Hardware in Haskell: Implementing Memories in a Stream-Based World

Richard Townsend  Martha A. Kim  Stephen A. Edwards

Columbia University, Department of Computer Science
Technical Report CUCS-017-15
September 21, 2015

ABSTRACT
Recursive functions and data types pose significant challenges for a Haskell-to-hardware compiler. Directly translating these structures yields infinitely large circuits; a subtler approach is required. We propose a sequence of abstraction-lowering transformations that exposes time and memory in a Haskell program, producing a simpler form for hardware translation. This paper outlines these transformations on a specific example; future research will focus on generalizing and automating them in our group’s compiler.

1. INTRODUCTION

We present a sequence of transformations that converts a specific Haskell program into a form permitting simple, syntax-directed translation into SystemVerilog (a standard hardware description language). These transformations represent the flow of our prototype Haskell-to-SystemVerilog compiler. We treat programs as having strict semantics instead of Haskell’s usual lazy on-demand policy and only consider programs that produce identical results under both semantics.

```
data List a = Cons a (List a) | Nil

main :: List Int
main = append (Cons 1 (Cons 2 Nil)) (Cons 3 Nil)

append :: List a → List a → List a
append z y = case z of
  Nil → y
  Cons x xs → Cons x (append xs y)
```

The program used throughout the paper is shown above; we employ a pidgin Haskell notation to mirror our compiler’s intermediate representation [7]. The main variable defines the program’s output: the concatenation of two integer lists, each implemented with a polymorphic, recursive List data type. The append function traverses the first list, leaving new Cons instances behind:

```
append (Cons 1 (Cons 2 Nil)) (Cons 3 Nil)
= Cons 1 (append (Cons 2 Nil) (Cons 3 Nil))
= Cons 1 (Cons 2 (append Nil (Cons 3 Nil)))
= Cons 1 (Cons 2 (Cons 3 Nil))
```

Representing append as purely combinational would require an infinitely large circuit since the number of recursive calls is unbounded and each requires a copy of the case expression’s logic. Incorporating a register-based feedback loop prevents this logic explosion, but two issues remain: arbitrating between recursive calls and Cons construction, and storing each element of the first list as we recurse. Furthermore, sequential circuits need a clock, but our program has no notion of time.

We circumvent these problems with three transformations: rewriting the function in Continuation-Passing Style (CPS) ensures that all recursive calls complete before Cons construction begins (Section 2), introducing an infinite Stream data type provides a clock (Section 3), and adding a stack provides storage for the first list’s data (Section 4).

The unboundedness of the List data type presents another challenge. We implement data type instances with statically-sized bit vectors, but our compiler cannot bound the length of an arbitrary List. We mitigate this issue by replacing data type recursion with explicit pointers and defining a heap to manage them. The pointers will refer to heap locations, and functions will pass pointers instead of fully realized data structures (Section 5).

The transformations in Section 2 and the syntax-directed translation to SystemVerilog have been generalized and automated within our compiler [8]. Here, we focus on the introduction of streams and memories to implement the recursion in our specific example.

2. PRE-STREAM TRANSFORMATIONS

We begin by removing polymorphism by specialization (done, e.g., in the MLton compiler [1]): in this example, we restrict lists to integers. We keep the same names here; in practice our compiler would rename the List type to List_Int to distinguish it from other specialized variants.

```
data List = Cons Int List | Nil

main :: List
main = append (Cons 1 (Cons 2 Nil)) (Cons 3 Nil)

append :: List → List → List
append z y = case z of
  Nil → y
  Cons x xs → Cons x (append xs y)
```

The next pass converts recursive functions into tail form by rewriting them in Continuation-Passing Style [2, 6]. A CPS function takes an extra “continuation” argument $k$ that describes what to do with the result of each call. The initial continuation returns the function’s result; subsequent continuations use each call’s result to construct a new Cons and pass it to the previous continuation $k$. We apply this chain of continuations to $y$ in the base case, resulting in the final concatenated list.

```
data List = Cons Int List | Nil

main :: List
main = append (Cons 1 (Cons 2 Nil)) (Cons 3 Nil)

append :: List → List → List
append z y = case z of
  Nil → y
  Cons x xs → Cons x (append xs y)
```

