}
          proof (rule disjCond)
            fix w
            assume whSet: w : setOf t2
            assume whEq: w : eqs h e
            from whSet s have o1: h x < h w by simp
            from whEq eqs-def have o2: h w = h e by fastforce
            from o2 xeqe have o3: ~ h x < h w by simp
            from o1 o3 show False by contradiction
          qed
          ultimately show ?thesis by auto
        qed
        from this show ?thesis by simp
      qed
    qed
   qed
  qed
 qed
qed
```

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) &
              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 ~= Tip --> wrmrm h t = (wrm h t, rm h t)
apply (induct-tac t)
apply simp-all
done

lemma rm-set: t ~= Tip & sortedTree h t --> rm h t : setOf t
apply (induct-tac t)
apply simp-all
done

lemma wrm-set: t ~= Tip & sortedTree h t -->
               setOf (wrm h t) = setOf t - {rm h t} (is ?P t)
proof (induct t)
  show ?P Tip by simp
  fix t1 :: 'a Tree assume h1: ?P t1
  fix t2 :: 'a Tree assume h2: ?P t2
  fix x :: 'a
```

```
  show ?P (T t1 x t2)
  proof (rule impI, erule conjE)
    assume s: sortedTree h (T t1 x t2)
    show setOf (wrm h (T t1 x t2)) =
          setOf (T t1 x t2) − {rm h (T t1 x t2)}
    proof (cases t2 = Tip)
    case True note t2tip = this
      from t2tip have rm-res: rm h (T t1 x t2) = x by simp
      from t2tip have wrm-res: wrm h (T t1 x t2) = t1 by simp
      from s have x ~: setOf t1 by auto
      from this rm-res wrm-res t2tip show ?thesis by simp
    next case False note t2nTip = this
      from t2nTip have rm-res: rm h (T t1 x t2) = rm h t2 by simp
      from t2nTip have wrm-res: wrm h (T t1 x t2) = T t1 x (wrm h t2) by simp
      from s have s2: sortedTree h t2 by simp
      from h2 t2nTip s2
      have o1: setOf (wrm h t2) = setOf t2 − {rm h t2} by simp
      show ?thesis
      proof (simp add: rm-res wrm-res t2nTip h2 o1)
        show insert x (setOf t1 Un (setOf t2 − {rm h t2})) =
              insert x (setOf t1 Un setOf t2) − {rm h t2}
        proof −
          from s rm-set t2nTip have xOk: h x < h (rm h t2) by auto
          have t1Ok: ∀ l ∈ setOf t1. h l < h (rm h t2)
          proof safe
            fix l :: 'a  assume ldef: l : setOf t1
            from ldef s have lx: h l < h x by auto
            from lx xOk show h l < h (rm h t2) by auto
          qed
          from xOk t1Ok show ?thesis by auto
        qed
      qed
    qed
  qed
qed

lemma wrm-set1: t ~= Tip & sortedTree h t −−> setOf (wrm h t) <= setOf t
by (auto simp add: wrm-set)

lemma wrm-sort: t ~= Tip & sortedTree h t −−> sortedTree h (wrm h t) (is ?P
t)
proof (induct t)
  show ?P Tip by simp
  fix t1 :: 'a Tree assume h1: ?P t1
  fix t2 :: 'a Tree assume h2: ?P t2
  fix x :: 'a
  show ?P (T t1 x t2)
  proof safe
    assume s: sortedTree h (T t1 x t2)
```

```
    show sortedTree h (wrm h (T t1 x t2))
    proof (cases t2 = Tip)
    case True note t2tip = this
      from t2tip have res: wrm h (T t1 x t2) = t1 by simp
      from res s show ?thesis by simp
    next case False note t2nTip = this
      from t2nTip have res: wrm h (T t1 x t2) = T t1 x (wrm h t2) by simp
      from s have s1: sortedTree h t1 by simp
      from s have s2: sortedTree h t2 by simp
      from s2 h2 t2nTip have o1: sortedTree h (wrm h t2) by simp
      from s2 t2nTip wrm-set1 have o2: setOf (wrm h t2) <= setOf t2 by auto
      from s o2 have o3: ∀ r ∈ setOf (wrm h t2). h x < h r by auto
      from s1 o1 o3 res s show sortedTree h (wrm h (T t1 x t2)) by simp
    qed
  qed
qed

lemma wrm-less-rm:
