Hardware Synthesis from a Recursive Functional Language

Kuangya Zhai Richard Townsend Lianne Lairmore Martha A. Kim
Stephen A. Edwards
Columbia University, Department of Computer Science
Technical Report CUCS-007-15
April 1, 2015

ABSTRACT

Abstraction in hardware description languages stalled at the register-transfer level decades ago, yet few alternatives have had much success, in part because they provide only modest gains in expressivity. We propose to make a much larger jump: a compiler that synthesizes hardware from behavioral functional specifications. Our compiler translates general Haskell programs into a restricted intermediate representation before applying a series of semantics-preserving transformations, concluding with a simple syntax-directed translation to SystemVerilog. Here, we present the overall framework for this compiler, focusing on the IRs involved and our method for translating general recursive functions into equivalent hardware. We conclude with experimental results that depict the performance and resource usage of the circuitry generated with our compiler.

1. INTRODUCTION

Hardware designer productivity continues to lag far behind improvements in silicon fabrication technology. Designers rely on IP reuse to fully utilize the billions of transistors on today's integrated circuits. Unfortunately, creating, optimizing, and verifying those IP cores remains stuck at the register-transfer level. Decades of research on high-level synthesis have produced some successes [11], but most have been limited to signal processing algorithms and provide only a minuscule improvement in abstraction over RTL: nothing like what software programmers have had for years.

We believe a far more radical approach is necessary. We must raise the abstraction level drastically, going even higher than "high-level" C models that are often considered the cutting edge of input formats for hardware synthesis [5]. To that end, we consider using a functional programming language as an input specification, which confers a host of benefits. The user gains productivity-enhancing abstractions such as recursive functions, algebraic data types, and type inference. The compiler can use the firm, mathematical foundation of a pure functional language to implement sophisticated semantics-preserving optimizations originally developed for software.

In this paper, we present a compiler that transforms programs written in a functional language (Haskell) into synchronous digital circuits, which we then synthesize onto an FPGA for prototyping purposes. The compiler performs a series of semantics-preserving transformations on a restricted dialect of Haskell, ultimately producing a stream-based program that permits a syntax-directed translation to SystemVerilog. We use a single functional intermediate representation based on the Glasgow Haskell Compiler's Core [23], allowing us to use Haskell and GHC as a source language. Reynolds [24] pioneered the methodology of transforming programs into successively restricted subsets, which inspired our compiler and many others [2, 15, 23].

Figure 1 depicts our compiler's framework as a series of transformations yielding a hardware description. One pass eliminates polymorphism by making specialized copies of types and functions. To implement recursion, our compiler gives groups of recursive functions a stack and converts them into tail form. These functions are then scheduled over time by rewriting them to operate on infinite streams that model discrete clock cycles. Finally, the resultant stream program is converted into synthesizable SystemVerilog via a syntax-directed translation.

In this paper, we make the following contributions:

- We propose a Haskell-to-hardware compilation framework based on a series of semantics-preserving transformations that operate on a common functional intermediate representation.
- We show how a stream-based dialect of this IR models synchronous digital hardware and admits a syntax-directed translation into synthesizable SystemVerilog.
- We illustrate how recursive algorithms can be implemented in hardware through the use of our framework.
- We present experimental results that suggest our circuits are practical.

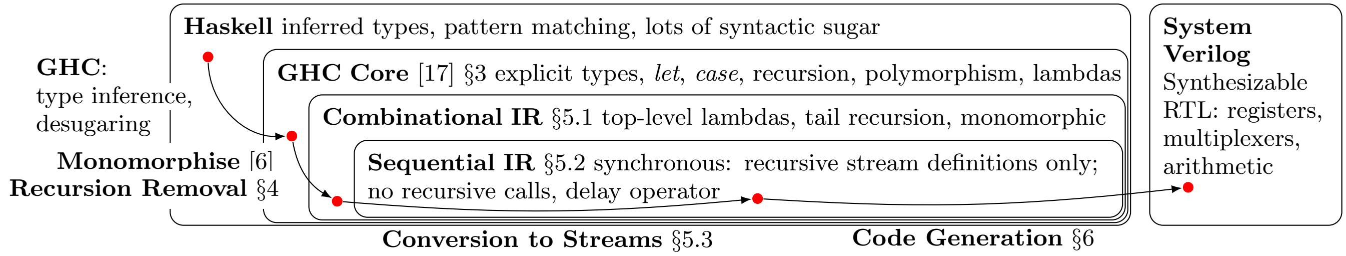


Figure 1: Overview of Our Compilation Flow: We rewrite Haskell programs into increasingly simpler dialects until we can perform a syntax-directed translation into SystemVerilog

2. AN EXAMPLE: FIBONACCI

Our compiler is designed as a sequence of rewriting steps that each lower the source program's abstraction level; the final form enables easy interpretation as hardware. To illustrate the power of this technique, we show how it can remove recursion from the familiar recursive Fibonacci number function. The algorithm itself remains recursive, but the final implementation uses an explicit stack instead of recursive function calls (since the latter cannot be translated into hardware form). Of course, far better algorithms exist for computing Fibonacci numbers; our goal here is to illustrate the recursion removal procedure.

Throughout the paper, we use a pidgin Haskell notation that resembles our IR (Figure 2). The function below compares the integer argument n with constants 1 and 2 to determine whether it has reached a base case, for which it returns 1, or needs to recurse on $n-1$ and $n-2$.

```
fib  n = case n of 1  → 1
                   2  → 1
                   n  → fib  (n−1) + fib  (n−2)
```

Translating this recursive function into hardware is not obvious. The two recursive function calls are the challenge. The usual technique of inlining calls (e.g., typical for VHDL and Verilog RTL synthesis tools) would attempt to generate an infinitely large circuit unless we limited the recursion depth. Interpreting the structure of this program literally would produce a circuit with multiple combinational loops (multiple ones from each recursive call), but it would likely oscillate unpredictably. Inserting registers in the feedback loops would prevent the oscillation, but since this is not simply tail recursion, it is not obvious how to arbitrate between the two call sites. Instead, our compiler restructures this program into a semantically equivalent form that is straightforward to translate into hardware.

