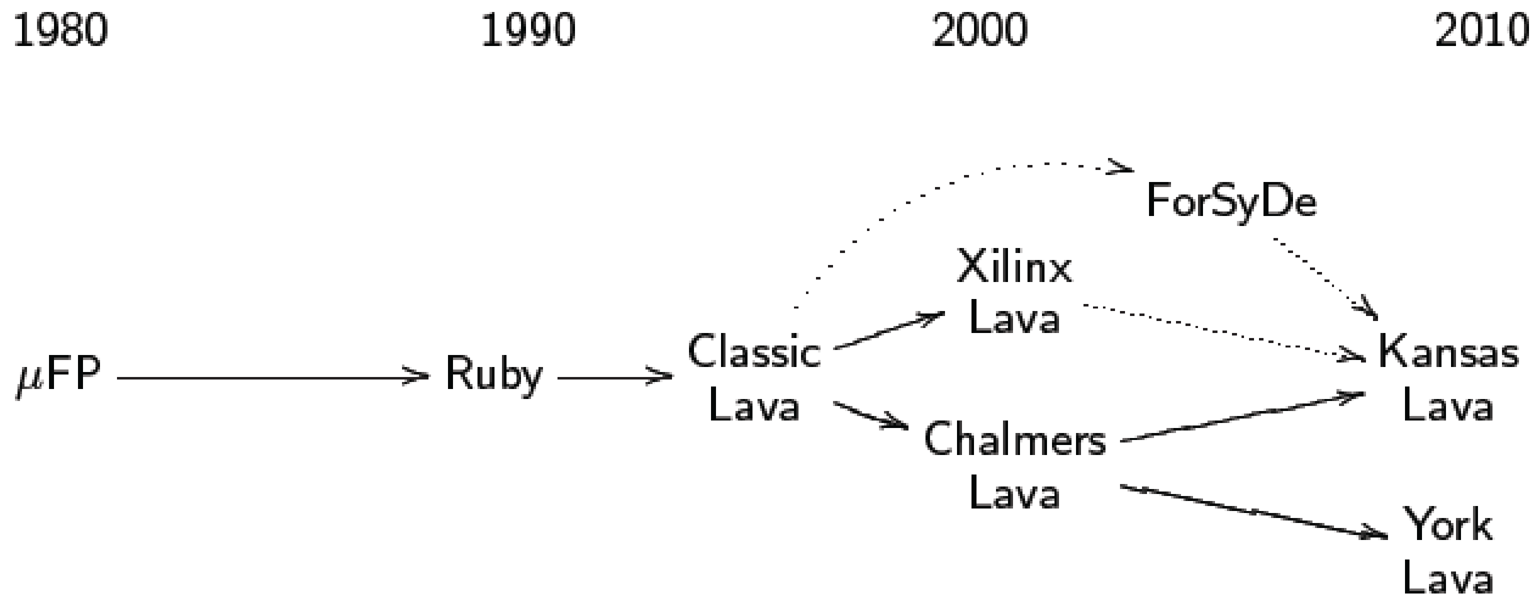
Functional programming and hardware design: where to now??

