Implementing Untyped λ-Calculus

in Haskell

slide: Jaiyalas
Lin Jen-Shin (godfat)

godfat.org/slide/2012-05-08-lambda-draft.pdf

Who Am I?

What I learned?

- 2007∼present: (learning) Haskell
- 2006∼present: Ruby
- 2005∼2008: C++
- 2001∼2004: C

What I worked?

- roodo.com
- cardinalblue.com

Where you can find me

- github.com/godfat
- twitter.com/godfat
- profiles.google.com/godfat

How I started to learn Haskell

- PLT at `ssh://bbs@ptt.cc`

How I started to learn Haskell

- PLT at `ssh://bbs@ptt.cc`
- IIS at Sinica

How I started to learn Haskell

- PLT at `ssh://bbs@ptt.cc`
- IIS at Sinica
- FLOLAC

Table of Contents

- What can Haskell do?
- What is λ-Calculus?
- Why Implement λ-Calculus?
- Let's Implement λ-Calculus
- Questions?
- References

- What can Haskell do?
- What is λ-Calculus?
- Why Implement λ-Calculus?
- Let's Implement λ-Calculus
- Questions?
- References

Defined in 1990

Successor of Miranda from 1985

Haskell 98

Haskell 2010

GHC (Glasgow Haskell Compiler)

- STM (Software Transactional Memory)

GHC (Glasgow Haskell Compiler)

- STM (Software Transactional Memory)
- Template Haskell

GHC (Glasgow Haskell Compiler)

- STM (Software Transactional Memory)
- Template Haskell
- GADT (Generalized Algebraic Data Type)

Notable Projects

- Audrey Tang's (唐鳳) Pugs

Notable Projects

- Audrey Tang's (唐鳳) Pugs
- xmonad

Notable Projects

- Audrey Tang's (唐鳳) Pugs
- xmonad
- Darcs

Parallelism vs Concurrency?

- par-tutorial

Parallelism vs Concurrency?

- par-tutorial
- Parallelism $\neq$ Concurrency

Parallelism vs Concurrency?

- par-tutorial
- Parallelism $\neq$ Concurrency
- Parallelism is not concurrency

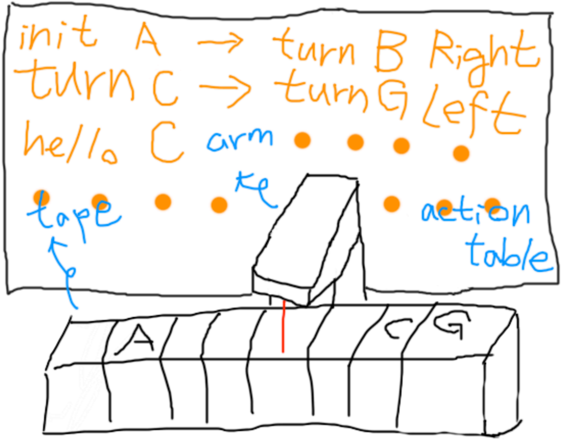
- What can Haskell do?
- What is λ-Calculus?
- Why Implement λ-Calculus?
- Let's Implement λ-Calculus
- Questions?
- References

What is λ-Calculus?

An important formal system
for functional programming

Turing Machine

by Alan Turing in 1936


init A → turn B Right
turn C → turn G Left
hello C
arm
tape
action
table
A
C
G

Turing Machine

- Infinite tape which stores symbols (memory)

Turing Machine

- Infinite tape which stores symbols (memory)
- Finite action table which represents
  (state, symbol) $\rightarrow$ action (program)

Turing Machine

- Infinite tape which stores symbols (memory)
- Finite action table which represents

  (state, symbol) $\rightarrow$ action (program)
- Robotic arm (CPU with a register which stores current state)

Turing Machine

- Infinite tape which stores symbols (memory)
- Finite action table which represents

  (state, symbol) $\rightarrow$ action (program)
- Robotic arm (CPU with a register which stores current state)
  - Read the symbol on the tape at current position

Turing Machine

- Infinite tape which stores symbols (memory)
- Finite action table which represents

  (state, symbol) $\rightarrow$ action (program)
- Robotic arm (CPU with a register which stores current state)
  - Read the symbol on the tape at current position
  - Write a symbol on the tape at current position

Turing Machine

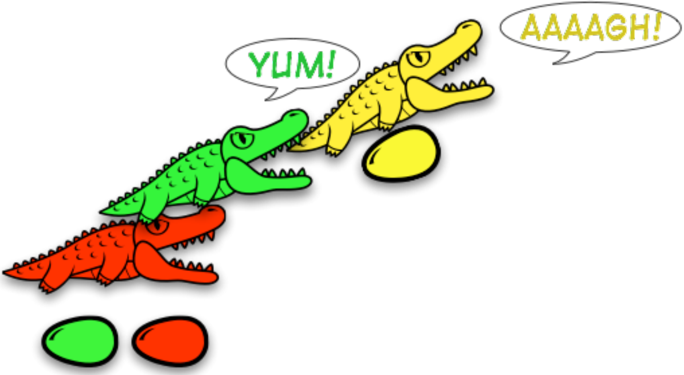
- Infinite tape which stores symbols (memory)
- Finite action table which represents

