Introduction to Haskell

Ian-Woo Kim (PH-TH) and Dániel Pék (IT-PES)
ian-woo.kim@cern.ch, daniel.pek@cern.ch

Agenda

- Introduction
- A taste of Haskell
- Summary

Introduction

- History
- What is Haskell?
- Functional paradigm
- Real-world Haskell

What is Haskell (peek)

- Haskell is a lazy functional programming language.

David Turner (Miranda)

Philip Wadler (type theory, monads in FP)

John Hughes (QuickCheck)

Simon Peyton Jones ("Father" of GHC)

Paul Hudak

IFIP WG2.8 Oxford, 1992

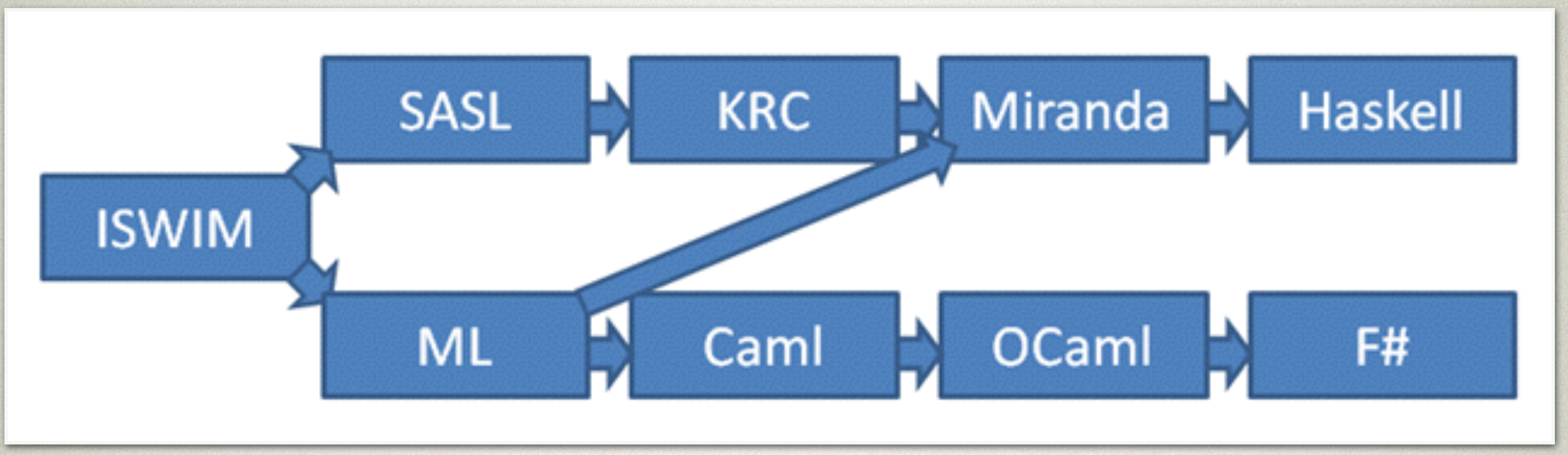
Source: "A History of Haskell: being lazy with class"
Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler

History (1)

- ‘50s - ‘70s: birth of the functional era
  - Big names involved: John Backus (BNF), John McCarthy (AI), Edsger Dijkstra…
  - Importance of **lambda calculus**
  - Lisp, Scheme, SASL, ISWIM, ML…
  - **Strict** (call-by-value) functional languages with imperative features

History (2)

- ‘80s
  - Idea of **lazy** (non-strict, call-by-need) evaluation
  - Lazy Lisp evaluator, lazy SASL, KRC, Miranda, Lazy ML, Orwell, Alfl, Id, Clean, Ponder, Daisy
  - A tower of Babel…



Some connections

Source: Rick Minerich
http://www.atalasoft.com/cs/blogs/rickm/archive/2009/01/29/f-and-haskell-estranged-cousins.aspx

History (3)

- 1987, birth of Haskell
  - At FPCA conference, spontaneous meeting
    - Simon Peyton Jones, Paul Hudak, Philip Wadler
  - Need for a common lazy functional language

History (4)

- 1987 - 1997
  - Committee meetings, e-mails, design decisions, several reports
    - Naming: after Haskell Brooks Curry
    - Haskell 1.0 (April 1, 1990!) - 1.4
    - February 1999: the first standard: **The Haskell 98 Report**

History (5)

- Haskell was a carefully designed committee language
- After 1999
  - Haskell 98 standard as a solid base
  - Language extensions, experimenting
  - Language (type-system) laboratory for the academia

What is Haskell?

Haskell is a programming language, which is

- standardised,
- general-purpose,
- pure,
- functional,
- lazy evaluated,
- strongly and statically typed with type inference.

Standardised

- "Freezes" the **language feature-set**
- Commitment to support these standards indefinitely
  - Haskell 98 (1999, revised in 2003)
  - Haskell 2010

General-purpose

- Wide variety of application domains
- **Not domain-specific**
- Wide range of libraries, modules, tools are available for different domains

Pure

- Function is a **function in the mathematical meaning** of the word
- f(24) guaranteed to return the very same value every time (**no side-effects**, aka. user input, random generator, output, etc.)
- Non-pure function, or “procedure”:

```
function f(int x)
    string str = getStringFromUser()
    return (stringToInt(str) + x)
```

- Now, f(24) is … ?

Functional

- Variables are **constants**, immutables, “symbols”
- **Function is** not special, it’s **like a value**
- You can do fancy stuff with them easily
  - **Composition**: $h = f \circ g$
  - **Partial application**: $g(x) = f(2, x)$
  - **Pass it as a parameter**: $g(f, 2, \dots)$, where f is a function

Lazy

- "Don't calculate it until you need it"
- **Compiler decides** over **the order** of instructions
- Several advantages:
  - Makes the program declarative
  - Infinite datatypes
  - Compiler optimisations

Strong and static typing with type inference

- Only **explicit type conversion**
- **Type system** is expressive and **powerful**
  - Catch errors at compile time!
- **Types** can be **inferred** by the compiler
  - No need to write down explicit types
  - Conflicts between user-given and inferred types => compile-time error

Functional Paradigm

- **Function as 1st class value**
  - Store functions in variables as other values
  - Higher-order function: function as an argument, function as a return value
- **Purity:** explicit control of side effects
- **Recursion:** instead of loops
- **Evaluation strategy:** strict vs non-strict
- **Type systems:** Static vs Dynamic
- **Popular Languages:** Scheme, Common Lisp, ML, OCaml, F#, Clojure, Scala, …

