A TASTE OF HASKELL

Simon Peyton Jones
Microsoft Research

A tutorial presented at the O'Reilly Open Source Convention, July 2007

Video of this tutorial (3 hrs)
http://research.microsoft.com/~simonpj/papers/haskell-tutorial
What is Haskell?

- Haskell is a programming language that is
  - purely functional
  - lazy
  - higher order
  - strongly typed
  - general purpose
Why should I care?

- Functional programming will make you think differently about programming
  - Mainstream languages are all about state
  - Functional programming is all about values
- Whether or not you drink the Haskell Kool-Aid, you'll be a better programmer in whatever language you regularly use
"Most research languages"

Practitioners

Geeks

The quick death

1yr  5yr  10yr  15yr
Successful research languages

The slow death
C++, Java, Perl, Ruby

Threshold of immortality

The complete absence of death

The complete absence of death
“Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out” (blog Apr 2007)

“I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions” (blog Mar 2007)

The second life?
xmonad is an X11 tiling window manager written entirely in Haskell.
Why I'm using xmonad

Because it’s

- A real program
- of manageable size
- that illustrates many Haskell programming techniques
- is open-source software
- is being actively developed
- by an active community
<table>
<thead>
<tr>
<th></th>
<th>Code</th>
<th>Comments</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>metacity</td>
<td>&gt;50k</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>ion3</td>
<td>20k</td>
<td>7k</td>
<td>C</td>
</tr>
<tr>
<td>larswm</td>
<td>6k</td>
<td>1.3k</td>
<td>C</td>
</tr>
<tr>
<td>wmii</td>
<td>6k</td>
<td>1k</td>
<td>C</td>
</tr>
<tr>
<td>dwm 4.2</td>
<td>1.5k</td>
<td>0.2k</td>
<td>C</td>
</tr>
<tr>
<td>xmonad 0.2</td>
<td>0.5k</td>
<td>0.7k</td>
<td>Haskell</td>
</tr>
</tbody>
</table>

"Manageable size"
Inside xmonad

- X11
- Events (mouse, kbd, client)
- Window placement

FFI
- Configuration data
- Layout algorithm

State machine

Session state
The window stack

module Stack( Stack, insert, swap, ... ) where

import Graphics.X11( Window )

type Stack = ...

insert :: Window -> Stack
-- Newly inserted window has focus
insert = ...

swap :: Stack -> Stack
-- Swap focus with next
swap = ...

A ring of windows
One has the focus

Export list

Define new types

Import things defined elsewhere

Specify type of insert

Comments
The window stack

Stack should not exploit the fact that it’s a stack of windows

module Stack( Stack, insert, swap, ...) where

    type Stack w = ...

    insert :: w -> Stack w
    -- Newly inserted window has focus
    insert = ...

    swap :: Stack w -> Stack w
    -- Swap focus with next
    swap = ...

No import any more

A stack of values of type w

Insert a 'w' into a stack of w's
The window stack

A list takes one of two forms:
- [], the empty list
- \((w:ws)\), a list whose head is \(w\), and tail is \(ws\)

Functions are defined by pattern matching

\[
\text{swap} :: \text{Stack } w \rightarrow \text{Stack } w
\]

\[
\begin{align*}
\text{swap } [] & = [] \\
\text{swap } (w : []) & = w : [] \\
\text{swap } (w1 : w2 : ws) & = w2 : w1 : ws
\end{align*}
\]

The type \([w]\) means “list of \(w\)”

The ring above is represented \([c,d,e,...,a,b]\)

\(w1:w2:ws\) means \(w1 : (w2 : ws)\)
Syntactic sugar

\[
\begin{align*}
\text{swap} & \; [\;] \; = \; [\;] \\
\text{swap} & \; [w] \; = \; [w] \\
\text{swap} & \; (w1:w2:ws) \; = \; w2:w1:ws
\end{align*}
\]

\[
\begin{align*}
\text{swap} & \; (w1:w2:ws) \; = \; w2:w1:ws \\
\text{swap} & \; ws \; = \; ws
\end{align*}
\]

\[
\text{swap} \; ws \; = \; \text{case} \; ws \; \text{of}
\]

\[
\begin{align*}
[\;] & \rightarrow [\;] \\
[w] & \rightarrow [w] \\
(w1:w2:ws) & \rightarrow w2:w1:ws
\end{align*}
\]

Equations are matched top-to-bottom

[a,b,c] means a:b:c:[]
Running Haskell

- **Download:**
  - ghc: [http://haskell.org/ghc](http://haskell.org/ghc)
  - Hugs: [http://haskell.org/hugs](http://haskell.org/hugs)

- **Interactive:**
  - ghci Stack.hs
  - hugs Stack.hs

- **Compiled:**
  - ghc -c Stack.hs

Demo ghci
Rotating the windows

A ring of windows
One has the focus

focusNext :: Stack -> Stack
focusNext (w:ws) = ws ++ [w]
focusnext [] = []

Pattern matching forces us to think of all cases

Type says “this function takes two arguments, of type [a], and returns a result of type [a]”

(++) :: [a] -> [a] -> [a]
-- List append; e.g. [1,2] ++ [4,5] = [1,2,4,5]

Definition in Prelude (implicitly imported)
Recursion

List append; e.g. [1,2] ++ [4,5] = [1,2,4,5]

[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

Execution model is simple rewriting:

[1,2] ++ [4,5]
= (1:2:[]) ++ (4:5:[])
= 1 : (2:[]) ++ (4:5:[]) = 1 : 2 : ([] ++ (4:5:[])) = 1 : 2 : 4 : 5 : []
Rotating backwards

**Function application**
by mere juxtaposition

- **focusPrev :: Stack -> Stack**
  - **focusPrev ws = reverse (focusNext (reverse ws))**

- **reverse :: [a] -> [a]**
  - e.g. **reverse [1,2,3] = [3,2,1]**
  - **reverse [] = []**
  - **reverse (x:xs) = reverse xs ++ [x]**

**Function application**
binds more tightly than anything else:

- **(reverse xs) ++ [x]**
Function composition

\[
\text{focusPrev} :: \text{Stack} \to \text{Stack} \\
\text{focusPrev ws} = \text{reverse } (\text{focusNext } (\text{reverse ws}))
\]

can also be written

\[
\text{focusPrev} :: \text{Stack} \to \text{Stack} \\
\text{focusPrev} = \text{reverse} \ . \ \text{focusNext} \ . \ \text{reverse}
\]

\[
(f \ . \ g) \ x = f \ (g \ x)
\]
Function composition

(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)

(f \cdot g) \, x = f \, (g \, x)
Just testing
Just testing

- It’s good to write tests as you write code
- E.g. focusPrev undoes focusNext; swap undoes itself; etc

```haskell
module Stack where

...definitions...

-- Write properties in Haskell
type TS = Stack Int -- Test at this type

prop_focusNP :: TS -> Bool
prop_focusNP s = focusNext (focusPrev s) == s

prop_swap :: TS -> Bool
prop_swap s = swap (swap s) == s
```
Test interactively

Test.QuickCheck is simply a Haskell library (not a “tool”)

bash$ ghci Stack.hs
Prelude> :m +Test.QuickCheck

Prelude Test.QuickCheck> quickCheck prop_swap
+++ OK, passed 100 tests

Prelude Test.QuickCheck> quickCheck prop_focusNP
+++ OK, passed 100 tests

...with a strange-looking type

Prelude Test.QuickCheck> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
Test batch-mode

runHaskell Foo.hs <args>
runs Foo.hs, passing it <args>

A 25-line Haskell script

Look for “prop_” tests in here

bash$ runhaskell QC.hs Stack.hs
prop_swap: +++ OK, passed 100 tests
prop_focusNP: +++ OK, passed 100 tests
Things to notice
Things to notice...

No side effects. At all.

- A call to swap returns a new stack; the old one is unaffected.

  swap :: Stack w -> Stack w

  prop_swap s = swap (swap s) == s

- A variable ‘s’ stands for an immutable value, not for a location whose value can change with time. Think spreadsheets!
Purity makes the interface explicit

- Takes a stack, and returns a stack; that's all

- Takes a stack; may modify it; may modify other persistent state; may do I/O
Pure functions are easy to test

```prop_swap s = swap (swap s) == s```

- In an imperative or OO language, you have to
  - set up the state of the object, and the external state it reads or writes
  - make the call
  - inspect the state of the object, and the external state
  - perhaps copy part of the object or global state, so that you can use it in the postcondition
Types are everywhere

- Usual static-typing rant omitted...

- In Haskell, **types express high-level design**, in the same way that UML diagrams do; with the advantage that the type signatures are machine-checked

- Types are (almost always) optional: type inference fills them in if you leave them out

```haskell
swap :: Stack w -> Stack w
```
Improving the design
Changing focus moves the windows around: confusing!

```haskell
type Stack w = [w]
-- Focus is head of list

enumerate :: Stack w -> [w]
-- Enumerate the windows in layout order
enumerate s = s
```
Data type declaration

Constructor of the type

Represented as
MkStk [b,a] [c,d,e,f,g]

Want: a fixed layout, still with one window having focus

A sequence of windows
One has the focus

A sequence of windows
One has the focus

INVARIANT: if ‘right’ is empty, so is ‘left’

data Stack w = MkStk [w] [w] -- left and right resp
-- Focus is head of ‘right’ list
-- Left list is *reversed*
-- INVARIANT: if ‘right’ is empty, so is ‘left’
Want: a fixed layout, still with one window having focus

Represented as
\[
\text{MkStk} \left[ b, a \right] \left[ c, d, e, f, g \right]
\]

```
data Stack w = MkStk [w] [w] -- left and right resp
-- Focus is head of 'right' list
-- Left list is *reversed*
-- INVARIANT: if 'right' is empty, so is 'left'

enumerate :: Stack w -> [w]
enumerate (MkStack ls rs) = reverse ls ++ rs
```
Moving focus


data Stack w = MkStk [w] [w] -- left and right resp

focusPrev :: Stack w -> Stack w
focusPrev (MkStk (l:ls) rs) = MkStk ls (l:rs)
focusPrev (MkStk [] rs) = ...???

Nested pattern matching

Choices for left=[]:
• no-op
• move focus to end

We choose this one
data Stack w = MkStk [w] [w]  -- left and right resp
-- Focus is head of ‘right’

focusPrev :: Stack w \rightarrow Stack w
focusPrev (MkStk (l:ls) rs) = MkStk ls (l:rs)
focusPrev (MkStk [] rs) = case (reverse rs) of
  (l:ls) \rightarrow MkStk ls [l]

Note: I fixed a bug on this slide subsequent to presenting the tutorial
Pattern matching forces us to confront all the cases

Efficiency note: reverse costs $O(n)$, but that only happens once every $n$ calls to focusPrev, so amortised cost is $O(1)$. 

