Why learn Haskell?

Keegan McAllister

SIPB Cluedump

October 11, 2011

Composability

The central challenge of programming (Dijkstra, 2000):

How not to make a mess of it

It helps to build programs from composable parts

- Combine in flexible yet well-defined ways

Haskell is a language uniquely suited to this goal

Functions

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Whitespace for function application

`f x` not `f(x)`

Parentheses only for grouping

Lists

A list is either

- the empty list `[]`, or
- a first element `x` and a remaining list `xs`, written `(x:xs)`

Use these patterns to build and to inspect lists

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

Declarative programming

Describe results, not individual steps

```haskell
-- merge two sorted lists
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
  | x < y      = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

Equational reasoning

Functions on functions

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

(f . g) x = f (g x)
```

Reason by substituting equals for equals

```
map f (map g xs)  ≡  map (f . g) xs
map f . map g     ≡  map (f . g)
```

Lazy evaluation

Expressions aren't evaluated until result is needed

```
-- two infinite lists
evens = 0 : map (+1) odds
odds  = map (+1) evens
```

```
GHCi> take 16 evens
[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]
```

Laziness separates concerns: example 1

```
minimum = head . sort
```

$$
\begin{aligned}
\texttt{sort} &\in O(n \log n) \\
\texttt{minimum} &\in O(n)
\end{aligned}
$$

…for careful `sort` implementations

Laziness separates concerns: example 2

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

True  || x = True
False || x = x

or = foldr (||) False

any p = or . map p
```

```
GHCi> any (> 7) [1..]  -- an infinite list
True
```

Static types

Types exist at compile time; writing them is optional

```
not :: Bool -> Bool

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

Types catch mistakes but stay out of your way otherwise

Algebraic data

Define and inspect data by enumerating cases

```
data Tree
  = Leaf
  | Node Int Tree Tree

depth :: Tree -> Int
depth Leaf = 0
depth (Node n x y)
  = 1 + max (depth x) (depth y)
```

Pattern-matching is composable

Patterns can be nested

```
rotate :: Tree -> Tree

rotate (Node m (Node n x y) z)
       = Node n x (Node m y z)

rotate t = t
```

Parametric polymorphism

```
data Tree t
  = Leaf
  | Node t (Tree t) (Tree t)

treeMap :: (a -> b) -> Tree a -> Tree b
```

Polymorphic type disallows hidden special cases

```
-- ok
treeMap f (Node v x y) = ...

-- error: not polymorphic!
treeMap f (Node [2,7] x y) = ...
```

Sharing immutable data

```
data Tree t
  = Leaf
  | Node t (Tree t) (Tree t)

insert x Leaf = Node x Leaf Leaf
insert x (Node y a b)
  | x < y      = Node y (insert x a) b
  | otherwise = Node y a (insert x b)
```

New tree shares nodes with old

- Great for lock-free concurrency

Embedded languages

Libraries can feel like specialized languages

```
tree :: Parser (Tree String)
tree = leaf <|> node where
  leaf = Leaf <$ char '.'
  node = Node <$> some alphaNum <* char '('
              <*> tree <*> tree <* char ')'
```

```
GHCi> parseTest tree "x(y(..).)"
Node "x" (Node "y" Leaf Leaf) Leaf
```

Power of embedded languages

Embedded languages use Haskell features for free

```
many    :: Parser a -> Parser [a]
satisfy :: (Char -> Bool) -> Parser Char
```

Grammar description for parser can use

- functions
- recursion
- lists and other data structures

IO in Haskell

IO is an imperative language embedded in Haskell

```
-- IO action
getChar :: IO Char

-- function returning IO action
putChar :: Char -> IO ()
```

An IO action is an ordinary first-class value
An inert description of IO which *could* be performed

Evaluation $\neq$ execution

Combining IO actions

Use result of one IO action to compute another

```
(>>=) :: IO a -> (a -> IO b) -> IO b

main =
  getLine >>= (\name ->
  putStrLn ("Hello " ++ name))
```

Special syntax is available:

```
main = do
  name <- getLine
  putStrLn ("Hello " ++ name)
```

