High level architectural modelling for early estimation of power and performance

Wouter Swierstra
joint work with Koen Claessen, Carl Seger, Mary Sheeran, and Emily Shriver
Project stats

- Started February 2009.
- One year funding from Intel.
- Collaboration between:
  - Intel (Emily Shriver and Carl Seger);
  - Chalmers (Koen Claessen, Mary Sheeran, and myself).
Aim

- Try to design a language that:
  - can work at different levels of abstraction;
  - is capable of early estimations of performance and power;
  - builds on our functional programming expertise.
- Behavioural
  - Hawk (Cook, Launchbury, Matthews)
- Structural
  - Lava (Bjesse, Claessen, Sheeran, Singh)
Behavioural

- Hawk (Cook, Launchbury, Matthews)

- Lava (Bjesse, Claessen, Sheeran, Singh)

- Wired (Axelsson, Claessen, Sheeran)

Structural
Behavioural

- Hawk (Cook, Launchbury, Matthews)
- Lava (Bjesse, Claessen, Sheeran, Singh)
- Wired (Axelsson, Claessen, Sheeran)

Structural

- This project
Lava

• A data type for primitive gates (and, not,...);
• Haskell combinators to assemble circuits (sequential composition, butterfly circuits, ...)
• VHDL generation for circuits;
• Simulation and testing using QuickCheck;
• Hooks into automatic theorem provers.
Hawk

- **Idea:** use Haskell as an executable hardware specification language.
- "Shallow embedding" – there is no separate data type to represent the AST.
Signals assign values to every clock cycle:

```haskell
type Signal a = Int -> a
```
Hawk combinators – I

Haskell functions to manipulate signals:

constant :: a -> Signal a
constant x = \c -> x

lift :: (a -> b) -> Signal a -> Signal b
lift f signal = \c -> f (signal c)
Hawk combinators – II

delay :: a -> Signal a -> Signal a
delay x s = \c -> if c == 0 then x else s (c-1)

select :: Signal Bool -> Signal a -> Signal a -> Signal a
select cs ts es = \c -> if cs c then ts c else es c
Counter example

• Using these combinators we can define a resettable counter:

```haskell
counter :: Signal Bool -> Signal Int
counter resets = out
  where out = select reset (constant 0)
        (delay 0 (lift incr out))
```

• (This definition relies on lazy evaluation.)
Non-trivial examples

- Hawk has been used to describe microprocessors
- ALU and register files;
- pipelining;
- branch prediction;
- ...

•
**ALU**

```haskell
data Cmd = ADD | SUB | INCR

alu :: Signal Cmd -> Signal (Int,Int) -> Signal Int
alu = lift eval

where eval ADD  (x,y) = x + y
     eval SUB  (x,y) = x - y
     eval INCR (x,_) = x + 1
```
**Register file**

```
data R = R0 | R1 | R2 | R3

type Regs = (Int, Int, Int, Int)

regFile :: Signal (Reg, Int) -> Signal Reg -> Signal Reg -> (Signal Int, Signal Int)
  regFile = loop initRegs regStep

where
  loop :: s -> (s -> (a, s)) -> Signal a
  regStep :: Regs -> ((Int, Int), Regs)
```
We can assemble these pieces:

```
sham :: (Signal Cmd, Signal Reg, Signal Reg, Signal Reg) -> (Signal Reg, Signal Int)
sham (cmds, destReg, srcA, srcB) = ...
```

... by using our register file to lookup the state of the source registers;

and passing this on to the alu.
Hawk review

- **Pro**: easy to write down executable specs;
- **Con**: you can’t do anything with these specs besides execute them.
  - No generating VHDL;
  - No automatic theorem proving;
  - No power or performance analysis.
Goal

• Can we design a Hawkish specification language that
  • is capable of early power and performance estimates?
  • can be integrated with structural languages like Wired and Lava?
Problem

