The FHW Project: High-Level Hardware Synthesis from Haskell Programs

Stephen A. Edwards
with
Martha A. Kim, Richard Townsend, Kuangya Zhai, and Lianne Lairmore

August, 2019

1 Introduction

The goal of the FHW project was to produce a compiler able to translate programs written in a functional language (we chose Haskell) into synthesizable RTL (we chose SystemVerilog) suitable for execution on an FPGA or ASIC that was highly parallel. We ultimately produced such a compiler, relying on the Glasgow Haskell Compiler (GHC) as a front-end and writing our own back-end that performed a series of lowering transformations to restructure such constructs as recursion, polymorphism, and first-order functions, into a form suitable for hardware, then transform the now-restricted functional IR into a dataflow representation that is then finally transformed into synthesizable SystemVerilog.

We have not yet released our compiler in open-source form, but this is largely because we have not gone to the effort to make it feasible for others to install and run it. As of this writing, our compiler is tied to a very specific, now-deprecated version of GHC (7.6.3) which would be difficult for most users to install. We have plans to address this issue (e.g., through a portable VM image), but have not yet found the time to make a proper release.

2 History

I started thinking seriously about compiling Haskell to hardware while visiting Microsoft Research in Cambridge, UK in the summer of 2010. I had been invited by Satnam Singh, who introduced me to the "Lambda Corridor" of Simon Peyton Jones, Simon Marlow, Andrew Kennedy, and others. While I had caught the functional programming bug some years before, this experience solidified it to me: I saw functional programming as where programming languages will have to go to handle the challenge of efficient, correct parallelism. I still believe this, although there remain many, many challenges.

In my 2012 technical report [11], I showed a series of transformations to produce an implementation of the recursive Fibonacci number calculator in VHDL, much like the example in section 3.1 demonstrates, although we ultimately abandoned VHDL in favor of SystemVerilog because of the availability of superior tools such as the Verilator open-source simulator.

Later that summer, MS student Neil Deshpande [10] wrote a pass for GHC that experimented with limited inlining of recursive functions to expose greater parallelism. The idea is that recursive calls of a function are usually performed sequentially; inlining can make the body of a small recursive function bigger, allowing parts of it to run in parallel. Ultimately, Richard Townsend would implement this sort of inlining in the FHW compiler [31].

In 2013, I wrote a revised version [12] of my earlier report [11] in which I refined the procedure and switched to using SystemVerilog, the language we would ultimately target with our later compiler. Later that year, I wrote another report [13] that outlined the vision of the FHW project (much of this was taken from the NSF proposal), looked at performing classical high-level synthesis scheduling with resource binding in a functional setting (something that has yet to be realized in the FHW compiler), and discussed how inlining types and functions could increase parallelism, which Richard Townsend ultimately implemented as part of his thesis work [31].

Richard Townsend joined Columbia in 2013 as a Ph.D candidate and started working on the FHW project. His first paper [32], co-authored with FHW co-PI Martha Kim, looked at efficiently implementing "map" functions (a common idiom in functional programs) in parallel and balancing the number of processing units with buffer memory needed to handle variable execution times.

We published our first major paper on FHW in 2015 at CODES+ISSS [35]. Our main contribution in this paper was our technique for compiling away recursion (which we had described in earlier technical reports), which we describe in detail below in section 3.1. The authors of this paper also include Kunagya Zhai and Lianne Lairmore, who also joined Columbia in 2013 as Ph.D candidates to start working on the project, but did not complete the program.

Also in 2015, we published a paper at MEMOCODE [5] were we took the first step toward the hardware dataflow implementation we ultimately adopted as our IR. In this paper, we define a simple valid/stop flow-control protocol and show how "patient" blocks can communicate safely by communicating through input and output buffers that speak this protocol. In this paper, we use this technique to implement a range partitioning pipeline we designed for a hardware database accelerator, which is not directly part of the FHW project, but we later adopted this technique.

Our next two papers appeared in 2017 and reflected the transition to a dataflow intermediate representation. The first, which we presented at Compiler Construction [34], describes the translation from our higher-level IR (a dialect of the GHC Core dubbed "Floh") into our simple dataflow IR. Challenges in this translation include groups of mutually recursive functions, which require a "lock" node at the inputs to prevent multiple simultaneous invocations of such functions; literal constants, for which tokens are generated only when "control" reaches them during execution through a network of valueless "Go" channels; and pattern-matching *case* expressions, which require steering logic so that only one branch is ever activated. We started with Arvind and Nikhil's 1990 paper [2] that generates acyclic dataflow networks as a functional program is running (e.g., from recursion) and adapted it to our static setting in which the network is fixed when the program is compiled and uses cycles in the dataflow graph to handle recursive calls.