Our IR has no side effects, so the evaluation of fib (n−1) and fib (n−2) can occur in any order. To avoid arbitration circuitry, we impose a particular order on them by transforming the function into continuation-passing style [7, 29], or CPS. In CPS, each function is given an extra "continuation" argument (traditionally named "k"): a function called with the result of the computation. For example, the continuation passed to the first recursive call of *fib* names the result of the call n1 and calls fib (n−2).

```
call  n k = case n of 1  → k 1
                      2  → k 1
                      n  → call  (n−1) (λn1  →
                           call  (n−2) (λn2  →
                           k (n1  + n2)))
fib  n = call  n (λx  → x)
```

Here, *call* is a CPS helper function that does the real work: fib invokes *call* with a function that returns the result to the outside world. The structure of *call* now represents the control flow explicitly: recurse on (n−1), then name the result n1, recurse on (n−2) name its result n2, compute n1 + n2, and finally pass the result to the continuation k.

This transformation to CPS has both scheduled the calls of *fib* and transformed the program to use only tail recursion, which can easily be implemented in hardware as a state machine. Two issues remain before we can translate this into hardware: the second continuation references n1, which is defined by the first continuation, not the second; and more seriously, unnamed function (continuations: lambda expressions) are being passed as arguments.

We perform lambda lifting [16] to address these issues: any variables that are not defined within their continuation (here, n, k, and n1) are added as additional arguments and each lambda term is named to become a top-level function.

```
call  n k = case n of 1  → k 1
                      2  → k 1
                      n  → call  (n−1) (k1  n k)
k1 n   k r   = call  (n−2) (k2  n1 k)
k2 n1  k n2  = k (n1  + n2)
k0       x   = x
fib  n = call  n k0
```

Here, k0 is the identity, the k1 function performs the second recursive call, and k2 produces the result by adding n1 to n2. Each of these continuations are passed as partially applied functions, e.g., k1 takes three arguments, but is only given n and k when passed to *call*. The remaining argument is appended when the continuation is actually called.

Control flow is now more explicit, but we continue to pass around partially-applied functions (e.g., (k2 n1 k) only lists two of the three arguments to k2; the third is passed as the result of a call). To eliminate these, we perform defunctionalization [4] by encoding functions passed as arguments in the algebraic data type *Cont* and introducing the helper function *ret* that applies a result to a continuation.

```
data Cont = K0
            K1 Int  Cont
            K2 Int  Cont

call  n k = case n of 1  → ret  k 1
                      2  → ret  k 1
                      n  → call  (n−1) (K1 n k)
ret  k r  = case k of K1 n   k  → call  (n−2) (K2 r  k)
                      K2 n1  k  → ret  k (n1  + r)
                      K0        → r
fib  n = call  n K0
```

Now, we merge *ret* with *call* to create the *next* function. We introduce the algebraic data type *Op* to encode calls *call* and *ret*. This makes *next* nearly a state machine: it pattern matches its argument (the current state) then calls itself iteratively with the next state (an instance of the *Op* data type) after performing some arithmetic.

```
data Cont = K0
            K1 Int  Cont
            K2 Int  Cont

data Op = Call  Int  Cont
          Ret  Cont Int

next z  = case z  of
  Call  1  k           → next  (Ret  k  1)
  Call  2  k           → next  (Ret  k  1)
  Call  n  k           → next  (Call   (n−1)  (K1 n  k))
  Ret  (K1 n    k)  r  → next  (Call   (n−2)  (K2 r  k))
  Ret  (K2 n1  k)  r   → next  (Ret  k  (n1  + r))
  Ret  K0           r  → r
fib  n  = next  (Call   n  K0)
```

This is now much closer to a hardware implementation. The body of *next* is a transition table for a finite-state machine: for each condition on the left, perform some arithmetic and transition to a new state. The *k* argument functions like a top-of-stack: creating a continuation effectively pushes onto the stack (the old top-of-stack, *k*, is linked to this new entry); using a *k* obtained from an existing continuation (e.g., in Ret (K1 n k) r) pops the stack.

In Section 5.3, we will return to this example and describe what remains to generate hardware for this.

3. OUR IR: DERIVED FROM GHC CORE

In this section, we begin a detailed description of our compiler. We use the Glasgow Haskell Compiler to convert a Haskell source program into "external core" format [30], a simplified form of which is shown in Figure 2. Our compiler reads the textual external core format. The Intel labs Haskell research compiler [19] adopts a similar architecture.

A program in our IR defines a collection of algebraic data types. ADTs are a powerful feature of modern functional languages that subsume records, enumerations, and union types. An ADT consists of multiple variants; each variant has a name (a "data constructor": *Dcon-id* in Figure 2) and zero or more fields, each with its own type. Field types are either concrete (composed only of "type constructors" *Tcon-id* that name other ADTs or basic types) or polymorphic (containing one or more type variables *tvar-id*). Any type variable used in a variant must appear as an argument to its type definition.

