



# CλasH

## From Haskell To Hardware

Christiaan Baaij & Matthijs Kooijman

September 3, 2009

Computer Architecture for Embedded Systems (CAES) group  
Faculty of Electrical Engineering, Mathematics and Computer Science  
University of Twente  
Enschede, The Netherlands

<http://caes.ewi.utwente.nl>

- Small tour: what can we describe in CλasH
- Quick real demo



## What is CλasH?

- CλasH: CAES Language for Hardware Descriptions
- Rapid prototyping language
- Subset of Haskell can be translated to Hardware (VHDL)
- Structural Description of a Mealy Machine



## What is CλasH?

- CλasH: CAES Language for Hardware Descriptions
- Rapid prototyping language
- Subset of Haskell can be translated to Hardware (VHDL)
- Structural Description of a Mealy Machine



## What is CλasH?

- CλasH: CAES Language for Hardware Descriptions
- Rapid prototyping language
- Subset of Haskell can be translated to Hardware (VHDL)
- Structural Description of a Mealy Machine



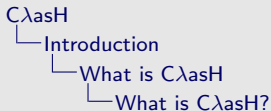
## What is CλasH?

- CλasH: CAES Language for Hardware Descriptions
- Rapid prototyping language
- Subset of Haskell can be translated to Hardware (VHDL)
- Structural Description of a Mealy Machine



## What is CλasH?

- CλasH: CAES Language for Hardware Descriptions
- Rapid prototyping language
- Subset of Haskell can be translated to Hardware (VHDL)
- Structural Description of a Mealy Machine



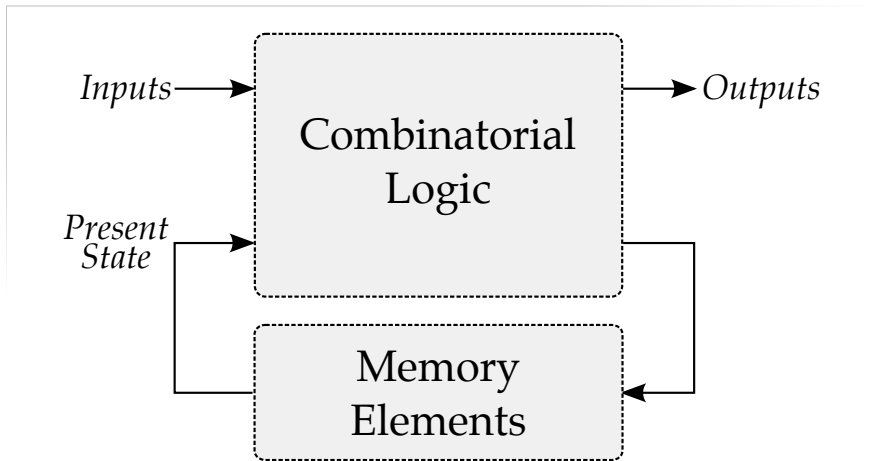
- CλasH: CAES Language for Hardware Descriptions
- Rapid prototyping language
- Subset of Haskell can be translated to Hardware (VHDL)
- Structural Description of a Mealy Machine

- We are a Computer Architectures group, this has been a 6 month project, no prior experience with Haskell.
- CλasH is written in Haskell, of course
- CλasH is currently meant for rapid prototyping, not verification of hardware designs
- Functional languages are close to Hardware
- We can only translate a subset of Haskell
- All functions are descriptions of Mealy Machines





## What again is a Mealy Machine?



2009-08-26

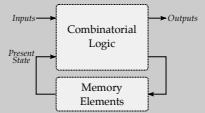
CλasH

Introduction

Mealy Machine

What again is a Mealy Machine?

What again is a Mealy Machine?



