Haskell for Erlangers

(c) Simon Thompson
University of Kent, 2015

Rationale

Functional languages

- 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!

Haskerl

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 ≈ 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
- http://www.ibm.com/developerworks/java/library/j-jtp02183/j-jtp02183-pdf.pdf

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.

```
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.

```
N = 46,
N = 23+23,
N = 35,
 …

receiveFrom(Pid) ->
  receive
    {Pid,Payload} -> …
    … -> …
  end.

receive {foo,Foo} -> … end,
receive {bar,Bar} -> … end …
```

Open Telecom Platform

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

```
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 Erlangers

Strongly typed

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

```
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 .

```
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.

```
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

Modules in Haskell

- 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`.

```
module Demo where

import Demo2 hiding (foo)

bar … = … baz …
```

```
module Demo2(foo,baz) where

baz … = … …
```

The basics of Haskell

Function application

- In Erlang: traditional function application

```
iff(true,false)
```

- In Haskell: uses juxtaposition, just put the arguments after the function, separated by white space

```
iff True False
```

Type declarations

- The type declaration is optional.
- `:type iff` in GHCi will tell you the most general type.

```
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 …

```
'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** `Bool`ean expression.
- Erlang: compare with `when`

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

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

max’ x y
  | x>=y       = x
  | otherwise  = y

max’’ x y
  | x>y = x
max’’ x y
  = y
```

Local definitions

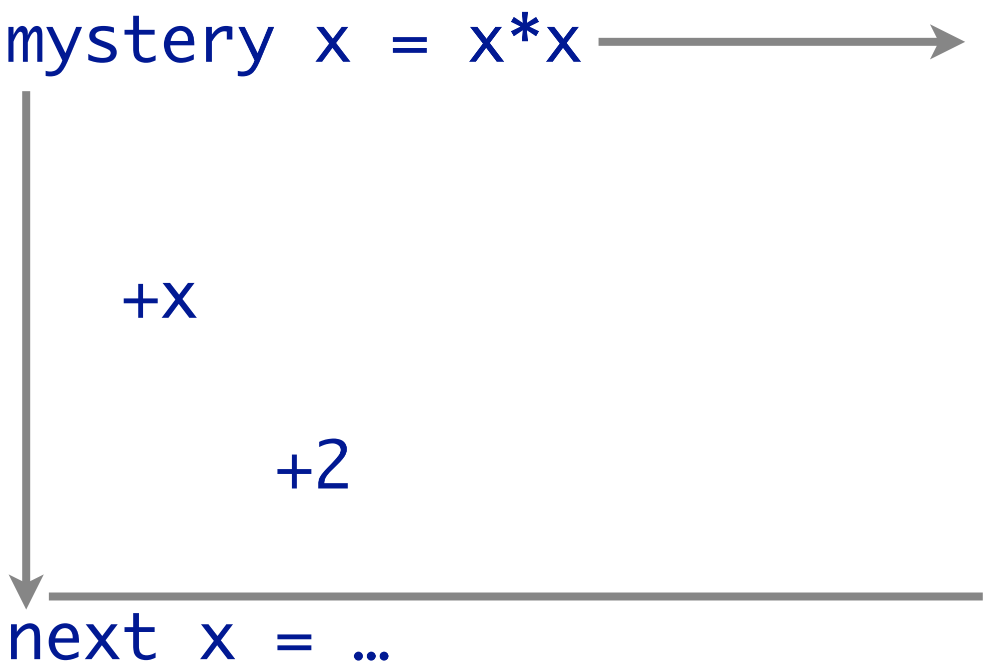
- Definitions can be local: `where` and `let`.
- `where`s are local to function equations.
- `let` definitions are local to expressions.
- Size of Haskell … .

```
triArea a b c
  = sqrt(s*(s-a)*(s-b)*(s-c))
    where
    s = (a+b+c)/2
```

```
triArea a b c
  = let
     s = (a+b+c)/2
    in
     sqrt(s*(s-a)*(s-b)*(s-c))
```

Layout sensitive

- The first character of a definition opens up a box …
- … which is closed only when something below or to the left.
- “Offside rule”



Layout in practice

- In Emacs with Haskell mode, repeated tabbing will take you through various sensible layout options.

```
mystery x y …
  | guard1    = result1
  | guard2    = res…
                   …ult2
   …
  | guardn    = resultn
    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

Rock - Paper - Scissors

- Enumerated type with three elements.
- Plus a bit of type class magic (later).
- Definitions by pattern matching.

