A Coq Library For Internal Verification of Running-Times

Jay McCarthy[1], Burke Fetscher[2], Max New[2],
Daniel Feltey[2], and Robert Bruce Findler[2]

[1] University of Massachusetts at Lowell
jay.mccarthy@gmail.com
[2] Northwestern University
{burke.fetscher, max.new, daniel.feltey, 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, BigNum addition, and Okasaki's Braun Tree algorithms all have their expected running times.

1 Introduction

For some programs, proving that they have correct input-output behavior is only part of the story. As Crosby and Wallach (2003) observed, incorrect performance characteristics can also lead to security vulnerabilities. Indeed, some programs and algorithms are valuable precisely because of their performance characteristics. For example, mergesort is preferable to insertion sort only because of its improved running time. Unfortunately, defining functions in Coq or other theorem proving systems does not provide enough information in the types to be able to state these intensional properties.

Our work provides a monad (implemented as a library in Coq) that enables us to include abstract running times in types. We use this library to prove several important algorithms have their expected running times. Our library has two benefits over Danielsson (2008)'s. First, it allows programmers to write idiomatic code without embedding invariants in data types, so we can reason about a wider variety of programs. Second, and more significantly, our monad adds no complexity computations to the extracted OCaml code, so it has no verification overhead on running time. We elaborate these details and differences throughout the paper and, in particular, in section 8.

The rest of the paper is structured as follows. In section 2, we give an overview of how the library works and the style of proofs we support. In section 3, we discuss the cost model our proofs deal with. In section 4, we explain the extraction of our programs to OCaml. In these first three sections, we use a consistent example that is introduced in section 2. Following this preamble, section 5 walks through the definition and design of

the monad itself. Section 6 describes the results of our case study, wherein we proved properties of a variety of different functions. Section 7 discusses accounting for the runtimes of various language primitives. Finally, section 8 provides a detailed account of our relation to similar projects. Our source code and other supplementary material is available at `http://github.com/rfindler/395-2013`.

2 Overview of Our Library

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 explicitly 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, which provide for efficient growable vectors, 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`.

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 special notation: `{! «result id» !:! «simple type» !<! «running time id !>! «property» !}` where the braces, exclamation marks, colons, less than, and

```
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 => t' <- insert j t;
                       += 1; <== (bt_node i t' s)
  end.
```

Figure 1: Braun tree insertion

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 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 simplify proving that 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. 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 on empty input.

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, since we know that the tree is not empty, we can replace `n` with `s_size + t_size + 1` and simplify to arrive at this goal:

```
forall i j s t bt an s_size 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 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.

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. Even worse, making mistakes when writing `+=` expressions can produce un-provable claims or cause our proofs about the running times to be useless, as they will prove facts that are irrelevant 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. The user must define this function separately and the translation's output must be used in that context:

```
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 =>
   t' <- insert j t;
   <== bt_node i t' 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 =>
      t' <- insert j t;
      += 9;
      <== (bt_node i t' s)
  end.
```

Figure 2: Inserting += into insert

```
Definition insert_time n := 9 * fl_log n + 6.
Definition insert_result (A : Set) (i : A) (b:bin_tree) (res:bin_tree) 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 uses `SequenceR` (a proposition we wrote) to insist 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 list.

Rather than develop a novel, and potentially controversial cost semantics, we show the utility of our monad by adopting the Rosendahl (1989) cost model. This model treats each function call, variable lookup, and case-dispatch as a single unit of abstract time. In figure 2, the first return is annotated with a cost of 6 because it references 4 variables, calls 1 function, and does 1 case-dispatch. The second return is annotated with a cost of 9 because it references 6 variables (the self-reference is not counted), calls 2 functions, and does 1 case-dispatch.

Our translation function is straightforward and is included in the supplementary materials (`add-plusses/check-stx-errs` in `rkt/tmonad/main.rkt`). Our monad could support different cost semantics, without modification, provided a function could map them to the program's syntax in a straightforward way.

