Haskell: an introduction

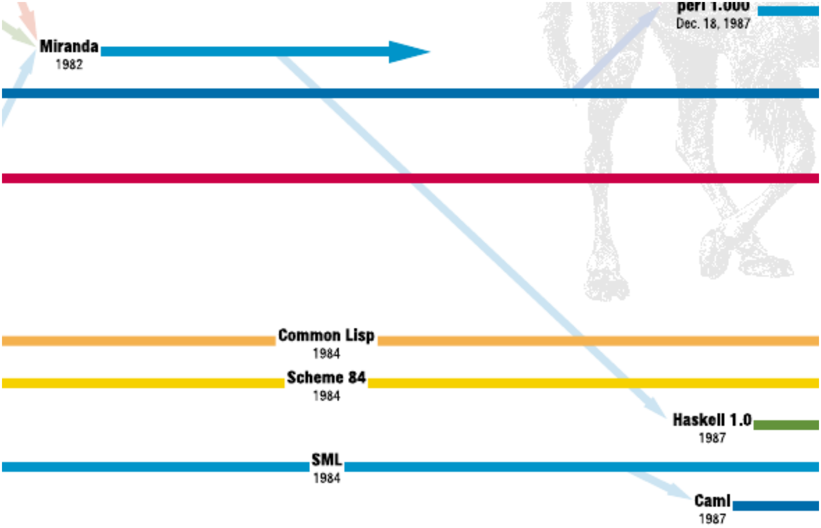
Sylvain HENRY
sylvain.henry@inria.fr

University of Bordeaux - LaBRI - Inria

February 5th, 2013

Outline

Outline


perl 1.000
Dec. 18, 1987
Miranda
1982
Common Lisp
1984
Scheme 84
1984
Haskell 1.0
1987
SML
1984
Caml
1987

Haskell

1. purely functional programming language
2. lazily evaluated
3. statically typed
4. elegant and concise

Functional: functions

$$f : x \mapsto 10 + x \quad h = f \circ g$$
$$g : x \mapsto x \times 2 \quad r = h(10)$$

```
f x = 10 + x
g x = x * 2
h = f . g
r = h 10
```

$$k : (x, y) \mapsto x + y$$
$$r2 = k(10, 20)$$

```
k x y = x + y
r2 = k 10 20
f = k 10    -- currying
```

Functional: higher-order functions

```haskell
g f x = (f x) * (f (f x))
```

```haskell
square x = x ^ 2

r = g square 2                       -- r = 64

r = g (\x -> x*2 + x^2) 2            -- r = 640

r = map (* 2) [1,2,3]                -- r = [2,4,6]
```

Functional: pattern-matching

```haskell
-- Lists are "cons-lists"
[1,2,3] == 1 : [2,3]
        == 1 : 2 : [3]
        == 1 : 2 : 3 : []
```

```haskell
length list = case list of
                []   -> 0
                x:xs -> 1 + (length xs)
```

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

Purely functional

Properties

- Immutable variables
  - if $a = 5$, it is true for the entire scope
- Referential transparency
  - f x *always* returns the same result for given f and x
- No side effect
  - IO...

Very good properties to

- reason about programs (program transformations, proofs...)
- compose functions together to write complex programs

Purely functional

Properties

- Immutable variables
  - if $a = 5$, it is true for the entire scope
- Referential transparency
  - f x *always* returns the same result for given f and x
- No side effect
  - IO...

Very good properties to

- reason about programs (program transformations, proofs...)
- compose functions together to write complex programs

Purely functional

Properties

- Immutable variables
  - if $a = 5$, it is true for the entire scope
- Referential transparency
  - f x *always* returns the same result for given f and x
- No side effect
  - IO...

Very good properties to

- reason about programs (program transformations, proofs. . .)
- compose functions together to write complex programs

Purely functional

Properties

- Immutable variables
  - if $a = 5$, it is true for the entire scope
- Referential transparency
  - f x *always* returns the same result for given f and x
- No side effect
  - IO...