Wouter Swierstra, Koen Claessen,
Carl Seger, Emily Shriver,
Mary Sheeran

Lava History



Circuits as Relations

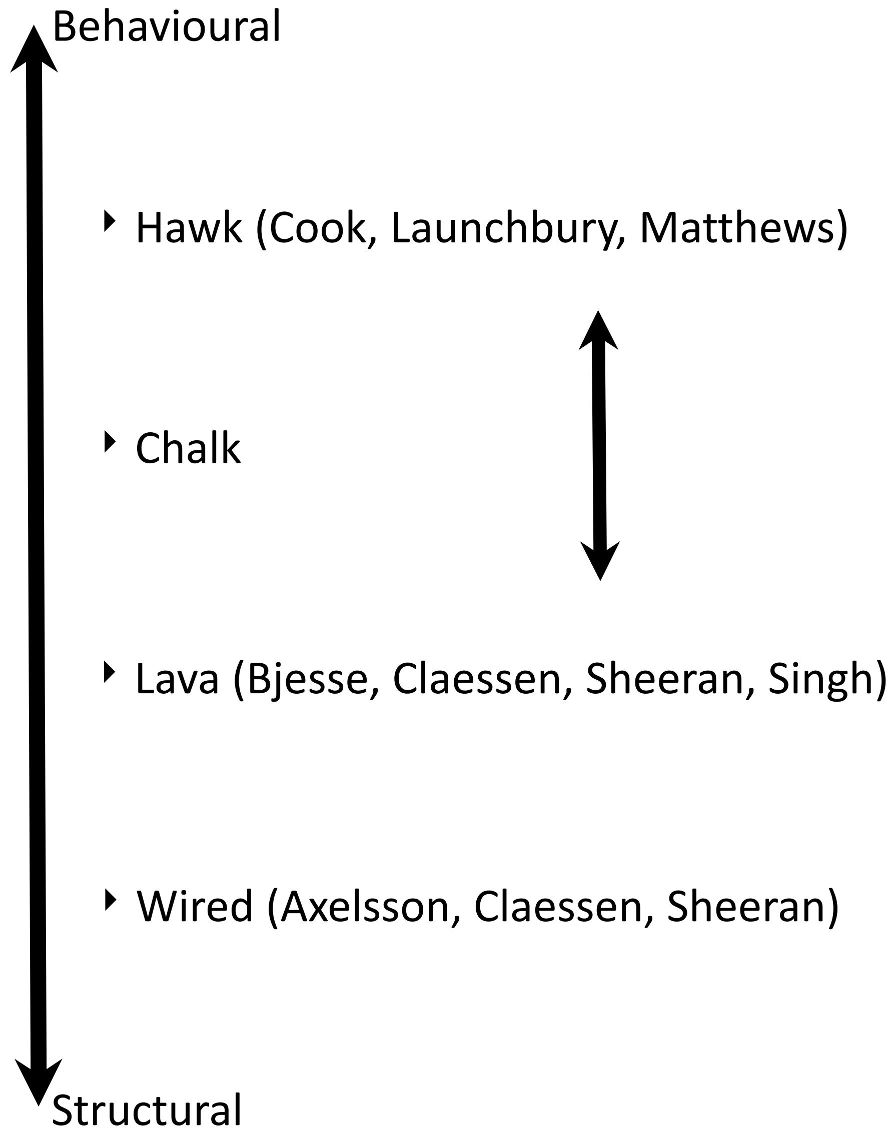
Deep DSL of Functions

Overloading of Interpretations

Addressing Observable Sharing

Modern Functional Programming

with thanks to Andy Gill (U Kansas)

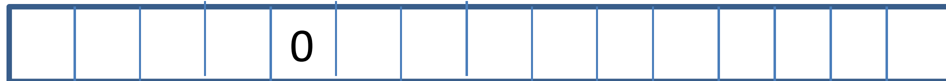

Behavioural
Hawk (Cook, Launchbury, Matthews)
Chalk
Lava (Bjesse, Claessen, Sheeran, Singh)
Wired (Axelsson, Claessen, Sheeran)
Structural

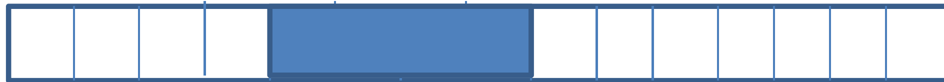
Lava

```
data Gate c where
   And :: c -> c -> Gate
   Or   :: c -> c -> Gate
   Not  :: c -> Gate
   …

data Lava where
   Circuit :: Ref (Gate Lava) -> Lava
```

Lava

Key FP idea is use of higher order functions (combinators)

