Putting Dependent Types to Work

A Coq Library That Enforces Correct Running-Times via Type-Checking

Jay McCarthy
Brigham Young University
jay@cs.byu.edu

Burke Fetscher
Northwestern University
burke.fetscher@eecs.northwestern.edu

Max New
Northwestern University
max.new@eecs.northwestern.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Abstract

This paper presents a Coq library that lifts an abstract yet precise notion of running-time into the type of a function and uses it to prove all nine of the algorithms in Okasaki's paper *Three Algorithms on Braun Trees* have the claimed running times.

A secondary goal of this paper is to be an introduction to Coq. While Coq is never going to be as straight-forward as Logo or Scratch, it has has the capacity to write amazingly rich specifications of functions and, once you have done so, the corresponding proofs can be only as complex as they need to be in order to actually establish the specified properties. This paper assumes only that its readers are familiar with ordinary, tree-processing functions and that they are willing to dig into very precise specifications of such functions. No familiarity with Coq itself is assumed.

1. Warming up to Dependent Typing in Coq

This section works through an example with the goal of bringing across just enough Coq to be able to read the code fragments in the rest of the paper.

One way to think about Coq's dependent type system is to start with a type system much like that of ML or OCaml, and then layer in the ability for types to refer not just to types in the surrounding context, but also to ordinary program values. This ability brings great power of specification (the conventional example being that array operations' types can insist on in-bound access), but at the cost of making the type checking problem much more complex than it is in non-dependently typed programming languages.

http://dx.doi.org/10.1145/ **Draft as of Tuesday, March 25th, 2014**

1.1 A First Dependently-Typed Function: `drop`

To get started, consider the definition of a `drop` function. It accepts a natural number `n` and a list and it returns the list, but without its first `n` elements.

```
Program Fixpoint drop (n : nat)
                      (len : nat)
                      (l : list_with_len len)
: list_with_len (if (le_dec n len)
                 then (len - n)
                 else 0)
  := if zerop n
     then l
     else match l with
            | empty => empty
            | (cons _ hd tl) =>
              drop (n-1) (len-1) tl
          end.
```

Ignore the types and the `len` argument for a moment (`len` is there only in support of the types). The other two arguments are `n`, the number of elements to drop, and `l`, the list to drop them from. The body of the function starts just after the `:=` (the rest of the unexplained parts of the header are types) and it compares `n` to `0`. If they are equal, the result of the function is just `l`. If they aren't equal, then the `match` expression destructures the list. If the list is empty, then `drop` is being asked to drop more elements than there are in the list, so we just return the empty list. If the list isn't empty, then we recur with the tail of the list and `n-1`.

In a language like ML that supported a natural number type `nat`, the type of `drop` would be something like

```
drop : nat -> list 'a -> list 'a.
```

Using Coq's dependent types, however, we can establish a relationship between the length of the input list, the length of the output list, and the number n. This is what the type annotations do. Specifically, the type of the l argument is list_with_len len, promising that l's length is len. Using dependency we can then promise that the result list's length is the if expression:

```
    if (le_dec n len) then (len - n) else 0
```

or, in English, that drop returns a list whose length is len - n, as long as n is less than or equal to len, and 0 otherwise.

Which brings us to the remaining argument: len. It is there just to be able to state the type of l — without it, the reference to len in the type of l would be a free variable, which is not allowed. This rationale is unsatisfactory, however, and it is arguments like this that the work in this paper aims to avoid.

1.2 Type Checking in Coq: Some Assembly Required

Before we can try to remove that argument, however, we still have the problem of how to actually check the types for drop. As you might expect, being able to mix values and types like this can lead to thorny type-checking problems where even decidability seems unattainable. And, indeed, Coq is not able to type-check this program without help. Instead it establishes some basic properties of the function (e.g. that drop is called with the right number of arguments) and then drops you into a theorem proving system with particular theorems to be proven in order to finish type checking.

In this case, there are four. The simplest one is the one for the empty case in the match expression:

```
forall len n,
  len = 0 ->
  0 = (if (le_dec n len)
       then (len - n)
       else 0)
```

This corresponds to proving that the length of the result list (0 since it is empty) is equal to the declared length in drop's header, where we get to assume that len is equal to 0. The assumption comes from the match expression: we are in the empty case so that means l is empty, which means that l's length is 0. To prove this, consider the two possible outcomes of the if expression. The else case is immediate, and the then case follows from the test of the conditional. The test says that n is less than or equal to 0, so that means that n must be exactly 0 (since we're using the natural number type, not integers) and since we know that len is 0, this case simplifies to just 0 = 0 - 0.

The other three correspond to the then branch of the if in the body of drop, that the arguments to the recursive call match up, and the result of the cons case at the end of drop. The most complex of these is the last one. Here's what Coq asks you to prove, and it holds by very similar reasoning to the case above, just with a few more situations to consider:

```
forall tl len n,
  (length tl) + 1 = len ->
  (if le_dec (n - 1) (len - 1)
   then len - 1 - (n - 1)
   else 0)
  =
  (if le_dec n ((length tl) + 1)
   then (length tl) + 1 - n
   else 0)
```

1.3 Extraction: Too Much Information

Coq supports a way to extract programs with these rich types into OCaml in order to efficiently run programs that have been proven correct. Unfortunately, when applied to our drop function, we get a function that still accepts the len argument, even though it is not actually used to compute the result. Indeed, the extracted code never does anything with that value; it just pipes it around in the function and lets it pollute drop's interface.

