Haskell for Erlangers

(c) Simon Thompson
University of Kent, 2015
Rationale
Erlang … you know.

Haskell … this week.

- Miranda, ML, OCaml, F#, …
- Strongly-typed, rich type languages, …
- LISP, scheme: weakly typed, macros, eval …
Functional languages

• If by that you mean including *lambdas*
  
• Java
• JavaScript
• Ruby
• C++
• ...

Why learn Haskell?

- A different perspective … change the way you write Erlang (or Java or …).
- Different tools for different jobs.
  - Transformation / language processing.
  - DSLs.
- It’s fun!
Non-strict, purely-functional languages, such as Haskell, are perceived to be inadequate for everyday, get-the-job-done tasks; in particular, they are seen to be "bad at I/O". Consequently, an informal working group has been designing an extended variant of Haskell to address these requirements …

The Perl language is nothing if not "good for everyday, get-the-job-done" tasks - it puts UNIX at the programmer's fingertips. … What follows is an informal note about what we call the "Haskerl" extension to Haskell …

http://www.dcs.gla.ac.uk/~partain/haskerl/partain-1.html
Immutability
Immutability

- Objects whose state doesn’t change …
- … if you want a different object, create one.
- Objects $\approx$ Values in functional languages.
Immutability

- Java theory and practice: To mutate or not to mutate? Immutable objects can greatly simplify your life
- Brian Goetz, Principal Consultant, Quiotix Corp
Immutability

• They can only be in one state, so as long as they are properly constructed … never get into an inconsistent state.

• You can freely share and cache references to immutable objects without having to copy or clone them; you can cache their fields … without worrying about the values becoming stale or inconsistent with the rest of the object's state.

• They are inherently thread-safe, so you don't have to synchronize access to them across threads.
Inefficient?

- Compare with garbage collection …
- … gain from the lack of a whole class of errors.
Implementing functional languages

• A functional implementation can share references to the same object, so no need for copy to support mutation.

• On “update” copy only the part of the structure that is affected …

• … smart data structure design can minimise this.
Erlang recap
Weakly typed

- Numbers, atoms, tuples and lists.
- (Extensible) records: syntactic sugar.
- Dynamic aspects.

Val = [12,"34",[56],{{78}}].

NewTree =
    Tree#tree{value=42}.

F = list_to Atom("blah"),
    apply(?MODULE,F,Args).
Concurrency at the core

- Processes.
- No shared memory.
- Asynchronous message passing.
- Process ids or names.

```erlang
Pid = spawn(server,fac,[]),
Pid ! {self(),N},
receive
  {ok,Result} -> ...
  stopped        -> ...
end, ...

fac() ->
  receive
    {From, stop} ->
      From ! stopped;
    {From, N} ->
      From ! {ok,fact(N)},
      fac()
  end.
```
Pattern Matching

- Haskell-style, but ...
- Single assignment.
- Bound variables can appear in patterns.
- Selective receive.

\[
\begin{align*}
N &= 46, \\
N &= 23 + 23, \\
N &= 35, \\
&\ldots \\
\text{receiveFrom}(\text{Pid}) &\rightarrow \\
\text{receive} &\rightarrow \\
\{\text{Pid}, \text{Payload}\} &\rightarrow \ldots \\
&\ldots \rightarrow \ldots \\
&\text{end.} \\
\text{receive} &\{\text{foo}, \text{Foo}\} \rightarrow \ldots \text{end}, \\
\text{receive} &\{\text{bar}, \text{Bar}\} \rightarrow \ldots \text{end} \\
\end{align*}
\]
Open Telecom Platform

- Erlang + OTP.
- Design patterns.
- Generic behaviours.
- Server, FSM, event handler, supervisor.
- Callback interface.

```erlang
init(FreqList) ->
    Freqs = [FreqList, []],
    {ok, Freqs}.

terminate(_, _) -> ok.

handle_cast(stop, Freqs) ->
    {stop, normal, Freqs}.

handle_call(allocate, From, Freqs) ->
    {NewFreqs, Reply} =
        allocate(Freqs, From),
    {reply, Reply, NewFreqs}.
```
Other Erlang features

