From Haskell To Hardware

Matthijs Kooijman, Christiaan Baaij & Jan Kuper
Designing Hardware

- **Behavioral descriptions:**
  - *What* the hardware does

- **Structural descriptions:**
  - *How* the hardware does it
`Holy Grail’

- Algorithms often described as a set of mathematical equations

- `Holy Grail’ Hardware descriptions:
  - Input: Mathematical equation
  - Output: Perfect Hardware
Hardware & Functional Languages

- Calculate: $2 \times 3 + 3 \times 4$
- Just like functional languages, there is no pre-ordained order in combinatorial hardware.
- Just like functional languages, operations in hardware can happen in parallel.
- Parallel execution is default in hardware!
Purity & State

- Purity: Same arguments, Same result
- Hardware has State...
- How do we make pure hardware description that have state?
State

\[ \begin{array}{ccc}
A & B & \text{Out} \\
1 & 1 & 1 \\
1 & 2 & 3 \\
1 & 1 & 4 \\
2 & 2 & 8 \\
\end{array} \]
State

\[
macc \ (State \ s) \ (a,b) = \begin{cases} 
    \text{let } \ sum = s + a * b \\
    \text{in } \ (State \ sum, \ sum)
\end{cases}
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Diagram: 
- Multiplication (\(*\))
- Addition (+)
- States flow from left to right.
Simulation

- Simulation is easy:
- Map hardware over series of input variables, using State as accumulator

\[
\text{run } f \ s \ (i:is) = o : (\text{run } f \ s' \ is) \\
\text{where} \\
(s',o) = f \ s \ i
\]
FIR filter

Dot-product:

\[ y = \vec{x} \cdot \vec{h} \]

Applied to a stream of values:
FIR filter

\[ \text{fir (State pxs) } x = (\text{State (pxs}++x), \text{pxs} ** \text{hs}) \]
\[
\text{where}
\]
\[
\text{hs} = [2,3,-2,4]
\]

\[ \text{pxs}: \text{Previous x's (state)} \]
\[ x: \text{New input value} \]
\[ \text{hs}: \text{Coefficients} \]
\[ \text{pxs}++x: \text{Remember new x, remove oldest} \]
\[ \text{pxs}**\text{hs}: \text{dot-product} \]
\[\text{pxs} \leftarrow \text{tail} \ \text{pxs} \ + \text{[x]}\]
\[\text{pxs} \ + \text{hs} = \text{foldl} \ (+) \ 0 \ (\text{zipWith} \ (*) \ \text{pxs} \ \text{hs})\]
We want to translate a functional description to hardware.

Hardware is usually represented by a netlist, a series of components connected by wires.

We translate Haskell to VHDL, an existing hardware description language with available tooling.
CλaSH

• Not all of Haskell has a direct correspondence with hardware:
  • Infinite Lists
  • Dynamic Lists
  • Recursion
  • etc.

• This means there are certain restrictions
CAES Language for Synchronous Hardware

(Mostly) structural descriptions of hardware for synchronous hardware.

Structural properties are not inferred, but have to be specified by the hardware designer.
FIR in CLaSH

type Word = SizedInt D16
type Vec4 = Vector D4 Word

fir :: State Vec4 -> Word -> (State Vec4, Word)
fir (State pxs) x = (State (pxs<++x), pxs ** hs)
  where
    hs = ([2,3,-2,4] :: Vec4)
**FIR in CλaSH**

```haskell
type Word = SizedInt D16

type Vec4 = Vector D4 Word

fir :: State Vec4 -> Word -> (State Vec4, Word)
fir (State pxs) x = (State (pxs<++x), pxs ** hs)
where
  hs = ([2,3,-2,4] :: Vec4)
```

- **hs** actually has to be specified as such:
FIR in CλaSH

```
type Word = SizedInt D16
type Vec4 = Vector D4 Word

fir :: State Vec4 -> Word -> (State Vec4, Word)
fir (State pxs) x = (State (pxs<++x), pxs ** hs)
where
  hs = ([2,3,-2,4] :: Vec4)
```

- hs actually has to be specified as such:
  \hs = $(vectorTH [2::Word,3,-2,4])
Vectors

- The size of the vector is part of the type:
  - **Unconstrained Vector type:**
    \[ \text{NaturalT} \ n \Rightarrow \text{Vector} \ n \ a \]
  - **Example of Constrained Vector type:**
    \[ \text{Vector} \ D4 \ a \]
Compilation Pipeline

\[
\begin{align*}
Haskell & \quad \xrightarrow{\text{GHC (front-end)}} \quad \text{Core} \\
\text{Normalization} & \quad \xrightarrow{} \quad \text{Core} \\
\text{Back-end} & \quad \xrightarrow{} \quad \text{VHDL} \\
\text{Synthesis Tool} & \quad \xrightarrow{} \quad \text{Netlist}
\end{align*}
\]
Normalization

- Normalization: apply transformations until description is in normal form.
- A reduction system
- Around 20 transformation rules
- Properties such as Termination, Church-Rosser are assumed and likely, but not yet proven.
Why normalization?

- Netlist: components connected by wires
- Core does not always correspond directly to a netlist
- Example problem: What is the name of the output port of the following function?

\[
\text{square } x = x \times x
\]
Why normalization?

\[
\text{square } x = x \times x
\]
Transformation

\[
func = E \\
func = \textbf{let} \ res = E \ \textbf{in} \ res
\]

\[
square x = x \times x \\
square x = \textbf{let} \ res = x \times x \ \textbf{in} \ res
\]
Normal form

- Square is now in *normal form*:

```
square :: SizedInt D16 -> SizedInt D16
square x = let res = x * x in res
```

Entity input port Architecture output port
**VHDL**

```
square :: SizedInt D16 -> SizedInt D16

square \( x \) = let res = x * x in res

entity square is
  port (x : in signed (0 to 15);
       res : out signed (0 to 15));
end entity square;

architecture structural of square is begin
  res = resize(x * x, 16);
end architecture structural;
```
Problems

- Dependent types in Haskell are *fake*, only possible through certain extensions to the language.
- At times, we need to prove and add invariants, such as commutativity of addition.
- Haskell lacks proper support for specifying such invariants and their proofs.
Consequences

• Invariants can only be incorporated through term-level proof builders, which are cumbersome in use.

• Invariants usually come into play when dealing with the specification of recursive functions.

• We chose not to expose this need for proofs to a developer.
Consequences

- As proof builders are not supported, developers can not specify recursive functions!

- Temporary solution: (Limited) set of recursive vector transformations are compiled using predefined VHDL templates.
Summary

• $\text{C\lambda\text{aSH}}$ has a solid base
• Lots of work to be done
• Will be used in courses on HW design, and as such hopefully attract many master students
Thanks