  (state, symbol) $\rightarrow$ action (program)
- Robotic arm (CPU with a register which stores current state)
  - Read the symbol on the tape at current position
  - Write a symbol on the tape at current position
  - Move the tape left or right

Turing Complete

So what is λ-calculus?

Alligator Eggs!



http://worrydream.com/AlligatorEggs/

by Alonzo Church in 1930s

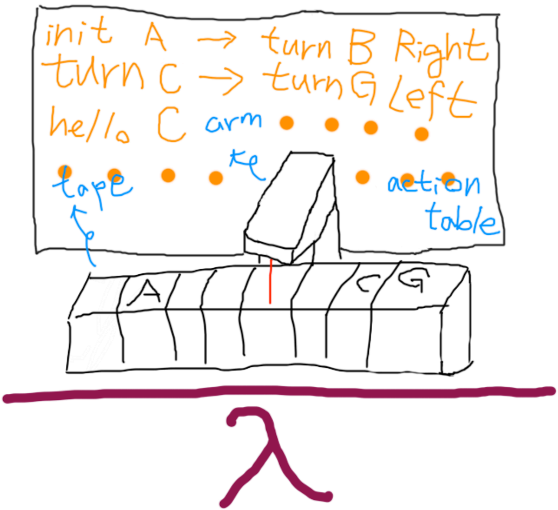
Also Turing complete

but why?

Define λ-Complete

Whatever λ-calculus can do

Implement Turing machine in λ-calculus



λ-Complete

- So λ-calculus can do anything TM can do

λ-Complete

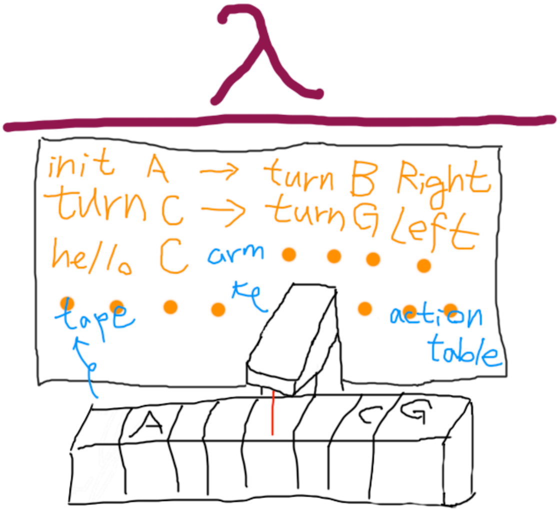
- So λ-calculus can do anything TM can do
- Plus what λ-calculus can do without a TM

λ-Complete

- So λ-calculus can do anything TM can do
- Plus what λ-calculus can do without a TM
- Thus λ-calculus is Turing complete

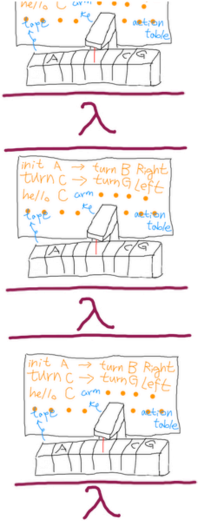
On the other hand...

Implement λ-calculus in Turing machine



So Turing machine is also λ-complete

They have the same computability



What exactly is λ-calculus?

- `(λx. x+1)`

What exactly is λ-calculus?

- `(λx. x+1)`
- `(λx. x+1) 1`

What exactly is λ-calculus?

- `(λx. x+1)`
- `(λx. x+1) 1`
- `= (1+1)`

What exactly is λ-calculus?

- `(λx. x+1)`
- `(λx. x+1) 1`
- `= (1+1)`
- `= 2`

Does it look like Lisp?

```
(λf.(λx.f (x x)) (λx.f (x x)))
```

Church Encoding

Natural Numbers

- 0 ≡ λf.λx. x
- 1 ≡ λf.λx. f x
- ...
- n ≡ λf.λx. f^n x

Computation with Natural Numbers

- `succ  ≡ λn.λf.λx. f (n f x)`
- `plus  ≡ λm.λn.λf.λx. m f (n f x)`

Booleans

- `true  ≡ λa.λb. a`
- `false ≡ λa.λb. b`

Computation with Booleans

- `and  ≡ λm.λn. m n m`
- `or   ≡ λm.λn. m m n`
- `not  ≡ λm.λa.λb. m b a`
- `if   ≡ λm.λa.λb. m a b`