- Mealy machine bases its output on current input and previous state



## Haskell Description

```
mealyMachine ::  
  InputSignals →  
  State →  
  (State, OutputSignals)  
mealyMachine inputs state = (new_state, output)  
where  
  new_state = logic state input  
  outputs   = logic state input
```

- Current state is part of the input
- New state is part of the output



## Haskell Description

```
mealyMachine ::  
  InputSignals →  
  State →  
  (State, OutputSignals)  
mealyMachine inputs state = (new_state, output)  
where  
  new_state = logic state input  
  outputs   = logic state input
```

- Current state is part of the input
- New state is part of the output



## Haskell Description

```
mealyMachine ::  
  InputSignals →  
  State →  
  (State, OutputSignals)  
mealyMachine inputs state = (new_state, output)  
where  
  new_state = logic state input  
  outputs   = logic state input
```

- Current state is part of the input
- New state is part of the output

```
mealyMachine ::  
  InputSignals →  
  State →  
  (State, OutputSignals)  
mealyMachine inputs state = (new_state, output)  
where  
  new_state = logic state input  
  outputs   = logic state input
```

- Current state is part of the input
- New state is part of the output

- State is part of the function signature
- Both the current state, as the updated State



## Simulating a Mealy Machine

```
run func state [] = []  
run func state (i : input) = o : out  
where  
  (state', o) = func i state  
  out         = run func state' input
```

- State behaves like an accumulator



## Simulating a Mealy Machine

```
run func state [] = []  
run func state (i : input) = o : out  
  where  
    (state', o) = func i state  
    out        = run func state' input
```

- State behaves like an accumulator





## Simulating a Mealy Machine

```
run func state [] = []  
run func state (i : input) = o : out  
where  
  (state', o) = func i state  
  out          = run func state' input
```

- State behaves like an accumulator

```
run func state [] = []
run func state (i : input) = o : out
  where
    (state', o) = func i state
    out         = run func state' input
```

■ State behaves like an accumulator

- This is just a quick example of how we can simulate the mealy machine
- It sort of behaves like MapAccumN



## Small Use Case

- Small Polymorphic, Higher-Order CPU
- Each function is turned into a hardware component
- Use of state will be simple



## Small Use Case

- Small Polymorphic, Higher-Order CPU
- Each function is turned into a hardware component
- Use of state will be simple



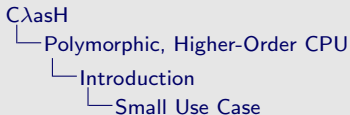
## Small Use Case

- Small Polymorphic, Higher-Order CPU
- Each function is turned into a hardware component
- Use of state will be simple



## Small Use Case

- Small Polymorphic, Higher-Order CPU
- Each function is turned into a hardware component
- Use of state will be simple



- Small Polymorphic, Higher-Order CPU
- Each function is turned into a hardware component
- Use of state will be simple

- Small "toy"-example of what can be done in CλasH
- Show what can be translated to Hardware
- Put your hardware glasses on: each function will be a component
- Use of state will be kept simple



## Imports

Import all the built-in types, such as vectors and integers:

```
import CLasH.HardwareTypes
```

Import annotations, helps CLasH to find top-level component:

```
import CLasH.Translator.Annotations
```





## Imports

Import all the built-in types, such as vectors and integers:

```
import CLasH.HardwareTypes
```

Import annotations, helps CLasH to find top-level component:

```
import CLasH.Translator.Annotations
```



## Imports

Import all the built-in types, such as vectors and integers:

```
import CLasH.HardwareTypes
```

Import annotations, helps CLasH to find top-level component:

```
import CLasH.Translator.Annotations
```



Import all the built-in types, such as vectors and integers:

```
import Clash.Hardware.Types
```

Import annotations, helps ClashH to find top-level component:

```
import Clash.Translator.Annotations
```

- The first input is always needed, as it contains the builtin types
- The second one is only needed if you want to make use of Annotations



## Type definitions

First we define some ALU types:

```
type Op s a = a → Vector s a → a  
type Opcode = Bit
```

And some Register types:

```
type RegBank s a = Vector (s + D1) a  
type RegState s a = State (RegBank s a)
```