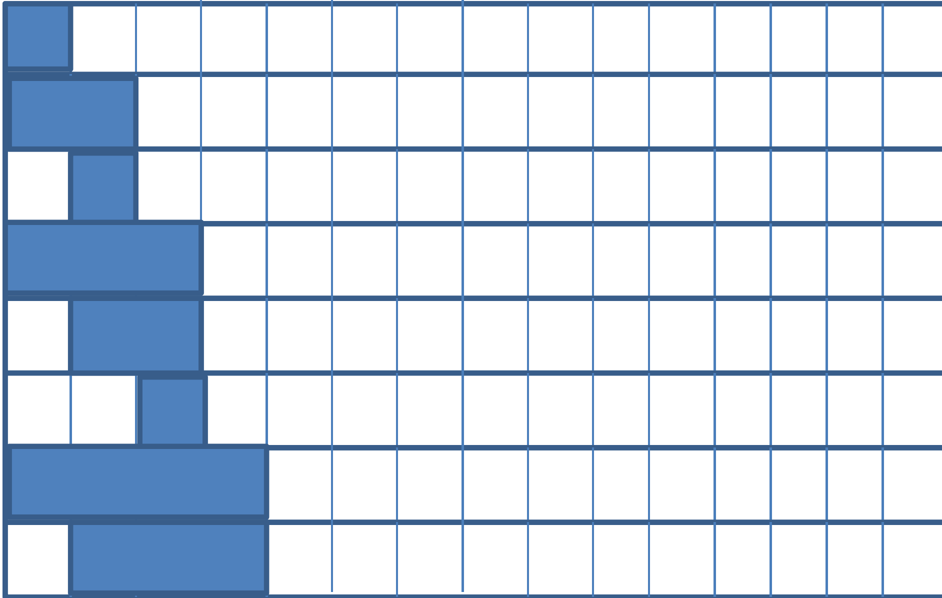




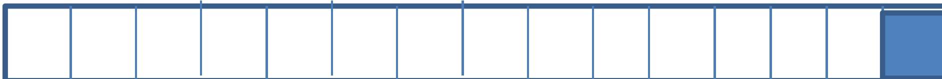
j i

bitBlock i j comp

sorter



. . .



sorter

```
itSort n = compose [blockBit (n-j) (j-k) comp | j <- [0..n], k <-[0..j]]
```

. . .

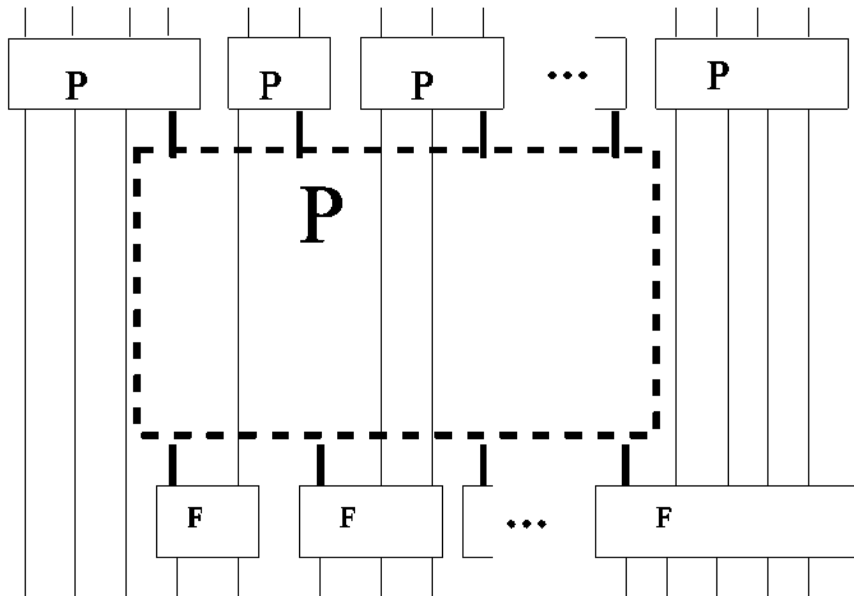
Lava

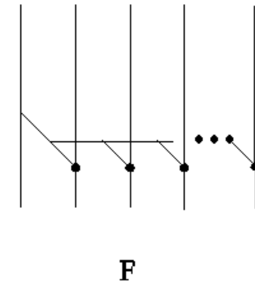
Things really take off when you use the host language in sophisticated ways

