Writing code that I'm not smart enough to write

A funny thing happened at Lambda Jam

Background

* "Let's make a lambda calculator" — Rúnar Bjarnason
* Task: write an interpreter for the lambda calculus

Lambda Calculus

* Variables: f, g, x, y, z, & etc.
* Function application: f x ("f applied to x")
* Lambda abstraction
  λx.y
  * (meaning, an anonymous function that takes x and returns y)

More on λ calculus

* It's like* a Turing machine, you can calculate anything with it!

* 0: $\lambda f.\lambda x.x$
  1: $\lambda f.\lambda x.f\ x$
  2: $\lambda f.\lambda x.f\ (f\ x)$
  3: $\lambda f.\lambda x.f\ (f\ (f\ x))$
  ... etc.

* add: $\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$

* "like" meaning "provably equivalent to"

Add one and one

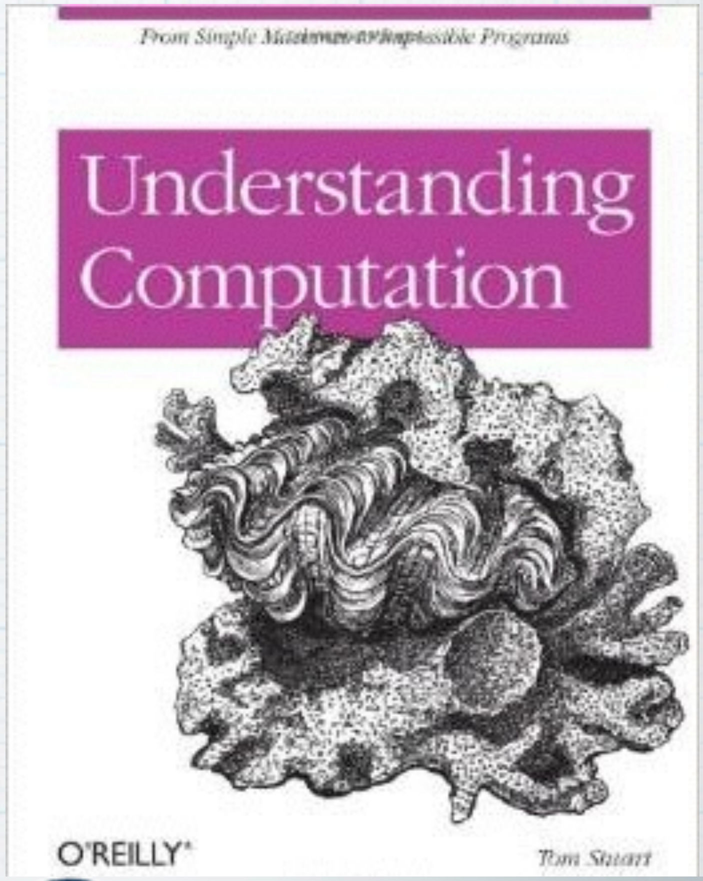
- one: $\lambda f.\lambda x.f\ x$, two: $\lambda f.\lambda x.f\ (f\ x)$
- add: $\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$