The next pass converts recursive functions into tail form by rewriting them in Continuation-Passing Style [2, 6]. A CPS function takes an extra “continuation” argument $k$ that describes what to do with the result of each call. The initial continuation returns the function’s result; subsequent continuations use each call’s result to construct a new Cons and pass it to the previous continuation $k$. We apply this chain of continuations to $y$ in the base case, resulting in the final concatenated list.
main :: List
main = append (Cons 1 (Cons 2 Nil))
     (Cons 3 Nil)
     (Result :> result)

append :: List -> List -> (List -> List) -> List
append z y k = case z of
  Nil -> k y
  Cons x xs -> append xs y (\l -> k (Cons x l))

To avoid anonymous functions in hardware, we perform a
lambda-lifting pass [5] that names each continuation as
a top-level function (here, c1 and c2) and adds free vari-
ables as formal arguments; these precede the result argu-
ment passed to every continuation.

main :: List
main = append (Cons 1 (Cons 2 Nil))
     (Cons 3 Nil)
     c2

append :: List -> List -> (List -> List) -> List
append z y k = case z of
  Nil -> k y
  Cons x xs -> append xs y (\c1 x k -> c1 x k)

We merge the continuation functions into append with
the Continuation and Action data types: the former rep-
resents the partially applied continuations, while the latter
partitions append’s behavior into recursive calls (Call) and
continuation evaluation (Ret).

data Continuation = C1 Int Continuation | C2
data Action = Call List List Continuation
              | Ret Continuation List

main :: List
main = append (Call (Cons 1 (Cons 2 Nil))
              (Cons 3 Nil))
              (C2)

append :: Action -> List
append action = case action of
  Call Nil y k -> append (Ret k y)
  Call (Cons x xs) y k -> append (Call xs y (C1 x k))
  Ret (C1 x k) l -> append (Ret k (Cons x l))
  Ret C2 l -> l

Append now operates in two phases: Calls push the first
list’s elements into a stack of continuations, then Ret s pop
the values from the continuations onto the head of the result.
These transformations leave the semantics of the program
unchanged.

Below, we illustrate the behavior of this variant using the
more concise Haskell list notation. E.g., [1,2] represents
Cons 1 (Cons 2 Nil).

append (Call [1,2] [3] C2) = append (Call [2] [3] (C1 1 C2))
= append (Call [] [3] (C1 2 (C1 1 C2)))
= append (Ret (C1 2 (C1 1 C2)) [3])
= append (Ret (C1 1 C2) [2,3])
= append (Ret C2 [1,2,3])
= [1,2,3]

3. THE STREAM DATATYPE
The final append function above resembles a finite state
machine’s transition table: given a current Action on the
left, generate a new Action on the right or return a final
result. Each transition can be computed with combinational
logic, but the unbounded nature of the recursion demands a
sequential circuit.

Describing sequential circuits requires a notion of time.
We express the behavior of signals over time using a poly-
morphic, recursive Stream data type inspired by the Lustre
language [3]. We follow the implementation of Hinze [4]:

data Stream a = <> a Stream a

We construct a Stream instance with the (infix) data con-
structor “<>”, called delay, which prepends a value of type a
to a Stream carrying elements of the same type. The lack of
a base case in the type definition captures the infinite aspect
of streams; every element in a Stream is always followed by
another. Ultimately when we synthesize hardware, each <>
operator becomes a hardware register [8]. We only permit
streams of bounded types since they are meant to model the
behavior of finite groups of wires over time. In particular,
we do not allow streams of streams.

The code snippet below illustrates two streams defined
with infix notation: x is the Boolean stream True False True
False False ..., and y is x delayed by a cycle: True True
False False False False ....

x = True <> False <> False <> x
y = True <> x

Before introducing streams into our program, we first make
append truly combinational. Each Action argument replaces
its recursive call, and a new NOP action symbolizes the func-
tion’s termination. The Start action triggers append’s first
Call; this will be useful when we incorporate memories into
our program.

append :: Action -> Action
append action = case action of
  Call Nil y k -> Ret k y
  Call (Cons x xs) y k -> Call xs y (C1 x k)
  Ret (C1 x k) l -> Ret k (Cons x l)
  Ret C2 l -> NOP
  NOP -> NOP
  Start -> Call (Cons 1 (Cons 2 Nil)) (Cons 3 Nil) C2

We now introduce sMap, which applies a (combinational)
function to a stream. Later, we will use other list-inspired
stream functions.

