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...

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

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
The Cray-2: Immersed in Fluorinert

1985 ECL 150 kW
Heat Flux in IBM Mainframes: A Familiar Trend

Liquid Cooled Apple Power Mac G5

2004 CMOS 1.2 kW
"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
Parallelism for Performance; Locality for Efficiency

Dally: “Single-thread processors are in denial about these two facts”

We need different programming paradigms and different architectures on which to run them.
Dark Silicon
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
Taylor and Swanson’s Conservation Cores

**C-core Generation**

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

Huang, Hormati, Bacon, and Rabbah, Liquid Metal, ECOOP 2008.
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.)

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

Greaves and Singh’s Kiwi

```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 \( \text{Gcd}(a, b) \) from Example 1.

Guarded commands and functions to synchronous logic

Hoe and Arvind, *Term Rewriting*, VLSI 1999
Sheeran et al.’s Lava

```haskell
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
```

Functional specifications of regular structures

Kuper et al.’s $\texttt{C}\lambda\texttt{aSH}$

$$\texttt{fir} \ (\texttt{State} \ (xs, \ hs)) \ x = \ (\texttt{State} \ (\texttt{shiftInto} \ x \ xs, \ hs), (x \triangleright xs) \cdot hs)$$

More operational Haskell specifications of regular structures
Baaij, Kooijman, Kuper, Boeijink, and Gerards. $\texttt{C}\lambda\texttt{ash}$, DSD 2010
My Crusade
### Typical Models of Computation That Provide Deterministic Concurrency

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

```haskell
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
data IntList = Cons Int IntList
              | 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

\[
\text{fib } n = \begin{cases} 
1 & \rightarrow 1 \\
2 & \rightarrow 1 \\
\text{n} & \rightarrow \text{fib } (n-1) + \text{fib } (n-2)
\end{cases}
\]
fibk n k = case n of
  1 → k 1
  2 → k 1
  n → fibk (n-1) (λn1 →
       fibk (n-2) (λn2 →
                   k (n1 + n2)))
fib n = fibk n (λx → x)
fib\_k\ n\ k \ = \ \texttt{case}\ n\ \texttt{of}\n\ 1 \ \rightarrow \ k\ 1
\ 2 \ \rightarrow \ k\ 1
\ n \ \rightarrow \ \text{fib}_x\ (n-1)\ (k\ n\ k)

k\ n\ k\ n1 = \ \text{fib}_x\ (n-2)\ (k\ n1\ n2)
k\ n1\ k\ n2 = \ k\ (n1 + n2)
k0\ x = \ x
fib\ n = \ \text{fib}_x\ n\ k0
Represent Continuations with a Type

\[
\textbf{data} \ \text{Cont} = \text{K0} \mid \text{K1 Int Cont} \mid \text{K2 Int Cont}
\]

\[
\text{fibk } n \ k = \text{case } (n,k) \text{ of } \\
\quad (1, \ k) \rightarrow \text{kk } k \ 1 \\
\quad (2, \ k) \rightarrow \text{kk } k \ 1 \\
\quad (n, \ k) \rightarrow \text{fibk } (n-1) \ (\text{K1 } n \ k)
\]

\[
\text{kk } k \ a = \text{case } (k, \ a) \text{ of } \\
\quad ((\text{K1 } n \ k), \ n1) \rightarrow \text{fibk } (n-2) \ (\text{K2 } n1 \ k) \\
\quad ((\text{K2 } n1 \ k), \ n2) \rightarrow \text{kk } k \ (n1 + n2) \\
\quad (\text{K0}, \ x) \rightarrow x
\]

\[
\text{fib } n = \text{fibk } n \ \text{K0}
\]
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
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)
```

Diagram: Flowchart showing data flow from `lp` to `s` with an accumulator and tail-recursion mechanism.
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

$$\text{sum } lp \ s = \cases{\text{Nil} \rightarrow s \cr \text{Cons } x \ xs \rightarrow \text{sum } xs \ (s + x)}$$
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

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

Sum a list using an accumulator and tail-recursion

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

Sum a list using an accumulator and tail-recursion

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

Sum a list using an accumulator and tail-recursion

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

Sum a list using an accumulator and tail-recursion

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

Sum a list using an accumulator and tail-recursion

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

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

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

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

Sum a list using an accumulator and tail-recursion

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

Sum a list using an accumulator and tail-recursion

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

Sum a list using an accumulator and tail-recursion

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

- Speedup from non-strict functions due to pipelining
- Best possible speedup from unbounded buffers
Dataflow to Hardware
A Latency-Insensitive Protocol

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

Input Buf.

Input

data

ready

Core

Output Buf.

Output

data

ready

Combinational paths broken:

Input buffer breaks ready path

Output buffer breaks data/valid path
Larger Systems Run Just As Fast

<table>
<thead>
<tr>
<th>Splitters</th>
<th>Token Bits</th>
<th>$F_{\text{max}}$ MHz</th>
<th>Area Resources ALMs</th>
<th>%</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32</td>
<td>167</td>
<td>189</td>
<td>1</td>
<td>414</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>157</td>
<td>350</td>
<td>1</td>
<td>798</td>
</tr>
<tr>
<td>2</td>
<td>128</td>
<td>152</td>
<td>672</td>
<td>2</td>
<td>1573</td>
</tr>
<tr>
<td>32</td>
<td>128</td>
<td>137</td>
<td>10821</td>
<td>26</td>
<td>25536</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>140</td>
<td>21704</td>
<td>52</td>
<td>51168</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>158</td>
<td>700</td>
<td>2</td>
<td>1621</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>145</td>
<td>1409</td>
<td>3</td>
<td>3261</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>147</td>
<td>2826</td>
<td>7</td>
<td>6559</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>144</td>
<td>5682</td>
<td>14</td>
<td>13148</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>138</td>
<td>11404</td>
<td>27</td>
<td>26414</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>140</td>
<td>22914</td>
<td>55</td>
<td>53087</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