

Analysis of dependent types in Coq through the deletion of the largest node of a binary search tree

Sneha Popley and Stephanie Weirich

August 14, 2008

Abstract

Coq reflects some significant differences when compared to imperative programming due to the function being a model of computation. The attached ability to prove code simultaneously is another manifestation of these differences. Analysis of data structures like binary search trees reveals these sharp contrasts. The extension of the lemma representing the deletion of the largest node in a binary search tree to include the inner recursive algorithm reflects the similarity in proof complexity and basic lemmas implemented in the proof. However, this exercise reveals idiosyncracies in Coq such as surjective pairing to reflect the bigger picture: the complexity of functional programming and the reasons behind it.

1 Introduction

Imperative languages have a sense of sequencing due to their focus on implicit states through modification via commands in the source language. Most of the popular languages such as Java and C/C++ are imperative languages. On the other hand, declarative languages have no implicit state and, in the case of functional languages, rely on the function as a model of computation. In declarative languages, the state is carried around explicitly. Haskell, OCaml, and Coq are some examples of declarative languages [Hud89].

The factorial program can be considered to be the "Hello World" program of functional programming, and Figure 1 demonstrates this algorithm in Java, an imperative language, and Coq, a functional language. In the Java version of this program, the `for` loop ensures `i` loops from 1 to `n` and creates a sequence. Also, the assignment statement for `result` alters its implicit state to bind a number to the variable. Other examples of sequence-creating commands are `while` loops, `goto` statements, and the conditional statement (not demonstrated in Figure 1). In contrast, Coq explicitly carries around the state of the natural number `n` [Log]. Additionally, looping is carried out through recursion. In this case, the code denotes a value (the desired factorial) rather than a sequence of commands. The declarative version of the factorial program focuses on the

Java:

```
public static int factorial ( int n ){
    int result = 1;
    for ( int i = 1; i <= n; i ++ )
        result = i * result;
    return result;
}
```

Coq:

```
Fixpoint fact (n:nat) : nat :=
  match n with
  | 0 => 1
  | S n => S n * fact n
  end.
```

Figure 1: Comparison between Java and Coq

description of what is happening rather than how a factorial can be computed. It closely resembles the mathematical definition of a factorial [Hud89].

Dependent types are types represented in terms of data. They allow us to express the important properties of the data. Functional programming can also include dependent types, and, as a result, yield proof-carrying code. Such dependently typed languages close the semantic gap between programs and their properties by enabling one to prove the validity of the code [AMM05].

Coq is a formal proof management system that integrates a functional programming language with dependent types. It provides interactive proof methods and decision algorithms that enable one to mathematically verify a proof. Coq then extracts a certified program from the constructive proof. The Coq code below proves that addition is commutative for two numbers m and n if they are equal [Log]. Hence, it proves commutivity for one of the possible cases of addition [Pie07].

```
Lemma plus_id_common : forall m n:nat,
  m = n -> plus m n = plus n m.    (* -> is pronounced "implies" *)
Proof.
  intros m n.      (* move both quantifiers into the context at once *)
  intros eq.      (* move the hypothesis into the context *)
  rewrite -> eq.  (* Rewrite the goal according to the hypothesis *)
  reflexivity.
Qed.
```

Loops are replaced by recursion in functional languages (due to its nature and reliance on Church's lambda calculus) [Hud89]. Hence, trees prove to be one of the basic and intuitive data structures implementable in functional languages.

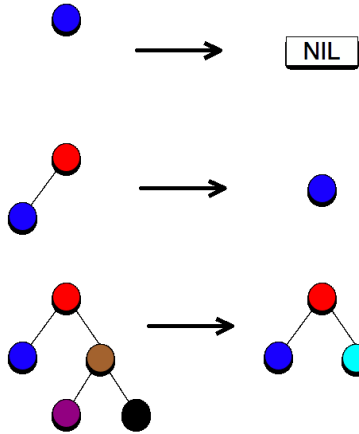


Figure 2: Cases of deletion of greatest node of binary search tree

The analysis of `rmax`, the algorithm for the deletion of the greatest node in a binary search tree, as well as the extension of its definition to improve its clarity and the comparison of the proof complexity in the original and modified `rmax`, reveals underlying idiosyncrasies in Coq.

