Lambda Calculus

Summary, transition to ISWIM

CS 6520, Spring 2000

Syntax

$$
\begin{array}{lcl}
M & := & X \mid (\lambda X.M) \mid (M\ M) \\
X & := & \text{a variable}
\end{array}
$$

$M_1 \equiv M_2$ means "M_1 and M_2 are the same expression".

Shorthand: $(\lambda X_1 X_2.M) \equiv (\lambda X_1.(\lambda X_2.M))$

When parentheses are omitted,

- Application associates to the left: $M_1\ M_2\ M_3 \equiv ((M_1\ M_2)\ M_3)$
- Application is stronger than abstraction: $\lambda X.M_1 M_2 \equiv X.(M_1\ M_2)$

Free Variables

$$
\begin{array}{lcl}
\mathcal{FV}(X) & = & \{X\} \\
\mathcal{FV}((\lambda X.M)) & = & \mathcal{FV}(M) \setminus \{X\} \\
\mathcal{FV}((M_1 M_2)) & = & \mathcal{FV}(M_1) \cup \mathcal{FV}(M_2)
\end{array}
$$

Substitution

$$
\begin{array}{lcll}
X_1[M/X_1] & = & M & \\
X_2[M/X_1] & = & X_2 & \text{where } X_1 \not\equiv X_2 \\
(\lambda X_1.M_1)[M_2/X_1] & = & (\lambda X_1.M_1) & \\
(\lambda X_1.M_1)[M_2/X_2] & = & (\lambda X_3.M_1[X_3/X_1][M_2/X_2]) & \text{where } X_1 \not\equiv X_2 \text{ and } X_3 \notin (\mathcal{FV}(M_1) \cup \mathcal{FV}(M_2)) \\
(M_1\ M_2)[M_3/X] & = & (M_1[M_3/X]\ M_2[M_3/X]) &
\end{array}
$$

Reductions

$$
\begin{array}{lcll}
(\lambda X_1.M) & \xrightarrow{\alpha} & (\lambda X_2.M[X_2/X_1]) & \text{where } X_2 \notin \mathcal{FV}(M) \\
((\lambda X.M_1)\ M_2) & \xrightarrow{\beta} & M_1[M_2/X] & \\
(\lambda X.(M\ X)) & \xrightarrow{\eta} & M & \text{where } X \notin \mathcal{FV}(M)
\end{array}
$$

General reduction:

$M \longrightarrow M'$ iff $M \xrightarrow{\alpha} M'$ or $M \xrightarrow{\beta} M'$ or $M \xrightarrow{\eta} M'$

Reduction in contexts:

$C \ := \ [\,] \mid (\lambda X.C) \mid (C\ M) \mid (M\ C)$ $\qquad$ $C[M]$ means "replace the $[\,]$ in C with M".

$C[M] \ \longrightarrow \ C[M']$ where $M \longrightarrow M'$

Multiple-step reductions:
$M \quad \twoheadrightarrow \quad M' \quad$ iff $\quad M \equiv M'$ or, for some M'', $M \;\longrightarrow\; M''$ and $M'' \twoheadrightarrow M'$

Equations

Definition of equality:

$$M = M' \quad \text{iff} \quad \text{for some } M_1 \ldots M_n, \quad \begin{array}{l} M \equiv M_1, \\ M' \equiv M_n, \text{ and} \\ \text{for } i \in [2,n], \text{ either } M_{i-1} \longrightarrow M_i \text{ or } M_i \longrightarrow M_{i-1}. \end{array}$$

In other words, "you can get from M to M' through a series of reductions and inverse reductions."

Churh-Rosser theorem: $M = M' \quad$ iff $\quad$ for some M'', $\quad M \twoheadrightarrow M''$ and $M' \twoheadrightarrow M''$

Normal Order

An expression is in *normal form* if none of its subexpressions can be reduced via $\xrightarrow{\beta}$ or $\xrightarrow{\eta}$.

By Church-Rosser, if two expressions have a normal form, they are equivalent exactly when they have the same normal form (up to α-conversion).

But not all reductions lead to a normal form, even if an expression has one. For example, $(\lambda x.q)((\lambda x.(x\ x))(\lambda x.(x\ x)))$ has the normal form, q, but it can also be reduced to itself forever.

Normalization theorem: If an expression has a normal form, it can be found by repeatedly applying the leftmost β- or η-reduction.

"Leftmost" means the application (for β) or abstraction (for η) available fo reduction whose opening parenthesis comes first in the expression. Mxamples: $(\underline{(}\lambda x.(y\ x))\ w)$, $(z\ \underline{(}\lambda x.(yx)))$.

Note that, using this strategy, $(\lambda x.q)((\lambda x.(x\ x))(\lambda x.(x\ x)))$ reduces immediately to q.

This reduction strategy is called *normal-order reduction* or *call-by-name*.

Not all useful expressions have a normal form. For example, the fixpoint operator has no normal form: $(\lambda f.((\lambda g.f(gg))(\lambda g.f(gg))))$. There is a notion of *head-normal form* which captures all of the "useful" expressions (see Gordon), but we won't go in that direction.

Applicative Order

In most programming languages, if M loops forever, then so does $(f\ M)$, even if f ignores its argument. Such languages use *applicative order reduction* or *call-by-value*; they evaluate function arguments before applying the function.

We can formalize applicative order by defining expressions which are values, restricting application to value arguments, and then defining an evaluation context in terms of values:

$$V \quad := \quad X \mid (\lambda X.M)$$

$$((\lambda X.M)\ V) \quad \xrightarrow{\beta_v} \quad M[V/X]$$

$$C[M] \quad \xrightarrow{v} \quad C[M'] \quad \text{where } M \xrightarrow{\beta_v} M'$$

Normal order can discover any value that applicative order finds, but not vice-versa. In other words,

$$M \overset{v}{\twoheadrightarrow} V \quad \Rightarrow \quad M \twoheadrightarrow V \quad \text{, but}$$
$$M \twoheadrightarrow V \quad \not\Rightarrow \quad M \overset{v}{\twoheadrightarrow} V$$