

Proving the correctness of an implementation of ordered binary trees and functions operating upon them

Jeremiah Penery

May 1, 2008

Contents

1 Overview

The ordered binary tree (hereafter called ‘OBT’) is an extraordinarily important and ubiquitous data structure. Many fundamental problems of computer science can be represented with the OBT, including problems of searching, sorting, and storing data that underlie everything else. Besides their relative simplicity, the most attractive quality of ordered binary trees is speed. The defining property of the OBT is the ordering. Operating on non-ordered trees is no better than operating on unordered linked-lists, and every advantage of using OBTs would be completely negated. Maintaining the tree’s ordered property under various operations yields an average logarithmic-time performance, instead of the linear performance we would get otherwise. Verifying an implementation of the functions on an OBT allows us to have a fast, powerful data structure and gives us confidence in the safety of our data.

Because verification in PVS can be a lengthy process, it is critical to define and test the algorithms we want to prove beforehand. Proving a faulty algorithm is impossible and a waste of time, but trying to prove a correct but poorly formulated algorithm may be even worse. To streamline this effort, I modeled and tested this implementation first in Java to ensure an absence of major bugs. Then I translated the algorithms to PVS, parameterized them, and generated some lemmas that should hold true for any implementation. Some lemmas are trivial to prove on their own, most require intermediate steps, and a few are axiomatized.

I will first present an outline of my implementation and a brief discussion of techniques, followed by a more technical presentation.

2 Implementation

Presented here is an implementation of the OBT type that fulfills the stated goals. It uses a recursive structure parameterized over types (T) with a strict total ordering on $<$.

```
obt[T: TYPE, <: (strict_total_order?[T])]: THEORY
```

`strict_total_order?` is defined in the PVS prelude file, and breaks down to the following: `irreflexive?`, `transitive?`, and `trichotomous?`. `irreflexive?` states that no element can be smaller than itself; `transitive?` states that `<` is a transitive function; and `trichotomous?` states that of two elements, one is either equal to, larger than, or smaller than the other. If any two elements satisfy more than one of the trichotomous relationships, it can be shown through transitivity that irreflexivity is violated. Thus, `strict_total_order?` guarantees that each element of the tree is unique.

An OBT is represented either by the empty tree or a node containing a value (of type T) and a left and right subtree. The ordering of trees is determined by the following `ordered?` predicate, which parameterizes all other functions and lemmas.

```
ordered?(TR): RECURSIVE bool =
  CASES TR OF
    empty: TRUE,
    node(x, TR1, TR2):
      every((λ y: y < x), TR1) ∧
      every((λ y: x < y), TR2) ∧ ordered?(TR1) ∧ ordered?(TR2)
  ENDCASES
  MEASURE size(TR)
```

This says that every element in the left subtree is strictly less than the root value, every element in the right-subtree is strictly greater than the root value, and both the left and right subtrees are themselves ordered. Every function is defined recursively in a similar manner, to more naturally operate on the recursive structure of the tree, and to facilitate the use of induction for proofs. Tree size is the measure for all induction on trees.

The `find` and `insert` operations are relatively trivial. `find` takes an element and a tree, and returns a boolean that signifies whether the element is present in the tree. It recurs down the tree based on the comparison between the given element and the root value of the tree; if the two values match it returns true, and false is returned if the empty tree is reached.

```
find(x: T, TR: (ordered?): RECURSIVE bool =
  CASES TR OF
    empty: FALSE,
    node(y, TR1, TR2):
      (x = y) ∨ IF (x < y) THEN find(x, TR1) ELSE find(x, TR2) ENDIF
  ENDCASES
  MEASURE size(TR)
```