First-class IO

Define your own control flow!

```
forever x = x >> forever x

for [] f = return ()
for (x:xs) f = do
  f x
  for xs f

for2 xs f = sequence_ (map f xs)

main = forever (for [1,2,3] print)
```

Example: scoped resources

```haskell
bracket
  :: IO a              -- acquire
  -> (a -> IO b)       -- release
  -> (a -> IO c)       -- do work
  -> IO c              -- result

withFile
  :: FilePath -> (Handle -> IO t) -> IO t
withFile name =
  bracket (openFile name WriteMode) hClose

main = withFile "foo.txt" (\h -> hPrint h 3)
```

Concurrency

Lightweight threads

```
forkIO :: IO () -> IO ThreadId
```

Message channels

```
newChan   :: IO (Chan a)
readChan  :: Chan a      -> IO a
writeChan :: Chan a -> a -> IO ()
```

Concurrency example

```
startLogger :: IO (String -> IO ())
startLogger = do
  chan <- newChan
  forkIO (forever
    (readChan chan >>= putStrLn))
  return (writeChan chan)

main :: IO ()
main = do
  lg <- startLogger
  lg "Hello, world!"
```

Chan is hidden; expose only what's needed

Software transactional memory

How do threads coordinate access to shared state?

- Locks are error-prone and don't compose

Transactions provide an alternative

- Build transactions the same way as IO actions
- Atomic execution is guaranteed

Building transactions

Example: transfer funds between accounts

```
transfer amount sender receiver = do
  -- read current balances
  senderBal   <- readTVar sender
  receiverBal <- readTVar receiver

  -- write new balances
  writeTVar sender   (senderBal   - amount)
  writeTVar receiver (receiverBal + amount)
```

Concurrent transfers would let you double-spend money!
Can't happen because this is all one transaction

Composing transactions

We can combine transactions:

```
sale cost buyer seller = do
 transfer 1    (goods seller) (goods buyer )
 transfer cost (money buyer ) (money seller)
```

Still a single transaction; still atomic

Running transactions

Run any transaction atomically

```
atomically :: STM a -> IO a

main = do
  ...
  atomically (sale 3 alice bob)
  ...
```

Transaction guarantees

Transactions have a different type from IO actions

```
atomically :: STM a -> IO a
```

So transactions can't

- affect the outside world
- run outside `atomically`

Lacking this property is why Microsoft's transactions for C# failed

Transaction failure

What if the sender has insufficient funds?

```
transfer amount sender receiver = do
  senderBal <- readTVar sender
  when (senderBal < amount)
    retry
  ...
```

Acts like immediate retry

Implementation is more efficient

Parallelism without concurrency

So Haskell supports a few approaches to threading

What about pure computation on multiple cores?

- Shouldn't need explicit threads at all

Pure parallelism

```
resS = map              complexFunction bigInput
resP = parMap rseq complexFunction bigInput
```

We know `resS` equals `resP`

- but `resP` might evaluate faster

Can place parallelism hints anywhere

- without changing results
- without fear of race conditions or deadlock

The real world

Haskell code looks nice...

but can we use it to solve real problems?

Commercial users

A niche language with many niches

- Amgen*: biotech simulations
- Bluespec: hardware design tools
- Eaton*: EDSL for hard realtime vehicle systems
- Ericsson: digital signal processing
- Facebook*: automated refactoring of PHP code
- Galois*: systems, crypto projects for NASA, DARPA, NSA
- Google*: managing virtual machine clusters
- Janrain: single sign-on through social media
- Lots of banks: ABN AMRO*, Bank of America, Barclays*, Credit Suisse*, Deutsche Bank*, Standard Chartered

*paper / talk / code available

Open-source applications in Haskell

xmonad: tiling window manager for X11

- Fast and flexible
- Great multi-monitor support
- Configured in Haskell, with seamless recompile

pandoc: markup format converter

- Markdown, HTML, LaTeX, Docbook, OpenDocument, ...
- Syntax highlighting, math rendering
- Used in making these slides