And a simple Word type:

```
type Word = SizedInt D12
```



## Type definitions

First we define some ALU types:

```
type Op s a = a → Vector s a → a  
type Opcode = Bit
```

And some Register types:

```
type RegBank s a = Vector (s + D1) a  
type RegState s a = State (RegBank s a)
```

And a simple Word type:

```
type Word = SizedInt D12
```



## Type definitions

First we define some ALU types:

```
type Op s a = a → Vector s a → a  
type Opcode = Bit
```

And some Register types:

```
type RegBank s a = Vector (s + D1) a  
type RegState s a = State (RegBank s a)
```

And a simple Word type:

```
type Word = SizedInt D12
```



## Type definitions

First we define some ALU types:

```
type Op s a = a → Vector s a → a
```

```
type Opcode = Bit
```

And some Register types:

```
type RegBank s a = Vector (s + D1) a
```

```
type RegState s a = State (RegBank s a)
```

And a simple Word type:

```
type Word = SizedInt D12
```

Polymorphic, Higher-Order CPU

Type Definitions

Type definitions

First we define some ALU types:

```
type Op s a = a -> Vector s a -> a
type Opcode = Bit
```

And some Register types:

```
type RegBank s a = Vector (s + D1) a
type RegState s a = State (RegBank s a)
```

And a simple Word type:

```
type Word = SizedInt D12
```

- The first type is already polymorphic, both in size, and element type
- It's a small example, so Opcode is just a Bit
- State has to be of the State type to be recognized as such
- SizedInt D12: One concrete type for now, to make the signatures smaller





## Operations

We make a primitive operation:

```
primOp :: (a → a → a) → Op s a  
primOp f a b = a 'f' a
```

We make a vector operation:

```
vectOp :: (a → a → a) → Op s a  
vectOp f a b = foldl f a b
```

■ We support Higher-Order Functionality



## Operations

We make a primitive operation:

```
primOp :: (a → a → a) → Op s a  
primOp f a b = a 'f' a
```

We make a vector operation:

```
vectOp :: (a → a → a) → Op s a  
vectOp f a b = foldl f a b
```

■ We support Higher-Order Functionality



## Operations

We make a primitive operation:

```
primOp :: (a → a → a) → Op s a  
primOp f a b = a 'f' a
```

We make a vector operation:

```
vectOp :: (a → a → a) → Op s a  
vectOp f a b = foldl f a b
```

■ We support Higher-Order Functionality



## Operations

We make a primitive operation:

```
primOp :: (a → a → a) → Op s a  
primOp f a b = a 'f' a
```

We make a vector operation:

```
vectOp :: (a → a → a) → Op s a  
vectOp f a b = foldl f a b
```

- We support Higher-Order Functionality

└ Polymorphic, Higher-Order CPU

└└ Frameworks for Operations

└└└ Operations

We make a primitive operation:

```
primOp :: (a -> a -> a) -> Op s a
primOp f a b = a'f' a
```

We make a vector operation:

```
vectOp :: (a -> a -> a) -> Op s a
vectOp f a b = foldl f a b
```

■ We support Higher-Order Functionality

- These are just frameworks for 'real' operations
- Notice how they are High-Order functions



## Simple ALU

We define a polymorphic ALU:

*alu* ::

*Op s a* →

*Op s a* →

*Opcode* → *a* → *Vector s a* → *a*

*alu op1 op2 Low a b* = *op1 a b*

*alu op1 op2 High a b* = *op2 a b*

- We support Patter Matching



## Simple ALU

We define a polymorphic ALU:

*alu* ::

*Op s a* →

*Op s a* →

*Opcode* → *a* → *Vector s a* → *a*

*alu op1 op2 Low a b* = *op1 a b*

*alu op1 op2 High a b* = *op2 a b*

- We support Patter Matching

Polymorphic, Higher-Order CPU

Polymorphic, Higher-Order ALU

Simple ALU

We define a polymorphic ALU:

```
alu ::  
  Op s a ->  
  Op s a ->  
  OpCode -> a -> Vector s a -> a  
alu op1 op2 Low a b = op1 a b  
alu op1 op2 High a b = op2 a b
```