Our second 2017 paper [15], a follow-up to our 2015 MEMOCODE paper, presents the details of a hardware implementation of our dataflow networks. We show how to implement nodes such as multiplexers that have non-uniform (i.e., data-dependent) token consumption and production behavior ("firing rules"), something we did not consider in our 2015 paper. The other big problem was avoiding inadvertently introducing combinational cycles in the circuit. In general, we want connected blocks to communicate combinationally (in the same cycle) and only want to introduce cycles of latency when absolutely necessary to meet clock frequency constraints. This suggests combinational handshaking logic, but introducing handshaking between arbitrary combinational blocks with complex firing rules quickly leads to combinational cycles. In particular, a naïve implementation of a fork followed by a strict operator (that requires both inputs, such as an addition), appears to produce a combinational

cycle. Our solution prohibits combinational communication from any node's *ready* network (backpressure) to its *valid* network. One unexpected implication of this approach is the need for *fork* nodes to fire across multiple cycles (e.g., copy the input token to the first output in the first cycle and copy it to the second output and consume it in the second cycle).

In 2019, we published an extended version of the hardware dataflow work [14] that includes a description of the textual dataflow format we use to communicate between the FHW compiler proper and the SystemVerilog back end. One notable feature of this format is a novel, formal way of characterizing dataflow blocks with variable numbers of inputs and outputs. We define a type system that characterizes such blocks and check that the generated dataflow IR from our compiler generates well-typed networks.

Richard Townsend's 2019 Ph.D Thesis [31] is a far more thorough summary of the FHW project than this report. He goes over the details of how we translate GHC Core into our IR, e.g., by eliminating recursion as discussed earlier, describes the translation of this IR into a dataflow network, then shows how these networks are implemented in RTL SystemVerilog. He also describes two optimization procedures: one that duplicates the computational core of a divide-and-conquer recursive procedure to improve parallelism, and one that packs recursive data types into denser objects that improves data locality and hence performance by reducing the number of memory transfers.

3 Contributions

3.1 *Recursion to Tail Recursion*

Recursive functions are fundamental to functional programming languages, yet the richer behavior they produce is rarely supported by existing HLS tools, which generally only accept nested loops with affine array indices [18, 19]. One exception is Ghica et al. [17], who implement local variables in recursive functions with register files that function as stacks to hold functions' activation record data in a distributed fashion.

We transform recursive functions into tail-recursive functions that explicitly manipulate a stack. An advantage of the functional approach is that this transformation can be shown correct as a series of well-known semantics-preserving operations. We first proposed this technique in a series of technical reports in 2012 [11] and 2013 [12] before publishing it at CODES+ISSS in 2015 [35]. Townsend [31, §3.3.4] summarizes our technique.

The following example (Fibonacci) from Zhai et al. [35] illustrates the steps. Starting from a recursive function consisting of a *case* with two recursive calls,

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

we transform the program into the well-known continuation-passing style (CPS) [16, 29, 1], which imposes a sequential order on the two recursive calls and transforms all recursive calls to tail calls at the expense of introducing lambda expressions and function arguments:

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

Here, *k* is the continuation — the function to which the result is passed.

Next, we perform lambda lifting [21] to capture non-local variables as function arguments:

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

then name each lambda expression (i.e., *ko*, *k1*, ...) and hoist it to the top level:

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

Finally, we perform defunctionalization [7] and introduce an algebraic data type (*Cont*) to encode the *k* arguments and avoid the need to pass actual functions as arguments:

```
data Cont = Ko | K1 Int Cont | K2 Int Cont

call n k = case n of 1 → ret k 1
                     2 → ret k 1
                     _ → 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)
                    Ko       → r
fib n = call n Ko
```

The program is now in a form well-suited to implementation in hardware: each function consists of pattern matching followed by some arithmetic and a tail call to another explicit function.

The *Cont* type functions as a stack of activation records. The various *K*'s encode the return address and the *ret* function decodes them and executes the code at each return address. Creating an element of the *Cont* type amounts to pushing an activation record on the stack.

Townsend [31, §3.3.4] provides more details about our technique, which also handles mutually recursive functions.

3.2 Coat Check: the Functional Memory Paradigm

A goal of this project has always been to experiment with synthesizing memory systems that are tuned for specific algorithms. We chose early in the project to adopt an immutable memory model, which would free us from the sequential memory consistency issues plaguing parallel programs on modern multi-core processors. We were confident this model was rich enough based on observations of pioneers such as Dennis [9, 8] along with the success of Haskell. Furthermore, we knew of the work of Bacon et al. [3, 4], which demonstrated garbage collection in pure hardware was realistic.

Thinking of memory operations in a purely functional setting naturally leads to a "coat check" model in which write operations are the inverse of read operations. In this model, read operations behave in the familiar way: *read* is given a pointer to an object and returns a copy of the requested object. Write, however, behaves more like the C++ *new* operator: data is passed to *write*, which returns a pointer to it. This is like checking your coat at a restaurant: when you turn in your coat, you're given a number that will let you retreive it later. In Haskell types,

```
read  :: Pointer → Data
write :: Data    → Pointer
```

The coat check memory model eliminates read-after-write hazards provided the memory system never returns a pointer to an object before it has successfully stored the object. This effectively turns the object sharing problem into simple data dependency, which is a fundamental issue that has to be solved anyway.

