Compiling Away Recursion for Hardware

Jared R. Pochtar Stephen A. Edwards
Columbia University
jpochtar@gmail.com sedwards@cs.columbia.edu

Abstract

To provide a superior way of coding, compiling, and optimizing parallel algorithms, we are developing techniques for synthesizing hardware from functional specifications. Recursion, fundamental to functional languages, does not translate naturally to hardware, but tail recursion is iteration and easily implemented as a finite-state machine. In this paper, we show how to translate general recursion into tail recursion with an explicit stack that can be implemented in hardware. We give examples, describe the algorithm, and argue for its correctness. We also present experimental result that demonstrate the method is effective.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Algorithms

Keywords recursion, tail recursion, hardware synthesis

1. Introduction

Parallel programming has been a continual challenge to computer science, in part because of the seductive simplicity of sequential behavior coupled with its highly efficient use of hardware resources presents such an attractive approach. Unfortunately, nearly a decade ago, processor architects hit a power wall and ceased being able to deliver faster sequential processors. Parallelism is the only alternative.

Despite years of research, we still do not have a widely accepted, successful architecture for parallel processors. Symmetric multiprocessors with shared memory and coherent caches is a current favorite, largely because it provides an incremental path, but few think they will be the long term solution.

As part of a broader project that aims to harness the ideas of functional programming for providing a better way to specify parallel algorithms, we are developing ways to implement functional programs in hardware. The functional approach has the advantage of being much higher-level than traditional imperative languages. In its pure form, its immutable data structures provide an attractive high-level model of data that can avoid the costly parallel coherency problem, an observation made by Dennis [4], among others.

While most of the lambda calculus is easy to translate into hardware [2, 8], true recursion is difficult because it implies a kind of resource sharing that circuitry does not naturally implement. Yet many interesting algorithms, especially those for graphs, are fundamentally recursive. Not supporting recursive programming is not viable; we need some way of implementing recursion in hardware.

In this paper, we present an algorithm that transforms a functional program with true recursion into an equivalent one that relies only on tail recursion. Tail recursion is simply iteration and can be easily implemented in hardware. We effectively employ the standard technique of pushing activation records onto a stack, but express them with algebraic data types—standard in modern functional languages and fairly easy to translate into hardware.

Ghica et al. [5] attack a similar problem, but take a different approach. They, too, synthesize recursive algorithms in hardware, but start instead from an imperative language. Their solution is to replace all the local registers in a function with stacks of registers, each of which is selected by the stack pointer for the function. By contrast, our technique does not treat recursion as a special case: we reduce it to standard types and objects in our formalism. As such, our approach parallels that of Reynolds [9], who shows how to transform a rich functional representation into a subset that is straightforward to implement. Danvy et al. [3] has also applied this idea extensively. Our approach was also inspired by continuation-passing style and the approach to expressing sequential computation in a lambda calculus setting promoted by Appel [1].

Our algorithm transforms recursive code by passing around a stack of activation records with return addresses (effectively continuations) expressed as algebraic data types in a continuation-passing style. As part of its operation, our algorithm combines groups of mutually-recursive functions into a single function with a leading conditional that directs incoming calls to the proper code.

Below, we introduce our algorithm through an example, describe our algorithm and its operation in detail, argue for its correctness, and finally present some experimental results that show our algorithm works.

2. An Example

Figure 1 illustrates the operation of our algorithm on a naïve Fibonacci number generator that calls itself recursively at two sites. This example is shown in Haskell; it should be possible to adapt our algorithm to any functional language with algebraic data types.

Starting with a recursive function (Figure 1a), our algorithm first schedules the code into a linear representation (Figure 1b) that will enable it to later slice the code into segments between call sites. These segments, which run after a function would normally return, will become continuations. Subexpressions—nested *let*s, function parameters, and the scrutinees of *case* expressions—are pulled out and replaced with fresh intermediate variables. Also, parallel *let* constructs (i.e., that bind multiple expressions) are also made sequential, lest one or more of them make recursive calls.