■ We support Patter Matching

- Alu is both higher-order, and polymorphic
- We support pattern matching





## Register Bank

Make a simple register bank:

```
registerBank ::
```

```
(Some context...) ⇒ (RegState s a) → a → RangedWord s →  
RangedWord s → Bit → ((RegState s a), a)
```

```
registerBank (State mem) data_in rdaddr wraddr wrenable =  
((State mem'), data_out)
```

**where**

```
data_out = mem ! rdaddr
```

```
mem' | wrenable ≡ Low = mem
```

```
    | otherwise      = replace mem wraddr data_in
```

■ We support Guards



## Register Bank

Make a simple register bank:

```
registerBank ::
```

```
(Some context...) => (RegState s a) -> a -> RangedWord s ->  
RangedWord s -> Bit -> ((RegState s a), a)
```

```
registerBank (State mem) data_in rdaddr wraddr wrenable =  
((State mem'), data_out)
```

**where**

```
data_out = mem ! rdaddr
```

```
mem' | wrenable ≡ Low = mem
```

```
      | otherwise      = replace mem wraddr data_in
```

### ■ We support Guards

Polymorphic, Higher-Order CPU

Register bank

Register Bank

Make a simple register bank:

```
registerBank ::
  (Some context...) => (RegState s a) --> a --> RangedWord s --
  RangedWord s --> Bit -- ((RegState s a), a)
registerBank (State mem) data_in rdaddr wraddr wrenable =
  ((State mem'), data_out)
  where
    data_out = mem ! rdaddr
    mem' | wrenable == Low = mem
         | otherwise      = replace mem wraddr data_in
```

■ We support Guards

- RangedWord runs from 0 to the upper bound
- mem is statefull
- We support guards



## Simple CPU

Combining ALU and register bank:

```
{-#ANN actual_cpu TopEntity#-}
```

```
actual_cpu ::
```

```
(Opcode, Word, Vector D4 Word, RangedWord D9,  
RangedWord D9, Bit) → RegState D9 Word →  
(RegState D9 Word, Word)
```

```
actual_cpu (opc, a, b, rdaddr, wraddr, wren) ram = (ram', alu_out)
```

**where**

```
alu_out = alu (primOp (+)) (vectOp (+)) opc ram_out b  
(ram', ram_out) = registerBank ram a rdaddr wraddr wren
```

■ Annotation is used to indicate top-level component



## Simple CPU

Combining ALU and register bank:

```
{-#ANN actual_cpu TopEntity#-}
```

```
actual_cpu ::
```

```
(Opcode, Word, Vector D4 Word, RangedWord D9,  
RangedWord D9, Bit) → RegState D9 Word →  
(RegState D9 Word, Word)
```

```
actual_cpu (opc, a, b, rdaddr, wraddr, wren) ram = (ram', alu_out)
```

**where**

```
alu_out = alu (primOp (+)) (vectOp (+)) opc ram_out b  
(ram', ram_out) = registerBank ram a rdaddr wraddr wren
```

- Annotation is used to indicate top-level component



## Simple CPU

Combining ALU and register bank:

```
{-#ANN actual_cpu TopEntity#-}
```

```
actual_cpu ::
```

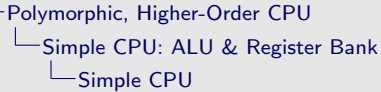
```
(Opcode, Word, Vector D4 Word, RangedWord D9,  
RangedWord D9, Bit) → RegState D9 Word →  
(RegState D9 Word, Word)
```

```
actual_cpu (opc, a, b, rdaddr, wraddr, wren) ram = (ram', alu_out)
```

**where**

```
alu_out = alu (primOp (+)) (vectOp (+)) opc ram_out b  
(ram', ram_out) = registerBank ram a rdaddr wraddr wren
```

