Functioning Hardware from Functional Programs

Stephen A. Edwards
Columbia University, Department of Computer Science
Technical Report CUCS–027–13

October 8, 2013

Abstract

To provide high performance at practical power levels, tomorrow's chips will have to consist primarily of application-specific logic that is only powered on when needed. This paper discusses synthesizing such logic from the functional language Haskell. The proposed approach, which consists of rewriting steps that ultimately dismantle the source program into a simple dialect that enables a syntax-directed translation to hardware, enables aggressive parallelization and the synthesis of application-specific distributed memory systems. Transformations include scheduling arithmetic operations onto specific data paths, replacing recursion with iteration, and improving data locality by inlining recursive types. A compiler based on these principles is under development.

Most of us can remember a world in which transistor speed limited chip performance and waiting a year would bring bigger, faster chips with comparable prices and power consumption, thanks to Moore's Law [15] and Dennard scaling [8]. This world has gone, causing significant changes in how we must design and use the chips of the future.

Power dissipation now limits chip performance. While the future promises chips with more, faster transistors, running all these transistors at maximum speed would produce more heat than practical cooling methods could dissipate. Intel's chips with Turbo Boost [4] are a harbinger of this: they are normally underclocked to respect thermal limits, but a core can run briefly at full speed.

My group at Columbia, and other adherents of the "Dark Silicon" movement [22, 9], believe that to achieve decent performance at reasonable power levels, future chips will have to consist mostly of application-specific logic that is only activated when needed.

To improve performance per watt, designers today employ multicores: arrays of tens or hundreds of modestly sized processor cores on a single die. Pollack's law [19] justifies them: doubling uniprocessor performance generally requires four times the transistors; the performance of a multicore should almost double with only twice the transistors.

But multicores are at best a temporary fix. Ignoring for the moment the myriad difficulties of programming them, implementing shared memory for hundreds of cores, and Amdahl's law [1], the benefits of multicores will diminish over time because fewer and fewer of them will be able to be powered on at any time. Again, Intel's Turbo Boost portends this trend.

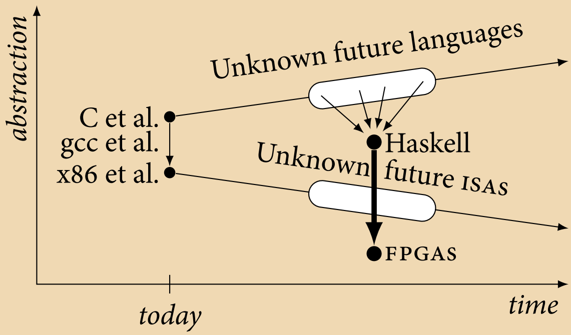
Future chips will have to consist mostly of heterogeneous application-specific logic synthesized from high-level specifications (i.e., not simply copies of a single block or standard blocks written by others) if they are to achieve decent performance at reasonable power levels. Specialization will be mandatory: if at any one time we can only ever use a small fraction of a chip's transistors, they had better be doing as much useful work as possible. Future designs should minimize the number of logical transitions required to complete a task, not just maximize the number of operations per second.

In this paper, I describe the beginnings of a tool to synthesize efficient hardware from algorithms expressed in the functional language Haskell. This should enable designers to quickly design the application-specific logic needed for tomorrow's vast, dark chips.

Swanson and Taylor's Conservation Cores [22] arose from similar goals. Their tools identify and extract "energy-intensive code regions," synthesize each into a specialized offload engine, then patch the original code to invoke the engines at the appropriate time.

They connect each of their cores directly to the host processor's data cache, a key architectural choice with many implications. Simplicity is the main advantage: each core uses the same address calculations as the original code and no explicit data transfer between processor and engine is necessary. However, it also inhibits any performance gains from improving the data memory hierarchy. For example, their optimized code reads and writes the same sequence of data as the processor would. Thus, they effectively specialize only the datapath and instruction fetch and decode logic, not the data memory system.

In contrast to the Conservation Cores project, a central goal of our project is to exploit opportunities for specializing data memory systems. The large fraction of power consumed by today's memory systems is one motivation: Dally et al. [6] found in one RISC processor, communication of data and instructions consumed 70% of the total chip power. Clock and control accounted for 24%, and arithmetic, the "real" computation, was only 6%. The standard flat memory model also impedes programs from taking full advantage of data locality. Caches attempt to harness whatever locality happens to be in the program, but it is difficult to write a truly cache-aware program. Finally, parallel emulation of the flat model has led to absurdities such as weak memory consistency.



Trend 1: Programmers demand ever more abstract languages to improve their productivity

Trend 2: ISAs must expose more hardware realities to enable better optimized software

To make progress, future programs will have to be exposed to the physical reality of complex, distributed memory systems and higher-level languages will have to conceal even more implementation details to deliver reasonable programmer productivity. It falls to compilers to span this growing gap, as depicted in the figure above. But the exact structure of the gap is not yet clear. Language design moves slowly and unpredictably since programmers' reactions to any new advance can be as important as any underlying mathematics, so prognosticating on potential programming paradigms is perilous. At the same time, what physical realities to expose is also unclear and the subject of ongoing research [17].

1 Functional Programs to Hardware

The dark arrow above represents our project: a compiler that synthesizes digital logic for field-programmable gate arrays—FPGAs—from the Haskell functional language. This is intended as a step towards building tomorrow's heterogeneous, specialized chips. While neither today's functional languages nor today's programmable hardware are ideal solutions, both have properties that make them excellent research platforms for exploring the way forward.

