Haskell = type-oriented + purely functional + lazy + syntax
(5-minute lightning talk)

Franklin Chen
http://franklinchen.com

October 14, 2014

Types

Basic types

```haskell
-- Type names must be Capitalized
dozen :: Int
dozen = 12

-- Product (tuple) type
groceryItem :: (String, Int)
groceryItem = ("apple", 2)

-- Type synonym: syntactic sugar, not new type
type Item = (String, Int)

anotherItem :: Item
anotherItem = ("banana", 5)

-- Unit (0-tuple)
onlyOneOfThese :: ()
onlyOneOfThese = ()
```

Sum type (tagged union)

- One or more variants, each with a *constructor* (think "factory")
- Each variant has one or more fields

```
-- Constructors: Yes has 2 fields, No has 1, Ignore has 0
data OptIn = Yes Email Phone | No Reason | Ignore

violateMyPrivacy :: OptIn
violateMyPrivacy = Yes "fake@fake.com" "111-1111"
```

- A *record* is syntactic sugar

```
data Info = Contact { email :: Email, phone :: Phone }
```

Function types

- A function is *pure*: return type shows *all* it can do[1]

```haskell
scaryAdd :: Bool -> Int -> Int -> Int
scaryAdd evil x y =
  if evil then
    x + y + 1
  else
    x + y

notFour :: Int
notFour = scaryAdd True 2 2
```

- No *impure* functions that do something other than return value

```haskell
unsafeAdd :: Int -> Int -> Int
unsafeAdd x y =
  -- Type error, must have effect
  sendToNSA x y
```

[1] White lie

Generic types

- Type parameters are listed after the type name

```
type KeyValues key value = [(key, value)]
```

- List is a sum type with special syntax (brackets)

```
responses :: [OptIn]
responses = [
  Ignore,
  Yes "fake@fake.com" "111-1111",
  No "I hate spam"]
```

Effects use the generic type `IO a`

```haskell
-- "get back String in IO context"
getLine :: IO String

-- "given string, get back unit in IO context"
putStrLn :: String -> IO ()

-- slurp file contents into String in IO context"
readFile :: FilePath -> IO String
```

- Haskell runtime performs effects
  - Through user-defined value `main :: IO ()`

```haskell
main :: IO ()
main = do -- "begin block for IO context"
  s <- getLine -- "get String s within IO context"
  putStrLn ("hello " ++ s ++ "!!")
```

Type classes: Haskell's distinguishing feature

- Type class is an *interface* with required "methods"

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)

newtype UserName = U String

instance Eq UserName where
  U x == U y = toLower x == toLower y

itsTrue = U "Chen" == U "chen"
```

- Type class constraints in types

```
lookup :: Eq a => a -> KeyValues a b -> Maybe b

foundCity = lookup (U "chEn")
                   [(U "bob", "LAX"),
                    (U "chen", "PIT")]
```

UI features for type-oriented programming

Type inference

- Type checker puts in types *before compilation happens*
  - Inference includes type class constraints
- In many situations, you can leave out type annotations
  - Sometimes have to resolve ambiguity
- Good practice: write top-level type annotations as documentation

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’
```

Other features

Laziness

- Expressions are evaluated (by default) *outside-in*, versus *inside-out*

```
odds :: [Integer]
odds = [1, 3..]

sumFiveOdds = sum (take 5 odds)
```

- Allows modularity
  - separate *producing* from *consuming*
  - turn *control flow* into *data flow*: hide if/then/while control
- Can be efficient, save needless computation

```
-- Depending on sort algorithm, can be efficient
import Data.List (sort)

smallest2 = take 2 (sort hugeList)
```

Haskell is obsessed with concise syntax

```haskell
-- List comprehension (Python stole from Haskell)
oldGrandparents =
  [grandparent | parent <- [mother, father]
               , grandparent <- parentsOf parent
               , ageOf grandparent > 75]

-- Operator section (/ 100.0)
-- <$> is infix map operator
percents :: [Float]
percents = (/ 100.0) <$> [58.8, 95.5]

-- Composition pipeline
addOneThenDouble
addOneThenDouble = ((+1) >>> (*2)) <$> [3, 4, 5]
```

Derivations: compiler-generated boilerplate

Recursion considered harmful.

```
import Data.Foldable as Foldable

data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)
  deriving (Show, Functor, Foldable)

tree1 :: Tree Int
tree1 = Node (Node (Leaf 1) 2 (Leaf 3))
             4
             (Node (Leaf 5) 6 Empty)

added100 = (+100) <$> tree1

twentyOne = Foldable.sum tree1

oneToSix = Foldable.foldr (:) [] tree1
```

Template Haskell: compile-time metaprogamming

- Transform code before it reaches compiler
- Many libraries use macros to remove boilerplate, enable nice syntax

Ecosystem features

Development

- GHC: optimizing *compiler* to native code
- GHCi: [interpreter] with REPL
- Cabal: build and package code (like Ruby Bundler)

IDEs:

- EclipseFP
- Leksah
- `ghc-mod` for Emacs, Vim, Sublime
- FP Complete Haskell Center cloud IDE

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
  - I just counted about 7,000 uploaded packages
- Pandoc: convert text markup formats
- Yesod: Web framework
- Repa: parallel numeric computing
- Free book "Parallel and Concurrent Programming in Haskell"
- Many others: `awesome-haskell`

Haskell-based languages compiling to JavaScript:

- Elm: for functional reactive programming (FRP)
- PureScript