Even worse, however, is that these extra arguments are not just in drop, but they are built into the list data structure too, for very similar reasons. Here's the definition of list_with_len:

```
Inductive list_with_len : nat -> Type :=
| empty : list_with_len 0
| cons (tl_len : nat)
       (hd : A)
       (tl : list_with_len tl_len)
  : (list_with_len (tl_len+1)).
```

The keyword Inductive is Coq's version of ML's datatype declaration and the corresponding ML datatype (but without the list length information) is

```
datatype 'a list =
  empty
| cons of 'a * 'a list
```

In keeping with Coq's dependency, an Inductive can be parameterized over more than just types. In this case, the first line declares that it is parameterized over a natural number, which we use to represent the length of the list. To support such a parameterization, Coq requires that each of the constructors be annotated with its return type (in ML, this annotation is not necessary as it can always be inferred from the header). So, the empty constructor is given the type list_with_len 0 to say that it constructs lists whose lengths are 0. The cons constructor similarly declares that it builds lists whose lengths are tl_len + 1. To do so, it must be supplied with the length of the tail of the list, an element for the head of the list, and the actual tail. As before, the presence of tl_len is disappointing.

Coq supports declarations that tell it which arguments should be dropped in the extracted code, but it cannot always tell when it is sound to do so. In our example, Coq can tell that the `tl_len` argument to `cons` is sound to eliminate, but not the `len` argument to `drop`.

1.4 Dependently-typed Datastructures: Proofs as Values

The conventional work-around to avoid the extra arguments in the extracted code is to define two `Inductive`s. One is a match for the extracted version of the code and thus does not include any dependency and the other describes only the lengths of the lists and thus can be used in auxiliary theorems that describe how `drop` treats lists.

Separating the theorems from the function, however, typically requires longer and more complex proofs because the proof essentially has to mimic the structure of the function. Still, the combination of dependent types and data definitions is remarkably expressive and an important part of our approach. Our library combines this power with expressive types directly on functions. For now, however, lets see how the separation plays out for `drop`.

Fundamentally, the combination of dependent types and data structure definitions gives us the power to define a datatype whose elements correspond to proofs and whose types correspond to facts. To see how this works, lets start with the definition of `list`:

```
Inductive list : Type :=
| empty : list
| cons (hd : A) (tl : list) : list.
```

It has no dependency and is a one-for-one match with the earlier ML fragment that defines lists. Accordingly, it will extract directly into the ordinary OCaml list type.

Here's the `Inductive` definition that captures the length of a list:

```
Inductive list_len : list -> nat -> Type :=
| L_empty : list_len empty 0
| L_cons (tl_len : nat)
         (hd : A)
         (tl : list)
         (tl_ok : list_len tl tl_len)
  : list_len (cons hd tl) (tl_len + 1).
```

The first line doesn't just say `list_len : Type` like the `list` declaration did. Instead, it indicates that in order to get a type, `list_len` must first be supplied with two arguments, a `list` and a `nat`. Our goal is to restrict the pairs of arguments such that `(list_len empty 0)` is a type of a tree that we can actually construct, but `(list_len empty 2)` is not.

We can control which trees we allow by controlling the types of the constructors of `list_len`. That is, with an ordinary data-type definition, the only way to make values of the given type is by combinations of constructors. The same is true in Coq, but since the constructors can be given dependent types, we can encode much more interesting restrictions. So interesting, in fact, that we can consider the elements of the datatype to be proofs and the types to be propositions, only some of which will be provable, since only some of the trees we can build will be well-typed. Put another way, we can read a judgment form definition from the declaration of `list_len`; it defines a relation on pairs of lists and natural numbers that holds only when the list has the corresponding length.

The simplest tree is just to write `L_empty`. This constructor has the type `list_len empty 0`, as it is written in the second line of the definition of `list_len`. And thus we know that the empty list is considered to have length `0`.

If we want to use `L_cons`, then that constructor is going to build us a value of type `(list_len (cons hd tl) (tl_len + 1))`, but only if we can supply well-typed arguments according to the types written as arguments to `L_cons`. And, unsurprisingly, we need to supply all of the values whose types correspond to the restrictions you'd expect, the key one being a value that tells us that `tl` has length `tl_len`. As an example, we can write

```
(L_cons 0 true empty L_Empty) :
(list_len (cons true empty) 1)
```

to demonstrate that the singleton list containing `true` has length `1`.

Once we have `list_len` defined, proving the same property of `drop` corresponds to this `Theorem` declaration in Coq:

```
Theorem drop_lengths :
  forall n len l,
         list_len l len ->
         list_len (drop n l)
                  (if (le_dec n len)
                   then (len - n)
                   else 0).
```

This version of `drop` and `list` are easily extractable, producing the expected code without any extra declarations to guide Coq.

Unfortunately, the proof that `drop` behaves properly is now significantly longer. In the version from section 1.1, there were four proofs, each of which requires only one line of Coq code. The corresponding proof for this version of the function is 20 non-trivial lines. Perhaps unsurprisingly the proof for the current version of `drop` has the previous version's proof embedded inside it. Most of the extra is boilerplate proof manipulation to mimic the case dispatching that the function itself does.[1]

2. Introducing Our Library: Braun Tree Insert

The core of our library is a monad that, as part of its types, tracks the running time of functions. To use the library, programs must be written using the usual return and bind

[1] See `l.v` and `lwl.v` at `https://github.com/rfindler/395-2013/tree/master/paper` for the fully worked examples from this section using both approaches.

monadic operations. In return, the result type of a function can use not only the argument values to give it a very precise specification, but the type also has access to an abstract step count describing how many primitive operations the function executes.

To see how this works, we start with a definition of Braun trees (Braun and Rem 1983) and the insertion function where the running time for the function is declared as part of the body of the function. In the next section, we make the running times implicit.

Braun trees are a form of balanced binary trees where the balance condition allows only a single shape of trees for a given size. Specifically, for each interior node, either the two children are exactly the same size or the left child's size is one larger than the right child's size.

