Haskell for everyone! (at Code and Supply)

Franklin Chen /
Pittsburgh Haskell

February 2, 2015

Outline

Agenda for today's meeting

- Introductions (abbreviated because of large turnout)
- Pittsburgh Haskell's mission statement
- Lightning talk: the big picture
- Hands-on guided coding workshop

What Haskell means to me

Haskell is the *only* programming language that

- I am still *actively* using **20 years** after. . .
- . . . I first learned it
- . . . and used it in real life!

My Haskell history

- 1991: finished college, majoring in physics
  - no CS or programming courses
  - no coding
- 1992: dropped out of physics grad school, needed new career
  - so I taught myself C, Unix
- 1993: first job as software engineer
- **1994: discovered Haskell on shareware floppies**
- 1995: wrote Haskell code for internal tool at job
- 1996: taught myself CS to prepare to apply to grad school
- 1997: admitted to CS PhD program at Yale
  - declined admission: did not join Yale Haskell research group
  - attended CMU instead, stopped using Haskell
- 2012: **15 years** later, regained interest in Haskell
- 2015: finally using Haskell at work again

Why Pittsburgh Haskell?

Mission statement from http://PittsburghHaskell.org/:

- The Pittsburgh Haskell meetup is for everyone who is currently using or interested in learning the Haskell programming language (and related languages such as PureScript, Elm, Idris).
- We want to create and grow a *fun* and friendly local community of Haskell developers of all levels of experience, through learning and sharing exciting ideas, useful libraries, and insights gained from building things.
- We will emphasize practical hands-on coding as a way to both write useful programs and also deepen understanding of foundational concepts that are useful and applicable beyond just Haskell.

History

Why Haskell was created in 1990 (25 years ago)

- 1980s: frustration for languages researchers and implementers
  - many different competing lazy, purely functional languages
  - practical need for a unified community and code base
- *committee* formed to create a new language from scratch

Haskell community culture: pragmatism

- compiler hackers, theorists, everyday programmers
- lively debates within community
- emphasis on *shipping*
  - awareness of tradeoffs
  - not about up-front perfection
- emphasis on experimentation and evolution
  - using feature toggles (like GCC)
- emphasis on providing many useful features (like Perl)
  - not elegant minimalism
- many (most?) contributors work in private industry, not academia

How has Haskell turned out?

- Pros
  - main compiler, GHC is still the main compiler 25 years later!!
  - huge amount of *continuous* language evolution
  - example: GHC 7.10 soon to be released this month with exciting new features
- Cons
  - hacks or mistakes have to be reversed
  - legacy, limited features and lingo
  - still missing important features
  - lack of formal semantics
    - theorists originally intended to develop one
    - language quickly got too complex, big

Language features

All of Haskell, in 4 buzzwords

- Typed
- Purely functional
- Lazy
- Sweet

Typed

What are expressions?

Expression:

- a source code fragment, a piece of *syntax*
- evaluates, when run, to a **value**
- exists only in your source code
- does not exist inside a running process: values exist there

Example expressions:

```
12      -- evaluates to number of months in a year
12 + 3  -- evaluates to fifteen
"apple" -- evaluates to a string with 5 chars
length "apple" + length "banana" -- evaluates to eleven
```

What are types?

Type:

- a cookie-cutter *template* for some kind of "shape"
- *every* **expression** in Haskell must have at least one valid type
- example expressions with type annotations:

```haskell
-- Type names must be Capitalized
aSum :: Int
aSum = 12 + 3

aGreeting :: String
aGreeting = "hello" ++ " " ++ "world"
```

- if an expression does not have a type, compilation fails.

```haskell
doesNotCompile = length 12 -- no valid type
```

- typed programming: "fitting" shaped expressions together

Where and when do types exist?

Types

- **do not exist at run time!**
- exist *only* for the programmer and compiler
- are assigned to source code fragments, not to data
- are *not* attached to data in a running process

Functions have types

In Haskell, functions have type `X -> Y` where:

- `X` is the input parameter's type
- `Y` is the return value's type

```haskell
increment :: Int -> Int
increment i = i + 1

isTooLong :: String -> Bool
isTooLong word = length word > 12
```

What about “multiple parameters” to functions?

