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 concatentation 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
append :: List → List → List
```

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)
              (λresult → result)

append :: List → List → (List → List) → List
append z y k = case z of
  Nil       → k y
  Cons x xs → append xs y (λl → 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 variables as formal arguments; these precede the result argument 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 :: Int → (List → List) → List → List
c1 x k l = k (Cons x l)

c2 :: List → List
c2 result = result
```

We merge the continuation functions into *append* with the Continuation and Action data types: the former represents 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 Rets 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 polymorphic, 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 constructor "▷", 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 False True False False ..., and y is x delayed by a cycle: True True False False True 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 function's termination. The Start action triggers *append*'s first Call; this will be useful when we incorporate memories into our program.

```
data Action = Call List List Continuation
            | Ret Continuation List
            | Start
            | NOP

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.

```
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 l ▷ _) = l
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, intially 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 $i-1$st 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 (a1 ▷ a2 ▷ ...) (b1 ▷ b2 ▷ ...) =
  f a1 b1 ▷ f a2 b2 ▷ ...
```

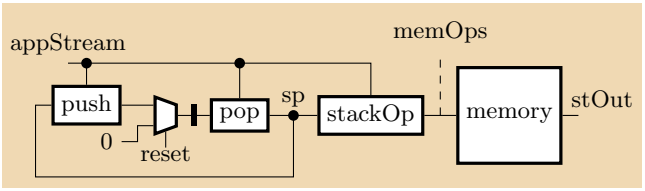
Each cycle, *stackOp* observes the current action and stack pointer to determine the appropriate memory operation: Calls write continuations, Rets read them, and other actions (Start and NOP) generate dummy reads where we ignore the output.

```
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

stOut	compFlow	appStream	memOps
C2	Start	Call ...	Write 0 C2
C2	Call [1,2] [3] C2	Call ...	Write 1 (C1 1)
C2	Call [2] [3] (C1 1)	Call ...	Write 2 (C1 2)
C2	Call [] [3] (C1 2)	Ret ...	Read 2
C1 2	Ret (C1 2) [3]	Ret ...	Read 1
C1 1	Ret (C1 1) [2,3]	Ret ...	Read 0
C2	Ret C2 [1,2,3]	NOP	Read 0

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.

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.

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

appStream :: Stream Action
appStream = sMap append compFlow

compFlow :: Stream Action
compFlow = sZipWith mergeStack
                    (Start ▷ appStream) stOut

mergeStack :: Action → Continuation → Action
mergeStack action k = case action of
  Ret _ n → Ret k n
  _       → action

main :: List
main = result compFlow
```

Figure 1 shows how these streams behave over time, with the order of columns following the flow of data. First, *append* uses *compFlow* to produce *appStream*, which the stack machinery interprets to generate *memOps*. We then feed *memOps* into the stack memory, producing *stOut* in the next cycle. Finally, *mergeStack* combines *stOut* and the last value of *appStream* to form *compFlow*.

Other than the continuations, *compFlow* is identical to *appStream* from Section 3; by design, introducing a stack has not changed the program's semantics.

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
            | InNOP

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.

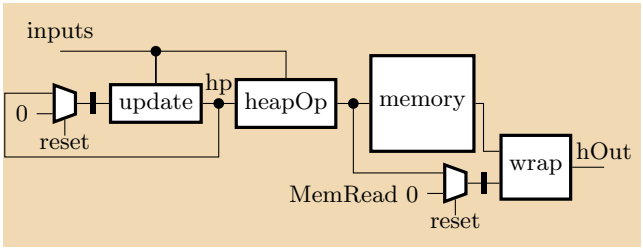
```
heapOps :: Stream (MemOp List)
heapOps = sZipWith heapOp inputs hp

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
```

```
update :: HeapIn → Addr → Addr
update input addr = case input of
  Write _ → addr+1
  _       → addr
```

The block diagram of our heap resembles our stack:



5.2 The heap controller

Before *append* can perform any computation, its intial 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* its 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 :: Stream a → Stream b → Stream (a,b)
sZip (a1 ▷ a2 ▷ ...) (b1 ▷ b2 ▷ ...) =
  (a1,b1) ▷ (a2,b2) ▷ ...

allInputs :: Stream ((HeapIn,HeapIn),HeapIn)
allInputs = sZip (sZip b1 b2) 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.

