A Coq Library For Internal Verification of Running-Times

Jay McCarthy
Vassar College
jay.mccarthy@gmail.com

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.

Our library is based on a monad that counts abstract steps, controlled by one of the monadic operations. The monad's computational content, however, is simply that of the identity monad so programs written in our monad (that recur on the natural structure of their arguments) extract into idiomatic OCaml code.

We evaluated the expressiveness of the library by proving that red-black tree insertion and search, merge sort, insertion sort, Fibonacci, iterated list insertion, and Okasaki's Braun Tree algorithms all have their expected running times.

1. Introduction

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 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 also an abstract step count describing how many primitive operations (function calls, matches, variable references etc.) that the function executes.

To give a sense of how code using our library looks, we start with a definition of Braun trees (Braun and Rem 1983) and the insertion function where the contributions to the running time are explicitly declared as part of the body of the function. In the next section, we make the running times implicit (and thus not trusted or spoofable).

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 need the inductive `Braun` to specify which binary trees are Braun trees (at a given size `n`).

```
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`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

, .
Copyright © ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/ **Draft as of Saturday, February 28th, 2015**

```
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

Figure 1 shows the insertion function. Let us dig into this function, one line at a time. 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).

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.

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 operation takes a single unit of computation. That is, the type of `+=` insists that `+=` 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. This swapping is what preserves the Braun invariant. 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; using a `let`-style notation. 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 branch as a single abstract step of computation.

We exploit Sozeau (2006)'s `Program` to smooth proving these functions have their types. In this case, we are left with two proof obligations, 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
```

The assumption is saying that `n` is the size of the empty Braun tree, which tells us that `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
```

Thus, we may 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. So, we must show that `bt_node i bt s` is a Braun tree of size `n + 1` and that the running time is correct.

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.

This theorem corresponds precisely to what we need to know in order to prove that the recursive case of `insert` works. The assumptions correspond to the facts we gain from the input to the function and from the result of the recursive call. The conclusion corresponds to the facts we need to establish for this case. This precision of the obligation is thanks to `Program` and the structure of our monad.

2. 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. As well as 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 a single unit of abstract time. The function is straight-forward and is included in the supplementary materials (`add-plusses/check-stx-errs` in `rkt/tmonad/main.rkt`). Different cost semantics are possible, provided a function could map them to the program's syntax in a straight-forward way.

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 resulting Braun tree into a list, then it is the same as linearizing the input and consing the new element onto the front of the list.

3. 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 is 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; there are no extra data-structures or function arguments or other artifacts of the information used to prove the running time correct.

4. The Monad

One way to account for cost is to use the monad to pair an actual value (of type `B`) with a natural number representing the computation's 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`

```
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

`x bt` would return a new tree and a number, violating 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 `Set`s 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 intention is to think of the natural number as the running time and `P` as some function-specific specification of running time (and possibly also correctness). Importantly, the right-hand side of this pair is a proposition, so it contributes no computational content when extracted into OCaml.

For our `insert` function, we write the result type as:

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

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) (c:nat) =>
     (forall n,
        Braun b n ->
        (Braun res (n+1) /\
         c = fl_log n + 1))))
```

One important aspect of the `C` type is that the `nat` is bound only by an existential, and thus is not connected to the value or the computation. Therefore, when we know an expression has the type `C A P`, we do not know that its running time is correct. This is because the proof that the expression is that type can supply any `nat` 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. We build a set of operations that can be combined in arbitrary ways but such that their combination ensures that the `nat` used must be the running time.

The first of these operations is the monadic unit, `ret`. Suppose a 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 (asking the user to provide proofs, if necessary, as we saw in section 1).

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 construct a C A P only when given a proof, Pa0, that the correctness/runtime property holds between the actual value returned a and the natural number 0. In other words, ret requires P to predict the running time as 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 with property P, that takes exactly one step to compute. We write this using the expression:

```
 += 1;
<== a
```

We would like our proof obligation for this expression to be P a 1. We know, however that the obligation on <==, namely P a 0, is irrelevant or worse, wrong. There is a simple way out of this bind, however: what if the P for the ret were different than the property for of the entire expression? In code, what if the obligation were P' a 0? At worst, such a change would be irrelevant because there may not be a connection between P' and P. With this in mind, we can choose a P' such that P' a 0 is the same as P a 1.