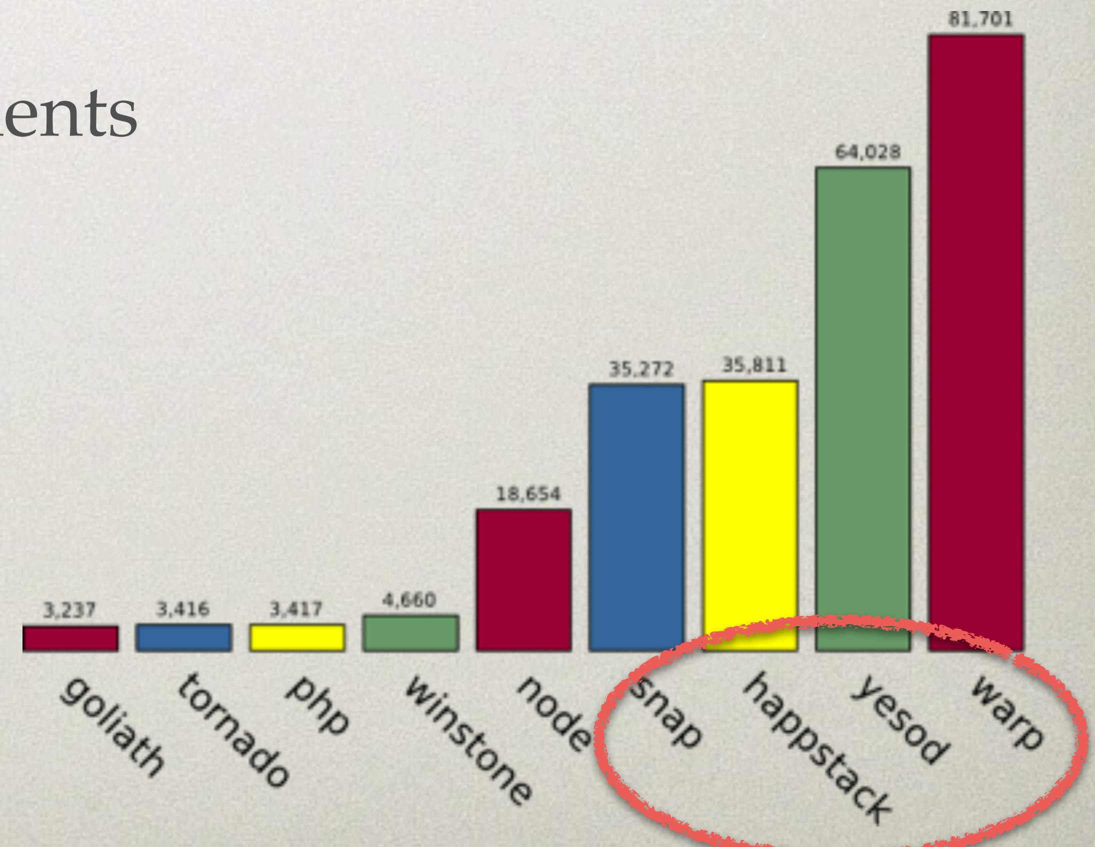
Real-world Haskell: Language Analysis

- Haskell a.k.a. language making language
- Excellent for Abstract Syntax Tree analysis
  - Language analysis tools: ex) language-c, ecmascript
  - Compiler/Interpreter: ex) PUGS (Perl 6)
  - Domain Specific Language: ex) Kansas Lava (FPGA)
  - Embedded DSL: ex) Accelerate (GPU)
- Implementation language for more experimental type system: useful for theorem provers, ex) Agda, Idris

Real-world Haskell: Web frameworks

- Web server frameworks
  - Happstack
  - Snap Framework
  - Yesod
  - Many open source components
- Haskell-to-Javascript!
  - UHC, Fay, Haste, ghcjs …
  - Static types to Javascript!
- Fast!

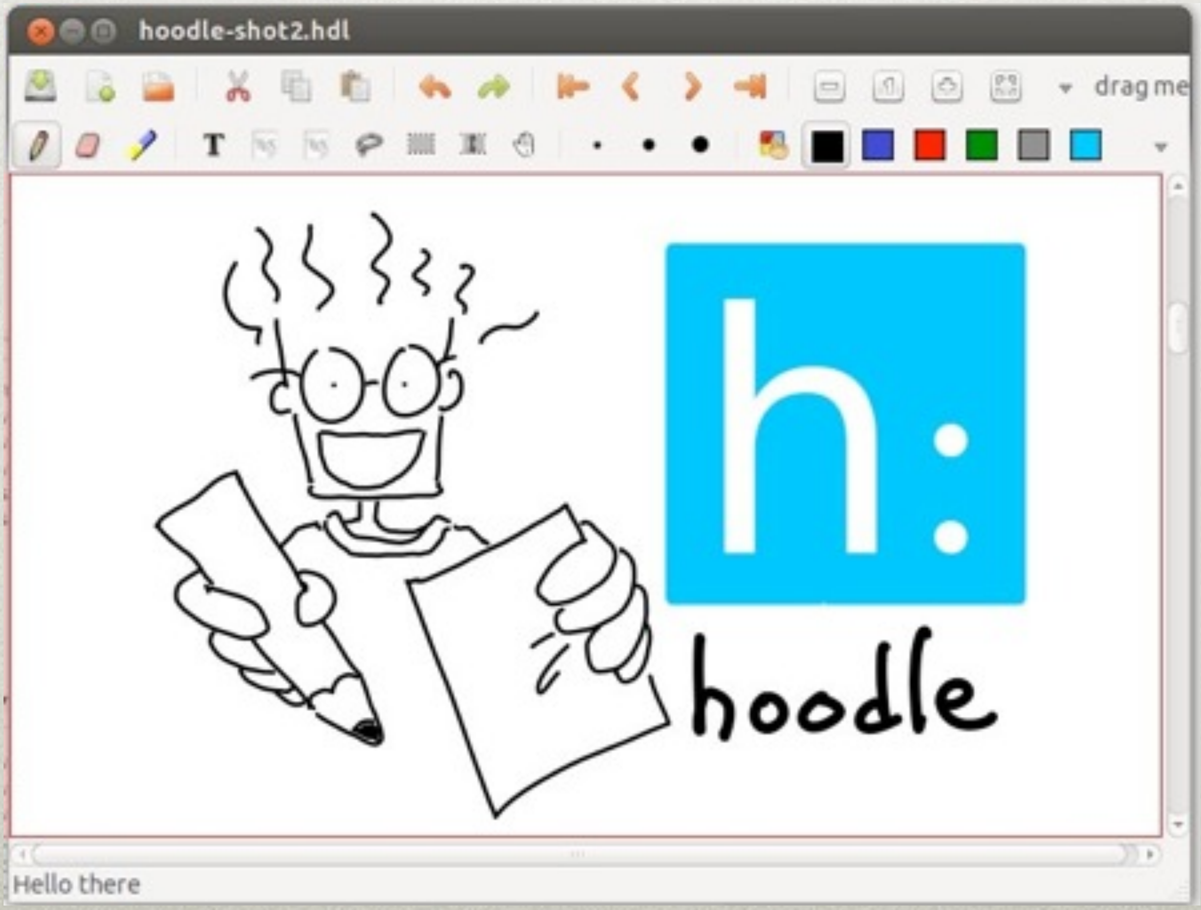
requests/sec Higher is better



Haskell!

Real-world Haskell: Application

- Haskell is ready for general purpose app.
- Easy to make FFI to libraries, such as GUI.
- Ex) hoodle : pen note-taking program (by IWK)



Demo: Hoodle

Youtube link

- Hoodle is written in haskell (~20k loc)
- Functionality:
  - Graphical pen note-taking functions
  - PDF annotation / asynchronous PDF rendering
  - Linking / doc database / version control
  - Web publishing
  - Tablet driver in C (~20 lines) integrated in Haskell
- External libraries: cairo, gtk, poppler, dbus
  - Haskell binding: Gtk2hs, poppler

Haskell Communities and Activities Report

http://tinyurl.com/haskcar

Twenty-Sixth Edition — May 2014

- We only skimmed a very small part of the whole haskell activities.
- One can find a more detail activity report in HCAR (still only small part)
- Industry leaders are using Haskell.

A taste of Haskell

- Basic syntax
- Types, type classes
- Performing IO
- Examples
- Ecosystem

Definitions

- Haskell definitions use =

```
-- value definition
x = 3
y = “string”
z = (1,2,3)

-- function definition
f x = x + 1
```

- = means a real value equivalence
- Haskell variables are not variable

```
x = x + 1 -- INVALID! THIS MEANS INFINITY!
```

Definitions

- Function definition :

```
-- unary function
f x = x + 1
-- binary function
f x y = x + y
-- ternary function
f x y z = (x + y) * z
```

- Lambda expression: function on the fly

```
f = \x -> x + 1
```

- Note that tuples are separate types.

```
f x y = x + y
g (x,y) = x + y
```

Data Type Declaration

- Data types are easy: Algebraic Data Types

```
data Shape = Dot
           | Circle Double
           | Box Double Double
```

- Multiple cases in a single data: sum type
- Record syntax

```
data Shape = Dot
           | Circle { radius :: Double }
           | Box    { width :: Double
                    , height :: Double }

rect = Box { width = 3, height = 4 }
```

Pattern Matching

- Cases are matched using `case .. of`

```
f x = case x of
        Dot      -> “dot”
        Circle r -> “circle with radius ”
                    ++ show r
        Box w h  -> “box with dimensions ”
                    ++ show w ++ “,” ++ show h
```

- or directly using function definition

```
f Dot        = “dot”
f (Circle r) = “circle with radius ” ++ show r
f (Box w h)  = “box with dimensions “
               ++ show w ++ “,” ++ show h
```

Type Annotation and Inference

- Haskell is a strongly typed language; we simply omitted the type signature until now

```
f :: Shape -> String
f Dot          = “dot”
f (Circle r) = “circle with radius ” ++ show r
f (Box w h)  = “box with dimensions ”
                ++ show w ++ “,” ++ show h
```

- Omitted type signatures are inferred. `f` is `Shape -> String` type clearly from `Dot`, `Circle` .. and resultant expressions.