```
inputs :: Stream HeapIn
inputs = sMap arbitrate allInputs

arbitrate :: ((HeapIn,HeapIn),HeapIn) → HeapIn
arbitrate inTup = case inTup of
  ((Write l,_      ),_)     → Write l
  ((_      ,Write l),_)     → Write l
  (_                ,input) → input
```

5.2.2 Collecting append's intial 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 intial 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 → b → c → d) →
             Stream a → Stream b →
             Stream c → Stream d
sZipWith3 f (a1 ▷ ...) (b1 ▷ ...) (c1 ▷ ...) =
  f a1 b1 c1 ▷ ...

args :: Stream (Maybe Addr,Maybe Addr)
args = sZipWith3 getPtr allInputs hOut
                 ((Nothing,Nothing) ▷ args)

getPtr :: ((HeapIn,HeapIn),HeapIn)
       → HeapOut
       → (Maybe Addr,Maybe Addr)
       → (Maybe Addr,Maybe Addr)
getPtr hInputs output prevArgs = case output of
  Wout p → case hInputs of
   ((InNOP,Write _),_) → case prevArgs of
     (Nothing,Nothing)   → (Just p,Nothing)
     _                   → prevArgs
   ((InNOP,InNOP),_)   → case prevArgs of
     (Just arg1,Nothing) → (Just arg1,Just p)
     _                   → prevArgs
   _                     → prevArgs
  _      → (Nothing,Nothing)
```

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 intial 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*.

```
data Message  = Ready Addr Addr
              | Ack
              | Nack
```

We assign Messages to functions with the Controls data type: the first two Messages are for the builders and the

third is for *append*. These functions always inspect their assigned Message field, ignoring the others.

```
data Controls = C Message Message Message
```

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

```
data Master = M Controls HeapOut
```

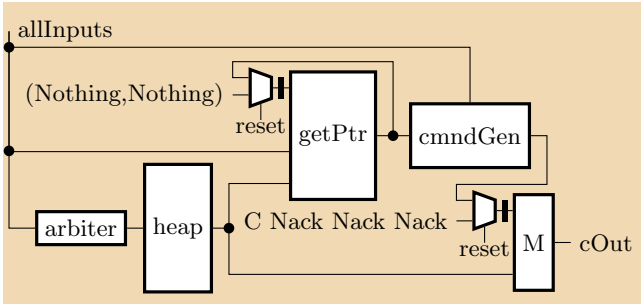
The *cmndGen* function generates the appropriate messages on each cycle, yielding a stream of Controls (*controls*). If both builders have terminated, we collect their final pointers in a Ready message for *append*. Otherwise, we use *allInputs* and our arbitration scheme (prioritize the builders) to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

```
controls :: Stream Controls
controls = C Nack Nack Nack ▷
           sZipWith cmndGen allInputs args

cmndGen :: ((HeapIn,HeapIn),HeapIn)
        → (Maybe Addr,Maybe Addr)
        → Controls
cmndGen inTup argTup = case argTup of
  (Just p1,Just p2) → C Nack Nack (Ready p1 p2)
  _ → case inTup of
        ((Write _,_),_) → C Ack Nack Nack
        ((_,Write _),_) → C Nack Ack Nack
        _               → C Nack Nack Ack
```

We bundle *controls* with *hOut* to form the heap controller's output stream *cOut*:

```
cOut :: Stream Master
cOut = sZipWith M controls hOut
```

Here is the 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 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 *append*'s arguments were hardcoded in our example.

```
b1 :: Stream HeapIn
b1 = sZipWith build1 cOut (Write Nil ▷ b1)

build1 :: Master → HeapIn → HeapIn
build1 controller hIn = case controller of
  M (C Nack _ _) _         → hIn
  M (C Ack  _ _) (Wout 0) → Write (Cons 2 0)
  M (C Ack  _ _) (Wout 1) → Write (Cons 1 1)
  _                        → InNOP

b2 :: Stream HeapIn
b2 = sZipWith build2 cOut (Write Nil ▷ b2)

build2 :: Master → HeapIn → HeapIn
build2 controller hIn = case controller of
  M (C _ Nack _) _         → hIn
  M (C _ Ack  _) (Wout 3) → Write (Cons 3 3)
  -                        → InNOP
```

Before updating *append* and its associated functions, we modify the Action type to reflect the heap's presence. We remove the Start action entirely; the Ready message serves the same purpose. The heap supplies *append*'s list arguments, so we replace Call's list fields with pointers. We cannot modify Rets similarly; the first Ret generated carries a pointer, while the rest carry new list elements. We use the Write and Read variants of the HeapIn type to encode this distinction: Writes pass new list elements directly to the heap, while a Read carries the first Ret's pointer back into *append*.

```
data Action = Call Addr Addr Continuation
            | Ret Continuation HeapIn
            | NOP
```

The *getOp* function translates these actions into heap inputs. Calls use their pointers to issue Reads, Rets already carry the appropriate heap command, and NOP actions correspond to InNOP heap inputs.

```
funcOps :: Stream HeapIn
funcOps = sMap getOp appStream

getOp :: Action → HeapIn
getOp action = case action of
  Call ref _ _ → Read ref
  Ret _ heapIn → heapIn
  NOP          → InNOP
```