`insert` is much the same, except that it returns an ordered tree. Where `find` would return false, `insert` returns a new tree containing the given value and two empty subtrees. If `find` would return true, because the element already exists in the list, `insert` simply returns the original tree, due to the strict ordering restriction which prevents duplicate elements from existing.

```

insert( $x: T$ , TR: (ordered?)): RECURSIVE binary_tree[ $T$ ] =
CASES TR OF
  empty: node( $x$ , empty, empty),
  node( $y$ , TR1, TR2):
    COND ( $x = y$ )  $\rightarrow$  TR,
      ( $x < y$ )  $\rightarrow$  node( $y$ , insert( $x$ , TR1), TR2),
      ( $y < x$ )  $\rightarrow$  node( $y$ , TR1, insert( $x$ , TR2))
    ENDCOND
  ENDCASES
MEASURE size(TR)

```

To rigorously test that the `insert` function preserves tree ordering, a `build_tree` function is defined, which takes a list and recursively adds the list elements to the tree. The lemma `ordered_insert` proves that building a tree from an arbitrary list results in an ordered tree.

`delete` is theoretically similar to `find` and `insert`, but it must do more work to preserve the tree ordering. When it finds the desired element to remove, there are three choices:

1. Both subtrees are empty \rightarrow Return an empty tree.
2. One subtree is empty \rightarrow Return the non-empty subtree.
3. Neither subtree is empty \rightarrow Find the largest element of the left subtree, delete it, and place it here.

This works because of the ordering - i.e., the largest element of the left subtree is guaranteed to be smaller than everything in the right subtree, and similarly larger than every other element in the left subtree. Thus, placing it at the root does indeed preserve the ordered property. Because `delete` requires us to find the largest element of a non-empty subtree, two more recursive procedures are defined, but not given here. `largest` and `smallest` find the appropriate elements from a given tree. It is important to remember that these only operate on non-empty trees, because we can't return an element from an empty tree.

```

delete( $x: T$ , TR: (ordered?)): RECURSIVE binary_tree[ $T$ ] =
CASES TR OF
  empty: empty,
  node( $y$ , TR1, TR2):
    COND ( $x = y$ )  $\rightarrow$ 
      COND (empty?(TR1))  $\rightarrow$  TR2,
        (empty?(TR2))  $\rightarrow$  TR1,
        ELSE  $\rightarrow$  LET  $z =$  largest(TR1) IN
          node( $z$ , del_large(TR1), TR2)
      ENDCOND,
    ( $x < y$ )  $\rightarrow$  node( $y$ , delete( $x$ , TR1), TR2),
    ( $y < x$ )  $\rightarrow$  node( $y$ , TR1, delete( $x$ , TR2))
  ENDCOND
  ENDCASES
MEASURE size(TR)

```

Where there is a call to the function `del_large`, there was originally a call to `delete` itself. `del_large(TR)` is equivalent to `delete(largest(TR), TR)`. And though it's intuitively obvious that `delete` should function properly there, getting PVS to agree was a monumental challenge. With the two-argument `delete` present in the induction, it was nearly impossible to apply the induction hypothesis due to differences in the second argument. The proof is almost certainly possible, but replacing the inner call to `delete` with `del_large`, which takes only a tree as its argument, generates simpler proof trees, and allows for more obvious application of the induction hypothesis. For the sake of completeness, the equivalence between `delete` and `del_large` is subsequently proven.

Unfortunately, I didn't realize the cause of the difficulty with `delete` until I'd put in a great deal of time trying to prove the original function. Once this change was made, the proof was still tedious, but not extraordinarily difficult. This demonstrates the benefits of simplification and subdivision in the implementation, to ease the process of verification. This is a concept that came up repeatedly in the course of proving the lemmas necessary to verify the implementation of OBTs. Simplification of functions and simplification or reformulation of lemmas are equally important parts of the process. Poorly formulated lemmas lead to meaningless, unprovable states just as easily as poorly formulated functions.

3 Technical Details

Proving the correctness of even these relatively simple functions is a decidedly non-trivial task. Several lemmas were generated and proven, only to be found useless; many lemmas were generated and then discarded; and still more lemmas had to be generated and proven simply as a means by which even more lemmas could be proven. However, through a systematic approach, the problem can be conquered.