Currying and Partial Application

- Type signature of multi-argument function
- Multiple arguments are not multiple!

```
f :: Int -> Int
f x = x + 1

f :: Int -> Int -> Int
  -- Int -> (Int -> Int)
f x y = x + y

f :: Int -> Int -> Int -> Int
  -- Int -> (Int -> (Int -> Int))
f x y z = (x + y) * z
```

- Called currying named after Haskell Curry.

Currying and Partial Application

- Partially applied functions are automatic!

```
f :: Int -> Int -> Int -> Int
  -- Int -> (Int -> (Int -> Int))
f x y z = (x + y) * z

g :: Int -> (Int -> Int)
g = f 3   -- i.e. g = \y z -> (3+y)*z


h :: Int -> Int
h = g 4   -- i.e. h = \z -> (3+4)*z
```

Guards

- Concise definition with multiple `if`s

```
f x = if x < 0
      then “negative”
      else if x < 1
           then “positive, less than 1”
           else “greater than or equal to 1”

f x | x < 0     = “negative”
    | x < 1     = “positive, less than 1”
    | otherwise = “greater than or equal to 1”
```

List Comprehension

- Haskell list definition is using [ ].

```
lst = [1,2,3]      -- equiv to 1:(2:(3:[]))
lst = [1..5]       -- [1,2,3,4,5]
lst = [1,3..9]     -- [1,3,5,7,9]
```

- Elegant syntax for constructing lists

```
lst = [(i,j) | i <- [1,2], j <- [1..3]]
```

```
lst = [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
```

- Local definition using `let`

```
lst = [ (i,j) | i <- [1..], let k = i*i, j <- [1..k]]
```

```
lst = [(1,1),(2,1),(2,2),(2,3),(2,4),(3,1),..]
```

Let and Where

- Local definition inside function

```
-- using let
f x y = let z = x + y
        in if x > 0 then z*z else y

-- using where
f x y = if x > 0 then z*z else y
      where z = x + y
```

- `where` enables declarative programming.
- Lazy evaluation: statement order doesn’t imply execution order.

Declarative Style
ex) Erastothenes Sieve

- Find a list of all prime numbers.

```
primes :: [Int]
primes = sieve [2..]   -- [2,3,4,5,…]
  where sieve (p:xs) =
          p:sieve [ x | x <- xs, x `mod` p > 0 ]
```

```
take 1000 primes
  = [2,3,5,7,11,13,…,7919]
```

- Lazy evaluation, where, and list comprehension

Parametrised Types

- Types can be parameterised.

```
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)
```

```
swap (1,2)      = (2,1)      :: (Int,Int)
swap (1,“str”) = (“str”,1) :: (String,Int)
swap (1,Dot)    = (Dot,1)    :: (Shape,Int)
```

- Detect errors by type inference.

```
x = 1
y = Circle 3
z = fst (swap (x,y)) + 1  -- Error! Circle + Int
```

Parameterised Data Types

- Data types can also be parameterised, Higher-order types.

```
-- List of type a. built-in definition
data [a] = [] | a : [a]

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

lst :: [Int]
lst = 1 : (2 : (3 : [])) -- [1,2,3]

tree :: Tree Shape
tree = Node (Leaf (Box 1 2))
            (Node (Leaf (Circle 1)) (Leaf Dot))
```

- Generic functions can be defined.

Higher-Order Functions

- Function can take functions as an argument and produce functions as a result.

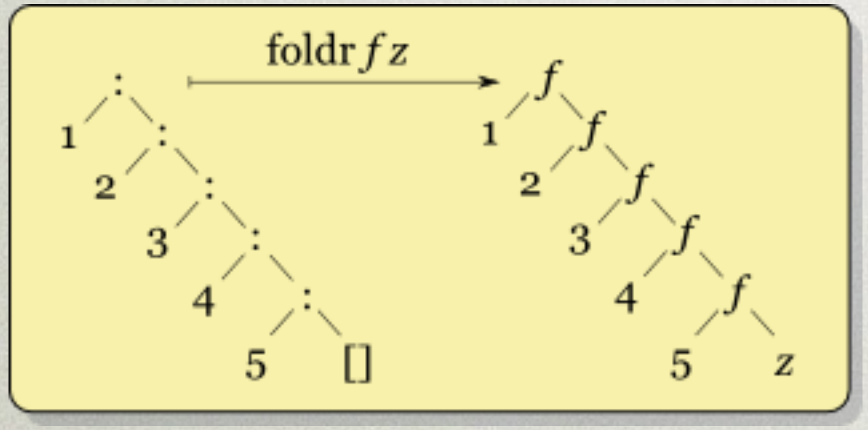
```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

```
plusOne x = x + 1
map plusOne (1:(2:[])) = 2:(3:[])

plusA x = x ++ “A”
map plusA (“1”:(“2”:[])) = “1A”:(“2A”:[]))
```

Higher-Order Functions

- Higher-order functions are elegantly useful!

```
foldr :: (a->b->b) -> b -> List a -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```




```
  foldr f z (1:(2:(3:(4:(5:[])))))
= f 1 (foldr f z (2:(3:(4:(5:[])))))
= f 1 (f 2 (foldr f z (3:(4:(5:[])))))
= f 1 (f 2 (f 3 (foldr f z (4:(5:[])))))
= f 1 (f 2 (f 3 (f 4 (foldr f z (5:[])))))
= f 1 (f 2 (f 3 (f 4 (f 5 (foldr f z [])))))
= f 1 (f 2 (f 3 (f 4 (f 5 z))))
```

- Summation of list elements

```
sum lst      = foldr (+) 0 lst
```

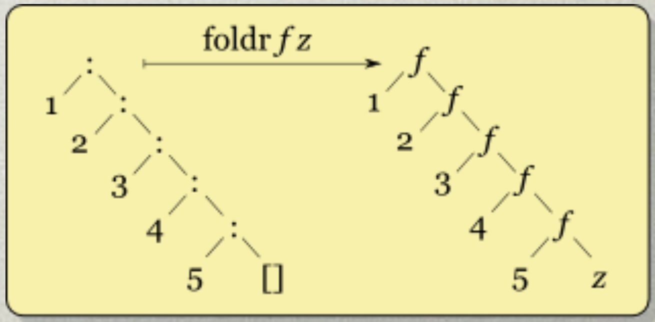
Higher-Order Functions

- Higher-order functions are elegantly useful!

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr f acc []= acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

- `foldr` shows generic definitions of similar operations. Many functions are in this class.

```
sum     lst = foldr (+) 0 lst
product lst = foldr (*) 1 lst
concat  lst = foldr (++) “” lst
```



Type Classes

- By the way, what is the type of `sum`?

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

-- sum :: [a] -> b ??
sum lst = foldr (+) 0 lst
```

- `(+)` operator can be defined only for a special class of types. Therefore, `sum` has the following type.