- Eager evaluation.
- Side effects.
- Name/arity identify a function.
- Bindings: shadows, multiple BOs.
- Macros.
Pragmatics

• One implementation, one standard.
• Well-defined, controlled release cycle.
• Open Source but … Ericsson effort.
• Erlang Extension Proposals.
Haskell for Erlang
Strongly typed

- Built-in types.
- User-defined types
- Most general types, at compile time.
- Polymorphism and overloading.
- Higher types, kinds.

```haskell
type String = [Char]

data Tree a =
  Leaf a |
  Node (Tree a) (Tree a)

sort :: (Ord a) => [a] -> [a]

:type <any-expression>
```
Laziness at the core

- Language is pure: no side-effects.
- Evaluation is lazy.
- Only evaluate when a value is needed ...
- ... and only to the extent that’s needed.

```haskell
ifThenElse :: Bool -> a -> a
ifThenElse True x y  = x
ifThenElse False x y = y

replicate :: Int -> a -> [a]
replicate n x
    = take n (repeat x)

repeat x
    = xs
    where
        xs = x : xs
```
Pattern Matching

- Erlang-style, but ...
- It’s not assignment.
- Bound variables can’t appear in patterns.
- No repeated variables in patterns.

N = 46,
N = 23+23,
N = 35,
...

booksBorrowed pers dbase
  = [ bk |
      (pers,bk) <- dbase ]

booksBorrowed pers dbase
  = [ bk |
      (p,bk) <- dbase,
      p==pers ]
Controlled side-effects

- Monads: ADT for side-effecting computations.
- \( m \ a = \) computations returning value of type \( a \)
- do notation: syntactic sugar for clarity.

```haskell
goUntilEmpty :: IO ()
goUntilEmpty
  = do line <- getLine
       if (line == [])
          then return ()
          else (do putStrLn line
                     goUntilEmpty)

sumTree :: Tree Int -> Id Int
sumTree Nil = return 0
sumTree (Node n t1 t2)
  = do num <- return n
       s1  <- sumTree t1
       s2  <- sumTree t2
       return (num + s1 + s2)
```
Other Haskell features

- Overloading and type classes.
- Local definitions.
- Module system more complex than Erlang.
- No macros (but there is Template Haskell).
- Language of choice for DSLs.
Pragmatics

- GHC predominates, others exist.
- Standards: Haskell 2010, ... cf GHC.
- Haskell Platform: controlled releases.
- HackageDB and Cabal: 3000+ contributed Open Source packages.
- No stable production quality GUI lib.
GHCi and the Haskell Platform
The Haskell Platform

• The latest version of the compiler GHC, the “shell” version GHCi, and various standard libraries.

• Download the platform

http://www.haskell.org/platform/
**ghci commands**

- **expression**: Evaluate expression
- **:type expr**: Give the most general type of expr
- **:load Foo**: Load and compile the module Foo
- **:reload**: Reload the last module loaded
- **:help**: Give help on the ghci commands
- **:quit**: Quit
The unit of compilation is a module.

Demo lives in Demo.hs

By default everything is exported.

Can hide on import.

Can import qualified: name thus: Demo.bar.
The basics of Haskell
Function application

- In Erlang: traditional function application
  \[ \text{iff}(\text{true}, \text{false}) \]

- In Haskell: uses juxtaposition, just put the arguments after the function, separated by white space
  \[ \text{iff True False} \]
Type declarations

• The type declaration is optional.

• `:type iff` in GHCi will tell you the most general type.

```haskell
exOr :: Bool -> Bool -> Bool
exOr True y  = not y
exOr False y = y

iff x y = not (x `exOr` y)
```
Characters and strings

- Characters: `Char`.
- type `String = [Char]`
- `putStr` is part of the IO system using the IO monad.
- `show` and `read` are overloaded ...

```haskell
'a',..., '0',..., 'Z' :: Char
'\n','\',','\"','\t' :: Char
fromEnum :: Char -> Int
toEnum    :: Int -> Char
"string" :: String
putStr :: String -> IO ()
show :: a -> String
read :: String -> a
```
Guards

- Switch between different alternatives using guards.
- Guard can be any Boolean expression.
- Erlang: compare with when

```haskell
max :: Int -> Int -> Int

max x y |
| x=x y = x
| x<y = y

max' x y |
| x=x y = x
| otherwise = y

max'' x y |
| x>y = x

max''' x y
= y
```
Definitions can be local: where and let.

wheres are local to function equations.

let definitions are local to expressions.