```haskell
data Stack w = MkStk [w] [w]  -- left and right resp
            -- Focus is head of ‘right’

focusPrev :: Stack w -> Stack w
focusPrev (MkStk (l:ls) rs) = MkStk ls (l:rs)
focusPrev (MkStk [] rs) = case (reverse rs) of
    (l:ls) -> MkStk ls [l]
    [] -> MkStk [] []
```

Warning: Pattern match(es) are non-exhaustive
In the case expression: ...
Patterns not matched: []
Data types

- A new **data type** has one or more constructors
- Each **constructor** has zero or more arguments

```haskell
data Stack w = MkStk [w] [w]
data Bool = False | True
data Colour = Red | Green | Blue
data Maybe a = Nothing | Just a
```

Built-in syntactic sugar for lists, but otherwise lists are just another data type

```haskell
data [a] = [] | a : [a]
```
Data types

- **Constructors are used:**
  - as a function to construct values ("right hand side")
  - in patterns to deconstruct values ("left hand side")

```
data Stack w = MkStk [w] [w]
data Bool = False | True
data Colour = Red | Green | Blue
data Maybe a = Nothing | Just a
```

```haskell
isRed :: Colour -> Bool
isRed Red   = True
isRed Green = False
isRed Blue  = False
```
Data types

- Data types are used
  - to describe data (obviously)
  - to describe “outcomes” or “control”

Data types are used

module Stack (focus, ...) where

focus :: Stack w -> Maybe w
-- Returns the focused window of the stack
-- or Nothing if the stack is empty
focus (MkStk _ []) = Nothing
focus (MkStk _ (w:_)) = Just w

module Foo where
import Stack

foo s = ...case (focus s) of
  Nothing -> ...do this in empty case...
  Just w  -> ...do this when there is a focus...

A bit like an exception...

...but you can’t forget to catch it
No “null-pointer dereference” exceptions
Data type abstraction

module Operations(...) where

import Stack( Stack, focusNext )

f :: Stack w -> Stack w
f (MkStk as bs) = ...

OK: Stack is imported

NOT OK: MkStk is not imported

module Stack(... where

data Stack w = MkStk [w] [w]

focusNext :: Stack w -> Stack w
focusNext (MkStk ls rs) = ...

Stack is exported, but not its constructors; so its representation is hidden
Haskell's module system

- Module system is merely a name-space control mechanism

- Compiler typically does lots of cross-module inlining

- Modules can be grouped into packages

```haskell
module X where
  import P
  import Q
  h = (P.f, Q.f, g)

module P(f,g) where
  import Z(f)
  g = ...

module Q(f) where
  f = ...

module Z where
  f = ...
```
Type classes
The need for type classes

```
delete :: Stack w -> w -> Stack w
-- Remove a window from the stack
```

- Can this work for ANY type w?

```
delete :: ∀w. Stack w -> w -> Stack w
```

- No - only for w’s that support equality

```
sort :: [a] -> [a]
-- Sort the list
```

- Can this work for ANY type a?

- No - only for a’s that support ordering
The need for type classes

serialise :: a -> String
-- Serialise a value into a string

- Only for w's that support serialisation

square :: n -> n
square x = x*x

- Only for numbers that support multiplication

- But square should work for any number that does; e.g. Int, Integer, Float, Double, Rational
Type classes

If a function works for every type that has particular properties, the type of the function says just that.

- delete :: Eq w => Stack w -> w -> Stack w

- If a function works for every type that has particular properties, the type of the function says just that.

  - sort :: Ord a => [a] -> [a]
  - serialise :: Show a => a -> String
  - square :: Num n => n -> n

Otherwise, it must work for any type whatsoever.

- reverse :: [a] -> [a]
- filter :: (a -> Bool) -> [a] -> [a]
square :: Num n => n -> n
square x = x*x

class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  ...etc..

instance Num Int where
  a + b = plusInt a b
  a * b = mulInt a b
  negate a = negInt a
  ...etc..

FORGET all you know about OO classes!
The class declaration says what the Num operations are

An instance declaration for a type T says how the Num operations are implemented on T’s
When you write this...

```haskell
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```haskell
square :: Num n -> n -> n
square d x = (*) d x x
```

The “Num n =>” turns into an extra value argument to the function.
It is a value of data type Num n.

A value of type (Num T) is a vector of the Num operations for type T.
How type classes work

When you write this...