The familiar Boolean type is an algebraic type with two variants (each with no fields) built into our standard library:

```
data Bool = True
            False
```

Two other common (polymorphic) examples are singly-linked lists and binary trees. Here, the *a*'s represent arbitrary types, allowing for lists of, say, 8-bit integers.

```
data List  a  = Nil
                Cons a  List
data Tree  a  = Leaf  a
                Node (Tree   a)  (Tree   a)
```

program	::=	*type-def** *var-def*$^+$	
type-def	::=	**data** *Tcon-id* (*tvar-id*)* = *variant*$^+$	
variant	::=	*Dcon-id* (*type*)*	
type	::=	*Tcon-id*	
		tvar-id	
		type → *type*	Function
		type type	Type application
var-def	::=	*var-id* **::** *type* = *expr*	
expr	::=	*literal*	
		var-id	
		Dcon-id	
		expr expr	Application
		λ (*vbind*)$^+$ → *expr*	
		let *var-def*$^+$ **in** *expr*	
		case *expr* **of** *case-alt*$^+$	
vbind	::=	*var-id* **::** *type*	
case-alt	::=	*Dcon-id* (*vbind*)* → *expr*	
		literal → *expr*	
		_ → *expr*	Default
Tcon-id	::=	Type constructor (capitalized identifier)	
Dcon-id	::=	Data constructor (capitalized identifier or ▷)	
literal	::=	Integer literal	
var-id	::=	Variable (lowercase identifier or +, −, etc.)	
tvar-id	::=	Type variable (lowercase identifier)	

Figure 2: Abstract Syntax of Our Intermediate Representation: a variant of GHC's Core [17]

Along with Boolean, our standard library provides 8-, 16-, and 32-bit signed and unsigned integers.

Variable definitions include functions and comprise the rest of a program. Each binds a type and an expression to a name. Expressions are terms in the typed lambda calculus; functions are lambda expressions with typed variable bindings e.g. "λ x::Int → + x 1" defines a function that increments its argument, which must be of type Int. We extend this basic calculus with *let*, which introduces local variables, and *case*, which matches algebraic types and integer literals in order. The default pattern "_" matches anything.

To illustrate, the following defines a variable that is a recursive function that computes the depth of a binary tree:

```
depth :: Tree a → Int =
  λ t::Tree a → case t of
                  Leaf → 1
                  Node l::(Tree a) r::(Tree a) →
                    let ld::Int  = depth l
                        rd::Int  = depth r
                    in + 1 (max ld rd)
```

The function uses a *case* to evaluate its argument *t* and select the next step of computation. If *t* is a *Leaf*, the function simply returns 1. Otherwise, *t* must be a *Node* with two sub-trees (that have the same polymorphic type argument *a* as the *Node* itself), in which case the function calls itself recursively on the two sub-trees and adds 1 to the maximum depth it found, using a function *max* that returns the larger of two integer arguments.

Our compiler transforms a program in this IR (which GHC generates from Haskell source) into a simpler form that can be translated into hardware. Next, we present details of how we transform a program in the IR to remove recursion.

4. COMPILING RECURSION

Here, we present the recursion removal procedure we introduced in Section 2. It starts from any collection of functions, which may be tail-recursive, self-recursive, or mutually recursive, and produces an equivalent collection of functions that are at most tail-recursive. This procedure assumes all function calls are explicit, e.g., that the recursive functions being called are named directly rather than being passed around as lambda terms. Similarly, it assumes no partial function applications.

Our procedure operates by identifying groups of mutually recursive functions, merging each group into a single self-recursive function, explicitly scheduling the recursive calls, then splitting apart each function at recursive call sites, inserting continuation control at each point. Finally, we introduce an explicit stack. We describe these steps in detail.

4.1 Combining Mutually Recursive Functions

We begin by combining mutually recursive functions into a single function. We build a static call graph of all the functions in the program and treat each strongly connected component (SCC) as a group of mutually recursive functions to be merged.

Each function in an SCC can have a different return type, so to merge the functions, we need to merge the return types. Consider two mutually recursive functions f and g that return types T and U:

```
f ::  X → T
f x = ... g a ... ff

g ::  Y → U
g y = ... f b ... gg
```

To merge f and g, we introduce an algebraic data type that can hold either T or U,

```
data FG = FG_f  T
          FG_g  U
```

introduce variants of f and g that return this new type

```
f′ ::  X → FG
f′ x = ... g a ... (FG_f ff)

g′ ::  Y → FG
g′ y = ... f b ... (FG_g gg)
```

and add wrapper functions with the original signatures

```
f ::  X → T
f x = case f′ x of FG_f r → r
g ::  Y → U
g y = case g′ y of FG_g r → r
```

Inlining these wrappers leaves only f' and g' as mutually recursive and all return the same type.

```
f′ ::  X → FG
f′ x = ... (case g′ a of FG_g r → r) ... (FG_f ff)
g′ ::  Y → FG
g′ y = ... (case f′ b of FG_f r → r) ... (FG_g gg)
```

To merge these two functions, we need to also unify their argument types. Again, we introduce an algebraic type that can represent either:

```
data FGc = F X
           G Y
```

and merge their bodies with a *case* expression, replacing calls to f' and g' with calls to *fg* passed a constructor. We make a similar modification to their wrappers.

```
fg ::  FGc → FG
fg a = case a of
  F x = ... (case fg (G a) of FG_g r → r) ... (FG_f ff)
  G y = ... (case fg (F b) of FG_f r → r) ... (FG_g gg)
f ::  X → T
f x = case fg (F x) of FG_f r → r
g ::  Y → U
g y = case fg (G y) of FG_g r → r
```

After these transformations, groups of mutually recursive functions are fused into single self-recursive functions.

This procedure resembles Danvy's defunctionalization [4], which introduces an *apply* function that takes a function identifier as the first argument, similar to our *fg* function.

4.2 Sequencing Recursive Call Sites

In preparation for transforming each self-recursive function into CPS form, we transform each function so that each recursive call appears only in a *let* with a single binding. This effectively imposes a linear order on all the recursive calls. The body of each such *let* is exactly the code to be executed after the recursive call returns, i.e., its continuation. We will later slice the function at these points.