Starting from a pure (side-effect-free) functional language solves many key problems. It is inherently deterministic, freeing us from data-dependent races and deadlocks. The immutable memory model sidesteps the shared memory consistency problem, opening opportunities to improve data locality through duplication. It is inherently parallel since the underlying model, the lambda calculus, admits a wide variety of evaluation policies, including parallel ones. Also compelling are the sophisticated compiler optimizations that have been developed for functional languages. While the simple mathematical formalism helps, the absence of side-effects, pointer aliasing, and related problems is the real enabler.

Modern FPGAS have heterogeneous, distributed, configurable on-chip memory and do not constrain us to legacy architectures. While the on-chip memory of existing FPGAS is modest, its structure is representative of the heterogeneous, distributed memories expected on future parallel processors. Furthermore, FPGAS' extreme flexibility avoids the biases present in today's parallel computers. In particular, FPGAS do not mandate a specific instruction set and instead permit arbitrary control strategies, ranging from a classical stored-program computer to completely hardwired.

This work targets irregular algorithms, not the well-studied high-performance scientific or graphics workloads. Instead, the goal is algorithms similar to those of the Galois project: "data-parallel irregular applications" that "manipulate large pointer-based data structures like graphs" [14]. Galois even appears to be adopting aspects of the functional style: their "optimistic iterators" are equivalent to the *map* and *fold* functions present in every functional language.

Our choice of Haskell was due in part to the availability of the open-source Glasgow Haskell Compiler (GHC), a sophisticated yet reasonably well-documented piece of software. We were also attracted by its purity (unlike, say, programs in OCaml or Standard ML, Haskell programs have no side-effects) and its type system, both of which have attracted others [16]. We are, however, treating Haskell as being applicative (strict) instead of adopting its lazy semantics, which seem like they would demand excessive bookkeeping overhead in hardware.

We compile Haskell into hardware by expressing the program in a functional intermediate representation then transforming it into successively more restricted forms until a syntax-directed translation suffices to generate reasonable hardware. Reynolds took such an approach in his classic 1972 paper [20], in which he shows how to express a LISP interpreter in LISP. At first this seems unhelpfully circular, but Reynolds proceeds to transform the interpreter into a simple dialect of LISP that is easy to implement. We take a similar approach to generating hardware by dismantling features such as lambda expressions and recursion into simpler forms that are semantically equivalent.

We employ GHC's core intermediate representation [13], which consists of the lambda calculus augmented with literals, primitive operations, data constructors, and case expressions. It is powerful enough to represent all of Haskell after some desugaring by the GHC front end, yet consists of only six constructs.

In the remainder of this paper, we demonstrate these ideas through a series of examples. The first shows how Haskell's algebraic datatypes can be implemented in hardware, a cornerstone of later techniques. Next, we demonstrate how an arithmetic-intensive algorithm can be transformed into efficient hardware, how to transform recursion into easy-to-implement iteration by adding an explicit stack, and how inlining code and recursive types can improve parallelism and locality.

2 Implementing Algebraic Datatypes in Hardware

Interesting algorithms demand interesting data types. While it is possible to do everything with just bytes, programmers have long relied on aggregate types and pointers to make sense of complexity. Providing only integers and arrays, as is often done in traditional hardware description languages, is not enough.

Synthesizing hardware specialized for abstract data types is a central goal of this project. The additional flexibility this provides the compiler should enable more optimizations than if, say, the type system only provided bit vectors and arrays, as is typical of most hardware description languages.

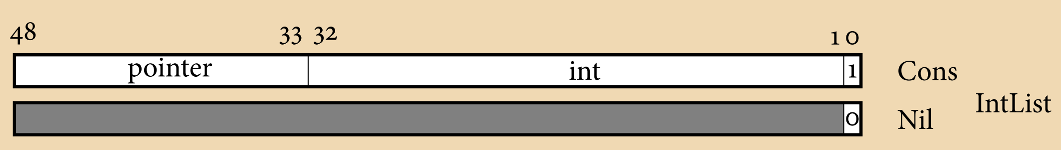
Modern functional languages provide recursive algebraic data types, a unification of aggregate and enumerated types. First introduced in 1980 [3], they provide an elegant, abstract means of structuring virtually every kind of data, and are well-suited to expressing data in custom hardware. In particular, they provide a high-level memory abstraction that gives a compiler far more flexibility in choosing (and optimizing) their implementation.

An algebraic data type consists of a set of variants. Each variant has a name (data constructor)—analogous to a name in an enumerated type—and zero or more data types associated with that variant. A basic example is a singly-linked list of integers: a list is either the empty list ("Nil") or a cell ("Cons") consisting of an integer and the rest of the list. In Haskell, such a type can be expressed as follows.

```
data  IntList  = Cons Int  IntList
               | Nil
```

Under this definition, "Nil" is the empty list, "Cons 42 Nil" is the list consisting of just "42," "Cons 42 (Cons 17 Nil)" is the list consisting of "42" followed by "17," and so forth.

Custom hardware is not subject to the tyranny of the byte, so we have a lot of flexibility in how we can encode such objects. We can even consider using a variety of representations: one for manipulation in a datapath, another for memory.