- Annotation is used to indicate top-level component



Combining ALU and register bank:

```

{-#ANN actual_cpu TopEntity #-}
actual_cpu ::
  (Opcode Word Vector D4 Word RangedWord D9,
   RangedWord D9 Bit) --> RegState D9 Word --
  (RegState D9 Word Word)
actual_cpu (opc, a, b, rdaddr, wraddr, wren) ram = (ram', alu_out)
  where
    alu_out = alu (primOp (+)) (vectOp (+)) opc ram_out b
    (ram', ram_out) = registerBank ram a rdaddr wraddr wren
  
```

■ Annotation is used to indicate top-level component

- We use the new Annotation functionality to indicate this is the top level
- the primOp and vectOp frameworks are now supplied with real functionality, the plus (+) operations
- No polymorphism or higher-order stuff is allowed at this level.
- Functions must be specialized, and have primitives for input and output



## Demo

- We will simulate the small CPU from earlier
- Translate that CPU code to VHDL
- Simulate the generated VHDL
- See the hardware schematic of the synthesized VHDL





## More than just toys

- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design



## More than just toys

- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design



## More than just toys

- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design



## More than just toys

- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design



## More than just toys

- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design

- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design

- Toys like the poly cpu one are good to give a quick demo
- But we used CλasH to design 'real' hardware
- Reduction circuit sums the numbers in a row of a (sparse) matrix
- Half speed is nice, considering we don't optimize for speed



## So how do you make Hardware from Haskell?

In three simple steps really:

- No Effort:  
GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:  
Transform resulting Core, GHC's Intermediate Language,  
to a normal form
- Easy:  
Translate Normalized Core to synthesizable VHDL



## So how do you make Hardware from Haskell?

In three simple steps really:

- No Effort:  
GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:  
Transform resulting Core, GHC's Intermediate Language,  
to a normal form
- Easy:  
Translate Normalized Core to synthesizable VHDL





## So how do you make Hardware from Haskell?

In three simple steps really:

- No Effort:  
GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:  
Transform resulting Core, GHC's Intermediate Language,  
to a normal form
- Easy:  
Translate Normalized Core to synthesizable VHDL



## So how do you make Hardware from Haskell?

In three simple steps really:

- **No Effort:**  
GHC API Parses, Typechecks and Desugars the Haskell code
- **Hard:**  
Transform resulting Core, GHC's Intermediate Language, to a normal form
- **Easy:**  
Translate Normalized Core to synthesizable VHDL



## So how do you make Hardware from Haskell?

In three simple steps really:

- No Effort:  
GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:  
Transform resulting Core, GHC's Intermediate Language, to a normal form
- Easy:  
Translate Normalized Core to synthesizable VHDL

└─ How do you make Hardware from Haskell?

└─ So how do you make Hardware from Haskell?

So how do you make Hardware from Haskell?

In three simple steps really:

- No Effort:  
GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:  
Transform resulting Core, GHC's Intermediate Language, to a normal form
- Easy:  
Translate Normalized Core to synthesizable VHDL

- Here is a quick insight as to how WE translate Haskell to Hardware
- You can also use TH, like ForSyDe. Or traverse datastructures, like
- We're in luck with the GHC API update of 6.10 and onwards
- Normal form is a single lamda and a let expression, every let binder is a simple assignment



## Some final words

- Still a lot to do: translate larger subset of Haskell
- Real world prototypes can be made in CλasH
- CλasH is another great example of how to bring functional expressivity to hardware designs



# Thank you for listening

ClasH Clone URL:

`git://github.com/christiaanb/clash.git`



## Complete signature for registerBank

```
registerBank ::  
  (NaturalT s  
  , PositiveT (s + D1)  
  , ((s + D1) > s) ~ True) =>  
  (RegState s a) → a → RangedWord s →  
  RangedWord s → Bit → ((RegState s a), a)
```