Our algorithm lifts out the scrutinee of any *case* expression and the arguments of any recursive call and binds each such subexpression to a new temporary. Next, our algorithm flattens any nested *let* expression to yield a simple sequence of *let* expressions. To illustrate, consider the following recursive function *f*, which contains several recursive calls to itself along with calls to h and g.

```
f arg = ...
  case h arg of
    _ → let a = f (g (f b)) in c ...
```

We change this to bind the result of the inner call of f to new temporary *t1*:

```
f arg = ...
  case h arg of
    _ → let a = f (let t1 = f b in g t1) in c ...
```

Next, we lift out the scrutinees of *case* expressions and recursive call argument by binding them to new temporaries so that all recursive calls are in A-normal form [25]. Here, *t2* and *t3* hold the argument of f and the *case* scrutinee.

```
f arg = ...
  let t3 = h arg
    in case t3 of
      _ → let a =
              let t2 = (let t1 = f b in g t1)
              in f t2
          in c ...
```

Finally, we flatten all nested *let* expressions, giving

```
f arg = ...
  let t3 = h arg
    in case t3 of
      _ → let t1 = f b in
          let t2 = g t1 in
          let a = f t2 in c ...
```

4.3 Dividing Functions into Continuations

After the last step, recursive calls are in a linear sequence and located in the local bindings of simple *let* expressions. Our ultimate goal is to replace these recursive calls with tail-recursive calls that manipulate continuations on a stack.

Rather than use lambda expressions, which we would ultimately have to eliminate in a hardware implementation, we directly introduce an algebraic data type "*Cont*" that represents partially applied continuations (each is missing the value returned by the recursive call), then introduce a continuation-handling function "*ret*" that takes a continuation and a result from a recursive call and evaluates each continuation in a *case* expression.

There is always a single continuation implying a return of the result to the environment. We call this *K0*, giving us

```
data Cont = K0

ret k r = case k of K0 → r
```

The original recursive function is renamed *call* and given an additional argument *k* representing the continuation that will receive the result. Finally, the original function becomes a wrapper that calls the new entry point with *K0*:

```
call x1 ... xn k = ...

f x1 ... xn = call x1 ... xn K0
```

After the last step, the bodies of the *let* expressions with recursive calls are exactly the continuations for recursive function calls, so we divide up the function at these *let* expressions and replace each with a call to the *call* function, add a alternative to the *Cont* type that captures all the free variables in the body of the *let* (effectively performing lambda lifting), and add the body of the *let* to the branch of the *case* in the *ret* function.

For example, if from the last step we have

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

we turn it into the following fragment

```
let v1 = ... in
...
let vn = ... in
call a1 ... am (K1 v1 ... vn)
```

and extend *Cont* type and *ret* function as follows:

```
type Cont = K0
            K1 T1 ... TN
            ...

ret k r = case k of
  K0 → r
  K1 v1 ... vn → let z = r in ... v1 .... vn ...
```

Once this is completed, each recursive function has been replaced by a pair of mutually recursive functions *call* and *ret*. We then combine these two functions into a single tail-recursive function using the procedure we described earlier in Section 4.1.

5. IR DIALECTS

As seen in Figure 1, our compilation process uses three distinct dialects of our IR to describe a program on its way from Haskell to SystemVerilog. The first dialect is GHC's explicitly typed, polymorphic IR, which removes a number of Haskell constructs but still contains recursion. We ignore a number of features provided by GHC's IR (foreign function calls, newtype definitions, type coercions, etc.), yielding the IR previously illustrated in Section 3. Our monomorphise and recursion removal passes (Section 4) transform this IR into its second form, described below. The third IR version eliminates all recursive functions by introducing infinite data streams that model the passage of time. Section 5.2 discusses this final form in detail.

5.1 Combinational IR

We produce our second IR dialect by removing a number of language features that cannot be represented in hardware, resulting in programs that almost look like combinational logic. We first apply a monomorphise pass from the MLton compiler [6]: we specialize and rename each polymorphic function and data type based on their concrete type arguments, which are provided by GHC's type inference rules.

The recursion removal pass converts all recursive functions into tail form, removing all nested lambda expressions in the process. We still require lambda expressions to define top-level functions, but any nested lambdas are lifted and renamed. Top-level functions will later be translated into RTL modules, described in Section 6. Tail recursive functions are still permitted at this stage; recursive calls to these functions are interpreted as clock cycle boundaries (making this dialect not truly combinational).

The last transformation in this stage removes recursive algebraic data types definitions. We represent user-defined data types as statically-sized bit vectors in hardware; if a definition is recursive, we cannot determine how many bits to assign to the recursive component at compile time. Thus, we replace recursive portions of these data types with a pointer data type e.g. a non-empty integer list defined as Cons Int List becomes Cons Int Ref, where Ref can be defined as a primitive numeric type. This transformation is not yet automated; future work will focus on how to determine an efficient size for the Ref data type based on how much heap space a program requires.

5.2 Sequential IR: Streams for Time

By design, a program restricted to our second IR dialect is essentially combinational logic: finite, tail-recursive functions operating on hardware-realizable data types. To synthesize interesting hardware, however, we need to be able to represent the passage of time.

We use recursively defined infinite streams to model the behavior of synchronous signals over time. Although the stream is infinite, its elements are produced by bounded computation. This structure mirrors synchronous digital circuitry: gates and wires do a finite amount of computation each clock cycle, but the system is assumed to run forever. The Lustre [12] synchronous language suggested this approach; we adapted it for a Haskell environment as presented by Hinze [13]. An advantage of our approach is that it does not require any new semantics: infinite streams were already lurking in our IR via lazy evaluation, which Haskell provides for us automatically.

We have one built-in data constructor named *delay* (represented "▷") for building streams, defined as follows:

```
data Stream a = ▷ a (Stream a)
```

This constructor takes a value of type *a* and a Stream of elements (each of type *a*), defining a stream that consists of the initial value provided followed by the stream provided.