3. THE STREAM DATATYPE
sMap :: (a → b) → Stream a → Stream b
sMap f (a0 ▷ a1 ▷ a2 ▷ ...) = f a0 ▷ f a1 ▷ f a2 ▷ ...

We express the behavior of append over time using appStream:

appStream :: Stream Action
appStream = sMap append (Start ▷ appStream)

This recursive definition gives

appStream = append Start ▷
  append (append Start) ▷
  append (append (append Start)) ▷ ...
  = Call [1,2] [3] C2
  ▷ Call [2] [3] (C1 1 C2)
  ▷ Call [ ] [3] (C1 2 (C1 1 C2))
  ▷ Ret (C1 2 (C1 1 C2)) [3]
  ▷ Ret (C1 1 C2) [2,3]
  ▷ Ret C2 [1,2,3]
  ▷ NOP
  ▷ ...

This stream contains the arguments of the sequence of function calls at the end of Section 2. This is a syntactic isomorphism: each “= append” call listed earlier becomes “▷” here: the sequence of reductions is performed in successive clock cycles. We retrieve the actual result of append with a projection function:

main :: List
main = result appStream

result :: Stream Action → List
result (Ret C2 1 ▷ _) = 1
result (_, ▷ s) = result s

4. CONTINUATIONS ON THE STACK

We now consider the representation of recursive data types in hardware. Our general solution uses explicit pointers and a heap, but we can do better for the Continuation type.

Because our Action-based append function models recursive calls, continuations exhibit a stack discipline: each Call pushes a new continuation on the stack that holds the next input list element; each Ret pops the continuation to build Cons instances. An explicit stack orders these continuations properly, obviating the need for a recursive type definition:

data Continuation = C1 Int | C2

The memory function—a primitive in our compiler—is the core component of our stack; it models memory with a stream of arrays that represents the state of memory over time. Here we specify that the memory array has three entries, initially populated with dummy C2 continuations. Writing to memory generates a new array on the following cycle, reflecting the change in memory; reading does not affect the array stream.

We interact with memory via a stream of memory operations (memOps): MemRead reads a specified address (Addr represents an arbitrary numeric type), while MemWrite writes a value and returns the previous contents at that address. Memory outputs a stream of continuations (stOut), where the ith continuation is the result of the i-1st memory operation. To avoid undefined behavior, the function outputs the dummy argument (here, C2) as its initial stream element.

data MemOp val = MemRead Addr
  | MemWrite Addr val

stOut :: Stream Continuation
stOut = memory 3 C2 memOps

We define memOps with the sZipWith function, which applies a two-argument function pointwise to two streams.

sZipWith :: (a → b → c) → Stream a → Stream b → Stream c
sZipWith f (a0 ▷ a1 ▷ a2 ▷ ...) (b1 ▷ b2 ▷ ...) = f a0 b1 ▷ f a1 b2 ▷ ...

Each cycle, stackOp observes the current action and stack pointer to determine the appropriate memory operation: Calls write continuations, Ret reads them, and other actions (Start and NOP) generate dummy reads where we ignore the output.

memOps :: Stream (MemOp Continuation)
memOps = sZipWith stackOp appStream sp

stackOp :: Action → Addr → MemOp Continuation
stackOp action addr = case action of
  Call _ _ k → MemWrite addr k
  Ret _ _ → MemRead addr
  _ → MemRead 0

The sp stream implements the stack pointer with a post-increment, pre-decrement policy. If the current action is a Call, push increments the pointer after the associated write has been issued; we impose this delay by prepending the zipped push stream with an initial value of 0. Conversely, a Ret decrements the pointer before the corresponding read occurs.

sp :: Stream Addr
sp = sZipWith pop appStream
  (0 ▷ sZipWith push appStream sp)

pop :: Action → Addr → Addr
pop action addr = case action of
  Ret _ _ → addr - 1
  _ → addr

push :: Action → Addr → Addr
push action addr = case action of
  Call _ _ _ → addr + 1
  _ → addr

This stack design is easily implemented in hardware, as shown in the block diagram below. Wires carry stream values, and blocks of combinational logic implement functions. The delay operator (▷) becomes a register driven by a multiplexer that sets the stack pointer to 0 on reset and otherwise passes the output of push to pop.