An alternative approach would be to follow Danner et al. (2013) and build a Coq model of a machine and programming language. We would then define a cost judgement for this machine and prove its soundness with respect to the machine's reduction lengths. Finally, we would show that our monadic types allow incremental proofs of their cost results. In some sense, this "deep embedding" would be a more direct study of cost and cost proofs, but it would be no more directly connected with the running time of the programs, unless we could establish a connection to the OCaml VM.

4 Extracting the `insert` Function

One of the important benefits of our library is that none of the correctness conditions and running time infrastructure affect 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)
```

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.

5 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 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` related to `a` by `P`. The intention is to think of the natural number as the running time and `P` as some specification of running time (and possibly also correctness) specific to the particular function. Importantly, the right-hand side of this pair is a proposition, so it contributes no computational content when extracted into OCaml. To see this in practice, consider `insert`'s result type:

```
: {! 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 necessarily connected to the value or the runtime. Therefore, when we know an expression has the type `C A P`, we do not know that its running time is correct, because the property might be about anything and the proof might 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 monad's defining module. We build a set of operations that can be combined in arbitrary ways but subject to the restriction that the `nat` must actually 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 2). This is the type[3] 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: `inc` that adds to the count of the running time, and `bind` that combines two computations in the monad, summing their running times. We tackle `inc` next.

Suppose a program returns a value `a`, with property `P`, that takes exactly one step to compute. We represent such a program with 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: what if the `P` for the `ret` were different than the `P` 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`. But, 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 `A`s and `nat`s, 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.

The monadic operator `inc` encapsulates this logic and introduces `k` units of cost:

[3] The definition of `ret`, and all other monadic operations, are in the supplementary material and our public Github repo. The types are the most interesting part, however, so we focus on them.

```
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 the property for `A` is `PA` and `PB` for `B`, then an 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 cost perspective, 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 6 could not be written with this version of bind, although some could.

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.
```

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 extractions 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). Our versions of the monad law proofs use an auxiliary relation, written sig_eqv, rather than equality. This relation ensures that the values returned by monadic commands are equal and that their proofs are equivalent. In practice, this means that although the theorems proved by expressions such as (m >>= (\x -> f x >>= g)) and ((m >>= f) >>= g) are written differently, they imply each other. In particular, for that pair of expressions, one proves that (n_m + (n_f + n_g)) is an accurate prediction of running time and the other proves that ((n_m + n_f) + n_g) is an accurate prediction of running time, which are equivalant statements.

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.

6 Case Study

To better understand how applicable our monad is, we implemented a variety of functions: search and insert for red-black trees, insertion sort, merge sort, both the naive recursive version of the *n*th Fibonacci number function and the iterative version, a function that inserts *m* times into a list at position *n* using both lists and zippers, BigNum add1 and plus, and all of the algorithms mentioned in Okasaki (1997)'s paper, *Three Algorithms on Braun Trees*. We chose these algorithms by first selecting Okasaki's papers, because the project originated in a class and we knew Okasaki's paper to be well-written and understandable to undergraduates. From that initial selection, we moved

to an in-order traversal of Cormen et al. (2009) looking for functional algorithms that would challenge the framework.

To elaborate on the Braun tree algorithms, 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 of 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`: converts a list into a Braun tree (two n log n and a linear version).

In total, we implemented 19 different functions using the monad. For all of them, we proved the expected O running times. For merge sort, we proved it is $\Theta(n\log(n))$. For the naive `fib`, we proved that it is Θ of itself, $O(2^n)$, and $\Omega(2^{n/2})$, all assuming that the addition operation is constant time. For the iterative `fib`, we prove that it is $O(n^2)$. For the list insertion functions, we prove that when m is positive, the zipper version is O of the list version (because the zipper version runs in $O(m+n)$ while the list version runs in $O(n*m)$.) For BigNum arithmetic, we prove that `add1` is $O(\log(n))$ and that `plus` is $\Theta(\log(n))$. In all cases, except for `make_array_linear` and red-black tree insertion, the proofs of running time include proof of correctness of the algorithm. Finally, in the proofs for BigNum arithmetic and about the Fibonacci functions, we use a simplified cost model that reduces all `inc` constants to 1. The supplementary material contains all of the Coq code for all of the functions in our case study.