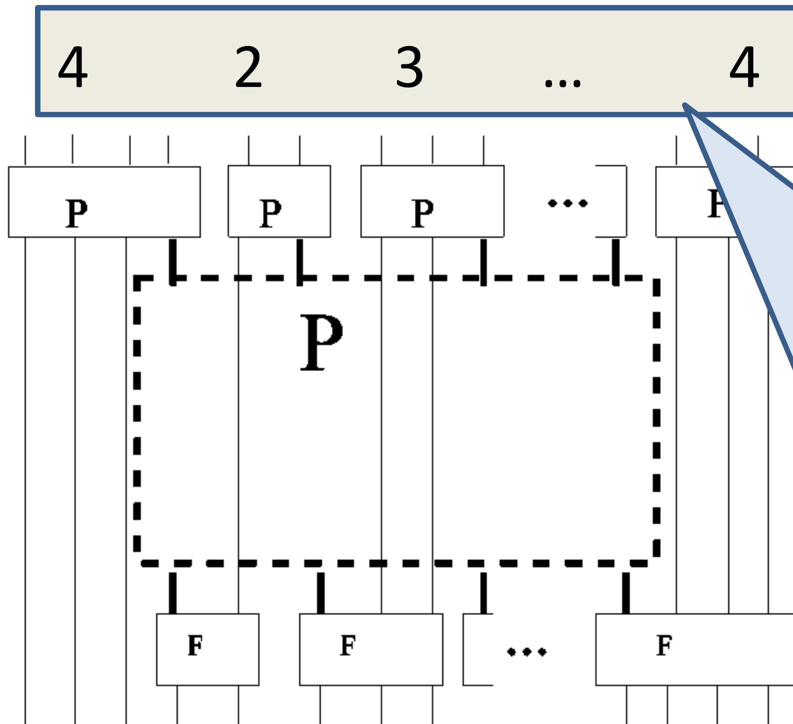
For example, searching for prefix networks

(I didn’t do this in Lava but could have.
We connected the work to Wired and produced layout.)

recursive pattern







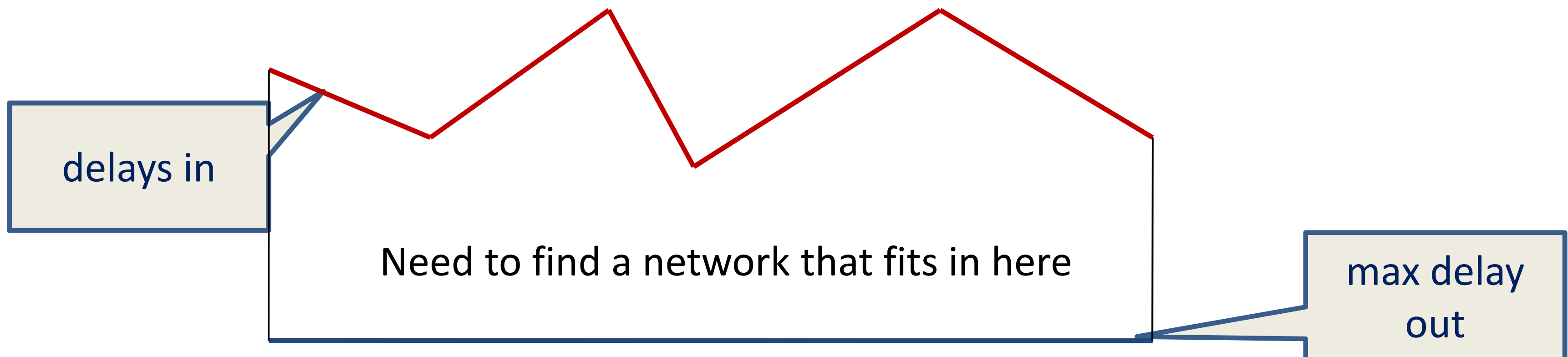
This sequence of numbers characterises one decomposition

Search!

need a measure function (e.g. number of operators)