Connecting the stack to append is simple. We first remove the “continuation” field from instances of C1 (since the
Figure 1: The behavior of our stream-based program with an explicit stack. Each row represents a clock cycle; each column represents a stream; time goes from top to bottom. Note that the compFlow stream embodies the evaluation of append shown at the end of Section 2.

The stack will handle the link), and assign each generated Ret a dummy C2 continuation. The real continuations now come from the stack; we use mergeStack to replace the dummies and leave other actions unchanged. The resulting compFlow stream represents the flow of computation in our program, requiring a redefinition of main to obtain the final result.

```
mergeStack :: Action -> Continuation -> Action
mergeStack action k = case action of
  Ret _ n -> Ret k n
  _ -> action
```

```
main :: List
main = result compFlow
```

5. Lists on the heap

In general, a stack cannot handle multiple lists, such as the two used in append, because each stack element necessarily has at most one predecessor and one successor. Instead, we manage append’s lists on a simple heap with no garbage collection.

Our heap stores non-recursive list objects, which now consist of an integer payload and a pointer to the next element in the list (the Addr type).

```
data List = Cons Int Addr | Nil
```

Writing a list object to the heap returns a pointer to the object; this makes the heap subtly different from a memory. The interface to our heap consists of two streams: an input that requests either a read, a write (allocation), or no operation; and an output that returns either the requested list object or the address of a newly allocated object.

```
data HeapIn = Read Addr | Write List |
```

```
data HeapOut = Rout List | Wout Addr
```

5.1 Implementing a heap

For this example, we define a small, 8-cell heap with a new memory instance driven by the heapOps stream. The wrap function inspects each memory operation and its result in the same cycle, generating hOut: a stream of HeapOut values. If the previous command was a read, then the resultant List is wrapped in a Rout. Given a write, we wrap the address used in a Wout and ignore the output of the memory.

```
hOut :: Stream HeapOut
hOut = sZipWith wrap (MemRead 0 >> heapOps)
      (memory 8 Nil heapOps)
```

```
wrap :: MemOp List -> List -> HeapOut
wrap memop val = case memop of
  MemRead _ -> Rout val
  MemWrite addr _ -> Wout addr
```

The heapOp function translates HeapIn commands into memory operations. Reads and Writes become MemReads and MemWrites, the latter using the current heap pointer as its address. Unlike the stack, the heap is not assigned to a single function; a simple arbiter chooses which command to pass to the heap each cycle, forming the inputs stream. We define the arbiter and inputs in Section 5.2.

```
heapOps :: Stream (MemOp List)
heapOps = sZipWith heapOp inputs hp
```

```
heapOp :: HeapIn -> Addr -> MemOp List
heapOp input hptr = case input of
  Read addr -> MemRead addr
  Write val -> MemWrite hptr val
  InNOP -> MemRead 0
```

Our allocation scheme is naïve: we simply increment the heap pointer after each Write. This is sufficient for our example; a realistic heap with garbage collection remains future work.

```
hp :: Stream Addr
hp = 0 >> sZipWith update inputs hp
```
The block diagram of our heap resembles our stack:

5.2 The heap controller

Before append can perform any computation, its initial list arguments must be built on the heap. We will define two “builder” functions to construct these arguments, which presents a new challenge: multiple functions contending for heap access.

We tackle this problem with a heap controller, which has a number of responsibilities. Given three input streams (generated by append and the two builders), the controller first arbitrates among them, selecting one command to pass to the heap each cycle. After executing the command, it bundles the result with a set of controls describing which function received heap access. It then sends the bundle to the three functions, dictating their behavior on the next cycle. When both builders have terminated, we also use the bundle to send append’s initial pointer arguments.

