CλasH

From Haskell To Hardware

Christiaan Baaij & Matthijs Kooijman

Computer Architecture for Embedded Systems
Faculty of EEMCS
University of Twente

August 25, 2009

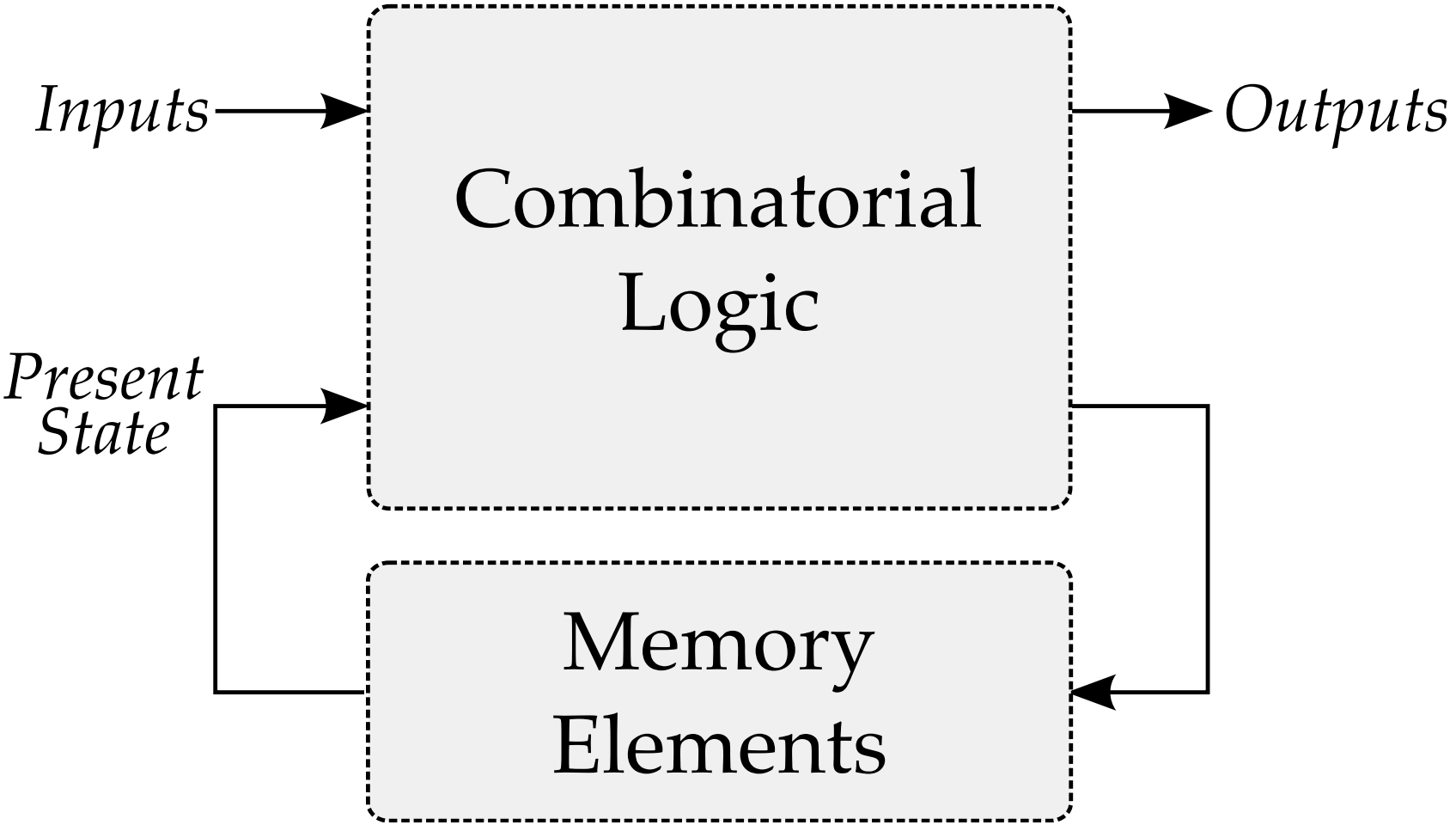
What will we see?

- 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

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

$$
\begin{aligned}
&\mathit{run}\ \mathit{func}\ \mathit{state}\ [\,] = [\,] \\
&\mathit{run}\ \mathit{func}\ \mathit{state}\ (i : \mathit{input}) = o : \mathit{out} \\
&\quad \textbf{where} \\
&\qquad (\mathit{state}', o) = \mathit{func}\ \mathit{state}\ i \\
&\qquad \mathit{out} = \mathit{run}\ \mathit{func}\ \mathit{state}'\ \mathit{input}
\end{aligned}
$$

Simulating a Mealy Machine

$$
\begin{aligned}
&\mathit{run}\ \mathit{func}\ \mathit{state}\ [\,] = [\,] \\
&\mathit{run}\ \mathit{func}\ \mathit{state}\ (i : \mathit{input}) = o : \mathit{out} \\
&\quad \textbf{where} \\
&\qquad (\mathit{state}', o) = \mathit{func}\ \mathit{state}\ i \\
&\qquad \mathit{out} = \mathit{run}\ \mathit{func}\ \mathit{state}'\ \mathit{input}
\end{aligned}
$$

