Fun With Haskell: Introduction

Nathaniel Wesley Filardo

January 11, 2012

Course Metadata

- This slide deck:
  - More functions
  - Types
  - ADTs
- Break
- Next slide deck: lazy evaluation and examples
- Coming up next: effects and monads

Functions
Interlude: Some Words on Induction

How do we prove that every domino in a line falls?

- The domino that doesn't have one behind it is a **base case**. *We* push that one over, causing it to fall.
- Any domino that *does* have one behind it falls after that one does.

(Formal aside: Haskell does not, really, use induction, and inductive arguments are only "mostly" correct. Because it is lazy, it more properly is said to use the categorical dual, coinduction. I know; if you also know or are curious, we can get into it later. A good place to start is "Fast and Loose Reasoning is Morally Correct" [2].)

Functions
Interlude: Some Words on Induction

What does this have to do with anything?

- Proofs are built up from *smaller* proofs.
- The line of 123787123 dominos all falls over because the prefix of 123787122 all fell over because .... because you pushed the first one over.

Functions
Functions on Lists: length

length tells us the length of a list.

- How do we think about that *using induction*?
- What is the base case?
- What is the inductive case?

Functions
Functions on Lists: length

length tells us the length of a list.

- How do we think about that *using induction*?
- What is the length of an empty list?
- What is the length of a non-empty list?

Functions
Functions on Lists: length

length tells us the length of a list:

Length.hs

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

(Disclaimer: this works but isn't how the library's function is defined. The details are important but we are not yet ready for them.)

Functions
Functions on Lists: map

- **map** does something *to each element* of a list:

```
Prelude> map (+1) [1,2,3,4]
[2,3,4,5]
Prelude> map Char.toLower "Hello, World!"
"hello, world!"
```

Functions
Functions on Lists: map

- **map** does something *to each element* of a list:

```
Prelude> map (+1) [1,2,3,4]
[2,3,4,5]
Prelude> map Char.toLower "Hello, World!"
"hello, world!"
```

- Which is to say

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

Functions
Functions on Lists: foldr

foldr is the *one list function to rule them all*:

- Foldr is "the natural eliminator" for lists.
  - Which means that it *captures the induction strategy* on lists.
- Think of it as "replace nil with the base case and cons with the induction step."

Functions
Functions on Lists: foldr

- Suppose we have a list

$$2 : 3 : 5 : []$$

- And we want to compute the product of all elements on it:

$$2 * 3 * 5 * 1$$

- Why did I choose to replace nil with 1?

In code, this substitution is done by foldr, which takes a function and the base case:

```
Prelude> foldr (*) 1 [2,3,5]
30
```

Functions
Functions on Lists: foldr

One possible definition of foldr is

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

So let's try some equational reasoning:

```
foldr (*) 1 [2,3,5]
2 * (foldr (*) 1 [3,5])
2 * (3 * (foldr (*) 1 [5]))
2 * (3 * (5 * foldr (*) 1 []))
2 * (3 * (5 * 1))
2 * (3 * 5)
2 * (15)
30
```

Functions
Functions on Lists: foldr

- We generated the expression $2 * (3 * (5 * 1))$.
- Isn't $((2 * 3) * 5) * 1$ just as good?

- The second is available, via `foldl`.
- The first *follows the structure of the list*; it is more "natural."
- If `xs` is some list, what is

```
foldr (:) [] xs
```

Types

- Everybody sort of have a feel for what's going on?
- Now is an excellent time to stop and go back.

Types
What are types?

- Types represent a *coarsened* version of your program
  - `4`, `1+1`, `sum [1,1,2,3,5,8]` all can have type `Int`.
  - Saying that something is of type `Int` doesn't tell us *which* `Int` it is.
- This coarser program is easier to reason about.
  - For the compiler...
  - And for humans, too!
- In many cases, can get away without specifying them!
  - It is convention and kind to other people to manually specify the type of top-level functions.
  - Can specify types anywhere: excellent debugging tool.

Types
What are types?

Can ask `ghci` what it has **inferred** a type to be:

```
Prelude> :t 1 < 2
1 < 2 :: Bool
Prelude> :t "Foo"
"Foo" :: [Char]
Prelude> let x = 1 :: Int
Prelude> :t (x, x+2)
(x, x+2) :: (Int, Int)
```

And it's not just "stuff" that has types! *Functions* have types:

```
Prelude> :t Char.toUpper
Char.toUpper :: Char -> Char
```

Types
Polymorphic Types

- What is the type of the function `fst`?
- Why is this a tricky question?