Recursion?
No problems

Y combinator

- Discovered by Haskell Curry
- `y    = (λf.(λx.f (x x)) (λx.f (x x)))`

Y combinator

- Discovered by Haskell Curry
- `y   = (λf.(λx.f (x x)) (λx.f (x x)))`
- `y f = (λf.(λx.f (x x)) (λx.f (x x))) f`

Y combinator

- Discovered by Haskell Curry
- `y   = (λf.(λx.f (x x)) (λx.f (x x)))`
- `y f = (λf.(λx.f (x x)) (λx.f (x x))) f`
- `y f = f (y f)`

Paul Graham,
I know what is Y combinator!

Fixed Point Combinator

- Strict languages need some delay
- `Z = (λx.f (λv.((x x) v))) (λx.f (λv.((x x) v)))`
- Discovered by Alan Turing
- `θ = (λx.λy. (y (x x y))) (λx.λy. (y (x x y)))`

Fixed Point Combinator

- Constructed by Jan Willem Klop
- `Yk = (L L L L L L L L L L L . . .)`

  `where L = λabcdefghijklmnopqstuvwxyzr. (r (thisisafixedpointcombinator))`

- What can Haskell do?
- What is λ-Calculus?
- Why Implement λ-Calculus?
- Let's Implement λ-Calculus
- Questions?
- References

- My interest in researching programming languages

- My interest in researching programming languages
- Implementing λ-calculus in Haskell could teach us a lot in programming languages

- My interest in researching programming languages
- Implementing λ-calculus in Haskell could teach us a lot in programming languages
- It's my most influential Haskell exercise

Let's Learn ~~Haskell The Hard Way~~

You a Haskell For Great Good!

But before we started...

- No parsing here

- No parsing here
- Parse tree (S-expression) interpreter only

- No parsing here
- Parse tree (S-expression) interpreter only
- Haskell is also good at parsing

- No parsing here
- Parse tree (S-expression) interpreter only
- Haskell is also good at parsing
- e.g. parser combinator and parsec

- What can Haskell do?
- What is λ-Calculus?
- Why Implement λ-Calculus?
- Let's Implement λ-Calculus
- Questions?
- References

First Expression and evaluate (source)

```
module Main where
```

First Expression and evaluate (source)

```
module Main where

data Expression = Literal Integer
                | Plus Expression Expression
```

First Expression and evaluate (source)

```
module Main where

data Expression = Literal Integer
                | Plus Expression Expression

evaluate :: Expression -> Integer
evaluate (Literal i)        = i
evaluate (Plus expr0 expr1) =
  evaluate expr0 + evaluate expr1
```

First Expression and evaluate (source)

```
module Main where

data Expression = Literal Integer
                | Plus Expression Expression

evaluate ::  Expression -> Integer
evaluate (Literal i)         = i
evaluate (Plus expr0 expr1) =
  evaluate expr0 + evaluate expr1

test0 = evaluate (Literal 1)
test1 = evaluate (Plus (Literal 1) (Literal 2))
test2 = evaluate (Plus (Plus (Literal 1)
                             (Literal 2))
                       (Literal 3))
```

Algebraic Datatypes

```
data Expression = Literal Integer
                | Plus Expression Expression
```

Think of interface and subclasses if you like OOP

Pattern Matching

```
data Expression = Literal Integer
                | Plus Expression Expression

evaluate :: Expression -> Integer
evaluate (Literal i)       = i
evaluate (Plus expr0 expr1) =
  evaluate expr0 + evaluate expr1
```

Think of dynamic_cast or instanceof with a switch if you like OOP

First Expression and evaluate (source)

```
module Main where

data Expression = Literal Integer
                | Plus Expression Expression

evaluate ::  Expression -> Integer
evaluate (Literal i)        = i
evaluate (Plus expr0 expr1) =
  evaluate expr0 + evaluate expr1

test0 = evaluate (Literal 1)
test1 = evaluate (Plus (Literal 1) (Literal 2))
test2 = evaluate (Plus (Plus (Literal 1)
                             (Literal 2))
                       (Literal 3))
```

Variable and Environment (source)

```
module Main where

data Expression = Literal Integer
                | Plus Expression Expression
                | Variable String
```

Variable and Environment (source)

```
module Main where

data Expression = Literal Integer
                | Plus Expression Expression
                | Variable String

type Environment = [(String, Integer)]
```

Variable and Environment (source)

```
module Main where

data Expression = Literal Integer
                | Plus Expression Expression
                | Variable String

type Environment = [(String, Integer)]

evaluate :: Expression -> Environment -> Integer
```