We previously described P as a relation between As and nats, but in Coq this is just a function that accepts an A and a nat and returns a proposition. So, 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 what was described above. The proof P a 0 becomes P a 1. In general, if the cost along a control-flow path to a ret has k units of cost, the proof will be P a k. Thus, we accrue the cost inside of the property itself.

We encapsulate this logic into a simple 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.

The key point in the definition is that the property in x's type is *not* PA, but a modified function that ensures the argument is at least k.

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 does not ensure that the cost for producing the A is accounted for along with the cost of producing the B.

Suppose that the cost of generating the A was 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 costs 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 bind proof 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, the missing PA appears not to 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, bind assumes that b has already been computed. That assumption means if the proof of PA is needed to compute b, then we will be stuck. The most common case where PA is neccessary occurs when bf performs non-structural recursion and must construct a well-foundness proof to perform the recursive call. These well-foundness proofs typically rely on the correctness of the a value. Some of the functions we discuss in our case study in section 5 could not be written with this version of bind.

It is simple to incorporate the PA proof 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 applies its second argument to its first. Furthermore, we have proven that they obey variants of the monad laws that incorporate the proof obligations (see the file monad/laws.v in the supplementary material).

In summary, the monad works by requiring the verifier to predict the running-time in the PA property and then prove that the actual cost (starting at 0 and incrementing as the property passes down) matches the prediction.

5. Case Study

To better understand how applicable our monad is, we implemented the search and insertion functions for red-black trees, insertion sort, merge sort, both the naive recursive version of the nth Fibonacci number function and the linear time version, a function that inserts m times into a list at position n using both lists and zippers, and all of the algorithms mentioned in Okasaki (1997)'s paper, *Three Algorithms on Braun Trees*. 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 $\circ$ 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).

In total, we implemented 17 different functions using the monad. For all of them, we proved the expected Big O running times. For merge sort, we proved it is Big O($n\log(n)$) and Big $\Omega(n\log(n))$. For the naive fib, we proved that it is Big Ω and Big O of itself, Big O(2^n), and Big $\Omega(2^{n/2})$. For the list insertion functions, we prove that when m is positive, the zipper version is Big O of the list version (because the zipper version runs in Big O($m+n$) while the list version runs in Big O($n*m$). We also prove correctness, except for make_array_linear and red-black tree insertion where we proved only the running times.

The supplementary material contains all of the Coq code for all of the functions in our case study.

5.1 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 definitions of the various functions, but without the correctness conditions (or any of the proofs or data structure definitions). There are more than 17 because a number of the functions needed helper functions that are in the monad (and thus require running time proofs). 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 and the red-black tree insertion function, 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 O.

File	Non-Proof Lines	Obligation Lines	Other Proof Lines
rbtree.v	155	0	126
rbt_search.v	56	106	6
bst_search_gen.v	26	0	0
Subtotal	237	106	132
rbt_insert.v	171	54	179
rbt_balance_gen.v	347	0	0
rbt_blacken_gen.v	11	0	0
rbt_insert_gen.v	8	0	0
rbt_insert_inner_gen.v	28	0	0
Subtotal	565	54	179
sorting.v	20	0	5
Subtotal	20	0	5
isort.v	62	132	53
insert_gen.v	18	0	0
isort_gen.v	13	0	0
Subtotal	93	132	53
merge_gen.v	25	0	0
mergesort.v	524	400	813
mergesort_gen.v	7	0	0
mergesortc_gen.v	20	0	0
split2_gen.v	18	0	0
clength_gen.v	12	0	0
Subtotal	606	400	813
fib.v	138	57	151
fib_iter_gen.v	18	0	0
fib_iter_loop_gen.v	12	0	0
fib_rec_gen.v	19	0	0
Subtotal	187	57	151
to_list_naive.v	58	53	22
cinterleave_gen.v	12	0	0
to_list_naive_gen.v	14	0	0
Subtotal	84	53	22
zip.v	235	270	70
from_zip_gen.v	5	0	0
insert_at_gen.v	18	0	0
minsert_at_gen.v	13	0	0
minsertz_at_gen.v	10	0	0
to_zip_gen.v	5	0	0
zip_insert_gen.v	5	0	0
zip_left_gen.v	11	0	0
zip_leftn_gen.v	13	0	0
zip_minsert_gen.v	13	0	0
zip_right_gen.v	11	0	0
zip_rightn_gen.v	13	0	0
Subtotal	352	270	70
Monad	245	0	114
Common	835	4	1,640