Though my first attempts to prove this theory were moderately successful, there were ultimately things that were unprovable, due to the use of unparameterized lemmas. Since each function relies on ordered input trees, passing in potentially unordered trees leads to unspecified behavior, which can't be reasoned about logically. This approach also generates numerous extra subgoals with singleton trees, obscuring the essence of the theory. Fully parameterizing every lemma in the theory solves most of the problems, but comes with some minor problems of its own.

The first set of lemmas proven were the ones dealing with `insert`. To verify that insertion is correct, it suffices to show that it preserves ordering, and that an inserted element can be found in the tree. Ancillary to this is a sanity check condition, asserting that a tree is never empty after having an element inserted into it. Though it isn't used at all in the final proof, the last condition was invoked often in the initial attempts to prove the correctness of `delete`.

```
ordered_insert: THEOREM
  ∀ (TR: (ordered?)): ordered?(insert(x, TR))
```

`find_prop1`: PROPOSITION
 $\forall (\text{TR}: (\text{ordered?})): \text{find}(x, \text{insert}(x, \text{TR})) = \text{TRUE}$

`ordered_insert` used an intermediate lemma, `ordered?_insert_step`, which PVS can prove automatically via *induct-and-simplify*, in its proof to simplify the application of the induction hypothesis. Furthermore, the restriction to ordered trees has caused a large number of easily dispatched proof obligations related to the ordering of subtrees to appear in the induction case. One case requires the application of the `trichotomous?` property as well. Throughout the proof of the OBT theory, many extra proof goals like this appeared, both near the base and near the leaves of the proof tree. Often, such goals are unavoidable, but scores of unnecessary and avoidable goals are easily generated through indiscriminate use of commands like *ground* or by application or instantiation of an incorrect formula.

Having proven correct operation of `insert` and `find`, all that remains is `delete`. Verifying the operation of delete required many reformulations of the basic lemmas involved, a simplification to the delete function, and the addition of many intermediate lemmas. The specific lemmas at the ultimate goal specify that an item deleted from a tree can not be found, `delete` preserves the ordered property of the tree, and lastly, if one item is in the tree and a different item is deleted, the original item is still in the tree (i.e., the only item removed from the tree by `delete` is the desired one).

`find_prop2`: PROPOSITION
 $\forall (\text{TR}: (\text{ordered?})): \neg \text{find}(x, \text{delete}(x, \text{TR}))$

`ord_del`: PROPOSITION
 $\forall (\text{TR}: (\text{ordered?})): \text{ordered?}(\text{delete}(x, \text{TR}))$

`delete_safe`: PROPOSITION
 $\forall (\text{TR}: (\text{ordered?})): \text{find}(x, \text{TR}) \wedge y \neq x \Rightarrow \text{find}(x, \text{delete}(y, \text{TR}))$

Judging by the length and small depth of the proof tree, `find_prop2` seems easy to prove. But following the chain of invocation shows that no less than fifteen other lemmas were used, plus the five type predicates pertaining to `strict_total_order?` and four axiomatized lemmas. All these lemmas were slowly discovered over the course of the proof process, and represent various impasses in the proof process. For example, we come to the following illustrative sequent during the proof of `delete_safe`:

{-1}	$(X < \text{largest}(\text{LT}))$
{-2}	$(X < \text{root})$
{-3}	$\text{find}(X, \text{LT})$
{-4}	$(Y = \text{root})$
{-5}	$\forall (x, y: T): \text{find}(x, \text{LT}) \wedge y \neq x \Rightarrow \text{find}(x, \text{delete}(y, \text{LT}))$
{-6}	$\forall (x, y: T): \text{find}(x, \text{RT}) \wedge y \neq x \Rightarrow \text{find}(x, \text{delete}(y, \text{RT}))$
{-7}	$\text{every}(\lambda y: y < \text{root}, \text{LT})$
{-8}	$\text{every}(\lambda y: \text{root} < y, \text{RT})$
{-9}	$\text{ordered}?(\text{LT})$
{-10}	$\text{ordered}?(\text{RT})$
<hr/>	
{1}	$(X = \text{largest}(\text{LT}))$
{2}	$\text{find}(X, \text{del_large}(\text{LT}))$
{3}	$(\text{empty}?(\text{RT}))$
{4}	$(\text{empty}?(\text{LT}))$
{5}	$Y = X$