```haskell
square :: Num a => a -> a
square x = x * x
```

...the compiler generates this

```haskell
square :: Num a => a -> a
square x = x * x
```

```haskell
class Num a where
    (+) :: a -> a -> a
    (*) :: a -> a -> a
    negate :: a -> a
    ...
```

The class decl translates to:

- A data type decl for Num
- A selector function for each class operation

A value of type (Num T) is a vector of the Num operations for type T
How type classes work

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
instance Num Int where
  a + b     = plusInt a b
  a * b     = mulInt a b
  negate a = negInt a
  ...etc..
```

```
dNumInt :: Num Int
dNumInt = MkNum plusInt
          mulInt
          negInt
          ...
```

An instance decl for type T translates to a value declaration for the Num dictionary for T

A value of type (Num T) is a vector of the Num operations for type T
All this scales up nicely

- You can build big overloaded functions by calling smaller overloaded functions

\[
\text{sumSq} :: \text{Num } n \rightarrow n \rightarrow n \\
\text{sumSq } x \ y = \text{square } x + \text{square } y
\]

\[
\text{sumSq } d \ x \ y = (+) \ d \ (\text{square } d \ x) \\
(\text{square } d \ y)
\]

- Extract addition operation from \( d \)
- Pass on \( d \) to \( \text{square} \)
All this scales up nicely

- You can build big instances by building on smaller instances

```haskell
class Eq a where
    (==) :: a -> a -> Bool

instance Eq a => Eq [a] where
    (==) [] [] = True
    (==) (x:xs) (y:ys) = x==y && xs == ys
    (==) _ _ = False

data Eq = MkEq (a->a->Bool)
    (==) (MkEq eq) = eq

dEqList :: Eq a -> Eq [a]
dEqList d = MkEq eql
    where
        eql [] [] = True
        eql (x:xs) (y:ys) = (==) d x y && eql xs ys
        eql _ _ = False
```

You can build big instances by building on smaller instances.
Example: complex numbers

class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: Integer -> a
  ...

inc :: Num a => a -> a
inc x = x + 1

data Cpx a = Cpx a a

instance Num a => Num (Cpx a) where
  (Cpx r1 i1) + (Cpx r2 i2) = Cpx (r1+r2) (i1+i2)
  fromInteger n = Cpx (fromInteger n) 0

Even literals are overloaded

"1" means "fromInteger 1"
quickCheck :: Test a => a -> IO ()

class Testable a where
test :: a -> RandSupply -> Bool

class Arbitrary a where
arby :: RandSupply -> a

instance Testable Bool where
test b r = b

instance (Arbitrary a, Testable b) => Testable (a->b) where
test f r = test (f (arby r1)) r2
where (r1,r2) = split r

split :: RandSupply -> (RandSupply, RandSupply)
prop_swap :: TS -> Bool

test prop_swap r
= test (prop_swap (arby r1)) r2
where (r1,r2) = split r
= prop_swap (arby r1)
class Arbitrary a where
  arby :: RandSupply -> a

instance Arbitrary Int where
  arby r = randint r

instance Arbitrary a => Arbitrary [a] where
  arby r | even r1 = []
  | otherwise = arby r2 : arby r3
  where
    (r1,r') = split r
    (r2,r3) = split r'

split :: RandSupply -> (RandSupply, RandSupply)
randInt :: RandSupply -> Int
QuickCheck uses type classes to auto-generate
- random values
- testing functions

based on the type of the function under test

Nothing is built into Haskell; QuickCheck is just a library

Plenty of wrinkles, esp
- test data should satisfy preconditions
- generating test data in sparse domains
In OOP, a value carries a method suite

With type classes, the method suite travels separately from the value

- Old types can be made instances of new type classes (e.g. introduce new Serialise class, make existing types an instance of it)
- Method suite can depend on result type e.g. fromInteger :: Num a => Integer -> a
- Polymorphism, not subtyping
Type classes have proved extraordinarily convenient in practice

- Equality, ordering, serialisation
- Numerical operations. Even numeric constants are overloaded; e.g. \( f \ x = x^2 \)
- And on and on....time-varying values, pretty-printing, collections, reflection, generic programming, marshalling, monads, monad transformers....
Type classes are the most unusual feature of Haskell’s type system.
Type-class fertility

- Wadler/Blott type classes (1989)
  - Higher kinded type variables (1995)
  - Multi-parameter type classes (1991)
  - Overlapping instances
  - "newtype deriving"
  - Derivable type classes

- Implicit parameters (2000)
  - Extensible records (1996)
  - Functional dependencies (2000)
  - Associated types (2005)

- Computation at the type level
- Generic programming
- Testing

Applications

Variations
Type classes summary

- A much more far-reaching idea than we first realised: the automatic, type-driven generation of executable “evidence”

- Many interesting generalisations, still being explored

- Variants adopted in Isabel, Clean, Mercury, Hal, Escher

- Long term impact yet to become clear
Doing I/O
Where is the I/O in xmonad?

- All this pure stuff is very well, but sooner or later we have to
  - talk to X11, whose interface is not at all pure
  - do input/output (other programs)

A functional program defines a pure function, with no side effects

The whole point of running a program is to have some side effect
All this pure stuff is very well, but sooner or later we have to
- talk to X11, whose interface is not at all pure
- do input/output (other programs)
Idea:

```
putStr :: String -> ()
-- Print a string on the console
```

BUT: now we might do arbitrary stateful things.

And what does this do?

```
[putStr "yes", putStr "no"]
```

- What order are the things printed?
- Are they printed at all?

Order of evaluation!