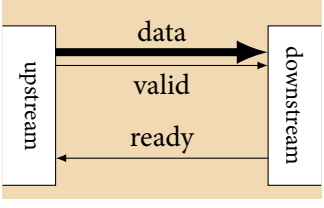
The immutable memory model similarly eliminates write-after-read hazards since objects are never written after being created, so there is no danger of a read picking up an earlier version of the data than it should because there can only ever be one version.

The immutable memory model leads to different ways of thinking about the role of memory and data in programs. For instance, the coat check memory model can also be thought of as a sort of extreme data compression scheme: arbitrarily large objects can be "compressed" down to a code word the size of a pointer object; reading an object from memory amounts to consulting a code book. Garbage collection amounts to knowing which codes have been forgotten and can be reused, or equivalently, which codes will never be used again.

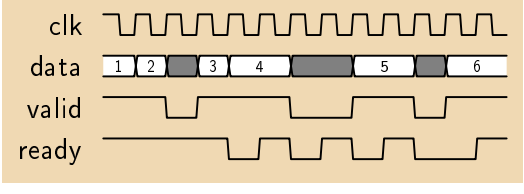
Our compiler translates its functional Core-like intermediate language into a dataflow representation that can be easily translated into efficient parallel hardware with distributed flow control. We first proposed these sorts of dataflow networks at MEMOCODE in 2015 [5], then refined our technique to support conditional and other actors in 2017 [15].

While many have proposed implementing dataflow networks in hardware, notably Li et al. [24], whose latency-insensitive blocks served as inspiration for our work, and Cortadella et al. [6], we believe ours are the first to be compositional in the sense that combinational blocks with flow control may be connected in feed-forward networks without fear of introducing spurious combinational cycles in the fundamentally cyclic handshaking logic.

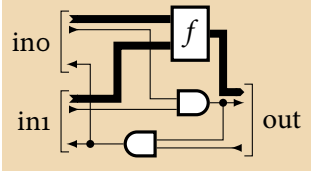
Our dataflow blocks communicate using a simple flow-control protocol that indicates the validity of a data token (*valid*) and provides backpressure (i.e., when *ready* is false).



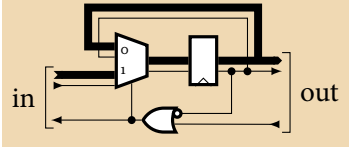
valid	ready	Meaning
0	–	No token to transfer
1	1	Token transferred
1	0	Token valid but not consumed



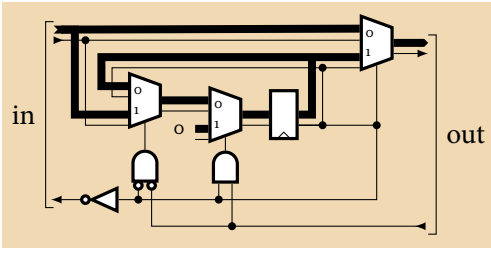
In our communication protocol, in each cycle, *valid* indicates data is available upstream; *ready* indicates downstream is capable of consuming data immediately. A token is transferred only when both are asserted; upstream must otherwise hold any valid data.



A unit-rate actor such as an adder consists of the combinational function f and two flow-control gates that indicate the output is valid only when both inputs are and that the inputs are consumed only when the output is valid and downstream is ready.

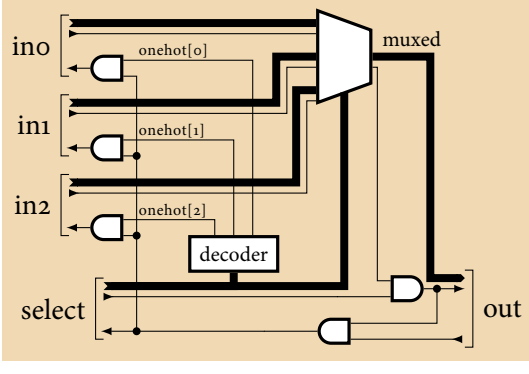


A data buffer, effectively just a familiar pipeline buffer, breaks combinational paths in the data/valid network by introducing a cycle of latency. The input is ready when the output is ready or the buffer is empty.

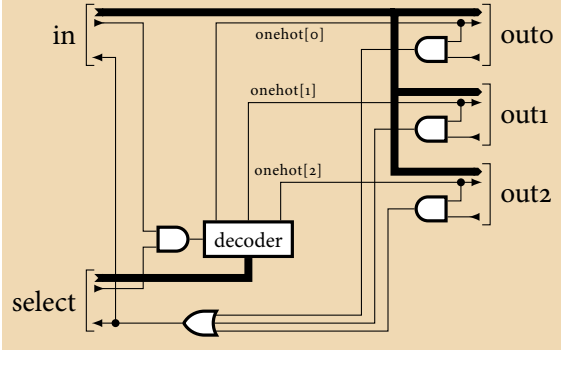


A control buffer breaks combinational paths in the upstream *ready* network by providing a "spill buffer." Nominally, the input is ready and any input token flows immediately to the output. However, if the downstream is not ready, the buffer cannot announce this in the current cycle because to do so would require a combinational path from output *ready* to input *ready*. In this case, the spill buffer captures any incoming token and holds it to the next cycle in which the downstream is ready. In cycles where the buffer holds a token, the input is not ready.

