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?

\[
isEven: \text{int} \rightarrow \text{bool} \quad 5: \text{int} \\
\]

\[
isEven(5): \text{bool}
\]
Notice any similarities?

\[ \text{isEven : int} \rightarrow \text{bool} \quad \text{5 : int} \]

\[ \text{isEven(5) : bool} \]

\[ p \rightarrow q \quad p \]

\[ q \quad \text{Modus ponens} \]
Curry-Howard isomorphism

• A type system is a logic;
• a type is a proposition;
  • \( a \rightarrow b \rightarrow a \)
• a program is a proof.
  • \( \lambda x \lambda y. x \)
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
  - Hawk (Cook, Launchbury, Matthews)
- Structural
  - Lava (Bjesse, Claessen, Sheeran, Singh)
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 \( x_1 \) \( x_2 \) = 

\((\text{and } x_1 \ x_2, \ \text{xor } x_1 \ x_2)\)
byte_adder = row 8 bit_adder
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.
• **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:

\[
\text{type } \forall a \text{ Signal } = \text{Int } \rightarrow a
\]
Hawk combinators – I

Haskell functions to manipulate signals:

\[ \text{constant :: } \forall a. \text{ 'a -> 'a Signal} \]
\[ \text{constant } x = \lambda c \rightarrow x \]

\[ \text{lift :: } (\forall a. \forall b. \text{ 'a -> 'b}) \rightarrow \text{ 'a Signal -> b’ Signal} \]
\[ \text{lift } f \text{ signal } = \lambda c \rightarrow f \left( \text{signal } c \right) \]
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:

```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

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

```haskell
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**

\[
data \text{ Cmd} = \text{ADD} \mid \text{SUB} \mid \text{INCR}
\]

\[
\text{alu} :: \text{Chalk Cmd} \rightarrow \text{Chalk (Int,Int)} \rightarrow \text{Chalk Int}
\]

\[
\text{alu cmds args} =
\]

\[
\text{pure eval <*> cmds <*> args}
\]

\[
\text{where eval ADD } (x,y) = x + y
\]

\[
\text{eval SUB } (x,y) = x - y
\]

\[
\text{eval INCR } (x,_) = x + 1
\]
Example - recursion

- We can still use recursion:

```haskell
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:

```haskell
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 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.
Example: pipeline depth

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