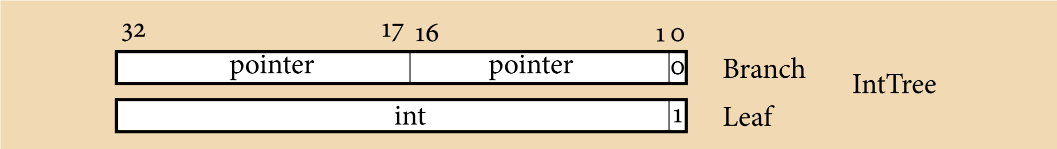
The figure above shows one way to encode a single IntList object as a 49-bit vector. The least significant bit indicates the variant, i.e., whether the object is a Nil or a Cons cell (unlike a C union, it is always necessary for an algebraic data type to encode which variant is being represented). A zero indicates the object is a Nil and the rest of the bits are don't-cares. A one indicates the object is a Cons cell, the next thirty-two bits represent the integer, and the final sixteen represent a pointer to the next element of the list.

Choosing the size and interpretation of the pointer raises interesting questions. A processor-like solution is to represent all pointers uniformly, following the fiction of memory as a uniform array of bytes, but hardware has no such constraint. Instead, segregating IntList objects into a special, dedicated memory may be wiser. This enables a type-specific caching strategy (I assume any type-specific local memory would be backed by much larger off-chip memory) and simpler memory management since all objects are of the same type. Finally, the number of bits used to represent a pointer to an IntList can be minimized based on an estimated bound on the maximum number of active IntList objects at runtime.

We have flexibility in storing "Nil" objects. Since they are all identical, it is unnecessary to store more than one in memory, so the IntList memory manager could always direct them to the same address. It could go even farther and simply never store Nil objects in memory, instead reserving a specific pointer value (e.g., zero) to represent the Nil object.

Another algebraic data type poster child is the binary tree:

```
data IntTree = Branch IntTree IntTree | Leaf Int
```



The meaning of these bits vary depending on the variant being represented. The first bit distinguishes leaves from branches; the remaining thirty-two either represent a single integer or a pair of pointers.

As with the IntList type above, there is a choice of how many bits to use for the pointers, which can depend on the maximum number of such objects at runtime. Again, it would be natural to dedicate a memory system to such objects, using a type-specific memory management scheme.

Distinguishing variants (e.g., Cons vs. Nil) is straightforward when there are only two, but there is more flexibility as the number grows. A binary encoding is the most compact, but other encodings, such as one-hot, can lead to simpler datapath logic since it is effectively already decoded. Especially here, it may be better to represent in-memory data differently (e.g., in binary) and add translation logic in the memory controller.

While there is little flexibility in the layout of these examples, more complex objects admit more alternatives and thus have optimization potential. In particular, how bits are shared among a type's variants can affect the size and complexity of the datapath and thus could be optimized.

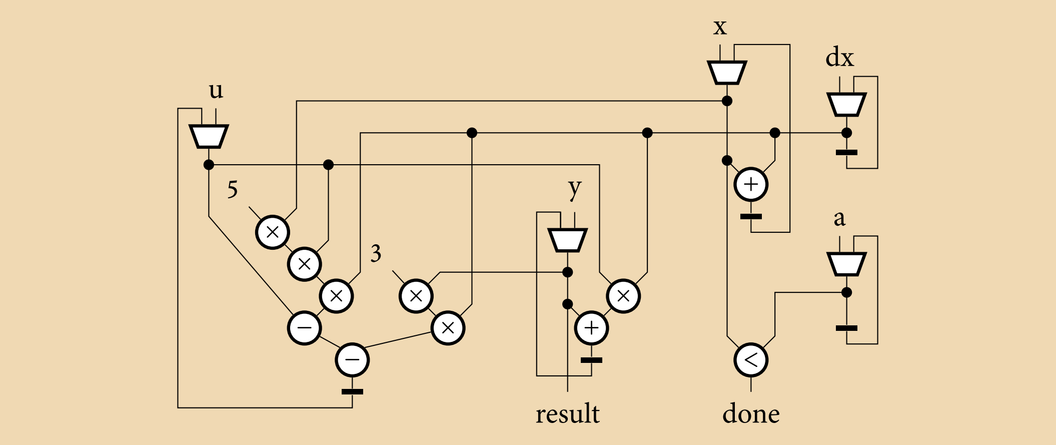
3 Arithmetic and Hardware Datapaths

The ease and elegance of manipulating programs in the functional style motivates using it to express algorithms destined for hardware. Since the functional style is based on Church's lambda calculus [5], transforming such programs looks like elementary algebra. The referential transparency of the lambda calculus enables this: an expression means exactly its value and nothing more, rendering moot the problems of unexpected aliases and side-effects that bedevil imperative language compilers.