Types
Polymorphic Types

- `fst` has to specify that it takes a pair. . .
- But a pair *of what?!*
- *Any* pair!
- Use **type variables** to only partially specify the type.

```
fst :: (a,b) -> a
```

- This is called **polymorphism**.

Types
Higher-Order Types

The type of functions taking and/or returning functions!

- What is the type of foldr?
- What did it take?
  - A function, replacing cons,
  - A base case, replacing nil,
  - A list.
- So we know it has a basic skeleton of

$$??? \to ??? \to ??? \to ???$$

Types
Higher-Order Types

- What is the type of foldr?
- Some refinement of

$$??? \rightarrow ??? \rightarrow ??? \rightarrow ???$$

- Call the list elements `a` and the base case `b`

$$??? \rightarrow b \rightarrow [a] \rightarrow ???$$

- The return type is the type of the base case (think: empty list)

$$??? \rightarrow b \rightarrow [a] \rightarrow b$$

- The function takes an element and an intermediate and produces an intermediate:

$$(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$$

Types
Higher-Order Types: Partial Application

- Recall: function applications are just written `f a b ...`
- This isn't just clever (lack of) syntax!
- Consider `foldr (&&) True`.
  - `foldr :: (a -> b -> b) -> b -> [a] -> b`,
  - `&& :: Bool -> Bool -> Bool`,
  - `foldr (&&) :: Bool -> [Bool] -> Bool`,
  - `True :: Bool`,
  - `foldr (&&) True :: [Bool] -> Bool`.
- Functions can be **partially applied** (**unsaturated**)
  - The result is *another function*!

Types
Higher-Order Types

- `foldr` in fact can be used to implement any function *g* of this form (i.e. given a *z* and a *f*):

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

- Then `g = foldr f z`.
- Higher-order functions open the door to factoring out *recursion strategies* from processing functions.
- There's an entire (category-theoretic) landscape here, which we could get into if people are interested. [6, 5]

Types
Type Classes

- Sometimes, many different types of things all have “the same ability.”
  - In Java, we could express this as an `interface`.
- Examples:
  - Some things can be added, subtracted, ...
  - Some things can be compared for equality or ordering
  - Some things can be printed out
- Haskell groups types together into **type classes**.
  - Please do not confuse these with OO classes.
- This gives us type-directed overloading in a nice way.
  - A type may be a **instance** of a class.

Types
Type Classes

Consider equality:

- The class Eq specifies two functions:

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

- An instance might then look like:

```
instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False

    {- ... -}
```

Types
Type Classes

- Another instance might be

```
instance (Eq a, Eq b) => Eq (a,b) where
    (a,b) == (c,d) = (a == c) && (b == d)
    {- ... -}
```

- The `(Eq a, Eq b)` to the left of the double arrow => is called a **context**.
- It says that we can define equality on pairs if we have a definition of equality on the constituent types.

Types
Type Classes

- Try `:info Eq` at `ghci`'s prompt.
  - (The list will vary depending on *which modules* you have in scope; more on that some other day.)
- Try this: `map (+1) [1,2] == [2,3]`.

Types
Type Classes

- The `Show` class provides (among other fiddly bits) the `show` function:

```
class Show a where
    show :: a -> String
    -- ...
```

- Haskell uses a class called `Num` to capture the basics of numbers:

```
class (Eq a, Show a) => Num a where
    (+) :: a -> a -> a
    -- ...
    fromInteger :: Integer -> a
```

Types
Type Classes

- `Ord` refines equality for fully-ordered types:

```
class (Eq a) => Ord a where
    compare :: a -> a -> Ordering
    (<) :: a -> a -> Bool
    max :: a -> a -> a
    -- ...
```

- `Enum` captures what it means to be an enumerable type.
- `Bounded` adds limits.
- Other classes (e.g. `Integral` and `Fractional`) provide more numeric functionality:
    - Only some types support division.
    - Only some types can represent their own reciprocals.

Types
Type Classes

- The `Read` class provides a (fragile!) parser:

```
read :: Read a => String -> a
```

- Useful when hacking things together, but don't depend on it.
- Throws exceptions when it can't read the right thing.

```
Prelude> read "3" :: Int
3
Prelude> read "\"Foo\"" :: String
"Foo"
```

An Algebraic Take on Data
Recreating Pairs

- We aren't restricted to built-in data types.
- Define our own with `data` declarations.