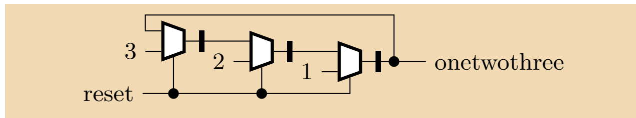
We usually write ▷ as an infix operator instead of using the prefix notation associated with data constructors, so instead of

```
onetwothree = ▷ 1 (▷ 2 (▷ 3 onetwothree))
```

we write the equivalent

```
onetwothree = 1 ▷ 2 ▷ 3 ▷ onetwothree
```

A syntax-directed translation suffices to transform such infinite stream definitions into hardware. Each ▷ becomes a multiplexer-driven register that takes the first argument to the ▷ when *reset* is asserted and the second one otherwise.

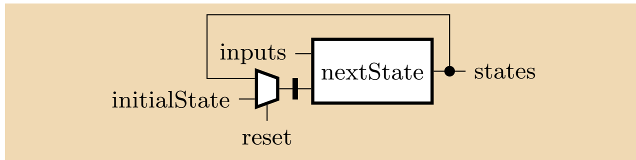


The *delay* constructor plus the *zipWith* function found in most functional languages lets us express state machines. The *zipWith* function, which our code generator treats specially, applies a (combinational) function piecewise to a pair of input streams to produce an output stream, i.e.,

```
zipWith f (a0 ▷ a1 ▷ a2 ▷ ···) (b0 ▷ b1 ▷ b2 ▷ ···) =
    f a0 b0 ▷ f a1 b1 ▷ f a2 b2 ▷ ···
```

This allows us to express arbitrary Mealy FSMs in our IR:

```
states = zipWith nextState inputs (initialState ▷ states)
```



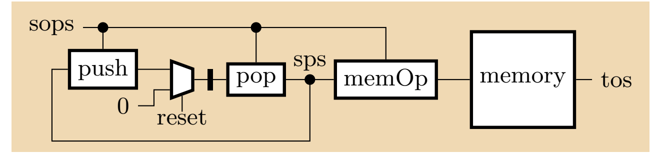
We can easily model a memory block as a polymorphic function that transforms a stream of read and write operations into a stream of read values. Read and write operations are represented as instances of a polymorphic data type:

```
data Memop adr dat = Read adr
                     Write  adr  dat
memory :: Stream (Memop adr dat) → Stream dat
```

This memory function can be used to build a stack of Ints that takes a stream of push/pop operations ("*sops*") and returns a stream of top-of-stack values ("*tos*").

```
data SOP = Push Int | Pop | NOP
pop c sp = case c of Pop → sp − 1
                     _   → sp
push c sp = case c of Push _ → sp + 1
                      _      → sp
sps = zipWith pop sops (1 ▷ zipWith push sops sps)
memOp c sp = case c of Push k → Write sp k
                       _      → Read sp
tos = memory (zipWith memOp sops sps)
```

Here, *sps* is the stream of stack pointers. Separating the *push* and *pop* functions lets us pre-decrement and post-increment the stack pointer. Graphically, this is



5.3 Adding Streams to Fibonacci

We now return to the Fibonacci example of Section 2 to illustrate how we augment a tail-recursive-only IR program into one with streams.

The *Cont* data type is recursive and cannot be represented as a finite-length bit vector since we cannot guarantee a bound on its depth. In general, it is possible to replace recursive references with pointers and store them on a heap, but we can do better here. Continuations used for recursion follow a stack discipline and can be managed accordingly: constructing a *Call* involves pushing a continuation on the stack, and constructing a *Ret* involves popping a continuation from the stack. As such, we can simply drop the recursive reference to the *Cont* type, leaving us with

```
data Cont = K0
            K1 Int
            K2 Int
```

To complete the translation into our IR-with-streams, we rewrite the *next* function as combinational and connect it with a stack via the *mergetos* function:

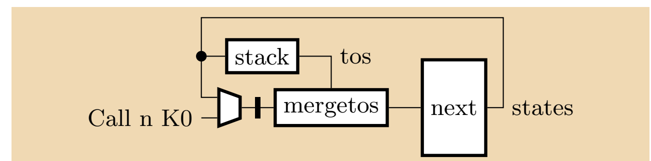
```
next c  = case c of
  Call 1 k        → Ret k 1
  Call 2 k        → Ret k 2
  Call n _        → Call (n−1) (K1 n)
  Ret (K1 n)   r  → Call (n−2) (K2 r)
  Ret (K2 n1)  r  → Ret K0 (n1 + r)

pop c sp = case c of Ret _ _ → sp − 1
                     _       → sp
push c sp = case c of Call _ _ → sp + 1
                      _        → sp
sps = zipWith pop states (1 ▷ zipWith push states sps)
memop c sp = case c of Call _ k → Write sp k
                       Ret  _ _ → Read sp
tos = memory (zipWith memop states sps)

mergetos c k = case c of Ret _ n → Ret k n
                         c       → c

states = map next (zipWith mergetos
                    (Call n K0 ▷ states) tos)
```

where *map* is a unary version of *zipWith*: it applies a combinational function pointwise to a stream. Graphically,



6. TRANSLATION TO SYSTEMVERILOG

Once a program is in our sequential IR dialect, a syntax-directed translation suffices to generate equivalent SystemVerilog. We present the general translation scheme for each language construct along with an example of each scheme. In the following examples, a black line separates IR code (on the left) from the associated SystemVerilog (on the right). We omit the begin-end statements from the SystemVerilog code to decrease clutter.