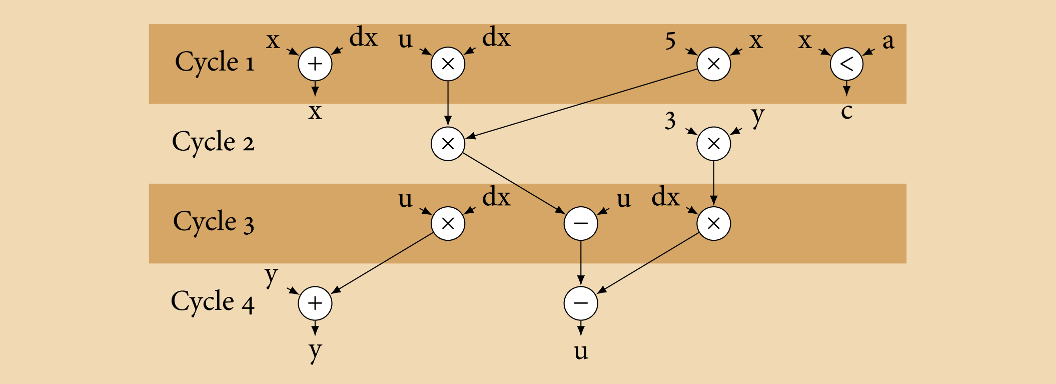
As an illustration, consider a well-worn example from the high-level synthesis literature [18]: a numerical solver for the differential equation $y'' + 5xy' + 3y = 0$. Introducing $u = y'$ gives $u' = -5xu - 3y$ and $y' = u$; applying Euler's method gives the algorithm below, coded in Haskell as a tail-recursive function of five arguments. This integrates from x to a, stepping by dx, and starting from initial values y and $u = y'$. Running this algorithm with integers actually gives nonsensical results, but I will do so anyway for consistency with the literature.

```
diffeq  a  dx  x  u  y  =
  if  x  <  a  then  diffeq  a  dx  (x  +  dx)  (u  −  5*x*u*dx  −  3*y*dx)  (y  +  u*dx)  else  y
```

This is easy to translate into correct hardware. Treating the tail-recursive call as a clock cycle boundary and asserting the *call* signal in a cycle where we apply new arguments to the inputs *a*, *dx*, *x*, *u*, and *y*, the circuit below produces the result when *done* is true.



This single-cycle implementation is correct but uses a lot of multipliers. Mapping a computation to a resource-constrained datapath is a classical problem in high-level synthesis; many effective techniques for it exist. For example, Paulin et al. consider a two-multiplier datapath with an adder, subtractor, and comparator [18]. Their schedule below illustrates how to compute the function using this datapath.



While devising schedules that minimize, say, the number of required temporary registers is a nontrivial optimization problem [7], restructuring a functional program according to a schedule is easy to do in a functional setting. That it can be expressed as a series of semantics-preserving transformations is a strong argument for adopting the functional model.

To schedule this function onto a datapath, we need to refactor the function into a sequence of operations. First, consider decomposing the expressions into a series of simple arithmetic operations, sequencing them with the familiar *let* construct:

```
diffeq  a  dx x u y =
  if  x  < a  then
   let  pa = u *  dx
        pb = 5  *  x
        x1  = x + dx in
   let  pa = pa *  pb
        pb = 3  *  y  in
   ...
```

But this is not enough: our goal is to enable a syntax-directed hardware translation yet we would need to infer the operators in the datapath, the states in the controller, etc. Instead, we will start with a function that models a datapath with two multipliers, an adder, a subtractor, and a comparator:

```
dpath  m1 m2 m3 m4 a1 a2  s1  s2  c1  c2  k  =
   k  (m1 *  m2) (m3 * m4) (a1  + a2)  (s1  − s2)  (c1  < c2)
```

Instead of returning its results, this function passes them as arguments to a “continuation” function k. This is continuation-passing style [10], and it follows from the *let*-form shown above: a *let* binding is equivalent to applying a lambda expression, that is, “let $x = e$ in f” is equivalent to “$(\lambda x.f)e$”. We can move e back to the left of f by passing $(\lambda x.f)$ as an argument: $(\lambda k.k\ e)(\lambda x.f)$. This is now in continuation-passing style (CPS), a form that has been used to great success in a number of compilers, such SML/NJ by Appel [2] and others.

We then decompose the arithmetic expressions into a a series of nested lambda expressions that use the datapath function to perform the arithmetic.

```
diffeq  a dx x u y =
  dpath u  dx 5  x  x  dx 0  0  x  a  (λpa pb x  _ c → if not c then y else
  dpath pa pb 3  y  0  0  0  0  0  0  (λpa pb _  _ _ →
  dpath u  dx dx pb 0  0  u  pa 0  0  (λpa pb _  d _ →
  dpath 0  0  0  0  y  pa d  pb 0  0  (λ_  _  s  d _ → diffeq  a dx x d s ))))
```

The *diffeq* function begins by calling *dpath*, which passes its result to the first continuation (the first λ term). The call to *dpath* calculates $u * dx$, $5 * x$, $x + dx$, and $x < a$, which are then bound to ma, mb, x, and c. The subtractor is not needed in this step, so it is passed 0's and its result is not bound (indicated by the underline). The next three lines behave similarly.

While this is a convenient model of the dataflow and sequencing in the hardware we want, we wish to eliminate the lambda terms, since their implementation in hardware is not obvious. We proceed by "lambda lifting" [12]: transforming all the in-scope variables into formal arguments and then name each of the lambda terms as separate functions.

To illustrate lambda lifting, consider evaluating $a * b + c$ with a *mul* function in CPS form:

```
mul x y k = k (x * y)
f a b c = mul a b (λt → t + c)
```

We want to "lift" the lambda term (λt → t + c) into a global function k0. To do this, we must add c as an argument to k0 since it would otherwise occur free in the body of k0.

```
mul x y k = k (x + y)
f a b c = mul a b (k0 c)
k0 c t = t + c
```

Note that the continuation being passed to *mul* is a partially evaluated function, not the result of calling k0. Although k0 has two arguments, we are only passing c. When *mul* calls k, it appends the final argument t and thus calls k0 with all its arguments.