File	Non-Proof Lines	Obligation Lines	Other Proof Lines
copy_linear.v	21	22	1
copy_linear_gen.v	13	0	0
Subtotal	34	22	1
copy_fib_log.v	146	90	313
copy_fib_log_gen.v	17	0	0
Subtotal	163	90	313
copy_log_sq.v	75	56	179
copy_log_sq_gen.v	16	0	0
Subtotal	91	56	179
copy_log.v	39	28	21
copy_log_gen.v	9	0	0
copy2_gen.v	18	0	0
Subtotal	66	28	21
make_array_nlogn1.v	43	12	79
make_array_nlogn1_gen.v	13	0	0
Subtotal	56	12	79
make_array_nlogn1_fold.v	43	13	59
Subtotal	43	13	59
make_array_nlogn2.v	64	57	64
make_array_nlogn2_gen.v	17	0	0
unravel_gen.v	15	0	0
Subtotal	96	57	64
make_array_linear.v	180	39	241
make_array_linear_gen.v	13	0	0
rows.v	120	115	145
rows1_gen.v	6	0	0
rows_gen.v	20	0	0
take_drop_split.v	78	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	48	41	2
build_gen.v	14	0	0
Subtotal	578	286	414
size_linear.v	19	16	1
size_linear_gen.v	13	0	0
Subtotal	32	16	1
size_log_sq.v	83	100	155
diff_gen.v	19	0	0
size_log_sq_gen.v	13	0	0
Subtotal	115	100	155
Totals	3,628	1,364	3,750
Total number of lines:	8,742		

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

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.
```

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 parallel definition makes it straightforward to prove that copy_log_sq's running time equals copy_log_sq_time, but leaves as a separate issue the proof that copy_log_sq_time is Big O of the square of the log. 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 O of the square 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 first expressed 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 O library, a Log library, and some common facts and definitions about Braun trees.

5.2 Extraction

The extracted functions naturally fall into three categories.

In the first category are functions that recur on the natural structure of their inputs, e.g., functions that process lists from the front–one element at a time, functions that process trees by processing the children and combining the result, and so on. In the second category are functions that recursively process numbers by counting down by ones from a given number. In the third category are functions that "skip" over some of their inputs. For example, in Okasaki's algorithms, functions recur on natural numbers by dividing the number by 2 instead of subtracting one, and merge sort recurs by dividing the list in half at each step.

Functions in the first category extract into precisely the OCaml code that you would expect, just like insert, as discussed in section 1.

Functions in the second category could extract like the first, except because we extract Coq's nat type, which is based on Peano numerals, into OCaml's big_int type, which has a different structure, a natural match expression in Coq becomes a more complex pattern in OCaml. A representative example of this pattern is zip_rightn. Here is the extracted version:

```
let rec zip_rightn n z =
  (fun f0 fS n -> if (eq_big_int n zero_big_int)
then f0 () else fS (pred_big_int n))
    (fun _ ->
    z)
    (fun np ->
    zip_rightn np (zip_right z))
    n
```

The body of this function is equivalent to a single conditional that returns z when n is 0 and recursively calls zip_rightn on n-1 otherwise. This artifact in the extraction is simply a by-product of the mismatch between nat and big_int. We expect that this artifact can be automati-

cally removed by the OCaml compiler. This transformation into the single conditional corresponds to modest inlining, since `fO` and `fS` occur exactly once and are constants.

Functions in the third category, however, are more complex. They extract into code that is cluttered by Coq's support for non-simple recursion schemes. Because 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 must have supplemental arguments that record the decreasing nature of their input. After extraction, these additional arguments clutter the OCaml code with useless data structures equivalent to the original set of arguments.

The function `cinterleave` is one such function. Here is the extracted version:

