Lecture notes for CS 6110 (Spring'09) taught by Andrew Myers at Cornell; edited by Amal Ahmed, Fall'09.

1 The Lambda Calculus

Lambda calculus is a notation for describing mathematical functions and programs. It is a mathematical system for studying the interaction of *functional abstraction* and *functional application*. It captures some of the essential, common features of a wide variety of programming languages. Because it directly supports abstraction, it is a much more natural model of universal computation than a Turing machine is.

1.1 Syntax

A λ-calculus term is:

1. a variable $x \in \mathbf{Var}$, where **Var** is a countable infinite set of variables;
2. a function e_0 applied to an argument e_1, usually written $e_0\ e_1$ or $e_0\ (e_1)$; or
3. a lambda term, an expression $\lambda x.\, e$ representing a function with input parameter x and body e. Where a mathematician might write $x \mapsto x^2$, in the λ-calculus we would write $\lambda x.\, x^2$.

In BNF notation,

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

Note that we used the word *term* instead of *expression*. A term is an expression that describes a computation to be performed. In general, programs may contain expressions that are not terms; for example, type expressions. However, in the untyped lambda calculus that we are now studying, all expressions are terms.

Parentheses are used just for grouping; they have no meaning on their own. Like other familiar binding constructs from mathematics (e.g., sums, integrals), lambda terms are greedy, extending as far to the right as they can. Therefore, the term $\lambda x.\, x\ \lambda y.\, y$ is the same as $\lambda x.\, (x\ (\lambda y.\, y))$, not $(\lambda x.\, x)\ (\lambda y.\, y)$.

For simplicity, multiple variables may be placed after the lambda, and this is considered shorthand for having a lambda in front of each variable. For example, we write $\lambda x, y.\, e$ as shorthand for $\lambda x.\, \lambda y.\, e$. This shorthand is an example of *syntactic sugar*. The process of removing it in this instance is called *currying*.

We can apply a curried function like $\lambda x.\, \lambda y.\, x$ one argument at a time. Applying it to one argument results in a function that takes in a value for x and returns a constant function, one that returns the value of x no matter what argument it is applied to. As this suggests, functions are just ordinary values, and can be the results of functions or passed as arguments to functions (even to themselves!). Thus, in the lambda calculus, functions are first-class values. Lambda terms serve both as functions and data.

1.2 Recap—BNF Notation

In the grammar

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

describing the syntax of the pure λ-calculus, the e is not a variable in the language, but a *metavariable* representing a syntactic class (in this case λ-terms) in the language. It is not a variable at the level of the programming language. We use subscripts to differentiate syntactic metavariables of the same syntactic class. For example, e_0, e_1, and e all represent λ-terms.

1.3 Recap—Variable Binding

Occurrences of variables in a λ-term can be *bound* or *free*. In the λ-term $\lambda x.\, e$, the lambda abstraction operator λx binds all the free occurrences of x in e. The *scope* of $\lambda x\ \lambda x.\, e$ is e. This is called *lexical scoping*; the variable's scope is defined by the text of the program. It is "lexical" because it is possible to determine its scope before the program runs by inspecting the program text. A term is *closed* if all variables are bound. A term is *open* if it is not closed.

1.4 Digression—Terms and Types

There are different kinds of expressions in a typical programming language: *terms* and *types*. We have not talked about types yet, but we will soon. A term represents a value that exists only at run time; a type is a compile-time expression used by the compiler to rule out ill-formed programs. For now there are no types.

2 Substitution and β-reduction

Now we get to the question: How do we run a λ-calculus program? The main computational rule is called *β-reduction*. This rule applies whenever there is a subterm of the form $(\lambda x.\, e)\ e'$ representing the application of a function $\lambda x.\, e$ to an argument e'.

To perform a β-reduction, we substitute the argument e' for all free occurrences of the formal parameter x in the body e. This corresponds to our intuition for what the function $\lambda x.\, e$ means.

We have to be a little careful; we cannot just substitute e' blindly for x in e, because bad things could happen which could alter the meaning of expressions in undesirable ways. We only want to replace the free occurrences of x within e, because any other occurrences are bound to a different binding; they are really different variables. There are some additional subtleties to substitution that well return to later.

There are many notations for substitution, which can be confusing. Pierce writes $[x \mapsto e']e$. Other notations for the same idea are encountered frequently, including $e[x \mapsto e']$, $e[x \leftarrow e']$, $e[x := e']$. Because we will be using brackets for other purposes, we will use the notation $e\{e'/x\}$.

Rewriting $(\lambda x.\, e)\ e'$ to $e\{e'/x\}$ is the basic computational step of the λ-calculus. In the pure λ-calculus, we can start with a term and perform β-reductions on subterms in any order. However, for modeling programming languages, it is useful to restrict which β-reductions are allowed and in what order they can be performed.