```
-- (+) :: Num a => a -> a -> a
sum :: Num a => [a] -> a
sum lst = foldr (+) 0 lst
```

Type Classes

- Type class: a constraint requirement to types

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: a -> a
```

- Declare a type as an instance of a type class.

```
instance Num Int where
  x + y = …
  x * y = …
  …
```

Type Classes ex) Monoid

```
class Monoid m where
  mempty :: m
  (<>) :: m -> m -> m

instance Monoid Int where
  mempty = 0
  (<>) = (+)
```

$(\mathbb{Z}, +, 0)$

(String, ++, “”)

- Define generic `fold` for `Monoid`

```
fold :: (Monoid m) => [m] -> m
fold xs = foldr (<>) mempty xs

-- sum = fold
```

- Higher-order functions and type classes are very versatile build blocks

Programming with Combinators ex) sort

- Generic interface to sorting

```
data Ordering = LT | EQ | GT
sortBy :: (a->a->Ordering) -> [a] -> [a]
flip f = \x y -> f y x

class Ord a where
   compare :: a -> a -> Ordering

sort :: Ord a => [a] -> [a]
sort xs = sortBy compare xs

reverseSort :: Ord a => [a] -> [a]
reverseSort xs = sortBy (flip compare) xs
```

- Usual `sort` is a special case of `sortBy` with default comparing function.

Programming with Combinators ex) sort

- Sort on the 1st elements in a list of pairs.

```
data Ordering = LT | EQ | GT

sortBy :: (a -> a -> Ordering) -> [a] -> [a]

on :: (b->b->c) -> (a->b) -> a -> a -> c
g `on` f = \x y -> g (f x) (f y)

sortOnFst :: Ord a => [(a,b)] -> [(a,b)]
sortOnFst lst = sortBy (compare `on` fst) lst
```

PERFORMING I/O

- Echo with capitalising letters and return.

```
==============================
    Program Specification
==============================
x <- capEcho : x is result
------------------------------
abc          : user input
ABC          : term output
==============================
print x      : result
------------------------------
ABC          : result = output
```

- Apparently, the concept of purity conflicts with the concept of side effects…

PERFORMING I/O

- Echo with capitalising letters and return.

```
capEcho = do ostr <- getLine
             let nstr = toUpper ostr
             putStrLn nstr
             return nstr
```

- It's simply **do**able!

PERFORMING I/O

- Echo with capitalising letters and return.

```
capEcho :: IO String
capEcho = do ostr <- getLine
             let nstr = toUpper ostr
             putStrLn nstr
             return nstr

getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
toUpper :: String -> String
```

- I/O Effects are wrapped in `IO a`
- `IO` is `Monad.`

E. Moggi P. Wadler

Do Notation

- Syntatic Sugar for `Monad`

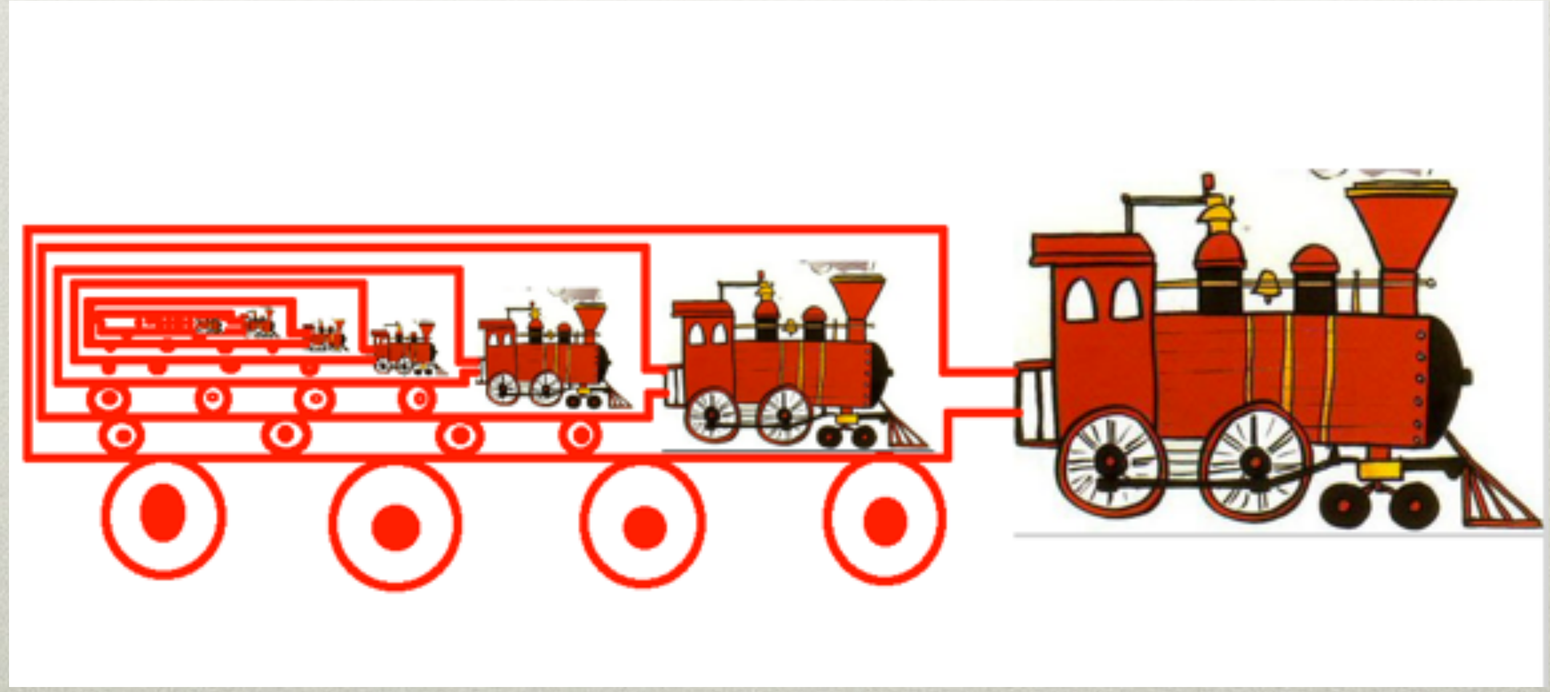
```
capEcho :: IO String
capEcho = do ostr <- getLine
             let nstr = toUpper ostr
             putStrLn nstr
             return nstr

capEcho :: IO String
 capEcho =
   getLine >>=
     (\ostr -> let nstr = toUpper ostr
               in putStrLn nstr
                  >>= (\_ -> return nstr) )
```

- Combine multiple sub-operations by chaining side effects.

Monad Separation

- Capture effects in Monadic jail

```
capEcho :: IO String
capEcho =
  getLine >>=
    (\ostr -> let nstr = toUpper ostr
              in putStrLn nstr
                 >>= (\_ -> return nstr) )
```



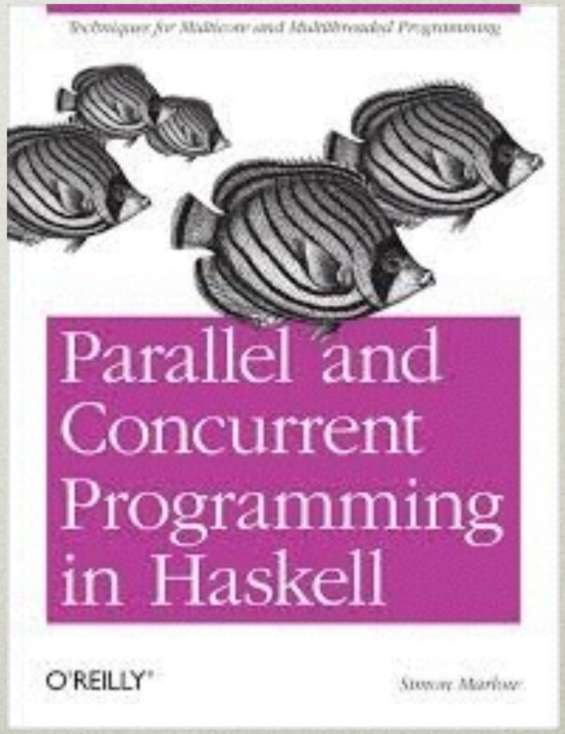
Monad is Everywhere

- Effects are everywhere…
  - `Maybe a, Either r a` : Success/Failure
  - `State s a` : Stateful operation
  - `Parser a` : Special state monad with string stream
  - `List a ([a])` : Intermediate solution space
- Monadic abstraction separates pure code from effectful code.

```
capEcho = do ostr <- getLine             -- monadic
             let nstr = toUpper ostr     -- pure!
             putStrLn nstr               -- monadic
             return nstr                 -- monadic
```

Parallelism/ Concurrency

- Haskell is excellent in parallelism and concurrency!
- GHC Run Time System (RTS)
  - Lightweight threads (spark): very little overhead compared with OS threads
  - Very efficient parallel I/O management
  - Constructs for concurrent operation
  - Software Transactional Memory (STM)
- Full of interesting ideas!