The Glorious Glasgow Haskell Compiler

GHC implements the Haskell language

- with many extensions

GHC produces optimized native-code executables

- directly or via LLVM

GHCi: interactive interpreter

GHC as a library: Haskell eval in your own app

GHC runtime system

One OS thread per CPU core

- Haskell threads are scheduled preemptively
- Spawn 100,000 threads on a modest system

Parallel generational garbage collector

- All OS threads GC at the same time

Special support for transactions, mutable arrays, finalizers

High-performance concurrent IO

You use threads and simple blocking IO

GHC implements with event-based IO: `select`, `epoll`, etc.

Don't turn your code inside-out!

Good performance with one thread per client:

- 10,000 HTTP / sec with 10,000 active clients*
- 17,000 HTTP / sec with 10,000 idle clients

---

*O'Sullivan and Tibell. "Scalable I/O Event Handling for GHC." *2010 ACM SIGPLAN Haskell Symposium*, pp. 103-108.

C foreign function interface

Calling C from Haskell is easy:

```
foreign import ccall sqrtf :: Float -> Float
main = print (sqrtf 2.0)
```

Full-featured:

- also call Haskell from C
- work with pointers, structs, arrays
- convert Haskell function ⟷ C function pointer

Making a high-level API is still hard!

Rewrite rules

Libraries can include rules for the optimizer

```
{-# RULES "myrule"
    forall f g xs.
      map f (map g xs) = map (f . g) xs
  #-}
```

Haskell tools

Besides compiling, we need to

- run tests
- benchmark and profile
- generate documentation
- manage library dependencies
- package and distribute our code

QuickCheck library

```haskell
sort :: [Int] -> [Int]

prop1 xs  =  sort (sort xs) == sort xs
prop2 xs  =  xs == sort xs
```

```
GHCi> quickCheck prop1
+++ OK, passed 100 tests.

GHCi> quickCheck prop2
*** Failed! Falsifiable (after 6 tests and 7 shrinks):
[1,0]
```

Test against properties or reference implementation

Test coverage: hpc

hpc.html

file:///tmp/hpc.html

module	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
module Main	100%	4/4		66%	4/6		91%	34/37	
Program Coverage Total	100%	4/4		66%	4/6		91%	34/37	

```
import Data.List
import Test.QuickCheck

k `divides` n
  | k <= 0      = False
  | otherwise = (n `mod` k) == 0

factor 1 = []
factor n = case find (`divides` n) [2..] of
  Just k  -> k : factor (n `div` k)
  Nothing -> error "impossible"

prop :: Integer -> Property
prop n = (n > 0) ==> product (factor n) == n

main = quickCheck prop
```

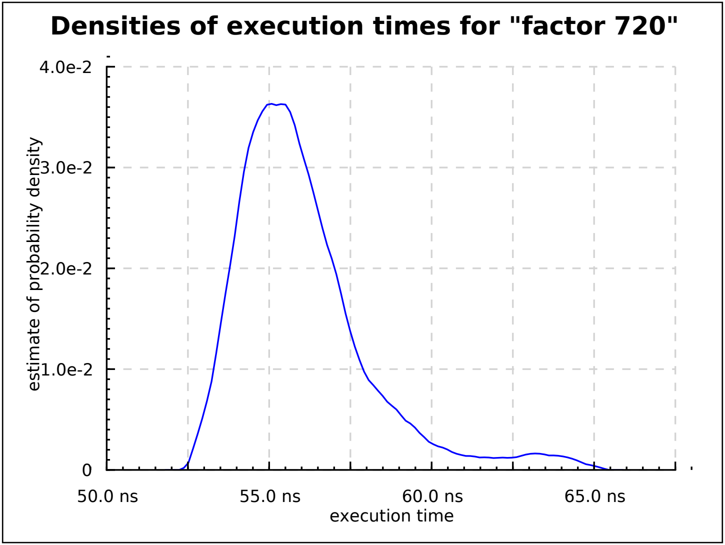
Benchmarking: Criterion

```haskell
import Criterion.Main
main = defaultMain [bench "factor 720"
                    (whnf factor 720)]
```

```
estimating cost of a clock call...
mean is 88.16269 ns (43 iterations)
found 4 outliers among 43 samples (9.3%)

benchmarking factor 720
mean: 56.01964 ns, lb 55.67899 ns, ub 56.46515 ns,
  ci 0.950
```