Liberally applying this technique to the *diffeq* example gives the code below, which consists of four k functions. Note that each has nine or ten arguments, yet only four or five are passed in the continuation. The *dpath* function appends the remaining five arguments.

```
k0 a dx x       _  _  s d _ = dpath d  dx 5  x  x  dx 0  0  x  a  (k0 a dx     d s)
k1 a dx   u y  pa pb  s _ c = if not c then y else
                              dpath pa pb 3  y  0  0  0  0  0  0  (k2 a dx s u y)
k2 a dx x u y  pa pb  _ _ _ = dpath u  dx dx pb 0  0  u  pa 0  0  (k3 a dx x   y)
k3 a dx x   y  pa pb  _ d _ = dpath 0  0  0  0  y  pa d  pb 0  0  (k0 a dx x     )

 diffeq  a dx x u y = k0 a dx x 0 0 y u False
```

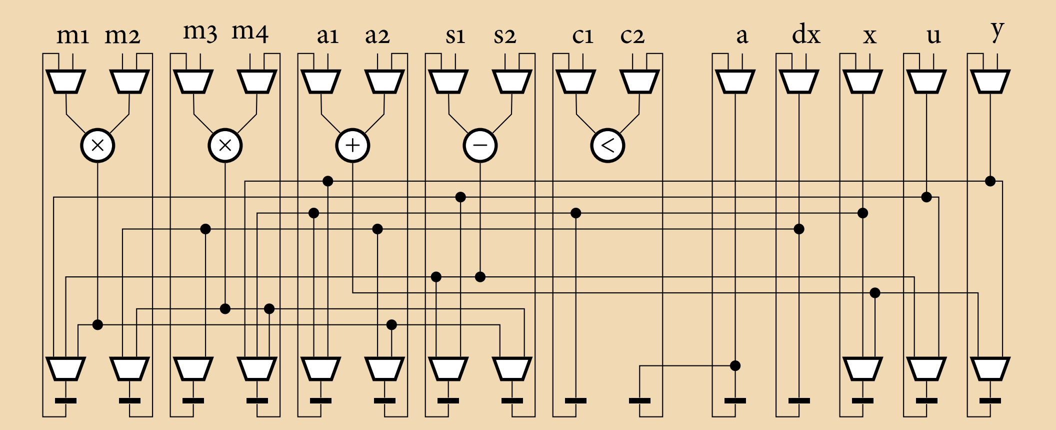
At this point, we are still passing around partially evaluated functions as arguments, which is not obvious to do in hardware. To eliminate these, we introduce a type that models the continuations passed to the *dpath* function and merge the *k* functions into a single function *kk* that performs the function of the continuation passed to it as an argument.

```
data Cont = K0 Int Int Int
          | K1 Int Int     Int Int
          | K2 Int Int Int Int Int
          | K3 Int Int Int     Int

dpath m1 m2 m3 m4 a1 a2 s1 s2 c1 c2 k =
   kk k (m1 * m2) (m3 * m4) (a1 + a2) (s1 − s2) (c1 < c2)

kk k m1 m2 a s c =
 case (k, m1, m2, a, s, c) of
 (K0 a dx x     ,_ ,_ ,s,d,_) → dpath d  dx 5  x  x dx 0 0  x a  (K1 a dx    d s)
 (K1 a dx    u y,pa,pb,s,_,c) → if not c then y else
                                dpath pa pb 3  y  0 0  0 0  0 0 (K2 a dx s u y)
 (K2 a dx x u y,pa,pb,_,_,_)  → dpath u  dx dx pb 0 0  u pa 0 0 (K3 a dx x    y)
 (K3 a dx x    y,pa,pb,_,d,_) → dpath 0  0  0  0  y pa d pb 0 0 (K0 a dx x     )

 diffeq a dx x u y = kk (K0 a dx x) 0 0 y u False
```

Finally, we inline the call to *kk* in *dpath*, resulting in a single function that performs the arithmetic and checks the state (the continuation) before calling itself again.

```
dpath m1 m2 m3 m4 a1 a2 s1 s2 c1 c2 k =
 case (k, m1*m2, m3*m4, a1+a2, s1−s2, c1<c2) of
 (K0 a dx x     ,_ ,_ ,s,d,_) → dpath d  dx 5  x  x dx 0 0  x a  (K1 a dx    d s)
 (K1 a dx    u y,pa,pb,s,_,c) → if not c then y else
                                dpath pa pb 3  y  0 0  0 0  0 0 (K2 a dx s u y)
 (K2 a dx x u y,pa,pb,_,_,_)  → dpath u  dx dx pb 0 0  u pa 0 0 (K3 a dx x    y)
 (K3 a dx x    y,pa,pb,_,d,_) → dpath 0  0  0  0  y pa d pb 0 0 (K0 a dx x     )
 diffeq a dx x u y            = dpath u dx 5 x x dx 0 0 x a (K1 a dx u y)
```

This is now easy to translate into hardware. The K's encode control states; the tail calls represent clock cycle boundaries; the *case* construct checks the state and generates arguments to feed to the next call of the function. The wiring, which may appear complex, follows directly from the definitions and uses of each identifier.