“Multiple parameters” are simulated:

- (most commonly) using a return type that itself is a function type that takes the “next” parameter as its parameter

```haskell
greet :: String -> String -> String -> String
-- sugar for: String -> (String -> (String -> String))

greet greeting name terminator =
    greeting ++ ", " ++ name ++ terminator
```

- (less commonly) using a tuple type for input

```haskell
greetTupled :: (String, String, String) -> String
greetTupled (greeting, name, terminator) =
    greeting ++ ", " ++ name ++ terminator
```

“Tagged union” types (aka enums, variant records)

Tagged union type:

- one or more variants, each *tagged* with a *data constructor*
- each variant has one or more fields

```haskell
-- Actual definition in standard library.
-- 2 variants, both with 0 fields attached
data Bool = False | True

-- 'Yes' variant has 2 fields, 'No' has 1 field,
-- 'Ignore' has 0 fields
data OptIn = Yes AccountNumber | No Why | Ignore

violateMyPrivacy :: OptIn
violateMyPrivacy = Yes 1234
```

Which of these is not like the others, and why?

- `Int`
- `List`
- `String`

Why List is not a simple type

```haskell
list1 :: List -- not legal Haskell
list1 = [12, "hello", True]

list2 :: List -- not legal Haskell
list2 = [False, 7]

-- What could MysteryType possibly be?
nthElem :: Int -> List -> MysteryType

addFirsts :: List -> List -> Int
addFirsts list1 list2 =
  nthElem 0 list1 + nthElem 0 list2
```

Type constructors

`List` is a type constructor, not a simple type.

Type constructor:

- a *type level function* (run only at compile time) that returns a type
- also called "higher-kinded type" (very confusing)

Analogy:

- a type is a cookie-cutter template for a shape
- a type constructor is a *machine* that takes cookie-cutter templates and builds a new cookie-cutter template
- the compiler runs the machine for you so that you can use the resulting cookie-cutter template

Programming with type constructors:

- defining machines that make types

The List type constructor

Note: type parameters for a type constructor are often called "generic types" or "type variables"

```
-- 'List' has type parameter 'elemType'
data List elemType = End
                   | Construct elemType (List elemType)

-- Applying 'List' type constructor to 'Int' returns a type
-- as though we manually wrote:
data ListInt = End | Construct Int ListInt

ourList :: List Int
ourList = Construct 7 (Construct 42 (Construct 12 End))
```

List: Haskell's special syntax

Haskell's actual list constructor syntax uses:

- brackets
- infix operator :

```
-- For illustration
data [elemType] = [] -- pronounced "nil"
                | elemType : [elemType] -- pronounced "cons"

unsweetList :: [Int]
unsweetList = 7 : (42 : (12 : []))

sweetList :: [Int]
sweetList = [7, 42, 12]
```

Polymorphic functions: functions with parameters of type variables

Polymorphic function: another kind of *template*.

- (different meaning of word "polymorphic" from OO world)

```haskell
-- "lifting"
--
-- "for all types 'input' and 'output', convert any function
-- on an element type into a function from the list type of
-- that element"
map :: (input -> output) -> ([input] -> [output])

length :: [elem] -> Int

allCaps :: String -> String
allCaps s = map Char.toUpper s
```

Tons of even fancier types

- outside the scope of today's session
- types are the foundation of Haskell's practical usefulness in
  - reducing boilerplate
  - precisely expressing design

Purely functional

What does "pure" mean in Haskell?

A pure function

- returns the same result when passed the same argument values
- does not cause an observable side effect

How can this possibly work in the real world?!

"Effects as a service": the `IO` type constructor API

The standard library defines type constructor `IO a`.