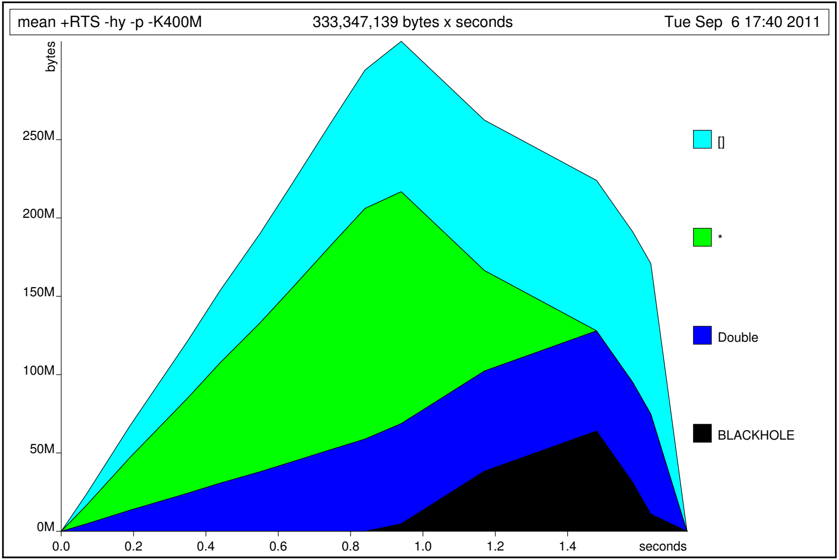
Criterion's density estimation



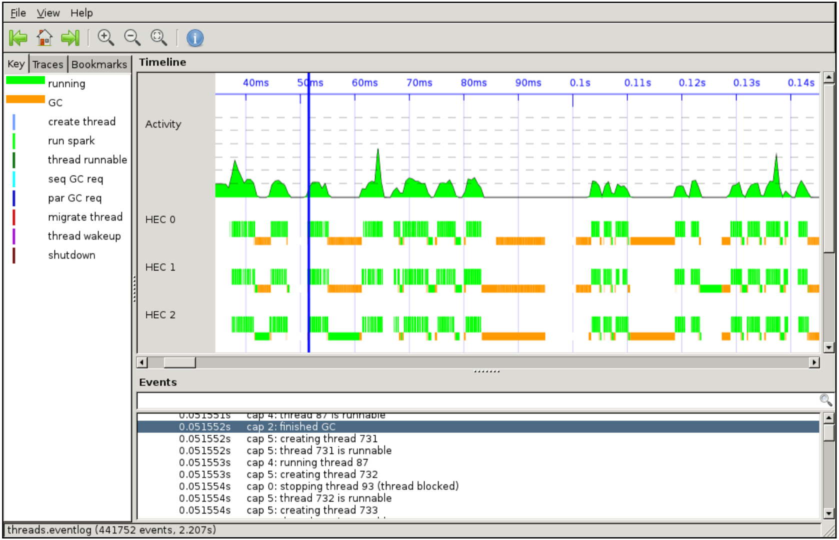
Time profiling

COST CENTRE	individual %time	individual %alloc	inherited %time	inherited %alloc
MAIN	0.0	0.0	100.0	100.0
CAF:main	0.0	0.0	0.0	0.0
CAF:main	0.0	0.0	98.6	97.6
main	44.4	30.7	98.6	97.6
keepNew	1.4	3.6	1.4	3.6
keepOld	4.2	3.6	4.2	3.6
diff	0.0	10.7	48.6	59.8
number	1.4	2.5	13.9	30.7
zipLS	12.5	28.2	12.5	28.2
solveLCS	0.0	0.0	34.7	18.4
longestIncreasing	0.0	0.0	0.0	0.0
unique	34.7	18.4	34.7	18.4