  t ~= Tip & sortedTree h t -->
   (∀ l ∈ setOf (wrm h t). h l < h (rm h t)) (is ?P t)
proof (induct t)
  show ?P Tip by simp
  fix t1 :: 'a Tree assume h1: ?P t1
  fix t2 :: 'a Tree assume h2: ?P t2
  fix x :: 'a
  show ?P (T t1 x t2)
  proof safe
    fix l :: 'a assume ldef: l : setOf (wrm h (T t1 x t2))
    assume s: sortedTree h (T t1 x t2)
    from s have s1: sortedTree h t1 by simp
    from s have s2: sortedTree h t2 by simp
    show h l < h (rm h (T t1 x t2))
    proof (cases t2 = Tip)
    case True note t2tip = this
      from t2tip have rm-res: rm h (T t1 x t2) = x by simp
      from t2tip have wrm-res: wrm h (T t1 x t2) = t1 by simp
      from ldef wrm-res have o1: l : setOf t1 by simp
      from rm-res o1 s show ?thesis by simp
    next case False note t2nTip = this
      from t2nTip have rm-res: rm h (T t1 x t2) = rm h t2 by simp
      from t2nTip have wrm-res: wrm h (T t1 x t2) = T t1 x (wrm h t2) by simp
      from ldef wrm-res
      have l-scope: l : {x} Un setOf t1 Un setOf (wrm h t2) by simp
      have hLess: h l < h (rm h t2)
      proof (cases l = x)
      case True note lx = this
        from s t2nTip rm-set s2 have o1: h x < h (rm h t2) by auto
        from lx o1 show ?thesis by simp
      next case False note lnx = this
```

```
        show ?thesis
        proof (cases l : setOf t1)
        case True note l-in-t1 = this
          from s t2nTip rm-set s2 have o1: h x < h (rm h t2) by auto
          from l-in-t1 s have o2: h l < h x by auto
          from o1 o2 show ?thesis by simp
        next case False note l-notin-t1 = this
          from l-scope lnx l-notin-t1
          have l-in-res: l : setOf (wrm h t2) by auto
          from l-in-res h2 t2nTip s2 show ?thesis by auto
        qed
      qed
      from rm-res hLess show ?thesis by simp
    qed
  qed
qed

lemma remove-set: sortedTree h t -->
  setOf (remove h e t) = setOf t - eqs h e (is ?P t)
proof (induct t)
  show ?P Tip by auto
  fix t1 :: 'a Tree assume h1: ?P t1
  fix t2 :: 'a Tree assume h2: ?P t2
  fix x :: 'a
  show ?P (T t1 x t2)
  proof
    assume s: sortedTree h (T t1 x t2)
    show setOf (remove h e (T t1 x t2)) = setOf (T t1 x t2) - eqs h e
    proof (cases h e < h x)
    case True note elx = this
      from elx have res: remove h e (T t1 x t2) = T (remove h e t1) x t2
      by simp
      from s have s1: sortedTree h t1 by simp
      from s1 h1 have o1: setOf (remove h e t1) = setOf t1 - eqs h e by simp
      show ?thesis
      proof (simp add: o1 elx)
        show insert x (setOf t1 - eqs h e Un setOf t2) =
              insert x (setOf t1 Un setOf t2) - eqs h e
        proof -
          have xOk: x ~: eqs h e
          proof
            assume h: x : eqs h e
            from h have o1: ~ (h e < h x) by (simp add: eqs-def)
            from elx o1 show False by contradiction
          qed
          have t2Ok: (setOf t2) Int (eqs h e) = {}
          proof (rule disjCond)
            fix y :: 'a
            assume y-in-t2: y : setOf t2
```

```
          assume y-in-eq: y : eqs h e
          from y-in-t2 s have xly: h x < h y by auto
          from y-in-eq have eey: h y = h e by (simp add: eqs-def)
          from xly eey have nelx: ~ (h e < h x) by simp
          from nelx elx show False by contradiction
        qed
        from xOk t2Ok show ?thesis by auto
      qed
    qed
  next case False note nelx = this
    show ?thesis
    proof (cases h x < h e)
    case True note xle = this
       from xle have res: remove h e (T t1 x t2) = T t1 x (remove h e t2) by
simp
      from s have s2: sortedTree h t2 by simp