Suppose we want to write an interpreter for this language:

```haskell
data Expr = Val Int
           | Add Expr Expr
           | Eq Expr Expr
           | If Expr Expr Expr
```
Evaluation

eval (Val i) = i

eval (Add l r) = eval l + eval r

eval (Eq x y) = eval x == eval y

eval (If c t e) =

    if (eval c) then eval t else eval e
Evaluation

\[
\begin{align*}
\text{eval} & \::\: \text{Expr} \rightarrow ??? \\
\text{eval} \ (\text{Val} \ i) & = i \\
\text{eval} \ (\text{Add} \ l \ r) & = \text{eval} \ l + \text{eval} \ r \\
\text{eval} \ (\text{Eq} \ x \ y) & = \text{eval} \ x \ == \text{eval} \ y \\
\text{eval} \ (\text{If} \ c \ t \ e) & = \\
\text{if} & \ (\text{eval} \ c) \ \text{then} \ \text{eval} \ t \ \text{else} \ \text{eval} \ e
\end{align*}
\]
data Expr a where

Val :: Int -> Expr Int
Add :: Expr Int -> Expr Int -> Expr Int
Eq :: Expr Int -> Expr Int -> Expr Bool
If :: Expr Bool ->

Expr a -> Expr a -> Expr a
Evaluation revisited

eval :: Expr a -> a

eval (Val i) = i

eval (Add l r) = eval l + eval r

eval (Eq x y) = eval x  == eval y

eval (If c t e) =

    if (eval c) then eval t else eval e
Deeper embedding

```
data Hawk a where
    Pure :: a -> Hawk a
    App :: Hawk (b -> a) -> Hawk b -> Hawk a
    Delay :: a -> Hawk a -> Hawk a
```
Deeper embedding

```haskell
data Hawk a where
    Pure :: a -> Hawk a
    App :: Hawk (b -> a) -> Hawk b -> Hawk a
    Delay :: a -> Hawk a -> Hawk a
```

I’ll use an infix operator `<*>` instead of `App`
Example - mux

select :: Hawk Bool -> Hawk a -> Hawk a -> Hawk a
select cs ts es =
pure (\c t e -> if c then t else e)
  <*> cs
  <*> ts
  <*> es
Example - recursion

- We can still use recursion:

```haskell
iterate :: a -> Hawk (a -> a) -> Hawk a
iterate x h =
    delay x (h <*> iterate x h)
```
Counter example revisited

- We can still define the counter:

```haskell
counter :: Signal Bool -> Signal Int
counter reset = iterate 0 c
    where c = select reset (pure (const 0))
          (pure increment)
```
It is easy to extract a pure Haskell function that executes “the first clock cycle”

exec :: Hawk a -> a
exec (Pure x) = x
exec (Delay x _) = x
exec (App f x) = (exec f) (exec x)
Simulation

- It is easy to extract original Hawk signal functions:

```haskell
simulate :: Hawk a -> Signal a
simulate (Pure x) = \c -> x
simulate (Delay x h) = \c -> if c == 0 then x else h (c-1)
simulate (App f x) = \c -> (simulate f c) (simulate x c)
```
Recap

• Hypothesis: writing specs using these combiners is no harder than in Hawk;

• ...but we now have more structure at our disposal.

• We can use this info to do other analyses.
Example: circuit visualisation

- If we assign names to the pure components, we can traverse the circuit to extract the call graph...
- ...and visualise the circuit using Graphviz.
The counter graph
So what?

- We can (hopefully) use this structure to analyse processor specifications.
- For example, to estimate performance, pass a token through the flow network, assigning symbolic delay to each edge.
- *Processor Performance Modeling using Symbolic Simulation*; Omid Azizi, Jamison Collins, Dinesh Patil, Hong Wang and Mark Horowitz
Open questions

- How can we combine behavioural (Hawk) and structural (Lava, Wired) languages?
- How can we “plug” Lava circuits into a Hawk spec? Or show that a Lava circuit implements some Pure part of the spec?
- How can we change the level of abstraction? Or iteratively develop circuits? What is the right meta-abstraction?