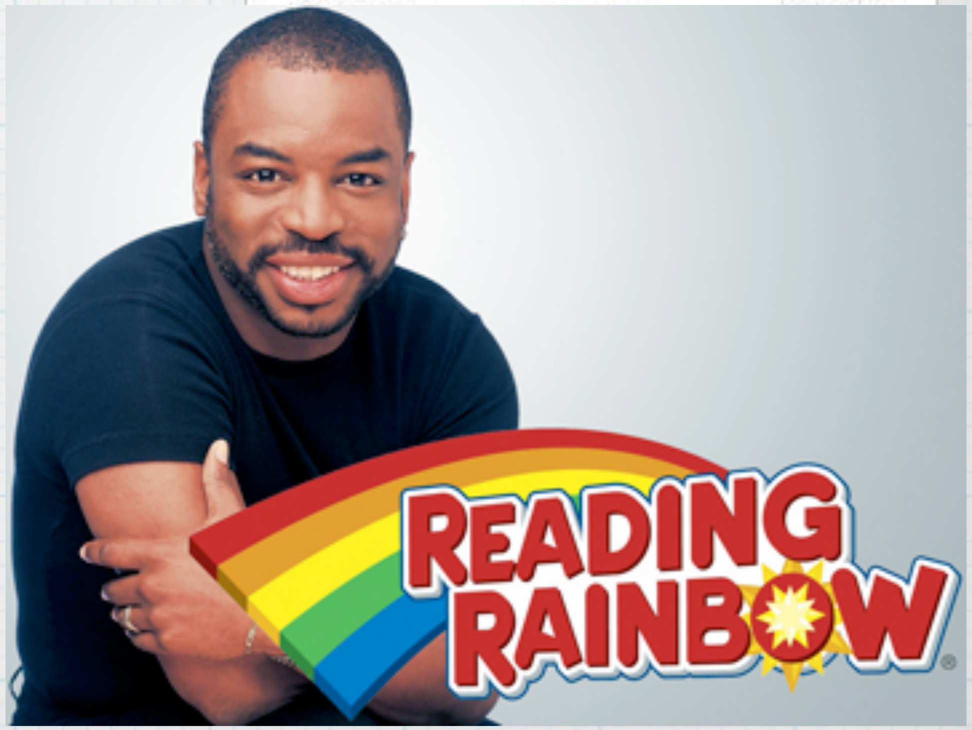
substitution

- $\lambda f.\lambda x.\text{one}\ f\ (\text{one}\ f\ x)$
  α-rename
  $\lambda f.\lambda x.(\lambda g.\lambda y.g\ y)\ f\ ((\lambda h.\lambda z.h\ z)\ f\ x)$
  $\lambda f.\lambda x.(\lambda g.\lambda y.g\ y)\ f\ (f\ x)$
  β-reduction
  $\lambda f.\lambda x.(\lambda y.f\ y)\ (f\ x)$
  $\lambda f.\lambda x.f\ (f\ x)$ — two!

It actually works

* But don't take my word for it!
* "Understanding Computation" by Tom Stuart





Example

```ruby
(1..100).map do |n|
  if (n % 15).zero?
    'FizzBuzz'
  elsif (n % 3).zero?
    'Fizz'
  elsif (n % 5).zero?
    'Buzz'
  else
    n.to_s
  end
end
```

In the lambda calculus

Back to me

* So I'm supposed to implement an interpreter for this language

What I'm given: Terms

```
data Term
  = Var String
  | Lit Int
  | App Term Term
  | Lam String Term
```

f x

λx.x

What I'm given: Values

```
data Value
   = Val Int
   | Fun (Value -> Value)
```

This is the "runtime" representation of functions

What I'm given: Values

This is where variables "live"

```
type Env = [(String, Value)]
```

Okay, now go write it...

* I had no idea how to do this
* BUT... "follow the types"

Some parts are easy

```haskell
find :: Env -> String -> Value
-- gets a value from the env.

eval :: Env -> Term -> Value
eval e (Var s) = find e s
eval _ (Lit i) = Val i
-- harder stuff…
```

Then it got harder

```
eval :: Env -> Term -> Value
eval e (Var s) = find e s
eval _ (Lit i) = Val i
eval e (App f x) = let (Fun f') = eval e f
                       x'       = eval e x
                   in f' x'
```

Since eval returns a Value, f' must be Value -> Value

similarly, x' must be a Value

My brain errored-out on this one

```
eval :: Env -> Term -> Value
eval e (Var s) = find e s
eval _ (Lit i) = Val i
eval e (App f x) = let (Fun f') = eval e f
                       x'       = eval e x
                   in f' x'

eval e (Lam s t)
  = Fun (\v -> eval (e ++ [(s, v)]) t)
```

Value -> Value

the lambda evals in a new environment

My brain errored-out on this one

```haskell
eval :: Env -> Term -> Value
eval e (Var s) = find e s
eval _ (Lit i) = Val i
eval e (App f x) = let (Fun f') = eval e f
                       x'       = eval e x
                   in f' x'
```

```haskell
eval e (Lam s t)
  = Fun (\v -> eval (e ++ [(s, v)]) t)
```

I didn't really know how to write this. I followed the types

Meditations on learning Haskell

* "I **routinely** write code in Haskell that I am **not smart enough to write**."
* "...I just break it down into simple enough pieces and make the **free theorems** strong enough by using **sufficiently abstract types** that **there is only one definition**."
* http://bitemyapp.com/posts/2014-04-29-meditations-on-learning-haskell.html

Free theorems?

Theorems for free!

- Great paper that starts with a game:
- Tell me the type of a polymorphic function, but don't let me see how it's implemented...

Theorems for free!

Philip Wadler
University of Glasgow*

June 1989

Abstract

From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

1 Introduction

Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.

The purpose of this paper is to explain the trick. But first, let's look at an example.

Say that r is a function of type

$$r : \forall X.\ X^* \to X^*.$$

Here X is a type variable, and X^* is the type "list of X". From this, as we shall see, it is possible to conclude that r satisfies the following theorem: for all types A and A' and every total function $a : A \to A'$ we have

$$a^* \circ r_A = r_{A'} \circ a^*.$$

Here $\circ$ is function composition, and $a^* : A^* \to A'^*$ is the function "map a" that applies a elementwise to a list of A yielding a list of A', and $r_A : A^* \to A^*$ is the instance of r at type A.

The intuitive explanation of this result is that r must work on lists of X for *any* type X. Since r is provided with no operations on values of type X, all it can do is rearrange such lists, independent of the values contained in them. Thus applying a to each element of a list and then rearranging yields the same result as rearranging and then applying a to each element.