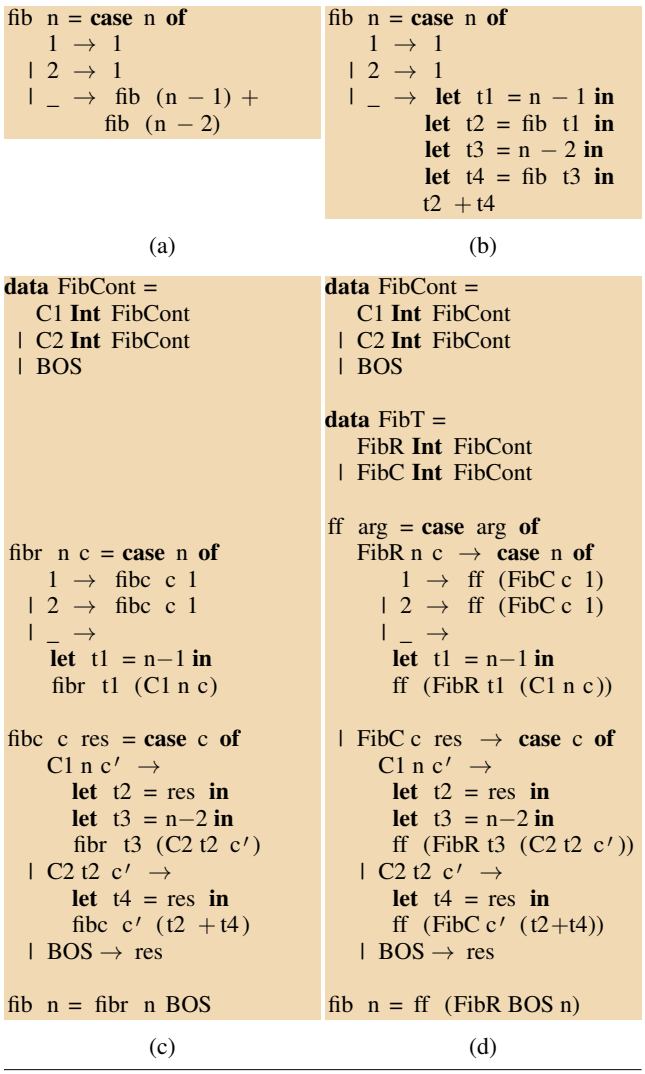

Figure 1. Our algorithm applied to the recursive Fibonacci algorithm expressed in the Haskell functional language. The starting point (a) is linearized (b); split into segments at recursive calls and divided into "recurse" and "continue" functions (c); and finally merged into a single tail-recursive function.

In Figure 1b, our algorithm introduced temporaries t1, t2, t3, and t4 and transformed the body of one of the cases into a cascade of *let*s. Although there are other ways of scheduling this code, we have not yet considered optimizing this procedure.

Next, our algorithm splits the recursive function into two pieces: *fibr*, a "recurse" function that contains the first segment of the original function (from its entry point to the first recursive calls) and *fibc*, a "continue" function that contains all the other segments—those immediately following any recursive calls (Figure 1c).

Our algorithm adds a CPS-style continuation parameter *c* to the *fibr* and *fibc* functions, which relieves them from ever having to return by instead passing them explicit continuations.

The *fibc* function executes continuations. Although continuation-passing style typically expresses continuations as Lambda expressions, we do not, in part because it is not clear how to directly implement them in hardware, but also because we know exactly which continuations are possible. Instead, *fibc* pattern matches on its continuation parameter and executes the code that would otherwise have been present in the corresponding Lambda expression.

The second parameter to *fibc*, *res* is the result from the function call that would otherwise have been passed as an argument to the continuation, had it been expressed as a Lambda expression. Note that the two recursive calls of *fib* in Figure 1b bind their results to *t2* and *t4*—exactly where the various branches of *fibc* bind the *res* argument.

In this second step, our algorithm also introduces the *FibCont* data type, which encodes the three possible "return addresses" for the *fib* function: *C1*, the code immediately after the first recursive call (just before t2 is bound); *C2*, the code immediately after the second recursive call (just before *t4* is bound), and *BOS* ("bottom of stack"), meaning the final result has been computed and *fib* should return to the function that called it from outside.

The *FibCont* type represents a stack of continuations/activation records. All but *BOS* contain a single recursive reference to *FibCont* type, which gives it the structure of a list. Each record holds the variables that are live across the corresponding call site: *n* across the first (used to compute *t3*) and *t2* (the result of the first call) across the second.

