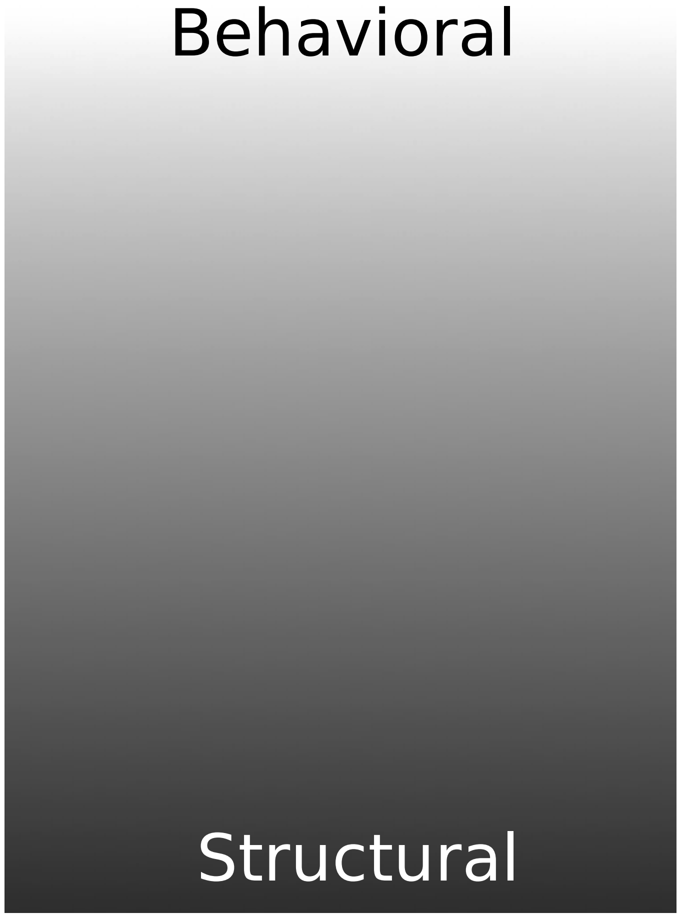
From Haskell To Hardware

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

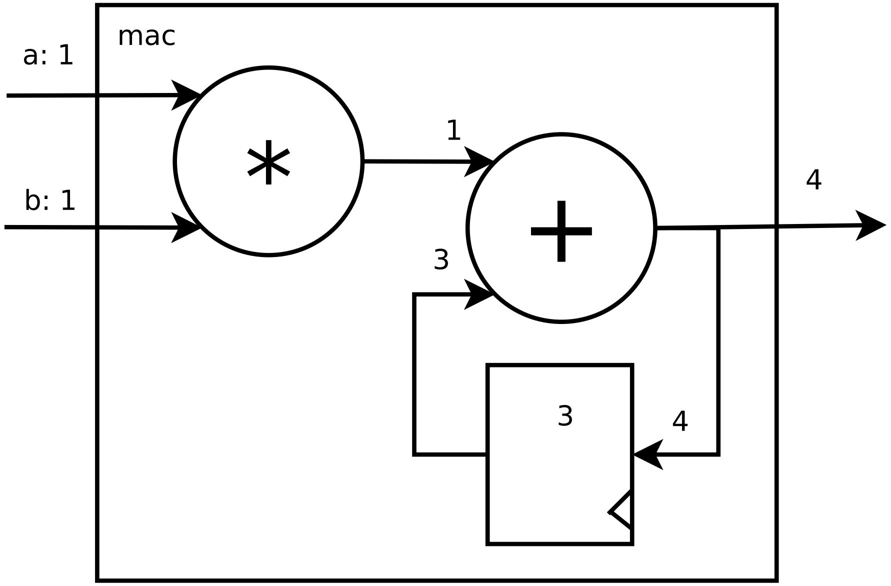
Hardware & Functional Languages

- Calculate: $2*3+3*4$
- Just like functional languages, there is no preordained 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

