First Order Haskell

Neil Mitchell

York University

www.cs.york.ac.uk/~ndm

First order vs Higher order

- Higher order: functions are values
  - Can be passed around
  - Stored in data structure
- Harder for reasoning and analysis
  - More syntactic forms
  - Extra work for the analysis
- Can we convert automatically?
  - Yes (that’s this talk!)

Which are higher order?

[not x | x ← xs]

let xs = x:xs in xs

putChar 'a'

foldl (+) 0 xs

\x → not x

map not xs

1

not . odd

(+1)

a < b

not $ odd x

const 'N'

Higher order features

- Type classes are implemented as dictionaries
  - (==) :: Eq a ⇒ a → a → Bool
  - (==) :: (a→a→Bool, a→a→Bool) → a → a → Bool
- Monads are higher order
  - (>>=) :: Monad m ⇒ m a → (a → m b) → m b
- IO is higher order
  - newtype IO a = IO (World → (World, a))

A map example

```
map f []     = []
map f (x:xs) = f x : map f xs
heads xs     = map head xs
```

- head is passed higher order
- map takes a higher order argument
- heads *could* be first order

John C. Reynolds, Definitional Interpreters for Higher-Order Programming Languages

Reynold's Style Defunctionalisation

```
data Func          = Head
apply Head x       = head x
map f []           = []
map f (x:xs)       = apply f x : map f xs
heads xs           = map Head xs
```

- Move functions to data

Reynold's Style Defunctionalisation

- Good
  - Complete, works on all programs
  - Easy to implement
- Bad
  - No longer Hindley-Milner type correct
  - Makes the code more complex
  - Adds a level of indirection
  - Makes program analysis harder

Specialisation

```
map_head []     = []
map_head (x:xs) = head x : map_head xs
heads xs        = map_head xs
```

- Move functions to code

Specialisation

- Find: map head xs
  - A call to a function (i.e. map)
  - With an argument which is higher order (i.e. head)
- Generate: map_head xs = …
  - A new version of the function
  - With the higher order element frozen in
- Replace: map_head xs
  - Use the specialised version

Specialisation fails

```
(.) f g x  = f (g x)
even       = (.) not odd
check x = even x
```

- Nothing available to specialise!
- Can be solved by a simple inline

```
check x = (.) not odd x
```

An algorithm

1. Specialise as long as possible
2. Inline once
3. Goto 1

- Stop when no higher order functions remain

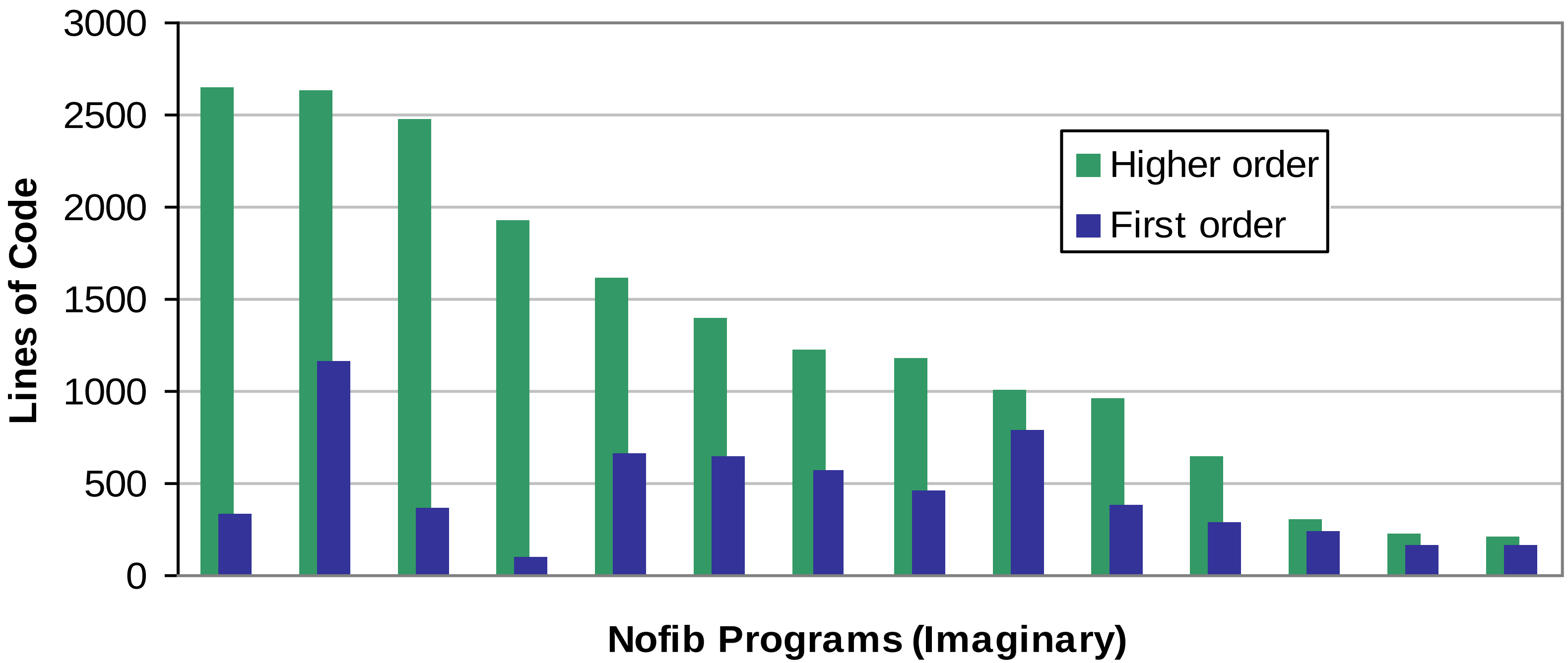
Algorithm fails

```
data Wrap a = Wrap (Wrap a) | Value a
f x         = f (Wrap x)
check       = f (Value head)
```

- In practice, this is rare – requires a function to be stored in a recursive data structure and …
- Detect, and revert to Reynold's method

Code Size

- Specialisation approach *reduces* code volume
  - Average about 55% smaller code (20%-95% range)



Current uses

- Performance optimiser
  - The first step, makes the remaining analysis simpler
  - Already increases the performance
- Analysis tool
  - Catch, checking for pattern match safety
  - Keeps the analysis simpler
- Implemented for Yhc (York Haskell Compiler)

Conclusion

- Higher order functions are good for programmers
- Analysis and transformation are simpler in a first order language
- Higher order functions can be removed
- Their removal can reduce code size