6.1 Line Counts

Our supplementary material contains a detailed account of the lines of Coq code produced for our study. We separate the line counts into proofs that are inside obligations (and thus correspond to establishing that the monadic types are correct) and other lines of proofs. In total there are 12,877 lines of code. There are 5,328 lines that are not proofs. There are 1,895 lines of code in obligations and 5,654 lines of other proofs.

We have built a library of general proofs about the monad (such as the monad laws), an asymptotic complexity library, a Log library, and some common facts and definitions about Braun trees. This library accounts for over 25% of the code of each category.

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 an asymptotic one in terms of O.

For example, Figure 3 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 `copy_log_sq` itself, not as the product of the logs:

```
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 3: copy_log_sq

```
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 allows a straightforward proof that copy_log_sq's running time is copy_log_sq_time, but leaves as a separate issue the proof that copy_log_sq_time is $O(\log^2 n)$. 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 $O(\log^2 n)$.

For simple functions (those 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.

6.2 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, 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 one from a given number. In the third category are functions that "skip" over some of their inputs. For example, some 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 2.

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 automatically removed by the OCaml compiler. This transformation into the single conditional corresponds to modest inlining, since `f0` 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 (x0, xs) -> Cons (x0, (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`, `add1`, `tplus`. 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.

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 is the right behavior. 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 we can assume amortized constant time (but not something our framework handles).

More interestingly, and more often overlooked, however, are numeric primitives. In a language implementation with BigNums, integers are generally represented as a list of

digits in some large base and grade-school arithmetic algorithms implement the various operations. Most of these operations do not take constant time.

If we assume that the base is a power of 2[4], then 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 take 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. We did not account for the time of additions in the recursive implementation of `fib`. We have proved, however, that the iterative `fib` function, which requires linear time when additions are not counted, requires quadratic time when we properly account for primitive operations.

Our implementation of addition has a run time that is linear in the number of bits of its input. Using this fact, we can prove that iterative `fib` has a run time that is asymptotic to the square of its input. To prove that `fib`'s run time is bounded below by n^2, we first observe that for all $n \geqslant 6$ we have that $2^{n/2} \leqslant fib(n)$. In the nth iteration of the loop, `fib` adds numbers with $\frac{n}{2}$ bits in their binary representation, which takes time on the order of $\frac{n}{2}$. For large enough n, this implies that the run time of the additions in the iterative `fib` function are bounded below by $\frac{1}{2}(6+7+\cdots+n)$. This sum has a quadratic lower bound. Since the other primitives used in calculating `fib` run in constant time, the run time is dominated by the addition operations, thus the run time of `fib` is bounded below by a factor of n^2.

A similar argument shows that the run time of `fib` has a quadratic upper bound. Combining these two results proves that the run time of the iterative version of `fib` is asymptotically n^2 when we account for primitive operations. The supplementary material contains proofs of these facts in Coq (`fib/fib_iter.v`).

Although our analysis of `fib` properly accounts for addition, it does not consider all language primitives. Specifically, the above analysis of the `fib` function ignores the subtraction that occurs in each iteration of the loop. For example, in the extracted OCaml code for `fib`, pattern matching against `S n` becomes a call to the `pred_big_int` function. Subtracting 1 from a number represented in binary requires more than constant time; `sub1` may need to traverse the entire number to compute its predecessor.

[4] This is the case if BigNums are represented as lists of bits

To be certain that the non-constant run time of `sub1` does not affect our calculation of `fib`'s run time, we argue that the implicit subtractions in the implementation of `fib` take amortized constant time. Although subtraction by 1 is not always a constant time operation, it does require constant time on half of its possible inputs. On any odd number represented in binary, subtracting by 1 is a constant time operation. It follows that any number equivalent to 2 modulo 4 will require 2 units of time to perform the `sub1` operation because `sub1` will terminate after two iterations. In general, there is a $\frac{1}{2^n}$ chance that `sub1` terminates after n iterations. To account for all uses of `sub1` in the implementation of `fib`, we note that we will perform the `sub1` operation on each number from 1 to n. This gives a cost in terms of the iterations required by `sub1` that is bounded above by $n * (\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{n}{2^n} + \cdots)$. One can show that this infinite sum converges to $2 * n$, thus for a sequence of n `sub1` operations this shows that subtraction implicit in the definition of `fib` requires amortized constant time. Overall, the run time of the additions performed by `fib` will dwarf the time required by subtraction. This justifies the fact that we do not explicitly consider the time taken by `sub1` operations.