Size of Haskell … .
The first character of a definition opens up a box …

… which is closed only when something below or to the left.

“Offside rule”

\[ \text{mystery } x = x^x + x + 2 \]

next \( x = \ldots \)
• In Emacs with Haskell mode, repeated tabbing will take you through various sensible layout options.

```
mystery x y ...
  | guard1    = result1
  | guard2    = result2
  |    ...
  |    ...
where
  local1 = ...
  ...
  local2 ... = ...
```
Types: tuples and lists
Tuples

• Tuples enclosed in parentheses: (...,...,...)

• Heterogeneous.

• Access by pattern matching (...,...,...).

• Erlang compare with {...,...,...}

addPair :: (Int,Int) -> Int
addPair (n,m) = n+m

type Person = (String,Int)
showPers :: Person -> String
showPers (name,age) = name ++ show age
Lists

• Lists in square brackets: \([..., ..., ...]\)

• Access by pattern matching over the constructor \((x:xs)\).

• Homogeneous.

• Static typing still OK.

addLst :: [Int] -> Int
addLst [] = 0
addLst (n:l) = n + addLst l

add2elem :: [Int] -> Int
add2elem [n,m] = n+m

-- what do these do?
puzzle [n:l] = n + puzzle l
puzzle’ [n:l] = n+1
Defining data types
• Enumerated type with three elements.

• Plus a bit of type class magic (later).

• Definitions by pattern matching.

```haskell
data Move
    = Rock | Paper | Scissors
    deriving (Show, Eq)

beat :: Move -> Move
beat Rock     = Paper
beat Paper    = Scissors
beat Scissors = Rock

outcome :: Move -> Move -> Int
outcome Rock Rock     = 0
outcome Rock Paper    = -1
outcome Rock Scissors = 1
...
Elements of the `People` type are of the form

```
Person n a
```

where `n` is a `String` and `a` an `Int`.

```haskell
type Name = String
type Age  = Int

data People = Person Name Age
             deriving (Eq,Show)

Person "Ronnie" 14
Person "Simon" 44

showPerson :: People -> String
showPerson (Person n a) = n ++ " -- " ++ show a
```
Terminology

- **Person** is a constructor used to build elements.
- **Person** is a function.
- Constructors begin with capitals.
- **Erlang:** compare with `{person, Name, Age}`
Compare

• Compare product types with tuples.

```haskell
data People
  = Person Name Age
  deriving (Eq,Show)

type People
  = (Name, Age)
```
• Different alternatives, built by the different constructors.

• Incredibly useful for modelling: usually things come in a number of forms.

data Shape =
    Circle Float |
    Rect Float Float
deriving (Eq, Show, Ord, Read)

isRound :: Shape -> Bool
isRound (Circle _) = True
isRound (Rect _ _) = False

area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rect h w) = h * w
Questions

• Define a function to give the perimeter of a shape.

• Add triangles to the type and the function definitions.

• Compare with Java?

data Shape =
  Circle Float |
  Rect Float Float
 deriving (Eq, Show)

isRound :: Shape -> Bool
isRound (Circle _) = True
isRound (Rect _ _) = False

area :: Shape -> Float
area (Circle r) = pi*r*r
area (Rect h w) = h*w
Syntax ... ( )
Parentheses

- Tuples **must be** constructed like this: 
  \((..., ..., ...)\)

- Operators as functions, 
  \((\&\&).\)

- Operator sections, 
  
  \(1+.\), \(\text{`rem`2).}\)

  \((\&\&) \text{ True False} \rightarrow \text{ False}\)

  \(\text{map (1+)} [2,3] \rightarrow [3,4]\)

  \(\text{filter ((/=0).(`rem`2)) [1..9]}\)

  \(\rightarrow [1,3,5,7,9]\)
Parentheses

- Grouping: in \textit{deriving}, contexts, ...

- Parsing
  - Pattern matching constructor applications.
  - General expressions
  - Type annotations

\begin{align*}
\cdots & \text{deriving \ (Eq, Show)} \\
\cdots & \text{(Eq } a, \text{ Show } a) \Rightarrow a \rightarrow \text{Int} \\
\text{sum (Node } t1 \ t2) &= \cdots \\
\text{sum (x:xs)} &= \cdots \\
4-(3-2) &= \cdots \\
\text{foldr (*) (1::Integer)} &\cdots \text{[1..1000]} \\
\end{align*}
Lazy evaluation
Lazy evaluation

- Evaluate arguments only when their values are needed.