Finite types (non-stream, non-function) in our IR are represented as bit vectors (SystemVerilog provides tagged union types, but we found no tools to support them). The $\log_2 n$ least-significant bits encode which of the n variants is being represented; we provide an additional m bits, where m is large enough to store all the fields in the largest variant.

For example, our *Int* type is a 32-bit signed integer, so the bit vector representing the *Cont* type from our Fibonacci example needs 34 bits to (possibly) hold an integer along with two tag bits to distinguish the three variants.

```
data Cont = K0              | typedef logic [33:0] Cont;
          K1 Int            |
          K2 Int            |
```

Each variant is defined with a SystemVerilog function that builds and returns the corresponding bit vector. For example, the following function constructs a K1 variant of the Cont type:

```
function Cont K1(Int t1);
  K1 [1:0] = 2'd1; // Set the tag
  K1 [33:2] = t1;  // Set the integer field
endfunction
```

Each *case* expression becomes a multiplexer that selects the appropriate case alternative. If matching on an algebraic data type, the SystemVerilog code uses the corresponding bit vector's tag bits to select the proper case alternative.

```
c = case x of          | Int c, n;
    K0 → 0             | assign n = x[33:2];
    K1 n → n           | always_comb
    K2 n → n           |   unique case (x[1:0])
                       |     default: c = {32'bx};
                       |     0: c = 32'd0;
                       |     1: c = n;
                       |     2: c = n;
                       |   endcase
```

Each top-level function definition becomes a SystemVerilog module with additional clock and reset inputs. Calls to a function are translated into module instantiations. For example, given a primitive SystemVerilog module *plus* that adds two integer arguments,

```
myAdd :: Int→Int→Int =   | module myadd(
  λ a b → (+) a b        |     output Int out,
                         |     input logic clk, reset,
                         |     input Int a, b);
                         |   plus a1(out, clk, reset, a, b);
                         | endmodule
```

The *zipWith* function produces the same SystemVerilog code as a standard function call (e.g. the plus function in the above example): the first argument to *zipWith* (a function) becomes a module instantiation, and the following two Stream arguments are registers passed to the module.

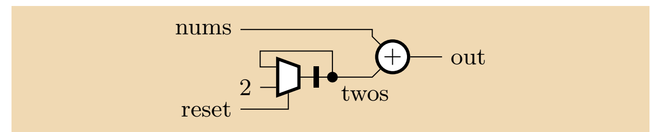
Implementing streams is, by design, very simple. Each *delay* constructor (▷) becomes a register with a mux controlling how it should be reset.

```
ys :: Stream Int   | Int ys;
ys = x ▷ xs        | always_ff @(posedge clk)
                   |   if ( reset ) ys <= x;
                   |   else         ys <= xs;
```

A *let* expression in a function introduces local variables in the corresponding module declaration. We rename these variables if needed to avoid any collisions.

Below is an example illustrating how functions, *let*, *zipWith*, and ▷ interact.

```
add2 :: Stream Int          | module add2(
     → Stream Int           |     output Int out,
add2 nums =                 |     input logic clk, reset,
  let twos = 2 ▷ twos       |     input Int nums);
  in zipWith (+)            |   Int twos;
       nums twos            |   always_ff @(posedge clk)
                            |     if (reset) twos <= 32'd2;
                            |     else       twos <= twos;
                            |   plus a1(out, clk, reset,
                            |           nums, twos);
                            | endmodule
```



A primitive *memory* function (which implements the function shown in Section 5.3 that we provide to the user) takes a size, a write enable stream, an address stream and a data input stream, producing a data output stream. We represent this function with an array declaration and a sequential process that reads from or writes to it. FPGA synthesis tools infer this description as a clocked RAM block.

```
do = memory 16 we a di   | Int c1 [15:0], m;
                         | always_ff @(posedge clk)
                         |   if (we) c1[a] <= di;
                         |   do <= c1[a];
```

7. EXPERIMENTAL RESULTS

We have implemented our compiler in Haskell, which fully automates the rewriting passes outlined in previous sections with one exception: the combinational-to-sequential pass is still under development, but it can currently handle a few recursive example programs, which we use here to test our end-to-end pipeline. The full pipeline begins with GHC generating an external core file from the source program, which we then parse into an abstract syntax tree using an adapted version of the parser from Tolmach et al. [30]. This AST is the input to our compiler, which monomorphises the program, removes recursion using the algorithm described in Section 4, adds streams, and finally generates synthesizable SystemVerilog. We then synthesized the resultant circuit designs using Altera's Quartus 14.0 and targeted a recent Stratix V FPGA.

7.1 Synthesis Results

There are two types of examples in Table 1: the first group are classical small recursive algorithms, including the Fi-

Example	f_{max}	ALMs	RAM	Regs
Fib(20)	306 MHz	131	1 360 bits	55
Fib(25)	318	131	1 700	49
Fib(30)	306	132	2 040	48
Ack(3,6)	320	173	1 700	138
Ack(3,7)	325	162	3 400	135
Ack(3,8)	315	173	5 100	130
Sum(10K)	398	57	11 000	52
Sum(100K)	258	71	110 000	65
Sum(1M)	159	140	1 100 000	76
Bresenham	147	117	0	64
Relay Station	650	11	0	31
Length	346	142	2 048	148
Append	232	355	3 136	286

Results from Altera Quartus 14.0.0 targeting a Stratix V 5SGXEA7H3F35C3 with 234,720 ALMs, 52,428,800 RAM bits.