Need the idea of a context into which a network should fit

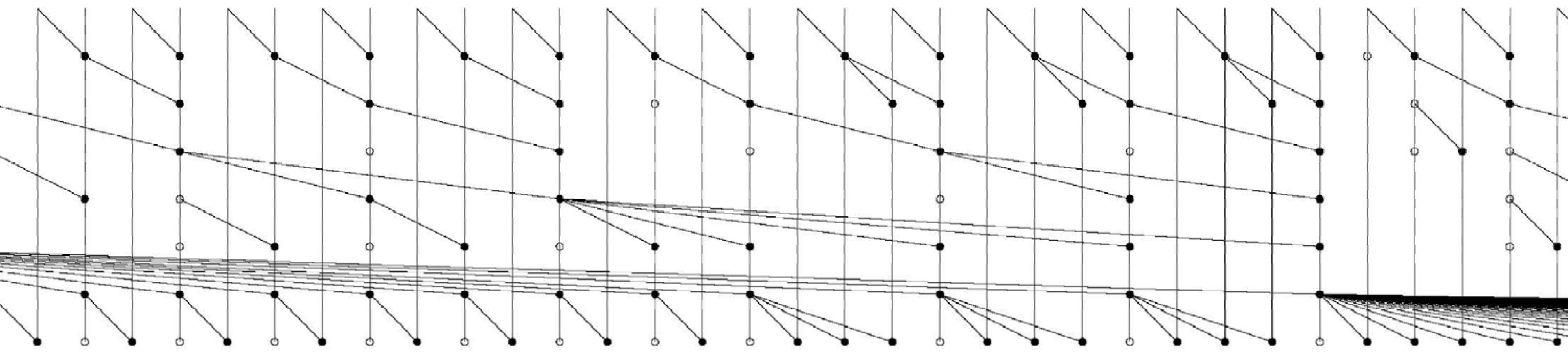
```
type Context = ([Int],Int)
```



```
parpre3 :: Int -> Int -> ([Net] -> Int) -> Context -> ANW
parpre3 f g opt ctx = maybe (error "no fit") unWrap (prefix ctx)
  where
    prefix = memo pm

    pm ([i],o) = try wire ([i],o)
    pm (is,o) | 2^(maxd is o) < length is = Nothing
    pm (is,o) | fits bser (is,o) = Just (Wrap bser)
    pm (is,o) = bestOn is opt $ mapMaybe makeNet (tops3 permsUp (is,o) f g)
      where
        makeNet ds = do let sis = split ds is
                        let js = map (last.(ser delF)) $ init sis
                        pr <- prefix' $ last sis
                        p  <- prefix' js
                        return $ build1 ds pr p
        prefix' ins = prefix (ins,o-1)
```

```
parpre3 :: Int -> Int -> ([Net] -> Int) -> Context -> ANW
parpre3 f g opt ctx = maybe (error "no fit") unWrap (prefix ctx)
  where
    prefix = memo pm

    pm ([i],o) = try wire ([i],o)
    pm (is,o) | 2^(maxd is o) < length is = Nothing
    pm (is,o) | fits bser (is,o) = Just (Wrap bser)
    pm (is,o) = bestOn is opt $ mapMaybe makeNet (tops3 permsUp (is,o) f g)
      where
        makeNet ds = do let sis = split ds is
                        let js = map (last.(adladF delF)) $ init sis
                        pr <- prefix' $ last sis
                        p  <- prefix' js
                        return $ build1 ds pr p
        prefix' ins = prefix (ins,o-1)
```

(NSI)