The final step (Figure 1d) merges the *fibr* and *fibc* functions into a single function (*ff*) that is exclusively tail-recursive. Another type, *FibT*, is introduced that encodes calls to either the initial segment (*fibr*) or later segments (*fibc*).

3. Our Algorithm

Our algorithm starts with any set of functions, including recursive functions, tail recursive functions, and one or more subsets of mutually recursive functions, that do not contain lambda expressions. It produces a set of functions that includes a function with a matching name and semantics for each input function, but all of the produced functions' recursions are tail-self-recursions.

As a first step, it merges each group of mutually recursive functions into one self-recursive function per group. The result is a set of functions that are, at most, simply recursive.

Then, for each recursive function, our algorithm makes it tail-recursive by transforming it in several steps. First, we transform the function into a linear structure. Next, we extract *case* constructs that include recursive calls into separate functions. Then, we add CPS-style continuations. After that, we split apart the linear structure at recursive call sites, adding their continuations to the CPS-style machinery. Finally, we once more use our machinery for combining functions to merge the recursion, continuation, and any case fragments we created in the second step. The recursive function we finally generate is only tail recursive.

Below, we describe this process in detail, illustrating each transformation with an abstract example.

3.1 Combining a Group of Mutually Recursive Functions

In this section, we describe how our algorithm merges groups of mutually recursive functions into a single function. Our algorithm performs this operation twice: once at the beginning to ensure the remainder of our algorithm only need consider simply recursive functions and finally after we have broken a recursive function into continuations to deliver a result that is only tail-recursive.

The first step in combining a group of functions is to normalize them so they each return an object of the same type. To do this, our algorithm creates a sum type that combines all their return types. Next, each function is made to return an object of this type. Corresponding "unwrappers" are generated for each component function which call the transformed function and unwrap the returned sum type.

To illustrate, if "f" and "g" are two functions that return types "t2" and "t4,"

```
f :: t1 → t2
f x = ... g a ... k

g :: t3 → t4
g y = ... f b ... q
```

our algorithm generates

```
data RT_f_g = RT_f t2 | RT_g t4

f′ :: t1 → RT_f_g
f′ x = ... g a ... (RT_f k)

g′ :: t3 → RT_f_g
g′ y = ... f b ... (RT_g q)

f :: t1 → t2
f x = case (f′ x) of RT_f r → r

g :: t3 → t4
g y = case (g′ y) of RT_g r → r
```

Next, these unwrapper functions (the new "f" and "g") are inlined at their call sites (e.g., f′ and g′) so only the versions of the functions that return the same type are called.

```
data RT_f_g = RT_f t2 | RT_g t4

f′ :: t1 → RT_f_g
f′ x = ... (case (g′ a) of RT_g r → r) ... (RT_f k)

g′ :: t3 → RT_f_g
g′ y = ... (case (f′ b) of RT_f r → r) ... (RT_g q)

f :: t1 → t2
f x = case (f′ x) of RT_f r → r

g :: t3 → t4
g y = case (g′ y) of RT_g r → r
```

After our algorithm normalizes the return types of f′ and g′, they are merged as described below. That step produces wrappers for the f′ and g′ functions, but since these wrappers are only used by the unwrappers we produced here ("f" and "g"), the wrappers for the f′ and g′ functions are inlined and later discarded.

Once our algorithm has ensured that each function in the group of functions has the same return type, it can combine them into a single simply recursive function as described below. This new function takes a single sum-typed argument and pattern matches it to recover the function and arguments that were being called. The sum type has one option for each original function; each option's components are the arguments for the corresponding function. The *case* statement has one branch per original function. It binds the sumtype's components to its function's argument names and sends control to the code for the original function. Wrappers are generated for each of the component functions, which are inlined into the combined function at all call sites. This way, any calls between the component functions become calls to the combined function; all mutual recursion becomes simple self-recursion.