Background information on the basic definitions, lemmas, and binary search tree properties in Coq is required to understand the correctness of `rmax`. An analysis of the Coq tactics and idiosyncrasies involved in the assertion and proof of `rmax` reflect some of the current issues in functional programming. Unless explicitly stated, examples are in Coq.

2 Review of Binary Trees

Binary trees are data structures in which each node has at most two children. Such binary trees form the foundation for the creation of binary search trees where the left subtree contains values less than the root while the right subtree contains values greater than the root. Binary search trees have wide-ranging applications in sorting and searching.

There are three cases of deletion of the greatest node in a binary search tree as shown in Figure 2:

1. Deleting a node with no children
Such a deletion returns an empty tree (NIL).
2. Deleting a node with no right child
Since the root is the greatest node in the tree, the left tree is returned as the result.

3. Deleting a node with two children

A recursive algorithm can be used to reach the greatest element in the rightmost child of the tree. The returned tree contains the root and left child, but has a modified right child.

3 Binary Trees in Coq

A binary tree of natural numbers can be defined as follows [Log]:

```
Inductive nat_tree : Set :=
| NIL -> nat_tree
| bin -> nat -> nat_tree -> nat_tree -> nat_tree.
```

`Inductive` indicates the beginning of a simple inductive definition. `nat_tree` represents the definition of a simple inductive type that lies in the universe of `Set` (the universe for program types). `NIL` and `(bin n t1 t2)` are constructors.

`nat_tree` and `nat -> nat_tree -> nat_tree -> nat_tree` are the types of the constructors in `Set`. The three arguments to `bin`, `nat`, `nat_tree`, and `nat_tree` represent the primary structure of a binary tree shown in Figure 3.

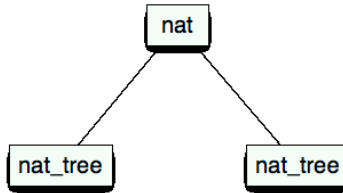


Figure 3: Structure of `nat_tree`

4 Essential Definitions

4.1 Non-empty Binary Tree

```
Inductive binp : nat_tree -> Prop :=
| binp_intro : forall (n:nat) (t1 t2 : nat_tree),
  binp (bin n t1 t2).
```

`binp` is another inductive definition that asserts that the given `nat_tree` is a non-empty binary tree. This is essential in the specification of `rmax` because deletion from an empty tree is undefined. Just as `Set` is the universe of program types, `Prop` is the universe of logical proposition. `binp_intro` can be applied in proofs to complete them as shown below [Log].

```

Lemma binp_example :
binp (bin 7 NIL NIL).
Proof.
apply binp_intro.
Qed.

```

In the above proof `binp_example`, we prove that `bin 7 NIL NIL` is a non-empty tree by a single application of `binp_intro`. `Qed` validates the proof and adds it to the programming environment [Log].

4.2 Occurrence

```

Inductive occ : nat_tree -> nat -> Prop :=
| occ_root : forall (n:nat) (t1 t2: nat_tree), occ (bin n t1 t2) n
| occ_l : forall (n p:nat) (t1 t2: nat_tree), occ t1 p ->
  occ (bin n t1 t2) p
| occ_r : forall (n p:nat) (t1 t2: nat_tree), occ t2 p ->
  occ (bin n t1 t2) p.

```

`occ` checks for the presence of a natural number (`nat`) in a tree [Log]. It has three constructors: `root`, `left`, and `right`. These constructors can be applied in proofs to prove the occurrence of a natural number in a tree. At `occ_root`, if the number is equal to the root of the tree, the number occurs in the tree. However, in the case of `occ_l` and `occ_r`, the left and right tree respectively are recursively checked for the occurrence of the number.

```

Lemma occ_example:
occ (bin 3 NIL (bin 4 NIL NIL)) 4.
Proof.
apply occ_r. apply occ_root.
Qed.