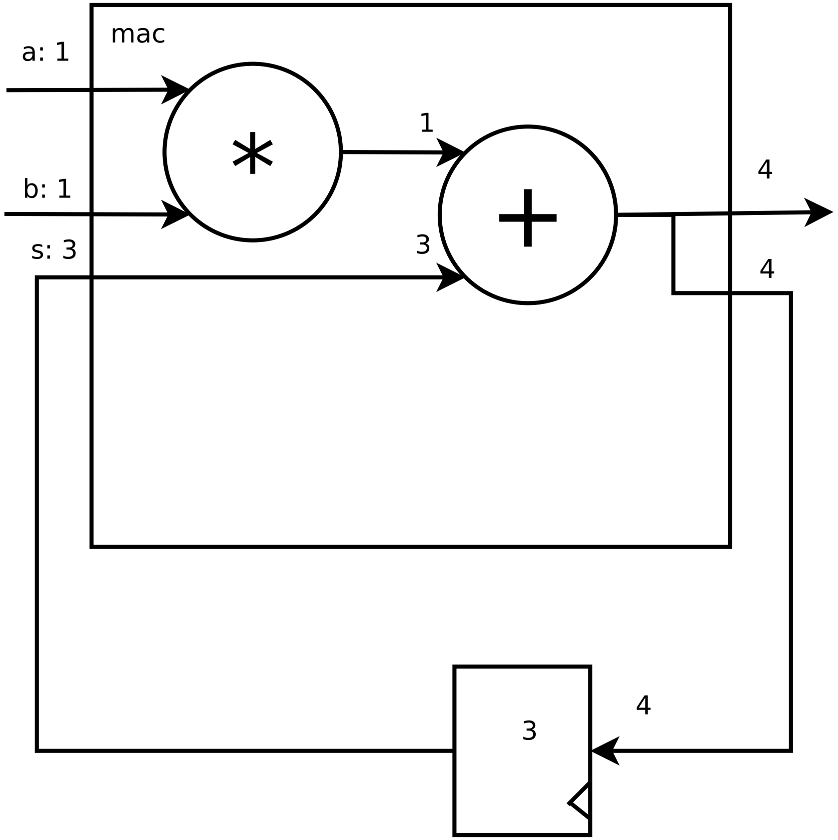


A	B	Out
1	1	1
1	2	3
1	1	4
2	2	8

State

```
macc (State s) (a,b)
      = let sum = s + a * b
        in  (State sum, sum)
```

A	B	S	Out
1	1	0	1
1	2	1	3
1	1	3	4
2	2	4	8



Simulation

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

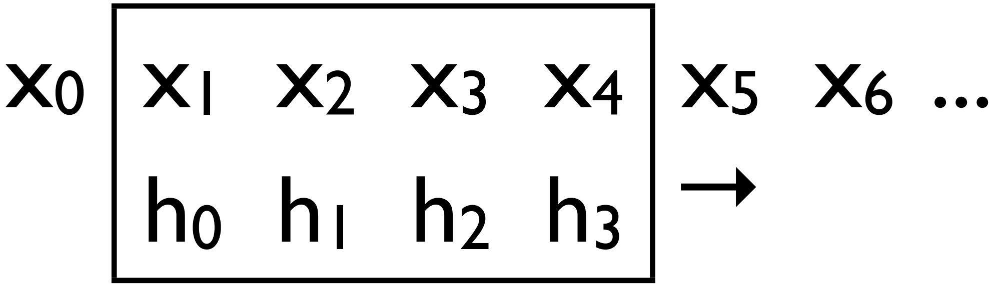
```
run f s (i:is) = o : (run f s’ is)
  where
    (s’,o) = f s i
```

FIR filter

Dot-product:

$$y = \vec{x} \bullet \vec{h}$$

Applied to a stream of values:



x_0 $\boxed{\begin{matrix} x_1 & x_2 & x_3 & x_4 \\ h_0 & h_1 & h_2 & h_3 \end{matrix}}$ $\overset{x_5}{\rightarrow}$ x_6 ...