Table 1: FPGA Synthesis Results

```
fib :: Int → Int
fib 0 = 0
fib 1 = 1
fib n = fib (n−1) + fib (n−2)

sum :: Int → Int
sum 0 = 0
sum n = 1 + sum (n−1)

ack :: Int → Int → Int
ack 0 n = n + 1
ack m 0 = ack (m−1) 1
ack m n = ack (m−1) (ack m (n−1))
```

Figure 3: Recursive programs for experiments

bonacci and Ackermann functions, whose source is shown in Figure 3; the second group are examples written manually in our sequential IR (i.e., with streams). *Bresenham* computes points along a line using 16-bit integers with the familiar algorithm; it uses no memory. *Relay Station* is a latency-insensitive buffer whose input and output channels support backpressure; it performs no arithmetic and was carefully designed to have short combinational paths. *Length* and *Append* are the familiar list-processing algorithms operating on a rudimentary heap.

Table 1 lists the performance of each synthesized circuit and what resources it consumes. The f_{max} column lists the maximum operating frequency (in MHz) of the circuit on the FPGA. The ALMs column estimates circuit area: each adaptive logic module is a basic FPGA building block that corresponds roughly to a pair of 16-bit RAMs. The RAM column lists the number of bits of on-chip block RAM the circuit uses; the Regs column lists the number of (one-bit) registers in the circuit.

As shown in Table 1, the designs generated by our compiler run at frequencies ranging from 147 MHz to 650 MHz on this particular FPGA. The first group of examples uses RAM to hold the stack. For example, the Fib(20) example uses 1360 bits because we set the stack depth to 40 and the biggest continuation on the stack holds a 32-bit integer and must distinguish three variants, which requires two additional bits. Hence $34 \times 40 = 1360$ bits. The stacks for the Sum examples are only one bit wide but very deep to accommodate their recursion depths.

Example	C++		Haskell	FPGA	
	time	cycles	time	time	cycles
Fib(20)	0.06 ms	204 k	1.67 ms	0.14 ms	43 k
Fib(25)	0.33	1 122	10.4	1.53	486
Fib(30)	3.20	10 880	61.8	17.6	5 385
Ack(3,6)	0.53	1 802	10.1	1.08	344
Ack(3,7)	2.09	7 106	24.1	4.27	1 387
Ack(3,8)	8.09	27 506	85.5	17.7	5 571
Sum(10K)	0.04	136	1.83	0.05	20
Sum(100K)	0.32	1 088	17.3	0.78	200
Sum(1M)	5.45	18 530	138	12.6	2 000

C++ and Haskell versions run on a Linux desktop with an Intel Core i5-3570K CPU @ 3.40GHz. C++ programs were compiled by g++ 4.6.3. Haskell programs were compiled by GHC 7.6.3.

Table 2: Performance Results for an FPGA

7.2 Performance Results

We give performance result of the example programs after they are compiled by our compiler and synthesized to the target FPGA. To compare how they perform relative to a pure software implementation, we also tested the performance of those examples implemented in Haskell and C++. Table 2 shows running times on the CPU and FPGA.

We did not turn on compiler optimization for the C++ or Haskell examples since our compiler is also non-optimizing.

As shown from Table 2, our FPGA implementations are generally slower than the C++ implementations but faster than the Haskell implementations. Part of this is the unavoidable overhead in an FPGA implementation due to its use of programmable wiring, look-up tables, etc. We expect an ASIC implementation would be substantially faster, although we have not run those experiments. However, note that these circuits consume a very small fraction of the FPGA's resources, whereas the software implementations are consuming an entire processor core. Something like the Fib(30) circuit is using much less than 1% of the FPGA's resources, so it would be possible to run 100 such circuits on the FPGA, promising much higher throughput than the software counterparts.

8. RELATED WORK

This project synthesizes work from both functional languages and hardware design; Gammie [8] provides a good overview of the literature on this.

8.1 Functional Descriptions to Hardware

The Cλash compiler [1, 2] adopts many of the same techniques we have to compile a particular Haskell subset to hardware. They, too, start from a GHC front-end to generate Core, then apply rewrite rules (including Danvy's defunctionalization [4] inspired by Reynolds [24]) to transform it into a normal form that can be translated into an HDL. Like us, they also support algebraic data types. Their focus, however, has been more on structural, combinational circuits, and they do not support recursion.

The SHard compiler [26] generates sequential circuits from a Scheme dialect and supports recursion. Like our work and the Cλash compiler, their compiler performs series of semantics-preserving rewrites, including CPS conversion and lambda lifting. Most notably, it supports memory-resident

closures managed with a read-once discipline instead of a stack. However, they have very limited support for data types, forcing their example programs to embed lists in closures. Furthermore, their compiler exercises little control over scheduling since it generates hardware from an IR that does not express clock cycle boundaries, unlike our streams.

Sharp and Mycroft's FLaSH compiler [22] focuses on sharing a single implementation of a function among multiple callers, typically placing an arbiter and router around a function to ensure that it correctly handles multiple, simultaneous invocations, something we expect to have to do in our own work. Despite its focus on function call semantics, it does not support recursion.

Like FLaSH, resource sharing is a focus of Bluespec [14], a hardware description language with a strong functional influence (e.g., higher-order functions, polymorphism). Bluespec's most distinctive feature is its use of "rules"—guarded atomic actions that are scheduled by a synthesized rule scheduler. Unlike FLaSH, SHard, and our work, however, Bluespec is fundamentally a hardware description language, forcing designers to describe behavior at a clock-cycle granularity, and therefore provides a lower level of abstraction to designers. We considered generating Bluespec from our compiler, but did not want to rely on a proprietary tool.

A variety of other projects have taken a functional approach to describing hardware *structure* as opposed to sequential algorithms, which are our focus. Sheeran [28] pioneered this area, starting with the FP-inspired language μfp [27] designed to specify complex combinational structures. Sheeran and her group later produced the far richer Lava [3] as a domain-specific language embedded in Haskell, but it still focuses on describing circuit structures. Gill et al. [10] also developed a Lava variant. Separate from Lava, Li and Leeser's HML [18] also takes a functional approach to describing circuit structure.

