Why learn Haskell?

Keegan McAllister

SIPB Cluedump

October 11, 2011
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
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

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

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

Reason by substituting equals for equals

\[
\begin{align*}
\text{map } f \; (\text{map } g \; xs) & \equiv \text{map } (f \; . \; g) \; xs \\
\text{map } f \; . \; \text{map } g & \equiv \text{map } (f \; . \; g)
\end{align*}
\]
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
```

\[ \text{sort} \in O(n \log n) \]
\[ \text{minimum} \in O(n) \]

...for careful sort implementations
Laziness separates concerns: example 2

```haskell
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
```
Types exist at compile time; writing them is optional

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool}
\]

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

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

Types catch mistakes but stay out of your way otherwise
Define and inspect data by enumerating cases

```haskell
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

\[
\text{rotate} :: \text{Tree} \rightarrow \text{Tree}
\]

\[
\text{rotate } (\text{Node } m (\text{Node } n x y) z) = \text{Node } n x (\text{Node } m y z)
\]

\[
\text{rotate } t = t
\]
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

```haskell
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
Embedded languages use Haskell features for free

\[
\begin{align*}
\text{many} & : \text{Parser } a \rightarrow \text{Parser } [a] \\
\text{satisfy} & : (\text{Char} \rightarrow \text{Bool}) \rightarrow \text{Parser } \text{Char}
\end{align*}
\]

Grammar description for parser can use

- functions
- recursion
- lists and other data structures
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 ≠ execution**
Combining IO actions

Use result of one IO action to compute another

\[(\gg\gg=) \ :: \ IO \ a \ \rightarrow \ (a \ \rightarrow \ IO \ b) \ \rightarrow \ IO \ b\]

main =
    getName \gg\gg= (\name \rightarrow
    putStrLn ("Hello " ++ name))

Special syntax is available:

main = do
    name <- getName
    putStrLn ("Hello " ++ name)
First-class IO

Define your own control flow!

\[
\begin{align*}
\text{forever } x &= x >> \text{forever } x \\
\text{for } [] f &= \text{return } () \\
\text{for } (x:xs) f &= \text{do} \\
&\quad f x \\
&\quad \text{for } xs f \\
\text{for2 } xs f &= \text{sequence\_} (\text{map } f \ xs) \\
\text{main} &= \text{forever } (\text{for } [1,2,3] \ \text{print})
\end{align*}
\]
Example: scoped resources

```
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

\[
\text{forkIO :: IO ()} \rightarrow \text{IO ThreadId}
\]

Message channels

\[
\begin{align*}
\text{newChan :: IO (Chan a)} \\
\text{readChan :: Chan a} & \rightarrow \text{IO a} \\
\text{writeChan :: Chan a} & \rightarrow a \rightarrow \text{IO ()}
\end{align*}
\]
Concurrency example

\[
\begin{align*}
\text{startLogger :: IO (String -> IO ())} \\
\text{startLogger = do} \\
\quad \text{chan <- newChan} \\
\quad \text{forkIO (forever} \\
\quad \quad (\text{readChan chan >>= putStrLn}) \\
\quad \text{return (writeChan chan)} \\
\text{main :: IO ()} \\
\text{main = do} \\
\quad \text{lg <- startLogger} \\
\quad \text{lg "Hello, world!"}
\end{align*}
\]

Chan is hidden; expose only what’s needed
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
Example: transfer funds between accounts

```haskell
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
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
Run any transaction atomically

\[
\text{atomically :: STM } a \rightarrow \text{ IO } a
\]

\[
\text{main } = \text{ do}
\]
\[
\text{...}
\]
\[
\text{... atomically (sale 3 alice bob)}
\]
\[
\text{...}
\]
Transactions have a different type from IO actions

\[
\text{atomically} :: \text{STM } a \rightarrow \text{IO } a
\]

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?

```haskell
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

\[
\begin{align*}
\text{resS} &= \text{map} \quad \text{complexFunction} \quad \text{bigInput} \\
\text{resP} &= \text{parMap} \quad \text{rseq} \quad \text{complexFunction} \quad \text{bigInput}
\end{align*}
\]

We know \(\text{resS}\) equals \(\text{resP}\)

- but \(\text{resP}\) 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

*paper / talk / code available

Keegan McAllister

Why learn Haskell?
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, \text{\LaTeX}, Docbook, OpenDocument, \ldots
- Syntax highlighting, math rendering
- Used in making these slides
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

---

C foreign function interface

Calling C from Haskell is easy:

```haskell
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 $\leftrightarrow$ 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

\[
\text{sort :: [Int] -> [Int]} \\
\text{prop1 \( xs \) = sort (sort \( xs \)) == sort \( xs \)} \\
\text{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

<table>
<thead>
<tr>
<th>module</th>
<th>Top Level Definitions</th>
<th>Alternatives</th>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% covered / total</td>
<td>% covered / total</td>
<td>% covered / total</td>
</tr>
<tr>
<td>module Main</td>
<td>100% 4/4</td>
<td>66% 4/6</td>
<td>91% 34/37</td>
</tr>
<tr>
<td>Program Coverage Total</td>
<td>100% 4/4</td>
<td>66% 4/6</td>
<td>91% 34/37</td>
</tr>
</tbody>
</table>

```haskell
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
```
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
### Time profiling

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>individual</th>
<th>inherited</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%time</td>
<td>%alloc</td>
</tr>
<tr>
<td>MAIN</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF:main</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>main</td>
<td>44.4</td>
<td>30.7</td>
</tr>
<tr>
<td>keepNew</td>
<td>1.4</td>
<td>3.6</td>
</tr>
<tr>
<td>keepOld</td>
<td>4.2</td>
<td>3.6</td>
</tr>
<tr>
<td>diff</td>
<td>0.0</td>
<td>10.7</td>
</tr>
<tr>
<td>number</td>
<td>1.4</td>
<td>2.5</td>
</tr>
<tr>
<td>zipLS</td>
<td>12.5</td>
<td>28.2</td>
</tr>
<tr>
<td>solveLCS</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>longestIncreasing</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>unique</td>
<td>34.7</td>
<td>18.4</td>
</tr>
</tbody>
</table>

Keegan McAllister

Why learn Haskell?
Heap profiling: hp2ps

Keegan McAllister

Why learn Haskell?
Data.Algorithm.Patience

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

Patience diff

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

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

```
data Item t
```

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 will

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

(a,b) -> (b,a)

MPS.Light.swap :: (a, b) -> (b, a)
mps No description. Source

Data.Tuple.HT.swap :: (a, b) -> (b, a)
utility-ht No description. Source

Air.Light.swap :: (a, b) -> (b, a)
air No description. Source

FRP.Yampa.swap :: (a, b) -> (b, a)
Yampa No description. Source

Text.XML.HXT.DOM.Util.swap :: (a, b) -> (b, a)
hxt No description. Source

Top 15 Modules
FRP 4
Data 2
Text 1
Synthesizer 1
Prelude 1
Nettle 1
Music 1
MPS 1
Lava 1
Generics 1
Extension 1
Air 1

Top 15 Packages
Yampa 2
Animas 2
utility-ht 1
synthesizer 1
scyther-proof 1
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
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!

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):
<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
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