The Haskell Motorcycle Diaries: A Crash Course in Haskell for Games

Matthew Sackman and Tristan Oliver Richard Allwood
matthew@wellquite.org, tora@zonetora.co.uk

Haskell

- Functional programming language
- Lazy
- Referentially transparent right down to I/O
- Industrial strength optimising super-dooper compiler (GHC) paid for by Microsoft!
- Robust interface to C libraries and native code
- OpenGL bindings in standard libraries (unlike Java!)

```
import Data.Char

data Tree a = Empty | Node (Tree a) a (Tree a)
              deriving (Show)

insert :: (Ord a) => a -> Tree a -> Tree a
insert v Empty = Node Empty v Empty
insert v orig@(Node left v' right)
    | v < v' = Node (insert v left) v' right
    | v > v' = Node left v' (insert v right)
    | otherwise = orig
```

```
data Tree a = Empty | Node (Tree a) a (Tree a)
              deriving (Show)

insert :: (Ord a) => a -> Tree a -> Tree a
...

instance Functor Tree where
    fmap _ Empty = Empty
    fmap f (Node left v right)
        = Node (fmap f left) (f v) (fmap f right)

myTree :: Tree Char
myTree = foldr insert Empty "helloWorld"

intTree :: Tree Int
intTree = fmap ord myTree
```

```
isIn :: (Ord a) => a -> Tree a -> Bool
isIn _ Empty = False
isIn v (Node left v' right)
    | v < v' = isIn v left
    | v > v' = isIn v right
    | otherwise = True

infTree :: Tree Int
infTree = Node (fl (-1)) 0 (fr (1))
  where
    fl n = Node (fl (pred n)) n Empty
    fr n = Node Empty n (fr (succ n))
```

```
main :: IO ()
main = do putStrLn . show $ myTree
          putStrLn . show $ intTree
          putStrLn . show . fmap (chr . (+1)) $ intTree
          putStrLn . show . isIn 101 $ intTree
          putStrLn . take 1000 . show $ infTree
          putStrLn . show . isIn 1031 $ infTree
```

```
*Main> main
Node (Node Empty 'W' Empty) 'd' (Node (Node Empty 'e' (Node
Node (Node Empty 87 Empty) 100 (Node (Node Empty 101 (Node H
Node (Node Empty 'X' Empty) 'e' (Node (Node Empty 'f' (Node
True
Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (
True
```

Reasons to use Haskell for games

- Higher-order functions: aids reuse and leads to very concise code
- Very rich type system: requires much better discipline from the programmer than with C++/Java
- Laziness creates possibility of optimisations that are difficult to achieve in non-lazy languages
- No pointer arithmetic, no null pointer exceptions, segfaults etc etc
- Much easier to reason about and be able to understand the effect of code at first glance
- Very easy to link and wrap native code

On the flip side

- Optimising Haskell can be done and can result in really fast code, but sometimes at the cost of readability
- Garbage collection issues, though much less pronounced than Java
- Profiling and debugging tools are not as mature and featureful as for other languages
- Learning curve: Type Classes, Monads, GADTs, Functional Dependencies, Phantom Types etc: a rich academic playground!
- Tends to distract from doing a PhD…

On the flip side

- Optimising Haskell can be done and can result in really fast code, but sometimes at the cost of readability
- Garbage collection issues, though much less pronounced than Java
- Profiling and debugging tools are not as mature and featureful as for other languages
- Learning curve: Type Classes, Monads, GADTs, Functional Dependencies, Phantom Types etc: a rich academic playground!
- Tends to distract from doing a PhD…
- And now the game!