Parallel Functional Array Programming

Gabrielle Keller
UNSW

July 12, 2018

1 Parallel Functional Programming

- Parallel Programming
  - performance
- Functional Languages
  - abstraction
  - higher order functions
  -

2 Haskell

3 Deep Embedding

Captures DSL expression as abstract syntax tree, allowing multiple interpretations.

```
data Expr
= Add Expr Expr
| Mult Expr Expr
| Neg Expr
| Const Float

data Expr where
Add   :: Expr -> Expr -> Expr
Mult  :: Expr -> Expr -> Expr
Neg   :: Expr -> Expr
Const :: Foat -> Expr

sampleExpr
 = Mult (Add (Const 1) (Const 3)) (Const 5)
```

```
simplify :: Expr -> Expr
```

4 Generalised Algebraic Data Types (GADTs)

```
data Expr a where
    Add :: Expr Float -> Expr Float -> Expr Float
    Mult :: Expr Float -> Expr Float -> Expr Float
    Neg :: Expr Float -> Expr Float
    Less :: Expr Float -> Expr Float -> Expr Bool
    Const :: a -> Expr a
    If :: Expr Bool -> Expr a -> Expr a -> Expr a

eval :: Expr a -> a
eval (Const c) = c
eval (If cond e1 e2) = if (eval cond)
                         eval e1
                       else
                         eval e2
```

5 Accelerate

Dot Products in Haskell

```
import Prelude
dotp :: Num a => [a] -> [a] -> a
dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
foldl :: (b -> a -> b) -> b -> [a] -> b
```

Dot product on vectors:

```
import Data, Vector, Unboxed
dotp :: (Num a, Unbox a) => Vector a -> Vector a -> a
dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

Dot product on vectors using Accelerate:

```
import Data, Array, Accelerate
dotp :: (num a, Elt a) => Acc (Vector a)
        -> Acc (Vector a) -> Acc (Scalar a)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)

zipWith : (Elt a, Elt b, Elt c)
=> (Exp a -> Exp b -> Exp c)
```