Finding Functional Pearls

Detecting Recursion Schemes in Haskell Functions via Anti-Unification

Adam D. Barwell, Christopher Brown, and Kevin Hammond
University of St Andrews

Email: adb23@st-andrews.ac.uk

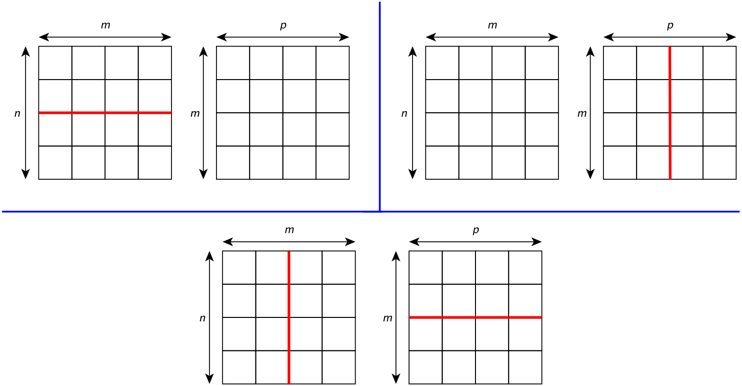
Lambda Days
9 February 2017

Parallelism

- Parallel devices are ubiquitous
  - Phones, tablets, laptops, &c. are all multicore
  - Heterogeneity

By Béria L. Rodríguez, CC BY-SA 3.0, Wikipedia

Matrix Multiplication



Matrix Multiplication

```haskell
type Matrix = [[a]]
data Action = DHL | DVR | DB
data Tree = Leaf Matrix Matrix
          | Node Action Tree Tree

matmult :: Matrix -> Matrix -> Matrix
matmult a b = (join . split) a b
```

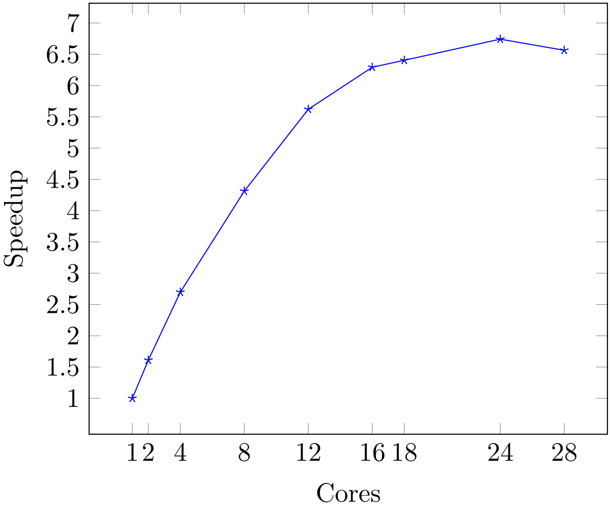
Matrix Multiplication

```
join :: Tree -> Matrix
join t = foldTree multiply h t
  where
    h :: Action -> Matrix -> Matrix -> Matrix
    h DHL a b = a ++ b
    h DVR a b = zipWith (++) a b
    h DB  a b = sum’ a b
```

Matrix Multiplication

```
parChunkTree :: Int
             -> (Matrix -> Matrix -> Matrix)
             -> (Action -> Matrix -> Matrix -> Matrix)
             -> Strategy (Either Tree Matrix)
parChunkTree d f g (Leaf a b) = do
  m' <- rpar (h a b)
  return (Right m')
parChunkTree 0 f g (Node c l r) = do
  (Right a) <- evalFoldTree f g l
  (Right b) <- evalFoldTree f g r
  m <- rdeepseq (g c a b)
  return (Right m)
parChunkTree d f g (Node c l r) = do
  (Right a) <- parChunkTree (d-1) f g l
  (Right b) <- parChunkTree (d-1) f g r
  m <- rpar (g c a b)
  return (Right m)
```

Matrix Multiplication



1552x1552 matrices, average of 10 runs.

Alternative Parallelisations

- Adjust depth, size of matrices at leaves, functions `par`'d
- Split the fold into a map & a fold
- Use the Par monad, Eden, &c.
- Call to a GPU (Accelerate)
- Call to a distributed system

The Good

```haskell
join :: Tree -> Matrix
join t = foldTree multiply h t
  where
    h :: Action -> Matrix -> Matrix -> Matrix
    h DHL a b = a ++ b
    h DVR a b = zipWith (++) a b
    h DB  a b = sum' a b
```

- Only need to swap the fold for a parallel version
- Applicable to other recursion schemes
  - *map*, *unfold*, &c.