Because this invariant is so strong, explicit balance information is not needed in the data structure that represents Braun trees; we can use a simple binary tree definition.

```
Inductive bin_tree {A:Set} : Set :=
| bt_mt : bin_tree
| bt_node : A -> bin_tree -> bin_tree
         -> bin_tree.
```

To be able to state facts about Braun trees, however, we use `Braun` to define when a binary tree is a Braun tree of size `n` (much like our definition of lists of length `n` in section 1).

```
Inductive Braun {A:Set} :
  (@bin_tree A) -> nat -> Prop :=
| B_mt : Braun bt_mt 0
| B_node :
    forall (x:A) s s_size t t_size,
      t_size <= s_size <= t_size+1 ->
      Braun s s_size ->
      Braun t t_size ->
      Braun (bt_node x s t) (s_size+t_size+1).
```

This says that the empty binary tree is a Braun tree of size `0`, and that if two numbers `s_size`, `t_size` are the sizes of two Braun trees `s` `t`, and if `s_size <= t_size <= s_size + 1`, then combining the `s` and `t` into a single tree produces a braun tree of size `s_size+t_size+1`.

Figure 1 shows the insertion function. It is fairly complex, so let us look carefully at each piece. It accepts an object `i` (of type `A`) to insert into the Braun tree `b`. Its result type uses a new notation:

```
{! «result variable» !:! «simple result type»
  !<! «running time variable» !>!
  «property of the function» !}
```

where the braces, exclamation marks, colons, less than, and greater than are all fixed parts of the syntax and the portions enclosed in « » are filled in based on the particulars of the insert function. In this case, it is saying that `insert` returns a binary tree and, if the input is a Braun tree of size `n`, then the result is a Braun tree of size `n+1` and the function takes `fl_log n + 1` steps of computation (where `fl_log` computes the floor of the base 2 logarithm).

```
Program Fixpoint insert {A:Set} (i:A)
                        (b:@bin_tree A)
: {! res !:! @bin_tree A !<! c !>!
     (forall n,
        Braun b n ->
        (Braun res (n+1) /\
         c = fl_log n + 1)) !} :=
  match b with
    | bt_mt =>
      += 1;
      <== (bt_node i bt_mt bt_mt)
    | bt_node j s t =>
      bt <- insert j t;
      += 1;
      <== (bt_node i bt s)
  end.
```

Figure 1: Braun tree insertion

These new `{! ... !}` types are the types of computations in the monad. The monad tracks the running time as well as tracking the correctness property of the function. Since we use a monadic type as the result of the function, we are saying that the function operates inside the monad and thus we can track its running time and its correctness properties.

The body of the `insert` function begins with the `match` expression that determines if the input Braun tree is empty or not. If it is empty, then the function returns a singleton tree that is obtained by calling `bt_node` with two empty children. This case uses `<==`, the return operation that injects simple values into the monad and `+=` that declares that this simple operation takes a single unit of computation. That is, the type of `+=` insists that it accepts a natural number `k` and a computation in the monad taking some number of steps, say `n`. The result of `+=` is also a computation in the monad just like the second argument, except that the running time is `n+k`.

In the non-empty case, the insertion function recurs with the right subtree and then builds a new tree with the subtrees swapped in order to preserve the Braun invariant. That is, since we know that the left subtree's size is either equal to or one larger than the right's, when we add an element to the right and swap the subtrees, we will also end up with a new tree whose left subtree's size is either equal to or one greater than the right.

The `«var» <- «expr» ; «expr»` notation is the monadic bind operator; if behaves much like a `let` expression. The first, right-hand side expression must be a computation in the monad; the result value is pulled out of the monad and bound to `var` for use in the body expression. Then, as before, we return the new tree in the monad after treating this as a single abstract step of computation.

Once this function is submitted to Coq, it responds with requests to prove two claims, one from each of the cases of the function. The first one is:

```
forall n,
  Braun bt_mt n ->
   Braun (bt_node i bt_mt bt_mt) (n + 1) /\
   1 = fl_log n + 1
```

which may look a bit baroque. The left-hand side of the implication is saying that we get to assume that n is the size of the empty binary tree. Of course, that tells us than n must be zero. So simplifying we are asked to prove that:

```
Braun (bt_node i bt_mt bt_mt) 1
/\
1 = fl_log 0 + 1
```

both of which follow immediately from the definitions. Note that this proof request corresponds exactly to what we need to know in order for the base case to be correct: the singleton tree is a Braun tree of size 1 and the running time is correct when the input is empty.

For the second case, we are asked to prove:

```
forall i j s t bt an n,
  (forall m : nat, Braun t m ->
     Braun bt (m + 1) /\ an = fl_log m + 1)
  Braun (bt_node j s t) n
  ->
  Braun (bt_node i bt s) (n + 1) /\
  an + 1 = fl_log n + 1
```

This is saying that we get to assume a slightly more general inductive hypothesis (the inner `forall`) than we need (it is specialized to the recursive call that `insert` makes but not the size of the tree) and that the tree `bt_node j s t` is a braun tree of size n. Given that, we must show that `bt_node i bt s` is a Braun tree of size `n + 1` and that the running time is right.

Because the size information is not present in the actual insertion function, Coq does not know to specialize the inductive hypothesis to the size of `t`. To clarify that we can replace `m` with `t_size` and specialize the first assumption to arrive at this theorem to prove

```
forall i j s t bt n t_size,
 Braun bt (t_size + 1)
 an = fl_log t_size + 1
 Braun (bt_node j s t) (s_size + t_size + 1)
 ->
 Braun (bt_node i bt s)
       (s_size + t_size + 1 + 1) /\
 an + 1 =
 fl_log (s_size + t_size + 1) + 1
```

which we can prove by using facts about logarithms and the details of the definition of Braun trees.

Note also that this theorem corresponds precisely to what we need to know in order to prove that the recursive case of `insert` works. That is, the assumptions correspond to the facts we gain from the input to the function and from the result of the recursive call and the final result corresponds to the facts we need to establish for this case.