Simon Marlow

Ex) Arithmetic interpreter (1)

- (1 + 3.5 * 2) * 2.2 + 2.0
- Cheat: right-associative infix operators
- Grammar (Backus-Naur form)

```
<expr> ::= <term> '+' <expr> | <term>
<term> ::= <factor> '*' <term> | <factor>
<factor> ::= '(' <expr> ')' | <number>
```

Ex) Arithmetic interpreter (2)

```
import Data.Attoparsec.ByteString.Char8
import Control.Applicative

-- “interpreter” part
times = (*) <$ char '*'
add = (+) <$ char '+'

-- <expr> ::= <term> ‘+’ <expr> | <term>
expr = term <**> add <*> expr <|> term

-- <term> ::= <factor> ‘*’ <term> | <factor>
term = factor <**> times <*> term <|> factor

-- <factor> ::= ‘(‘ <expr> ‘)’ | <number>
parens = char '(' *> expr <* char ')'
factor = parens <|> double
```

Ex) Quicksort

- Inefficient, but easy-to-understand implementation

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Source: http://www.haskell.org/haskellwiki/Introduction#Quicksort_in_Haskell

Ex) Intuitive IO

```
myAction tempname temph = do
       putStrLn "Welcome to tempfile.hs"
       putStrLn ("I have a temporary file at " ++ tempname)

       pos <- hTell temph
       putStrLn ("My initial position is " ++ show pos)

       let tempdata = show [1..10] -- Creates a string "[1,2,3,4...
       putStrLn ("Writing one line containing " ++
                  show (length tempdata) ++ " bytes: " ++
                  tempdata)
       hPutStrLn temph tempdata

       pos <- hTell temph
       putStrLn ("After writing, my new position is " ++ show pos)

       putStrLn ("The file content is: ")
       hSeek temph AbsoluteSeek 0

       c <- hGetContents temph

       putStrLn c
```

http://book.realworldhaskell.org/read/io.html

Building and Packaging

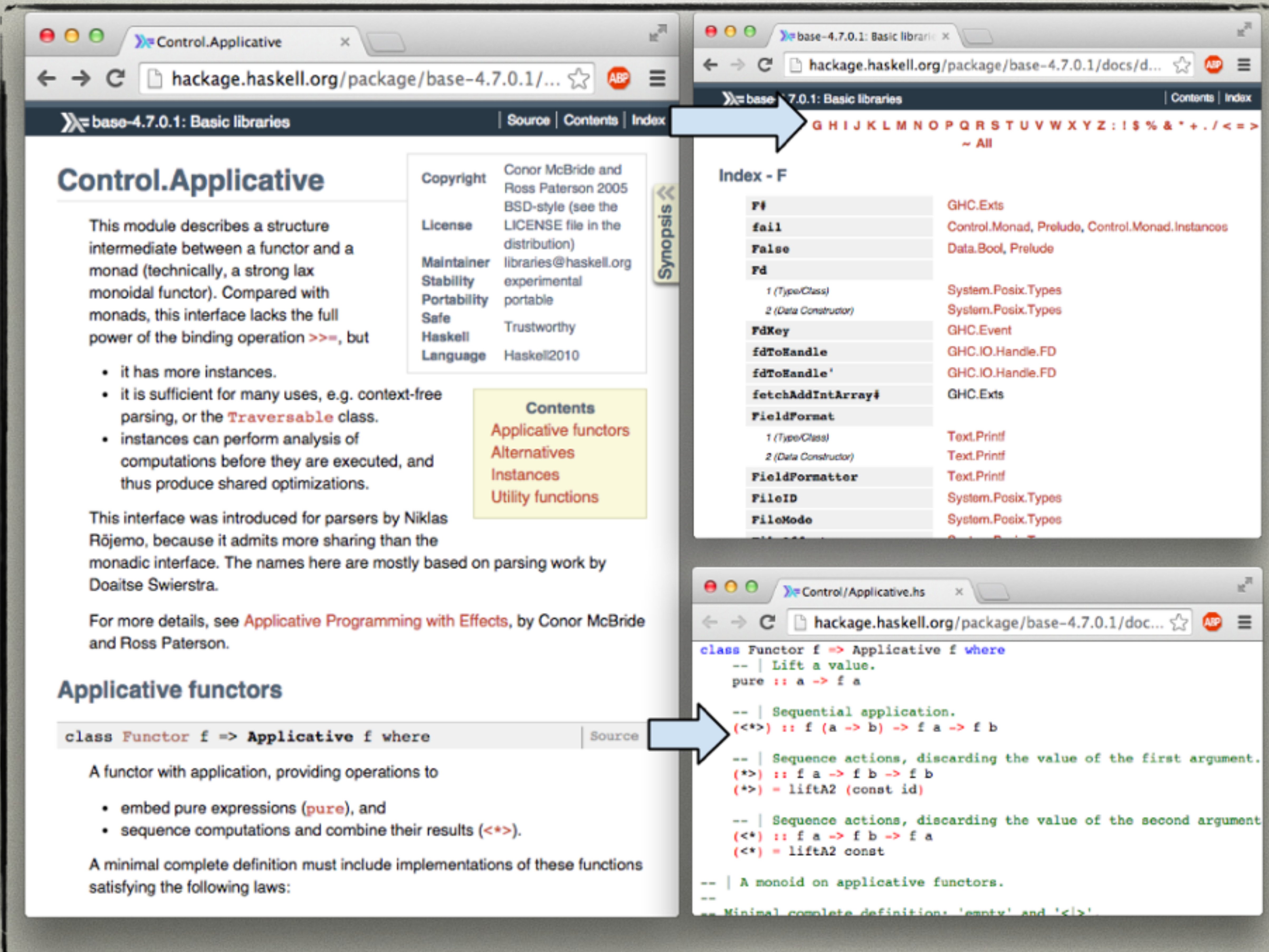
- **Cabal**: building and packaging Haskell libraries and programs
  - Analogy: Ruby gem, Python pip
- Features
  - Project scaffolding
  - Environments (sandbox)
  - Packaging, dependencies
  - Publishing

Cabal quickstart

- Install Haskell Platform (https://www.haskell.org/platform/)
- `# mkdir myproject; cd myproject`
- `# cabal init`
- Do some coding…
- Tailor myproject.cabal
- `# cabal install`
- => profit :)

Hackage (1)

- **Hackage:** central package archive
  - Analogy: CPAN, pypi, **rubygems.org**
- Features
  - Easy to publish: `cabal upload`
  - Package, source code and documentation in one place
  - Well indexed, searchable (see also: Hoogle)