Very good properties to

- reason about programs (program transformations, proofs. . .)
- compose functions together to write complex programs

Purely functional

Properties

- Immutable variables
  - if $a = 5$, it is true for the entire scope
- Referential transparency
  - f x *always* returns the same result for given f and x
- No side effect
  - IO...

Very good properties to

- reason about programs (program transformations, proofs. . .)
- compose functions together to write complex programs

Purely functional: *what* to compute vs *how*

$$\text{e.g. } \|\mathbf{x}\| = \sqrt{x_1^2 + \cdots + x_n^2}$$

```
double norm(int n, double * xs) {   // C
    double s = 0.0;
    for (int i=0; i<n; i++)
        s += xs[i] * xs[i];
    return sqrt(s);
}
```

```
norm = sqrt . sum . map (^2)        -- Haskell
```

Lazily evaluated

Only compute what is needed to produce the result, nothing more

```
doubles = map (* 2) [1,2,3,4,5]

r = doubles !! 1    -- indexed access (r = 4)

-- 2, 6, 8 and 10 are NOT computed
```

Lazily evaluated: infinite data!

```haskell
doubles = map (* 2) [1,2..]

r = doubles !! 15    -- r = 32

r = length doubles   -- !!! infinite "loop"
```

Lazily evaluated: infinite data!

Fibonacci number:

```haskell
fibs = [0,1] ++ zipWith (+) fibs (tail fibs)

r = fibs !! 15          -- r = 610
```

fibs	=	[0, 1, 1, 2, 3, 5, 8...]
tail fibs	=	[1, 1, 2, 3, 5, 8, 13...]
zipWith (+) fibs (tail fibs)	=	[1, 2, 3, 5, 8, 13, 21...]

Lazily evaluated: infinite data!

Fibonacci number:

```
fibs = [0,1] ++ zipWith (+) fibs (tail fibs)

r = fibs !! 15          -- r = 610
```

fibs	=	[0, 1, 1, 2, 3, 5, 8...]
tail fibs	=	[1, 1, 2, 3, 5, 8, 13...]
zipWith (+) fibs (tail fibs)	=	[1, 2, 3, 5, 8, 13, 21...]

Lazily evaluated: infinite data!

Fibonacci number:

```
fibs = [0,1] ++ zipWith (+) fibs (tail fibs)

r = fibs !! 15          -- r = 610
```

$$\begin{aligned} \text{fibs} &= [0, 1, 1, 2, 3, 5, 8...] \\ \text{tail fibs} &= [1, 1, 2, 3, 5, 8, 13...] \\ \text{zipWith } (+) \text{ fibs (tail fibs)} &= [1, 2, 3, 5, 8, 13, 21...] \end{aligned}$$

Lazily evaluated: infinite data!

Fibonacci number:

```haskell
fibs = [0,1] ++ zipWith (+) fibs (tail fibs)

r = fibs !! 15          -- r = 610
```

fibs	=	[0, 1, 1, 2, 3, 5, 8...]
tail fibs	=	[1, 1, 2, 3, 5, 8, 13...]
zipWith (+) fibs (tail fibs)	=	[1, 2, 3, 5, 8, 13, 21...]

Lazily evaluated: infinite data!

Fibonacci number:

```haskell
fibs = [0,1] ++ zipWith (+) fibs (tail fibs)

r = fibs !! 15          -- r = 610
```

$$\begin{array}{rcl} \text{fibs} & = & [0, 1, 1, 2, 3, 5, 8...] \\ \text{tail fibs} & = & [1, 1, 2, 3, 5, 8, 13...] \\ \text{zipWith } (+) \text{ fibs (tail fibs)} & = & [1, 2, 3, 5, 8, 13, 21...] \end{array}$$

Statically typed: type inference

Strong type checking at compile time

```
ghci> let doubles = map (* 2) [1,2..]
ghci> doubles !! 15
32

ghci> :type doubles
doubles :: [Integer]

ghci> :type (doubles !!)
(doubles !!) :: Int → Integer
```

- ghci is the REPL (Read-Eval-Print-Loop) of GHC
- Very useful to quickly test and check!