Simulating a Mealy Machine

$$
\begin{aligned}
&\mathit{run}\ \mathit{func}\ \mathit{state}\ [\,] = [\,] \\
&\mathit{run}\ \mathit{func}\ \mathit{state}\ (i : \mathit{input}) = o : \mathit{out} \\
&\quad \textbf{where} \\
&\qquad (\mathit{state}', o) = \mathit{func}\ \mathit{state}\ i \\
&\qquad \mathit{out} = \mathit{run}\ \mathit{func}\ \mathit{state}'\ \mathit{input}
\end{aligned}
$$

Small Use Case

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

Imports

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

Imports

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

Imports

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

First we define some ALU types:

$$\textbf{type } \mathit{Op}\ s\ a\ = a \rightarrow \mathit{Vector}\ s\ a \rightarrow a$$
$$\textbf{type } \mathit{Opcode} = \mathit{Bit}$$

And some Register types:

$$\textbf{type } \mathit{RegBank}\ s\ a = \mathit{Vector}\ (s + \mathit{D1})\ a$$
$$\textbf{type } \mathit{RegState}\ s\ a = \mathit{State}\ (\mathit{RegBank}\ s\ a)$$

And a simple Word type:

$$\textbf{type } \mathit{Word} = \mathit{SizedInt}\ \mathit{D12}$$

First we define some ALU types:

$$\textbf{type } \mathit{Op}\ s\ a\ = a \rightarrow \mathit{Vector}\ s\ a \rightarrow a$$
$$\textbf{type } \mathit{Opcode} = \mathit{Bit}$$

And some Register types:

$$\textbf{type } \mathit{RegBank}\ s\ a = \mathit{Vector}\ (s + \mathit{D1})\ a$$
$$\textbf{type } \mathit{RegState}\ s\ a = \mathit{State}\ (\mathit{RegBank}\ s\ a)$$

And a simple Word type:

$$\textbf{type } \mathit{Word} = \mathit{SizedInt}\ \mathit{D12}$$

First we define some ALU types:

$$\textbf{type } \mathit{Op}\ s\ a\ = a \rightarrow \mathit{Vector}\ s\ a \rightarrow a$$
$$\textbf{type } \mathit{Opcode} = \mathit{Bit}$$

And some Register types:

$$\textbf{type } \mathit{RegBank}\ s\ a = \mathit{Vector}\ (s + \mathit{D1})\ a$$
$$\textbf{type } \mathit{RegState}\ s\ a = \mathit{State}\ (\mathit{RegBank}\ s\ a)$$

And a simple Word type:

$$\textbf{type } \mathit{Word} = \mathit{SizedInt}\ \mathit{D12}$$

We make a primitive operation:

$$primOp :: (a \rightarrow a \rightarrow a) \rightarrow Op\ s\ a$$
$$primOp\ f\ a\ b = a\ `f`\ a$$

We make a vector operation:

$$vectOp :: (a \rightarrow a \rightarrow a) \rightarrow Op\ s\ a$$
$$vectOp\ f\ a\ b = foldl\ f\ a\ b$$

We make a primitive operation:

$$primOp :: (a \rightarrow a \rightarrow a) \rightarrow Op\ s\ a$$
$$primOp\ f\ a\ b = a\ `f`\ a$$

We make a vector operation:

$$vectOp :: (a \rightarrow a \rightarrow a) \rightarrow Op\ s\ a$$
$$vectOp\ f\ a\ b = foldl\ f\ a\ b$$

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
```

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:

$$
\begin{aligned}
&\mathit{registerBank} :: \\
&\quad (\mathit{Some\ context...}) \Rightarrow \\
&\quad (\mathit{RegState\ s\ a}) \rightarrow a \rightarrow \mathit{RangedWord\ s} \rightarrow \\
&\quad \mathit{RangedWord\ s} \rightarrow \mathit{Bit} \rightarrow ((\mathit{RegState\ s\ a}), a) \\
&\mathit{registerBank}\ (\mathit{State\ mem})\ \mathit{data\_in\ rdaddr\ wraddr\ wrenable} = \\
&\quad ((\mathit{State\ mem'}), \mathit{data\_out}) \\
&\quad \textbf{where} \\
&\qquad \mathit{data\_out} = \mathit{mem} \,!\, \mathit{rdaddr} \\
&\qquad \mathit{mem'} \quad \mid \mathit{wrenable} \equiv \mathit{Low} = \mathit{mem} \\
&\qquad\qquad\quad \mid \mathit{otherwise} \quad\ = \mathit{replace\ mem\ wraddr\ data\_in}
\end{aligned}
$$

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 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 CλasH?

As a library:

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

Use 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 word designs work
- We bring functional expressivity to hardware designs

Thank you for listening