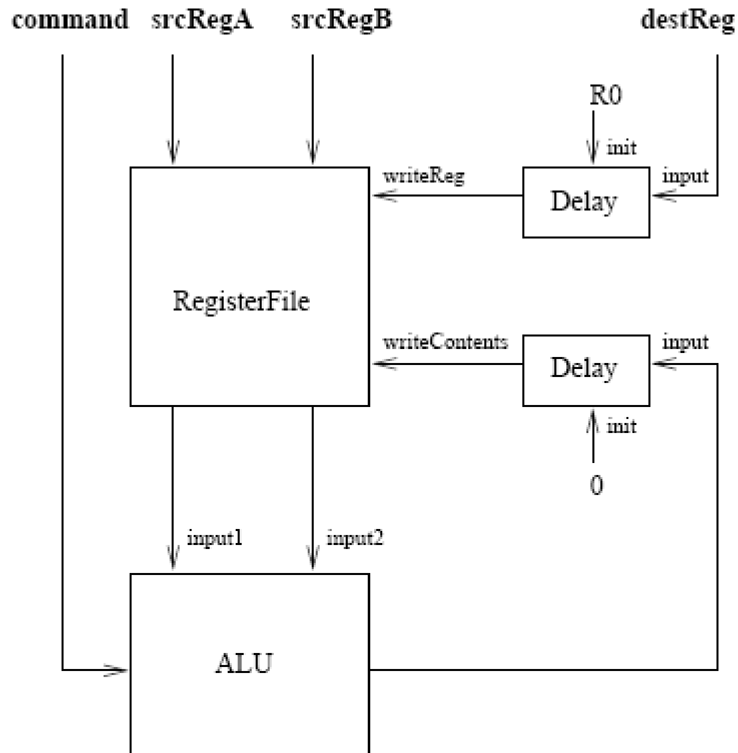
Hawk

```
type Hawk a = [a]            -- (called Signal a in Hawk papers)

constant :: a -> Hawk a
constant x = x : constant x

lift :: (a -> b) -> Hawk a -> Hawk b
lift f (x : xs) = f x : lift f xs

delay :: a -> Hawk a -> Hawk a
delay x xs = x : xs
```

Hawk

```
mux  :: Hawk Bool -> Hawk a -> Hawk a -> Hawk a
mux (c:cs) (t:ts) (e:es) =  x : mux cs ts es
    where
        x = if c then t else e
```

Hawk

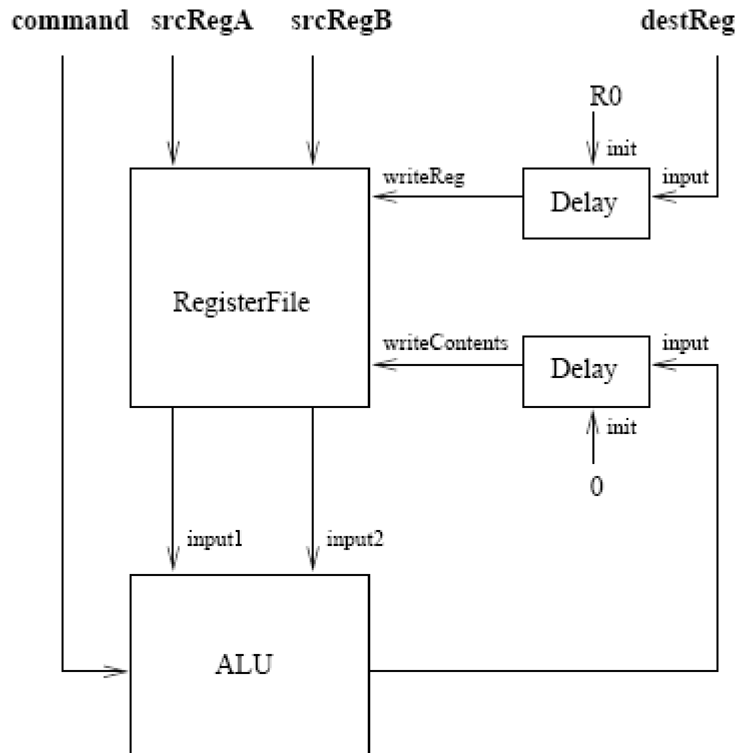


data Reg = R0 | R1 | R2 … |R7

data Cmd = Add | SUB | INC

diagram from ”Microprocessor Specification in Hawk”