      from s2 h2 have o1: setOf (remove h e t2) = setOf t2 - eqs h e by simp
      show ?thesis
      proof (simp add: o1 xle nelx)
        show insert x (setOf t1 Un (setOf t2 - eqs h e)) =
             insert x (setOf t1 Un setOf t2) - eqs h e
        proof -
          have xOk: x ~: eqs h e
          proof
            assume h: x : eqs h e
            from h have o1: ~ (h x < h e) by (simp add: eqs-def)
            from xle o1 show False by contradiction
          qed
          have t1Ok: (setOf t1) Int (eqs h e) = {}
          proof (rule disjCond)
            fix y :: 'a
            assume y-in-t1: y : setOf t1
            assume y-in-eq: y : eqs h e
            from y-in-t1 s have ylx: h y < h x by auto
            from y-in-eq have eey: h y = h e by (simp add: eqs-def)
            from ylx eey have nxle: ~ (h x < h e) by simp
            from nxle xle show False by contradiction
          qed
          from xOk t1Ok show ?thesis by auto
        qed
      qed
    next case False note nxle = this
      from nelx nxle have ex: h e = h x by simp
      have t2Ok: (setOf t2) Int (eqs h e) = {}
      proof (rule disjCond)
        fix y :: 'a
        assume y-in-t2: y : setOf t2
        assume y-in-eq: y : eqs h e
        from y-in-t2 s have xly: h x < h y by auto
```

```
  from y-in-eq have eey: h y = h e by (simp add: eqs-def)
  from y-in-eq ex eey have nxly: ~ (h x < h y) by simp
  from nxly xly show False by contradiction
qed
show ?thesis
proof (cases t1 = Tip)
case True note t1tip = this
  from ex t1tip have res: remove h e (T t1 x t2) = t2 by simp
  show ?thesis
  proof (simp add: res t1tip ex)
    show setOf t2 = insert x (setOf t2) − eqs h e
    proof −
      from ex have x-in-eqs: x : eqs h e by (simp add: eqs-def)
      from x-in-eqs t2Ok show ?thesis by auto
   qed
  qed
next case False note t1nTip = this
  from nelx nxle ex t1nTip
  have res: remove h e (T t1 x t2) =
            T (wrm h t1) (rm h t1) t2
  by (simp add: Let-def wrmrm-decomp)
  from res show ?thesis
  proof simp
    from s have s1: sortedTree h t1 by simp
    show insert (rm h t1) (setOf (wrm h t1) Un setOf t2) =
          insert x (setOf t1 Un setOf t2) − eqs h e
    proof (simp add: t1nTip s1 rm-set wrm-set)
      show insert (rm h t1) (setOf t1 − {rm h t1} Un setOf t2) =
            insert x (setOf t1 Un setOf t2) − eqs h e
      proof −
        from t1nTip s1 rm-set
        have o1: insert (rm h t1) (setOf t1 − {rm h t1} Un setOf t2) =
                  setOf t1 Un setOf t2 by auto
        have o2: insert x (setOf t1 Un setOf t2) − eqs h e =
                  setOf t1 Un setOf t2
        proof −
         from ex have xOk: x : eqs h e by (simp add: eqs-def)
          have t1Ok: (setOf t1) Int (eqs h e) = {}
          proof (rule disjCond)
            fix y :: 'a
            assume y-in-t1: y : setOf t1
            assume y-in-eq: y : eqs h e
            from y-in-t1 s ex have o1: h y < h e by auto
            from y-in-eq have o2: ~ (h y < h e) by (simp add: eqs-def)
            from o1 o2 show False by contradiction
          qed
          from xOk t1Ok t2Ok show ?thesis by auto
        qed
        from o1 o2 show ?thesis by simp
```

```
              qed
            qed
          qed
        qed
      qed
    qed
  qed
qed

lemma remove-sort: sortedTree h t -->
                    sortedTree h (remove h e t) (is ?P t)
proof (induct t)
  show ?P Tip by auto
  fix t1 :: 'a Tree assume h1: ?P t1
  fix t2 :: 'a Tree assume h2: ?P t2
  fix x :: 'a
  show ?P (T t1 x t2)
  proof