To illustrate, consider a pair of mutually recursive functions "f" and "g." Their arguments are $x_1 \cdots x_n$ and $y_1 \cdots y_m$, whose types are $t_1 \cdots t_n$ and $u_1 \cdots u_m$ respectively. Both f and g must have the same return type t. Assume "f" and "g" call each other recursively with arguments $v_1 \cdots v_m$ and $w_1 \cdots w_m$.

```
f :: t1 → t2 → ... → tn → t
g :: u1 → u2 → ... → um → t
f x1 ... xn = ... (g v1 ... vm) ...
g y1 ... ym = ... (f w1 ... wn) ...
```

Our algorithm introduces a new type "FG" that encapsulates the arguments passed to "f" and "g":

```
data FG = F t1 ... tn
        | G u1 ... um
```

Any calls to the two functions are replaced with calls to a merged function "fg" that encapsulates the arguments to the original function with an object of the "FG" type

```
f x1 ... xn = fg (F x1 ... xn)
g y1 ... ym = fg (G y1 ... ym)
```

Finally, it creates a combined function "fg" that takes a single argument of the "FG" type and moves the bodies of the f and g functions into options for a *case* expression that pattern-matches on the argument.

```
fg arg = case arg of
    F x1 ... xn → ... (fg (G v1 ... vm)) ...
  | G y1 ... ym → ... (fg (F w1 ... wn)) ...
```

3.2 Linearizing

Once any groups of mutually recursive functions have been combined, only self-recursive functions remain, so our algorithm can work on single functions. Our algorithm transforms each non-tail recursive function so that recursive calls only happen in a *let* with a single binding to the result of a recursive call—what we call a recursive-let. We do this because it makes the body of such a *let* exactly the code to be executed after the called function returned—it will become one of the continuations in the third step of our algorithm.

Our algorithm lifts out the scrutinee of any *case* and the argument of any call that recurses, binding the subexpression to a new temporary. It uses a similar technique to break up nested *let*s. The result is a simple sequence of *let*s.

For example, starting from

```
let a = f (g (h i)) in c
```

the inner call of h is lifted out, giving

```
let a = f (let t1 = (h i) in g t1) in c
```

then the argument of f is also lifted to give

```
let a =
  let t2 =
    let t1 = h i in
    g t1 in
  f t2 in
c
```

Finally, nested *let*s are flattened, giving

```
let t1 = hi in
let t2 = g t1 in
let a = f t2 in
c
```

3.3 Restructuring Case Constructs

Even with all recursions occurring in recursive-lets, the continuation (*in* clause) of a recursive-let is not yet always the entirety of the continuation needed after the recursion. Consider a recursion in a branch of a *case* in a *let* binding, such as

```
f x =
  ...
  let y = case x of
    a → s
  | b → f z ... r
  in q
```

Here, since the body of the *let*, "q" is executed after every branch of the *case*, it would be most natural to simply copy it into the continuation that follows "f," but there is a danger that this would lead to a code size explosion.

Instead, we create a case-continuation function for each such "q," which takes the free variables of "q" as arguments. We push the *let* binding into the return expression of the *case*, and call the case-continuation function in its body. The above example becomes

```
f x =
  ...
  case x of
    a → let y = s
        in cfA x y
  | b → f z ...
        let y = r
        in cfA x y

cfA x y = q
```

3.4 Adding Continuations

With the function now normalized, the second step of our algorithm adds a CPS-like continuation parameter to the function, which obviates the need for the function to ever return from a recursive call. Instead, it passes results to explicit continuations. CPS traditionally uses lambdas, which do not have a clear implementation in hardware. Instead, because we will know all of the possible continuations, we use a single "continue" function and pass it the continuation parameter, an algebraic data type that describes which continuation should be taken. Initially, the only continuation type option is the bottom-of-stack label, whose corresponding branch simply returns the value passed to the continuation function:

```
data Cont = BOS
fc continuation result = case continuation of BOS → result
```

Calling the original function, then, represents an additional recursion, as opposed to an additional continuation, so the original function is renamed to be the "recurse" function. This is also in part because the function now takes a different argument list, so calls to it would be incompatible. The original function, maintaining its name and signature, becomes a wrapper around the recurse function:

```
f x = fr x BOS
```