3. Implicit Running Times

One disadvantage to the code in the previous section is that the running times are tangled with the body of the insertion function and, even worse, making a mistake when writing the += expressions can cause our proofs about the running times to be useless, as they will prove facts that aren't actually relevant to the functions we are using.

To handle this situation, we've written a simple Coq-to-Coq translation function that accepts functions written in our monad without any += expressions and turns them into ones with += expressions in just the right places.

Our translation function accepts a function written in the monad, but without the monadic type on its result and produces one with it. For example, the `insert` function shown on the left in figure 2 is translated into the one on the right. In addition to adding += expressions, the translation process also generates a call to `insert_result` in the monadic result type. This function must then be defined separately and the translation's output must be used in that context.

We follow Rosendahl (1989) and treat each function call, variable lookup, and case-dispatch as counting as a single unit of abstract time. Other cost functions are also possible to account for different cost semantics. Indeed, we could even adapt the machinery in a straight-forward way to count multiple aspects of the computation separately, e.g., the running time, memory accesses, or the number of allocations.

Here is the definition of `insert_result`:

```
Definition insert_time n := 9 * fl_log n + 6.
Definition insert_result (A : Set) (i : A)
                         (b:@bin_tree A)
                         (res:@bin_tree A) c :=
     (forall n,
        Braun b n ->
        (Braun res (S n) /\
         (forall xs, SequenceR b xs
                  -> SequenceR res (i::xs))
         /\ c = insert_time n)).
```

Unlike the previous version, this one accounts for the larger constant factors and it also includes a stricter correctness condition. Specifically, the new conjunct insists that if you linearize the result Braun tree into a list, then you get the same thing as linearizing the input and consing the new element onto the front of the list.

```
Program Fixpoint insert {A:Set} (i:A)
                        (b:@bin_tree A)
: @bin_tree A :=
match b with
 | bt_mt =>
   <== bt_node i bt_mt bt_mt
 | bt_node j s t =>
   bt <- insert j t;
   <== bt_node i bt s
end.
```

```
Program Fixpoint insert {A:Set} (i:A) (b:@bin_tree
A)
: {! res !:! @bin_tree A !<! c !>!
     insert_result A i b res c !} :=
  match b with
    | bt_mt =>
      += 6;
      <== (bt_node i bt_mt bt_mt)
    | bt_node j s t =>
      bt <- insert j t;
      += 9;
      <== (bt_node i bt s)
  end.
```

Figure 2: Inserting += into insert

4. Extracting the insert Function

One of the important benefits of our library is that none of the correctness conditions and running time infrastructure affects Coq's extraction process. In particular, our monad extracts as the identity monad, which means that the OCaml code produced by Coq does not require any modifications.

For example, here is how insert extracts:

```
type 'a bin_tree =
| Bt_mt
| Bt_node of 'a * 'a bin_tree * 'a bin_tree

 let rec insert i = function
| Bt_mt ->
  Bt_node (i, Bt_mt, Bt_mt)
| Bt_node (j, s, t) ->
  Bt_node (i, (insert j t), s)
```

This exactly the code that Coq produces, whitespace, newlines and all. The only declarations we added to aid Coq's extraction was the suggestion that it should inline the monad operations. And since the extracted version of our monad is the identity monad, the monad operations simply evaporate when they are inlined.

More importantly, however, note that this code does not have any proof residue, unlike the code extracted for drop in section 1.3.

5. The Monad

One way to account for cost is to use the monad to pair an actual values with a natural number representing the computations current cost, and then ensure that this number is incremented appropriately at each stage of the computation. Unfortunately, this cost would be part of the dynamic behavior of the algorithm. In other words, insert x bt would, return a new tree and a number. Unfortunately, this violates our goal of having no complexity residue in extracted programs.

In Coq parlance, the problem is that we have a pair of two Set values—the B and the nat—and Sets are, by definition, part of the computational content. Instead, we need to have a Set paired with something from the universe of truth propositions, Prop. The trouble is finding the right proposition.

We use a new function C that consumes a type and a proposition that is parameterized over values of the type and numbers. Specifically, we define C:

```
Definition C (A:Set) (P:A -> nat -> Prop) : Set
:=
   {a : A | exists (an:nat), (P a an)}.
```

For a given A and P, C A P is a dependent pair of a, a value of type A, and a proof that there exists some natural number an for which a and an are related by P. The right-hand side of this pair is a proposition, so it contributes no computational content.

For our insert function, we wrote

```
: {! res !:! @bin_tree A !<! c !>!
     (forall n,
        Braun b n ->
        (Braun res (n+1) /\
         c = fl_log n + 1)) !}
```

for the result type. This is a shorthand (using Coq's notation construct) for the following call to C, in order to avoid duplicating the type between !:! and !<!:

```
(C (@bin_tree A)
   (fun (res:@bin_tree A) =>
     (forall n,
        Braun b n ->
        (Braun res (n+1) /\
         c = fl_log n + 1))))
```

One important aspect of these C types is that the nat is bound only by an existental, and thus is not connected to the value or the computation. That is, when we know an expression has the type C A P, we don't know that its running time is really correct since the proof that the expression has that type can supply any nat it wants to satisfy the existential.

Thus, in order to guarantee the correct running times, we treat types of the form C A P as private to the definition of the monad and built a set of operations that can be combined in arbitrary ways but such that their combination ensures that the nat must be used as the running time.

The first of these operations is the monadic unit, ret. Suppose an program returns an empty list, <== nil. Such a program takes no steps to compute, because the value is readily available. This logic applies to all places where a computation ends. To do this, we define <== x to be ret _ _ x _, a use of the monad operator ret. The underscores ask Coq to fill in well-typed arguments (resorting to asking the user to provide proofs if necessary, as we saw in section 2).

