CS 4110 – Programming Languages and Logics
Lecture #11: More λ-calculus

Based on material by Stephen Chong, Greg Morrisett, Andrew Myers, George Necula, and Radu Rugina

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:

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

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):

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

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:

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

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 $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$ then there exists e' such that $e_1 \rightarrow^* e'$ and $e_2 \rightarrow^* e'$.*

Confluence can be depicted graphically as follows, where $\twoheadrightarrow$ is the evaluation relation.

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

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:

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

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*:

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

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.

4 de Bruijn Notation

One way to avoid the tricky interaction between free and bound names in the substitution operator is to pick a representation for expressions that doesn't have any names at all! Intuitively, we can think of a bound variable is just a pointer to the λ that binds it. For example, in $\lambda x.\lambda y.y\ x$, the y points to the first λ and the x points to the second λ.

So-called *de Bruijn* notation uses this idea as the representation for λ expressions. Here is the grammar for λ expressions in de Bruijn notation:

$$e ::= n \mid \lambda.e \mid e\ e$$

Variables are represented by integers n that refer to (the index of) their binder while $lambda$-abstractions have the form $\lambda.e$. Note that the the variable bound by the abstraction is not named—i.e., the representation is nameless.

As examples, here are several terms written using standard notation and in de Bruijn notation:

Standard	de Bruijn
$\lambda x.x$	$\lambda.0$
$\lambda z.z$	$\lambda.0$
$\lambda x.\lambda y.x$	$\lambda.\lambda.1$
$\lambda x.\lambda y.\lambda s.\lambda z.x\ s\ (y\ s\ z)$	$\lambda.\lambda.\lambda.\lambda.3\ 1\ (2\ 1\ 0)$
$(\lambda x.x\ x)\ (\lambda x.x\ x)$	$(\lambda.0\ 0)\ (\lambda.0\ 0)$
$(\lambda x.\lambda x.x)\ (\lambda y.y)$	$(\lambda.\lambda.0)\ (\lambda.0)$

To represent a λ-expression that contains free variables in de Bruijn notation, we need a way to map the free variables to integers. We will work with respect to a map Γ from variables to integers called a *context*. As an example, if Γ maps x to 0 and y to 1, then the de Bruijn representation of $x\ y$ with respect to Γ is $0\ 1$, while the representation of $\lambda z.\ x\ y\ z$ with respect to Γ is $\lambda z.\ 1\ 2\ 0$. Note that in this second example, because we have gone under a λ, we have shifted the integers representing x and y up by one to avoid capturing them.

In genreal, whenever we work de Bruijn representations of expressions containing free variables (i.e., when working with respect to a context Γ) we will need to modify the indices of those variables. For example, when we substitute an expression containing free variables under a λ, we will need to shift the indices up so that they continue to refer to the same numbers with respect to Γ after the substitution as they did before. For example, if we substitute $0\ 1$ for the variable bound by the outermost λ in $\lambda.\lambda.1$ we should get $\lambda.\lambda.2\ 3$, not $\lambda.\lambda.0\ 1$. We will use an auxiliary function that shifts the indices of free variables above a cutoff c up by i:

$$\begin{array}{rcl}
\uparrow^i_c (n) & = & \begin{cases} n & \text{if } n < c \\ n+i & \text{otherwise} \end{cases} \\
\uparrow^i_c (\lambda.e) & = & \lambda.(\uparrow^i_{c+1} e) \\
\uparrow^i_c (e_1\ e_2) & = & (\uparrow^i_c\ e_1)\ (\uparrow^i_c\ e_2)
\end{array}$$

The cutoff keeps track of the variables that were bound in the original expression and so should not be shifted as the shifting operator walks down the structure of an expression. The cutoff is 0 initially.

Using this shifting function, we can define substitution as follows:

$$\begin{array}{rcl}
n\{e/m\} & = & \begin{cases} e & \text{if } n = m \\ n & \text{otherwise} \end{cases} \\
(\lambda.e_1)\{e/m\} & = & \lambda.e_1\{(\uparrow^1_0 e)/m+1\})) \\
(e_1\ e_2)\{e/m\} & = & (e_1\{e/m\})\ (e_1\{e/m\})
\end{array}$$

Note that when we go under a λ we increase the index of the variable we are substituting for and shift the free variables in the expression e up by one.

