cs611 Lecture 20 **Lambda Calculus** 16 October 2002

Scribe: Zoya Svitkina, Veselin Stoyanov Lecturer: Radu Rugina

In previous lectures we studied a functional language, REC. One deficiency of REC is the absence of constructs that allow the creation of higher order functions, i.e., using functions as values. We will overcome this deficiency with Lambda Calculus, which can be used as programming language. Many real-world programming languages, such as ML, Lisp, and Haskell are based on this abstraction.

1 Syntax

The syntax of λ-calculus can be expressed with the following grammar:

$$\begin{array}{rlll} e := & & x & \text{(variable)} \\ & \mid & e_0\ e_1 & \text{(application)} \\ & \mid & \lambda x.\, e & \text{(abstraction)} \end{array}$$

The abstraction $\lambda x.e$ is written in the following ways in real-world programming languages:

- ML: `fn x => e`
- Lisp: `(lambda (x)(e))`
- Haskell: `\x -> e`

2 Examples

Here are some example expressions in the language:

- $\lambda x.x$ identity
- $\lambda x.\lambda y.y$ higher order function that returns the identity regardless of argument x.
- $\lambda x.\lambda f.fx$ returns a higher order function

3 Parsing issues

Note that the grammar is ambiguous. For instance, the expression xyz can be parsed in two different ways:

$$\begin{array}{ccccc} & \text{apply} & & & \\ \swarrow & & \searrow & & \\ x & & \text{apply} & & \\ & \swarrow & & \searrow & \\ & y & & & z \end{array} \qquad \begin{array}{ccccc} & & & \text{apply} & \\ & & \swarrow & & \searrow \\ & \text{apply} & & & z \\ \swarrow & & \searrow & & \\ x & & y & & \end{array}$$

We will not worry about these parsing issues. Like before, we will assume that programs are augmented with a sufficient number of parentheses as to avoid ambiguities. Nevertheless, we will use the following two implicit parsing rules, which will reduce the number of parantheses required to make expressions unambiguous:

1. Application is left-associative. For example, xyz is parsed as $(xy)z$ instead of $x(yz)$;
2. Bodies of λ-abstractions extend as much as possible to the right. For example, the following expression: $\lambda x.\lambda y.yx$ will be parsed as $\lambda x.(\lambda y.yx)$ and not as $\lambda x.(\lambda y.y)x$.

4 Scope

We would like to define what is a valid expression in our language. For instance is the following expressions valid: $\lambda x.y$? Or y? The problem is that y is not in the scope of any enclosing λ.

We define the following concepts:

- an occurrence of a variable x is *bound* if it occurs in the body t of an abstraction $\lambda x.\, t$.
- otherwise an occurrence of a variable is *free*.
- *closed terms* (also called *combinators*) are expressions where all occurrences of all variables are bound.

The simplest combinator is the identity, $\lambda x.x$. The following expression: $(\lambda x.\, x)x$ is not a combinator, since the last occurrence of x is free, although the occurrence in the body of $\lambda x.\, x$ is bound. That is why the above definitions refer to occurrences of variables rather than variables.

5 Semantics

To compute with λ-calculus, we apply functions to their arguments, as in the following example:

$$\begin{array}{ll} & ((\lambda x.x)(\lambda y.y))(\lambda z.z) \\ \rightarrow & (\lambda y.y)(\lambda z.z) \\ \rightarrow & \lambda z.z \end{array}$$

Computations in λ-calculus are series of applications of the form $(\lambda x.e)e' \rightarrow e[x \rightarrow e']$, called *$\beta$-reductions*. The way to perform these substitutions is not trivial because there may be occurrences of the same variable with different bindings. This will be discussed in a later lecture. An expression $(\lambda x.e)e'$, which can be reduced in this way, is called a *reducible expression*, or *redex*.

One problem is that an arbitrary expression may contain multiple reducible expressions, so we must choose which one to reduce first. Consider for example $e = (\lambda x.x)((\lambda y.y)(\lambda z.(\lambda t.t)z))$ or, for short, $id(id(\lambda z.id\ z))$, where id is the identity function $id = \lambda x.x$. The expresion e contains 3 reducible expressions.