This is the type of ret:

```
Definition ret (A:Set)
               (P:A -> nat -> Prop)
               (a:A)
               (Pa0:P a 0) : C A P.
```

This specifies that ret will only construct a C A P when given a proof, Pa0, that the correct/runtime property holds between the actual value return a and the natural number 0.

There are two other operations in our monad: bind that combines two computations in the monad, summing their running times, and inc that adds to the count of the running time. We tackle inc next.

Suppose a program returns a value, a, that takes exactly one step to compute. We would like our proof obligation to be P a 1. This means that the obligation on ret, P a 0, will be irrelevant or worse, wrong. However, there is a simple way out of this bind: what if the property for the ret were different than the property for the entire program? In code, what if the obligation were P' a 0? At worse, such a change would be irrelevant because there may not be a connection between P' and P. But, with this in mind we can choose a P' such that P' a 0 *is* P a 1.

We previously described P as relation between As and nats, but in Coq this is just a function that accepts a A and nat and returns a proposition. Thus, we can make P' be the function: fun a an => P a (an+1). This has the effect of transforming the runtime obligation on ret from above: as more steps take cost, the property itself accrues the cost so the proof that the verifier must provide that 0 is an appropriate cost is transformed into whatever the actual cost along that path was.

We enapsulate this logic into a simple extra-monadic operator, inc, that introduces k units of cost:

```
Definition inc (A:Set)
           k
           (PA : A -> nat -> Prop)
           (x:C A (fun x xn =>
                     forall xm,
                       xn + k = xm ->
                       PA x xm))
: C A PA.
```

In programs using our monad, we write += k e, a shorthand for inc _ k _ e.

In principle, the logic for bind is very similar. A bind represents a composition of two computations: an A-producing one and an A-consuming, B-producing one. If we assume that property for A is PA and PB for B, then a first attempt at a type for bind is:

```
Definition bind1
           (A:Set) (PA:A -> nat -> Prop)
           (B:Set) (PB:B -> nat -> Prop)
           (am:C A PA)
           (bf:A -> C B PB)
: C B PB.
```

This definition is incorrect from the perspective of cost, because it misses the key point of ensuring that whatever the cost was for producing the A, it is accounted for along with the cost of producing the B.

Suppose that the cost of generating the A were 7, then we should transform the property of the B computation to be fun b bn => PB b (bn+7). Unfortunately, we cannot "look inside" the A computation to know that it cost 7 units. Instead, we have to show that *whatever* the cost for A was, the cost of B is still as expected. This suggests a second attempt at a definition of bind:

```
Definition bind2
           (A:Set) (PA:A -> nat -> Prop)
           (B:Set) (PB:B -> nat -> Prop)
           (am:C A PA)
           (bf:A ->
               C B
                 (fun b bn =>
                    forall an,
                      PB b (bn+an)))
: C B PB.
```

Unfortunately, this is far too strong of a statement because there are some costs an that are too much. The only an costs that our proof about an application of bind must be concerned with are those that respect the PA property given the *actual* value of a that the A computation produced. We can use a dependent type on bf to capture the connection between the costs in a third attempt at the type for bind.

```
Definition bind3
           (A:Set) (PA:A -> nat -> Prop)
           (B:Set) (PB:B -> nat -> Prop)
           (am:C A PA)
           (bf:forall (a:A),
                 C B
                   (fun b bn =>
                      forall an,
                        PA a an ->
                        PB b (bn+an)))
: C B PB.
```

This version of `bind` is complete from a cost perspective but has one problem for practical theorem proving. The body of the function `bf` has access to the value `a`, but it does not have access to the correctness part of the property `PA`. At first blush, this doesn't matter because the proof of correctness for the result of `bf` *does* have access through the hypothesis `PA a an`. But, that proof context is not available when producing the `b` result. Instead, it assumes that `b` has already been computed. This means that if the proof of `PA` is necessary to compute `b`, then we will be stuck. The simplest case where this occurs is when `bf` performs non-structural recursion and must construct a well-foundness proof to perform the recursive call and this proof relies on the correctness of the `a` value. This occurs in some of the functions we discuss in our case study in section 6.

It is simple to incorporate this information into the type of `bf`, once you realize the need for it, by adding an additional proposition argument that corresponds to the right-hand side of the `C A PA` value `am`:

```
Definition bind
           (A:Set) (PA:A -> nat -> Prop)
           (B:Set) (PB:B -> nat -> Prop)
           (am:C A PA)
           (bf:forall (a:A)
                      (pa:exists an, PA a an),
                 C B
                   (fun b bn =>
                      forall an,
                        PA a an ->
                        PB b (an+bn)))
: C B PB.
```

And finally, when writing programs we use the notation

```
«x» <- «expr1» ; «expr2»
```

as a shorthand for

```
bind _ _ _ _ expr1 (fun (x : _) (am : _) =>
expr2)
```

Because all of the interesting aspects of these operations happen in their types, the extraction of these operations have no interesting dynamic content. Specifically `ret` is simply the identity function, `inc` is a function that just returns its second argument and `bind` simply applies its second argument to its first. Furthermore, we have proven that they obey variants of the monad laws that incorporate the proof obligations.

In summary, the monad works by requiring the verifier to predict the running-time with the `PA` property and then provide evidence that the actual cost (that starts at `0` and is incremented as the property passes down) is the same as this prediction.

6. Case Study

To better understand how well our monad works, we implemented all of the algorithms in Okasaki (1997)'s paper, *Three Algorithms on Braun Trees*. The source code for our case study is on github:

`https://github.com/rfindler/395-2013`.

Okasaki's paper contains several versions of each of the three functions, each with different running times, in each case culminating with efficient versions. The three functions are

- `size`: computes the size of a Braun tree (a linear and a log squared version)
- `copy`: builds a Braun tree a given size filled entirely with a given element (a linear, a fib ∘ log, a log squared, and a log time version), and
- `make_array`: convert a list into a Braun tree (two n log n versions and a linear version).