Control.Applicative
hackage.haskell.org/package/base-4.7.0.1/...
base-4.7.0.1: Basic libraries
Source | Contents | Index
Control.Applicative
Copyright Conor McBride and Ross Paterson 2005
License BSD-style (see the LICENSE file in the distribution)
Maintainer libraries@haskell.org
Stability experimental
Portability portable
Safe Haskell Trustworthy
Language Haskell2010
Synopsis
This module describes a structure intermediate between a functor and a monad (technically, a strong lax monoidal functor). Compared with monads, this interface lacks the full power of the binding operation >>=, but
it has more instances.
it is sufficient for many uses, e.g. context-free parsing, or the Traversable class.
instances can perform analysis of computations before they are executed, and thus produce shared optimizations.
Contents
Applicative functors
Alternatives
Instances
Utility functions
This interface was introduced for parsers by Niklas Röjemo, because it admits more sharing than the monadic interface. The names here are mostly based on parsing work by Doaitse Swierstra.
For more details, see Applicative Programming with Effects, by Conor McBride and Ross Paterson.
Applicative functors
class Functor f => Applicative f where
Source
A functor with application, providing operations to
embed pure expressions (pure), and
sequence computations and combine their results (<*>).
A minimal complete definition must include implementations of these functions satisfying the following laws:
base-4.7.0.1: Basic librari
hackage.haskell.org/package/base-4.7.0.1/docs/d...
base-4.7.0.1: Basic libraries
Contents | Index
G H I J K L M N O P Q R S T U V W X Y Z : ! $ % & * + . / < = >
~ All
Index - F
F# GHC.Exts
fail Control.Monad, Prelude, Control.Monad.Instances
False Data.Bool, Prelude
Fd
1 (Type/Class) System.Posix.Types
2 (Data Constructor) System.Posix.Types
FdKey GHC.Event
fdToHandle GHC.IO.Handle.FD
fdToHandle' GHC.IO.Handle.FD
fetchAddIntArray# GHC.Exts
FieldFormat
1 (Type/Class) Text.Printf
2 (Data Constructor) Text.Printf
FieldFormatter Text.Printf
FileID System.Posix.Types
FileMode System.Posix.Types
Control/Applicative.hs
hackage.haskell.org/package/base-4.7.0.1/doc...
class Functor f => Applicative f where
-- | Lift a value.
pure :: a -> f a
-- | Sequential application.
(<*>) :: f (a -> b) -> f a -> f b
-- | Sequence actions, discarding the value of the first argument.
(*>) :: f a -> f b -> f b
(*>) = liftA2 (const id)
-- | Sequence actions, discarding the value of the second argument
(<*) :: f a -> f b -> f a
(<*) = liftA2 const
-- | A monoid on applicative functors.
--

Demo

- https://www.youtube.com/watch?v=tF6lhqAR_L0
- https://www.youtube.com/watch?v=z4X6XHFII5w
- https://www.youtube.com/watch?v=duKYImvXL2o