Variable and Environment (source)

```
module Main where

data Expression = Literal Integer
                | Plus Expression Expression
                | Variable String

type Environment = [(String, Integer)]

evaluate ::  Expression -> Environment -> Integer
evaluate (Literal i)        env = i
evaluate (Plus expr0 expr1) env =
  evaluate expr0 env + evaluate expr1 env
evaluate (Variable name)    env =
  case lookup name env of (Just i) -> i
```

Variable and Environment (source)

```
test0 = evaluate (Variable "var") [("var", 1)]
test1 = evaluate (Plus (Variable "var") (Literal
2)) [("var", 1)]
```

Type Alias

```
module Main where

data Expression = Literal Integer
                | Plus Expression Expression
                | Variable Name

type Name = String
type Environment = [(Name, Integer)]

evaluate ::  Expression -> Environment -> Integer
evaluate (Literal i)        env = i
evaluate (Plus expr0 expr1) env =
  evaluate expr0 env + evaluate expr1 env
evaluate (Variable name)    env =
  case lookup name env of (Just i) -> i
```

Pair of a and b

```
("var", 2) :: (String, Integer)
(2, "var") :: (Integer, String)
```

List of a

```
[1,2,3] :: [Integer]
["a","b","c"] :: [String]
[("var", 1)] :: [(String, Integer)]
```

Curried Functions

```
evaluate :: Expression -> Environment -> Integer
```

Curried Functions

```
evaluate :: Expression -> (Environment -> Integer)
```

Uncurried Functions

```
evaluate :: (Expression, Environment) -> Integer
```

Uncurried Functions

```
threeArguments ::
(String, String, String) -> Integer
```

Curried Function

```
threeArguments ::
String -> String -> String -> Integer
```

Curried Function

```
threeArguments ::
String -> (String -> String -> Integer)
```

Curried Function

```
threeArguments ::
String -> (String -> (String -> Integer))
```

(->) is right-associative

Partially Applied Function

```
filter :: (a -> Bool) -> [a] -> [a]

above60 :: [a] -> [a]
above60 = filter (>=60)

above60 [1..65] -- [60,61,62,63,64,65]
```

Partially Applied Function

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

div2 :: [Integer] -> [Integer]
div2 = map (`div`2)

div2 [1..5] -- [0,1,1,2,2]
```

Function Composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c

above60AndDiv2 :: [Integer] -> [Integer]
above60AndDiv2 = div2 . above60

above60AndDiv2 [1..65] -- [30,30,31,31,32,32]
```

Function Composition

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose g f x = g (f x)
```

Curried functions make function composition powerful,
function composition makes curried function even more useful.

List of Partially Applied Functions

```
mapPlus = map (+) -- think of [(+), (+), ...]

mapPlus1to5 = mapPlus [1..5]
-- think of [(1+), (2+), ...]

map ($1) mapPlus1to5 -- [2,3,4,5,6]


-- applicative functor style
(+) <$> [1..5] <*> [1] -- [2,3,4,5,6]
```

Exception Handling

```
data Expression = Literal Integer
                | Plus Expression Expression
                | Variable Name

type Name = String
type Environment = [(Name, Integer)]

type Value = Maybe Integer

evaluate :: Expression -> Environment -> Value
```

Exception Handling

```
data Expression = Literal Integer
                | Plus Expression Expression
                | Variable Name

type Name = String
type Environment = [(Name, Integer)]

type Value = Maybe Integer

evaluate :: Expression -> Environment -> Value
evaluate (Literal i)       env = Just i
evaluate (Variable name)   env = lookup name env
```

Exception Handling

```
evaluate (Plus expr0 expr1) env =
  let val0 = evaluate expr0 env
      val1 = evaluate expr1 env
  in
      case val0 of
        (Nothing) -> Nothing
        (Just i0) -> case val1 of
                       (Nothing) -> Nothing
                       (Just i1) -> Just (i0 + i1)
```

Maybe Monad with do notation

```
evaluate (Plus expr0 expr1) env =
  do
    val0 <- evaluate expr0 env
    val1 <- evaluate expr1 env
    return (val0 + val1)
```

Do notation underneath

```
evaluate (Plus expr0 expr1) env =
  evaluate expr0 env >>= \val0 ->
    evaluate expr1 env >>= \val1 -> return (val0 +
val1)
```

liftM2

```
evaluate (Plus expr0 expr1) env =
  evaluate expr0 env `plus` evaluate expr1 env where
   plus = liftM2 (+)
```

Sorry! To be continued...

Peek the final work

https://github.com/godfat/sandbox/blob/master/haskell/fpug/01/l

We're hiring

cardinalblue

cblue.tw/jobs

Feel free to ask me questions online

- github.com/godfat
- twitter.com/godfat
- profiles.google.com/godfat

References

- To be listed...
-
-
-
-
-