Lecture 3

Topics

1. Brief review of \(\lambda\)-calculus syntax.
   Varieties of syntax: Thompson, Barendregt, Stenlund, Abstract syntax (also Coq from Software foundations, designed for the typed lambda calculus.)

2. Discuss \textit{capture} of open terms by bound variables, what it means, why it is dangerous, Barendregt’s \textit{variable convention}.

3. Values versus open terms.


5. Lambda (equality) theory from Barendregt, syntactic equality, \(\alpha\)-equality, \(\beta\)-equality.

See CS6110 Spring 2012 Lecture 2 \[\text{here}\]
See \textit{Software Foundations} on the Lambda Calculus \[\text{here}\]

1. Review

   We left off with the convention and the \(\beta\)-reduction rule.

   \textbf{Variable Convention}: In an application of a function, we assume that the binding variables of the function expression are disjoint from the free variables of the argument.

   \[ap(\lambda(x.\lambda(y...b(x,y,...));a))\]

   \textbf{Substitution}: \(b[a/x]\) is simple in this case, we gave the definition. It’s in the notes and Thompson.

   \textbf{\(\beta\)-Reduction} (lazy evaluation): \(ap(\lambda(x.b);a) \Downarrow b[a/x]\)

   Example- \(ap(\lambda(x.\lambda(y.x));a)) \Downarrow \lambda(y.a)\)

   The output is a constant function.

   OCaml version- \((fun\ x\ \rightarrow\ (fun\ y\ \rightarrow\ x))\ a\ ;;\)

   \((fun\ x\ \rightarrow\ a)\)

2. Why do we need the variable convention? Because of \textit{capture}. Applying \(\lambda(x.\lambda(y.x))\) to a constant, say 0, gives

   \[ap(\lambda(x.\lambda(y.x));0) \Downarrow \lambda(y.0),\]

   a constant function. Capture of \(y\) produces the identity function.
\[ ap(\lambda(x.\lambda(y.x));z) \downarrow \lambda(y.z) \]

This is an “arbitrary constant function”.

What is happening in the general case? Capture example:

\[ ap(\lambda(x.\lambda(y.b(x,y))); a(y)) \downarrow \lambda(y.b(a(y),y)) \]

There might be a “meaning for y” in a context, say \( a(y) \) but then \( \lambda(y.b(x,a(y))) \) the external reference is broken. This could happen inside an abstraction.

\[
\begin{align*}
ap(\lambda(y.ap(\lambda(x.\lambda(y.b(x,y))); a(y))); c) \downarrow \\
ap(\lambda(x.\lambda(y.b(x,y))); a(c)) \downarrow \\
\lambda(y.b(a(c),y))
\end{align*}
\]

Doing the reasoning first, we get:

\[ ap(\lambda(y.\lambda(x.\lambda(z.b(x,z))); a(y))); c) \downarrow \\
ap(\lambda(x.\lambda(z.b(a(y),z))); c) \downarrow \\
\lambda(z.b(a(c),z)) \]

We note that \( \lambda(z.b(a(c),z)) =_\alpha \lambda(y.b(a(c),y)) \).

The \( =_\alpha \) means equal up to renaming of bound variables.

What happens if we first do the inner \( \lambda(x.\_\_) \) application and fail to rename the inner \( \lambda(y.\_\_) \)?

3. Another way to understand the \( \lambda \)-calculus is to understand what the values are, the data or the mathematical objects. What are they so far?

Is \( x \) a value?

Is \( \lambda \) a value?

Is \( \lambda(x.x) \) a value? Is \( \lambda(x.\lambda(y.x)) \)?

Is \( \lambda(x.ap(\lambda(y.x);x)) \) a value? Is \( \lambda(x.\lambda(y.x)) \)?

Values are closed abstractions.