The β rule for terms in de Bruijn notation is as follows:

$$\beta\,\frac{}{(\lambda.e_1)\ e_2 \rightarrow \uparrow^{-1}_0 (e_1\{\uparrow^1_0 e_2/0\})}$$

That is, we substitute occurrences of 0, the index of the variable being bound by the λ, with e_2 shifted up by one. Then we shift the result down by one to ensure that any free variables in e_1 continue to refer to the same things after we remove the λ.

To illustrate how this works consider the following example, which we wrote as $(\lambda u.\lambda v.u\ x)\ y$ in standard notation. We will work with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{rl} & (\lambda.\lambda.1\ 2)\ 1 \\ \to & \uparrow_0^{-1} ((\lambda.1\ 2)\{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda.1\ 2)\{2/0\}) \\ = & \uparrow_0^{-1} \lambda.((1\ 2)\{(\uparrow_0^1 2)/(0+1)\}) \\ = & \uparrow_0^{-1} \lambda.((1\ 2)\{3/1\}) \\ = & \uparrow_0^{-1} \lambda.(1\{3/1\})\ (2\{3/1\}) \\ = & \uparrow_0^{-1} \lambda.3\ 2 \\ = & \lambda.2\ 1 \end{array}$$

which, in standard notation (with respect to Γ), is the same as $\lambda v.y\ x$.

5 Combinators

Another way to avoid the issues having to do with names in the λ-calculus is to work with closed expressions or *combinators*. It turns out that just using three combinators, S, K, and I, and application, we can encode the entire λ-calculus.

Here are the evaluation rules for S, K, and I,

$$\begin{array}{l} \mathsf{K}\ x\ y \to x \\ \mathsf{S}\ x\ y\ z \to x\ z\ (y\ z) \\ \mathsf{I}\ x \to x \end{array}$$

or, equivalently, here are their definitions as closed λ-expressions:

$$\begin{array}{l} \mathsf{K} = \lambda x.\lambda y.\ x \\ \mathsf{S} = \lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z) \\ \mathsf{I} = \lambda x.\ x \end{array}$$

It is not hard to see that I is not needed—it can be encoded as S K K.

To show how these combinators can be used to encode the λ-calculus, we have to define a translation that takes an arbitrary closed λ-calculus expression and turns it into a combinator term that behaves the same during evaluation. This translation is called *bracket abstraction*. It proceeds in two steps. First, we define a function $[x]$ that takes a combinator term M possibly containing free variables and builds another term that behaves like $\lambda x.M$, in the sense that $([x]\ M)\ N \to M\{N/x\}$ for every term N:

$$\begin{array}{rcll} [x]\ x & = & \mathsf{I} & \\ [x]\ N & = & \mathsf{K}\ N & \text{where } x \notin \mathit{fv}(N) \\ [x]\ N_1\ N_2 & = & \mathsf{S}\ ([x]\ N_1)\ ([x]\ N_2) & \end{array}$$

Second, we define a function $(e)*$ that maps a λ-calculus expression to a combinator term:

$$\begin{array}{rcl} (x)* & = & x \\ (e_1\ e_2)* & = & (e_1)*\ (e_2)* \\ (\lambda x.e)* & = & [x]\ (e)* \end{array}$$

As an example, the expression $\lambda x.\lambda y.\ x$ is translated as follows:

$$\begin{array}{rl} & (\lambda x.\lambda y.\ x)* \\ = & [x]\ (\lambda y.\ x)* \\ = & [x]\ ([y]\ x) \\ = & [x]\ (\mathsf{K}\ x) \\ = & (\mathsf{S}\ ([x]\ \mathsf{K})\ ([x]\ x)) \\ = & \mathsf{S}\ (\mathsf{K}\ \mathsf{K})\ \mathsf{I} \end{array}$$

We can check that this behaves the same as our original λ-expression by seeing how it evaluates when applied to arbitrary expressions e_1 and e_2.

$$\begin{array}{rl} & (\lambda x.\lambda y.\ x)\ e_1\ e_2 \\ = & (\lambda y.\ e_1)\ e_2 \\ = & e_1 \end{array}$$

and

$$\begin{array}{rl} & (\mathsf{S}\ (\mathsf{K}\ \mathsf{K})\ \mathsf{I})\ e_1\ e_2 \\ = & (\mathsf{K}\ \mathsf{K}\ e_1)\ (\mathsf{I}\ e_1)\ e_2 \\ = & \mathsf{K}\ e_1\ e_2 \\ = & e_1 \end{array}$$