Laziness!
A value of type \((\text{IO } t)\) is an “action” that, when performed, may do some input/output before delivering a result of type \(t\).

```
putStr :: String -> IO ()
-- Print a string on the console
```

- “Actions” sometimes called “computations”
- An action is a \textbf{first class value}
- \textbf{Evaluating} an action has no effect; \textbf{performing} the action has an effect
A value of type \((\text{IO } t)\) is an “action” that, when performed, may do some input/output before delivering a result of type \(t\).

\[
\text{type IO } a = \text{World} \rightarrow (a, \text{World})
\]

-- An approximation

A helpful picture

World in → IO a → World out

result :: a
 HELLO WORLD

getLine :: IO String
putStr :: String -> IO ()

Main program is an action of type IO ()

main :: IO ()
main = putStr "Hello world"
**Goal:**
read a line and then write it back out
We have connected two actions to make a new, bigger action.

echo :: IO ()
echo = do { l <- getline; putStrLn l }
getTwoLines :: IO (String, String)
getTwoLines = do { s1 <- getLine
                  ; s2 <- getLine
                  ; s3 <- getLine
                  ; s4 <- getLine
                  ; s5 <- getLine
                  }

We want to just return (s1, s2)
The \texttt{return} combinator

\begin{verbatim}
getTwoLines :: IO (String,String)
getTwoLines = do { s1 <- getLine
                   ; s2 <- getLine
                   ; return (s1, s2) }
\end{verbatim}

\texttt{return} :: \texttt{a} -> \texttt{IO a}
Desugaring do notation

- “do” notation adds only syntactic sugar
- Deliberately imperative look and feel

\[
\begin{align*}
do \{ x<-e; s \} &= e >>= (\lambda x \rightarrow do \{ s \}) \\
do \{ e \} &= e
\end{align*}
\]
A “lambda abstraction” 
\((\lambda x \to e)\) means
“a function taking one parameter, \(x\), and returning \(e\)”

\((>>=)\) :: \(\text{IO } a \to (a \to \text{IO } b) \to \text{IO } b\)
You can use
- explicit braces/semicolon
- or layout
- or any mixture of the two

```haskell
getTwoLines :: IO (String, String)
getTwoLines = do  s1 <- getLine
                 s2 <- getLine
                 return (s1, s2)
```
Scripting in Haskell
An example: scripting in Haskell

Run QuickCheck on all functions called "prop_xxx"

Write this script in Haskell

bash$ runhaskell QC.hs Stack.hs
prop_swap: +++ OK, passed 100 tests
prop_focusNP: +++ OK, passed 100 tests
module Main where

import System; import List

main :: IO ()
main = do { as <- getArgs
            ; mapM_ process as }

process :: String -> IO ()
process file = do { cts <- readFile file
                   ; let tests = getTests cts
                       ; if null tests then
                       putStrLn (file ++ ": no properties to check")
                       else do
                       { writeFile "script" $
                         unlines ([":l " ++ file] ++ concatMap makeTest tests)
                   ; system ("ghci -v0 < script")
                   ; return () } }

getTests :: String -> [String]
getTests cts = nub $ filter ("prop_ `isPrefixOf`") $
map (fst . head . lex) $ lines cts

makeTest :: String -> [String]
makeTest test = ["putStr \"" ++ p ++ ": \"", "quickCheck " ++ p]
module Main where

import System
import List

main :: IO ()
main = do { as <- getArgs ; mapM_ process as }

getArgs :: IO [String]
-- Gets command line args

mapM_ :: (a -> IO b) -> [a] -> IO ()
-- mapM_ f [x1, ..., xn]
-- = do { f x1;
-- ...  
--  f xn;
--  return () }
process :: String -> IO ()
-- Test one file
process file
  = do { cts <- readFile file
          ; let tests = getTests cts
          ...

readFile :: String -> IO String
-- Gets contents of file

getTests :: String -> [String]
-- Extracts test functions
-- from file contents

e.g. tests = ["prop_rev", "prop_focus"]
process file = do
  cts <- readFile file
  let tests = getTests cts

  if null tests then
    putStrLn (file ++ " : no properties to check")
  else do
    writeFile "script" (unlines ("[" ++ file ++ concatMap makeTest tests))
    ; system ("ghci -v0 < script")
    ; return ()

putStrLn :: String -> IO ()
writeFile :: String -> String -> IO ()
system :: String -> IO ExitCode
null :: [a] -> Bool
makeTest :: String -> [String]
concatMap :: (a->[b]) -> [a] -> [b]
unlines :: [String] -> String

:l Stack.hs
putStr "prop_rev"
quickCheck prop_rev
putStr "prop_focus"
quickCheck prop_focus
getTests :: String -> [String]
getTests cts = nub (filter ("prop_" `isPrefixOf` ) (map (fst . head . lex) (lines cts )))

lines

map (fst . head . lex)

filter ("prop_" `isPrefixOf` )

nub

“module Main where
import System...”

[“module Main where”, “import System”, ]

[“module”, “import”, ..., “prop_rev”, ]

[“prop_rev”, ]

[“prop_rev”, ]
getTests :: String -> [String]
getTests cts = nub (filter ("prop_" `isPrefixOf` ) (map (fst . head . lex) (lines cts )))

lines :: String -> [String]
lex :: String -> [(String,String)]
filter :: (a->Bool) -> [a] -> [a]
isPrefixOf :: String -> String -> Bool
nub :: [String] -> [String]
   -- Remove duplicates
makeTest :: String -> [String]
makeTest test = ["putStr \"" ++ p ++ ": \"\", "quickCheck " ++ p ]

makeTest "prop_rev"
= ["putStr \"prop_rev: \"", "quickCheck prop_rev"]
What have we learned

- Scripting in Haskell is quick and easy (e.g. no need to compile, although you can)
- It is strongly typed; catches many errors
- But there are still many un-handled error conditions (no such file, not lexically-analysable, ...)
What have we learned