MyPair.hs

```
data MyPair a b = MyPair a b
myFst (MyPair a b) = a
myMapSnd f (MyPair a b) = MyPair a (f b)
```

- Haskell programmers often pun and use the same name for the type and its constructor, especially when there's just one.
- Defining a data type gives us constructors and pattern matching *destructors* implicitly.

An Algebraic Take on Data
Recreating Pairs

- Can also have larger **product**s:

```
data Triple a b c = Triple a b c
data Quadruple a b c d = Quad a b c d
```

- (Usually you will see that constructors are shorter than type names, if it matters, because we write them more often).
- Can also have singletons:

```
data Id a = Id a
```

- And. . . zero-tons, pronounced "unit":

```
data () = ()
```

An Algebraic Take on Data
Choices

- Many times, types are used to express choices.
- Sometimes we have **Either** an a or a b:

```
data Either a b = Left a | Right b
```

- The | indicates a choice of constructors (**branch**).
  - Dually, a plurality of pattern matches to be done.
- Can have more than two constructors.
- Constructors do not need to take arguments.

An Algebraic Take on Data
Choices: The Maybe Type

Do these things scare you?

- "`NULL` pointer dereference"
    - `Segmentation Fault (core dumped)`
- "`NullPointerException`"

They probably should (Hoare [3]):

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

An Algebraic Take on Data
Choices: The Maybe Type

- What do we really want, instead?
- **Maybe** we have **Just** a thing, or
- **Maybe** we have **Nothing**.

```
data Maybe a = Nothing | Just a
```

An Algebraic Take on Data
Choices: The Maybe Type

- Hey, what's the head of an empty list?

```
Prelude> head []
*** Exception: Prelude.head: empty list
```

- Now that's not very nice!
  - Especially because we can't catch exceptions in pure code! (More on that some other day.)
- But it's certainly the case (formally: a total function) that a list Maybe has a head:

```
safeHead [] = Nothing
safeHead (x:_) = Just x
```

An Algebraic Take on Data
Recursive Types

- We've seen an example of a type that contains itself in one branch: a list.
- Binary trees are another excellent example of recursive types.
    - A tree could be empty.
    - Or it could have just one piece of data.
    - Or a root with two children, each trees.

An Algebraic Take on Data
Recursive Types

- We've seen an example of a type that contains itself in one branch: a list.
- Binary trees are another excellent example of recursive types.
  - A tree could be empty.
  - Or it could have just one piece of data.
  - Or a root with two children, each trees.
- That translates naturally to the type:

```
data Tree a = Empty
            | Singleton a
            | Node a (Tree a) (Tree a)
```

- What is the equivalent of foldr on such a tree?

An Algebraic Take on Data
What's "algebraic" about all this, anyway?

- Isomorphisms on types:
  - `Maybe a` $\simeq$ `Either () a`.
  - `a -> b -> c` $\simeq$ `(a,b) -> c`
- Let's talk about that second one:
  - In C or Java, functions take all their arguments at once, making them `(a,b) -> c`.
  - In functional languages, functions very often return other functions, called **closures**.
    - They "close over" the arguments they have been given thus far.
    - As we saw above, can be very handy: define `and` from `foldr`.
  - The witnesses to the isomorphism are available as

```
curry :: ((a,b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a,b) -> c
```

An Algebraic Take on Data
What's "algebraic" about all this, anyway?

- Can build up (up to isomorphism) all of these kinds of data from combinators:

```
data Id f = Id f
data Const k f = Const k
data :+: a b f = Left (a f) | Right (b f)
data :*: a b f = Pair (a f, b f)
data Mu f = In (f (Mu f))
```

- `[a]` $\simeq$ `Mu (Const () :+: (Const a :*: Id))`.
- Or more simply: $List(a) = 1 + a * List(a)$.

Bib

- Available from: `http://courses.cms.caltech.edu/cs11/material/haskell/index.html`.

- Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons.
  Fast and loose reasoning is morally correct.
  In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 206–217, New York, NY, USA, 2006. ACM.
  Available from: `http://doi.acm.org/10.1145/1111037.1111056`, `doi:http://doi.acm.org/10.1145/1111037.1111056`.

- Tony Hoare.
  Null references: The billion dollar mistake, 2009.

Available from: http://qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake.

Hal Daumé III.
Yet another haskell tutorial.
2002–2006.
Available from: http://www.cs.utah.edu/~hal/htut/.

Edward Kmett.
Rotten bananas, 2008.
Available from: http://comonad.com/reader/2008/rotten-bananas/.

Edward Kmett.
Recursion schemes: A field guide (redux), 2009.

Available from: `http://comonad.com/reader/2009/recursion-schemes/`.