CS611 Lecture 20 **The Untyped Lambda Calculus** October 15, 2003

Scribe: Milos Hasan, Wojciech Moczydlowski, Jr Lecturer: Radu Rugina

1 Definition of Lambda Calculus

In the languages we considered until now, functions were not first-class values. Now we will introduce λ-calculus which is a base to many languages which do not have this deficit. The language we present is untyped, in contrast to the typed lambda calculus of which we will talk in the later lectures.

First we will define a syntax for terms in λ-calculus.

$$e ::= x \mid e_0\ e_1 \mid \lambda x.e$$

The term is either a variable (x), or an application of term to a term ($e_0\ e_1$) or a λ-abstraction ($\lambda x.e$).

The intuition standing behind those terms is: in application $e_0 e_1$, e_0 can be thought of as a function, and e_1 as its argument. And the way to think about $\lambda x.e$ is that it is an anonymous function, taking x as an argument and returning e, where e can have x as a free variable.

In functional languages λ-abstraction is written in different ways. I. e. in ML we write: `fn x => e`, in Haskell: `\ x -> e`, and in LISP: `(lambda (x) (e))`.

Let's present several examples of λ-terms:

- $\lambda x.x$ — this can be thought of as an identity function and is usually denoted by I.
- $\lambda x.\lambda y.y$ — this takes two arguments and returns the second one. This term is traditionally denoted as K^*.
- $\lambda x.\lambda y.\lambda z.z\ y\ x$.

Note that the grammar is ambiguous – the expression $\lambda x.\lambda y.y\ x$ can be parsed either as $\lambda x.(\lambda y.y\ x)$ or as $\lambda x.(\lambda y.y)\ x$. We choose not to care and expect the user (student) to provide us with the unambigous expression. However, there are some conventions we will assume:

- Application is left-associative — $x\ y\ z$ parses as $(x\ y)\ z$.
- The body of an abstractions extends to the right as much as possible (to the first closing parenthesis).

1.1 Scope

The occurence of variable x in a term is *bound*, if there is an enclosing $\lambda x.e$. In the other case it is *free*. The term is *closed* if all its variables are bound. We call closed terms *combinators*.

As an example of this definition in use, in $(\lambda x.x)x$, the first x is bound while the second is free.

For now, we only consider combinators as valid terms.

2 Computation in Lambda Calculus

In order to be able to speak about lambda calculus as a programming language, we need some kind of semantics. First define a relation of β-reduction, which corresponds to the idea of applying a function $\lambda x.e$ to its argument e_1 — intuitively, what results, is the body of the function e with e_1 substituted for x. The following definition captures this intuition:

$$\frac{}{(\lambda x.e)\ e' \to_\beta e[x \mapsto e']}$$

We call the $(\lambda x.e)\ e'$ the β-redex and $e[x \mapsto e']$ its contractum. We can easily extend the definition to all terms, covering the intuition that a term β-reduces to the other, if some redex in its body reduces to its contractum. There are many way of extending this definition; in this lecture today we will discuss two of them: call by value and call by name.

The expression $e[x \mapsto e']$ represents *sustitution*; it denotes the expression e, with x substituted by e'. Defining substitution formally is a little tricky; we will do that in one of the following lectures.

An example computation:

$$(\lambda x.\lambda y.y\ x)\ (\lambda z.z)\ (\lambda t.t) \rightarrow_\beta (\lambda y.y\ (\lambda z.z))\ (\lambda t.t) \rightarrow_\beta (\lambda t.t)\ (\lambda z.z) \rightarrow_\beta \lambda z.z$$

3 Semantics of Lambda Calculus

Now we will define, similarly to REC, call-by-name (CBN) and call-by-value (CBV) semantics, and in both flavors — small-step and large-step. With call-by-value, an application $e_0 e_1$ evaluates its argument e_1 before β-reducing it. With call-by-name semantics, it reduces the application, without evaluating e_1.

3.1 Small-step CBN Semantics

Small-step CBN semantics is defined by these rules:

$$\frac{e_0 \rightarrow_1 e_0'}{e_0\ e_1 \rightarrow_1 e_0'\ e_1} \qquad \frac{}{(\lambda x.e)\ e' \rightarrow_1 e[x \mapsto e']}$$

Here we present an example computation:

$$(\lambda x.x)\ ((\lambda y.y)(\lambda z.(\lambda t.t)\ z) \rightarrow_1 (\lambda y.y)(\lambda z.(\lambda t.t)\ z) \rightarrow_1 \lambda z.(\lambda t.t)\ z$$

3.2 Small-step CBV Semantics

These are the rules that define small-step CBV semantics:

$$\frac{e_0 \rightarrow_1 e_0'}{e_0\ e_1 \rightarrow_1 e_0'\ e_1} \qquad \frac{e_1 \rightarrow_1 e_1'}{(\lambda x.e)\ e_1 \rightarrow_1 (\lambda x.e)\ e_1'} \qquad \frac{}{(\lambda x.e)\ (\lambda y.e') \rightarrow_1 e[x \mapsto \lambda y.e']}$$

An example evaluation:

$$(\lambda x.x)\ ((\lambda y.y)(\lambda z.(\lambda t.t)\ z) \rightarrow_1 (\lambda x.x)(\lambda z.(\lambda t.t)\ z) \rightarrow_1 \lambda z.(\lambda t.t)\ z$$

Note that it exactly corresponds to the idea of CBV — the argument is reduced first, then the reduced argument is passed to the function.

3.3 Large-step CBN Semantics

$$\frac{}{\lambda x.e \rightarrow \lambda x.e} \qquad \frac{e_0 \rightarrow \lambda x.e \quad e[x \mapsto e_1] \rightarrow v}{e_0\ e_1 \rightarrow v}$$

3.4 Large-step CBV Semantics

$$\frac{.}{\lambda x.e \rightarrow \lambda x.e} \qquad \frac{e_0 \rightarrow \lambda x.e \quad e_1 \rightarrow v_1 \quad e[x \mapsto v_1] \rightarrow v}{e_0\ e_1 \rightarrow v}$$

The v and v_1 in the expression above are *values*. In all of the above rules (both CBN and CBV), values are abstractions:

$$v \equiv \lambda x.\, e$$

One can prove that the respective definitions using large-step and small-step semantics are equivalent in the sense of the first homework.

In all cases, we decide to stop the reduction once we reach the so-called weak head normal form (WHNF) — a top-level λ-abstraction, when in the body of it there still may be some β-redexes. The v in the above rules is a WHNF. The alternative would be to keep doing reduction, until there aren't any β-redexes in it.

4 Non-termination

There can be non-terminating expressions in lambda calculus. Consider the expression *self*, traditionally denoted as ω:

$$\mathit{self} \equiv \lambda x.x\ x$$

If we try to evaluate the expression *self self*, we get (in both CBN and CBV):

$$\mathit{self}\ \mathit{self} \equiv (\lambda x.x\ x)\ \mathit{self} \rightarrow_1 \mathit{self}\ \mathit{self} \equiv \dots$$

The expresion *self self* is traditionally denoted as Ω.

Sometimes, the reduction of an expression can terminate in CBN, but not in CBV, as in the following example:

$$(\lambda x.\lambda y.y)\ (\mathit{self}\ \mathit{self}) \rightarrow_1 \lambda y.y \quad \text{(in CBN)}$$

$$(\lambda x.\lambda y.y)\ (\mathit{self}\ \mathit{self}) \rightarrow_1 (\lambda x.\lambda y.y)\ (\mathit{self}\ \mathit{self}) \rightarrow_1 \dots \quad \text{(in CBV)}$$