

## *Handout 4: Reduction semantics*

Lambda calculus can be treated as a rudimentary programming language. The computations in this language are carried out by applying the conversion laws of the lambda calculus. This form of computation is called “reduction,” to signify that we are reducing a term to something resembling its value. The study of the reduction mechanism is dubbed “reduction semantics.”

### Conversion laws

#### 1. Free variables

Recall that the  $\lambda$  construction *binds* a variable, i.e., it makes it a local variable. For example, in the term  $\lambda x. y x$ , the  $\lambda$  construction binds  $x$ . The intuitive meaning of this term is “receive an argument, call it  $x$ , apply the function  $y$  to  $x$  and return the result.” As you can see,  $x$  is a local variable name used in this statement. Outside the context of this term,  $x$  has no meaning. We say that  $x$  is a *bound variable* in the term  $\lambda x. y x$ . The variables that are not bound are said to be *free*. So,  $y$  is a free variable in term  $\lambda x. y x$ . If we consider a larger term  $\lambda y. \lambda x. y x$ , both  $x$  and  $y$  are bound variables in this term. It has no free variables.

To precisely define the free variables in a term, we use a function  $FV$  which, when applied to a lambda term, gives the set of its free variables. The definition is as follows:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x. M) &= FV(M) \setminus \{x\} \\ FV(M_1 M_2) &= FV(M_1) \cup FV(M_2) \end{aligned}$$

We could also state the content of this definition in English as follows:

- The free variables of the term  $x$  include just  $x$ .
- The free variables of the term  $\lambda x. M$  are all the free variables of  $M$  except  $x$ .
- The free variables of the term  $M_1 M_2$  include all the free variables of  $M_1$  and all the free variables of  $M_2$ .

For example, we can calculate the free variables of the somewhat tricky term  $x(\lambda x. yx)$  as follows:  $FV(x\lambda x. yx) = FV(x) \cup FV(\lambda x. yx) = \{x\} \cup (FV(yx) \setminus \{x\}) = \{x\} \cup (\{y, x\} \setminus \{x\}) = \{x\} \cup \{y\} = \{x, y\}$ . Note that the first occurrence of  $x$  in this term is a free occurrence.

#### 2. Substitution

The conversion laws of the lambda calculus involve substituting arguments for the formal parameters of functions. To simplify the function application  $(\lambda x. y x)0$ , we substitute the formal parameter  $x$  by  $0$  in the body  $y x$ , and obtain  $y 0$ .

We use the notation  $M[x := N]$  to mean the term obtained by substituting  $N$  for  $x$  in the term  $M$ . So,  $(yx)[x := 0]$  means substitute  $0$  for  $x$  in  $y x$ . The result of substitution is  $y 0$ . When we do such substitution, only the *free occurrences* of the variable should be substituted. Any bound occurrences of the variable are left alone. Consider, for example, the substitution  $(x \lambda x. y x)[x := 0]$ . The result is  $0 \lambda x. y x$ . The second occurrence of  $x$  is a bound occurrence. So, it is left alone.

A substitution  $M[x := N]$  is said to be *valid* if all free variables of  $N$  remain free in the result of the substitution. The substitutions mentioned above are valid because  $0$  remains free. However, consider the substitution  $(\lambda x. y x)[y := f x]$ . The result of substitution is  $\lambda x. f x x$  in which the variable  $x$  is not free any more. So this substitution is not valid. The problem is that the meaning of the term  $f x$  is altered when we bring it into the scope of the binding  $\lambda x. .$  This kind of situation is termed “variable capture”. To avoid variable capture, we can rename the bound variables of the term, e.g.,  $(\lambda x'. y x')[y := f x]$ . Now the substitution gives gives  $\lambda x'. f x x'$ .

We can precisely define the effect of a substitution  $M[x := N]$  by an inductive definition as follows:

$$\begin{aligned} y[x := N] &= \begin{cases} N, & \text{if } y \text{ is the same as } x \\ y, & \text{if } y \text{ is different from } x \end{cases} \\ (\lambda y. M)[x := N] &= \begin{cases} \lambda y. M, & \text{if } y \text{ is the same as } x \\ \lambda y. (M[x := N]), & \text{if } y \text{ is different from } x \text{ and } y \notin FV(N) \end{cases} \\ (M_1 M_2)[x := N] &= M_1[x := N] M_2[x := N] \end{aligned}$$

In the second rule above, we don't have a coverage of all cases. If  $y$  is *not* a free variable of  $N$ , then the definition says how to substitute inside  $M$ . But, if  $y$  is a free variable of  $N$ , such substitution would be invalid. So, it is not possible to do the substitution. The only way out is to rename the bound variable  $y$  using the  $\alpha$ -conversion law below.

### 3. $\alpha$ -conversion

The first conversion law of lambda calculus states that we can rename bound variables:

$$\lambda x. M \equiv \lambda y. (M[x := y])$$

This is called  $\alpha$  conversion. This form of renaming needs to be done often. So, we will treat equivalence under this law as if it were identity. In other words, we assume that bound variables can always be silently renamed to other variables as needed in every context.