5.2.1 Arbitration

We zip all heap inputs into a single stream (allInputs) before passing them off to the controller. The sZip function constructs a stream of tuples from two stream arguments. The argument streams may carry different types, so zipping a stream of tuples with another stream of heap inputs is acceptable. The heap input streams b1, b2, and funcOps come from the two builders and append, respectively; they will be defined in Section 5.3.

\[
sZip :: \text{Stream } a \rightarrow \text{Stream } b \rightarrow \text{Stream } \langle a, b \rangle
\]

\[
\text{allInputs} :: \text{Stream } \langle \text{HeapIn,HeapIn}, \text{HeapIn} \rangle
\]

\[
\text{allInputs} = \text{sZip } b1 \text{ b2 } \text{funcOps}
\]

We generate the inputs stream referenced in our heap design by mapping an arbitration function over allInputs. Since append cannot execute without its initial arguments, arbitrate prioritizes the builders’ Write commands (neither builder ever reads). Append obtains heap access once both builders have finished.

\[
\text{inputs} :: \text{Stream } \text{HeapIn}
\]

\[
\text{inputs} = \text{sMap } \text{arbitrate } \text{allInputs}
\]

\[
\text{arbitrate} :: \langle \text{HeapIn,HeapIn}, \text{HeapIn} \rangle \rightarrow \text{HeapIn}
\]

\[
\text{arbitrate } \text{inTup} = \text{case } \text{inTup of}
\]

\[
\langle \text{Write } l, _ \rangle, _ \rightarrow \text{Write } l
\]

\[
\langle _, \text{Write } l, _ \rangle \rightarrow \text{Write } l
\]

5.2.2 Collecting append’s initial arguments

We use a trick to capture the list pointers to be passed to the core append function: our builders (the processes responsible for constructing on the heap the two lists to be appended) issue InNOPs after their last Write; thus, a builder sends its first InNOP in the same cycle that hOut carries one of append’s initial pointer arguments.

We use this observation to define getPtr, which maintains the state of append’s pointers with a tuple of Maybe Addr types. The initial (Nothing,Nothing) tuple indicates that neither pointer is ready; the args stream carries this tuple until the first builder terminates. We wrap that builder’s final pointer (p) in a Just and maintain the resultant tuple until the second builder sends its first InNOP. We pass the final tuple of Justs to append, which will issue a Read to commence execution. This generates a Rout, resetting args to a tuple of Nothings.

\[
sZipWith3 :: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow
\]

\[
\text{Stream } a \rightarrow \text{Stream } b \rightarrow
\]

\[
\text{Stream } c \rightarrow \text{Stream } d
\]

\[
sZipWith3 f (a1 \triangleright \ldots) (b1 \triangleright \ldots) (c1 \triangleright \ldots) =
\]

\[
f a1 b1 c1 \triangleright \ldots
\]

\[
\text{args} :: \text{Stream } \langle \text{Maybe Addr,Maybe Addr} \rangle
\]

\[
\text{args} = \text{sZipWith3 getPtr allInputs hOut}
\]

\[
\text{getPtr} :: \langle \text{HeapIn,HeapIn}, \text{HeapIn} \rangle
\]

\[
\rightarrow \text{HeapOut}
\]

\[
\rightarrow \langle \text{Maybe Addr,Maybe Addr} \rangle
\]

\[
\rightarrow \langle \text{Maybe Addr,Maybe Addr} \rangle
\]

\[
\text{getPtr hInputs output prevArgs} = \text{case output of}
\]

\[
\langle \text{INOP}, \text{Write } l, _ \rangle, _ \rightarrow \text{case } \text{prevArgs of}
\]

\[
\langle \text{Nothing,Nothing} \rangle \rightarrow \langle \text{Just } p, \text{Nothing} \rangle
\]

\[
- \rightarrow \text{prevArgs}
\]

\[
\langle \text{INOP}, \text{INOP}, _ \rangle \rightarrow \text{case } \text{prevArgs of}
\]

\[
\langle \text{Just } arg1, \text{Nothing} \rangle \rightarrow \langle \text{Just } arg1, \text{Just } p \rangle
\]

\[
- \rightarrow \text{prevArgs}
\]

\[
- \rightarrow \langle \text{Nothing,Nothing} \rangle
\]

5.2.3 Encoding the heap’s behavior

Our heap-reliant functions depend on information hidden within the controller: which command the arbiter selected, the result of that command, and the state of append’s initial arguments. We introduce three new data types to encode this information.