```plaintext
ite :: Bool -> a -> a -> a

ite True  x y = x
ite False x y = y

let undef=undef::Int in
  ite True 2 undef
  --> 2
```
Lazy evaluation

- Evaluate arguments only as much as needed for computation to continue.
- Coroutines …

\[
\text{repeat} :: a \rightarrow [a] \\
\text{repeat } x \\
\quad = \ xs \\
\quad \text{where} \\
\quad \hspace{1em} xs = x : xs
\]

\[
\text{replicate} :: \text{Int} \rightarrow a \rightarrow [a] \\
\text{replicate } n \ x \\
\quad = \ \text{take } n \ (\text{repeat } x)
\]

\[
\text{take} :: \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{take } 0 \ x = [] \\
\text{take } n \ (x:xs) \\
\quad = \ x : \ \text{take } (n-1) \ xs
\]
Sieve

primes = sieve [2..]
sieve (x:xs) = x : sieve [ y | y<-xs, y `rem` x /= 0]

• Sieve of Eratosthenes.
• Generate as many primes as you want … .
Avoiding delay

- **sumI** creates a large sum expr, only evaluated at the end.

- So does **sumIA**!

- Add the annotation $!$ so that **strict** in this argument.

```haskell
sumI n m
  | n>m     = 0
  | otherwise = n + sumI (n+1) m

sumIA n m = accIA n m 0

accIA n m s
  | n>m     = s
  | otherwise = accIA (n+1) m (n+s)

sumIS n m = accIS n m 0

accIS n m s
  | n>m     = s
  | otherwise = accIS (n+1) m $! (n+s)
```
Types: going further
Polymorphism
Some examples

- General question: what **constraints** does the definition put on the type of the function?

```
lengt'h [ ]   = 0
lengt'h (x:xs) = 1 + lengt'h xs

fst (x,_) = x

map f [ ] = [ ]
map f (x:xs) = f x : map f xs

filter p [ ] = [ ]
filter p (x:xs)
  | p x   = x : filter p xs
  | otherwise = filter p xs

twice f x = f (f x)
```
List length

\[
\begin{align*}
\text{length } [] &= 0 \\
\text{length } (x:xs) &= 1 + \text{length } xs
\end{align*}
\]

length is a function \hspace{2cm} \text{result is an Int}

argument is a list \hspace{2cm} \text{no constraint on list elements}

\text{length} :: [a] -> \text{Int}
First of a pair

\[ \text{fst} \ (x, \_ ) = x \]

\text{fst} \ is \ a \ function \quad \text{result is the 1st element}

\text{argument is a pair} \quad \text{no constraint on 2nd elements}

\text{fst :: (a,b) -> a}
Mapping along a list

\[
\begin{align*}
\text{map } f \; [] & \; = \; [] \\
\text{map } f \; (x:x:s) & \; = \; f \; x \; : \; \text{map } f \; x:s
\end{align*}
\]

map is a function

\( f \) is a function    2nd arg is a list

2nd arg elements have \( f \) applied

result is a list

result elements are results of \( f \)

\[
\text{map} \; :: \; (a \rightarrow b) \rightarrow \; [a] \rightarrow \; [b]
\]
Other examples

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x   = x : filter p xs
  | otherwise = filter p xs

twice :: (a -> a) -> a -> a
twice f x = f (f x)
```
Definitions

• We can define polymorphic types:

  type Strategy a = [a] -> a

• Synonyms (type), e.g. generalised strategy.

  data Tree a
  = Leaf a
  | Node (Tree a) (Tree a)
  deriving ...