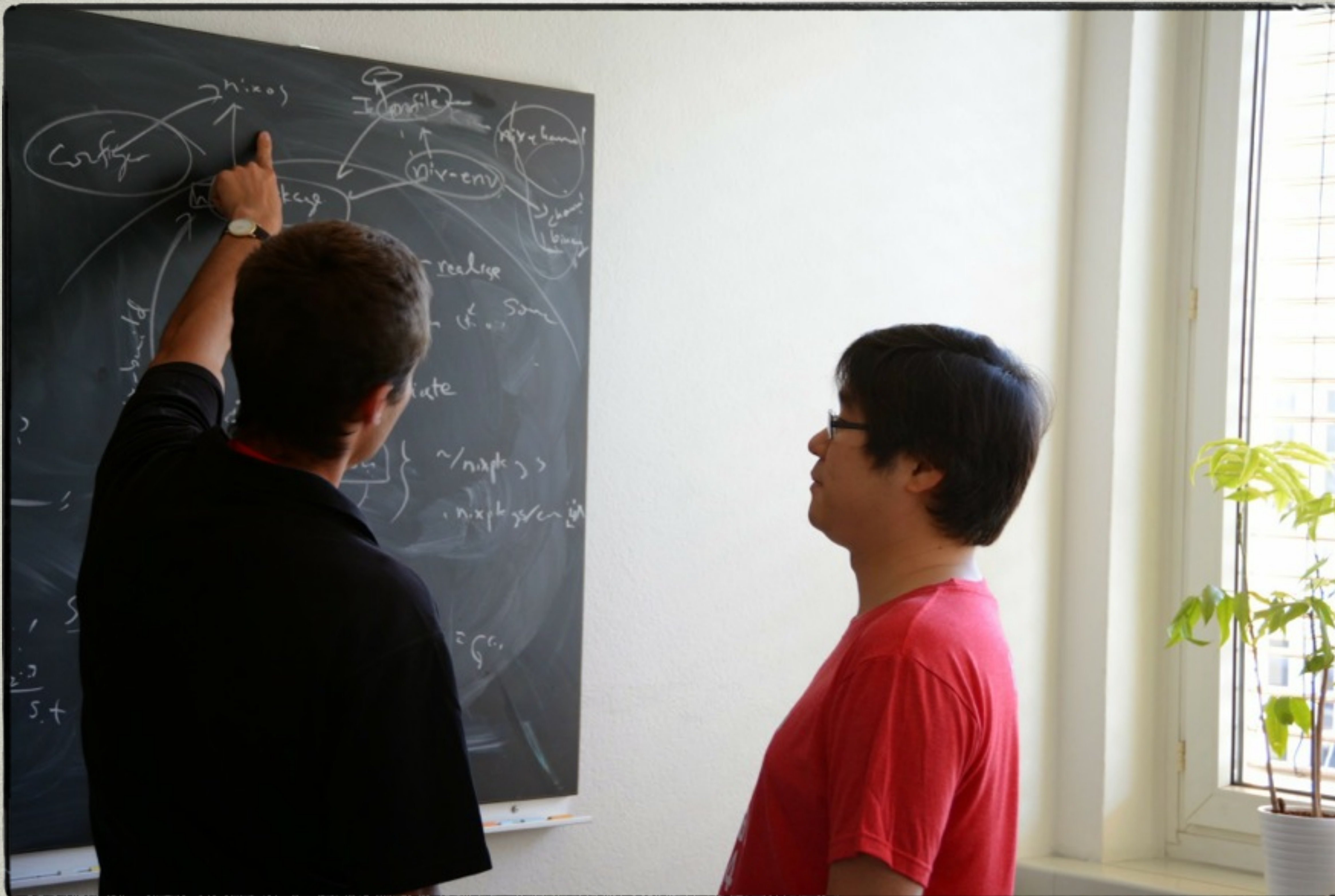
Summary

- Community
- Advantages
- Conclusion

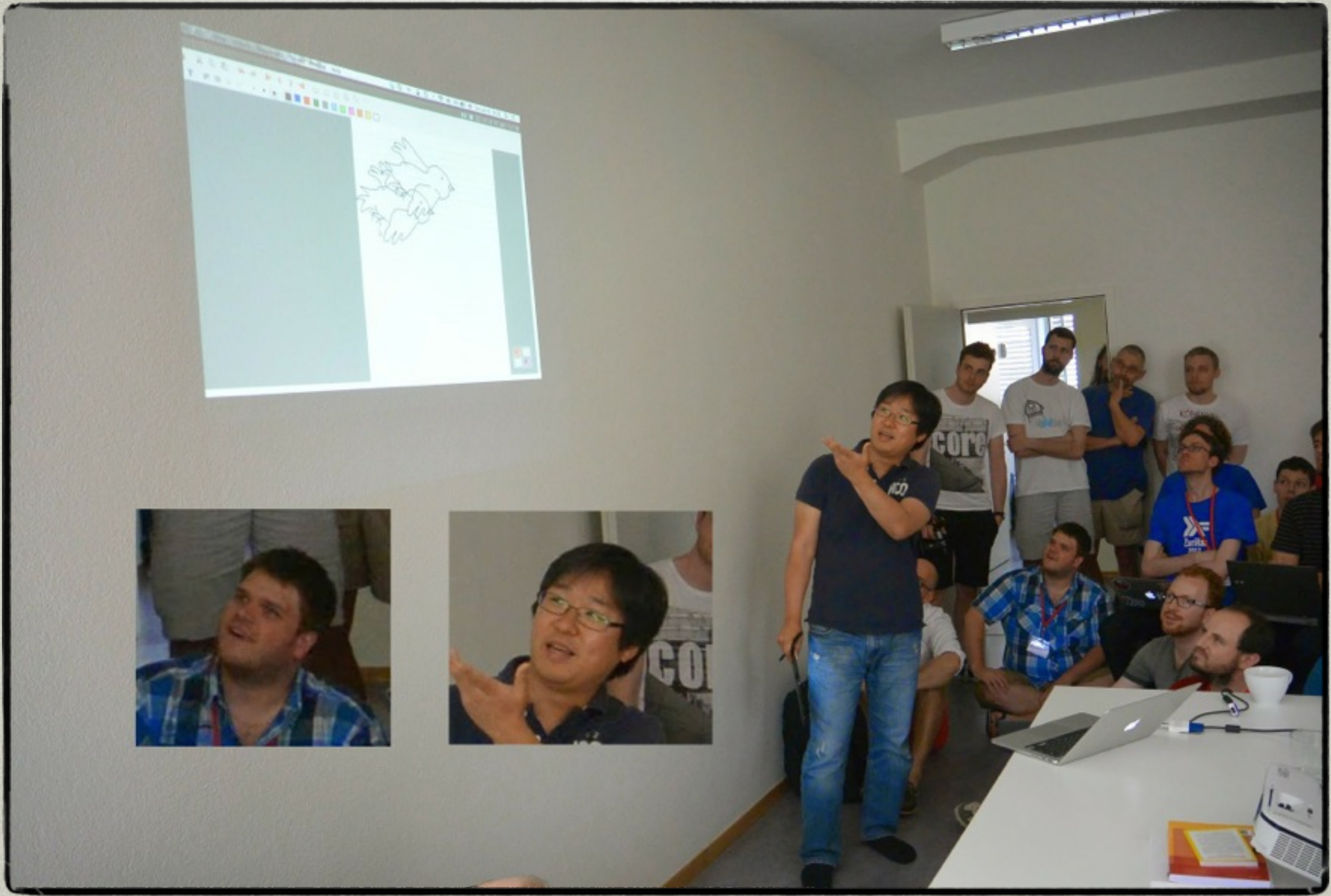
Community

- The real power of Haskell
- Active, Academic, Approachable :)
- Many things are going on:
  - social media, meetups, hackathons, conferences, schools, IRC, mailing-lists…
  - For reference, see also the last slide

ZuriHac 2014: before :(



ZuriHac 2014: during :|

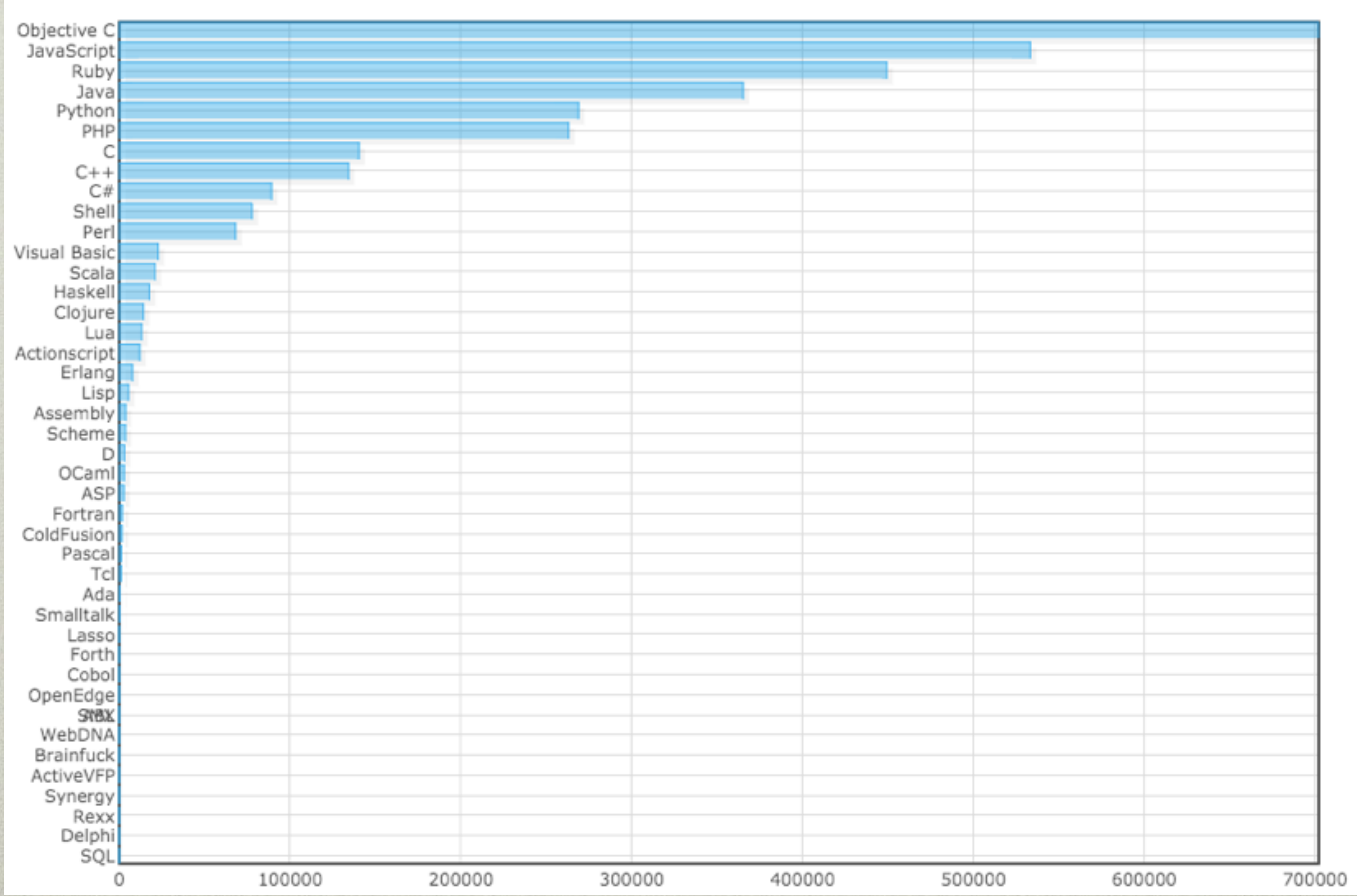


ZuriHac 2014: After :)

