Let's make a lambda calculator

Rúnar Bjarnason
@runarorama

Lambda Jam Chicago, July 2014

Lambda Calculus

- Variables

  `f`, `g`, `x`, `y`, `z`, etc.

- Function application

  `f x`

- Lambda abstraction

  `λx.y`

Lambda Calculus

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

Your quest:

1. Write an evaluator
2. Write a typer

Evaluation

```haskell
type Env = [(String, Term)]

eval :: Env -> Term -> Term
```

Evaluation

Beta-reduction:

`App (Lam x b) a`

Look for `Var x` in `b` and substitute `a`

Evaluation

free

Name capture:

```
App (Lam x (Lam y (Var x))) (Var y)
```

These two `y` should be kept distinct.

Evaluation

Alpha conversion:

```
App (Lam a (Lam b (Var a))) (Var y)
```

Now, `a` and `b` are *fresh*.

Alpha conversion

`(Lam v e)`

Come up with a new name `x` for `v`
such that `v` ***is not free in*** `e`
substitute `x` wherever `v` occurs in `e`.

Capture-avoidance strategies

- Always alpha convert
- Barendregt convention
- HOAS
- de Bruijn indexing
- Scope monads

Barendregt convention

All bound variables have globally unique names.

Higher-order abstract syntax (HOAS)

Use the host language's lambda!
Makes evaluation trivial but static analysis harder.

```
data Term
  = Var String
  | App Term Term
  | Lam (Term -> Term)
```

de Bruijn indexing

Count the number of binders

```
data Term
  = Free String
  | Bound Int
  | App Term Term
  | Lam Term
```

```
id = Lam (Bound 0)
const = Lam (Lam (Bound 1))
```

Compromise

Names for variables, and HOAS at runtime!

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

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

```
eval :: Env -> Term -> Value
```

Built-ins

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

  | Lit Int
  | Add Term Term
  | Mul Term Term
  | Ifz Int Term Term
```

Simply typed

```
data Term
   = Var String
   | App Term Term
   | Lam Type String Term
```

Simple types

```
data Type
  = Int
  | Fun Type Type
```

Evaluation strategies:

`f(x)`

Call by value:

1. Evaluate `x` to `v`
2. Evaluate `f` to `λy.e`
3. Evaluate `e[y/v]`

Call by name:

1. Evaluate `f` to `λy.e`
2. Evaluate `e[y/x]`

Neither one evaluates under a lambda.

Type system

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma}\,(1)$$

$$\frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T}\,(2)$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)}\,(3)$$

$$\frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau}\,(4)$$

Where to go from here?

- Polymorphic types
- Recursion
- Definitions (lets)
- Modules system
- Surface syntax (parser)
- Data types
- Pattern matching
- Runtime
- Foreign function interface
- I/O and effects
- Errors
- Compiler