The Message data type uses three variants to communicate with individual functions: Ack and Nack indicate whether a function received heap access on the previous cycle or not, while Ready carries the initial pointer arguments to append.

\[
\text{data } \text{Message} = \text{Ready Addr Addr}
\]

\[
| \text{Ack} |
\]

\[
| \text{Nack} |
\]

We assign Messages to functions with the Controls data type: the first two Messages are for the builders and the
third is for \textit{append}. These functions always inspect their assigned Message field, ignoring the others.

data \textit{Controls} = \text{C} \text{Message} \text{Message} \text{Message} \\

The Master data type carries these controls along with the heap’s output to each function.

data \textit{Master} = M \text{Controls} \text{HeapOut} \\

The \textit{cmdGen} function generates the appropriate messages on each cycle, yielding a stream of Controls (controls). If both builders have terminated, we collect their final pointers in a Ready message for \textit{append}. Otherwise, we use \textit{allInputs} and our arbitration scheme (prioritize the builders) to determine which function obtained heap access on the previous cycle; we send an Ack to that function and Nacks to the other two. The initial C Nack Nack Nack indicates that no function accessed the heap before the first cycle.

data \textit{Controls} = C \text{Message} \text{Message} \text{Message} \\

cmdGen :: ((\text{HeapIn},\text{HeapIn}),\text{HeapIn}) \\
    \rightarrow (\text{Maybe Addr},\text{Maybe Addr}) \\
    \rightarrow \text{Controls} \\

cmdGen \text{inTup} \text{argTup} = \text{case argTup of} \\
    (\text{Just p1},\text{Just p2}) \rightarrow \text{C Nack Nack Nack (Ready p1 p2)} \\
    _ \rightarrow \text{case inTup of} \\
        (\text{Write },_,\_) \rightarrow \text{C Ack Nack Nack} \\
        (_,\text{Write }\_) \rightarrow \text{C Nack Ack Nack} \\
        _ \rightarrow \text{C Nack Nack Ack} \\

We bundle \textit{controls} with \textit{hOut} to form the heap controller’s output stream \textit{cOut}:

cOut :: Stream \textit{Master} \\
cOut = \text{ZipWith} cmdGen \text{allInputs} \text{args} \\

cmdGen :: ((\text{HeapIn},\text{HeapIn}),\text{HeapIn}) \\
    \rightarrow (\text{Maybe Addr},\text{Maybe Addr}) \\
    \rightarrow \text{Controls} \\

cmdGen \text{inTup} \text{argTup} = \text{case argTup of} \\
    (\text{Just p1},\text{Just p2}) \rightarrow \text{C Nack Nack Nack (Ready p1 p2)} \\
    _ \rightarrow \text{case inTup of} \\
        (\text{Write },_,\_) \rightarrow \text{C Ack Nack Nack} \\
        (_,\text{Write }\_) \rightarrow \text{C Nack Ack Nack} \\
        _ \rightarrow \text{C Nack Nack Ack} \\

We bundle \textit{controls} with \textit{hOut} to form the heap controller’s output stream \textit{cOut}:

cOut :: Stream \textit{Master} \\
cOut = \text{ZipWith} M \text{controls} \text{hOut} \\

Here is the block diagram of the full controller:

![Block diagram of the full controller]

5.3 Communicating with the controller

The builders use the heap controller’s output to generate their input streams. Both send Write \textit{Nil} commands until receiving an Ack, at which point they use the heap’s output to determine which element to write next. This functionality relies on previous design decisions (the simple write-increment heap pointer, fully building one list argument before the other) and the fact that both of \textit{append}’s arguments were hardcoded in our example.

