Haskell to Hardware and Other Dreams

Stephen A. Edwards

Columbia University

June 12, 2018
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
  - Transistors: 291 M

- **Xeon E5**
  - 2012
  - 8 cores
  - Transistors: 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
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
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.
## 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</td>
</tr>
<tr>
<td></td>
<td>2001–2006</td>
</tr>
<tr>
<td>Kahn Networks</td>
<td>The 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
Recursion
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

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

fib \ n \ = \ fibk \ n \ (\lambda x \rightarrow x)
Name Lambda Expressions (Lambda Lifting)

```haskell
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

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
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

\[
\begin{align*}
\text{read} & \quad :: \quad \text{CRef} \rightarrow \text{Cont} \\
\text{write} & \quad :: \quad \text{Cont} \rightarrow \text{CRef} \\
\text{data} \quad \text{Cont} & \quad = \quad \text{K0} \mid \text{K1 Int CRef} \mid \text{K2 Int CRef} \\
\text{data} \quad \text{Call} & \quad = \quad \text{Fibk Int CRef} \mid \text{KK Cont Int} \\
\text{fibk} \ z & \quad = \quad \text{case} \ z \ \text{of} \\
& \quad \begin{cases} 
(\text{Fibk} \ 1 \ k) & \rightarrow \ \text{fibk} \ (\text{KK} \ (\text{read} \ k) \ 1) \\
(\text{Fibk} \ 2 \ k) & \rightarrow \ \text{fibk} \ (\text{KK} \ (\text{read} \ k) \ 1) \\
(\text{Fibk} \ n \ k) & \rightarrow \ \text{fibk} \ (\text{Fibk} \ (n-1) \ (\text{write} \ (\text{K1} \ n \ k))) \\
(\text{KK} \ (\text{K1} \ n \ k) \ n1) & \rightarrow \ \text{fibk} \ (\text{Fibk} \ (n-2) \ (\text{write} \ (\text{K2} \ n1 \ k))) \\
(\text{KK} \ (\text{K2} \ n1 \ k) \ n2) & \rightarrow \ \text{fibk} \ (\text{KK} \ (\text{read} \ k) \ (n1 + n2)) \\
(\text{KK} \ \text{K0} \ x) & \rightarrow \ x \\
\text{fib} \ n & \quad = \quad \text{fibk} \ (\text{Fibk} \ n \ (\text{write} \ \text{K0}))
\end{cases}
\end{align*}
\]
Functional IR to Dataflow
Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

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

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

Non-strict function: body starts evaluating \textit{lp} before \textit{s} is available
Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

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

Read: pointer → data
Write: data → pointer
Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

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

Read: pointer \rightarrow data
Write: data \rightarrow pointer
Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

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

Pattern matching with a decomposition mux
Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

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

Pattern matching with a decomposition mux
Sum a list using an accumulator and tail-recursion

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

Tail recursion: physical loop
Functional to Dataflow

[Townsend et al., CC 2017]
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-strictness enables pipeline parallelism: second list element is read before first processed
Sum a list using an accumulator and tail-recursion

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

Buffer sizes affect pipeline depth
Sum a list using an accumulator and tail-recursion

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

s arrives: can start computing sum
[Townsend et al., CC 2017]
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

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

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

![Diagram of dataflow](image)
Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

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

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

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

\[
\text{sum}\ l p\ s = \begin{cases} 
\text{Nil} & \rightarrow s \\
\text{Cons } x\ \text{xs} & \rightarrow \text{sum}\ \text{xs}\ (s + x)
\end{cases}
\]
[Townsend et al., CC 2017] Sum a list using an accumulator and tail-recursion

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

Want *patient* blocks to handle delays from

- Memory systems
- Data-dependent computations

- Full buffers
- Shared resources
- Busy computational units
Patience Through Handshaking

Want *patient* blocks to handle delays from

Memory systems
Data-dependent computations

Full buffers
Shared resources
Busy computational units

<table>