6.1 Line Counts

The line counts for the various implementations of the algorithms using our monad are shown in figure 3. The files whose names end in `gen.v` are the output of the script that inserts += expressions, so they contain the complete definitions of the various functions and their helper functions that we implemented. As you can see, the functions are generally short. The proofs are typically much longer.

We divided the line counts up into proofs that are inside obligations (and thus correspond to establishing that the monadic types are correct) and other lines of proofs. With the exception of the `make_array_linear` files, the proofs inside the obligations establish the correctness of the functions and establish a basic running time result, but not one in terms of Big Oh.

For example, Figure 4 is the definition of the `copy_log_sq` function, basically mirroring Okasaki's definition, but in Coq's notation.

The monadic result type is

```
Definition copy_log_sq_result
           (A:Set) (x:A) (n:nat)
           (b:@bin_tree A) (c:nat):=
  Braun b n /\
  SequenceR b (mk_list x n) /\
  c = copy_log_sq_time n.
```

File	Non-blank Lines	Obligation Lines	Other Proof Lines
size_linear.v	36	16	1
size_linear_gen.v	13	0	0
Subtotal	49	16	1
size_log_sq.v	338	100	155
diff_gen.v	19	0	0
size_log_sq_gen.v	13	0	0
Subtotal	370	100	155
copy_linear.v	44	22	1
copy_linear_gen.v	13	0	0
Subtotal	57	22	1
copy_fib_log.v	616	90	358
copy_fib_log_gen.v	17	0	0
Subtotal	633	90	358
copy_log_sq.v	310	56	179
copy_log_sq_gen.v	16	0	0
Subtotal	326	56	179
copy_log.v	88	28	21
copy_log_gen.v	9	0	0
copy2_gen.v	18	0	0
Subtotal	115	28	21
make_array_nlogn1.v	134	12	79
make_array_nlogn1_gen.v	13	0	0
Subtotal	147	12	79
make_array_nlogn1_fold.v	115	13	59
Subtotal	115	13	59
make_array_nlogn2.v	185	57	64
make_array_nlogn2_gen.v	17	0	0
unravel_gen.v	15	0	0
Subtotal	217	57	64
make_array_linear.v	460	39	241
make_array_linear_gen.v	13	0	0
rows.v	380	115	145
rows1_gen.v	6	0	0
rows_gen.v	20	0	0
take_drop_split.v	195	91	26
drop_gen.v	18	0	0
take_gen.v	18	0	0
pad_drop_gen.v	19	0	0
split_gen.v	7	0	0
foldr_build_gen.v	13	0	0
zip_with_3_bt_node_gen.v	24	0	0
build.v	91	41	2
build_gen.v	14	0	0
Subtotal	1278	286	414
Monad	217	0	53
Common	1504	4	787
Total	8335	1364	3502

Figure 3: Line Counts

```
Program  Fixpoint  copy_log_sq  {A:Set}  (x:A)
(n:nat) {measure n}
: {! res !:! bin_tree !<! c !>!
     copy_log_sq_result A x n res c !} :=
  match n with
    | 0 =>
      += 3;
      <== bt_mt
    | S n' =>
      t <- copy_log_sq x (div2 n');
      if (even_odd_dec n')
      then (+= 13;
            <== (bt_node x t t))
      else (s <- insert x t;
            += 16;
            <== (bt_node x s t))
  end.
```

Figure 4: copy_log_sq

which says that the result is a Braun tree whose size matches the input natural number, that linearizing the resulting tree produces the input list, and that the running time is given by the function copy_log_sq_time.

The running time function, however, is defined in parallel to log_sq itself, not as the product of the logs:

```
Program Fixpoint copy_log_sq_time (n:nat)
        {measure n} :=
  match n with
    | 0 => 3
    | S n' =>
      if (even_odd_dec n')
      then 13 + copy_log_sq_time (div2 n')
      else 16 +
           copy_log_sq_time (div2 n') +
           insert_time (div2 n')
  end.
```

This makes it straightforward to prove that the running-time matches that function, but then leaves as a separate issue the proof that this function is Big Oh of the square of the log. That is, there are 56 lines of proof to guarantee the result type of the function is correct and an additional 179 lines to prove that that copy_log_sq_time is Big Oh of the product of the log.

For the simpler functions (every one with linear running time except make_array_linear), the running time can be expressed directly in the monadic result (with precise constants). However, for most of the functions the running time is expressed first precisely in a manner that matches the structure of the function and then that running time is proven to correspond to some asymptotic complexity, as with copy_log_sq. The precise line counts can be read off of the columns in figure 3.

The Monad and Common lines count the number of lines of code in our monad's implementation (including the proofs of the monad laws) and some libraries used in multiple algorithms, including a Big Oh library, a Log library, the Braun tree definition, and some common facts and definitions about Braun trees.

6.2 Extraction

The extracted functions naturally fall into two categories. In the first category are functions that recur on the natural structure of their inputs, e.g., functions that process lists by walking down the list one element at a time, functions that process trees by processing the children and combine the result, or functions that recursively process numbers by counting down by ones from a given number. In the second category are functions that "skip" over some of their inputs. In our case study, the only such functions process numbers and generally recur by diving the number by 2 instead of subtracting one.

Functions in the first category extract into precisely the OCaml code that you would expect, just like insert, as discussed in section 2.

Functions in the second category extract into code that is cluttered by Coq's support for non-simple recursion schemes. That is, each function in Coq must be proven to be well-defined and to terminate on all inputs. Functions that don't simply follow the natural recursive structure of their input get additional wrappers and useless data structure.