\textit{b1} :: Stream \textit{HeapIn} \\
\textit{b1} = \text{ZipWith} build1 \textit{cOut} (Write \text{Nil} \triangleright \textit{b1})
Once execution has begun, `append` inspects the heap’s output and the previous action to determine its current state. If the previous Call read out a Cons, we generate another Call that wraps the data \( x \) in a continuation and passes the next pointer \( xs \) to the heap. Otherwise, we use a Read to carry the second list’s pointer \( y \) in a Ret; we ignore the Rout on the following cycle, instead extracting \( y \) from the passed Read to generate the first new Cons. Subsequent Rets use the pointers \( p \) from the heap to construct new Cons cells until the stack pops a C2 continuation.

\[
append :: \text{Master} \rightarrow \text{Action} \rightarrow \text{Action} \\
append \text{ controller action} = \text{case controller of} \\
M \ (C \_ \_ \ \text{cmd}) \ hOut \rightarrow \text{case cmd of} \\
\text{Nack} \rightarrow \text{action} \\
\text{Ready 11 12} \rightarrow \text{Call 11 12 C2} \\
\text{Ack} \rightarrow \text{case heapOut of} \\
\text{Rout 1} \rightarrow \text{case action of} \\
\text{Call _ y _} \rightarrow \text{case 1 of} \\
\text{Nil} \rightarrow \text{Ret C2 (Read y)} \\
\text{Cons x xs} \rightarrow \text{Call xs y (C1 x)} \\
\text{Ret (C1 x) retVal} \rightarrow \text{case retVal of} \\
\text{Read y} \rightarrow \text{Ret C2 (Write (Cons x y))} \\
\text{-} \rightarrow \text{NOP} \\
\text{NOP} \rightarrow \text{NOP} \\
\text{Wout p} \rightarrow \text{case action of} \\
\text{Ret (C1 x) retVal} \rightarrow \text{Ret C2 (Write (Cons x p))} \\
\text{-} \rightarrow \text{NOP}
\]

Figure 2 depicts the behavior of our final program. We again remove components of various stream elements for clarity’s sake: we omit the HeapOut data constructors from the `hOut` stream, distribute the messages from `cOut` into three message streams `M1`, `M2`, and `M3`, and remove Maybe constructors from the `args` stream (underscores represent Nothings).

We focus on the builders for the first 6 cycles. The blue components in a column specify the List written to the heap on that cycle (the second field of each Cons is taken from the `hOut` stream). When a builder issues its first InNOP, we collect the pointer from the `hOut` stream and store it in the `args` tuple (cycles 3 and 5). After cycle 5, both builders issue InNOPS, `args` resets to a tuple of Nothings, and `M1` and `M2` carry Nacks. We thus omit these streams from the rest of the table.

The controller passes the `args` pointers to `append` in cycle 6 (we use tuple notation instead of a Ready), generating `appendStream`’s first Call. The Lists read out by each Call are highlighted in red; the data and pointer from a Cons appear in `memOps` and `heapOps`, respectively. When a Nil is read out in cycle 9, `append` generates a Ret with a Read 4 heap operation; we show this pointer again in cycle 10, here highlighted in blue, to indicate its use in the first new Cons.

In cycle 12, `hOut` carries the pointer representing the final list (6), and `compFlow` carries a C2 continuation in a Ret (shown below the table in Figure 2). The `output` extracts this pointer and returns it as the result of our program. Future research will concern the implementation of a “heap interpreter” that uses this pointer to reconstruct the full list from the heap; this will validate that our transformations do not modify the underlying computation of the original program.

\[
\text{main :: Addr} \\
\text{main }= \text{output cOut compFlow}
\]
6. CONCLUSIONS

We have presented a set of transformations that convert a recursive Haskell program operating on recursive data types into a form suitable for simple, syntax-directed translation to SystemVerilog. The process of removing functional recursion has already been automated in our Haskell-to-hardware compiler, so we focused on the modifications that are still under development: lifting a program into the world of Streams, introducing explicit memory operations, and defining communication protocols among a program’s components. The extended `append` example will serve as a template for future research on a hardware implementation of the heap and its controller.

7. REFERENCES