Type checking

```
ghci> "Test" + 10

    No instance for (Num String) arising from
       a use of `+'
     Possible fix: add an instance declaration
       for (Num String)
     In the expression: "Test" + 10
```

- Ok it works

Elegant and Concise

```haskell
-- HsCat.hs
main = interact id
```

```haskell
-- HsLineCount.hs
main = interact (printf "%d" . length . lines)
```

- Compile with "ghc XXX.hs"

```
$ ./HsCat < betteraves.txt
Pif
Paf
Pouf
$ ./HsLineCount < betteraves.txt
3
```

Outline

Real World Haskell

- Not only a calculator language
- You can have
  - IO (files, GUI, networking. . .)
  - Random numbers
  - Threads
  - Bindings to C libraries
  - . . .
- Examples
  - xmonad (tiling X window manager)
  - ghc (Haskell compiler)
  - darcs (git-like version control system)
  - . . .


Side-effects...
You must be kiddin'

IO: first try

```
getLine   :: String         -- read a line on stdin
putStrLn  :: String -> ()   -- write a line on stdout
```

```
main =
 let lastName = getLine in
   let firstName = getLine in
     putStrLn ("Hi" ++ firstName ++ lastName)
```

- Haskell is referentially transparent
  - lastName = getLine = firstName
- Haskell is lazily evaluated
  - Is firstName or lastName evaluated first?
  - What if we never use firstName or lastName?
  - putStrLn does not return a result and will never be evaluated

IO: first try

```
getLine  :: String          -- read a line on stdin
putStrLn :: String -> ()    -- write a line on stdout
```

```
main =
 let lastName = getLine in
   let firstName = getLine in
     putStrLn ("Hi" ++ firstName ++ lastName)
```

- Haskell is referentially transparent
  - lastName = getLine = firstName
- Haskell is lazily evaluated
  - Is firstName or lastName evaluated first?
  - What if we never use firstName or lastName?
  - putStrLn does not return a result and will never be evaluated

IO: first try

```haskell
getLine   :: String         -- read a line on stdin
putStrLn  :: String -> ()   -- write a line on stdout
```

```haskell
main =
 let lastName = getLine in
   let firstName = getLine in
     putStrLn ("Hi" ++ firstName ++ lastName)
```

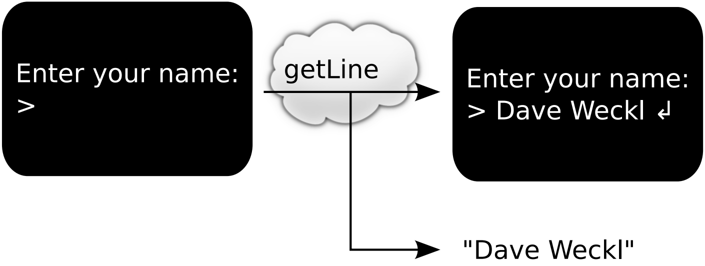
- Haskell is referentially transparent
  - lastName = getLine = firstName
- Haskell is lazily evaluated
  - Is firstName or lastName evaluated first?
  - What if we never use firstName or lastName?
  - putStrLn does not return a result and will never be evaluated