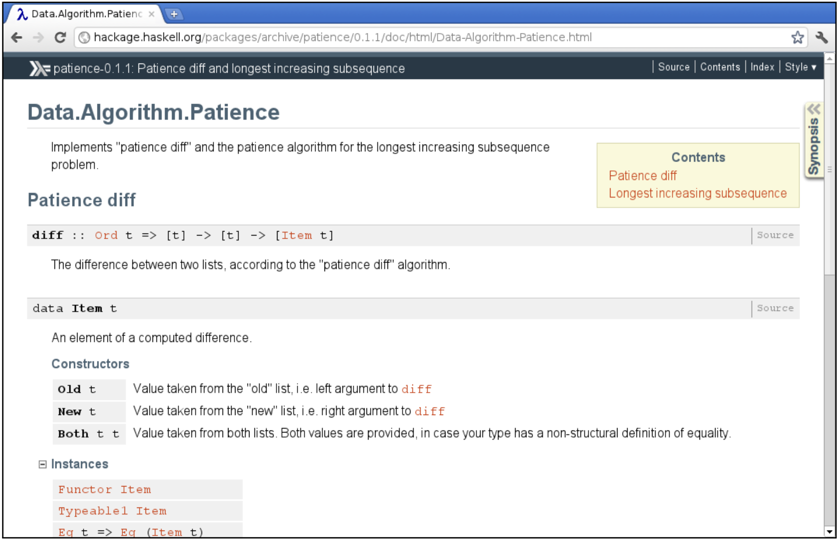
Heap profiling: hp2ps



Threadscope



Documentation: Haddock


Data.Algorithm.Patienc

hackage.haskell.org/packages/archive/patience/0.1.1/doc/html/Data-Algorithm-Patience.html

patience-0.1.1: Patience diff and longest increasing subsequence

Source | Contents | Index | Style

Data.Algorithm.Patience

Implements "patience diff" and the patience algorithm for the longest increasing subsequence problem.

Contents

Patience diff

Longest increasing subsequence

Synopsis

Patience diff

```
diff :: Ord t => [t] -> [t] -> [Item t]
```

Source

The difference between two lists, according to the "patience diff" algorithm.

```
data Item t
```

Source

An element of a computed difference.

Constructors

`Old t`	Value taken from the "old" list, i.e. left argument to `diff`
`New t`	Value taken from the "new" list, i.e. right argument to `diff`
`Both t t`	Value taken from both lists. Both values are provided, in case your type has a non-structural definition of equality.

Instances

```
Functor Item
Typeable1 Item
Eq t => Eq (Item t)
```

Cabal

Cabal will

- compile your code
- generate a source tarball
- handle a mixture of Haskell and C
- track installed packages and dependencies
- hyperlink documentation between packages

Cabal file

```
name:          patience
version:       0.1.1
license:       BSD3
synopsis:      Patience diff algorithm
maintainer:    Keegan McAllister

library
  exposed-modules:  Data.Algorithm.Patience
  ghc-options:      -Wall
  build-depends:
      base >= 3 && < 5
    , containers >= 0.2
```

Using Cabal

```
patience-0.1.1$ cabal install
Resolving dependencies...
Building patience-0.1.1...
[1 of 1] Compiling Data.Algorithm.Patience

Registering patience-0.1.1...
Running Haddock for patience-0.1.1...
Installing library in
  ~/.cabal/lib/patience-0.1.1/ghc-7.0.4
Updating documentation index
  ~/.cabal/share/doc/index.html
```

Hackage: the Haskell package repository

`http://hackage.haskell.org`

- Over 3,400 packages
- Most have permissive license (BSD or MIT)
- Dozens of uploads per day
- Hyperlinked documentation on the Web
- Cabal can download and install

Hoogle and Hayoo: search Hackage by type

Bad parts of the language

Standard Haskell changes slowly; extensions are

- not fully specified
- subject to change and deprecation

Some clear mistakes in the design

- e.g. monomorphism restriction

Records and modules are simplistic

- compare to OCaml

Ad-hoc overloading has annoying limitations

Trouble at runtime

Reasoning about performance is very hard

Magic optimizations are brittle

Lots of time is spent in garbage collection

- other threads blocked

Hard to track down run-time errors

Library woes

Which of those 3,400 packages are usable?