8.2 Compiling Recursion to Hardware

Ghica et al. [9] also implement recursive algorithms in hardware; we believe our technique is more flexible and produces better circuits. From a recursive function in an imperative Algol-like language, they begin by synthesizing its body as if it were not recursive, but they replace each register used to store a local variable with a little memory that operates as a stack. All the stacks for a particular function share a single stack pointer. In contrast, our technique considers exactly which variables are live across recursive calls and only stores those to a shared stack, which may lead to reduced memory usage. Furthermore, our technique is based on a series of rewriting transformations; theirs treats recursive functions as a special case.

Middendorf et al. [21] take a radically different approach to implementing recursion in hardware. They transform a C function into a set of rewrite rules that consume and produce a certain number of tokens, each of which represents a function call, an argument, or a result. Their basic architecture sends a stream of tokens around a loop that includes one or more blocks that implement the rewriting rules. We suspect this approach could also be employed in our setting, but we have not yet attempted this.

Maruyama et al. [20] also propose FPGA-friendly architectures for multithreaded execution of recursive functions. We also plan to consider their approach in our setting, although they do not propose a synthesis technique.

9. CONCLUSIONS

We presented a compiler able to take Haskell programs with recursion, algebraic data types, and polymorphic types and functions and synthesize digital logic from them that we ran on an FPGA. Our compiler demonstrates it is possible to compile recursive functional programs to hardware by performing a series of semantics-preserving rewritings that transform the program into a series of increasingly simple normal forms, the last of which corresponds directly to synchronous digital circuits.

We described our intermediate representation, which we derived from GHC's core, a procedure for "compiling away" recursion that substituted true recursion for tail recursion with explicit stack management, a subset of our IR from which synthesizable (RTL) System Verilog can be generated through a syntax-directed translation, and showed experimental results that demonstrated the technique is practical.

Our experimental results are encouraging, suggesting that such approach can be made practical. In the future, we plan to add additional optimizations to our compiler and add support for heap-resident data across multiple small memories: the long term goal of our larger project.

Acknowledgments

10. REFERENCES

[1] 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.

[2] C. P. R. Baaij and J. Kuper. Using rewriting to synthesize functional languages to digital circuits. In *Proceedings of the Trends in Functional Programming (TFP)*, Lecture Notes in Computer Science, pages 17–33, Provo, Utah, 2014. Springer.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 174–184, Baltimore, Maryland, 1998.

[4] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of Principles and Practice of Declarative Programming (PPDP)*, pages 162–174, New York, NY, USA, 2001. ACM.

[5] S. A. Edwards. The challenges of synthesizing hardware from C-like languages. *IEEE Design & Test of Computers*, 23(5):375–386, Sept. 2006.

[6] M. Fluet. Monomorphise, Jan. 2015. http://mlton.org/Monomorphise [Online; accessed 23-January-2015].

[7] D. P. Friedman and M. Wand. *Essentials of Programming Languages*. MIT Press, third edition, 2008.

[8] P. Gammie. Synchronous digital circuits as functional programs. *ACM Computing Surveys*, 46(2):article 21, Nov. 2013.

[9] 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.

[10] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. Introducing Kansas Lava. In *Proceedings of the International Symposium on Implementation and Application of Functional Languages*, volume 6041 of *Lecture Notes in Computer Science*, Nov. 2009.

[11] R. Gupta and F. Brewer. High-level synthesis: A retrospective. In *High-Level Synthesis: From Algorithm to Digital Circuit*, pages 13–28. Springer, 2008.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[13] R. Hinze. Function pearl: Streams and unique fixed points. In *Proceedings of the International Conference on Functional Programming (ICFP)*, Victoria, BC, Sept. 2008.

[14] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 511–518, Nov. 2000.

[15] S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, 1984.

[16] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, 1985. Springer.

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

[18] Y. Li and M. Lesser. HML: An innovative hardware design language and its translation to VHDL. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):1–8, Feb. 2000.

[19] H. Liu, N. Glew, L. Petersen, and T. Anderson. The Intel labs Haskell research compiler. In *Proceedings of the Haskell Symposium*, pages 105–116, Boston, Massachusetts, Sept. 2013.

[20] T. Maruyama, M. Takagi, and T. Hoshino. Hardware implementation techniques for recursive calls and loops. In P. Lysaght, J. Irvine, and R. Hartenstein, editors, *Proceedings of the Conference on Field-Programmable Logic and Applications (FPL)*, volume 1673 of *Lecture Notes in Computer Science*, pages 450–455. Springer Berlin Heidelberg, 1999.

[21] L. Middendorf, C. Bobda, and C. Haubelt. Hardware synthesis of recursive functions through partial stream rewriting. In *Proceedings of the Design Automation Conference (DAC)*, pages 1207–1215, San Francisco, California, 2012.

[22] 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.

[23] S. Peyton Jones and A. Santos. Compilation by transformation in the Glasgow Haskell Compiler. In K. Hammond, D. Turner, and P. Sansom, editors, *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 184–204. Springer London, 1995.

[24] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, 1972.

[25] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3–4):289–360, 1993.

[26] X. Saint-Mleux, M. Feeley, and J.-P. David. SHard: a Scheme to hardware compiler. In *Proceedings of the Scheme and Functional Programming Workshop (SFPW)*, pages 39–49, Portland, Oregon, Sept. 2006. University of Chicago technical report TR–2006–06.

[27] M. Sheeran. μFP, an algebraic VLSI design language. In *Proceedings of the ACM Symposium on LISP and Functional Programming (LFP)*, pages 104–112, Austin, Texas, Aug. 1984.

[28] M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7), July 2005.

[29] G. L. Steele. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT Press, 1978.

[30] A. Tolmach, T. Chevalier, and T. G. Team. An external representation for the GHC core language (for GHC 6.10), Apr. 2010.