3 Reduction Strategies

In general there may be many possible β-reductions that can be performed on a given λ-term. How do we choose which beta reductions to perform next? Does it matter?

A specication of which β-reduction to perform next is called a *reduction strategy*. In class we discussed four possible reduction strategies: full β-reduction, normal order, call-by-value, and call-by-name. See Pierce, pg 55 through 57 (subsection on *Operational Semantics*) for a full discussion. In class we discussed the evaluation of the term

$$(\lambda x.\, \lambda y.\, 1\ x)\ (\lambda z.\, z\ 3)$$

under each of these four reduction strategies.

Let us define a value to be a closed λ-term to which no β-reductions are possible, given our chosen reduction strategy. For example, $\lambda x.\, x$ would always be a value, whereas $(\lambda x.\, x)\ 1$ would most likely never be.

Most real programming languages based on the λ-calculus use a reduction strategy known as *Call By Value* (CBV). In other words, functions may only be applied to (or called on) values. Thus $(\lambda x.\, e)\ e'$ only reduces if e' is a value v. Here is an example of a CBV evaluation sequence, assuming tru, fls, and id (the identity function) are appropriately defined (and assuming that they are values).

$$((\lambda x.\, \lambda y.\, y\ x)\ \mathsf{tru})\ \mathsf{id} \longrightarrow (\lambda y.\, y\ \mathsf{tru})\ \mathsf{id} \longrightarrow \mathsf{id}\ \mathsf{tru} \longrightarrow \mathsf{tru}$$

Another strategy is *Call By Name* (CBN). We defer evaluation of arguments until as late as possible, applying reductions from left to right within the expression. In other words, we can pass an incomplete computation to a function as an argument. Terms are evaluated only once their value is really needed.

CBV is referred to as an *eager* or *strict* reduction strategy, whereas CBN is referred to *lazy* or *non-strict*.

4 Structural Operational Semantics (SOS)

Let's formalize CBV. First, we need to define the values of the language. These are simply the lambda terms:

$$v \ ::= \ \lambda x.\, e$$

Here we use the metavariable v to range over values.

Next, we can write inference rules to define when reductions are allowed:

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \qquad \frac{}{\lambda x.\, e\ v \longrightarrow e\{v/x\}}\ \beta\text{-reduction}$$

This is an example of an operational semantics for a programming language based on the lambda calculus. An operational semantics is a language semantics that describes how to run the program. This can be done through informal human-language text, as in the Java Language Specication, or through more formal rules. Rules of this form are known as a Structural Operational Semantics (SOS). They define evaluation as the result of applying the rules to transform the expression, and the rules are defined in term of the structure of the expression being evaluated.

This kind of operational semantics is known as a *small-step* semantics because it only describes one step at a time, by defining a transition relation $e \longrightarrow e'$. A program execution consists of a sequence of these small steps strung together. An alternative form of operational semantics is a *big-step* (or *large-step*) semantics that describes the entire evaluation of the program to a final value.

As defined above, CBV evaluation is deterministic: there is only one evaluation leading from any given term. (We leave proving this for later). If we allow evaluation to work on the right-hand side of an application, evaluation will be nondeterministic:

$$\frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'}$$

We will see other kinds of semantics later in the course, such as *axiomatic semantics*, which describes the behavior of a program in terms of the observable properties of the input and output states, and *denotational semantics*, which translates a program into an underlying mathematical representation.

Expressed as SOS, CBN has slightly simpler rules:

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{}{\lambda x.\, e\ e' \longrightarrow e\{e'/x\}}\ \beta\text{-reduction}$$

We don't need the rule for evaluating the right-hand side of an application because β-reductions are done immediately once the left-hand side is a value.

4.1 Recursion

Let us define an expression we will call Ω:

$$\Omega \ = \ (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

What happens when we try to evaluate it?

$$\Omega \ = \ (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \ \longrightarrow \ (x\ x)\{(\lambda x.\, x\ x)/x\} \ = \ \Omega$$

We have just coded an infinite loop! When an expression e can go through infinite sequence of evaluation steps, we write $e \Uparrow$. When it evaluates to a value v, we write $e \Downarrow v$ or just $e \Downarrow$ if we don't care what the value is. (More formally, $e \Downarrow$ is an abbreviation that denotes $\exists v.\ e \Downarrow v$.)

What happens if we try using Ω as a parameter? It depends. Consider this example:

$$e \ = \ (\lambda x.\, (\lambda y.\, y))\ \Omega$$

Using the CBV evaluation strategy, we must first reduce Ω. This puts the evaluator into an infinite loop, so $e \Uparrow_{CBV}$. On the other hand, CBN reduces the term above to $\lambda y.\, y$. CBN has an important property: CBN will not loop infinitely unless every other semantics would also loop infinitely, yet it agrees with CBV whenever CBV terminates successfully:

$$e \Downarrow_{CBV} \implies e \Downarrow_{CBN}$$