4 Recursion and Memory

Recursion is fundamental to functional programs; this section illustrates how it can be compiled into hardware. Tail recursion, such as that in the previous example, is easy to implement with flip-flops and feedback, but true recursion demands a stack. Fortunately, a series of rewriting steps like those in the last section also suffice to transform recursion into something that is easily implemented in hardware. This differs from the approach of Ghica et al. [11], which replaces each register in a recursive function with a stack.

To illustrate, consider the usual naïve algorithm for computing Fibonacci numbers.

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

First, consider how the expression " fib (n−1) + fib (n−2)" would be evaluated in an imperative setting. Typically, fib (n−1) would be called first, the result stored in a temporary, fib (n−2) would be called, and its result added to the temporary and returned. The imperative pseudocode below illustrates this.

```
tmp =  fib (n−1)      // Compute fib (n-1)
n1  = tmp
tmp =  fib (n−2)      // Compute fib (n-2)
n2  = tmp
tmp = n1  + n2        // Compute and return fib (n-1) + fib (n-2)
return  tmp
```

In assembly language, a procedure call is often considered to pass control to the statement following it, but the reality is more complex. It is actually an unconditional jump that passes the address of the next statement—the return address—to the called routine. The callee, when it wants to return, places its result in a mutually agreed-upon location (e.g., a register) and performs an unconditional jump back to the return address.

In continuation-passing style, “the address of the instruction following the call” is the continuation passed to the called function, which “returns” by calling this continuation. The argument of the continuation is the agreed-upon location for the result.

This view makes it straightforward to transform the recursive *fib* function into continuation-passing style. A continuation argument *k* is added to the function. The first recursive call is passed a continuation that stores the result before performing the second recursive call, which is passed a continuation that receives the result, adds it to the previous result, and “returns” it to the original continuation *k* by calling it.

```
fibk  n k = case  n  of
            1 → k 1
            2 → k 1
            n →  fibk  (n−1)  (λn1 →  -- Compute fib (n-1)
                 fibk  (n−2)  (λn2 →  -- Compute fib (n-2)
                 k  (n1  + n2)))      -- Compute and return fib (n-1) + fib (n-2)
fib  n = fibk  n  (λx → x)
```

As before, we next transform the lambda terms into top-level functions named *k*0, *k*1, and *k*2, being careful to capture live variables as arguments.

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

The next step is to merge the *k* functions and replace passed functions (the continuations) with a data type, but this time the *k* passed to *fib* is passed as part of the new continuations. This means the continuation data type needs to be recursive; it effectively describes the traditional stack used to implement recursion.

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

fibk n k = case (n,k) of
          (1,  k)            → kk k 1
          (2,  k)            → kk k 1
          (n,  k)            → fibk (n−1) (K1 n  k)
kk k a = case (k, a) of
          ((K1 n  k), n1)   → fibk (n−2) (K2 n1 k)
          ((K2 n1 k), n2)   → kk k (n1 + n2)
          (K0,        x )   → x
fib n =                       fibk n K0
```

To merge the *fibk* and *kk* functions, we introduce one more data type that distinguishes between calls to each.

```
data Cont = K0 | K1 Int Cont | K2 Int Cont
data Call = Fibk Int Cont     −− Recursive call
          | KK Cont Int       −− Call a continuation

fibk z = case z of
          (Fibk 1        k )  → fibk (KK k 1)
          (Fibk 2        k )  → fibk (KK k 1)
          (Fibk n        k )  → fibk (Fibk (n−1) (K1 n  k))
          (KK (K1 n  k)  n1)  → fibk (Fibk (n−2) (K2 n1 k))
          (KK (K2 n1 k)  n2)  → fibk (KK k (n1 + n2))
          (KK K0         x )  → x
fib n                         = fibk (Fibk n K0)
```

We need to be able to express the *Cont* recursive type in hardware. Interpreting it literally would require an unbounded number of bits, so instead we will take the standard approach of expressing each self-reference with a pointer and assume the presence of a memory system.

We introduce a new type, *CRef*—a reference to a *Cont* object, and two functions—*load* and *store*—that read and write memory. Note that the store function *returns* a pointer to the newly created object, which presumes the memory is responsible for its own allocation policy.

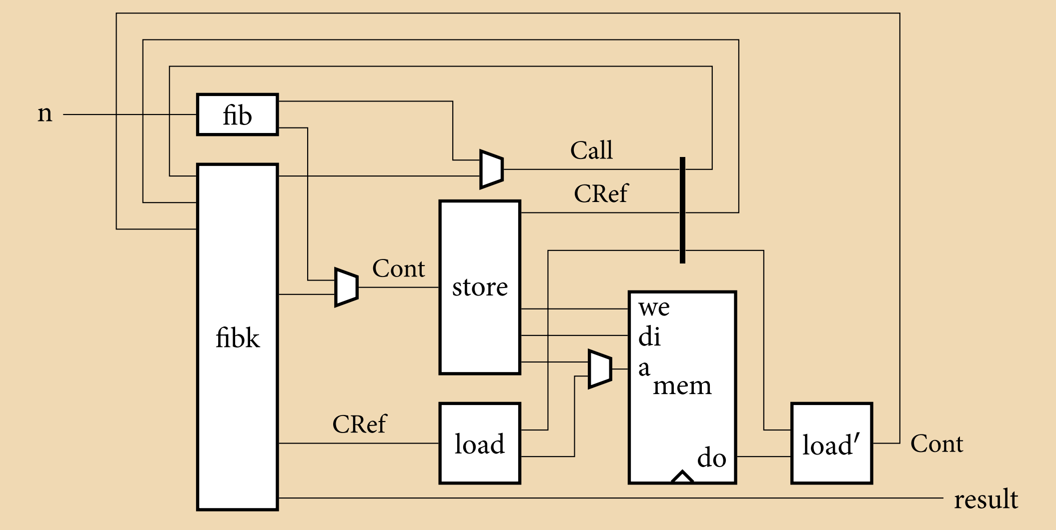
As usual, the types effectively dictate how we must write the program: where a *CRef* is required, we must construct a *Cont* and pass it to *store*; *load* gives us a *Cont* when we have a *CRef*.