```

In `occ_example`, the structure of the tree is as shown in Figure 4. We first traverse down the right subtree and apply `occ_root` to check if the number matches the root of the tree obtained.

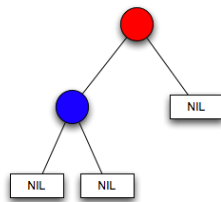


Figure 4: Structure of tree in `occ_example`

5 Essential Definitions

5.1 Binary Search Tree

```
Inductive search : nat_tree -> Prop :=
| nil_search : search NIL
| bin_search : forall (n:nat) (t1 t2:nat_tree),
search t1 -> search t2 -> maj n t1 -> min n t2 ->
search (bin n t1 t2).
```

`search` is the inductive definition of a binary search tree [Log]. `(maj n t1)` states that `n` is greater than every node in `t1` (the left child), while `(min n t2)` asserts that `n` is smaller than every node in `t2` (the right child). This definition is also recursive since it requires that the left and right children also be binary search trees.

6 RMAX

6.1 General lemma

```
Lemma rmax:
forall (t:nat_tree),
search t -> binp t ->
{q : nat & {t' : nat_tree | RMAX t t' q}}
```

This general lemma may read as:

```
if t is a binary search tree and
t is not NIL,
there exists q and t' such that RMAX t t' q.
```

In this case, `RMAX` (in Figure 5) is inductively defined to specify that `t'` is the resultant tree from the removal of the greatest node `n` of tree `t` [Log]. Since data structures are immutable in functional programming, a new tree is shown to be created when connections are changed. However, it may contain parts of `t` where connections are unaffected by deletion.

6.2 Definition of Specification

`rmax_intro` can be used to represent `RMAX` in terms of `occ` and `search` when it appears in a proof. The definition of `RMAX` can be read as shown in Figure 5.

6.3 Lemmas

Some basic lemmas are necessary to prove `rmax` in general cases. They include the three base cases in Figure 2 [Log]:

```

Inductive RMAX (t t' : nat) (n : nat) : Prop :=
rmax_intro : occ t n ->
(forall p:nat, occ t p -> p <= n) ->
(forall q:nat, occ t' q -> occ t q) ->
(forall q:nat, occ t' q -> occ t' q \ / n = q) ->
~ occ t' n -> search t' ->
RMAX t t' n.

```

Read as:

(RMAX t t' n) iff :

- n occurs in t
- n is greater than or equal to every item occurring in t
- Each item occurring in t' occurs also in t
- Each item occurring in t is equal to n, or occurs in t'
- n does not occur in t'
- t' is a search tree.

Figure 5: Specification of rmax

1. rmax_NIL_NIL

forall (n:nat), RMAX (bin n NIL NIL) (NIL) n.

In this case, (t = bin n NIL NIL) and (t' = NIL). Hence, it defines the case where t (the original tree) has only one node.

2. rmax_t_NIL

forall (t:nat.tree) (n:nat), (search bin n t NIL) -> (RMAX (bin n t NIL) t n).

When a search tree has no right child, applying rmax gives us the left child (t).

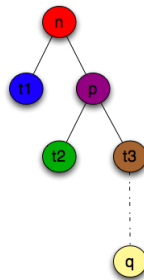


Figure 6: Structure of tree in rmax_t1_t2t3

```

simple induction t.
(* t is NIL *)
intros s b. exists 0. exists NIL.
absurd (binp NIL); auto with searchtrees v62.
(* t is bin *)
intros n t1 hr1 t2. case t2.
(* t2 is NIL *)
intros hr2 s b. exists n; exists t1.
apply rmax_t_NIL; auto.
(* t2 is bin *)
intros n' t1' t2' hr2 s b. elim hr2.
intros num ex. elim ex.
intros tr htr. exists num; exists (bin n t1 tr).
apply rmax_t1_t2t3; auto.
apply binp_intro.
apply search_r in b; auto.
Qed.