```
let rec cinterleave_func x =
  let e = let a,p = let x0,h = x in h in a in
  let o = let x0,h = let x0,h = x in h in h in
  let cinterleave0 = fun e0 o0 ->
   let y = __,(e0,o0) in cinterleave_func y in
  (match e with
   | Nil -> o
   | Cons (x, xs) ->
     Cons (x, (cinterleave0 o xs)))


let cinterleave e o =
 Obj.magic
  (cinterleave_func
   (__,((Obj.magic e),(Obj.magic o))))
```

All of the extra pieces beyond what was written in the original function are useless. In particular, the argument to `cinterleave_func` is a three-deep nested pair containing `__` and two lists. The `__` is a constant that is defined at the top of the extraction file that is never used for anything and behaves like `unit`. That piece of the tuple corresponds to a proof that the combined length of the two lists is decreasing. The function starts by destructuring this complex argument to extract the two lists, `e` and `o`. Next it constructs a version of the function, `cinterleave0`, that recovers the natural two argument function for use recursively in the body of the `match` expression. Finally, this same two argument interface is reconstructed a second time, `cinterleave`, for external applications. The external interface has an additional layer of strangeness in the form of applications of `Obj.magic` which can be used to coerce types, but here is simply the identity function on values and in the types. These calls correspond to use of `proj1_sig` in Coq to extract the value from a Sigma type and are useless and always successful in OCaml.

All together, the OCaml program is equivalent to:

```
let rec cinterleave e o =
 match e with
 | Nil -> o
 | Cons (x, xs) -> Cons (x, (cinterleave o xs))
```

This is exactly the Coq program and idiomatic OCaml code. Unlike the second category, it is less plausible that the OCaml compiler already performs this optimization and removes the superfluity from the Coq extraction output. However, it is plausible that such an optimization pass could be implemented, since it corresponds to inlining, de-tupling, and removing an unused `unit`-like argument. In summary, the presence of these useless terms is unrelated to our running time monad, but is an example of the sort of verification residue we wish to avoid and do successfully avoid in the case of the running time obligations.

The functions in the first category are: `insert`, `size_linear`, `size`, `make_array_naive`, `foldr`, `make_array_naive_foldr`, `unravel`, `to_list_naive`, `isort`'s `insert`, `isort`, `clength`, `minsert_at`, `to_zip`, `from_zip`, `zip_right`, `zip_left`, `zip_insert`, `zip_minsert`, `minsertz_at`, `bst_search`, `rbt_blacken`, `rbt_balance`, `rbt_insert`.

The functions in the second category are: `fib_rec`, `fib_iter`, `sub1`, `mergesort`'s `split`, `insert_at`, `zip_rightn`, `zip_leftn`.

The functions in the third category are: `copy_linear`, `copy_fib`, `copy_log_sq`, `copy2`, `diff`, `make_array_td`, `cinterleave`, `merge`, `mergesort`. Some of the functions in the second category are also in the third category.

6. 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 is 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 not something our framework handles).

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 in our study use two other numeric operations `+` and `sub1` (not counting the abstract comparison in the sorting functions).

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. Checking to see which case applies is also constant time: if the numbers are the same, the digits at the front of the respective lists will be the same and if they differ by 1, those digits will be different.

The uses of addition in `fib`, however, are not constant time and so our analysis of `fib` is not accurate at that level. We did not attempt to improve that treatment, but we did study `sub1` more carefully.

Subtracting 1 from a number in binary representation is not constant time. In some situations, `sub1` may need to traverse the entire number to compute its predecessor. To explore `sub1`'s behavior, we implemented it using only constant-time operations. Here's the implementation with explicit costs:

```
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 of 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`

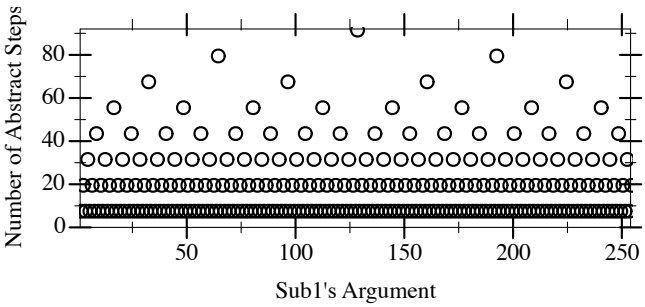

Figure 5: Running time of `sub1`

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.