Careful study shows us that the only way to make progress here is to demonstrate the truth of consequent formula number 2. All of the other consequent formulas are unprovable at this point, and none of the grounded antecedents is falsifiable. Unfortunately, the consequent we need to prove seems unrelated to anything else, so how can we proceed? The replacement of an inner call to `delete` in its recursion with the simpler `del_large` was discussed previously, and here is where it comes into play. Applying the lemma `equality_del` and instantiating it with the tree `LT` gives us something we can work with. Replacing `del_large(LT)` in consequent formula 2 with the equivalent `delete(largest(LT), LT)` yields something that looks identical in form to antecedent formula -5. Simple instantiation of the antecedent generates a statement that is tautologically true, and PVS easily asserts that this is the case.

Lemmas relating functions to one another are vital, as the above example shows. No less important, and far more numerous, are lemmas expressing fundamental information about relevant data types. Such lemmas may not be strictly necessary in terms of one's ability to complete a proof, but they can save extraordinary amounts of time and effort. Relatively obvious properties, such as those expressed in lemmas `large` and `small`, require tens of commands to prove, even with the help of other intermediate lemmas to streamline the process.

small: THEOREM

$\forall (\text{TR}: (\text{ordered?})):$
 $\text{empty?}(\text{TR}) \vee \text{empty?}(\text{left}(\text{TR})) \vee \text{smallest}(\text{TR}) < \text{val}(\text{TR})$

large: THEOREM

$\forall (\text{TR}: (\text{ordered?})):$
 $\text{empty?}(\text{TR}) \vee \text{empty?}(\text{right}(\text{TR})) \vee \text{val}(\text{TR}) < \text{largest}(\text{TR})$

Trivial expressions where these lemmas can be applied appear throughout the proof of the theory, and without the ability to prove or disprove those expressions immediately, the proof trees can become unmanageably complex. In the interest of making proofs as easy as possible, any trivial fact that PVS can't automatically dispatch should be turned into a lemma and proven on its own. In this proof, the parameterization of $<$ over generic type T prevents PVS from automatically reasoning about $<$ relations in every case. Therefore, it is often necessary to invoke (*typepred* "obt.<"), expand the resulting `strict_total_order?` into its components, and finally instantiate them with the desired values. The `transitive?` and `irreflexive?` properties are used so often that they are separated into their own lemmas, which we invoke directly.

There exist four important properties necessary to prove the correctness of `delete` for which a proof was not found.

`trans_lt`: AXIOM

$(x < y \wedge \text{every}((\lambda z: z < x), \text{TR})) \Rightarrow$
 $\text{every}((\lambda z: z < y), \text{TR})$

`trans_rt`: AXIOM

$(x < y \wedge \text{every}((\lambda z: y < z), \text{TR})) \Rightarrow$
 $\text{every}((\lambda z: x < z), \text{TR})$

`not_find_smaller`: AXIOM

$\forall (\text{TR}: (\text{ordered?})):$
 $\text{empty?}(\text{TR}) \vee (x < \text{smallest}(\text{TR})) \Rightarrow \neg \text{find}(x, \text{TR})$

`not_find_larger`: AXIOM

$\forall (\text{TR}: (\text{ordered?})):$
 $\text{empty?}(\text{TR}) \vee (\text{largest}(\text{TR}) < x) \Rightarrow \neg \text{find}(x, \text{TR})$

The first two axioms, `trans_lt` and `trans_rt` deal with the transitivity of $<$ mapped over a tree, while the second two are assertions that an element outside of the range of values in the tree can't be found in the tree. I deemed these properties sufficiently obvious to axiomatize them, but in order to truly present a complete proof of the OBT theory, these axioms must be proven. Once all the underlying properties are expressed as lemmas (or axioms), the theory can be completely proven.