The different orders in which the expressions correspond to the two ways of passing parameters to functions: *call-by-name* (CBN) and *call-by-value* (CBV).

6 Order of evaluation

Applying the functions without first reducing their arguments is analogous to CBN. Using this method, the above expression is evaluated as follows:

$$\begin{array}{ll} & id(id(\lambda z.id\ z)) \\ \rightarrow & x[x \rightarrow id(\lambda z.id\ z)] \equiv id(\lambda z.id\ z) \\ \rightarrow & x[x \rightarrow \lambda z.id\ z] \equiv \lambda z.id\ z \end{array}$$

We stop at this point because this is an abstraction and not a function application.

Reducing the arguments before the function is applied is analogous to CBV. Then our expression evaluates in the following way:

$$\begin{array}{ll} & id(id(\lambda z.id\ z)) \\ \rightarrow & id(x[x \rightarrow \lambda z.id\ z]) \equiv id(\lambda z.id\ z) \\ \rightarrow & x[x \rightarrow \lambda z.id\ z] \equiv \lambda z.id\ z \end{array}$$

This is the same result as with CBN, but the expressions are evaluated in a different order.

7 Operational Semantics

We will use operational semantics to formally define the way we evaluate expressions in λ-calculus. The rules in operational semantics will define a mapping $e \to e'$, i.e., reducing an expression to another expression.

7.1 CBN λ-calculus

First, try to reduce e_0 as much as possible:

$$\frac{e_0 \to_1 e_0'}{e_0 e_1 \to_1 e_0' e_1}$$

Then apply:

$$\frac{}{(\lambda x.e_0)e_1 \to_1 e_0[x \to e_1]} \ (\beta - reduction)$$

7.2 CBV λ-calculus

First, try to reduce e_0 as much as possible:

$$\frac{e_0 \to_1 e_0'}{e_0 e_1 \to_1 e_0' e_1}$$

Second, try to reduce e_1 as much as possible:

$$\frac{e_1 \to_1 e_1'}{v e_1 \to_1 v e_1'} \text{ , where } v = \lambda x.e$$

Lastly, apply the rule:

$$\frac{}{(\lambda x.e_0)v \to_1 e_0[x \to v]} \ (\beta - reduction)$$

We observe that every rule in the operational semantics that we just presented has at most one premise. Thus, this is the small-step operational semantics for λ-calculus. Hence the use of the symbol $\to_1$ in the rules.

8 Non-termination

With the previous languages that we studied, we saw that non-terminating programs could be constructed. This can also be done in λ-calculus. First, we define the function *self*:

$$\textit{self} = \lambda x.xx$$

Now we consider the function *self self* and try to reduce it:

$$\textit{self self} = (\lambda x.xx)(\lambda x.xx) \to_1 xx[x \to \lambda x.xx] = (\lambda x.xx)(\lambda x.xx) = \textit{self self}$$

Because we come to the same expression that we started with, we know that this process will not terminate.

As in REC, when there is non-termination, the CBN and CBV evaluation methods may give different results:

CBN:

$$(\lambda x(\lambda y.y))(\textit{self self}) \to_1^{CBN} \lambda y.y[x \to \textit{self self}] = \lambda y.y$$

Thus, the evaluation terminates.

CBV:

$$\frac{\textit{self self} \to_1^{CBV} \textit{self self}}{(\lambda x.(\lambda y.y))(\textit{self self}) \to_1^{CBV} (\lambda x.(\lambda y.y))(\textit{self self})}$$

In one step, this expression reduces to itself, so the evaluation does not terminate when using the CBV operational semantics.

9 Large-step Operational Semantics

The rules for large-step operational semantics of λ-calculus are:

CBN:

$$\frac{e_0 \to \lambda x.e_2 \quad e_2[x \to e_1] \to v}{e_0 e_1 \to v} \qquad\qquad \frac{}{v \to v}$$

CBV:

$$\frac{e_0 \to \lambda x.e_2 \quad e_1 \to v_1 \quad e_2[x \to v_1] \to v}{e_0 e_1 \to v} \qquad\qquad \frac{}{v \to v}$$

where $v := \lambda x.e$.
Note: in both cases, we need the axiom which shows that abstractions don't need to be futher reduced.