```

Figure 7: Proof of `rmax`

3. `rmax_t1_t2t3`

`RMAX (bin n t1 (bin p t2 t3)) (bin n t1 t') q.`

The structure of the original tree is shown in Figure 6 represented by `(bin n t1 (bin p t2 t3))`. The tree that results from removing `q` from the original tree is `(bin n t1 t')`. In the case of the presence of both left and right children in a search tree, the root (`n`) and left child (`t1`) remain unchanged after the application of `rmax`. Hence, `t'` is a modified version of `(bin p t2 t3)` without its greatest node `q`.

6.4 Proof

In order to prove `rmax`, we perform induction on `t`. For the case `t = NIL`, we substitute `q = 0` and `t' = NIL`. `RMAX NIL NIL 0` is an absurd claim (as the removal of a node from an empty tree is undefined), evidenced by the equally absurd claim `binp NIL`, as `binp` only allows non-empty binary trees. For `t = bin n t1 t2`, we carry out a case analysis on `t2`. Substitute `q = n` and `t' = t1` (the left child of `t`). This forms the statement `RMAX (n t1 NIL) t1 n` which corresponds to the lemma `rmax_t_NIL`. When `t2 = bin n t1' t2'`, case analysis leads to the introduction of more variables. Substitute `q = num` and `t' = bin n t1 tr`. The thus formed statement, `RMAX (bin n t1 (bin n' t1' t2')) (bin n t1 tr) num`, resembles case 3 of the three cases and coincides with `rmax_t1_t2t3`.

```

Fixpoint rmax_code (t: nat_tree) {struct t} : nat * nat_tree :=
  match t with
  | NIL => (0, NIL)
  | bin n t1 t2 =>
    match t2 with
    | NIL => (n, t1)
    | bin n' t1' t2' =>
      let max := fst (rmax_code t2) in
      let new_tree := snd (rmax_code t2) in
      (max, bin n t1 new_tree)
    end
  end.

```

Figure 8: Specification of `rmax_code`

6.5 Observation

The proof of the lemma does not completely demonstrate the structure of the algorithm. We can only infer the structure through prior knowledge of binary search trees and deletion. In an attempt to fix this deficiency, `rmax_code` is introduced into the lemma.

7 Modified rmax

7.1 New definition

```

forall (t t' : nat_tree) (q : nat),
  binp t -> search t ->
  rmax_code t = ( q, t') -> RMAX t t' q.

```

This general lemma introduces `rmax_code` into the implementation of `rmax`. It narrows the definition in Section 6.1 to the values returned by `rmax_code`, to improve the clarity of the algorithm.

7.2 Introduction to `rmax_code`

`rmax_code` (in Figure 8) uses the primitive form of definition: recursion over inductive objects. It recursively moves to the bottom of the right child of the tree to find the greatest node in a binary search tree. The definition returns the greatest node along with a new tree which is the resultant tree from the removal of the greatest node.

7.3 Proof

The structure of the proof (as shown in Figure 9) is similar to 6.4. Induction on `t` is followed by case analysis of the right child of `t` (`t2`). However, we have

```

simple induction t.
intros t' q s b a.
(* t is NIL *)
absurd (binp NIL); auto with searchtrees v62.
(* t is bin of something *)
intros n n0 hr1 t2. case t2.
(* t2 is NIL *)
intros t' t1 hr2 s b a. inversion a; subst. apply rmax_t_NIL; auto.
(* t2 is bin n' t1' t2' *)
intros n' t1' t2' hr2 s b t' q1 q2. remember (bin n' t1' t2') as t2''.
simpl in q2. subst t2''.
assert (fst (rmax_code (bin n' t1' t2'))) = b). injection q2; auto.
assert (bin n n0 (snd (rmax_code (bin n' t1' t2')))) = s). injection q2; auto.
subst s. apply rmax_t1_t2t3; auto. apply hr2. apply binp_intro.
apply search_r in q1; auto. subst b. destruct t2''; auto.
Qed.

```

Figure 9: Proof with `rmax_code`

to apply more tactics in the second case of `t2`.

