Scribe: Sam Arbesman and Muthuramakrishnan Venkitasubramaniam Lecturer: Andrew Myers

These notes provide the following:

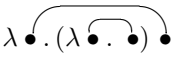
- α-equivalence
- η-reductions
- normal forms
- Church-Rosser Theorem
- IMP

1 α equivalence

$$\lambda x.\,(\lambda x.\,x)x \leftrightarrow \lambda x.\,(\lambda y.\,y)x$$

The above two lambda expressions are the same if we realize that in the first expression the x inside the paranthesis is different from the x outside.

We use stoy diagrams to represent such expressions. Here we associate the variables to the lambda term it is bound to.



Two α-equivalent terms have the same stoy diagram. Formally

$$\lambda x.\,e =_\alpha \lambda x'.\,e\{x'/x\} \text{ where } x' \notin FV(e)$$

This is also called α-renaming. Note that $=_\alpha$ is an equivalence relation.

While expressing lambda terms we can explicitly specify the domain. The successor function is shown as an example

$$\lambda m : \mathbf{Z}.\,(\lambda n : \mathbf{Z}.\,n+1)m$$

2 η reductions

Consider, $\lambda x.\,e\ x$ where $x \notin FV(e)$. Then

$$\lambda x.\,e\ x =_\eta e$$

This is called an η-reduction. The reverse operation is called an η-expansion.

$$\begin{aligned} (\lambda x.\,e_1)e_2 &\longrightarrow e_1\{e_2/x\} \\ (\lambda x.\,e\ x) &\longrightarrow e \text{ (if } x \notin FV(e)) \end{aligned}$$

The term in the LHS is a reducible expression or *redex*. In this case it is an η-redex.

Note that $\lambda x.\,(e\ x)$ and e could have different termination behavior in a call-by-value semantics. Converting e to $\lambda x.\,e\ x$ is called an η-expansion and is used to delay evaluation. This we have already seen in the fixed-point combinator Y.

3 Normal Forms

$$\begin{aligned}
ZERO &= \lambda f.\,\lambda x.\,x \\
ONE &= \lambda f.\,\lambda x.\,(fx) \\
n &= \lambda f.\,\lambda x.\,(f^n x) \\
SUCC &= \lambda n.\,\lambda f.\,\lambda x.\,f((nf)x) \\
SUCC\ ZERO &= \lambda f.\,\lambda x.\,f(\lambda f.\,\lambda x.\,x\ f\ x) \\
&\longrightarrow_\eta \lambda f.\,\lambda x.\,f(\lambda x.\,x)x \\
&\longrightarrow_\beta \lambda f.\,\lambda x.\,(fx)
\end{aligned}$$

When we reach a form where no more reductions are possihble because there are no redexes left then we call it a *normal form.*

4 Church-Rosser Theorem

Rewriting with β and η (or simply β) satisfies the *diamond property* (i.e. it is *confluent*). This means that the reductions can be applied in any order and still arrive at the same result (i.e., that normal forms, if they exist, are unique):

$$\begin{aligned}
(e_1 \rightarrow^*\ e_2) \wedge (e_1 \rightarrow^*\ e_2') &\Rightarrow \exists e_3.(e_2 \rightarrow^*\ e_3) \wedge (e_2' \rightarrow^*\ e_3) \\
(\text{where } e_1 \rightarrow e_2 &= e_1 \rightarrow e_1 \rightarrow e_1'' \rightarrow \cdots \rightarrow e_2)
\end{aligned}$$

This is called the diamond property for the reason that if the evaluations are drawn in a figure you get a diamond shape. Confluence is also called the Church-Rosser property.

Do all languages have this property? Let us examine C:

$$\begin{aligned}
(x = 1) + x\ \{0 = \text{x starting value}\} &\longrightarrow (x = 1) + 0\ \{0\} \longrightarrow 1 + 0\ \{1\} \longrightarrow 1\ \{1\} \\
&\longrightarrow 1 + x\ \{1\} \longrightarrow 1 + 1\ \{1\} \longrightarrow 2\ \{1\}
\end{aligned}$$

Since C does not specify which order to evaluate the sides of the plus operator (it is compiler-dependent), C does not have the diamond property and there is not necessarily a normal form for every expression.

A normal form in a language (as mentioned in the previous lecture) is one that can no longer be evaluated. For any language, there should ideally be an evaluation order of terms which guarantees a normal form. This order is called, unsurprisingly, the *normal order* of the language. In the lambda-calculus, the rule that guides the normal order is simply to reduce the leftmost redex. In this way, we are guaranteed to find a normal form, if one exists. The normal order corresponds to call-by-name and has the characteristic that it has a lazy evaluation of arguments (they are not actually evaluated until the last possible moment when they are needed).

Of course, the question then arises regarding programs that are extensionally equal but do not have the same normal form. How can we tell that these programs are equivalent? Stepping away from questions surrounding extensional equivalence, we move now to IMP.

5 IMP

IMP is a simple imperative language (i.e., one that is series of commands, deriving from the Latin for "command"). It is often used in analysis of programming languages due to its simplicity. An example of some IMP code is below (where the final value of y is 45):
$x := 1;$
while $x \langle 10$ do $(y := y + x; x := x + 1)$

The syntax of IMP is very simple and C-like, and is given by the following grammar:

$$
\begin{array}{rcl}
\mathit{Cmd}\ c & ::= & \mathsf{skip} \mid x := a \mid c_1; c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c \\
\mathit{Aexpr}\ a & ::= & n \mid a_1 \oplus a_2 \mid x \mid \mathsf{etc.} \\
\mathit{BExpr}\ b & ::= & \mathsf{true} \mid \mathsf{false} \mid b_1 \wedge b_2 \mid a_1 \leq a_2 \mid \mathsf{etc.}
\end{array}
$$

More details on IMP to follow!