IO: back to basics

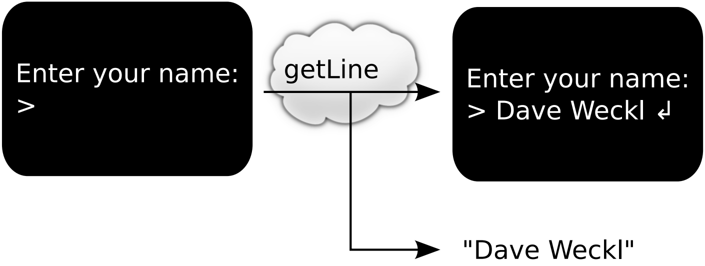
```
Enter your name:
>
```

IO: back to basics

IO: back to basics

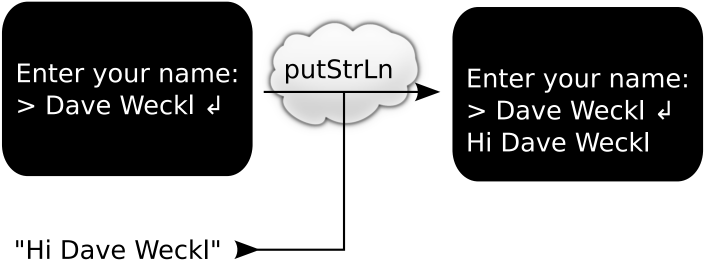


IO: back to basics

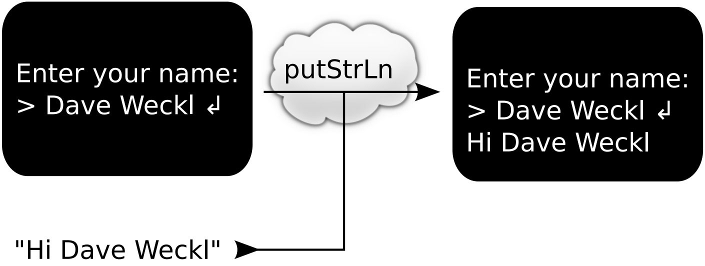


getLine :: World →(World, **String**)

IO: back to basics



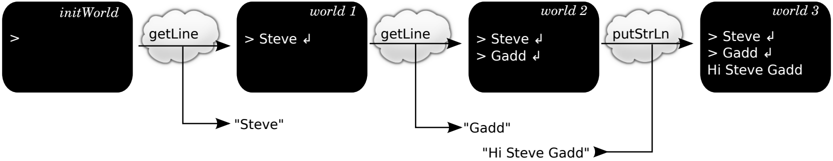
IO: back to basics



putStrLn :: **String** $\rightarrow$World $\rightarrow$(World, ())

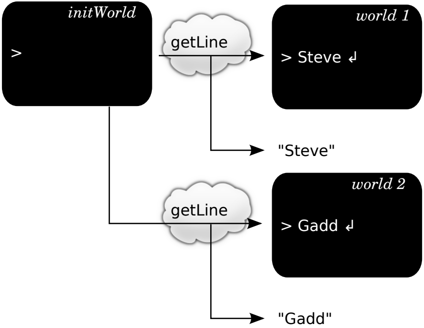
IO: explicit "world" variables

```
getLine   :: World → (World, String)
putStrLn  :: String → World → (World, ())
```




```
main :: World → (World, ())
main initWorld =
 let (world1, fi) = getLine initWorld in
  let (world2, la) = getLine world1 in
   putStrLn (printf "Hi␣%s␣%s" fi la) world2
```

Issue: what if a "world" variable is used several times?

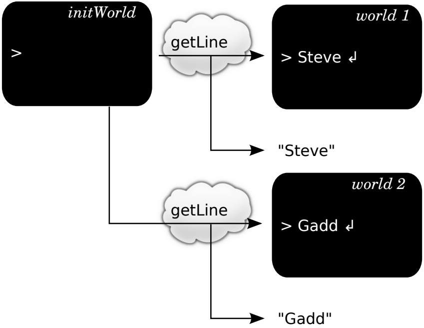



```haskell
main initWorld =
 let (w1, fi) = getLine initWorld in
 let (w2, la) = getLine initWorld in
 putStrLn (printf "Hi %s %s" fi la) w2
```

Two solutions (at least)

- Check that it is not the case (Clean's *uniqueness types*)
- No explicit "world" variables: provide composition operators instead (Haskell's monadic IO)

Issue: what if a "world" variable is used several times?

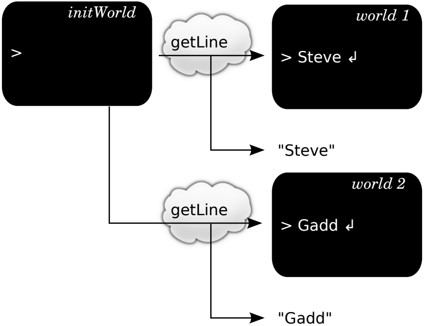



```haskell
main initWorld =
 let (w1, fi) = getLine initWorld in
 let (w2, la) = getLine initWorld in
 putStrLn (printf "Hi %s %s" fi la) w2
```

Two solutions (at least)