```
load   :: CRef → Cont   -- Type signatures of memory operations
store  :: Cont → CRef
data Cont = K0 | K1 Int CRef | K2 Int CRef
data Call = Fibk Int CRef | KK Cont Int

fibk z = case z of
        (Fibk       1   k) → fibk (KK (load k) 1)
        (Fibk       2   k) → fibk (KK (load k) 1)
        (Fibk       n   k) → fibk (Fibk (n−1) ( store (K1 n  k )))
        (KK (K1 n   k) n1) → fibk (Fibk (n−2) ( store (K2 n1 k )))
        (KK (K2 n1 k)  n2) → fibk (KK (load k) (n1 + n2))
        (KK K0          x) → x
fib n =                      fibk (Fibk n ( store K0))
```

Below is a block diagram illustrating how this can be translated into hardware. The main *fibk* block receives a *Call* object, which either it or the *fib* block generates. The *Call* include either a *CRef* from a call to *store*, or a *Cont* from a call to *load*. The logic inside the *fibk* mostly steers data according to the type of the *Call* object passed to it; it also computes *n-1*, *n-2*, and *n1 + n2*.

Memory for the stack is assumed to take one clock cycle: an address and possibly data in is presented at the end of one cycle and read data (if any) appears in the next.

The *load* function is broken into two pieces. The first operates before the clock tick and uses the *CRef* to calculate an address for the stack memory. The second piece, *load′*, packages the object by combining the integer from the stack memory with an adjusted *CRef*; effectively performing the "pop" operation.



5 Inlining Code and Recursive Types

Since replacing arguments with their values is the central operation in the lambda calculus, it is not surprising that inlining is the key operation in any optimizing functional language compiler [13]. Many imperative compilers inline functions, sometimes to reduce the overhead of recursion [21], but in a functional setting it also subsumes loop unrolling.

Inlining is even more useful in hardware, with its abundant available parallelism. Consider the following fragment of a Huffman decoder, which walks a binary tree while stepping through a list of bits.

```
data HTree = Branch HTree HTree
           | Leaf Char
data BitList = Cons Bool BitList
             | Nil

decode t hd = case t of
  Leaf c       → (c, hd)                 -- Reached a leaf; return the character
  Branch l r → case hd of
    Cons True  tl → decode l tl          -- Next bit is 1: follow the left branch
    Cons False tl → decode r tl          -- Next bit is 0: follow the right branch
```

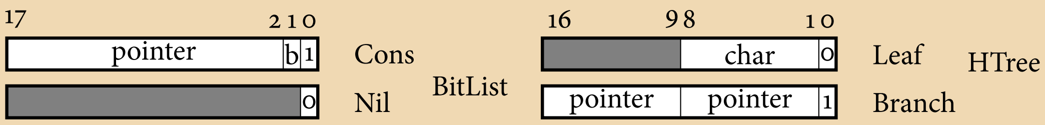
Inlining the function once lets us decode a pair of bits between recursive calls, which can be interpreted as increasing the amount of work done per clock cycle.

```
decode t hd = case t of
  Leaf c       → (c, hd)
  Branch l r → case hd of
    Cons True  tl → case l of -- decode l tl
      Leaf c         → (c, tl )
      Branch ll lr → case tl of
        Cons True ttl   → decode ll ttl
        Cons False ttl  → decode lr ttl
    Cons False tl → case r of -- decode r tl
      Leaf c         → (c, tl )
      Branch rl rr → case tl of
        Cons True  ttl  → decode rl ttl
        Cons False ttl  → decode rr ttl
```

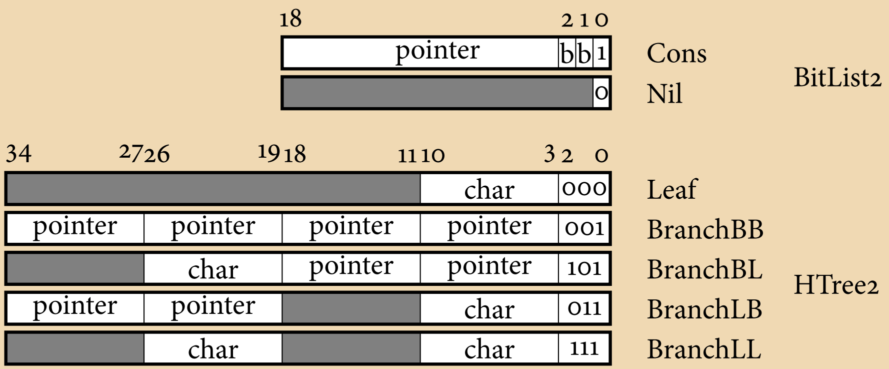
Going one step further, a compiler can inline the recursive HTree and BitList *types* to improve data locality. The algorithm now works on pairs of bits and triplets of tree nodes. The logic gets significantly more complicated, and the code below omits a number of corner cases that would have to be considered in practice, but this is exactly the sort of thing a compiler is good at worrying about.