FIR filter

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

pxs : Previous x’s (state)

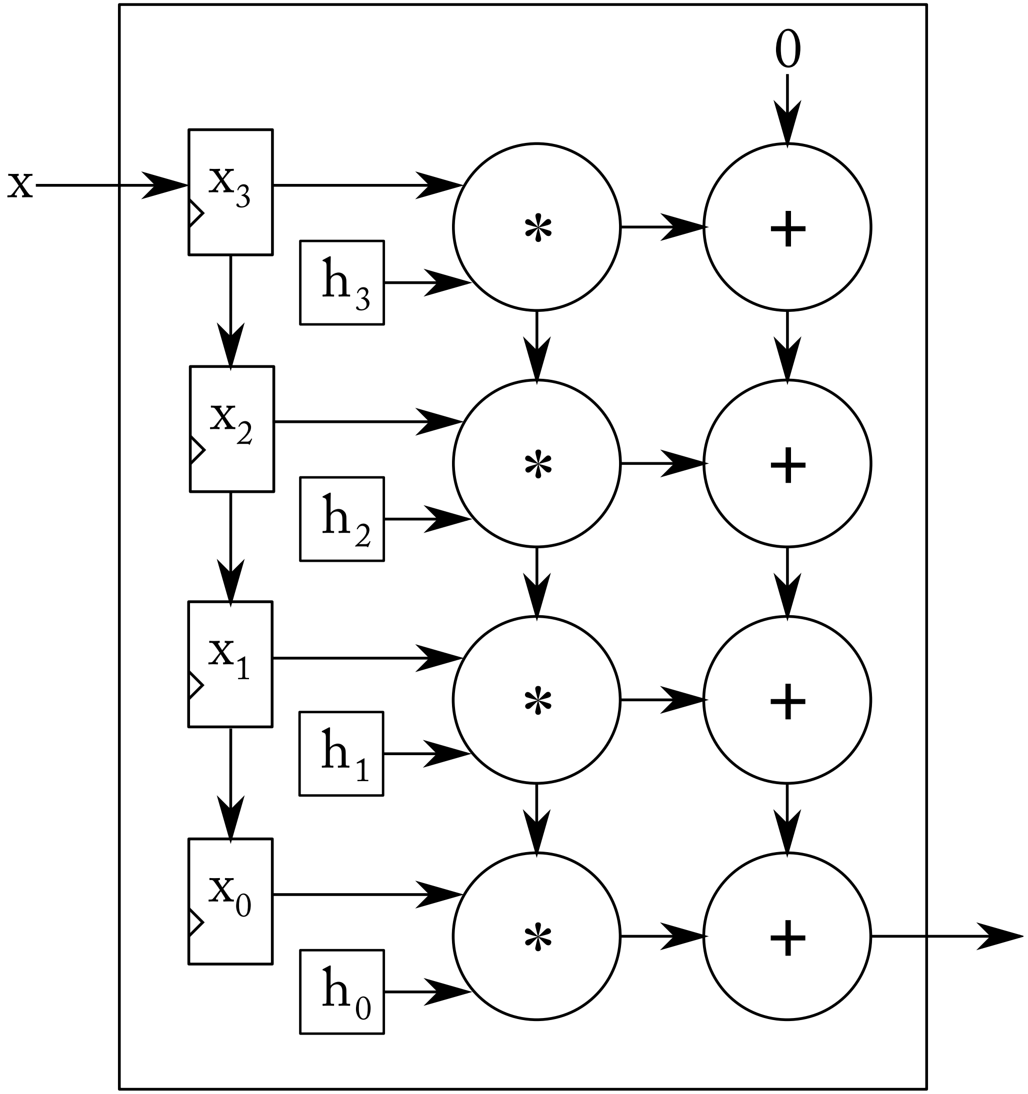
x : New input value

hs : Coefficients

pxs<++x : Remember new x, remove oldest

pxs**hs : dot-product

```
pxs <++ x = tail pxs ++ [x]
pxs ** hs = foldl (+) 0 (zipWith (*) pxs hs)
```



CλaSH

- 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

CλaSH

- 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 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)
```

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:

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:

```
NaturalT n => Vector n a
```

  - Example of Constrained Vector type:

```
Vector D4 a
```

Compilation Pipeline

$$\mathit{Haskell} \xrightarrow{\text{GHC (front-end)}} \mathit{Core}$$

$$\xrightarrow{\text{Normalization}} \mathit{Core}$$

$$\xrightarrow{\text{Back-end}} \mathit{VHDL}$$

$$\xrightarrow{\text{Synthesis Tool}} \mathit{Netlist}$$

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.

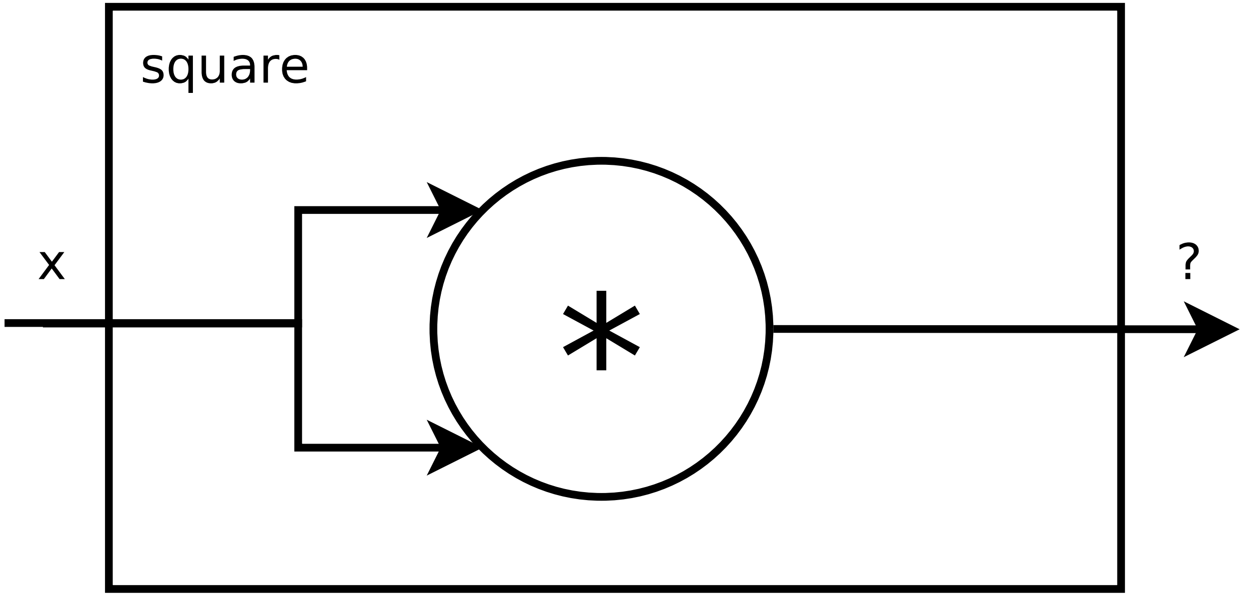
Why normalization?

- 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?

```
square x = x * x
```

Why normalization?

```
square x = x * x
```



Transformation

$$\frac{\text{func} = \quad E}{\text{func} = \quad \textbf{let}\ \text{res} = E\ \textbf{in}\ \text{res}}\ \text{E has no name}$$

$$\frac{\text{square x} = \quad x * x}{\text{square x} = \quad \textbf{let}\ \text{res} = x * x\ \textbf{in}\ \text{res}}$$

Normal form

- Square is now in *normal form*:

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

$$\underbrace{\text{square}}_{\text{Entity}}\quad \underbrace{x}_{\text{input port}}\quad = \underbrace{\text{let res} = x * x}_{\text{Architecture}}\ \text{in}\quad \underbrace{res}_{\text{output port}}$$

VHDL

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

$$\underbrace{\text{square}}_{\text{Entity}}\quad \underbrace{x}_{\text{input port}} \quad = \underbrace{\text{let res} = x * x}_{\text{Architecture}}\ \text{in} \quad \underbrace{res}_{\text{output port}}$$

```
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

- CλaSH 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