Although we can account for the recursion pattern using `sub1` described above that counts down from `n` to `0`, there are several other recursive uses of `sub1` found in our library. For example, our implementations of `copy2` and `copy_insert` loop by subtracting `1` then dividing by `2`. As for `fib`, we have not explicitly accounted for these other uses of `sub1`. We do, however, believe that the overhead of using `sub1` in these functions does not change their asymptotic complexity. Appendix A presents an informal argument in support of this claim.

8 Related Work

The most closely related work to ours is Danielsson (2008), which presents a monad that carries a notion of abstract time. Unlike our monad, his does not carry an invariant – in our terms his construction does not have the `P` argument. In our opinion, figuring out the design of monad operations that support the `P` argument is our major technical advance. Accordingly, his 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. Of course, one can bake the Braun invariants into the Braun data-structure itself, which would provide them to his monad via the function arguments, but this restricts the way the code is written, leaves residue in the extracted code, and moves the implementation away from an idiomatic style. Also, his monad leaves 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 the right moments. However, the state used in their monad has computational content and thus is not erased during extraction.

Charguéraud (2010) and Charguéraud and Pottier (2015)'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.

Similarly, others have explored different approaches for accounting for various resource bounds and costs, but we do not provide any contribution in this area. Instead, we take an off-the-shelf cost semantics (Rosendahl (1989)'s) and use it. We believe our approach applies to other cost models.

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. However, they are not technically monads for two reasons. First, the monad laws are written using an equality, but we use an equivalence relation appropriate to our type. Second, our types have more parameters than the single parameter used in monads, due to the proof information residing in the types, so our "monad" is actually a generalized form of a monad, a specialization of Atkey (2009)'s or Altenkirch et al. (2010)'s. Swierstra (2009) and Swamy et al. (2013) follow this same evocative naming convention.

Our code uses Sozeau (2006)'s `Program` facility in Coq for writing dependently-typed programs by separating idiomatic code and detail-oriented proofs in the program source. Without `Program`, our programs would have to mix the running time proofs in with the program, which would greatly obscure the code's connection to the original algorithm, as one does in Danielsson (2008).

We have experimented with supporting proofs about imperative programs by combining our monad's types with a variation of the Swierstra (2009) and Swamy et al. (2013) monads. The types and proofs work out, but are considerably more complicated, due in part to the complexity of proofs about imperative programs. We consider it future work to study whether there is a more elegant approach and develop a detailed case study.

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. Foundations of Software Science and Computation Structure*, 2010.

Robert Atkey. Parameterised Notions of Computation. *JFP* 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.

Arthur Charguéraud and François Pottier. Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In *Proc. ITP*, 2015.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms (3rd Edition). MIT Press, 2009.

Karl Crary and Stephanie Weirich. Resource bound certification. In *Proc. POPL*, 2000.

Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proc. USENIX Security Symposium*, 2003.

Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proc. POPL*, 2008.

Norman Danner, Jennifer Paykin, and James S. Royer. A Static Cost Analysis for a Higher-order Language. In *Proc. Workshop on Programming Languages meets Program Verification*, 2013.

Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proc. ESOP*, 2004.

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

Jan Hoffmann and Zhong Shao. Automatic Static Cost Analysis for Parallel Programs. In *Proc. ESOP*, 2015.

Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. POPL*, 2003.

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

Chris Okasaki. Three Algorithms on Braun Trees. *JFP* 7(6), 1997.

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

Matthieu Sozeau. Subset Coercions in Coq. In *Proc. TYPES*, 2006.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying Higher-order Programs with the Dijkstra Monad. In *Proc. PLDI*, 2013.

