



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>



What will we see?

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

2009-08-25

CLasH

Introduction

What will you see

What will we see?

What will we see?

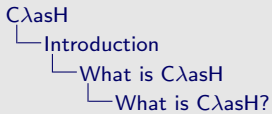
- Small tour: what can we describe in CLasH
- Quick real demo

Virtuele 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

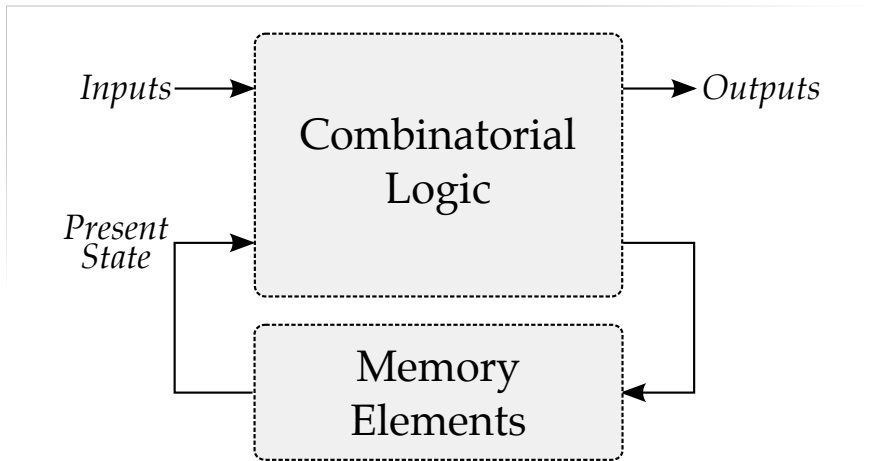


- CλasH: CAES Language for Hardware Descriptions
- Rapid prototyping language
- Subset of Haskell can be translated to Hardware (VHDL)
- Structural Description of a Mealy Machine

- Wij zijn wij
- CλasH voor rapid prototyping
- Subset haskell vertaalbaar
- Mealy machine beschrijving

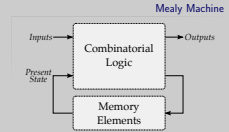


Mealy Machine



2009-08-25

CλasH
├── Introduction
│ ├── Mealy Machine
│ └── Mealy Machine



Voor wie het niet meer weet, dit is een mealy machine



Haskell Description

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




Haskell Description

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



Haskell Description

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



Simulating a Mealy Machine

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



Simulating a Mealy Machine

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



Simulating a Mealy Machine

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



Small Use Case

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




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
```



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
```



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
```



We make a primitive operation:

$$\begin{aligned} \text{primOp} &:: (a \rightarrow a \rightarrow a) \rightarrow \text{Op } s \ a \\ \text{primOp } f \ a \ b &= a \ 'f' \ a \end{aligned}$$

We make a vector operation:

$$\begin{aligned} \text{vectOp} &:: (a \rightarrow a \rightarrow a) \rightarrow \text{Op } s \ a \\ \text{vectOp } f \ a \ b &= \text{foldl } f \ a \ b \end{aligned}$$



We make a primitive operation:

$$\begin{aligned} \text{primOp} &:: (a \rightarrow a \rightarrow a) \rightarrow \text{Op } s \ a \\ \text{primOp } f \ a \ b &= a \ 'f' \ a \end{aligned}$$

We make a vector operation:

$$\begin{aligned} \text{vectOp} &:: (a \rightarrow a \rightarrow a) \rightarrow \text{Op } s \ a \\ \text{vectOp } f \ a \ b &= \text{foldl } f \ a \ b \end{aligned}$$



We make a primitive operation:

$$\begin{aligned} \text{primOp} &:: (a \rightarrow a \rightarrow a) \rightarrow \text{Op } s \ a \\ \text{primOp } f \ a \ b &= a \ 'f' \ a \end{aligned}$$

We make a vector operation:

$$\begin{aligned} \text{vectOp} &:: (a \rightarrow a \rightarrow a) \rightarrow \text{Op } s \ a \\ \text{vectOp } f \ a \ b &= \text{foldl } f \ a \ b \end{aligned}$$



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 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*



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



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



Combining ALU and register bank:

$\{-\#ANN \text{ actual_cpu } TopEntity\#-\}$

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 simpleOp vectorOp opc ram_out b
(ram', ram_out) = registerBank ram a rdaddr wraddr wren
simpleOp = primOp (+)
vectorOp = vectOp (+)



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 simpleOp vectorOp opc ram_out b
(ram', ram_out) = registerBank ram a rdaddr wraddr wren
simpleOp = primOp (+)
vectorOp = vectOp (+)



Is CλasH usable?

- It can be used for more than toy examples
- We designed a matrix reduction circuit
- We simulated it in Haskell
- Simulation results in VHDL match
- Synthesis completes without errors or warnings



Is CλasH usable?

- It can be used for more than toy examples
- We designed a matrix reduction circuit
- We simulated it in Haskell
- Simulation results in VHDL match
- Synthesis completes without errors or warnings



Is CλasH usable?

- It can be used for more than toy examples
- We designed a matrix reduction circuit
- We simulated it in Haskell
- Simulation results in VHDL match
- Synthesis completes without errors or warnings



Is CλasH usable?

- It can be used for more than toy examples
- We designed a matrix reduction circuit
- We simulated it in Haskell
- Simulation results in VHDL match
- Synthesis completes without errors or warnings



Is CλasH usable?

- It can be used for more than toy examples
- We designed a matrix reduction circuit
- We simulated it in Haskell
- Simulation results in VHDL match
- Synthesis completes without errors or warnings



Is CλasH usable?

- It can be used for more than toy examples
- We designed a matrix reduction circuit
- We simulated it in Haskell
- Simulation results in VHDL match
- Synthesis completes without errors or warnings



So how do you make Hardware from Haskell?

In three simple steps

- No Effort:
GHC API Parses, Typechecks and Desugars Haskell
- Hard.. sort of:
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

- No Effort:

GHC API Parses, Typechecks and Desugars
Haskell

- Hard.. sort of:

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

- No Effort:

GHC API Parses, Typechecks and Desugars
Haskell

- Hard.. sort of:

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

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



How do we use CLasH?

As a library:

- Import the module: `CLasH.Translator`
- And call `makeVHDLAnnotations ghc_lib_dir [files_to_translate]`

Customized GHC:

- Call GHC with the `-vhdl` flag
- Use the `:vhdl` command in GHCi



Real Demo

- We will simulate the small CPU from earlier
- Translate the CPU code to VHDL
- Simulate the generated VHDL
- Synthesize the VHDL to get a hardware schematic



Some final words

- Still a lot to do: make a bigger subset of Haskell translatable
- Real world designs work
- We bring functional expressivity to hardware designs



Thank you for listening



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)
```