    assume s: sortedTree h (T t1 x t2)
    from s have s1: sortedTree h t1 by simp
    from s have s2: sortedTree h t2 by simp
    from h1 s1 have sr1: sortedTree h (remove h e t1) by simp
    from h2 s2 have sr2: sortedTree h (remove h e t2) by simp
    show sortedTree h (remove h e (T t1 x t2))
    proof (cases h e < h x)
    case True note elx = this
      from elx have res: remove h e (T t1 x t2) = T (remove h e t1) x t2
      by simp
      show ?thesis
      proof (simp add: s sr1 s2 elx res)
        let ?C1 = ∀ l ∈ setOf (remove h e t1). h l < h x
        let ?C2 = ∀ r ∈ setOf t2. h x < h r
        have o1: ?C1
        proof −
          from s1 have setOf (remove h e t1) = setOf t1 − eqs h e by (simp add:
remove-set)
          from s this show ?thesis by auto
        qed
        from o1 s show ?C1 & ?C2 by auto
      qed
    next case False note nelx = this
      show ?thesis
      proof (cases h x < h e)
      case True note xle = this
         from xle have res: remove h e (T t1 x t2) = T t1 x (remove h e t2) by
simp
        show ?thesis
        proof (simp add: s s1 sr2 xle nelx res)
          let ?C1 = ∀ l ∈ setOf t1. h l < h x
```

```
          let ?C2 = ∀ r ∈ setOf (remove h e t2). h x < h r
          have o2: ?C2
          proof −
           from s2 have setOf (remove h e t2) = setOf t2 − eqs h e by (simp add:
remove-set)
            from s this show ?thesis by auto
          qed
          from o2 s show ?C1 & ?C2 by auto
        qed
      next case False note nxle = this
        from nelx nxle have ex: h e = h x by simp
        show ?thesis
        proof (cases t1 = Tip)
        case True note t1tip = this
          from ex t1tip have res: remove h e (T t1 x t2) = t2 by simp
          show ?thesis by (simp add: res t1tip ex s2)
        next case False note t1nTip = this
          from nelx nxle ex t1nTip
          have res: remove h e (T t1 x t2) =
                     T (wrm h t1) (rm h t1) t2
          by (simp add: Let-def wrmrm-decomp)
          from res show ?thesis
          proof simp
            let ?C1 = sortedTree h (wrm h t1)
            let ?C2 = ∀ l ∈ setOf (wrm h t1). h l < h (rm h t1)
            let ?C3 = ∀ r ∈ setOf t2. h (rm h t1) < h r
            let ?C4 = sortedTree h t2
            from s1 t1nTip have o1: ?C1 by (simp add: wrm-sort)
            from s1 t1nTip have o2: ?C2 by (simp add: wrm-less-rm)
            have o3: ?C3
            proof
              fix r :: 'a
              assume rt2: r : setOf t2
              from s rm-set s1 t1nTip have o1: h (rm h t1) < h x by auto
              from rt2 s have o2: h x < h r by auto
              from o1 o2 show h (rm h t1) < h r by simp
            qed
            from o1 o2 o3 s2 show ?C1 & ?C2 & ?C3 & ?C4 by simp
          qed
        qed
      qed
    qed
  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 assumes *h*: *mapOf t i* = *None*

shows *mapOf-lookup3*: *tlookup fst i t* = *None*
proof (*cases tlookup fst i t*)
case *None* **from** *this* **show** *?thesis* **by** *assumption*
next case (*Some ia*) **note** *tsome* = *this*
from *this* **have** *o1*: *tlookup fst i t* = *Some* (*fst ia*, *snd ia*) **by** *simp*
have *mapOf t i* = *Some* (*snd ia*)
by (*insert mapOf-lookup2* [*of i t fst ia snd ia*], *simp add*: *o1*)
from *this* **have** *mapOf t i* ~= *None* **by** *simp*
from *this h* **show** *?thesis* **by** *simp* — contradiction
qed

lemma assumes *v*: *valid-tmap t*
assumes *h*: *mapOf t i* = *Some a*
shows *mapOf-lookup4*: *tlookup fst i t* = *Some* (*i*,*a*)
