Lambda Calculus (Part of Functional Programming)

Joseph Cordina

February 5, 2008

1 Introduction

- Function represented by its graph i.e. a definition of its input and output parameters. This is extensional equality.
- But sorting algorithms all give same graph.
- So we are interested in intensional equality i.e Two functions are equal if they are both defined by the same rules
- Church and Kleene developed $\lambda$ calculus in 1930s

2 BNF Definition

$$< \lambda - term > ::= < variable > | (\lambda < variable > . < \lambda - term >) | (< \lambda - term >< \lambda - term >)$$

Some examples $x (xz) ((xz)(yz)) (\lambda x(\lambda y(\lambda z))(xz))$

- Abstraction is like defining a function.
- Application is the computation, applying $E'$ (params) to $E$ (function) in $EE'$.

3 Some Interesting facts

- Note lambda functions always take one argument. This is the process of currying where a function can give a result as a constant or just another function
- A function can take another function as an argument ex. $I I$ where $I = \lambda x.x$. This is the concept of higher order functions.
4 Lambda Theory

- In our theory
  - ‘=’ stands for an equivalence relation (convertibility)
  - ‘≡’ is syntactical equivalence
  - Note that \((\lambda x.x)y = y\) but \((\lambda x.x)y \neq y\).
  - Normally \(M \equiv N \Rightarrow M = N\) but \(\neg(M = N \Rightarrow M \equiv N)\)

- \(\lambda\)-Theory

\[
\begin{align*}
(\lambda x.M)N &= M[x := N] & (\beta) \\
M &= M & (1) \\
M &= N & (2) \\
N &= M & (3) \\
M &= N & N = L \\
M &= L & (4) \\
MZ &= NZ & (5) \\
M &= N & ZM = ZN \\
M &= N & (6) \\
\lambda x.M &= \lambda x.N & (7)
\end{align*}
\]

- The Xi (\(\xi\)) rule is called rule of weak extensionality.
- The \(\beta\) rule is function application. It corresponds to procedure calls.
- Reduction is the process of multiple applications of the \(\beta\) rule (there are multiple options sometimes).
- There is also the \(\alpha\) reduction (\(\Rightarrow_\alpha\)), i.e. the rule of substitution of bound variable names ex \(\lambda x.x \Rightarrow_\alpha \lambda y.y\).

- Consider

```plaintext
int x;
function f(y:real) {
    x = x + 1;
    return (x)
}

if x==1 f(7) else f(7)
```

- The value of \(f\) will depend on the value of \(x\), and thus \(f(7)\) returns a different result every time. This is bad and we want:
  - A function to depend totally and exclusively on passed parameters
– This is the concept of referential transparency, i.e. allowing a function to be replaced by its value without changing the behavior of your program
– \( f() \) above is not referential transparent.

• \( \lambda \) calculus is referential transparent

  – Consider \( (\lambda xy.x y)y \not\xrightarrow{\beta} \lambda y.yy \) since meaning of function changed.
  – Thus we have to \( (\lambda xy.x y)y \Rightarrow_{\alpha} (\lambda xz. x z)y \Rightarrow_{\beta} \lambda z. y z \)
  – In summary
    * Bound variables are those that appear as a \( \lambda \) parameter.
    * As a rule, all bound variables have to be different from all free variables.
    * To guarantee this, one applies \( \alpha \) rule to change variable names where necessary.

5 Some Cool Stuff (Lesson 2)

\[ T \equiv \lambda x y. x \equiv K \]
\[ F \equiv \lambda x y. y \equiv K I \]
and \( \equiv \lambda x y. x y F \)
if \( \equiv \lambda p c a. p c a \)
\[ [M, N] \equiv \lambda z. z M N \]
\[ \text{fst} \equiv \lambda p. p T \]
\[ \text{snd} \equiv \lambda p. p F \]

• One can define recursion

• Also numbers

• Only thing missing is types → typed lambda calculus