ZuriHac 2014: Celebrities

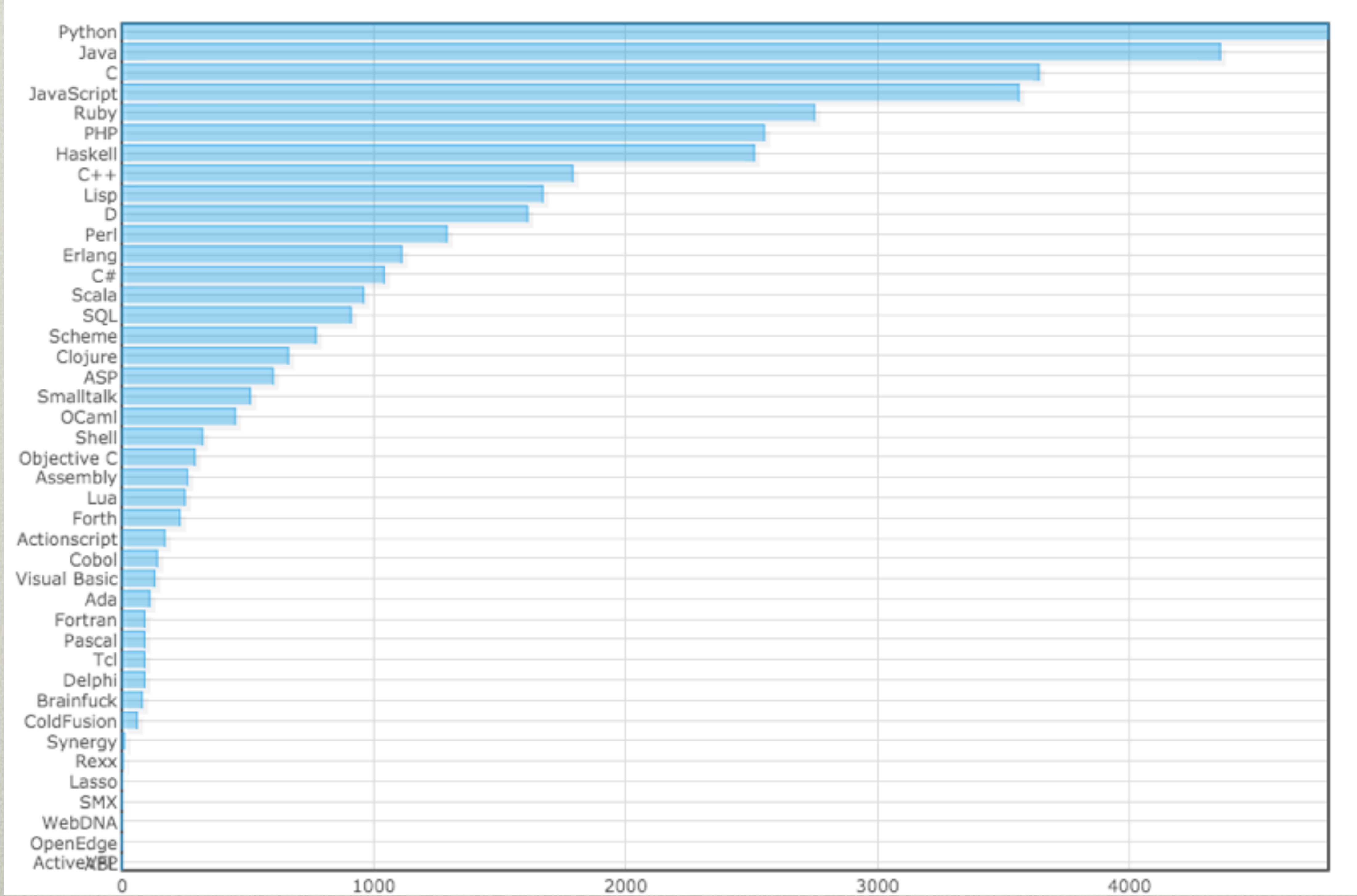
Simon Marlow (GHC), Edward Kmett (Lens)

Popularity - usage (github)



Source: langpop.com

Popularity - talked about (reddit)



Source: langpop.com

Geneva Haskell Group

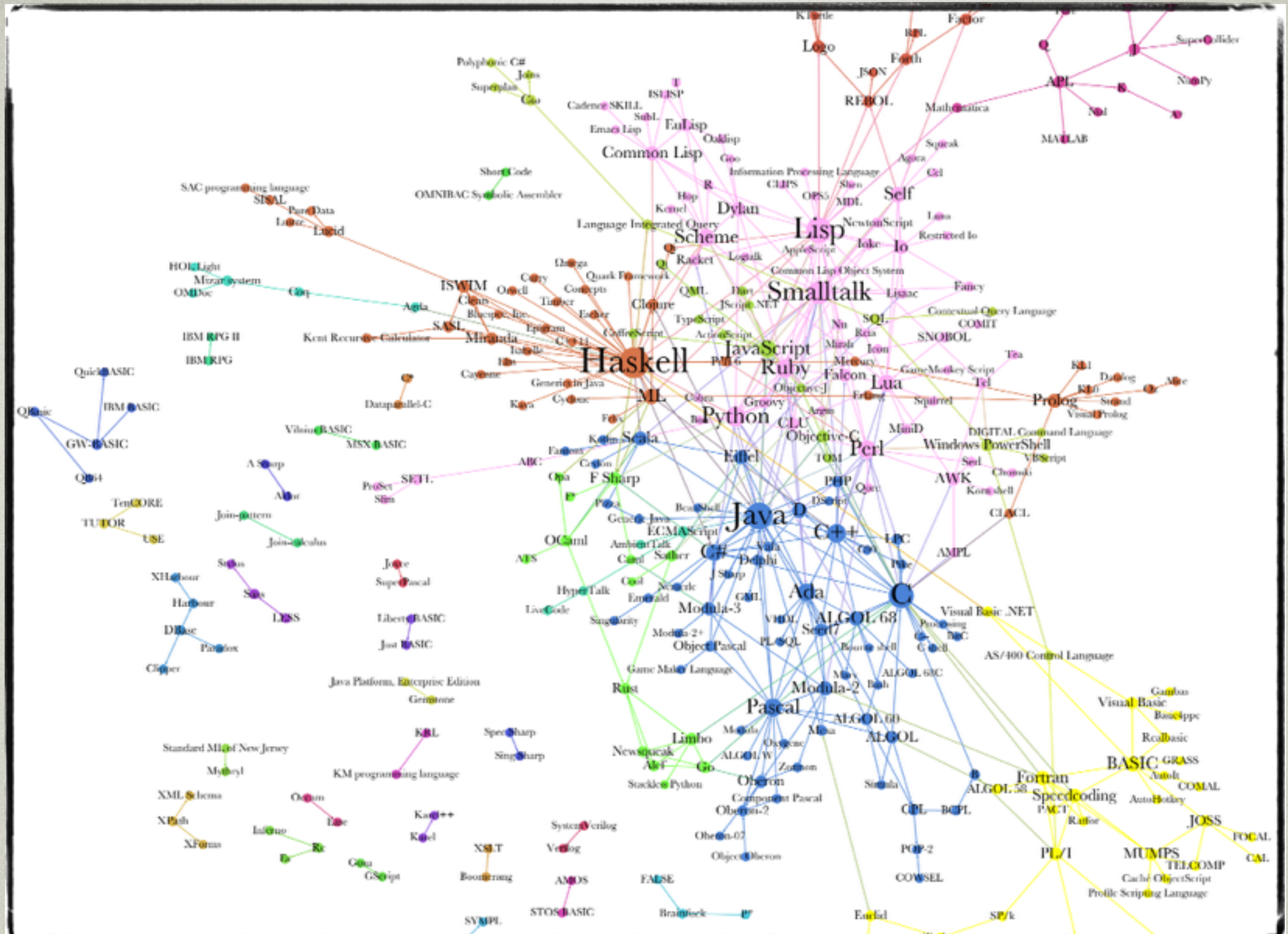
- Founded: 13 June, 2014
- ~40 members, **lot of beginners**
- Weekly meet-ups
  - Lectures, talks, demos, live-coding, coding puzzles, free coding, **beer**
- Core team: 5-7 people
- Typical attendance: 5-15 people

Haskell Advantages

- Excellent in abstraction
- Concise, Elegant and Modular code
- Explicit Control of Side Effects
- Concurrent and Parallel Programming
  - Software Transactional Memory (STM) with type safety
  - Implicit parallelism in pure functions
  - Explicit parallelism
  - Green threads

Haskell Influence

- Base syntax for newer languages:
  - Agda, Idris, Elm, Purescript,…
- Lazy evaluation : Flexible ordering of computation (C# ..)
- Generics: Parametric Polymorphism
  - Java Generics, C++1x library design
- Monad, List Comprehension
  - C# LINQ, F# Computation Expression ..
  - Python List Comprehension



Programming Language Influence

from Wikipedia

Source: http://brendangriffen.com/gow-programming-languages/

Conclusion

- Haskell is useful!
- Functional Programming is worth to learn!
- Haskell has a very active community!
- Coding is a fun!

Useful links

- Meet-ups, user groups, community, HacBerlin2014
  - http://www.meetup.com/Geneva-Haskell-Group
  - http://www.haskell.org/haskellwiki/User_groups
  - http://www.haskell.org/pipermail/haskell-cafe/
  - http://www.haskell.org/haskellwiki/HacBerlin2014
- General information
  - http://www.haskell.org/haskellwiki/Haskell
- Learning material
  - Haskell is Useless (YouTube): http://goo.gl/bBtLda :))
  - Learn You A Haskell: http://learnyouahaskell.com/chapters
  - Real World Haskell: http://book.realworldhaskell.org/read/
  - http://www.haskell.org/haskellwiki/Typeclassopedia

Thank you!