3.1 TCCs

For each defined function, and for many lemmas, the PVS typechecker generates a set of obligations called type-correctness conditions (TCCs). Until these TCCs are discharged, the theory is not considered type correct. While a theory can still be proven without all TCCs having been proven, it is considered incomplete. Verification of TCCs is the last step in the proof process. Once it is complete, the theory is finally proven in totality.

In the OBT theory, there are two unproven TCCs. The first, related to the `build_tree` function, is inconsequential, because that function is superfluous. The second unproven TCC, generated from the `delete` function, is problematic. `delete` is the subject of many proof lemmas, so its failure to typecheck correctly causes many lemmas to be considered proven but incomplete.

```
delete_TCC3: OBLIGATION
  ∀ (x: T, TR: (ordered?), y: T, TR1: binary_tree[T],
     TR2: binary_tree[T]):
    (x = y) ∧ TR = node(y, TR1, TR2) ⇒
      ¬ ((empty?[T](TR1)) ∧ (empty?[T](TR2)))
```

This obligation is generated from the inner `COND` clause in `delete`, and pertains specifically to the `ELSE` subclause. In this form, it is clearly an unresolvable obligation. Given no more information, we can't possibly determine whether the two subtrees are both empty in the general case. Examination of the code that generated this TCC provides confidence that this condition is extraneous, but a slight reformulation of `delete` may be able to obviate the need for this condition altogether, or at least render it provable.

In addition, TCCs generated by the lemmas `find_prop2` and `delete_safe` were resolved by invoking `ord_del`, which itself is considered proven but incomplete; therefore, these two TCCs are also considered proven but incomplete. Resolving `delete_TCC3` would fix these issues as well.

3.2 Lemmas

Herein I give a listing of the final lemmas in the theory, with a short description for each. Lemmas are listed in the approximate order in which they were conceived. The order in which they were proven is very different. Over the course of this work, several additional lemmas were generated and discarded, and several of the current ones went through several iterations. The last two lemmas proven were `find_prop2` and `delete_safe`, which alongside `ord_del` demonstrate the important properties of the `delete` function, and largely complete the proof of the theory.

ordered_insert: Assert that insertion preserves tree ordering.

ordered?_insert_step: Assists in the induction of `ordered_insert`.

all_ordered?: Test `build_tree` for preservation of ordering.

not_empty_insert: Make sure inserting something never produces an empty tree.

find_prop1: Assert that an inserted item can be found in the tree.

find_prop2: An item deleted from a tree can't be found in the tree.

small: If the left subtree is not empty, the smallest element of a tree is $<$ the root.

ord_del: Delete preserves ordering.

ordered?_delete_step: Removing an element from a tree doesn't affect whether a predicate applies to every element.

large: If the right subtree is not empty, the largest element of a tree is $>$ the root.

small2: If a predicate applies to every element of a tree, it applies to the smallest element.

large2: If a predicate applies to every element of a tree, it applies to the largest element.

small_large: Combination of the previous two lemmas.

l_ord: If the left subtree is not empty, its largest element is $<$ the root.

r_ord: If the right subtree is not empty, its smallest element is $>$ the root.

all_implies_alleft: A predicate that applies to the entire tree applies to the entire left subtree.

all_implies_allright: A predicate that applies to the entire tree applies to the entire right subtree.

find_small: The smallest element in a tree can be found.

trans_lt: If every element of a tree is $< x$, and $x < y$, every element of the tree is also $< y$.

trans_rt: If every element of a tree is $> x$, and $x > y$, every element of the tree is also $> y$.

not_find_larger: If x is greater than the largest element of a tree, it is not in the tree.

not_find_smaller: If x is smaller than the smallest element of a tree, it is not in the tree.

small2_inv: If x is $<$ the smallest element of a tree, it is $<$ every element of the tree.

ord_del_lrg: `del_large` preserves tree ordering.

all_del_lrg: Removing the largest element from a tree doesn't affect whether a predicate applies to every element.

equality_del: Equivalence relation between `delete` and `del_large`.

largest_gt_rest: The largest element of a tree is $>$ every element remaining after removing the largest.

trans: Distillation of **transitive?** from the strict ordering property.

trans2: Another distillation of **transitive?**.

irreflex: Distillation of the **irreflexive?** property.

find_prop4: The root element can't be found in either the left or right subtrees.

delete_safe: Ensure that delete removes only the desired item.