- Libraries are important; Haskell has a respectable selection
  - Regular expressions
  - Http
  - File-path manipulation
  - Lots of data structures (sets, bags, finite maps etc)
  - GUI toolkits (both bindings to regular toolkits such as Wx and GTK, and more radical approaches)
  - Database bindings

...but not (yet) as many as Perl, Python, C# etc
The types tell the story

```haskell

type Company = String

sort :: [Company] -> [Company]
-- Sort lexicographically
-- Two calls given the same
-- arguments will give the
-- same results

sortBySharePrice :: [Company] -> IO [Company]
-- Consult current prices, and sort by them
-- Two calls given the same arguments may not
-- deliver the same results
```

I deliver a list of Company

I may do some I/O and then deliver a list of Company
Haskell: the world’s finest imperative programming language

- Program divides into a mixture of
  - Purely functional code (most)
  - Necessarily imperative code (some)

- The type system keeps them rigorously separate

- Actions are first class, and that enables new forms of program composition (e.g. `mapM_`
First-class control structures

Values of type (IO t) are first class

So we can define our own “control structures”

```haskell
forever :: IO () -> IO ()
forever a = a >> forever a

repeatN :: Int -> IO () -> IO ()
repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a
```

e.g.

```haskell
forever (do { e <- getNextEvent ; handleEvent e })
```
In the end we have to call C!

Haskell

```
foreign import ccall unsafe "HsXlib.h XMapWindow"
mapWindow :: Display -> Window -> IO ()
```

C

```c
void XMapWindow( Display *d, Window *w ) {
    ...
}
```
All the fun is getting data across the border

```haskell
data Display = MkDisplay Addr#
data Window = MkWindow Addr#

foreign import ccall unsafe "HsXlib.h XMapWindow"
  mapWindow :: Display -> Window -> IO ()
```

Addr#: a built-in type representing a C pointer

'foreign import' knows how to unwrap a single-constructor type, and pass it to C
All the fun is getting data across the border

```haskell
data Display = MkDisplay Addr#
data XEventPtr = MkXEvent Addr#

foreign import ccall safe "HsXlib.h XNextEvent"
  xNextEvent :: Display -> XEventPtr -> IO ()

But what we want is

```haskell
data XEvent = KeyEvent ... | ButtonEvent ... |
             | DestroyWindowEvent ... | ...

nextEvent :: Display -> IO XEvent```
Getting what we want is tedious...

```haskell
data XEvent = KeyEvent ... | ButtonEvent ... | DestroyWindowEvent ... | ...

nextEvent :: Display -> IO XEvent
nextEvent d = do { xep <- allocateXEventPtr
                    ; xNextEvent d xep
                    ; type <- peek xep 3
                    ; if type == 92 then
                        do { a <- peek xep 5
                             ; b <- peek xep 6
                             ; return (KeyEvent a b) }
                        else if ... }
```

...but there are tools that automate much of the grotesque pain (hsc2hs, c2hs etc).
The rest of Haskell
Haskell is a **lazy** language

Functions and data constructors don't evaluate their arguments until they need them

```haskell
cond :: Bool -> a -> a -> a
cond True  t e = t
cond False t e = e
```

Same with local definitions

```haskell
abs :: Int -> Int
abs x | x>0       = x
| otherwise = neg_x
where
  neg_x = negate x
```

NB: new syntax guards
Why laziness is important

- Laziness supports **modular programming**
- Programmer-written functions instead of built-in language constructs

\[
(||) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]

- \text{True} || x = \text{True}
- \text{False} || x = x

Short-circuiting "or"
**Laziness and modularity**

```haskell
isSubString :: String -> String -> Bool
x `isSubStringOf` s = or [ x `isPrefixOf` t |
  t <- tails s ]

tails :: String -> [String]
-- All suffixes of s
tails [] = [[]]
tails (x:xs) = (x:xs) : tails xs

or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or [] = False
or (b:bs) = b || or bs
```

Note: The type `String` is defined as `Char`, which means a string is a list of characters.
Why laziness is important

- Typical paradigm:
  - generate all solutions (an enormous tree)
  - walk the tree to find the solution you want

\[
\text{nextMove} :: \text{Board} \rightarrow \text{Move} \\
\text{nextMove} b = \text{selectMove} \ \text{allMoves} \\
\text{where} \\
\quad \text{allMoves} = \text{allMovesFrom} b
\]

A gigantic (perhaps infinite) tree of possible moves
Why laziness is important

- Generally, laziness unifies \textit{data} with \textit{control}
- Laziness also keeps Haskell pure, which is a Good Thing
Other language features

Advanced types
- Unboxed types
- Multi-parameter type classes
- Functional dependencies
- GADTs
- Implicit parameters
- Existential types
- etc etc

Concurrent Haskell (threads, communication, synchronisation)

Software Transactional Memory (STM)

Nested Data Parallel Haskell

Generic programming
One program that works over lots of different data structures

Haskell language

Template Haskell (meta programming)

Rewrite rules (domain-specific compiler extensions)

Monads, monad transformers, and arrows
Haskell’s tool ecosystem

Interpreters
(e.g. GHCi, Hugs)

Compilers
(e.g. GHC, Jhc, Yhc)

Coverage testing

Testing
(e.g. QuickCheck, Hunit)