proof (*cases tlookup fst i t*)
case *None*
from *this mapOf-lookup1* **have** *mapOf t i* = *None* **by** *auto*
from *this h* **show** *?thesis* **by** *simp* — contradiction
next case (*Some ia*) **note** *tsome* = *this*
have *tlookup-some-inst*: *sortedTree fst t* & (*tlookup fst i t* = *Some ia*) −−>
ia : *setOf t* & *fst ia* = *i* **by** (*simp add*: *tlookup-some*)
from *tlookup-some-inst tsome v* **have** *ia* : *setOf t* **by** *simp*
from *tsome* **have** *mapOf t i* = *Some* (*snd ia*) **by** (*simp add*: *mapOf-def*)
from *this h* **have** *o1*: *snd ia* = *a* **by** *simp*
from *tlookup-some-inst tsome v* **have** *o2*: *fst ia* = *i* **by** *simp*
from *o1 o2* **have** *ia* = (*i*,*a*) **by** *auto*
from *this tsome* **show** *tlookup fst i t* = *Some* (*i*, *a*) **by** *simp*
qed

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
— ==>
assume *mapNone*: *mapOf t i* = *None*
from *v mapNone mapOf-lookup3* **have** *lnone*: *tlookup fst i t* = *None* **by** *auto*
show ∀ *a*. (*i*,*a*) ∉ *setOf t*
proof
fix *a*
show (*i*,*a*) ~: *setOf t*
proof
assume *iain*: (*i*,*a*) : *setOf t*
have *tlookup-none-inst*:
sortedTree fst t & (*tlookup fst i t* = *None*) −−> (∀ *x* ∈ *setOf t*. *fst x* ~= *i*)
by (*insert tlookup-none* [*of fst t i*], *assumption*)
from *v lnone tlookup-none-inst* **have** ∀ *x* ∈ *setOf t*. *fst x* ~= *i* **by** *simp*
from *this iain* **have** *fst* (*i*,*a*) ~= *i* **by** *fastforce*

from *this* **show** *False* **by** *simp*
qed
qed
— <==
next assume *h*: ∀ *a*. (*i*,*a*) ∉ *setOf t*
show *mapOf t i* = *None*
proof (*cases mapOf t i*)
case *None* **then show** *?thesis* **.**
next case (*Some a*) **note** *mapsome* = *this*
from *v mapsome* **have** *o1*: *tlookup fst i t* = *Some* (*i*,*a*) **by** (*simp add*: *mapOf-lookup4*)

from *tlookup-some* **have** *tlookup-some-inst*:
sortedTree fst t & *tlookup fst i t* = *Some* (*i*,*a*) −−>
(*i*,*a*) : *setOf t* & *fst* (*i*,*a*) = *i*
by (*insert tlookup-some* [*of fst t i* (*i*,*a*)], *assumption*)
from *v o1 this* **have** (*i*,*a*) : *setOf t* **by** *simp*
from *this h* **show** *?thesis* **by** *auto* — contradiction
qed
qed

lemma assumes *v*: *valid-tmap t*
shows *mapset-some*: (*mapOf t i* = *Some a*) = ((*i*,*a*) : *setOf t*)
proof
— ==>
assume *mapsome*: *mapOf t i* = *Some a*
from *v mapsome* **have** *o1*: *tlookup fst i t* = *Some* (*i*,*a*) **by** (*simp add*: *mapOf-lookup4*)
from *tlookup-some* **have** *tlookup-some-inst*:
sortedTree fst t & *tlookup fst i t* = *Some* (*i*,*a*) −−>
(*i*,*a*) : *setOf t* & *fst* (*i*,*a*) = *i*
by (*insert tlookup-some* [*of fst t i* (*i*,*a*)], *assumption*)
from *v o1 this* **show** (*i*,*a*) : *setOf t* **by** *simp*
— <==
next assume *iain*: (*i*,*a*) : *setOf t*
from *v iain tlookup-finds* **have** *tlookup fst* (*fst* (*i*,*a*)) *t* = *Some* (*i*,*a*) **by** *fastforce*
from *this* **have** *tlookup fst i t* = *Some* (*i*,*a*) **by** *simp*
from *this* **show** *mapOf t i* = *Some a* **by** (*simp add*: *mapOf-def*)
qed

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 assumes v: valid-tmap t
    shows mupdate-map: mapOf (mupdate i a t) = (mapOf t)(i |-> a)
proof
  fix i2
  let ?tr = binsert fst (i,a) t
  have upres: mupdate i a t = ?tr by (simp add: mupdate-def)
  from v binsert-set
  have setSpec: setOf ?tr = setOf t - eqs fst (i,a) Un {(i,a)} by fastforce
  from v binsert-sorted have vr: valid-tmap ?tr by fastforce
  show mapOf (mupdate i a t) i2 = ((mapOf t)(i |-> a)) i2
  proof (cases i = i2)
  case True note i2ei = this
