Greg Benson <benson@cs.usfca.edu> May 15, 2013 12:40 PM
To: CS 345 <cs345@cs.usfca.edu>
Lambda Calculus, Normal Order, and Applicative Order

---

In working with some students yesterday we came across a problem in order of evaluation that appeared to result in a incorrect answer.

Here is an explanation of the problem and how to properly evaluation lambda expressions.

The problem arose from evaluating church booleans.

true = \a.\b.a
false = \a.\b.b
or = \m.\n.m m n

Normal order:

or true false
(\m.\n.m m n) (\a.\b.a) (\a.\b.b)
beta
(\n.(\a.\b.a) (\a.\b.a) n) (\a.\b.b)
beta
(\a.\b.a) (\a.\b.a) (\a.\b.b)
beta
(\b. (\a.\b.a)) (\a.\b.b)
beta
(\a.\b.a) = true

It turns out that in lambda calculus that function application is left associative, so the expressions

E1 E2 E3

is really:

(E1 E2) E3

so consider:

or true false

This is really

(or true) false

So, technically we cannot apply false to true first, this is an invalid conversion (and not applicative order). If you try to do this, you will get an incorrect result.

So, what is applicative order then?

Applicative order means evaluating your arguments before passing them into functions.

So, for example, you could have:

(\x.xx) ((\x.y) z)

Normal order is:

(\x.xx) ((\x.x) z)
beta
((\x.x) z)((\x.x) z)
beta
z((\x.x) z)
beta
zz

Applicative order is

(\x.xx) ((\x.x) z)
beta
(\x.xx) z
beta
zz

Which requires one fewer step. This is what we expect from most conventional programming languages. So, when you see a list of expressions like this:

E1 E2 E3

make sure you group them in a left associative manner first:

(E1 E2) E3

before evaluating, whether you are doing normal order or applicative order. Think about applicative order as just reducing a single expression E.

This left associativity is also important when thinking about parsing lambda expressions.

Greg