Too much choice

- Do your text type, parser lib, IO iterator fit together?

Standard library has gaps and avoidable flaws

Best practices are still evolving

Obstacles to learning

Up-front effort for long-term gain

- un-learning old habits

Frustrating: easy things are hard

Many articles are confusing or plain wrong

- "a monad is like a burrito"

Where to learn Haskell

Books (free online)

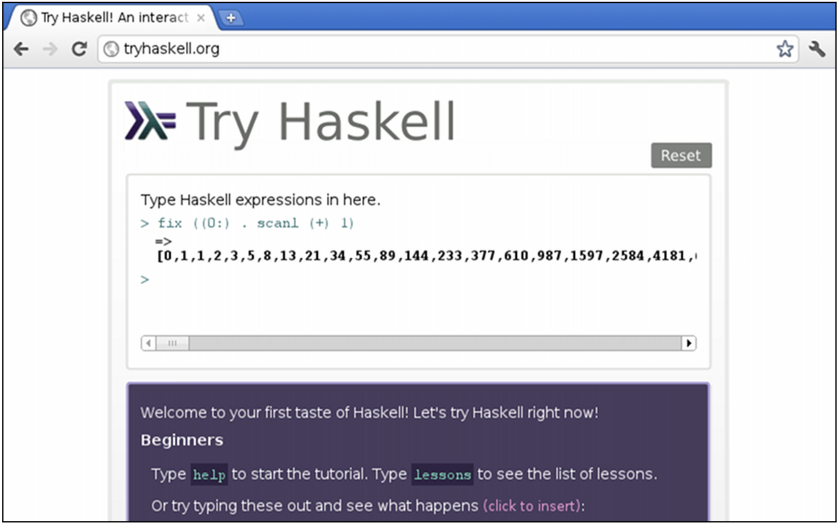
- *Learn You a Haskell For Great Good* by Lipovača
- *Real World Haskell* by O'Sullivan, Stewart, Goerzen

Real-time help from experts

- Freenode IRC `#haskell`: 750 users
- Stack Overflow: 4,000 questions asked

Reddit, blogs, mailing lists, HaskellWiki, academic papers, . . .

Try Haskell!


Try Haskell! An interact
tryhaskell.org
Try Haskell
Reset
Type Haskell expressions in here.
> fix ((0:) . scanl (+) 1)
=>
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,
>
Welcome to your first taste of Haskell! Let's try Haskell right now!
Beginners
Type help to start the tutorial. Type lessons to see the list of lessons.
Or try typing these out and see what happens (click to insert):

#haskell's lambdabot

```
<kmc> @run fix ((0:) . scanl (+) 1)
<lambdabot> [0,1,1,2,3,5,8,13,21,34,55,89,144,233,...

<kmc> @pl \x -> h (f x) (g x)
<lambdabot> liftM2 h f g

<kmc> @djinn ((a, b) -> c) -> a -> b -> c
<lambdabot> f a b c = a (b, c)

<kmc> @quote few.dozen
<lambdabot> _pizza_ says: i think Haskell is
  undoubtedly the world's best programming
  language for discovering the first few dozen
  numbers in the Fibonacci sequence over IRC
```

Haskell Platform: batteries included

Haskell Platform

GHC bundled with blessed tools and libraries

- HTTP, CGI, OpenGL, regex, parsers, unit testing

Available for Windows, Mac OS X, Linux, FreeBSD

Packaged in Ubuntu, Debian, Fedora, Arch, Gentoo

`http://haskell.org/platform`

In conclusion...

Haskell lets you build software out of flexible parts which combine in well-defined ways.

Start learning and get

- new ideas right away
- a practical tool later

Use those ideas in other languages, too

Questions?

Slides available at `http://t0rch.org`