Hawk




```
sham1 :: (Signal Cmd,Signal Reg,
          Signal Reg,Signal Reg) ->
         (Signal Reg,Signal Int)

sham1 (cmd,destReg,srcRegA,srcRegB) =
  (destReg',aluOutput')
  where
    (aluInputA,aluInputB) =
      regFile (destReg',aluOutput')
              srcRegA srcRegB
    aluOutput = alu cmd aluInputA aluInputB
    aluOutput' = delay 0 aluOutput
    destReg' = delay R0 destReg
```

diagram and code from ”Microprocessor Specification in Hawk”

Hawk

- Now you just have the whole of Haskell
- Beautiful descriptions e.g. using transactions
  to ease description of control parts
- implementation a bit clunky to use (the one time I tried)
- Problem is that it is hard to do anything with these descriptions (though there was some work on verification with Isabelle).

Chalk

Aim to get the best of both worlds

Use ”modern functional programming”

- GADTs
- Applicative Functors
- etc.

Chalk

pure :: a -> Circuit a

Examples

zero = pure False

invertor = pure not

adder = pure (+)

Chalk

```
<*> :: Circuit (a -> b) -> Circuit a -> Circuit b


mux  :: Circuit Bool -> Circuit a -> Circuit a -> Circuit a
mux cs ts es = pure cond <*> cs <*> ts <*> es
  where
    cond c t e = if c then t else e

delay :: a -> Circuit a -> Circuit a

component :: String -> Circuit a -> Circuit a

loop :: Circuit (s -> (a,s)) -> s -> Circuit a
```

Generalised Abstract Data Type (GADT)

Sort of poor man's dependent types (!)

now commonly used in DSELs. One nice example is use in Yampa (functional reactive programming) to permit more optimisations

```
data Term a where
   Lit :: Int -> Term Int
   Succ :: Term Int -> Term Int
   IsZero :: Term Int -> Term Bool
   If :: Term Bool -> Term a -> Term a -> Term a
   Pair :: Term a -> Term b -> Term (a,b)
```

Generalised Abstract Data Type (GADT)

```
eval :: Term a -> a
eval (Lit i) = i
eval (Succ  t) = 1 + eval t
eval (IsZero t) = eval t == 0
eval (If b e1 e2) = if eval b then eval e1 else eval e2
eval (Pair e1 e2) = (eval e1, eval e2)
```

- The key point about GADTs is that *pattern matching causes type refinement*. For example, in the right hand side of the equation

```
eval :: Term a -> a
eval (Lit i) = ...
```

the type a is refined to Int. That's the whole point!

(from ghc documentation)

Applicative functor

generalised monad
”somewhere between a monad and an arrow”

```
class Functor f where
   fmap :: (a -> b)-> (f a -> f b)   -- also called <$>

class Functor f => Applicative f where
    pure :: a -> fa
     <*> :: f (a -> b) -> f a -> f b
```

Chalk data type

```
data CircuitF circ a where
  Pure :: a -> CircuitF circ a
  App :: circ (b -> a) -> circ b -> CircuitF circ a
  Delay :: a -> circ a -> CircuitF circ a
  Component :: Name -> circ a -> CircuitF circ a
  Input :: Name -> CircuitF circ a

data Circuit a where
Circuit :: Ref (CircuitF Circuit a) -> Circuit a deriving (Typeable)


instance Applicative Circuit where
   pure x = Circuit (ref x)
   c1 <*> c2 = Circuit (ref (App c1 c2))
```

simulate

```
simulate :: Circuit a -> [a]
simulate (Circuit c) = sim (deref c)
  where
     sim :: CircuitF Circuit a -> [a]
     sim (Pure x) = repeat x
     sim (App f x) = zipWith id (simulate f) (simulate x)
     sim (Delay x xs) = x : simulate xs
     sim (Component nm c) = simulate c
```

Use of GADT necessary to get this to typecheck (App case)