Haskell language

Programmers

environments
(emacs, vim, Visual Studio)

Debugger

Generators
- parser (cf yacc)
- lexer (cf lex)
- FFI

Space and time profiling

Documentation generation
(Haddock)

Packaging and distribution
(Cabal, Hackage)

LIBRARIES
# Time profiling

**GHC timing profile viewer**

**Report**
Mon Mar 19 15:52 2007 Time and Allocation Profiling Report (Final)

**Command**
catch_opt_prof +RTS -p -RTS Bernoulli_Safe
regress -nolog -time

**Total time**
1.25 sec

**Total alloc**
72,214,048 bytes

<table>
<thead>
<tr>
<th>Cost Centre</th>
<th>Module</th>
<th>Entries</th>
<th>Individual %time</th>
<th>Individual %alloc</th>
<th>Inherited %time</th>
<th>Inherited %alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>MAIN</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>main</td>
<td>Main</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>96.0</td>
<td>99.6</td>
</tr>
<tr>
<td>execNormal</td>
<td>Main</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>92.0</td>
<td>99.6</td>
</tr>
<tr>
<td>concatMapM</td>
<td>General</td>
<td>3</td>
<td>8.0</td>
<td>0.0</td>
<td>8.0</td>
<td>0.0</td>
</tr>
<tr>
<td>execFile</td>
<td>Main</td>
<td>8</td>
<td>0.0</td>
<td>0.0</td>
<td>84.0</td>
<td>99.6</td>
</tr>
<tr>
<td>compile</td>
<td>Prepare</td>
<td>1</td>
<td>12.0</td>
<td>0.0</td>
<td>12.0</td>
<td>0.0</td>
</tr>
<tr>
<td>execMiddle</td>
<td>Main</td>
<td>12</td>
<td>0.0</td>
<td>0.0</td>
<td>56.0</td>
<td>82.1</td>
</tr>
<tr>
<td>loadStage</td>
<td>Main</td>
<td>7</td>
<td>0.0</td>
<td>0.0</td>
<td>8.0</td>
<td>14.8</td>
</tr>
<tr>
<td>getTask</td>
<td>Main</td>
<td>12</td>
<td>0.0</td>
<td>0.0</td>
<td>48.0</td>
<td>67.3</td>
</tr>
<tr>
<td>analyse</td>
<td>Analyse</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>16.0</td>
<td>17.4</td>
</tr>
<tr>
<td>precond</td>
<td>Analyse</td>
<td>24</td>
<td>0.0</td>
<td>0.0</td>
<td>16.0</td>
<td>16.8</td>
</tr>
<tr>
<td>backs</td>
<td>Analyse</td>
<td>891</td>
<td>0.0</td>
<td>0.1</td>
<td>16.0</td>
<td>13.8</td>
</tr>
</tbody>
</table>

Viewer written in Haskell using GTK binding
Fig. 18. Heap production of main by module, when compiling a small program.
This is an example of the table that provides the summary of coverage, with links to the individually marked-up files.

<table>
<thead>
<tr>
<th>module</th>
<th>Top Level Definitions</th>
<th>Alternatives</th>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%</td>
<td>covered / total</td>
<td>%</td>
</tr>
<tr>
<td>module CSG</td>
<td>100 %</td>
<td>0/0</td>
<td>100 %</td>
</tr>
<tr>
<td>module Construct</td>
<td>48 %</td>
<td>17/35</td>
<td>52 %</td>
</tr>
<tr>
<td>module Data</td>
<td>24 %</td>
<td>6/25</td>
<td>13 %</td>
</tr>
<tr>
<td>module Eval</td>
<td>70 %</td>
<td>22/31</td>
<td>60 %</td>
</tr>
<tr>
<td>module Geometry</td>
<td>75 %</td>
<td>42/50</td>
<td>69 %</td>
</tr>
<tr>
<td>module Illumination</td>
<td>61 %</td>
<td>11/18</td>
<td>49 %</td>
</tr>
<tr>
<td>module Intersections</td>
<td>63 %</td>
<td>14/22</td>
<td>38 %</td>
</tr>
<tr>
<td>module Interval</td>
<td>47 %</td>
<td>8/17</td>
<td>41 %</td>
</tr>
<tr>
<td>module Main</td>
<td>100 %</td>
<td>1/1</td>
<td>100 %</td>
</tr>
<tr>
<td>module Misc</td>
<td>0 %</td>
<td>0/1</td>
<td>0 %</td>
</tr>
<tr>
<td>module Parse</td>
<td>80 %</td>
<td>16/20</td>
<td>68 %</td>
</tr>
<tr>
<td>module Primitives</td>
<td>16 %</td>
<td>1/6</td>
<td>16 %</td>
</tr>
<tr>
<td>module Surface</td>
<td>36 %</td>
<td>4/11</td>
<td>24 %</td>
</tr>
</tbody>
</table>
reciprocal :: Int -> (String, Int)
reciprocal n | n > 1 = ('0' : '.': digits, recur)
otherwise = error
  "attemping to compute reciprocal of number <= 1"

where
 (digits, recur) = divide n 1 []

divide :: Int -> Int -> [Int] -> (String, Int)
divide n c cs | c `elem` cs = ([], position c cs)
 r == 0 = (show q, 0)
 r /= 0 = (show q ++ digits, recur)

where
 (q, r) = (c*10) `quotRem` n
 (digits, recur) = divide n r (c:cs)

position :: Int -> [Int] -> Int
position n (x:xs) | n==x = 1
 otherwise = 1 + position n xs

showRecip :: Int -> String
showRecip n =
  "1/" ++ show n ++ " = " ++
  if r==0 then d else take p d ++ "(" ++ drop p d ++ ")"
  where
  p = length d - r
  (d, r) = reciprocal n

main = do
  number <- readLn
  putStrLn (showRecip number)
  main
HackageDB (Haskell’s CPAN)