```
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
 …
```

data types

- Elements of the `People` type are of the form

  `Person n a`

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

```
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}`

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

```
Person
  :: Name -> Age -> People
```

Compare

- Compare product types with tuples.

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


type People
    = (Name, Age)
```

Alternatives

- 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+)`, ``(`rem`2)``.

```
(&&) True False --> False

map (1+) [2,3] --> [3,4]

filter ((/=0).(`rem`2)) [1..9]
  --> [1,3,5,7,9]
```

Parentheses

- Grouping: in `deriving`, contexts, …

```
… deriving (Eq, Show)

… (Eq a, Show a) => a -> Int
```

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

```
sum (Node t1 t2) = …

sum (x:xs) = …

4-(3-2)

foldr (*) (1::Integer)
      [1..1000]
```

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=undef::Int in
  ite True 2 undef
 --> 2
```

Lazy evaluation

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

```
repeat :: a -> [a]

repeat x
  = xs
    where
    xs = x : xs

replicate :: Int -> a -> [a]

replicate n x
  = take n (repeat x)

take :: Int -> [a] -> [a]

take 0 _ = []
take n (x:xs)
  = x : 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.

```
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?*

```
length []     = 0
length (x:xs) = 1 + length 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

```
length []     = 0
length (x:xs) = 1 + length xs
```

`length` is a function

result is an `Int`

argument is a list

no constraint on list elements

```
length :: [a] -> Int
```

First of a pair

```
fst (x,_) = x
```

`fst` is a function

result is the 1st element

argument is a pair

no constraint on 2nd elements

```
fst :: (a,b) -> a
```

Mapping along a list

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

`map` is a function

result is a list

`f` is a function

2nd arg is a list

result elements are results of `f`

2nd arg elements have `f` applied

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

Other examples

```
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:
- Synonyms (`type`), e.g. generalised strategy.
- Algebraic types (`data`)

```
type Strategy a
  = [a] -> a

data Tree a
  = Leaf a
  | Node (Tree a) (Tree a)
  deriving …
```

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 *memoise* the minima?

```
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.

```
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

```
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`

`a` *is an instance of the* `Eq` *type class*

```
elem :: (Eq a) => a -> [a] -> 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?

```
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:

```
type Parse a = String -> a
```

- Extract an object of type `a` from a `String`.

`bracket ”(234” --> ’(’`

`number ”234” --> 2` or `23` or `234` …

`bracket ”234” -->` no result

The Parse type

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

```
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.

```
type Parse a
  = String -> [(a,String)]
```

```
bracket "(234" --> [('(',"234")]

number "234" -->  [(2,"34"),
                   (23,"4"),
                   (234,"")]

bracket "234" --> []
```

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=undef::Int in
  ite True 2 undef
 --> 2
```

Lazy evaluation

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

```
repeat :: a -> [a]

repeat x
  = xs
    where
    xs = x : xs

replicate :: Int -> a -> [a]

replicate n x
  = take n (repeat x)

take :: Int -> [a] -> [a]

take 0 _ = []
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.

```
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.
- Partial application.

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

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

ite True "foo"
```

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.

```
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.
- `True` = empty
- `False` = occupied

```
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.

```
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.

```
type Maze = [[[Point]]]

type Point = (Int,Int)

type Path = [Point]
```

Type-driven development

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

```
type Maze = [[Bool]]

type Point = (Int,Int)

type Path = [Point]
```

Example

```haskell
mazeSt1 :: [String]

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

```haskell
makeMaze :: [String] -> Maze

makeMaze = map (map (=='.'))
```

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
```