Report: Verification Project: Braun Trees

Beerend Lauwers Tim Kuipers

July 10, 2012

1 Introduction

Braun Trees are balanced binary trees whose left subtree is either the same size as its right subtree, or exactly one element larger. Chris Okasaki's paper [1] gives several algorithms on Braun trees implemented in Haskell. We attempted to convert the paper's Haskell code to Coq, capture the balance condition and prove some interesting theorems about the algorithms. The rest of this document is structured as follows: Section 2 describes how we approached the project. The original paper is divided into several parts, and section 3 explains each part in greater detail and what kinds of problems we encountered. Finally, in section 4 we reflect on how we tackled the project and what we learnt from the challenges we faced.

2 Plan Of Attack

We started early with verification, attempting to convert the Haskell code into Coq and creating a fitting data structure along with some auxiliary functions. We reasoned that we could get most of the code working on Coq without having to concern ourselves with the balance condition; we could tackle that problem later on. Most algorithms worked on the generic tree datatype we defined as well, so our gambit of leaving out the balance condition paid off. However, some functions were already requiring some more advanced Coq features that, at the time, were above our skill level. Several naive algorithms were also rejected by the Coq typechecker, so we omitted them at the time.

At the end of the seminar course, we revisited our code and discovered that several functions required an extra proof of termination, or were very hard to prove because of our reliance on the advanced features of Coq such as `Function`.

We set out to create a datatype that captured the balance condition, which in turn could help us prove theorems and perhaps allow us to write some algorithm definitions in a different way. After a few days of trying out several ideas, we decided to not attempt to directly prove the algorithm definitions themselves, but their results: if the resulting tree was a Braun tree, we would have good reason to believe that our definition was correct. A `Prop` was created that described the balance condition of Braun trees, and we proved for several definitions that they indeed generate a Braun tree.

The next section goes into greater detail of how we approached each algorithm, and what problems we encountered.

3 Algorithms

3.1 Datatype

We used the following datatype as our binary tree:

```
Inductive tree (X:Type) : Type :=
  | leaf : tree X
  | node : X → tree X → tree X → tree X.

Notation "<<x,s,t>>" := (node x s t).
Notation "<<x>>" := (node x leaf leaf).
Notation "<<>>" := (leaf).
```

3.2 Auxiliary functions

- Many of the algorithms reason about Braun trees using the term $(2*x+1)$, using the value x further on in the algorithm. Hence, we created an auxiliary function `div2_and_rest1` that calculated this x:

```
Fixpoint div2_and_rest1 n :=
  match n with
  | O ⇒ (0,false)
  | S(O) ⇒ (0,true)
  | S(S(m)) ⇒
    match div2_and_rest1 m with
    | (q,w) ⇒ (S(q),w)
    end
  end.
```

- We copied the definition of `treecons` from the paper:

```
Fixpoint treecons {X} (x:X) (tr:tree X) :=
  match tr with
  | <<>> ⇒ <<x>>
  | <<y,s,t>> ⇒ <<x,treecons y t, s>>
  end.

Notation "x_<:>_t" := (treecons x t) (at level 60, right
    associativity).
```

  We prove that `treecons` preserves the balance condition of Braun trees, but the proof is quite long, so we omit it here.

- When we began proving theorems, we defined this `Prop` as a description of what constitutes a Braun tree:

```
Inductive Balance {X:Type} : (tree X) → (nat) → Prop :=
  | eBal : Balance <<>> 0
  | eqBal : forall (x:X) (l r : tree X) (n:nat),
            Balance l n → Balance r n → Balance <<x,l,r>> (S(
                n+n))
  | gtBal : forall (x:X) (l r : tree X) (n:nat),
            Balance l (S n) → Balance r n → Balance <<x,l,r>>
                (S(S(n+n)))

Definition isBalanced {X:Type} (t:tree X) : Prop :=
  exists n, Balance t n.
```

3.3 Tree size

The naive tree size function is trivial:

```
Fixpoint naivesize {X:Type} (t:tree X) : nat :=
  match t with
  | <<>> => 0
  | <<x,l,r>> => S( naivesize l + naivesize r )
  end.
```

The optimized version is slightly more involved, as it requires an auxiliary function `diff`:

```
Fixpoint size {X} (t:tree X) : nat :=
  match t with
  | <<>> => 0
  | <<_,t1,t2>> =>
    let m := size t2
    in 1 + 2*m + diff t1 m
  end.

Fixpoint diff {X} (t:tree X) (n:nat) : nat :=
  match t,n with
  (* base cases *)
  | <<>>, 0 => 0
  | <<_>> ,0 => 1
  (* induction case(s) *)
  | <<_,t1,t2>> , S(q) =>
    match div2_and_rest1 q with
    | (k,false) => diff t1 k (* case q=2k+0, so Sq=2k+1 *)
    | (k,true) => diff t2 k (* case q=2k+1, so Sq=2k+2 *)
    end
  | _ , _ => 666 (* other alternatives shouldn occur *)
  end.
```

The `diff` function has a bit of a hack: the pattern match `<<_,t1,t2>>,O` can never occur because `diff` is only called inside `size`, which always provides it with a non-zero value. `<<>>,S n'` can never occur because of the balance condition.

