CS 4110 – Programming Languages and Logics
Lecture #14: More λ-calculus

1 Lambda calculus evaluation

There are many different evaluation strategies for the λ-calculus. The most permissive is *full β reduction*, which allows any *redex*—i.e., any expression of the form $(\lambda x.\, e_1)\, e_2$—to step to $e_1\{e_2/x\}$ at any time. It is defined formally by the following small-step operational semantics rules:

$$\frac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\; e_2 \longrightarrow e_1\; e_2'} \qquad \frac{e_1 \longrightarrow e_1'}{\lambda x.\, e_1 \longrightarrow \lambda x.\, e_1'} \qquad \beta\, \frac{}{(\lambda x.\, e_1)\; e_2 \longrightarrow e_1\{e_2/x\}}$$

The *call by value* (CBV) strategy enforces a more restrictive strategy: it only allows an application to reduce after its argument has been reduced to a value (i.e., a λ-abstraction) and does not allow evaluation under a λ. It is described by the following small-step operational semantics rules (here we show a left-to-right version of CBV):

$$\frac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\; e_2 \longrightarrow v_1\; e_2'} \qquad \beta\, \frac{}{(\lambda x.\, e_1)\; v_2 \longrightarrow e_1\{v_2/x\}}$$

Finally, the *call by name* (CBN) strategy allows an application to reduce even when its argument is not a value but does not allow evaluation under a λ. It is described by the following small-step operational semantics rules:

$$\frac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2} \qquad \beta\, \frac{}{(\lambda x.\, e_1)\; e_2 \longrightarrow e_1\{e_2/x\}}$$

2 Confluence

It is not hard to see that the full β reduction strategy is non-deterministic. This raises an interesting question: does the choices made during the evaluation of an expression affect the final result? The answer turns out to be no: full β reduction is *confluent* in the following sense:

Theorem (Confluence)**.** *If $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ then there exists e' such that $e_1 \longrightarrow^* e'$ and $e_2 \longrightarrow^* e'$.*

Confluence can be depicted graphically as follows:

$$\begin{tikzcd}
 & e \arrow[dl, "*"'] \arrow[dr, "*"] & \\
e_1 \arrow[dr, "*"'] & & e_2 \arrow[dl, "*"] \\
 & e' &
\end{tikzcd}$$

Confluence is often also called the Church–Rosser property.

3 Substitution

Each of the evaluation relations for λ-calculus has a β defined in terms of a substitution operation on expressions. Because the expressions involved in the substitution may share some variable names (and because we are working up to α-equivalence) the definition of this operation is slightly subtle and defining it precisely turns out to be tricker than might first appear.

As a first attempt, consider an obvious (but incorrect) definition of the substitution operator. Here we are substituting e for x in some other expression:

$$\begin{array}{rcll} y\{e/x\} & = & \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} & \\ (e_1\ e_2)\{e/x\} & = & (e_1\{e/x\})\ (e_2\{e/x\}) & \\ (\lambda y.e_1)\{e/x\} & = & \lambda y.e_1\{e/x\} & \text{where } y \neq x \end{array}$$

The intuitive idea is that the last rule relies on α-equivalence to "rewrite" abstractions that use x so they do not conflict. Unfortunately, this definition produces the wrong results when we substitute an expression with free variables under a λ. For example,

$$(\lambda y.x)\{y/x\} = (\lambda y.y)$$

To fix this problem, we need to revise our definition so that when we substitute under a λ we do not accidentally bind variables in the expression we are substituting. The following definition correctly implements *capture-avoiding substitution*:

$$\begin{array}{rcll} y\{e/x\} & = & \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} & \\ (e_1\ e_2)\{e/x\} & = & (e_1\{e/x\})\ (e_2\{e/x\}) & \\ (\lambda y.e_1)\{e/x\} & = & \lambda y.(e_1\{e/x\}) & \text{where } y \neq x \text{ and } y \notin \mathit{fv}(e) \end{array}$$

Note that in the case for λ-abstractions, we require that the bound variable y be different from the variable x we are substituting for and that y not appear in the free variables of e, the expression we are substituting. Because we work up to α-equivalence, we can always pick y to satisfy these side conditions. For example, to calculate $(\lambda z.x\ z)\{(w\ y\ z)/x\}$ we first rewrite $\lambda z.x\ z$ to $\lambda u.x\ u$ and then apply the substitution, obtaining $\lambda u.(w\ y\ z)\ u$ as the result.