Untyped Lambda Calculus

Tikhon Jelvis (tikhon@jelv.is)

December 13, 2013

Why

- simple model of functions
  - everything else stripped away
- makes it easier to **reason** about programs
  - formal reasoning: proofs
  - informal reasoning: debugging
- designing languages
  - simple semantics—easy to extent
  - ML, Haskell, Lisp, . . .

Introduction to theory

- basis for type theory
- introduction to concepts & notation
- “mathematical mindset”

Abstract Syntax

- λ-calculus—syntactic manipulation
- made up of expressions (e)

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & \lambda x.e & \text{abstraction} \\ & | & e_1 e_2 & \text{application} \end{array}$$

Examples

- $\lambda x.x$ is the identity function
  - compare: $f(x) = x$
- $\lambda x.\lambda y.x$ constant function
  - implicit parentheses: $\lambda x.(\lambda y.x)$
  - compare: $f(x, y) = x$

Scoping

- static scope, just like most programming languages
- names do not matter (α equivalence):

$$\lambda x.x \equiv \lambda y.y$$

- variables can be shadowed:

$$\lambda x.\lambda x.x \equiv \lambda x.\lambda y.y$$

Free vs Bound

- **bound**: defined inside an expression:

$$\lambda x.x$$

- **free**: not defined inside an expression:

$$\lambda x.y$$

- free vs bound, y vs x:

$$\lambda x.yx$$

Evaluation

- core idea: **substitution**
    - replace name of argument with its value
- example: given yx, we can substitute $\lambda a.a$ for x:

$$y(\lambda a.a)$$

- careful with scoping!
    - just rename everything

Evaluation Rules

- function application (β-reduction)

$$\frac{(\lambda x.e_1)e_2}{[e_2/x]e_1}$$

- extension (η-reduction)

$$\frac{\lambda x.Fx}{F}$$

- as long as x does not appear in F

Writing an interpreter

- this is all we need to write an interpreter
- any typed functional language:
  - SML, F#, OCaml, Haskell, Scala
- I will use Haskell syntax

Type

$$\begin{array}{rcll} e ::= & x & & \text{variable} \\ & \mid & \lambda x.e & \text{abstraction} \\ & \mid & e_1 e_2 & \text{application} \end{array}$$

- translate to an algebraic data type:

```
type Name = Char

data Expr = Variable Name
          | Lambda Name Expr
          | App Expr Expr
```

Pattern Matching

- pattern matching: operate on ADT by cases

```
eval  Expr → Expr
eval (Lambda x e) = Lambda x e
eval (Variable n) = Variable n
eval (App e e)  = case eval e of
  Lambda x body → eval (subst x e body)
  result        → App result e
```

Substitution

```
subst   Name → Expr → Expr → Expr
subst x newVal (Lambda y body)
  | x   y       = Lambda y (subst x newVal body)
  | otherwise = Lambda y body
subst x newVal (App e e) =
  App (subst x v e) (subst x v e)
subst x newVal (Variable y)
  | x   y       = newVal
  | otherwise = Variable y
```

Evaluation Order

- How far to evaluate?

```
eval (Lambda x e) = Lambda x (eval e)
```

- What order to evaluate in?
  - when to evaluate arguments?

```
Lambda x body → eval (subst x (eval e) body)
```

Fun Stuff

- Write your own interpreter ($<$ 1hr)
- Add parsing, pretty printing and a REPL
- Experiment with different evaluation orders
- Add features like numbers

Numbers

- λ-calculus only has functions
- can we represent data structures and numbers?
- idea: numbers as repeated application
- zero: $\lambda f.\lambda x.x$
- one: $\lambda f.\lambda x.fx$
- two: $\lambda f.\lambda x.f(fx)$
- implement addition and subtraction*

Data Structures

- Lisp-style pairs
- idea: function that applies another function to two arguments
- cons:
$$\lambda x.\lambda y.\lambda f.fxy$$
- first:
$$\lambda x.\lambda y.x$$
- second:
$$\lambda x.\lambda y.y$$
- build up things like lists