Wouter Swierstra. A Hoare Logic for the State Monad. In *Proc. TPHOLS*, 2009.

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

Hongwei Xi. Dependently Typed Data Structures. In *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. POPL*, 1999.

A Appendix

Although we have proved the asymptotic time complexity of a number of functions, we have not always properly accounted for language primitives or implicit subtractions that arise from pattern matching over natural numbers. In the context of the `fib` function, as discussed in section 7, we argue that the primitive operations that we do not explicitly consider do not affect our calculations of asymptotic complexity. Overall, our case studies present five classes of recursion patterns, including that of `fib`, involving language primitives whose running times we have not accounted for within our monadic framework. In this section we present an informal argument to justify the claim that these uses of language primitives have no affect on our complexity analysis. For the first three classes of recursion patterns we present informal proofs and for the remaining two we discuss their complications and suggest directions that may lead to formal proofs.

A.1 The First Category: Linear Addition and Subtraction

The first of category of recursion patterns includes functions that perform a single `add1` or `sub1` operation in each recursive call that count up or down in a completely linear fashion. This pattern appears in the iterative `fib` function as discussed in section 7, where we argue that performing `sub1` on each number from n down to 0 takes linear time for the entire sequence of `sub1` operations and thus constant amortized time for each individual `sub1` operation. The `add1` operation is dual to `sub1`; on every even number adding 1 is a constant time operation whereas subtracting 1 is constant time on odd numbers. Therefore the same argument we apply to show that `sub1` takes amortized constant time applies to `add1`.

A.2 The Second Category: Operations on Braun Trees

Examples from the second category include functions that operate on Braun trees, such as `size_linear` and `size_log_sq`. In `size_linear`, reproduced below, the addition $lsize + rsize + 1$ is not obviously a constant time operation. The Braun tree invariant, however, allows for an efficient implementation of this addition. The invariant requires either that $lsize = rsize$ or $lsize = rsize + 1$. In the former case, the addition corresponds to doubling followed by adding 1. If numbers are represented by lists of bits with least significant bits at the front of the list, then this corresponds to consing a 1 onto the front of the list. In the second case, the addition is equivalent to doubling $lsize$, which can be implemented by consing a 0 onto the front of the list of bits. In either situation the addition operation can be transformed into a constant time operation and therefore does not invalidate our calculation of the running time of the `size_linear` function.

```
Program Fixpoint size_linear (bt:@bin_tree A)
: {! res !:! nat !<! c !>!
     size_linear_result bt res c !} :=
  match bt with
    | bt_mt =>
```

```
      += 3;
      <== 0
    | bt_node x l r =>
      lsize <- size_linear l;
      rsize <- size_linear r;
      += 10;
      <== (lsize + rsize + 1)
  end.
```

Similarly, the `size` function for Braun trees, reproduced below, performs an addition in each recursive call that is not tracked within the monad. Performing the sum, $1 + (2 * m) + zo$, can not always be replaced by a constant time operation. The Braun tree invariant, however, guarantees that zo is either 0 or 1 because it is the difference in size between the left and right subtrees of a Braun tree. Therefore, in the worst case, evaluating $1 + (2 * m) + zo$ requires time proportional to $\log m$. Evaluating `diff s m` to compute zo also requires time proportional to $\log m$. Therefore, ignoring the time complexity of the addition operation does not affect our analysis of the `size` function's running time.

```
Program Fixpoint size {A:Set} (b:@bin_tree A)
: {! res !:! nat !<! c !>!
     size_result A b res c !} :=
  match b with
    | bt_mt =>
      += 3;
      <== 0
    | bt_node _ s t =>
      m <- size t;
      zo <- diff s m;
      += 13;
      <== (1 + (2 * m) + zo)
  end.
```

A.3 The Third Category: Subtraction and Division Together

The copy functions, such as `copy_log_sq`, exhibit a more complicated recursion pattern. These functions apply two primitives for each recursive call, subtraction by 1 and division by 2. It is not obvious that this combination of operations is safe to ignore in run time calculations because whereas `div2` is a constant time operation, subtracting by 1, as we have already seen, is not.

```
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.
```

We argue by strong induction that for any binary number, if we perform a sequence of `sub1` and `div2` operations, the running time of the combination of the two operations is amortized constant time. More strongly, we claim that the total runtime of performing `sub1` and `div2` operations on a binary number b until we reach 0 is $3n$, where we count iterations of `sub1` and `div2` as a single unit of time and n is the number of bits in b.