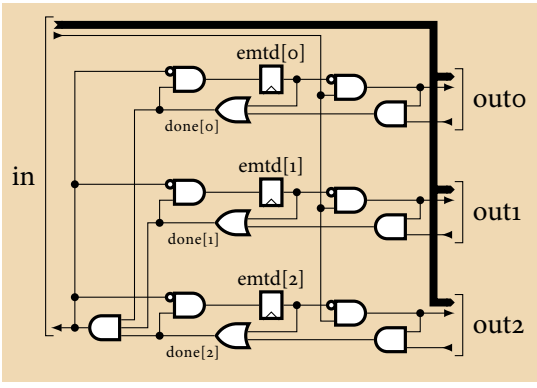
Our nodes may be connected arbitrarily provided each output goes to exactly one input and vice versa and each cycle in the communication network has at least one data buffer and one control buffer. Introducing additional buffers may affect performance (the clock rate and/or cycle latency), but will not induce deadlocks.



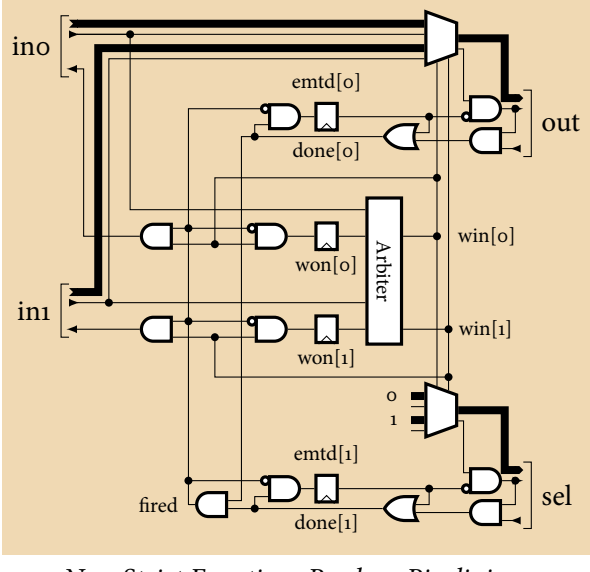
A multiplexer takes a *select* input and uses it to steer the data and *valid* signals from the selected input to the output when *select* is also valid. If the downstream is ready, the token on *select* and the corresponding input is consumed.



A demultiplexer is mostly handshaking. The data fans out, but at most one output is valid when the input and *select* channels have data. When they do, *select* sets which output is valid, and the input and *select* tokens are consumed only when the selected output is also ready.



The fork node is the most subtle of our nodes. To prevent inadvertent combinational cycles when nodes are connected, we prohibit combinational paths from the *ready* network to the *valid* network. Unfortunately, this precludes the most natural design for *fork*, which would wait until all outputs are ready before presenting valid output tokens. Instead, our *fork* sends a valid input token to all outputs that have not already consumed the input token (each output has a flip-flop that remembers whether the output previously consumed a token). Only after all the outputs have consumed the input token is it finally consumed and the process repeats.

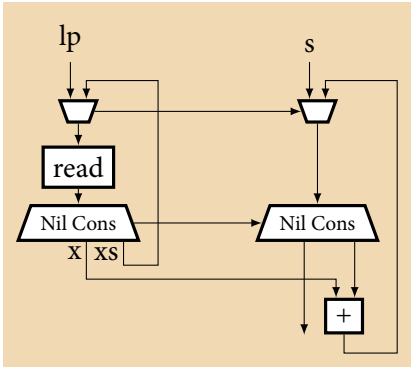


The nondeterministic merge node is complex because it also functions like a *fork*. An arbiter looks at the *valid* bits on the inputs to decide which input will be passed to the output. A *sel* output emits a token that records this choice. Complexity arises when one of these is not ready when the arbitration is completed. In that case, the node needs to remember which input won the arbitration and whether the output, *sel*, or both have not yet consumed the tokens emitted because of an arbitration.

3.4 Non-Strict Functions Produce Pipelining

In developing our dataflow translation [34], we found loop pipelining can be recovered by allowing tail-recursive functions to be non-strict. In classical software loop pipelining, the next iteration of a loop starts execution before the previous iteration has completely finished. In our setting, starting the next execution of a tail-recursive function before all its arguments have been completed produces a similar result. Consider the following example:

```
sum lp s = case read lp of
             Nil        → s
             Cons x xs  → sum xs (x + s)
```

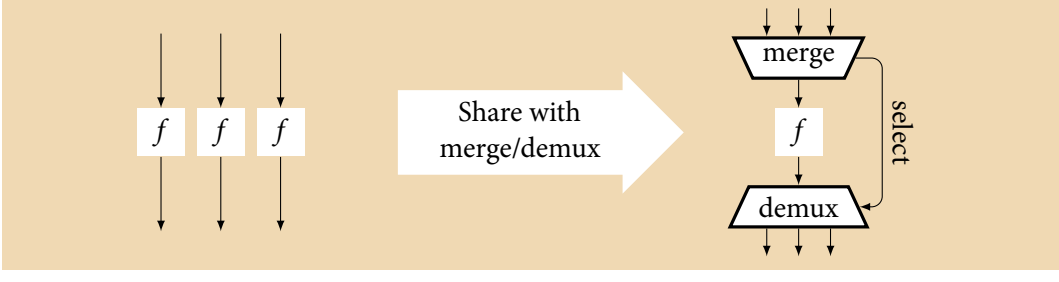


This tail-recursive function sums the elements of a list in the accumulator argument *s*. The dataflow network we generate for this function contains two loops, one per function argument. The *lp* loop walks down the list, feeding each element to the *s* loop, which accumulates the elements and ultimately emits their sum.