There are two cases for returned expressions, both of which we want to pass through the continuation function (in the new recurse function as well as any case-continuation functions). First, when a simple value is returned (e.g., a bound object or the result of something like an addition), our algorithm replaces it with a call to the continuation function, passing it the continuation and the result as the arguments. The other case is when the result is a tail recursion. By definition, a tail recursion's continuation is its calling function's continuation, so we simply transform the tail call with a tail call to the "recurse" function, with the same arguments, but with the continuation parameter appended as well. The *fibr* function in Figure 1c illustrates both these cases. For the "1" and "2" branches, a simple constant was being returned; now, it calls *fibc* with the passed-in continuation and the constant as arguments. For the tail-recursive case, the recursive call passes the argument (t1) but appends to it a continuation representing the work to be done afterward.

3.5 Extracting Continuations

When splitting the linearized body of the function, the algorithm only has to handle recursive-lets. This is convenient because with recursions in *case*s in *let*s removed, a recursive-let's body precisely represents that recursion's continuation.

Each recursive-let is transformed into a tail call of the recurse function; the body of the recursive-let is placed in the continue function, where it becomes a new branch that corresponds to a new option added to the continuation type. The tail call that replaces the recursive-let has all the arguments of the original call followed by an argument that encodes the continuation—the work to be done after the tail call "returns."

For the body of the *let* being replaced to execute in the same environment, it needs to capture the values of its free variables. Our algorithm identifies which free variables to capture for the continuation and adds them as components to its sumtype option. Similarly, its branch in the continuation function binds those components to the names of the expected free variables, and adds the pre-captured variables as arguments to the continuation label constructor for the new tail recursive call.

In order for the continue function to maintain only tail calls to the recurse function, this transformation needs to happen from the bottom up. It is run not only on the recurse function, but also on any case-continuation functions.

To illustrate, given

```
let v1 = ... in
...
let vn = ... in
let z = f a1 ... an in ... v1 ... vn ...
```

where $v_1, \ldots, v_n$ (of types $t_1, \ldots, t_n$) are variables that occur free in the body of the *let*, we generate

```
type Cont = ....
          | C t1 ... tn
          | BOS
```

The call to the recursive call in the recurse function becomes a tail-recursive call with the continuation saving the free variables tacked on to the end:

```
let v1 = ... in
...
let vn = ... in
fr a1 ... an (C v1 ... vn)
```

Finally, our algorithm adds a matching case to the "continue" function "fc" that executes the code for the continuation:

```
fc c r = case c of
         ...
       | C v1 ... vn →
           let z = r in ... v1 ... vn ...
       | BOS → r
```

3.6 Combining Generated Mutually Recursive Functions

At this point, any recursive calls or calls between the recurse functions, the continue function, and any case-continuation functions are tail calls. However, in many cases (i.e., when the original function has multiple continuations, such as in the Fibonacci example), the recurse function and continuation functions are mutually recursive. If this is the case, we merge them using our algorithm for

combining functions, which we described in Section 3.1. It maintains the tail-ness of calls in its component functions, so when it is run on, for example, the recurse function and continuation function, its resulting combined function is tail-recursive only; see Figure 1d.

4. Correctness

Here, we argue for the correctness of our algorithm. Broadly, our algorithm is a series of semantics-preserving transformations that impose various invariants along the way.

4.1 Intermediate Functions Only Tail Recurse

The intermediate recurse, continue, and any case-continuation functions, although possibly mutually recursive, are tail-recursive-only (TR-only) if the calls between them are considered self recursive, as they are once they are combined. They are TR-only because any non-TR-only (non-TR) forms of expressions in the source functions are transformed to TR-only expressions, as long as their subexpressions are TR-only. Because our transformations are applied as postorder traversals, inductively, the produced functions are TR-only.

The base cases for this induction are variable and literal expressions. These never recursive, so they are TR-only.

A function call is TR-only iff none of its arguments are recursive. Because we linearize the function, a call's arguments are always variables and thus nonrecursive.

Note that the TR-only condition is equally true of a recursive call: iff its argument expressions are recursive, the recursive call is no longer a tail-recursive expression. Otherwise, recursive calls are considered tail-recursive, so they are TR-only. Note that this does not mean that the recursion it produces may not be a non-tail recursion; *case*s and *let*s can be non-TR if certain of their subexpressions are recursive, even tail-recursive.