For the proof, consider a binary number b. If b is zero the result is trivial. If b is odd then there exists some $b' < b$ such that $b = 2 * b' + 1$. As a list of bits, b is represented by a 1 followed by the bits in b'. We write this representation as $b = 1 \cdot b'$ to make the lower order bits, upon which subtraction and division operate, explicit. Performing a sequence of `sub1` and `div2` operations on $b = 1 \cdot b'$ takes 2 units of time (1 each for `sub1` and `div2`) to reduce to b' plus the time to perform the sequence of operations on b'. By induction, we have that performing `sub1` and `div2` operations on b' will take at most $3 * (n-1)$ units of time. Adding these together, the sequence of operations takes no more than $3n$ units of time in total.

In the even case, for a non-zero binary number b of n bits, the list representation of b must begin with some number, k, of zeros followed by a 1 and then the representation of some smaller binary number. Therefore, there exists a b' such that $b = 0 \cdots 0 \cdot 1 \cdot b'$ with $k \leqslant n$ zeros at the front of the number. Subtracting 1 from this number takes $k+1$ units of time, therefore one combination of subtraction and division takes $k+2$ units of time and results in a number of the form $1 \cdots 1 \cdot 0 \cdot b'$ with $k-1$ ones at the front of the list. It is clear that the next $k-1$ iterations will each take 2 units of time. Thus, to reduce to the number $0 \cdot b'$ of length $n-k$ takes $3k$ units of time. Finally, applying the induction hypothesis to the smaller number $0 \cdot b'$ completes the proof.

This proof shows that repeatedly subtracting by 1 and dividing by 2 in each recursive call and terminating at 0 requires time that is linear in the number of recursive calls. Therefore, each use of subtraction followed by division takes amortized constant time in functions such as `copy_log_sq`, and ignoring these primitive operations does not affect our analysis of their running time.

A.4 The Fourth Category: Branching with Subtraction and Division

The fourth problematic recursion pattern appears in the implementation of `diff`, reproduced below. In the body of the last pattern match, (`bt_node x s t, S m'`), the function branches on the parity of its input, m, and if the input is even subtracts 2 then divides by 2, in the odd case we see the recursion described above of subtracting 1 then dividing by 2. Clearly, if control flow never reaches the even case then these operations are constant time and we may safely ignore them. If the evaluation of `diff` does reach