- Check that it is not the case (Clean's *uniqueness types*)
- No explicit "world" variables: provide composition operators instead (Haskell's monadic IO)

Issue: what if a "world" variable is used several times?




```haskell
main initWorld =
  let (w1, fi) = getLine initWorld in
  let (w2, la) = getLine initWorld in
  putStrLn (printf "Hi␣%s␣%s" fi la) w2
```

Two solutions (at least)

- Check that it is not the case (Clean's *uniqueness types*)
- No explicit "world" variables: provide composition operators instead (Haskell's monadic IO)

IO: composition operators

```
type W = World
```

```
-- Composition which uses retrieved value
(>>=) :: (W → (W, a)) → (a → W → (W, b)) → (W → (W, b))

main = getLine >>= putStrLn
     = getLine >>= (\input → putStrLn input)
```

```
-- Composition like imperative ";"
(>>) :: (W → (W, a)) → (W → (W, b)) → (W → (W, b))

main = putStrLn "Hello" >> putStrLn "World"
     = putStrLn "Hello" >>= (\dummy → putStrLn "World")
```

```
-- Lift pure value
return :: a → W → (W, a)

main = return "Ho" >>= (\s → putStrLn s >> putStrLn s)
```

IO: composition operators sugarized

```haskell
type IO a = World → (World, a)
```

```haskell
-- Composition which uses retrieved value
(>>=) :: IO a → (a → IO b) → IO b

main = getLine >>= putStrLn
     = getLine >>= (\input → putStrLn input)
```

```haskell
-- Composition like imperative ";"
(>>) :: IO a → IO b → IO b

main = putStrLn "Hello" >> putStrLn "World"
     = putStrLn "Hello" >>= (\dummy → putStrLn "World")
```

```haskell
-- Lift pure value
return :: a → IO a

main = return "Ho" >>= (\s → putStrLn s >> putStrLn s)
```

IO: working example using composition operators

```haskell
main :: IO ()
main =
  putStrLn "First name:" >>
  getLine >>= \firstname ->
    putStrLn "Last name:" >>
    getLine >>= \lastname ->
      return (hi firstname lastname) >>= \s ->
        putStrLn s

hi :: String -> String -> String
hi fi la = printf "Hello %s %s" fi la
```

IO: working example using do-notation

```
main :: IO ()
main = do
  putStrLn "First␣name:"
  firstname ← getLine
  putStrLn "Last␣name:"
  lastname ← getLine
  let s = hi firstname lastname
  putStrLn s

hi :: String → String → String
hi fi la = printf "Hello␣%s␣%s" fi la
```

Simon L. Peyton Jones and Philip Wadler.
Imperative functional programming.
In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.

Outline


AWESOMENESS
Ask Barney

Haskell Awesomeness 1/2

- List-Comprehensions:
  - e.g. $r = \{x + y, x \in xs, y \in ys, x \geq 10, y > x\}$

```
r = [x + y | x ← xs, y ← ys, x >= 10, y > x]
```

- QuickCheck

```
> quickCheck ((\s → reverse s == s) :: String → Bool)
*** Failed! Falsifiable (after 3 tests and 2 shrinks):
"ab"

> quickCheck ((\s → (reverse (reverse s) == s) :: String →
    Bool)
+++ OK, passed 100 tests.
```

Haskell Awesomeness 2/2

- Parallelism

```
a ← async (f x y)
b ← async (g w t)
r ← waitAny [a,b]
```

- Software Transactional Memory (STM)

```
transfer n account1 account2 = atomically $ do
   x ← readTVar account1
   when (x < n) retry
   writeTVar account1 (x − n)
   y ← readTVar account2
   writeTVar account2 (y + n)
```

Haskell Awesomeness 3/2

- Haskell language
  - Committee driven
  - Fully specified
- Glasgow Haskell Compiler (GHC)
  - BSD license
  - Actively developed
  - Generate native code
    - Optional LLVM backend
    - C bindings easy to write and use
  - Optimizing compiler
  - hlint: "checkstyle"-like for Haskell
- Community
  - #haskell, mailing-lists, wiki, papers
  - Hackage: 4500+ packages

Questions?