Chalk example

Main point is that it looks nearly as nice as Hawk!

```
data Reg = R0 | R1 | R2 | R3 deriving (Show, Eq)

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

data Cmd = ADD | SUB | INC deriving (Show, Eq)

type Operand = (Reg, Maybe Int)
data Transaction =
  Transaction {dest :: Operand, cmd :: Cmd, src :: [Operand]}
  deriving (Show, Eq, Typeable)

setDest :: Transaction -> Int -> Transaction
setDest (Transaction (r,_) cmd srcs) i = Transaction (r, Just i) cmd srcs
```

Chalk Examples

```
regFile :: Signal Transaction -> Signal Transaction -> Signal Transaction
regFile writes reads = loop (regStep <$> writes <*> reads) initRegs


regStep :: Transaction -> Transaction -> Regs -> (Transaction , Regs)
regStep write@(Transaction wrOp _ _) read regs
  = let regs' = updateReg wrOp regs
        read' = updateTransaction regs read
    in (read' , regs')

updateTransaction :: Regs -> Transaction -> Transaction
updateTransaction regs t = t {srcs = map (updateOperand regs) (srcs t)}

updateOperand regs (r, _ ) = (r , Just (lookupReg r regs))
lookupReg R0 (a,b,c,d) = a
lookupReg R1 (a,b,c,d) = b
lookupReg R2 (a,b,c,d) = c
lookupReg R3 (a,b,c,d) = d
```

Chalk example

```
alu :: Signal Transaction -> Signal Transaction
alu cmds = interpret <$> cmds
  where
  interpret :: Transaction -> Transaction
  interpret trans@(Transaction dest cmd srcs) =
    setDest trans (eval cmd (map (fromJust . snd) srcs))
  eval :: Cmd -> [Int] -> Int
  eval ADD [x, y] = x + y
  eval SUB [x, y] = x - y
  eval INC [x] = x + 1

sham :: Signal Transaction -> Signal Transaction
sham instrs = aluOutputD
  where
      aluInput = regFile aluOutputD instrs
      aluOutput = alu aluInput
      aluOutputD = delay nop aluOutput
```

Analysis?

Applicative functor also makes non-standard interpretation (NSI) straight-forward e.g. estimating costs

```
data Ticked a = T {val :: a, cost :: Double}

typed Tcircuit a = Circuit (Ticked a)

instance Functor Ticked where
    fmap f (T x cost) = T (f x) cost

instance Applicative Ticked where
   pure x = T x 0.0
   (T f c1) <*> (T x c2) = T (f x) (c 1 + c2)
```

Can now specify costs of components and count uses. Could form the basis for more sophisticated analyses.
There is a simple framework there to bring order to manipulations of the circuit data type.

Current state

Have tried various Hawk examples in Chalk

Have developed a series of analyses.

Aiming to mimic analyses from the literature on early power and performance analysis

Also need a way to do refinement. (See Steve Hoover's talk (given by John OL at an earlier DCC.) and Andy Martin's work)

Current state

No major stumbling blocks as yet but larger examples might reveal fundamental limitations!

Aiming for level of abstraction well above the sham examples. The ”uncore” now seems to be a big worry. Example question: What is the effect of this cache organisation on power and performance?

BUT project is stalled because it has no manpower

Current state

No major stumbling blocks as yet but larger examples might reveal fundamental limitations!

Aiming for level of abstraction well above the sham examples. The ”uncore” now seems to be a big worry. Example question: What is the effect of this cache organisation on power and performance?

BUT project is stalled because it has no manpower

and because we are not sure what the right next step is!

Related work on getting the benefits of both deep and shallow embedding

Recipe (Naylor, York)

Layer on top of Lava that provides behavioural programming constructs (mutable vars, parallel and sequential composition etc.)

based on Claessen and Pace (Flash)

also used to control Lego Mindstorms!

Feldspar (Axelsson, Sheeran, Svenningsson et al)

(DSEL for DSP algorithm design)

Discussion

- We need to decide where to go next
- Experimenting with what we can do in very high level architectural modelling and refinement with fancy types and other modern PL goodies is fun. But is it wise?
- Would it be worthwhile making a joint effort in high level architectural modelling?? The work would have to be done in collaboration with industrial colleagues (Intel, …).