Haskell to Hardware and Other Dreams

Stephen A. Edwards

Richard Townsend   Martha A. Kim
Lianne Lairmore   Kuangya Zhai

Columbia University

Synchron, Bamberg, Germany, December 7, 2016
Where Is My Jetpack?

Popular Science, November 1969
Where The Heck Is My 10 GHz Processor?
Moore’s Law

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.”

Closer to every 24 months

Four Decades of Microprocessors Later...

Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/
What Happened in 2005?

Pentium 4  
2000  
1 core  
Transistors: 42 M

Core 2 Duo  
2006  
2 cores  
291 M

Xeon E5  
2012  
8 cores  
2.3 G
Heat Flux in IBM Mainframes: A Familiar Trend

Liquid Cooled Apple Power Mac G5

2004
CMOS
1.2 kW
Dally: Calculation Cheap; Communication Costly

“Chips are power limited and most power is spent moving data

Performance = Parallelism

Efficiency = Locality

Bill Dally’s 2009 DAC Keynote, The End of Denial Architecture
Dally: “Single-thread processors are in denial about these two facts”

We need different programming paradigms and different architectures on which to run them.
Related Work
Xilinx’s Vivado (Was xPilot, AutoESL)

**SSDM (System-level Synthesis Data Model)**

- Hierarchical netlist of concurrent processes and communication channels

- Each leaf process contains a sequential program which is represented by an extended LLVM IR with hardware-specific semantics
  - Port / IO interfaces, bit-vector manipulations, cycle-level notations

SystemC input; classical high-level synthesis for processes

Jason Cong et al. ISARS 2005
Custom datapaths, controllers for loop kernels; uses existing memory hierarchy

Bacon et al.’s Liquid Metal

**Fig. 2.** Block level diagram of DES and Lime code snippet

**JITting Lime (Java-like, side-effect-free, streaming) to FPGAs**

Goldstein et al.’s Phoenix

```c
int squares()
{
    int i = 0,
    sum = 0;
    for (;i<10;i++)
        sum += i*i;
    return sum;
}
```

**Figure 3:** C program and its representation comprising three hyperblocks; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicate values. (This figure omits the token edges used for memory synchronization.)

**Figure 8:** Memory access network and implementation of the value and token forwarding network. The LOAD produces a data value consumed by the oval node. The STORE node may depend on the load (i.e., we have a token edge between the LOAD and the STORE, shown as a dashed line). The token travels to the root of the tree, which is a load-store queue (LSQ).

C to asynchronous logic, monolithic memory
Algo-like imperative language to handshake circuits

Ghica, Smith, and Singh. *Geometry of Synthesis IV, ICFP 2011*
In this section we demonstrate how a circuit that performs communication over an I2C bus can be expressed using the Kiwi library. The motivation for tackling such an example arises from the fact that the typical coding style for such circuits involves hand coding state machines using nested case statements in VHDL (or equivalent features in Verilog). In particular, the sequencing of operations can be represented as follows:

```csharp
public static void SendDeviceID()
{
    int deviceID = 0x76;
    for (int i = 7; i > 0; i--)
    {
        scl = false;
        sda_out = (deviceID & 64) != 0;
        Kiwi.Pause(); // Set it i-th bit of the device ID
        scl = true; Kiwi.Pause(); // Pulse SCL
        scl = false; deviceID = deviceID << 1;
        Kiwi.Pause();
    }
}
```

C# with a concurrency library to FPGAs

Arvind, Hoe, et al.’s Bluespec

**GCD Mod Rule**
\[ \text{Gcd}(a, b) \text{ if } (a \geq b) \land (b \neq 0) \rightarrow \text{Gcd}(a - b, b) \]

**GCD Flip Rule**
\[ \text{Gcd}(a, b) \text{ if } a < b \rightarrow \text{Gcd}(b, a) \]

*Figure 1.3* Circuit for computing Gcd(a, b) from Example 1.

Guarded commands and functions to synchronous logic
Hoe and Arvind, *Term Rewriting*, VLSI 1999
Sheeran et al.’s Lava

`bfly :: ComplexArithmetic m
    => [ComplexSig] -> m [ComplexSig]
bfly [i1, i2] =
    do o1 <- csubtract (i1, i2)
       o2 <- cplus (i1, i2)
       return [o1, o2]

bflys :: ComplexArithmetic m
    => Int -> [ComplexSig] -> m [ComplexSig]
bflys n =
    riffle >>= raised n two bfly >>= unriffle

Figure 9: A butterfly

Figure 10: A butterfly stage of size 8 expressed with riffling

Functional specifications of regular structures
Bjøsсе, Claessen, Sheeran, and Singh. Lava, ICFP 1998
Kuper et al.’s CλaSH

More operational Haskell specifications of regular structures

Baaij, Kooijman, Kuper, Boeijink, and Gerards. Cλash, DSD 2010
My Crusade
### Deterministic Concurrency: A Fool’s Errand?