Because information only flows one direction between the loops (i.e., from the *lp* loop to the *s* loop), inserting buffers on the channels between decouples the two loops, allowing the *lp* loop to race ahead and start reading the next list element before the addition operation has been completed. This is exactly pipelining: the next invocation of *sum* starts before the previous one has completed. Townsend [31, §4.5.2] explores how the presence of such non-strict functions improves performance.

3.5 *Resource Sharing the Root of All Evil*

When building a static dataflow network in which some function *f* is called non-recursively in multiple places, there are at least two options: inline each call of *f*, which duplicates the circuitry in the body of *f*; or share a single copy of *f* across the various call sites. The former typically requires far more area; exponentially more in the worst case, so we rarely do this. The latter is difficult to implement in true Kahn dataflow because it requires the system to decide the order in which the various call sites should be serviced. We were inspired by Sharp and Mycroft [28, 25, 26] to take the approach depicted below and described more in our 2017 CC paper [34] and Townsend's thesis [31, §4.3.2].



Here, three calls to a function *f* share a single functional unit for *f*. A nondeterministic merge node selects and delivers one input to *f* and also reports which input won each arbitration on a *select* channel. Once *f* processes the input, a demultiplexer block routes the result back to the appropriate call site based on the *select* token.

This seems safe, but already we found one subtle condition under which it would cause deadlocks. As shown, a single-input function is unlikely to introduce deadlocks, but when we

coupled this approach with the non-strict functions we described above, we found it could lead to deadlock. The problem was that our networks, as described in our 2017 CC paper [34], placed a nondeterministic merge actor on the first argument and used the results from the arbitration to control muxes on all further arguments. With this arrangement, code such as

```
f 1 (f 2 3)
```

could deadlock. If the "1" reaches *f* first, it would win the arbitration and force *f* to compute its outer call first. However, that call needs the result of the inner call of *f* to complete, which would never happen because inner call is blocked indefinitely by the choice made by the arbiter.

We resolved this problem by forcing non-recursive calls to always be strict, as explained in Townsend's thesis [31, §4.3.2], but this seems needlessly heavy-handed. It is an open question how better to handle this problem.

4 Regrets

4.1 GHC External Core

"Adopt or build?" is the perennial question facing software developers and we were no exception. In part from advice from others, we decided to adopt an existing compiler as a front-end and have it output its intermediate representation after optimization as input into our tools.

We had decided early on to use Haskell as a source language and GHC in particular as the base compiler for a variety of reasons. Other possible choices would have been one of the Standard ML compilers or OCaml, but we chose GHC because of its advanced state of development, which continues to be ongoing; the functional purity of the language (ML and OCaml allow side-effects and, while they isolate them with the type system, we did not understand the mechanism well and feared that the mutable memory model was deeply baked into the system); and the richer features of Haskell, including type classes, user-defined operators, and its "imperative" *do* notation.

We specifically chose not to directly adopt Haskell's lazy semantics, which allows programmers to code things like infinite lists but then only examine the first few elements. While a very interesting aspect of Haskell, we felt the bookkeeping involved in executing lazy semantics (which relies heavily on function pointers in every data structure [22, 23]) was not appropriate for hardware. Later, we relaxed this requirement to allow tail-recursive calls to be non-strict to enable pipelining, but doing so did not require additional bookkeeping.

Haskell's intermediate represention, known as *core*, is simple and well-documented [27]. It is essentially the Lambda Calculus (function application) with *let* bindings and pattern-matching *case* constructs, which simplifies syntax-directed translation into hardware because there are so few constructs to be considered.

Sitting beside *core*, however, is GHC's elaborate and growing type system, which now includes things like generalized algebraic data types, which are outside the scope of what we wanted to handle in hardware. In looking at the source, we quickly learned that the data

structures used in GHC's type system were far too complicated (and poorly documented) for our purposes.

We turned instead to Tolmach and Chevalier's External Core [30], a package developed to more-or-less do exactly what we wanted: provide a well-documented, simple interface that exported GHC's core. The quality of this decision has been mixed at best. While it did do more-or-less what we needed it to, support for External Core was dropped from GHC in April 2014, not long after we decided to adopt it, but long enough so that we had built a significant code base that depended on it.

There were also obscure bugs in the External Core system: compiling certain files would produce malformed external core files (i.e., that would not parse properly). This was a minor problem, however, and only showed up in the standard libraries, which we had to modify anyway. We were able to figure out what constructs were causing the problem and change the source to avoid them.