Unfortunately, we did not get around to proving that `size t = naivesize t`: one problem is that, because `size` relies on the shape of Braun trees to more efficiently determine the size, one would have to prove that for all trees, the left subnode must be equal or greater in size than the right subnode, after which that information could be used to complete the proof.

3.4 Tree copy

The naive version of `copy` did not typecheck due to unclear structural recursion:

```
Fixpoint copy {X:Type} (x:X) (m:nat) :=
  match m with
  | O => <<>>
  | S(q) =>
    match div2_and_rest1 q with
    (* case q=2k+0, so Sq=2k+1 : *)
    | (k,false) => let t := copy x k in <<x,t,t>>
    (* case q=2k+1, so Sq=2k+2 : *)
    | (k,true) => <<x,copy x (k+1), copy x k>>
    end
  end.
```

This is because Coq cannot infer that k is in fact structurally smaller than m.

We made it typecheck by adding a dummy value *decrease*:

```
Fixpoint naivecopy (decrease:nat) {X:Type} (x:X) (m:nat) :=
  match (m,decrease) with
   | (O, _) => <<>>
   | (S q, O) => <<>>
   | (S q, S dec) =>
    match div2_and_rest1 q with
    (* case q=2k+0, so Sq=2k+1 : *)
    | (k,false) => let t := naivecopy dec x k in <<x,t,t>>
    (* case q=2k+1, so Sq=2k+2 : *)
    | (k,true) => <<x,naivecopy dec x (k+1), naivecopy dec x k>>
    end
  end.
```

The optimized version does not typecheck either, and was also given a dummy value:

```
Fixpoint copy2 (decrease:nat) (X:Type) (x:X) (n:nat): tree X * tree
     X :=
  match (n, decrease) with
   | (0, _) => (<<x>>,<<>>)
   | (S q, 0) => (<<>>,<<>>) (* shouldn't happen, since copy2 is
      only called by copy *)
   | (S q, S dec) =>
    (*match div2_and_rest1 q with
    | (k,b) => *)
    match copy2 dec X x (div2 q) with
    | (s,t) =>
        match odd q with
        | false => (<<x,s,t>> , <<x,t,t>>) (* case q=2k+0, so Sq=2k
           +1 : *)
        | true => (<<x,s,t>> , <<x,s,t>>) (* case q=2k+1, so Sq=2k
           +2 : *)
        end
    end
  end.

Definition copy {X} x n := snd (copy2 n X x n).
```

The fact that we don't do structural recursion on the actual parameter `n`, but on the dummy value instead, makes it hard to prove that the function returns balanced trees. Such a proof would use induction, but applying induction on the `copy2` function as it is now, will produce induction hypotheses which work on the destructed, structurally smaller `n`, which is the predecessor of `n`, instead of splitting it in half. A solution could consist of introducing a new datatype that doesn't work in the linear way which is specific to natural numbers.

3.5 Converting a list to a tree

The indexing function was simple, but because Coq requires functions to be total, we needed to provide a default to be given if an empty tree was passed:

```
Fixpoint indexWithDefault {X} (default:X) (t : tree X) (n : nat) :
    X :=
  match t with
   | <<x,s,t>> =>
```

```
    match n with
    | O ⇒ x
    | S(q) ⇒ match div2_and_rest1 q with
             | (i,false) ⇒ indexWithDefault default s i
             | (i,true)  ⇒ indexWithDefault default t i
             end
   end
 | _ ⇒ default
 end.

Notation "s_x_!_i" := (indexWithDefault x s i) (at level 60).
```

The first naive implementation of `makeArray` is near-identical to its Haskell equivalent:

```
Definition makeArray {X:Type} (xs:list X) : tree X :=
  fold_right treecons (<<>> : tree X) xs.
```

We proved that `makeArray` generates Braun trees:

```
Theorem makeArray_bal : forall {X} (l : list X),
  isBalanced (makeArray l).
Proof.
  intros.
  unfold isBalanced.
  apply ex_intro with (witness := (length l)).
  induction l; simpl; intros.
  (**) apply eBal.
  (**)
    apply (treecons_bal_size a).
    apply IHl.
  Qed.
```

