Me and my research

Wouter Swierstra

Vector Fabrics, 6/11/09

Brief bio

- MSc in Software Technology (Utrecht);
- PhD entitled *A Functional Specification of Effects* (University of Nottingham);
- Postdoc position (Chalmers University of Technology).

Dependent types

Notice a pattern?

```
val split8 : Word16 -> Word8 * Word8

val split16 : Word32 -> Word16 * Word16

val split32 : Word64 -> Word32 * Word32

....
```

Dependent types

```
type Word : Nat -> Type
val split : (n : Nat) ->
  Word (n + n) -> Word n * Word n
```

Dependent types are expressive.

Notice any similarities?

$$\frac{\texttt{isEven : int -> bool} \qquad \texttt{5 : int}}{\texttt{isEven(5) : bool}}$$

Notice any similarities?

$$\frac{\texttt{isEven : int -> bool} \qquad \texttt{5 : int}}{\texttt{isEven(5) : bool}}$$

$$\frac{p \rightarrow q \qquad p}{q} \quad \text{Modus ponens}$$

Curry-Howard isomorphism

- A type system is a logic;
- a type is a proposition;
  - $a \to b \to a$
- a program is a proof.
  - λxλy. x

Simple types =
propositional logic;

Dependent types =
predicate logic.

Where's the research?

- The next generation of functional programming languages will have dependent types (Epigram, Coq, Agda, Trellys).
- Dependent types are great, but...
- ... programs must be terminating and pure;
- How can we write and verify 'real' programs?

Hardware description & functional languages

Project stats

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

Behavioural

Structural

Behavioural

- Hawk (Cook, Launchbury, Matthews)

- Lava (Bjesse, Claessen, Sheeran, Singh)

Structural

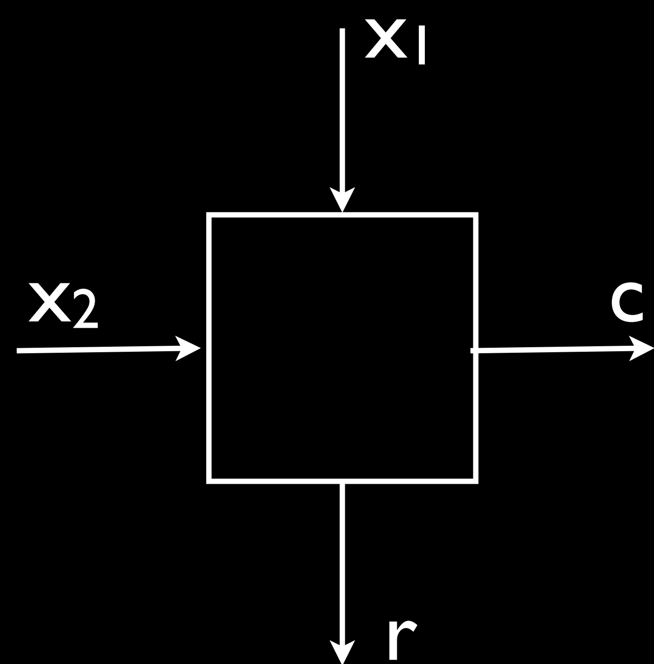
Behavioural

- Hawk (Cook, Launchbury, Matthews)
- Our project
- Lava (Bjesse, Claessen, Sheeran, Singh)

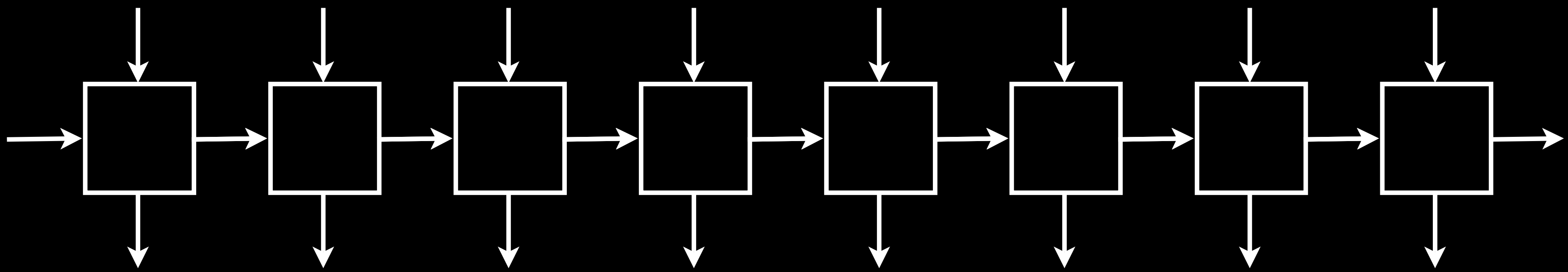
Structural

Lava – core type

```
type lava =
  And of lava * lava
  | Or of lava * lava
  | Not of lava
  | Const of bool
  | ...
```




```
bit_adder x1 x2 =
(and x1 x2, xor x1 x2)
```



```
byte_adder = row 8 bit_adder
```

Lava – simulation

```
let rec sim c = match c with
  | and c1 c2 = (sim c1) && (sim c2)
  | or c1 c2 = (sim c1) || (sim c2)
  | const b = b
  | ...
```

Lava – summary

- A data type for primitive gates (and, not,...);
- Haskell combinators to assemble circuits (sequential, parallel, row, 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 structure of our circuits.

Hawk - Signals

Signals assign values to every clock cycle:

```
type 'a Signal = Int -> a
```

Hawk combinators – I

Haskell functions to manipulate signals:

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

lift :: (‘a -> ‘b) ->
  ‘a Signal -> b’ Signal
lift f signal = \c -> f (signal c)
```

Hawk combinators – II

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

mux :: bool Signal
  -> ‘a Signal-> ‘a Signal -> ‘a Signal
mux cs ts es =
  \c -> if cs c then ts c else es c
```

Non-trivial examples

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

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 Lava?

Problem

Suppose we want to write an interpreter for this language:

```
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

```
eval :: Expr -> ???

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
```

GADTs

```
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
```

Chalk: a deeper embedding

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

Chalk: a deeper embedding

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

I'll use an infix operator `<*>` instead of `App`

ALU

```
data Cmd = ADD | SUB | INCR

alu :: Chalk Cmd -> Chalk (Int,Int) ->
       Chalk Int

alu cmds args =

  pure eval <*> cmds <*> args

  where eval ADD  (x,y) = x + y

        eval SUB  (x,y) = x - y

        eval INCR (x,_) = x + 1
```

Example - recursion

- We can still use recursion:

```
iterate ::
  a -> Chalk (a -> a) -> Chalk a
iterate x h =
  delay x (h <*> iterate x h)
```

Simulation

- It is easy to extract original Hawk signal functions:

```
simulate :: Chalk 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 combinators 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.

Example: pipeline depth

```
depth :: Chalk a -> Signal a
depth (Pure x) = 0
depth (Delay x h) = 1 + depth h
depth (App f x) = max (depth f) (depth x)
```

Latest results

- Provide users with a language to assigns ‘costs’ (power/performance/etc.) to various pure functions;
- Simulate these circuits and compute costs;
- This can be extended to handle symbolic simulation.

Questions?