A *case* is TR-only iff its scrutinee is nonrecursive and all of its branches are TR-only. Because we extract subexpressions, a *case*'s scrutinee must be a variable and is thus nonrecursive. Because we assume each subexpression is TR-only, the branch expressions are all TR-only, which, combined with a nonrecursive scruintee, makes the *case* TR-only.

let b = e **in** body is TR-only if its binding, "e," is nonrecursive and its body is TR-only. Because we assume all subexpressions are TR-only, we assume its body is TR-only. However, if "e" is a recursive expression, TR-only or not, the *let* becomes non-TR, as "body" needs to run after "e." Therefore, a *let* output by the algorithm cannot have a binding that includes a recursive call.

Our algorithm does not leave any recursive expressions in *let* bindings. If it was a nested *let*, it was flattened in the linearization process. If it was a *case*, it was extracted (and its recursive-let's body turned into a case-continuation function). Calls that are nonrecursive are naturally valid. Calls that are recursive, on the other hand, are explicitly taken out of their *let*s in the continuation extracting phase, and their recursive-let's bodies moved into the continue function.

The continue function and any *case* continuation functions are also TR-only. This is because they are constructed by tail-branching to subexpressions taken from the original function. The transformations are postorder, so when the subexpressions is taken, they are TR-only.

4.2 Combined Functions Only Tail Recurse

If each of a set of component functions are TR-only, pretending that calls between them were self recursions, then their combined function is TR-only, as these mutually recursive tail calls now are self recursive in the combined function. This happens because the wrappers are inlined; if they were not, they would be mutually recursive with the combined function, and neither would be TR-only.

The continuation argument passed to the function will always be captured because either the new continuation returns by calling the continuation function with it, or, because the algorithm does this transform in a postorder traversal, the continuation has a tail call with a continuation that itself captures it.

4.3 Semantics are Preserved at Each Step

The function combination, linearization, continuation adding, and continuation extracting operations each maintain semantic equivalence. Therefore this whole algorithm, a composition of those steps, maintains semantic equivalence.

Linearization is a combination of subexpression extraction and *let* flattening. Subexpression extraction is a common, semantics-preserving transformation. *Let* flattening does not modify the bindings or the outermost *let* body. All of the bindings become available to that outermost *let* body. All variable names are assumed unique, so the addition of extra, unused variables to its environment does not affect it.

Continuation addition preserves semantics because to begin with, the continue function is the identity function (only BOS can be passed to it at this point). The original function is moved to a new function, the recurse function, so that the continuation parameter may be added. The original function is rewritten to simply call the recurse function with an additional BOS parameter. Thus, as long as the recurse function, when passed a BOS, has the same semantics as the original function, the function is semantics-preserving. This is true because all returns are passed through the continue function, but with BOS, which makes the continue function the identity function. Thus, the recurse function has exactly the same semantics as the original function, when passed the BOS parameter.

The continuation extracting transformation is semantics preserving because it is effectively a partial CPS transformation. Transforming functions to CPS has been shown to be semantics-preserving [1]. Separately, the case-continuation extraction process is simply lambda abstraction; which is semantically equivalent in the lambda calculus.

The function combination algorithm is semantics preserving because of the simple semantic identities a = **case** B **of** B $\rightarrow$ a and **case** B **of** B $\rightarrow$ a = **case** B **of** B $\rightarrow$ a | X $\rightarrow$ y. Effectively, it transforms component functions from the left hand side of the former to the right, then combines them according to the latter.

5. Discussion

5.1 Generated types

The continuation type (e.g., FacCT) is separate from the combined type (e.g., FacT) because when you call the continuation function, you pass the calling recursion's result as well its continuation. The recursion's returned result is of the return type of the function, so the continuation function's signature is $rt \rightarrow cont \rightarrow rt$, while the function's may be $a \rightarrow cont \rightarrow rt$ where $a \neq rt$, or a could represent multiple arguments. Therefore, the function header (e.g., facR) cannot be treated like a continuation. Furthermore, because the result argument to the continuation function is not known at the time of construction of a continuation, said argument needs to be passed to the continuation function separately from the continuation-typed parameter. This forces the type of the continuation function, which is necessarily separate from the function header. If these are mutually recursive, there must be a new type as per the combine-functions algorithm.