We connect *append* to the heap controller's output by passing *cOut* as another stream argument and substituting *sZipWith* for *sMap*. Since the Start action no longer exists and *append* cannot execute until its arguments are ready, we use NOP as *compFlow*'s intial action.

```
appStream :: Stream Action
appStream = sZipWith append cOut compFlow

compFlow :: Stream Action
compFlow = sZipWith mergeStack
                    (NOP ▷ appendStream) stOut
```

The *append* function now determines the next action based on up to four factors: the previous action, a message from the heap controller, the heap's ouput, and a continuation from the stack. The control message takes precedence: a Nack tells *append* to repeat its previous action, Ready triggers *append*'s first Call, and an Ack indicates that *append* should generate its next action.

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 :: Master → Action → Action
append controller action = case controller of
  M (C _ _ cmnd) heapOut → case cmnd of
   Nack                  → action
   Ready l1 l2           → Call l1 l2 C2
   Ack → case heapOut of
    Rout l → case action of
     Call _ y _ → case l of
        Nil             → Ret C2 (Read y)
        Cons x xs       → Call xs y (C1 x)
     Ret (C1 x) retVal → case retVal of
        Read y          → Ret C2 (Write (Cons x y))
        _               → NOP
     NOP                → NOP
    Wout p → case action of
     Ret (C1 x) retVal  → Ret C2 (Write (Cons x p))
     _                  → 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 *appStream*'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.

```
main :: Addr
main = output cOut compFlow
```

Cycle:	*0*	*1*	*2*	*3*	*4*	*5*
hOut:	Nil	0	1	2	3	4
M1:	Nack	Ack	Ack	Ack	Nack	Nack
b1:	Nil	Cons 2	Cons 1	InNOP	InNOP	InNOP
M2:	Nack	Nack	Nack	Nack	Ack	Ack
b2:	Nil	Nil	Nil	Nil	Cons 3	InNOP
args:	(_,_)	(_,_)	(_,_)	(2,_)	(2,_)	(2,4)
heapOps:	W	W	W	W	W	R
	0	1	2	3	4	0

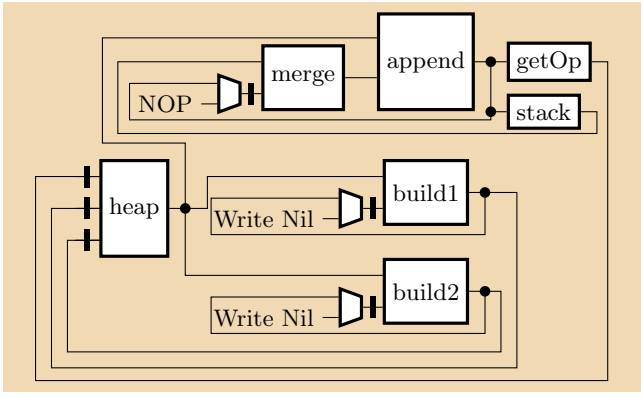
Cycle:	*6*	*7*	*8*	*9*	*10*	*11*
hOut:	Nil	Cons	Cons	Nil	Cons	5
stOut:	C2	C2	C2	C2	C1 2	C1 1
compFlow:	NOP	Call	Call	Call	Ret (4)	Ret
M3:	(2,4)	Ack	Ack	Ack	Ack	Ack
appStream:	Call	Call	Call	Ret	Ret	Ret
memOps:	W	W	W	R	R	R
	0	1	2	2	1	0
	C2	C1 1	C1 2			
heapOps:	R	R	R	R	W	W
	2	1	0	4	5	6
					Cons	Cons

```
compFlow = NOP ▷ NOP ▷ NOP ▷ NOP ▷ NOP ▷ NOP
         ▷ NOP
         ▷ Call 2 4 C2
         ▷ Call 1 4 (C1 1)
         ▷ Call 0 4 (C1 2)
         ▷ Ret      (C1 2) (Read 4)
         ▷ Ret      (C1 1) (Write (Cons 2 4))
         ▷ Ret      C2     (Write (Cons 1 5))
         ▷ ...
```

Figure 2: The behavior of our stream-based program using a stack and a heap. The table shows the abbreviated contents of the streams encapsulating the program's behavior; *compFlow* is shown in detail below.

```
output :: Stream Master → Stream Action → Addr
output (M _ (Wout p) ▷ _) (Ret C2 _ ▷ _) = p
output (_ ▷ s1) (_ ▷ s2) = output s1 s2
```

Here is the block diagram of the final program:



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

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

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

[3] 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.

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

[5] 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.

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

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

[8] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards. Hardware synthesis from a recursive functional language. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Amsterdam, The Netherlands, Oct. 2015.