In order to prove `rmax`, induction is performed on `t`. For the case `t = NIL`, we substitute `q = 0` and `t' = NIL`. `RMAX NIL NIL 0` is an absurd claim (as removal of a node from an empty tree is undefined) that can be proved by the equally absurd claim `binp NIL`, as `binp` only allows non-empty binary trees.

For `t = bin n t1 t2`, induction on `t2` is carried out. Substitute `q = n` and `t' = t1` (the left child of `t`). This forms the statement `RMAX (n t1 NIL) t1 n` which corresponds to the lemma `rmax_t_NIL`.

When `t2 = bin n t1' t2'`, further simplification leads to `RMAX (bin n n0 (bin n' t1' t2')) s b`. There is no existing lemma to account for this case, however, some substitutions can be done to substitute `s` as a `bin` (binary tree). After the application of `rmax_t1_t2t3`, applications of suppositions regarding `RMAX (bin n' t1' t2') t' q`, `binp_intro`, and `search` form the surjective pair which we can break down to complete the proof.

8 Comparison

8.1 Similarities

The structure of both proofs includes induction on `t` followed by case analysis of its right child `t2`. The basic lemmas used in both cases remain the same. Also, in both versions, the inductive definitions of `binp` and `search` are used to complete the proof.

8.2 Differences

The previous proof uses `exists` to arbitrarily assign values to `q` and `t'`. However, this ability is reduced significantly with `rmax_code` as the values of `t'` and `q` are dependent on the values passed to `rmax_code`. Hence, a slightly different set of tactics expose the absence of surjective pairing in Coq's notion of equality.

8.2.1 Surjective Pairing

Suppose `(func x)` returns a pair `(A, B)`. Then the following lemma holds:

```
func x = ( fst (func x), snd (func x) )
where fst (func x) = A and
      snd (func x) = B.
```

This lemma is not provided by Coq's notion of equality. Hence, every time surjective pairs appear in real-time proofs, one uses a lemma in the Coq library or actually carries out the proof (which is what is done in the proof of `rmax_code`). Carrying out the proof is not tremendously difficult, but the presence of this situation in the proof of `rmax_code` demonstrates the small base of ideas implemented in Coq. It also reflects the difficulties faced in proof completion due to such idiosyncrasies.

In the proof of `rmax`, surjective pairing rears its head in the final step of the proof where the assertion we are attempting to prove is

```
rmax_code (bin n' t1' t2') =
(fst(rmax_code(bin n' t1' t2')),
snd(rmax_code(bin n' t1' t2'))).
```

We solve this through application of the `destruct` tactic on `t2'` which performs a case analysis without recursion. We can also use the lemma `surjective_pairing` from the standard library by applying it to the above assertion. The lemma is given below.

```
Lemma surjective_pairing :
  forall (A B:Type) (p:A * B), p = pair (fst p) (snd p).
Proof.
  destruct p; reflexivity.
Qed.
```

9 Future Work

The lemma for the deletion of the maximum valued node from a binary search tree plays an influential role in the deletion of arbitrary elements. Future work involves analysis of this role as well the role of `rmax_code` in arbitrary deletion along with other miscellaneous binary search tree algorithms. Also, a deeper look into the reasons for the difficulties experienced in completing the proofs should give a better analysis of Coq and the fundamentals of dependently typed languages.

10 Conclusion

`rmax.code` introduces a greater level of clarity into the method of deletion while maintaining the basic structure and complexity level of the proof of `rmax`. `rmax.code` also reveals how surjective pairing is not provided for in Coq's notion of equality, but such pairing has to be handled with basic tactics or a lemma defined in the Coq library when it appears in a proof. Hence, this exercise reflects the issues at hand when faced with the ability to simultaneously prove code in dependently typed languages.

References

- [AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter, April 2005.
- [Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages, September 1989. Submitted for publication.
- [Log] LogiCal. The coq proof assistant.
- [Pie07] Benjamin Pierce. Software foundations. lecture 7., September 2007.