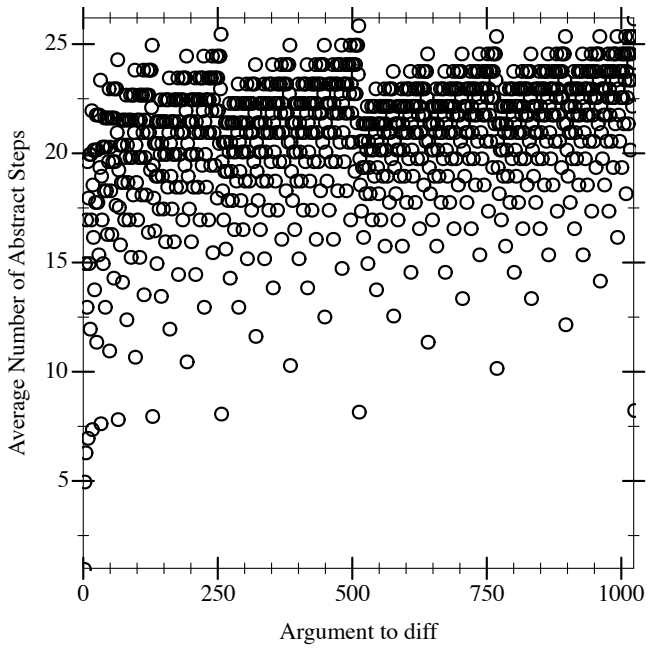

Figure 4: Average running time of `sub1` and `div2`

the even case, however, then we must be certain that the subtraction and division operations do not change our analysis. Subtracting 1 twice from an even number takes logarithmic time in the worst case. The first subtraction may traverse the entire number, but the second subtraction is from an odd number and takes constant time. Figure 4 presents a plot of the average[5] amount of abstract time required by subtraction and division in each recursive call of `diff`. Although the graph only extends from 0 to 1024 this pattern extends to larger numbers as well. The plot suggests that primitive operations used by `diff` require only amortized constant time and suggest that a proof of this claim should be possible.

```
Program Fixpoint diff {A:Set} (b:@bin_tree A) (m:nat) {measure m}
: {! res !:! nat !<! c !>!
     diff_result A b m res c !} :=
  match b, m with
    | bt_mt, _ =>
      += 4;
      <== 0
    | bt_node x _ _, 0 =>
      += 4;
      <== 1
    | bt_node x s t, S m' =>
      if (even_odd_dec m)
      then (o <- diff t (div2 (m' - 1));
            += 13;
            <== o)
      else (o <- diff s (div2 m');
            += 11;
            <== o)
  end.
```

A.5 The Fifth Category: A Tree of Subtraction and Division

Finally, the most complicated pattern is that used by `copy_linear`, which recursively calls itself on `n/2` and `(n-1)/2`. Figure 5 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. Proving that the uses of `div2` and `sub1` in this function contribute only a linear factor to the overall runtime is a significant challenge. Comparing to our proof that the primitive operations in functions like `copy_log_sq` which deals with a linear sequence of operations, a proof for the primitive operations in `copy_linear` must consider a tree of all possible sequences of the operations that evaluate `n/2` and `(n-1)/2`. A similar proof should be possible with the insight that each expensive computation of `(n-1)/2` takes the same number of operations to reach the next expensive computation regardless of the path taken down the tree, however, we have not attempted a formal proof of this claim.

[5] The average here is the toal amount of abstract time used by the primitive operations in a call to `diff` divided by the number of recursive calls.

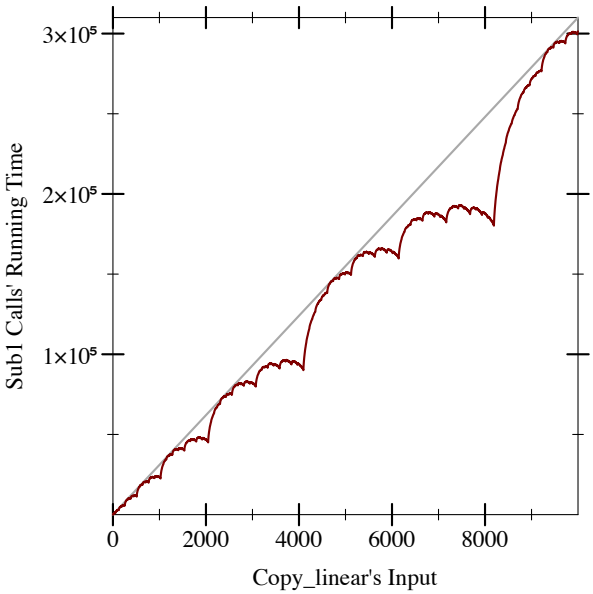

Figure 5: Running time of `copy_linear`

```
Program Fixpoint copy_linear {A:Set} (x:A) (n:nat) {measure n}
: {! res !:! bin_tree !<! c !>!
     copy_linear_result A x n res c !} :=
  match n with
    | 0 =>
      += 3;
      <== bt_mt
    | S n' =>
      l <- copy_linear x (div2 n);
      r <- copy_linear x (div2 n');
      += 14;
      <== (bt_node x l r)
  end.
```

A.6 Conclusion

The informal analysis presented above suggests that, although we have not accounted for all language primitives, our calculations of asymptotic run times remain unchanged. We have presented arguments that support that it is safe to ignore certain uses of language primitives, providing proof where possible and suggesting directions for more formal arguments in the remaining cases.