For instance, r may be the function $\mathit{reverse} : \forall X.\ X^* \to X^*$ that reverses a list, and a may be the function $\mathit{code} : \mathit{Char} \to \mathit{Int}$ that converts a character to its ASCII code. Then we have

$$\begin{array}{cl} & \mathit{code}^*\ (\mathit{reverse}_{\mathit{Char}}\ [\text{‘a’}, \text{‘b’}, \text{‘c’}]) \\ = & [99, 98, 97] \\ = & \mathit{reverse}_{\mathit{Int}}\ (\mathit{code}^*\ [\text{‘a’}, \text{‘b’}, \text{‘c’}]) \end{array}$$

which satisfies the theorem. Or r may be the function $\mathit{tail} : \forall X.\ X^* \to X^*$ that returns all but the first element of a list, and a may be the function $\mathit{inc} : \mathit{Int} \to \mathit{Int}$ that adds one to an integer. Then we have

$$\begin{array}{cl} & \mathit{inc}^*\ (\mathit{tail}_{\mathit{Int}}\ [1, 2, 3]) \\ = & [3, 4] \\ = & \mathit{tail}_{\mathit{Int}}\ (\mathit{inc}^*\ [1, 2, 3]) \end{array}$$

which also satisfies the theorem.

On the other hand, say r is the function $\mathit{odds} : \mathit{Int}^* \to \mathit{Int}^*$ that removes all odd elements from a list of integers, and say a is inc as before. Now we have

$$\begin{array}{cl} & \mathit{inc}^*\ (\mathit{odds}_{\mathit{Int}}\ [1, 2, 3]) \\ = & [2, 4] \\ \neq & [4] \\ = & \mathit{odds}_{\mathit{Int}}(\mathit{inc}^*\ [1, 2, 3]) \end{array}$$

and the theorem is *not* satisfied. But this is not a counterexample, because odds has the wrong type: it is too specific, $\mathit{Int}^* \to \mathit{Int}^*$ rather than $\forall X.\ X^* \to X^*$.

This theorem about functions of type $\forall X.\ X^* \to X^*$ is pleasant but not earth-shaking. What is more exciting is that a similar theorem can be derived for *every* type.

*Author's address: Department of Computing Science, University of Glasgow, G12 8QQ, Scotland. Electronic mail: wadler@cs.glasgow.ac.uk.

This is a slightly revised version of a paper appearing in: *4'th Internationl Symposium on Functional Programming Languages and Computer Architecture*, London, September 1989.

1

First

* The paper focuses on a different theorem for map (and we'll get to that) BUT
* The same sort of reasoning can also help us WRITE it in the first place.

map

You're given a function from a to b

* map :: (a -> b) -> [a] -> [b]

...and a list of a's

So...

* We don't know ANYTHING about what type 'a' and 'b' are (they could both be anything)
* We MUST produce a list of b
* And we're only given that function: a -> b
* therefore: we can't call any function on them EXCEPT the one we're given

map

We have this function

Must produce list of b

* map :: (a -> b) -> [a] -> [b]

Don't know what these are

map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
```

List has two constructors, the first is empty

```
data [a]
  = []
  | a : [a]
```

map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x
```

```
data [a]
  = []
  | a : [a]
```

Only way to get a 'b'.
But we also need a list...

map

```
map :: (a -> b) -> [
map _ [] = []
map f (x:xs) = f x : map f xs
```

...consed onto a list of 'b's

A list of 'b's is a 'b'...

':' aka "cons"
(:) :: b -> [b] -> [b]

map

```
map :: (a -> b) -> [a
map _ [] = []
map f (x:xs) = f x : map f xs
```

"The only list of 'b' that we have"

"The only 'b' that we have"

':' aka "cons"

(:) :: b -> [b] -> [b]

But there's more...

* Type variables in Haskell must work for ANY type
* This is a strong claim and it gives us extra information, just from the types!

map: free theorem!

```
map :: (a -> b) -> [a] -> [b]

reverse :: [a] -> [a]

map f . reverse == reverse . map f
```

any 'f'

any
function that just
rearranges

not
gonna
prove it

filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs
```

Filter: things to note

* The output list must be composed only from elements in the input list
* Only other things we know:
    * length of list
    * result of calling p on the list elements

Map: things to note

* map doesn't change the length of a list
* map f . map g = map (f . g)

Filter: free theorem!

* filter p (map h xs) = map h (filter (p . h) xs)

Intuitively, what's that mean?

* filter p (map h xs) = map h (filter (p . h) xs)
* "filtering transformed things is the same as transforming things that you've pre-filtered"

Final note: hiding in plain sight

* map :: (a -> b) -> [a] -> [b]
* In some sense, the 'a' type is "hidden"
* Or compose:
  (b -> c) -> (a -> b) -> a -> c
* The 'b' type never "escapes" and we can't do anything with it

Note

* The MORE polymorphic something is, the FEWER implementations are possible

Things to check out

* http://daniel.yokomizo.org/2011/12/understanding-higher-order-code-for.html
* "Theorems for free!" by Wadler

Thanks