```
data  HTree2 = Leaf  Char
              | BranchBB HTree2 HTree2  HTree2 HTree2
              | BranchBL HTree2 HTree2  Char
              | BranchLB Char           HTree2 HTree2
              | BranchLL Char           Char
data  BitList2  = Cons2 Bool Bool  BitList2
                | Cons1 Bool  BitList2
                | Nil

decode t  hd = case  t  of
  Leaf  c                    →  (c,  hd)
  BranchBB ll  lr  rl  rr  →  case  hd of
    Cons2 True True     ttl  →  decode  ll   ttl
    Cons2 True False    ttl  →  decode lr   ttl
    Cons2 False  True   ttl  →  decode rl   ttl
    Cons2 False  False  ttl  →  decode rr   ttl
  -- may other cases omitted
```

This implementation improves locality and reduces memory traffic by packing more data into a single "word." Consider the following layouts for HTree and BitList objects, which assumes 16-bit pointers for the BitList and 8-bit pointers for the HTree:



Reading a single bit from the BitList requires an entire word to be fetched from memory; each HTree object only contains information about a single bit. By contrast, BitList2 and HTree2 represent pairs of bits. Although each object is bigger (BitList2 is only one bit larger; HTree2 is twice as large), using them halves the number of memory operations required.



6 Looking Ahead

Our Haskell-to-hardware compiler is a work-in-progress. We are currently automating these transformations but do not yet have experimental results on their efficacy.

The goal of this paper was to introduce through examples our approach to translating a functional language into hardware: a series of semantics-preserving rewrite steps that ultimately produce a functional dialect that admits a reasonable syntax-directed translation into hardware.

One of the challenges this project suggests is a new class of encoding problem: data representation synthesis. Given a potentially recursive algebraic type, determine one or more ways to represent it in hardware that leads to small and/or fast hardware for manipulating it. While mechanical procedures can easily produce correct results, choosing a representation judiciously can greatly improve the quality of generated hardware. Part of this problem amounts to the classical finite-state machine state assignment problem, but it becomes more complicated when algebraic types that includes scalars and pointers are also considered.

Another challenge that arises in this approach is how much inlining/unrolling of expressions and types to apply. While some work has been done on this for software, the best choices for hardware will be different because of the abundance of parallelism, the even larger benefits of locality, and the availability of custom datapaths.

Configuring a distributed memory system and allocating data in it is a central challenge in this work, yet we have barely touched on it here. Broadly, the challenge is to partition the memory and data into appropriate-sized chunks and allocate them wisely. It is natural to place objects of different types into different memories, but this is just a starting point. We expect this aspect of the project will present many stimulating challenges.

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the AFIPS Joint Computer Conferences*, pages 483–485, Atlantic City, New Jersey, April 1967.

[2] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] R. M. Burstall, D. B. MacQueen, and D. T. Sanella. HOPE: An experimental applicative language. In *Proceedings of the conference on LISP and Functional Programming (LFP)*, pages 136–143, Stanford, California, August 1980.

[4] Jeff Casazza. First the tick, now the tock: Intel microarchitecture (Nehalem). White Paper, Intel Corporation, Santa Clara, California, 2009.

[5] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, April 1932.

[6] William J. Dally, James Balfour, David Black-Shaffer, James Chen, R. Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient embedded computing. *IEEE Computer*, 41(7):27–32, July 2008.

[7] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[8] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974.

[9] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 365–376, San Jose, California, June 2011.

[10] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions About Programs*, pages 104–109, Las Cruces, New Mexico, January 1972.

[11] Dan R. Ghica, Alex Smith, and Satnam Singh. Geometry of synthesis IV: Compiling affine recursion into static hardware. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 221–233, Tokyo, Japan, September 2011.

[12] 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.

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

[14] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of Program Language Design and Implementation (PLDI)*, pages 211–222, San Diego, California, June 2007.

[15] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 19 1965.

[16] Rishiyur S. Nikil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann, 2001.

[17] Mark Oskin, Josep Torrellas, Chita Das, John Davis, Sandhya Dwarkadas, Lieven Eeckhout, Bill Feiereisen, Daniel Jimenez, Mark Hill, Martha Kim, James Larus, Margaret

Martonosi, Onur Mutlu, Kunle Olukotun, Andrew Putnam, Tim Sherwood, James Smith, David Wood, and Craig Zilles. *Workshop on Advancing Computer Architecture Research (ACAR-II) Laying a New Foundation for IT: Computer Architecture for 2025 and Beyond*. Computing Research Association, Seattle, Washington, 2010.

[18] Pierre G. Paulin, John P. Knight, and E. F. Girczyc. HAL: A multi-paradigm approach to automatic data path synthesis. In *Proceedings of the 23rd Design Automation Conference*, pages 263–270, Las Vegas, Nevada, June 1986.

[19] Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, page 2, Haifa, Israel, November 1999. Abstract for keynote.

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

[21] Radu Rugina and Martin Rinard. Recursion unrolling for divide and conquer programs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 2017 of *Lecture Notes in Computer Science*, pages 34–48, Yorktown Heights, New York, August 2000.

[22] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, Pittsburgh, Pennsylvania, March 2010.