On the other hand, *case* continuations (continuations that are the result of a recursion in a *case* in a *let*) do not count as continuations in the continuation function, and are treated as separate

(potentially mutually recursive) functions. Unlike regular (recursion in a *let* binding-resulting) continuations, the "result" value of these is known as soon as the continuation would be constructed. In fact, we do not construct a continuation value for *case* continuations because they are not needed—they can be called immediately. Furthermore, their "result" value may be of a different type than the return type, so they could not have the type signature of the continuation function. Ultimately, we use combine-functions on *case* continuations as a matter of choice; we could instead simply append the continuations to each branch of the *case*. We choose not to because this would result in duplication of code.

5.2 Stack

There never are multiple continuations available at the same time, so each continuation captures exactly one other continuation; the BOS label captures nothing. Therefore, the continuations form a linked list. However, because they are only deconstructed ("popped" from) once, they behave as a stack. Thus, this algorithm effectively constructs an explicitly passed stack.

5.3 Stackless Tail-Recursive Functions

The given examples, fib and fac, have stackless tail-recursive equivalents, which the algorithm does not generate, because not all recursive algorithms have a stackless equivalent. Fac, for example, can be given as

```
fac′ n a = case n of 1 → a | _ → fac (n − 1) (a ∗ n)
fac n = fac′ n 1
```

which is preferable to what we generate because it is both TR-only and does not need a stack. However, not all recursive functions have a stackless tail-recursive implementation.

Our algorithm is meant to transform any function to an equivalent TR-only one, and some functions require a stack. For example, functions that postorder traverse tree (or equivalent) data structures need to know the parent node of the node they are visiting, something traditionally done by simply returning to the calling instance of the function, whose visited node is the parent of the node recursed on. For example, consider a function that finds the sum of the values of nodes in an n-tree. The activation record of the calling instance of any recursion, maintained on the stack, includes some information of its visited node; thus the stack effectively maintains (at least) a list of parent nodes. Any semantics-preserving transformation of the sum function must include this list of parent nodes in order to know which node to visit next; this list is effectively a stack, one way or another.

5.4 Performance

To test our algorithm, we implemented it in Haskell, operating over GHC Core [7**?**] and producing OCaml and Haskell. It worked for our test cases. We compared the performance of the transformed and untransformed programs, shown in Figure 2. The untransformed programs ran significantly faster than the transformed programs, but the difference varied. O'Caml code performed consistently about 3.7× slower after our transform, and Haskell ran slower still, over a range of 3× to 20×, depending on the input. We did not record the exact memory usage while these tests were running, but we found them to be needlessly high. It seems likely that this excessive use of memory, which certainly overflowed the cache, had a significant effect on the tests. Garbage collection may also have negatively impacted the performance; we did not attempt to measure it.

Clearly, the runtimes and compilers for O'Caml and Haskell handle non-tail recursion more efficiently than they do data structures that happen to be stacks. So although we do not recommend this algorithm be used to modify software, our intention was always

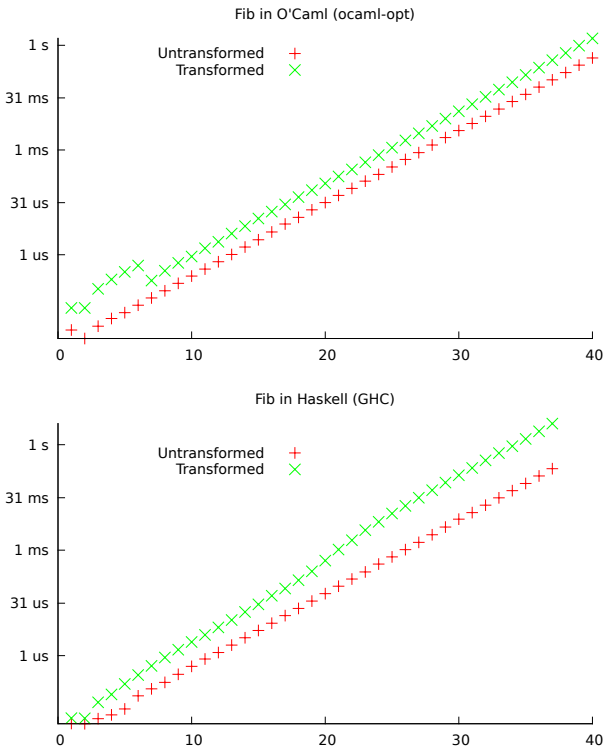

Figure 2. Execution time for various invocations of the *fib* function, before and after our transformation, using the ocaml-opt and GHC compilers.

that the algorithm be applied to restructure code before transforming it into hardware.