Packages by category

Categories: Code generation (1), Codec (9), Compilers/Interpreters (3), Composition (2), Control (6), Data (16), Data Mining (1), Data Structures (6), Database (25), Development (6), Distribution (5), Editor (3), Foreign (1), Generics (1), Graphics (16), Interfaces (3), Language (4), Monad (1), Network (18), Parsing (5), Scripting (1), Sound (3), System (21), Testing (4), Text (25), Tool (1), User Interfaces (7), Web (4), XML (1), Unclassified (15).

Code generation

harpy library: Runtime code generation for x86 machine code

Codec

base64-string library: Base64 implementation for String's.
bzlib library: Compression and decompression in the bzip2 format
Codec-Compression-LZF library: LZF compression bindings.
compression library: Common compression algorithms.
Crypto library and programs: DES, Blowfish, AES, SHA1, MD5, RSA, ...
mime-string library: MIME implementation for String's.
tar library: TAR (tape archive format) library.
utf8-string library: Support for reading and writing UTF8 Strings
zlib library: Compression and decompression in the gzip and zlib formats

Compilers/Interpreters

hiccup program: Simple tcl interpreter
his program: Javascript Parser
A downloaded package, p, comes with

- `p.cabal`: a package description
- `Setup.hs`: a Haskell script to build/install

```bash
bash$ ./Setup.hs configure
bash$ ./Setup.hs build
bash$ ./Setup.hs install
```
Standing back...
The central challenge

Useful

Useless

Arbitrary effects

No effects
The challenge of effects

- **Arbitrary effects**
  - Plan A (everyone else)
  - Nirvana
- **No effects**
  - Plan B (Haskell)
- **Useful**
  - Dangerous
  - Safe
- **Useless**
Two basic approaches: Plan A

Arbitrary effects

Examples

- Regions
- Ownership types
- Vault, Spec#, Cyclone, etc etc

Default = Any effect
Plan = Add restrictions
Two main approaches: Plan B

**Default** = No effects
**Plan** = Selectively permit effects

Types play a major role

Two main approaches:

- Domain specific languages (SQL, XQuery, MDX, Google map/reduce)
- Wide-spectrum functional languages + controlled effects (e.g. Haskell)
Lots of cross-over

- Arbitrary effects
- Nirvana
- No effects

- Plan A (everyone else)
- Plan B (Haskell)
- Envy
Lots of cross-over

Arbitrary effects

Idea; e.g. Software Transactional Memory (retry, orElse)

Plan A (everyone else)

Nirvana

Plan B (Haskell)

No effects
One of Haskell’s most significant contributions is to take purity seriously, and relentlessly pursue Plan B.

Imperative languages will embody growing (and checkable) pure subsets.

Knowing functional programming makes you a better Java/C#/Perl/Python/Ruby programmer.
More info: haskell.org

- The Haskell wikibook

- All the Haskell bloggers, sorted by topic
  - http://haskell.org/haskellwiki/Blog_articles

- Collected research papers about Haskell
  - http://haskell.org/haskellwiki/Research_papers

- Wiki articles, by category
  - http://haskell.org/haskellwiki/Category:Haskell

- Books and tutorials
  - http://haskell.org/haskellwiki/Books_and_tutorials
Haskell

Categories: Events

Haskell is a general purpose, purely functional programming language featuring static typing, higher order functions, polymorphism, type classes, and monadic effects. Haskell compilers are freely available for almost any computer.

1 About

Introduction
Language definition
History of Haskell
Future of Haskell
Implementations

GHC
Hugs
nhc98
Yhc

2 Learning Haskell

Haskell in 5 steps
Learning Haskell
Books and tutorials
Wiki articles
Blog articles
Wikibook
Research papers
Example code

3 Libraries

Standard libraries
Hackage library database
Applications and libraries
Hoogle: library search

5 Events

ICFP Programming Contest 2007
OSCON Haskell Tutorial
High-level Parallel Programming Workshop
IFL
Haskell Workshop
ICFP
Haskell Hackathon 2007 II
FPDag

Anywhere
Portland/Oregon
Tokyo/Japan
Freiburg/Germany
Freiburg/Germany
Freiburg/Germany
Utrecht/Netherlands
July 20-23, 2007
July 23, 2007
July 23-24, 2007
September 27-29, 2007
September 30, 2007
October 1-3 2007
September/October 2007
January 11, 2008

6 Headlines

- Haskell.org is a mentoring organisation in the 2007 Google Summer of Code. 9 students have been funded by Google to work on infrastructure projects for Haskell.
- The Haskell-prime committee has started work on defining the next minor revision of the language specification.
- The May 2007 Haskell Communities and Activities report is now out, documenting projects in the Haskell community.
- Haskell, for the third year running, was used by the winning team in the ICFP Programming Contest.

7 News

2007-05-07