- In a standalone program:
  - we provide entry point `main :: IO ()`
  - runtime performs effects through `main`
- In GHCi interpreter:
  - REPL treats top level `IO a` expressions specially and performs them rather than returning their values

IO API sampler

```haskell
-- effect: read from stdin
getLine :: IO String

-- effect: print to stdout
putStrLn :: String -> IO ()

-- effect: read contents of file
readFile :: FilePath -> IO String

-- convenient "do" notation to use API
main :: IO ()
main = do        -- "begin block for IO context"
  s <- getLine -- "get string s from stdin"
  putStrLn ("hello " ++ s ++ "!!")
```

Type classes: Haskell's notion of interfaces

```haskell
-- In standard library:
-- type 'showable' belongs to type class 'Show'
-- if 'show' is defined for 'showable'
class Show showable where
  show :: showable -> String

-- | Our own type.
data Color =  Red | Blue

-- | define 'Color' to belong to type class 'Show'
instance Show Color where
  show Red = "red"
  show Blue = "blue"

c :: Color
c = Blue
```

Type classes as **constraints** on type parameters

```haskell
-- polymorphic in showable, with constraint Show
print :: Show showable => showable -> IO ()

colorAction :: IO ()
colorAction = do
  print [False, True, False]
  print Red
```

UI features for type-oriented programming

What is type inference? “Type reconstruction”

If you write code *without* a type annotation, the compiler will

- try to reconstruct the best possible type annotation
- if one exists, it will insert it for you as though you manually wrote it
- if no solution to reconstruction exists, it will report a type error

```
mystery x y = length (x ++ [y] ++ x)
```

is successfully reconstructed internally as

```
mystery :: [elemType] -> elemType -> Int
mystery x y = length (x ++ [y] ++ x)
```

- compiler code generation phase *only ever sees fully type-annotated source*
- in dynamic languages, “type inference” has a different meaning (off topic)

Pragmatic features to help with types

- Typed holes

```
holeyGreeting = "hello " ++ _huh
-- Type checker says:
--
-- Found hole ‘_huh’ with type: [Char]
```

- Deferred type errors: set `-fdefer-type-errors`

```
-- type error becomes warning in this mode
illTypedGreeting = "hello " ++ True

-- If program reaches this code, then runtime error:
--
-- Couldn't match expected type ‘[Char]’ with
-- actual type ‘Bool’
```

Lazy

Lazy evaluation

Roughly, Haskell expressions are evaluated (by default) *outside-in*, versus *inside-out*:

```
-- | Only needs to evaluate the first 5 elements of
-- infinite list of odds
--
-- prop> sumFiveOdds == 1 + 3 + 5 + 7 + 9
sumFiveOdds :: Integer
sumFiveOdds = sum (take 5 [1, 3..])
```

- Good:
  - allows modularity, separating *producing* from *consuming*
  - can be efficient: compute only what is needed
- Bad:
  - can be inefficient: if you need to evaluate it all eventually anyway
  - can result in hard-to-debug space leaks

Sweet

Haskell has and encourages syntactic sugar

(More in the coding workshop.)

Template Haskell: compile-time metaprogamming

- Transform code during parsing of source
- Many popular libraries use Template Haskell macros to remove boilerplate, enable syntactic sugar

Next

Workshop goals

- learn language features by coding!
- use the GHCi interpreter REPL
- write tests, run them, implement code to make them pass
  - `doctest` comments
  - `HSpec` unit tests
  - `QuickCheck` generative tests
- use Haskell as no-compile "scripting" language
  - write and run interactive terminal-based program
- use Cabal to
  - use GHC optimizing native compiler to generate standalone binary to run
  - run an entire test suite
  - generate a package suitable for distribution

Appendix

Development

Core tools needed:

- GHC: optimizing *compiler* to native code
- GHCi: fast *interpreter* with featureful REPL
- Cabal: building and packaging tool

IDEs:

- `ghc-mod` for Emacs, Vim, Sublime
- EclipseFP
- Leksah
- IHaskell: uses IPython protocol
- FP Complete Haskell Center cloud IDE
- many others

Testing frameworks

- Example-based
  - HUnit: inspired by Java JUnit
  - HSpec: inspired by Ruby RSpec
- Property-based
  - QuickCheck
  - SmallCheck
  - SmartCheck
- Tasty: test runner
- doctest:
  - Extracts tests embedded in comments in source code, runs tests

Many awesome libraries

- Hackage: central community package archive
  - about 7,000 uploaded packages
- A curated list: `awesome-haskell`

Resources for learning

We only touched on a tiny fraction of Haskell. Great places to learn more:

- What I wish I knew when learning Haskell
- Learn Haskell
- Haskell bookmarks