The main problem, however, is that GHC has moved beyond External Core so our compiler remains tied to the very specific version of GHC that was available when we started development (GHC 7.6.3, the version provided with Ubuntu 16.04). As a result, it is difficult to release our compiler in a form that others could use it, since they're likely to have a much newer version of GHC installed and installing an older version is often difficult.

There are ways to work around this problem, such as to update External Core to work with newer versions of GHC or run the older version of GHC in a lightweight virtual environment like Docker. Ironically, the GHC system introduced the *Stack* build tool, a mechanism for managing reproducible builds by allowing users to select specific GHC versions, library versions, etc. However, *Stack* only supports GHC versions 7.8.3 and later, which do not have External Core support.

4.2 Haskell Intrinsics and the Standard Library

One of Haskell's strengths is its extreme malleability. Features that are baked into most languages, such as numeric types and arithmetic operators, are instead coded in the language itself in standard libraries such as the Haskell *Prelude*, which includes such basic types as *Int* and operators such as +.

This flexibility is very convenient for programmers, who can make use of it through new or existing libraries, but it makes life difficult for fledgling Haskell compiler writers.

Consider the innocuous-looking program that defines a single function that adds two to its integer argument.

```
add2 :: Int → Int
add2 x = x + 2
```

This little bit of code relies on the definition of the *Int* type, which is a wrapper for the built-in *Int#* class, and the *Num* typeclass—part of the Standard Prelude—of which the *Int* type is an instance. *Num* defines the familiar arithmetic operators excluding division.

```
data Int = I# Int#

infixl 7 *
infixl 6 +, -

class Num a where
    (+), (-), (*)        :: a → a → a
    negate               :: a → a
    abs                  :: a → a
    signum               :: a → a
    fromInteger          :: Integer → a

instance Num Int where
    I# x + I# y = I# (x +# y)
```

When it compiles the Standard Prelude, GHC names the + function in the *Num Int* typeclass $fNumInt_$c+. In External Core syntax (which is explicitly typed), it becomes

```
$fNumInt_$c+ :: Int → Int → Int =
  λ (x :: Int) (y :: Int) → case Int x of
                              I# (xx :: Int#) → case Int y of
                                                  I# (yy :: Int#) → I# (+# xx yy)
```

GHC implements typeclasses by passing around dictionaries of functions, e.g., for *Num*, it passes around a dictionary of the seven functions defined in the typeclass:

```
data Num a = D (a → a → a)   -- (+)
               (a → a → a)   -- (-)
               (a → a → a)   -- (*)
               (a → a)       -- negate
               (a → a)       -- abs
               (a → a)       -- signum
               (Integer → a) -- fromInteger
```

For *Num Int*, the dictionary is an instance of a *D* object named $fNumInt with the $fNumInt_$c+ function:

```
$fNumInt :: Num Int = D @ Int $fNumInt_$c+ $fNumInt_$c- $fNumInt_$c*
                              $fNumInt_$cnegate $fNumInt_$cabs
                              $fNumInt_$csignum $fNumInt_$cfromInteger
```

The + function that implements the + operator defined for the *Num* typeclass takes a dictionary as an argument and returns the function defined for the + operator, which is $fNumInt_$c+ for *Int* arguments.

```
+ :: forall a . Num a → a → a → a =
  λ @a (x :: Num a) → case (a → a → a) x of
                        D:Num (plus :: a → a → a)
                              (minus::a → a → a)
                              (times :: a → a → a)
                              (negate :: a → a)
                              (abs :: a → a)
                              (signum::a → a)
                              (fromInteger :: Integer → a) → plus
```

Finally, the compiled form of *add2* is a call to the + function, which is passed the *Int* type (the *@Int* argument), the *Num Int* dictionary, the *x* argument, and a *I#* constructor passed the literal 2, an *Int#*.

```
add2 :: Int → Int = λ (x :: Int) → + @Int $fNumInt x (I# (2:: Int#))
```

As a result, just to compile "x + 2," the compiler needs to support algebraic data types (to handle the *Num Int* dictionary and the *Int* type itself), first-class functions (for the $fNumInt_$c+ function in the dictionary), and pattern-matching.

To compile this example, which should be a sort of "hello world" program for our compiler, we need to inline the + function with its dictionary (which GHC usually does) and compile away the degenerate wrapping of *Int* types with the *I#* data constructor.

In retrospect, these aspects of Haskell and GHC gave us many headaches. We spent a fair amount of time reworking the standard prelude so that it would compile in our setting. In particular, we do not support Haskell's exception mechanism, which many standard prelude functions rely on (e.g., division can throw a division-by-zero exception). Such challenges meant we never realized our dream of compiling Haskell code written by others, in no small part because we did not attempt to implement its I/O library.

4.3 Streams

One notable dead end, documented in our 2015 CODES+ISSS paper [35], was "streams," our dialect of Core that had a near-trivial translation into hardware.