<thead>
<tr>
<th>valid</th>
<th>ready</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Token transferred</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Token valid; held</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>No token to transfer</td>
</tr>
</tbody>
</table>

Latency-insensitive Design (Carloni et al.)
Elastic Circuits (Cortadella et al.)
FIFOs with backpressure
Combinational Function Block

Strict/Unit Rate:
All input tokens required to produce an output

Datapath
Combinational function ignores flow control
Combinational Function Block

Strict/Unit Rate:
All input tokens required to produce an output

Valid network
Output valid if both inputs are valid
Combinational Function Block

Strict/Unit Rate:
All input tokens required to produce an output

Ready network
Input tokens consumed if output token is consumed
(output is valid and ready)
Multiplexer Block

in0 in1 in2
select
out

decoder

in0
in1
in2
select
out
Demultiplexer Block

Diagram showing a demultiplexer block with inputs and outputs labeled as follows:
- Input: in
- Select: select
- Outputs: out0, out1, out2

The diagram illustrates the connections and logic flow within the demultiplexer.
Buffering a Linear Pipeline

Combinational block

Data buffer:
Pipeline register with valid, enable

Control Buffer:
Register diverts token when downstream suddenly stops

Long Combinational Path (Data + Valid)
Long Combinational Path (Ready)

Cao et al. MEMOCODE 2015
Inspired by Carloni's Latency Insensitive Design (e.g., MEMOCODE 2007)
Buffering a Linear Pipeline

Long Combinational Path (Data + Valid)

Inspired by Carloni's Latency Insensitive Design (e.g., MEMOCODE 2007)
Buffering a Linear Pipeline

Data buffer:
Pipeline register with valid, enable

Control Buffer:
Register diverts token when downstream suddenly stops

Long Combinational Path (Data + Valid)
Long Combinational Path (Ready)

Cao et al. MEMOCODE 2015
Inspired by Carloni's Latency Insensitive Design (e.g., MEMOCODE 2007)
Buffering a Linear Pipeline

Combinational block: Data buffer with valid, enable

Control Buffer: Register diverts token when downstream suddenly stops

Long Combinational Path (Data + Valid)

Long Combinational Path (Ready)

Cao et al. MEMOCODE 2015

Inspired by Carloni's Latency Insensitive Design (e.g., MEMOCODE 2007)
Buffering a Linear Pipeline

Control Buffer:
Register diverts token when downstream suddenly stops

Cao et al. MEMOCODE 2015
Inspired by Carloni’s Latency Insensitive Design (e.g., MEMOCODE 2007)
The Problem with Fork

Combinational Block: inputs ready when both valid & output ready

Oops: Combinational Cycle
This is not compositional
The Problem with Fork

Combinational Block: inputs ready when both valid & output ready

Fork: outputs valid only when all are ready

Oops: Combinational Cycle

This is not compositional
The Problem with Fork

Combinational Block: inputs ready when both valid & output ready

Fork: outputs valid only when all are ready

Oops: Combinational Cycle
This is not compositional
The Problem with Fork

- **Combinational Block:** inputs ready when both valid & output ready

- **Fork:** outputs valid only when all are ready

Oops: Combinational Cycle
This is not compositional
**The Problem with Fork**

**Combinational Block:**
inputs ready when both valid & output ready

**Fork:**
outputs valid only when all are ready

**Oops: Combinational Cycle**
This is *not* compositional
The Solution to Combinational Loops

Allowed: Combinational paths from valid to ready

Prohibited: Combinational paths from ready to valid
The Solution to Combinational Loops

Allowed: Combinational paths from valid to ready

Prohibited: Combinational paths from ready to valid
The Solution to Combinational Loops

Allowed: Combinational paths from valid to ready

Prohibited: Combinational paths from ready to valid
The Solution to Combinational Loops

Allowed: Combinational paths from valid to ready

Prohibited: Combinational paths from ready to valid
Valid out ignores ready of other outputs
The Solution to Fork: A Little State

Flip-flop set after token sent suppresses duplicates

Valid out ignores ready of other outputs

Input consumed once one token sent on every output
The Solution to Fork: A Little State

Flip-flop set after token sent suppresses duplicates

Valid out ignores ready of other outputs

Input consumed once one token sent on every output
Nondeterministic Merge

Share with merge/demux
Two-Way Nondeterministic Merge Block w/ Select

“Two-way fork with multiplexed output selected by an arbiter”
- 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

Our project: A Pure Functional Language to FPGAs
- Removing recursion

- Functional to dataflow

- Dataflow to hardware