The function copy_log_sq is one such. Here is its output:

```
(** val copy_log_sq_func : (__, (__, int) sigT)
sigT -> __ bin_tree c **)

let rec copy_log_sq_func x =
  let x0 = let ExistT (a, p) = let ExistT (x0,
h) = x in h in a in
  let n = let ExistT (x1, h) = let ExistT (x1,
h) = x in h in h in
  let copy_log_sq0 = fun x1 n0 ->
    let y = ExistT (__, (ExistT (x1, n0)))
      in copy_log_sq_func y
  in
  ((fun fO fS n -> if n=0 then fO () else fS
(n-1))
     (fun _ ->
     Bt_mt)
     (fun n' ->
     let t = copy_log_sq0 x0 (div2 n') in
     (match even_odd_dec n' with
      | false -> Bt_node (x0, t, t)
      | true -> Bt_node (x0, (insert x0 t), t)))
     n)
```

```
(** val copy_log_sq : 'a1 -> int -> 'a1 bin_tree
c **)

let copy_log_sq x n =
  Obj.magic (copy_log_sq_func (ExistT (__, (Ex-
istT ((Obj.magic x), n)))))
```

All of the extra pieces beyond what was written in the original function are useless. In particular, the argument to copy_log_sq_func is a three-deep nested pair containing an integer (a real argument), the value in the tree (also a real argument), and __ a constant that is defined at the top of the extraction file that is never used for anything. Similarly, the body of the function has the anonymous function that begins fun f0 fS n -> that is simply an extra wrapper around a conditional. Simplifying these two away and inlining copy_log_sq_func and copy_log_sq0 results in this program:

```
let rec copy_log_sq x0 n =
  if n=0
  then Bt_mt
  else let n'=n-1 in
   let t = copy_log_sq x0 (div2 n') in
     match even_odd_dec n' with
      | false -> Bt_node (x0, t, t)
      | true -> Bt_node (x0, (insert x0 t), t))
```

which is precisely the code that we would expect (compare with the Coq code for the same function in the previous subsection). Similar simplifications apply to the other functions in the second category and these changes correspond to fairly aggressive inlining, something that high-quality functional language compilers tend to do.

7. Accounting for Language Primitives

Rosendahl (1989)'s cost function counts all primitive functions as constant (simply because it counts a call as unit time and then doesn't process the body). For most primitives, this exactly what you want. For example, field selection functions (e.g., car and cdr) are certainly constant time. Structure allocation functions (e.g., cons) are usually constant time, when using a two-space copying collector, as most garbage-collected languages do. Occasionally allocation triggers garbage collection which is probably amortized constant time but certainly something our framework does not handle.

More interestingly and more often overlooked, however, are the numeric primitives. In a language implementation with bignums, numbers are generally represented as a list of digits in some large base with grade-school arithmetic algorithms implementing the various operations. These operations are generally not all constant time.

Assuming that the base is a power of 2, division by 2, evenness testing, and checking to see if a number is equal to 0 are all constant-time operations. The algorithms discussed in section 6 use two operations on numbers besides those: + and sub1.

In general, addition of bignums is not constant time. However, certain uses of addition can be replaced by constant-time bit operations. For instance, doubling and adding 1 can be replaced by a specialized operation that conses a 1 on the front of the bitstring. Fortunately, every time we use addition in one of the functions in our Braun library, the operation can be replaced by one of these efficient operations.

One of the more interesting uses is in the linear version of size, which has the sum lsize+rsize+1 where lsize and rsize are the sizes of two subtrees of a Braun tree. This operation, at first glance, doesn't seem to be a constant-time. But the Braun invariant tells us that lsize and rsize are either equal, or that lsize is rsize+1. Accordingly, this operation can be replaced with either lsize*2+1 or lsize*2, both of which are constant-time operations. Also, checking to see which case applies is a constant time operation: if the numbers are the same the leading digits will be the same and if they differ by 1, the leading digits will be different.

In general, sub1 is not constant time. It some situations, it may need to traverse the entire number to compute its predecessor. To explore its behavior, we implemented it using only constant-time operations. Here's the result of our translation applied to its implementation:

```
Program Fixpoint sub1 (n:nat) {measure n}
: {! res !:! nat !<! c !>!
     sub1_result n res c !} :=
  match n with
    | 0 =>
      += 3;
      <== 0
    | S _ =>
      if (even_odd_dec n)
      then (sd2 <- sub1 (div2 n);
            += 12;
            <== (sd2 + sd2 + 1))
      else (+= 8;
            <== (n - 1))
  end.
```

This function uses a number of primitive operations. If we think the representation of numbers as a linked listed of binary digits, then these are the operations used by the function and their corresponding list operation:

- zero testing: empty list check,
- evenness testing: extract the first bit in the number
- floor of division by 2 (div2): cdr
- double and add 1 (sd2+sd2+1): cons 1 onto the list
- n-1 in the case that n is odd: replace the lowest bit with 0

So while each iteration of the list runs in constant time, sub1 is recursive and we do not know an a priori bound on the number of iterations.

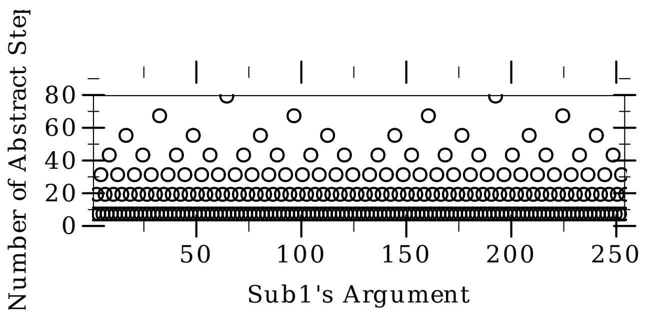

Figure 5: Running time of sub1

We can use our implementation of sub1 above to graph its running time against the value of its input as shown in Figure 5.

Half of the time (on odd numbers) sub1 is cheap, terminating after only a single iteration. One quarter of the time, sub1 terminates after two iterations. In general, there is a $\frac{1}{2^n}$ chance that sub1 terminates after n iterations.