In this dialect, we represented the sequence of values on wires in a synchronous digital circuit as infinite lists. We chose such a dialect of the lambda calculus as an IR because it has a nearly one-to-one translation into circuitry: the "delay" constructor for streams becomes a flip-flop whose reset value is its first argument; constant-latency memories are modeled as primitive stream functions (read/write commands in; read data out); and everything else becomes combinational logic. We expand on this in a 2015 technical report [33]. The

drawback of the streams approach was that it forced everything to be scheduled into clock cycles before hardware could be generated. As such, accommodating a block with variable latency, such as a more complicated memory system or a called function, would require inserting explicit handshaking logic in the IR, which we felt would make it overly detailed. We ultimately abandoned this direction in preference to a dataflow representation with implicit handshaking.

4.4 Algebraic Data Types in SystemVerilog

SystemVerilog is a far richer language than its predecessor Verilog or main competitor VHDL. In particular, the 2012 version of the standard includes tagged unions [20, §7.3.2], which are effectively the algebraic data types of Haskell.

Sadly, the SystemVerilog tools we have been using, specifically Verilator and Altera/Intel's Quartus, did not support tagged union types when we started this project, and still do not appear to. As a result, much of our SystemVerilog code generator is devoted to translating algebraic data types into (synthesizable) bit vectors and the resulting generated code is often difficult to read. Clearly, someone on the SystemVerilog standardization committee saw the utility of algebraic datatypes.

5 Conclusions and Future Work

It took a long time, but we did manage to create a working compiler able to generate hardware for complex, irregular algorithms expressed as functional programs. The richest example program to date is K-means clustering [31, §6.3, 7.2.3], which is used in machine learning and image processing.

There remain many small optimizations to be made to the generated dataflow networks. One obvious one, which Arvind and Nikhil [2] employed decades ago, is to merge literals into arithmetic operators. At the moment, our networks have rather complicated "go" networks that effectively track control flow and are used to generate literals as needed; Arvind and Nikhil preferred to merge literal constants into arithmetic operators, so for an expression like "a + 2," they would generate a custom "+2" node and feed the variable *a* into it.

Buffer insertion is another serious optimization issue that remains unaddressed. Choosing where to insert data and control buffers is equivalent to the scheduling problem in classical high-level synthesis and likely has a similar solution that looks at expected combinational latency and balances it with desired clock frequency to divide operations across multiple clock cycles. An algorithm for this would likely start from our current minimal buffer insertion policy (which inserts buffers in likely sources of loops, such as function call boundaries) and then decide where buffers are to be added to meet timing requirements. A challenge in such a setting is to properly take the timing of handshaking logic into account, which we suspect can dominate in networks that perform little arithmetic.

We have only just begun to look at one of the big problems we set out to address—synthesizing algorithm-specific memory systems—but we now have the infrastructure to begin making inroads on this problem. Townsend presents some preliminary results in his

thesis [31, §6.2] in which he looks at how to partition a fixed number of memory bits (e.g., those available on a particular FPGA) into multiple memory region, often type-specific, to improve parallelism and performance. One issue that came up in this work is how best to move data between on-chip and off-chip memories. One challenge is dealing with different word sizes: type-specific on-chip memory can use arbitrary word widths (much to its advantage), but an off-chip memory port is invariably a single, standard word size (e.g., 64 bits). This raises the usual challenge of packing and/or alignment. While standard solutions exist, which are best in our setting remains unclear.

Garbage collection remains the elephant in the room. We are currently developing a garbage collected memory system, but doing so presents many challenges. One is gathering the roots of the heap. In most settings, this is fairy easy: they reside in processor registers and on the stack. In our setting, the stack is part of the memory system (but may not be part of a garbage-collected heap), but the equivalent of processor registers are scattered across our dataflow networks. In ongoing work, we are developing a scheme for "peeking" into every buffer that could hold a pointer and collect and use those for the roots. Another challenge is coping with the heterogeneous collection of type-specific mini-heaps that we wish to synthesize. A primary goal of this work has always been to use distributed memory to enable parallelism, but this makes it all the more difficult to perform garbage collection since cross-heap pointers must be considered.

We suspect adding the ability to have multiple parallel invocations of pipelined tail-recursive functions will greatly improve the performance of certain algorithms. At the moment, any tail-recursive function (that is to say, most interesting functions) is implemented in such a way that only one call of it can be processed at any time. Put another way, only one set of function argument tokens are allowed to enter a tail-recursive function call until a result is finally generated. The reason for this is the difficulty of accommodating multiple calls simultaneously: the number of iterations required to produce a result from a tail-recursive function often depends on the arguments passed to it; multiple invocations may therefore terminate out-of-order. The solution for doing this will involve things like tagging tokens with a "color" that indicates which function invocation they belong to and reorder buffers for rectifying the out-of-order termination problem, but this may lead to very costly hardware that may not perform better enough to justify the increased area.

We suspect the ability to have multiple in-flight invocations of a function could also address the resource sharing problem. In particular, we would like to show that a merged sub-graph in a Kahn network does not introduce any more deadlocks than would be present in an equivalent non-merged sub-graph. That is, sharing produces the same result as inlining. We suspect this will require tokens of different colors to pass each other (our networks otherwise process tokens in strict order), which will be quite a jump over classical Kahn networks.