• Algebraic types (data)
Questions

- Find the minimum value in such a tree.
- Define trees with data \((a)\) at internal nodes as well.
- How can you use the internal values to \textit{memoise} the minima?

```haskell
data Tree a
    = Leaf a
    | Node (Tree a) (Tree a)
deriving ...
```
Overloading
Element of a list

elem x [] = False
elem x (y:ys) = 
  x==y || elem x ys

elem is a function  result is a Bool

2nd arg is a list  2nd arg elements same type as x

can compare elements x,y::a for equality

elem :: a -> [a] -> Bool
Type classes

• A **class** specifies an interface.

• An **instance** gives an implementation of that interface.

```haskell
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  True == x  = x
  False == x = not x

instance Eq a => Eq [a] where
  [] == []   = True
  [] == _    = False
  _  == []   = False
  (x:xs) == (y:ys)
    = x==y && xs==ys
```
**Element of a list**

\[
\text{elem } x \ [\ ] = \text{False} \\
\text{elem } x \ (y:ys) = \\
\quad x == y \ || \ \text{elem } x \ ys
\]

\text{elem} \ is \ a \ function \quad \text{result is a} \ Bool

2nd \ arg \ is \ a \ list \quad 2nd \ arg \ elements \ same \ type \ as \ x

\text{a \ is \ an \ instance \ of \ the} \ Eq \ type \ class

\text{elem} :: (\text{Eq } a) \Rightarrow a \to [a] \to \text{Bool}
Example: expressions
Expressions

• Integer expressions.

• Aim: want to have **parse** taking a **String** to an **Expr**.

• Exercise: how to add variables to the model?

```haskell
data Expr
  = Lit Int
  | Var Var
  | App Op Expr Expr

data Op = Add | Mul | Sub | ...

eval :: Expr -> Int

eval (Lit n) = n

eval (App Op e1 e2) = evalOp Op (eval e1) (eval e2)

evalOp Add    = (+)
evalOp Mul    = (*)
```
The Parse type

- First attempt: Extract an object of type \( a \) from a String.

Type: \( \text{type Parse } a = \text{String } \rightarrow a \)

Bracket: \( "(234" \rightarrow '(', 
Number: \( "234" \rightarrow 2 \text{ or } 23 \text{ or } 234 \ldots \)
Bracket: \( "234" \rightarrow \text{no result} \)
The Parse type

- Second attempt:
- Extract a collection of objects of type \( a \) from a \( String \).
- Here use list for collection.

```haskell
type Parse a = String -> [a]

bracket "(234" --> ['(']
number "234" --> [2, 23, 234]
bracket "234" --> []
```
The Parse type

• Third attempt:

• Extract a collection of objects of type \(a\) from a String.

• Pair each object with what’s left of the input.

\[
\text{type Parse } a \\
\quad = \text{String} \rightarrow [(a,\text{String})]
\]

\[
\text{bracket ”(234” } \rightarrow [(‘(‘,”234”)]
\]

\[
\text{number ”234” } \rightarrow [(2,”34”), (23,”4”), (234,””)]
\]

\[
\text{bracket ”234” } \rightarrow []
\]
Lazy evaluation
Lazy evaluation

• Evaluate arguments only when their values are needed.

ite :: Bool -> a -> a -> a

ite True  x y = x
ite False x y = y

let undef=undefined::Int in
ite True 2 undef
--> 2
Lazy evaluation

- Evaluate arguments only as much as needed for computation to continue.
- Coroutines ...

```haskell
define repeat :: a -> [a]
define repeat x
  = xs
    where
    xs = x : xs

define replicate :: Int -> a -> [a]
define replicate n x
  = take n (repeat x)

define take :: Int -> [a] -> [a]
define take 0 _ = []
define take n (x:xs)
  = x : take (n-1) xs
```
Lazy “streams”

- Lazy infinite lists look very like “streams” of values flowing along wires in a network.