The Inconvenient

- There may not be a fold to begin with…
- The *spectral* set of Haskell programs in NoFib suite
  - 48 programs of varying design and functionality
  - At least 19 have at least one function that can be rewritten as a *map* or *fold*
- Why?
  - (Left over from) an initial implementation
  - 'No need to define it, I'm only going to use it here.'
  - Don't know of their existence; e.g. *unfold*
  - Near patterns
- Not every recursion scheme is worth parallelising, but if they're there, we can pick the relevant ones

Anti-Unification

- First described by Plotkin and Reynolds in 1970
- Primarily used in clone detection & elimination
- Finds the *least general generalisation* of two terms

$$t_1 = a + (b - c)$$
$$t_2 = 5 * (b + c)$$

$$t = \alpha \ \beta \ (b \ \gamma \ c)$$

Applying Anti-Unification to Matrix Multiplication

```
join :: Tree -> Matrix
join t = foldTree multiply h t
  where
    h :: Action -> Matrix -> Matrix -> Matrix
    h DHL a b = a ++ b
    h DVR a b = zipWith (++) a b
    h DB  a b = sum’ a b
```

Applying Anti-Unification to Matrix Multiplication

```
join :: Tree -> Matrix
join (Leaf a b)   = multiply a b
join (Node x a b) = h x (join a) (join b)
  where
    h :: Action -> Matrix -> Matrix -> Matrix
    h DHL a b = a ++ b
    h DVR a b = zipWith (++) a b
    h DB  a b = sum' a b
```

Applying Anti-Unification to Matrix Multiplication

```
foldTree :: (Matrix -> Matrix -> Matrix)
         -> (Action -> Matrix -> Matrix -> Matrix)
         -> Tree
         -> Matrix
foldTree f g (Leaf a b) = f a b
foldTree f g (Node a l r) =
  g a (foldTree f g l) (foldTree f g r)
```

Applying Anti-Unification to Matrix Multiplication

```haskell
foldTree :: (Matrix -> Matrix -> Matrix)
         -> (Action -> Matrix -> Matrix -> Matrix)
         -> Tree
         -> Matrix
foldTree f g (Leaf a b) = f a b
foldTree f g (Node a l r) =
  g a (foldTree f g l) (foldTree f g r)

join :: Tree -> Matrix
join (Leaf a b)   = multiply a b
join (Node x a b) = h x (join a) (join b)
  where
    h :: Action -> Matrix -> Matrix -> Matrix
    h DHL a b = a ++ b
    h DVR a b = zipWith (++) a b
    h DB  a b = sum' a b
```

Applying Anti-Unification to Matrix Multiplication

```
au f g (Leaf a b)   = f a b
au f g (Node a l r) = g a (x l) (y r)
```

Applying Anti-Unification to Matrix Multiplication

```
join t = au multiply h t
  where
    h :: Action -> Matrix -> Matrix -> Matrix
    h DHL a b = a ++ b
    h DVR a b = zipWith (++) a b
    h DB  a b = sum’ a b

treeFold f g t = au f g t
```

Applying Anti-Unification to Matrix Multiplication

```
foldTree :: (Matrix -> Matrix -> Matrix)
         -> (Action -> Matrix -> Matrix -> Matrix)
         -> Tree
         -> Matrix
foldTree f g (Leaf a b) = f a b
foldTree f g (Node a l r) =
  g a (foldTree f g l) (foldTree f g r)
```

```
au f g (Leaf a b)   = f a b
au f g (Node a l r) = g a (x l) (y r)
```

Applying Anti-Unification to Matrix Multiplication

```
join :: Tree -> Matrix
join t = foldTree multiply h t
  where
    h :: Action -> Matrix -> Matrix -> Matrix
    h DHL a b = a ++ b
    h DVR a b = zipWith (++) a b
    h DB  a b = sum' a b
```

Not Just Matrix Multiplication

- Implemented a prototype of our approach in HaRe
- Applied our prototype to a range of functions inspired by the Haskell prelude
- Also to functions in Matrix Multiplication, N-Body, and Quicksort

Future Work

- More examples
  - NoFib
  - Real Haskell programs
- More patterns
  - Currently working on unfold
- Use equational reasoning, reduction, rewriting, &c. to make pattern discovery and argument derivation more flexible

Summary

- Use anti-unification to automatically discover recursion schemes in Haskell code
- Prototype of our approach implemented in HaRe
- Recursion schemes can be used as a 'stepping stone' for parallelisation
- Parallelisation becomes as simple as swapping sequential patterns for parallel ones.

adb23@st-andrews.ac.uk @rephrase_eu