There are four different recursion patterns using sub1 in our library. The first and simplest is a function with a loop counting down from n to 0. This pattern is found in the functions take, drop and pad-drop in the library. In this case, sub1 will be called once for each different number in that range and those times sum to a fixed number, proportional to the number of iterations.

Second is a function that loops by subtracting 1 and then dividing by 2. This pattern is found in our functions copy2 and copy_insert, and has a logarithmic overhead.

Third is the pattern used by diff, which recurs with either (n-1)/2 or (n-2)/2 depending on the parity of the index. This is again a logarithmic overhead to diff, which has logarithmic complexity.

Finally, the most complicated is the pattern used by copy_linear, which recursively calls itself on n/2 and (n-1)/2. Figure 6 is a plot of the running time of the sub1 calls that copy_linear makes. In gray is a plot of $\lambda x.31x+29$, which we believe is an upper bound for the function.

Accordingly, we believe that the overhead of using sub1 in these functions does not change their asymptotic complexity, but we have verified this only by testing (and, in the first case, by a pencil-and-paper proof).

8. Related Work

The most closely related work to our is Danielsson (2008). He presents a monad that, like ours, carries a notion of abstract time. Unlike our monad, his does not also carry an invariant – in our terms his monad construction does not have the P argument. This means that it is not possible to specify the running time of many of the Braun functions, since the size information is not available without the additional assumption of Braunness. Also, his monad would leave nat-

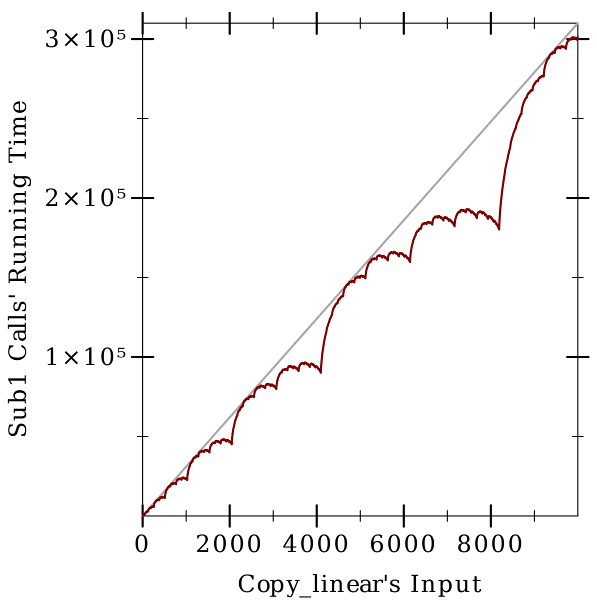

Figure 6: Running time of `copy_linear`

ural numbers in the extracted code; avoiding that is a major goal of this work.

Weegen and McKinna (2008) give a proof of the average case complexity of Quicksort in Coq. They too use monads, but design a monad that is specially tailored to counting only comparison operations. They side-step the extraction problem by abstracting the implementation over a monad transformer and use one monad for proving the correct running times and another for extraction.

Xi and Pfenning first seriously studied the idea of using dependent types to describe invariants of data structures in practical programming languages (Xi 1999a,b; Xi and Pfenning 1999) and, indeed, even used Braun trees as an example in the DML language, which could automatically prove that, for example, `size_log_sq` is correct.

Filliâtre and Letouzey (2004) implemented a number of balanced binary tree implementations in Coq with proofs of correctness (but not running time), with the goal of high-quality extraction. They use the technique described in section 1.4.

Swierstra (2009)'s Hoare state monad is like our monad in that it exploits monadic structure to make proof obligations visible at just the right moments. However, the state used in their monad has computational content and thus is not intended to be erased during extraction.

Charguéraud (2010)'s characteristic formula generator seems to produce Coq code with obligations similar to what our monad produces, but it does not consider running time.

Our code builds heavily on Sozeau (2006)'s `Program` facility in Coq.

Acknowledgments. Thanks to Neil Toronto for help with the properties of integer logarithms (including efficient implementations of them). This work grew out of a PL seminar at Northwestern; thanks to Benjamin English, Michael Hueschen, Daniel Lieberman, Yuchen Liu, Kevin Schwarz, Zach Smith, and Lei Wang for their feedback on early versions of the work.

Bibliography

W Braun and M Rem. A Logarithmic Implementation of Flexible Arrays. Eindhoven University of Technology, MR83/4, 1983.

Arthur Charguéraud. Characteristic Formulae for Mechanized Program Verification. PhD dissertation, Université Paris Diderot (Paris 7), 2010.

Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proc. Proc. Symposium on Principles of Programming Languages*, 2008.

Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proc. Proc. European Symposium on Programming*, 2004.

Chris Okasaki. Three Algorithms on Braun Trees. *Journal of Functional Programming* 7(6), 1997.

Mads Rosendahl. Automatic Complexity Analysis. In *Proc. Proc. International Conference on Functional Programming Languages And Computer Architecture*, 1989.

Matthieu Sozeau. Subset Coercions in Coq. In *Proc. Proc. Workshop of the Working Group Types*, 2006.

Wouter Swierstra. A Hoare Logic for the State Monad. In *Proc. Proc. Theorem Proving in Higher Order Logics*, 2009.

Eelis van der Weegen and James McKinna. A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. In *Proc. Proc. Workshop of the Working Group Types*, 2008.

Hongwei Xi. Dependently Typed Data Structures. In *Proc. Proc. Workshop on Algorithmic Aspects of Advanced Programming Languages*, 1999a.

Hongwei Xi. Dependently Types in Practical Programming. PhD dissertation, Carnegie Mellon University, 1999b.

Hongwei Xi and Frank Pfenning. Dependently Types in Practical Programming. In *Proc. Proc. Symposium on Principles of Programming Languages*, 1999.