- Recursion corresponds to a feedback loop in a network.

```haskell
hamming =
    1 : merge
        (map (*2) hamming)
        (map (*3) hamming)

merge (x:xs) (y:ys)
| x<y   = x : merge xs (y:ys)
| x==y  = x : merge xs ys
| x>y   = y : merge (x:xs) ys
```
Higher-order functions
Higher-order functions

- What happens when we apply `ite` to two arguments?
  - We get a function awaiting a string to return a string.
  
  ```haskell
  ite :: Bool -> a -> a -> a
  ite True  x y = x
  ite False x y = y
  ite True "foo"
  ```

- Partial application.
Examples

• Functions as arguments ...

• ... and results.

map :: (a -> b) -> [a] -> [b]

(=='.') :: Char -> Bool

map (=='.')
  :: String -> [Bool]

map (map (=='.'))
  :: [String] -> [[Bool]]
List comprehensions
Generate and test

- Combining mapping and filtering.

```haskell
doubleOdds xs = [ x*2 | x<-xs, odd x ]

odd = (/=0).(\`rem`2)

factors n = [ m | m<[1..n], n `rem` m == 0 ]

perms [] = [[]]
perms xs = [ x:p | x<-xs, p<-perms(xs--[x]) ]
```
Sieve

primes = sieve [2..]
sieve (x:xs) = x : sieve [ y | y<-xs, y `rem` x /= 0]

• Sieve of Eratosthenes.
• Generate as many primes as you want ...
Next … a ‘live’ example
Scenario
Type-driven development

- Define relevant types.
- 2D grid.
- \textbf{True} = empty
- \textbf{False} = occupied

```haskell
type Maze = [[Bool]]
type Point = (Int,Int)
type Path = [Point]
```
Example

```
mazeSt1 :: [String]
mazeSt1
  = ["..#..####",
     ".#..#...#",
     "..#...#.#",
     "##.##.#.#",
     ".##.##..#",
     ".#...#.#",
     ".#.##.##.#",
     ".#..#....#",
     ".##...##.#",
     "...#..##.."
  ]
```

```
makeMaze :: [String] -> Maze
makeMaze lines
  = map (map (=='.')) lines
```
Alternative types

- List (collection of empty points) ...
- ... plus grid size.

```haskell
type Maze = [Point]
type Maze = ([Point], Int, Int)
type Point = (Int, Int)
type Path = [Point]
```
Alternative types

- List of lines ...
- ... each line a list of places ...
- ... and a place is represented by the list of adjacent points.

```haskell
type Maze = [[[Point]]]
type Point = (Int,Int)
type Path = [Point]
```
Type-driven development

- Define relevant types.
- 2D grid.
- True = empty
- False = occupied

```haskell
type Maze = [[Bool]]
type Point = (Int,Int)
type Path = [Point]
```
Example

```haskell
makeMaze :: [String] -> Maze
makeMaze = map (map (=='.'))

mazeSt1 :: [String]
mazeSt1 = ["..#..#####", "#...#...#", ",#...#.#.", "##.##.#.#", ",....#...#",="#.#...#.#", ",#.##.##.#",="#..#....#", ",##...##.#",="#..#..##.."]
```
Type-driven development

- Define types of the main function …
- … and the auxiliary functions needed.

paths :: Maze -> Point -> Point -> [Path]

isPath :: Maze -> Path -> Bool

isEmpty :: Maze -> Point -> Bool

adjPoints :: Maze -> Point -> [Point]
Type-driven development

• Now develop definitions …

• … top-down or bottom-up.

• For top-down use dummy defs … can still type check.

paths :: Maze -> Point -> Point
    -> [Path]

isPath :: Maze -> Path -> Bool

isEmpty :: Maze -> Point -> Bool

adjPoints :: Maze -> Point
    -> [Point]

isPath = isPath -- dummy def