The second implementation, `makeArray2`, did not typecheck in this straightforward conversion:

```
Fixpoint makeArray2 {X:Type} (xs : list X) : tree X :=
  match xs with
  | nil ⇒ <<>>
  | h :: t ⇒ let (o,e) := unravel t in <<h, makeArray2 o,
      makeArray2 e>>
  end.
```

Because we were interested in seeing if `makeArray` and `makeArray2` produced the same values, we wrote down this definition of `makeArray2`, leaving the obligations unfulfilled:

```
Fixpoint unravel {X:Type} (t : list X) : (list X * list X) :=
  match t with
  | nil ⇒ (nil, nil)
  | h :: t ⇒
    match unravel t with
    | (o,e) ⇒ (h :: e, o)
    end
  end.

Program Fixpoint makeArray2 {X:Type} (xs : list X) {measure (length
     xs)} : tree X :=
  match xs with
  | nil ⇒ <<>>
  | h :: t ⇒ let (o,e) := unravel t in <<h, makeArray2 o,
      makeArray2 e>>
```

```
  end.
Admit Obligations.
```

This made proving `makeArray xs = makeArray2 xs` very hard, so we refrained from doing so. If we succeeded in proving this theorem, however, we would have been able to prove that `makeArray2` generated Braun trees as well. Several examples suggest that the outputs for `makeArray` and `makeArray2` are the same.

The fully optimized version utilizes several auxiliary functions, of which the most important one is the `rows` function:

```
Fixpoint rows' (decrease:nat) (A : Type) (k : nat) (xs : list A)
  : list (nat * list A) :=
  match (k, decrease) with
  | (0, _) => nil (* k should always be more than 0 *)
  | (_, 0) => nil (* decreasing parameter should begin big enough,
      through wrapper *)
  | (_, S dec) =>
    match xs with
    | nil => nil
    | t =>
      (k, firstn k t) :: rows' dec A (2 * k) (skipn k t)
    end
  end.

Definition rows (A : Type) (k : nat) (xs : list A) :=
  rows' (length xs) A k xs.
```

The original `rows` function caused us a lot of grief: the original definition would not terminate given $k = 0$, and it was rejected by the Coq typechecker. We then used Coq's `Function` construct and painstakingly proved that the definition was, in fact, correct. However, this made using it in proofs impossible, because even fully applying `rows` did not result in a value. We then used the "dummy value" trick to get the above `Fixpoint`-only version.

There is also the `build` function, which combines a row from the `rows` function with the list of its subtrees:

```
Definition build {X} (p : nat * list X) (ts : list (tree X)) :=
 match p with
  (k,xs) => let (ts1,ts2) := splitAt k (ts ++ repeatleaf (length xs
      ))
             in zipWith3 node xs ts1 ts2
  end.
```

Finally, the `makeArray3_1` function is defined in the same way as in Haskell:

```
Definition makeArray3 {X} (xs:list X) :=
  hd <<>>
  ( fold_right build (<<>>::nil)
  ( rows X 1 xs ) ).
Definition makeArray3_1 {X} := compose (hd <<>>) (compose (
    fold_right build (<<>>::nil)) (rows X 1)).
```

Note that we also made a version called `makeArray3` that fully parametrizes the `rows` function to make it easier to use in proofs.

Unfortunately, we were unable to complete our proof that `makeArray xs = makeArray3 xs` holds. Again this is caused by a parameter which is a natural number, but which gets divided in the recursive call, instead of taking its predecessor. We can see the problem in the `rows` function; there we use functions like `firstn` and `skipn`[1], which don't follow the recursive structure of the natural numbers.

4 Conclusions

Looking back at our efforts, we feel that jumping into the verification project so soon may have made proving theorems and lemmas about the functions much harder, as our simple tree datatype does not record any information that we may have been able to use. Someone more experienced with Coq could create such a datatype and rewrite the functions to use the new definition. We explored this direction, but it did not help us advance our proof progress. We also noted that Coq is quite unforgiving in its demand for structural recursion, not figuring out that `div2_and_rest1` made its argument x smaller than $S(x)$. Similar problems arose for other auxiliary functions as well (such as `unravel`). All in all, this has been an interesting project, but our proficiency with Coq is far from the level of expertise required to give a fully satisfactory implementation of Braun Trees. This hampered our ability to pour our ideas for solutions into actual code. But the numerous code files filled with attempts show that it did not hamper our enthusiasm to still try.

References

[1] C. Okasaki, "Three algorithms on braun trees," *J. Funct. Program.*, vol. 7, no. 6, pp. 661–666, 1997.

[1] Otherwise, the functionality of both can be achieved at once with `splitAt`.