    from i2ei have rhs-res: ((mapOf t)(i |-> a)) i2 = Some a by simp
    have lhs-res: mapOf (mupdate i a t) i = Some a
    proof -
      have will-find: tlookup fst i ?tr = Some (i,a)
      proof -
        from setSpec have kvin: (i,a) : setOf ?tr by simp
        have binsert-sorted-inst: sortedTree fst t -->
                         sortedTree fst ?tr
        by (insert binsert-sorted [of fst t (i,a)], assumption)
        from v binsert-sorted-inst have rs: sortedTree fst ?tr by simp
        have tlookup-finds-inst: sortedTree fst ?tr & (i,a) : setOf ?tr -->
                         tlookup fst i ?tr = Some (i,a)
        by (insert tlookup-finds [of fst ?tr (i,a)], simp)
        from rs kvin tlookup-finds-inst show ?thesis by simp
      qed
      from upres will-find show ?thesis by (simp add: mapOf-def)
    qed
    from lhs-res rhs-res i2ei show ?thesis by simp
  next case False note i2nei = this
    from i2nei have rhs-res: ((mapOf t)(i |-> a)) i2 = mapOf t i2 by auto
    have lhs-res: mapOf (mupdate i a t) i2 = mapOf t i2
    proof (cases mapOf t i2)
    case None from this have mapNone: mapOf t i2 = None by simp
      from v mapNone mapset-none have i2nin: ∀ a. (i2,a) ∉ setOf t by fastforce
      have noneIn: ∀ b. (i2,b) ∉ setOf ?tr
      proof
        fix b
        from v binsert-set
        have setOf ?tr = setOf t - eqs fst (i,a) Un {(i,a)}
        by fastforce
        from this i2nei i2nin show (i2,b) ~: setOf ?tr by fastforce
      qed
```

```
      have mapset-none-inst:
      valid-tmap ?tr --> (mapOf ?tr i2 = None) = (∀ a. (i2, a) ∉ setOf ?tr)
      by (insert mapset-none [of ?tr i2], simp)
      from vr noneIn mapset-none-inst have mapOf ?tr i2 = None by fastforce
      from this upres mapNone show ?thesis by simp
    next case (Some z) from this have mapSome: mapOf t i2 = Some z by simp
      from v mapSome mapset-some have (i2,z) : setOf t by fastforce
      from this setSpec i2nei have (i2,z) : setOf ?tr by (simp add: eqs-def)
      from this vr mapset-some have mapOf ?tr i2 = Some z by fastforce
      from this upres mapSome show ?thesis by simp
    qed
    from lhs-res rhs-res show ?thesis by simp
  qed
qed

lemma assumes v: valid-tmap t
      shows mupdate-valid: valid-tmap (mupdate i a t)
proof −
  let ?tr = binsert fst (i,a) t
  have upres: mupdate i a t = ?tr by (simp add: mupdate-def)
  from v binsert-sorted have vr: valid-tmap ?tr by fastforce
  from vr upres show ?thesis by simp
qed
```

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 (simp add: mremove-def)
  from v remove-sort
  show sortedTree fst (remove fst (i, undefined) t) by fastforce
qed

lemma assumes v: valid-tmap t
      shows mremove-map: mapOf (mremove i t) i = None
proof (simp add: mremove-def)
  let ?tr = remove fst (i, undefined) t
  show mapOf ?tr i = None
  proof −
    from v remove-spec
    have remSet: setOf ?tr = setOf t − eqs fst (i, undefined)
    by fastforce
    have noneIn: ∀ a. (i,a) ∉ setOf ?tr
    proof
      fix a
      from remSet show (i,a) ~: setOf ?tr by (simp add: eqs-def)
```

qed
from *v remove-sort* **have** *vr*: *valid-tmap ?tr* **by** *fastforce*
have *mapset-none-inst*: *valid-tmap ?tr ==>*
(*mapOf ?tr i = None*) = (∀ *a*. (*i*,*a*) ∉ *setOf ?tr*)
by (*insert mapset-none* [*of ?tr i*], *simp*)
from *vr this* **have** (*mapOf ?tr i = None*) = (∀ *a*. (*i*,*a*) ∉ *setOf ?tr*) **by** *fastforce*
from *this noneIn* **show** *mapOf ?tr i = None* **by** *simp*
qed
qed

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)*
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