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 the total running time is 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).

7. Related Work

The most closely related work to ours 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. In our opinion, figuring out the design of monad operations that support the `P` argument is the major

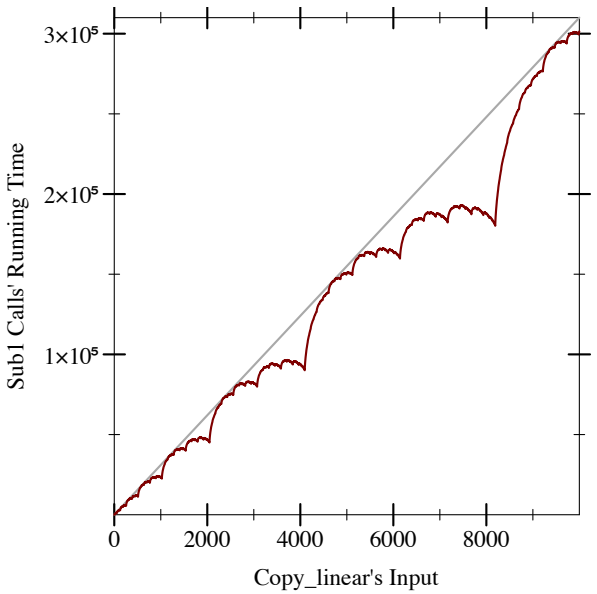

Figure 6: Running time of `copy_linear`

technical advance here. Accordingly, Danielsson (2008)'s system cannot 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 natural numbers in the extracted code; avoiding that is a major goal of this work.

While Crary and Weirich (2000)'s work does not leverage the full expressiveness of a theorem proving system like Coq's, it does share a similar resemblance to our approach. Also like Danielsson (2008)'s and unlike ours, it does not provide a place to carry an invariant of the data structures that can be used to establish running times.

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 an "external" approach, where the types do not carry the running time information, which makes the proofs more complex.

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.

Others have explored automatic techniques for proving that programs have particular resource bounds using a variety of techniques (Gulwani et al. 2009; Hoffmann and Shao 2015; Hofmann and Jost 2003; Hughes and Pareto 1999) These approaches are all weaker than our approach, but provide more automation.

We have consistently used the word "monad" to describe what our library provides and believe that that is a usefully evocative word to capture the essence of our library. It probably is not, however, technically accurate because the proof information changes the types of the operations, making it some kind of generalized form of monad, perhaps a specialization of Atkey (2009)'s or Altenkirch et al. (2010)'s.

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

Acknowledgments. Thanks to reviewers of previous versions of this paper. 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

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads Need Not Be Endofunctors. In *Proc. Proc. Foundations of Software Science and Computation Structure*, 2010.

Robert Atkey. Parameterised Notions of Computation. *Journal of Functional Programming* 19(3-4), 2009.

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.

Karl Crary and Stephanie Weirich. Resource bound certification. In *Proc. Proc. Symposium on Principles of Programming Languages*, 2000.

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.

Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. Proc. Symposium on Principles of Programming Languages*, 2009.

Jan Hoffmann and Zhong Shao. Automatic Static Cost Analysis for Parallel Programs. In *Proc. Proc. European Symposium on Programming*, 2015.

Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. Proc. Symposium on Principles of Programming Languages*, 2003.

John Hughes and Lars Pareto. Recursion and Dynamic Data-structures in bounded space: Towards Embedded ML Programming. In *Proc. Proc. Intl. Conference on Functional Programming*, 1999.

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

Mads Rosendahl. Automatic Complexity Analysis. In *Proc. Proc. Intl. 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.