The performance results show that explicit stack manipulation in O'Caml and Haskell's runtime systems is less efficient than implicit stack manipulation via function calls. Although this indicates that function calls and returns are efficient in OCaml and Haskell, at least compared to data manipulation, it also suggests that their runtimes could be more efficient when handling explicit stacks. Stacks are not uncommon data structures even outside the call stack and can likely be optimized by the runtime at a lower level than the IR.

It is possible to analyze a program and determine if a data type is used as a stack. Even if the data type is like the list type, which may be used in a non-stack fashion in some locations while treated just like a stack in others, type system analyses could be used to identify objects of that type that are treated as a stack and make them an identical but separate type. One possible check to determine if a type is a stack would be to establish that it is singly recursive, with one or more non-recursive options; that any time an object of that type is unwrapped (i.e. pattern matched, binding its components) it is not later used; and that any object of that type is only once passed to a function, including and especially a constructor, and then not used again. Although these criteria are conservative (a programmer could sneak a stack past them), it would at least identify those stacks produced by our algorithm. It, and others, could speed up the programs produced by the algorithm presented here, as well as many other programs that, for unrelated reasons, explicitly manipulate stacks or stack-like data structures.

6. Conclusion

We presented an algorithm that transforms general recursion in a functional program into tail recursion and argued for its correctness. The purpose of this algorithm is to transform recursive functional code into a form suitable for generating hardware, where tail recursion is easy to implement as iteration but recursion is not obvious. Our algorithm works largely by using ideas from CPS and combining mutually recursive functions when necessary. We saw that in transforming functions to CPS-style TR-only equivalents, we effectively produced an explicit stack.

When tested on the O'Caml and GHC's runtime, we saw that making the stack explicit produced a noticeable performance degradation. While this was somewhat expected, the magnitude of the slow-down suggests that many programs that currently manipulate explicit stacks (i.e., written by programs) could be greatly improved by adding a "stack identification" optimization to existing functional compilers, perhaps in the spirit of Deforestation [6].

In the future, we will use this work as part of a larger Haskell-to-hardware compiler, as this algorithm's removal of non-tail recursion is essential for automatically designing hardware for potentially recursive functions.

Acknowledgments

This work was supported by the NSF under grant CCF-SHF 1162124.

References

[1] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. Cλash: Structural descriptions of synchronous hardware using Haskell. In *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pages 714–721, Lille, France, Sept. 2010. doi: 10.1109/DSD.2010.21.

[3] O. Danvy, J. Johannsen, and I. Zerny. A walk in the semantic park. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 1–12, Austin, Texas, Jan. 2011.

[4] J. B. Dennis. General parallel computation can be performed with a cycle-free heap. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 96–103, Paris, France, Oct. 1998. doi: http://dx.doi.org/10.1109/PACT.1998.727177.

[5] D. R. Ghica, A. Smith, and S. Singh. Geometry of synthesis IV: Compiling affine recursion into static hardware. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 221–233, Tokyo, Japan, Sept. 2011. doi: 10.1145/2034773.2034805.

[6] A. Gill, J. Launchbury, and S. L. P. Jones. A short cut to deforestation. In *Proceedings of Functional Programming Languages and Computer Architecture (FPCA)*, pages 223–232, Copenhagen, Denmark, June 1993.

[7] S. P. Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12:393–434, Sept. 2002.

[8] A. Mycroft and R. W. Sharp. Hardware synthesis using SAFL and application to processor design. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*, number 2144 in Lecture Notes in Computer Science, pages 13–39, Livingston, Scotland, Sept. 2001.

[9] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, 1972. doi: 10.1145/800194.805852. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397 Dec. 1998.