4 Process

Because almost every lemma in the OBT theory requires ordered trees, they all have the same basic structure, and thus the steps in the proof are largely the same. The steps are as follows:

1. Induct over the tree,
2. Dispatch the base case(s) trivially,
3. Skolemize and flatten, repeatedly if necessary,
4. Repeatedly invoke other lemmas and instantiate formulas, applying automated decision procedures when applicable,
5. Dispatch any remaining ordering-related clauses (also trivial).

Proving the truthfulness of a consequent term is intuitively viewed as the natural way to prove a goal. In this theory, I've found that falsification of an antecedent term is far more common, and usually easier to accomplish. Any non-obvious expression is a candidate for lemma application, but care must be taken not to apply and instantiate formulas recklessly. Careful consideration of the antecedent and consequent expressions often leads to a much shorter solution than does the more obvious brute-force-like method. Recklessly applying commands in a mindless attempt to find something that changes the sequent is a surefire way to cause an explosion in tree size and an inability to complete the proof. In addition, one must be careful when allowing PVS to use any automated decision procedures. *inst?* can sometimes save a bit of typing, but as often as not it does not instantiate the variables 'correctly'. If improper instantiation is not caught immediately, a lot of subsequent work may be invalid.

Working toward a final proof solution in PVS can be approached in one of three general ways: forward, by formulating properties from the lowest level and working upward to the final proof-completing lemmas; backward, by formulating the final proof-completing lemmas first, and introducing lower level lemmas as needed; or mixed, by generating both low and

high-level lemmas first, and filling in the holes with intermediate lemmas. The approach I took to this proof was very much a backward one. In my attempts to prove the correctness of `delete`, I first generated the final lemma and tried to prove it directly, adding new lemmas as the need arose. Though this approach was ultimately successful, a mixed approach is almost certainly a better method. In practice, both the forward and backward strategies converge toward a mixed approach, as it is determined what PVS needs to discharge various proof goals.

With the insights gained over the course of this project, there are many improvements that could be made to this theory which might simplify the entire proof process. Chief amongst these improvements would be the parameterization of the datatype itself as ordered, rather than parameterizing each function and lemma separately. At the very least, this would save a lot of typing (and potential errors) in laying out the theory, but it should also eliminate a lot of the superfluous ordering-related goals generated by induction over an ordered tree. All of the proofs should be examined as well. Since some of the proofs were completed before low-level lemmas were in place, there is ample opportunity for simplification. Furthermore, there are times when I yielded to the temptation to become mechanical in the proof process. A few proofs are extremely long and messy as a result. Lastly, `delete` should be slightly reformulated, so that all the TCCs can be proven, and the axiomatized lemmas should be proven as well. These changes would complete the verification of the theory, and allow for it to be easily extended.

5 Future Work

A large number of tree structures exist which either derive directly from ordered binary trees or can be represented as such. Obvious choices to begin with are AVL, AA, and Red-Black trees, all of which are relatively simple extensions of the basic data type. More exotic types like threaded trees or B-trees (and its variants) might be possible. For several of these structures, `find` can remain unchanged altogether, while `insert` requires only minor changes. As is the case with the basic OBT, `delete` is always much more difficult and has many more cases to consider. Another avenue of extension is that of serializing the data into an array, or using it as an intermediate structure in order to perform sorting or searching operations for other structures. In contrast to the extensions of the OBT datatype itself, these would require a new framework to be built up around this one. The fundamental nature of the ordered binary tree in computer science provides almost endless possibilities for extension of this work.