#### What Models of Computation Provide Deterministic Concurrency?

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchrony</td>
<td>The Columbia Esterel Compiler 2001–2006</td>
</tr>
<tr>
<td>Kahn Networks</td>
<td>SHIM Model/Language</td>
</tr>
<tr>
<td></td>
<td>2006–2010</td>
</tr>
<tr>
<td>The Lambda Calculus</td>
<td>This Project</td>
</tr>
<tr>
<td></td>
<td>2010–</td>
</tr>
</tbody>
</table>
Our Project: Functional Programs to Hardware

\( \lambda f. (\lambda x. (f (x x))) \lambda x. (f (x x)) \)
Our Project: Functional Programs to Hardware
Our Project: Functional Programs to Hardware
Our Project: Functional Programs to Hardware
Our Project: Functional Programs to Hardware
Our Project: Functional Programs to Hardware
Our Project: Functional Programs to Hardware
Why Functional?

- Referential transparency simplifies formal reasoning about programs

- Inherently concurrent and deterministic (Thank Church and Rosser)

- Immutable data makes it vastly easier to reason about memory in the presence of concurrency
To Implement Real Algorithms, We Need

- Structured, recursive data types
- Recursion to handle recursive data types
- Memories
- Memory Hierarchy
Structured, Recursive Data Types
Algebraic Data Types

In modern functional languages: ML, OCaml, Haskell, . . .

An algebraic type is a sum of product types

Basic example: List of integers

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

Constructing a list:

Cons 42 (Cons 17 (Cons 2 (Cons 1 Nil)))

Summing the elements of a list:

```haskell
sum li = case li of
    Nil -> 0
    Cons x xs -> x + sum xs
```
An Interpreter in One Slide

Abstract syntax tree data type:

```haskell
data Expr = Lit Int |
           Plus Expr Expr |
           Minus Expr Expr |
           Times Expr Expr
```

Recursive evaluation function:

```haskell
eval e = case e of
    Lit x → x
    Plus e1 e2 → eval e1 + eval e2
    Minus e1 e2 → eval e1 − eval e2
    Times e1 e2 → eval e1 × eval e2
```

eval (Plus (Lit 42) (Times (Lit 2) (Lit 50)))
gives 42 + 2 × 50 = 142
Algebraic Datatypes in Hardware: Lists

data IntList = Cons Int IntList
  | Nil

```
48  33 32  1 0
   pointer int 1
     Nil 0
```

Cons
Nil
Recursion to Handle Recursive Data Types
What Do We Do With Recursion?

Compile it into tail recursion with explicit stacks

[Zhai et al., CODES+ISSS 2015]

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

[Proceedings of the ACM Annual Conference, 1972]

Really clever idea:

Sophisticated language ideas such as recursion and higher-order functions can be implemented using simpler mechanisms (e.g., tail recursion) by rewriting.
Removing Recursion: The Fib Example

\[
\text{fib } n = \text{ case } n \text{ of } \\
1 & \rightarrow 1 \\
2 & \rightarrow 1 \\
n & \rightarrow \text{fib } (n-1) + \text{fib } (n-2)
\]
Transform to Continuation-Passing Style

\[
\text{fibk } n \ k = \ \text{case } n \ \text{of}
\]
\[
\begin{align*}
1 & \rightarrow k \ 1 \\
2 & \rightarrow k \ 1 \\
n & \rightarrow \text{fibk } (n-1) (\lambda n1 \rightarrow \\
& \quad \text{fibk } (n-2) (\lambda n2 \rightarrow \\
& \quad \quad k \ (n1 + n2))) \\
\end{align*}
\]

\[
\text{fib } n = \text{fibk } n \ (\lambda x \rightarrow x)
\]
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
Represent Continuations with a Type

```haskell
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
```
Merge Functions

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

data Call = Fibk Int Cont | KK Cont Int

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)
```
Add Explicit Memory Operations

```
read :: CRef → Cont
write :: 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 (read k) 1)
  (Fibk 2 k) → fibk (KK (read k) 1)
  (Fibk n k) → fibk (Fibk (n - 1) (write (K1 n k)))
  (KK (K1 n k) n1) → fibk (Fibk (n - 2) (write (K2 n1 k)))
  (KK (K2 n1 k) n2) → fibk (KK (read k) (n1 + n2))
  (KK K0 x ) → x

fib n = fibk (Fibk n (write K0)) 1
```
Simplified Functional to Dataflow
Sum a list using an accumulator and tail-recursion

\[
\text{sum } \text{lp } s = 
\begin{align*}
\text{case } \text{read } \text{lp} & \text{ of} \\
\text{Nil} & \rightarrow s \\
\text{Cons x xs} & \rightarrow \text{sum xs} (s + x)
\end{align*}
\]
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