References

[1] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proceedings of Principles of Programming Languages (POPL)*, pages 293–302, Austin, Texas, January 1989.

[2] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[3] David F. Bacon, Perry Cheng, and Sunil Shukla. And then there were none: A stall-free real-time garbage collector for reconfigurable hardware. In *Proceedings of Program Language Design and Implementation (PLDI)*, pages 23–24, Beijing, China, June 2012.

[4] David F. Bacon, Perry Cheng, and Sunil Shukla. Parallel real-time garbage collection of multiple heaps in reconfigurable hardware. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 117–127, Edinburgh, United Kingdom, 2014. ACM.

[5] Bingyi Cao, Kenneth A. Ross, Martha A. Kim, and Stephen A. Edwards. Implementing latency-insensitive dataflow blocks. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 179–187, Austin, Texas, September 2015. The Institute of Electrical and Electronics Engineers (IEEE).

[6] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 657–662, San Francisco, California, July 2006.

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

[8] Jack B. Dennis. A parallel program execution model supporting modular software construction. In *Proceedings of the Third Working Conference on Massively Parallel Programming Models (MPPM)*, pages 50–60, London, UK, November 1997.

[9] Jack B. Dennis. General parallel computation can be performed with a cycle-free heap. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 96–103, Paris, France, October 1998.

[10] Neil Deshpande and Stephen A. Edwards. Statically unrolling recursion to improve opportunities for parallelism. Technical Report CUCS–011–12, Columbia University, Department of Computer Science, New York, New York, USA, July 2012.

[11] Stephen A. Edwards. Functional Fibonacci to a fast FPGA. Technical Report CUCS–010–12, Columbia University, Department of Computer Science, New York, New York, USA, June 2012.

[12] Stephen A. Edwards. A finer functional Fibonacci on a fast FPGA. Technical Report CUCS–005–13, Columbia University, Department of Computer Science, New York, New York, USA, February 2013.

[13] Stephen A. Edwards. Functioning hardware from functional programs. Technical Report CUCS–027–13, Columbia University, Department of Computer Science, New York, New York, USA, October 2013.

[14] Stephen A. Edwards, Richard Townsend, Martha Barker, and Martha A. Kim. Compositional dataflow circuits. *ACM Transactions on Embedded Computing Systems*, 18(1):5, February 2019.

[15] Stephen A. Edwards, Richard Townsend, and Martha A. Kim. Compositional dataflow circuits. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 175–184, Vienna, Austria, September 2017. Association for Computing Machinery.

[16] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. MIT Press, third edition, 2008.

[17] Dan R. Ghica and Alex Smith. Geometry of synthesis III: Resource management through type inference. In *Proceedings of Principles of Programming Languages (POPL)*, pages 345–356, Austin, Texas, January 2011.

[18] Sumit Gupta, Rajesh K. Gupta, Nikil D. Dutt, and Alex Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of the 16th International Conference on VLSI Design*, pages 461–466, New Delhi, India, January 2003.

[19] Sumit Gupta, Rajesh K. Gupta, Nikil D. Dutt, and Alex Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer, 2005.

[20] IEEE Computer Society, 345 East 47th Street, New York, New York. *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language (1800–2012)*, February 2013.

[21] Thomas 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.

[22] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[23] Simon Peyton Jones and David Lester. *Implementing Functional Languages: A tutorial*. Prentice Hall, 1992.

[24] Cheng-Hong Li, Rebecca Collins, Sampada Sonalkar, and Luca P. Carloni. Design, implementation, and validation of a new class of interface circuits for latency-insensitive design. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 13–22, Nice, France, May 2007. The Institute of Electrical and Electronics Engineers (IEEE).

[25] Alan Mycroft and Richard W. Sharp. The FLaSH project: Resource-aware synthesis of declarative specifications. In *Proceedings of the International Workshop on Logic Synthesis (IWLS)*, Dana Point, California, May 2000.

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

[27] Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12:393–434, September 2002.

[28] Richard W. Sharp and Alan Mycroft. The FLaSH compiler: Efficient circuits from functional specifications. Technical Report tr.2000.3, AT&T Laboratories Cambridge, 2000.

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

[30] Andrew Tolmach, Tim Chevalier, and The GHC Team. An external representation for the GHC core language (for GHC 6.10), April 2010.

[31] Richard Townsend. *Compiling Irregular Software to Specialized Hardware*. PhD thesis, Columbia University, Department of Computer Science, New York, New York, June 2019. Also technical report CUCS–002–19.

[32] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. Resource allocation for hardware implementations of map. In *Proceedings of the Workshop on Architectures and Systems for Big Data (ASBD)*, Minneapolis, Minnesota, June 2014.

[33] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. Hardware in Haskell: Implementing memories in a stream-based world. Technical Report CUCS-017-15, Columbia University, Department of Computer Science, September 2015.

[34] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings of Compiler Construction (CC)*, pages 76–86, Austin, Texas, February 2017. ACM.

[35] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. Hardware synthesis from a recursive functional language. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 83–93, Amsterdam, The Netherlands, October 2015. IEEE.