```
sum lp s = case read lp of
    Nil → s
    Cons x xs → sum xs (s + x)
```
Sum a list using an accumulator and tail-recursion

\[
\text{sum } \text{lp } s = \\
\text{case read \text{lp} of} \\
\text{Nil} \quad \rightarrow \quad s \\
\text{Cons } x \; xs \quad \rightarrow \quad \text{sum } xs \; (s + x)
\]
Sum a list using an accumulator and tail-recursion

\[
\text{sum } \text{lp } s = \begin{cases} \\
\text{Nil} & \rightarrow s \\
\text{Cons } x \text{ xs} & \rightarrow \text{sum } \text{xs } (s + x) \end{cases}
\]
Sum a list using an accumulator and tail-recursion

\[
\text{sum } \text{lp } s = \begin{cases} 
\text{Nil} & \rightarrow s \\
\text{Cons } x \text{ xs} & \rightarrow \text{sum } \text{xs } (s + x)
\end{cases}
\]
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp \ s = \\
\text{case read } lp \ of \\
\quad \text{Nil} \quad \rightarrow \quad s \\
\quad \text{Cons } x \ \text{xs} \quad \rightarrow \quad \text{sum } \text{xs} \ (s + x)
\]
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp \ s = \begin{cases} 
\text{Nil} & \rightarrow s \\
\text{Cons} \ x \ xs & \rightarrow \text{sum } xs (s + x)
\end{cases}
\]
Sum a list using an accumulator and tail-recursion

\[
\text{sum } \text{lp } s = \begin{cases} 
    s & \text{Nil} \\
    \text{sum } \text{xs} (s + x) & \text{Cons } x \text{ xs}
\end{cases}
\]
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp \ s = \\
\text{case } \text{read } lp \ \text{of} \\
\text{Nil} \quad \rightarrow \ s \\
\text{Cons } x \ xs \rightarrow \ \text{sum} \ xs (s + x)
\]
Sum a list using an accumulator and tail-recursion

```
sum lp s =
  case read lp of
    Nil    → s
    Cons x xs → sum xs (s + x)
```
Sum a list using an accumulator and tail-recursion

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

Sum a list using an accumulator and tail-recursion

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

- **Diagram**

  - **lp**
  - **s**
  - **read**
  - **Nil Cons**
  - **Nil Cons**
  - **x xs**
  - **+**

  Flow diagram illustrating the process of summing a list using an accumulator and tail-recursion.
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp \ s = \begin{cases} 
\text{Nil} & \rightarrow \ s \\
\text{Cons } x \ xs & \rightarrow \ \text{sum } xs \ (s + x)
\end{cases}
\]
Sum a list using an accumulator and tail-recursion

\[
\text{sum } \text{lp } s = \begin{cases} 
\text{Nil} & \rightarrow s \\
\text{Cons } x \text{ xs} & \rightarrow \text{sum } \text{xs } (s + x) 
\end{cases}
\]
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp \ s = \begin{cases} 
\text{Nil} & \rightarrow \ s \\
\text{Cons } x \ xs & \rightarrow \ \text{sum } xs \ (s + x) 
\end{cases}
\]
Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp\ s = \begin{ cases}
\text{Nil} &\rightarrow s \\
\text{Cons } x\ xs &\rightarrow \text{sum } xs\ (s + x)
\end{ cases}
\]
Non-strict functions enables pipelining

- Speedup from non-strict functions due to pipelining
- Best possible speedup from unbounded buffers

Cycles Relative to Strict

<table>
<thead>
<tr>
<th>Function</th>
<th>Append</th>
<th>DFS</th>
<th>Filter</th>
<th>Map</th>
<th>MergeSortTreeMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dataflow to Hardware
A Latency-Insensitive Protocol

Inspired by Carloni et al.
[Cao et al., Memocode 2015]
Input and Output Buffers

Combinational paths broken:
Input buffer breaks \textit{ready} path
Output buffer breaks \textit{data/valid} path
Larger Systems Run Just As Fast

<table>
<thead>
<tr>
<th>Splitters</th>
<th>Token</th>
<th>$F_{\text{max}}$</th>
<th>Area Resources</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bits</td>
<td>MHz</td>
<td>ALMs</td>
<td>%</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>167</td>
<td>189</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>157</td>
<td>350</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>128</td>
<td>152</td>
<td>672</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>128</td>
<td>137</td>
<td>10821</td>
<td>26</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>140</td>
<td>21704</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>158</td>
<td>700</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>145</td>
<td>1409</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>147</td>
<td>2826</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>144</td>
<td>5682</td>
<td>14</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>138</td>
<td>11404</td>
<td>27</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>140</td>
<td>22914</td>
<td>55</td>
</tr>
</tbody>
</table>

Synthesis results on an Altera Cyclone V. 166 MHz target clock rate.
Moore’s Law is alive and well

But we hit a power wall in 2005. Massive parallelism now mandatory

Communication is the culprit
- Dark Silicon is the future: faster transistors; most must remain off

- Custom accelerators are the future; many approaches

- Our project: A Pure Functional Language to FPGAs
Algebraic Data Types in Hardware

Removing recursion

Functional to dataflow

Dataflow to hardware