### 4. $\beta$ -conversion

The second conversion law states how to simplify applications of  $\lambda$ -functions:

$$(\lambda x. M) N \equiv M[x := N]$$

We have already seen several examples of this conversion. When the conversion law is applied left to right, i.e., to simply function applications, it is called " $\beta$ -reduction."

### 5. $\eta$ -conversion

The third conversion law states that if all we do inside a function  $\lambda x. \dots$  is to apply some function  $M$  to  $x$ , then this is just equivalent to the function  $M$ :

$$\lambda x. M x \equiv M \quad \text{if } x \notin FV(M)$$

For example,  $\lambda x. \mathbf{succ} x$  is simply  $\mathbf{succ}$ .

### 6. Equivalence

Two lambda terms  $M$  and  $N$  are said to be *equivalent* if their equivalence can be shown in some finite number of steps by replacing equals by equals as per the three conversion laws. More precisely,  $M$  and  $N$  are equivalent if there is a sequence of terms:

$$M_1 \equiv M_2 \equiv \dots \equiv M_n$$

where  $M_1$  is  $M$  and  $M_n$  is  $N$ , and each equivalence step  $M_i \equiv M_{i+1}$  is obtained by applying a conversion law to some *subterm* of  $M_i$ . Applying a conversion law means applying it either left-to-right or right-to-left, because equivalence is a symmetric relation.

Note that the conversion laws can always be applied to *subterms* of the given term. For example, consider the sequence of steps  $((\lambda x. \lambda y. \mathbf{add} x y) 2) 3 \equiv (\lambda y. \mathbf{add} 2 y) 3 \equiv \mathbf{add} 2 3$ . In the first step, we have applied  $\beta$ -equivalence to the underlined subterm, not the entire term. This follows our normal intuitive sense of equivalence. If  $M_1$  and  $M_2$  are equivalent (meaning, they represent the same values), then they are also equivalent when used in some context:  $\dots M_1 \dots \equiv \dots M_2 \dots$ . In our example above:  $\dots (\lambda x. \lambda y. \mathbf{add} x y) 2 \dots \equiv \dots (\lambda y. \mathbf{add} 2 y) \dots$

### 7. $\beta$ -reduction

The most useful conversion in lambda calculus is the  $\beta$ -conversion used left-to-right. Used in this way, the  $\beta$ -conversion law serves as an "evaluation" mechanism. It allows us to simplify function applications leading to their values. This use of the  $\beta$ -conversion law is termed " $\beta$ -reduction" and denoted by the symbol " $\longrightarrow_\beta$ ".

We say that  $M_1$   $\beta$ -reduces to  $M_2$  and write  $M_1 \longrightarrow_\beta M_2$  if

- there is some subterm of  $M_1$  which is of the form  $(\lambda x. M) N$  and
- $M_2$  is obtained by replacing this term with  $M[x := N]$ .

For example,  $((\lambda x. \lambda y. \mathbf{add} x y) 2) 3 \longrightarrow_\beta (\lambda y. \mathbf{add} 2 y) 3 \longrightarrow_\beta \mathbf{add} 2 3$ .

### 8. $\beta$ -redex

A subterm of the form  $(\lambda x. M) N$  is called a  $\beta$ -redex. (The term "redex" is just an acronym for "reducible expression.") Terms often have many  $\beta$ -redexes. For example, the term  $(\lambda x. x) ((\lambda y. z) 2)$  has two  $\beta$ -redexes, as shown by the two underlinings:

$$\begin{array}{c} \underline{(\lambda x. x)} ((\lambda y. z) 2) \\ (\lambda x. x) \underline{((\lambda y. z) 2)} \end{array}$$

The choice of each redex gives rise to a  $\beta$ -reduction:

$$\begin{aligned} (\lambda x. x) ((\lambda y. z) 2) &\longrightarrow_{\beta} (\lambda y. z) 2 \\ (\lambda x. x) ((\lambda y. z) 2) &\longrightarrow_{\beta} (\lambda x. x) z \end{aligned}$$

However, if we do another reduction step, we obtain the same final term in each case:

$$\begin{aligned} (\lambda x. x) ((\lambda y. z) 2) &\longrightarrow_{\beta} (\lambda y. z) 2 \longrightarrow_{\beta} z \\ (\lambda x. x) ((\lambda y. z) 2) &\longrightarrow_{\beta} (\lambda x. x) z \longrightarrow_{\beta} z \end{aligned}$$

This is no accident.  $\beta$ -reduction satisfies a property called *confluence* (defined below) which ensures that the choice of  $\beta$ -redexes doesn't matter. We obtain the same final term in the end.

So, the way to evaluate terms in the lambda calculus is to repeatedly reduce the  $\beta$ -redexes in a term until there are no further redexes left. This gives a reduction sequence of the form:

$$M \longrightarrow_{\beta} M_1 \longrightarrow_{\beta} M_2 \cdots \longrightarrow_{\beta} M_n$$

If we obtain a final term  $M_n$  which cannot be reduced any further, it is called the *normal form* of  $M$ . Obviously, a normal form is a term that does not have any  $\beta$ -redexes.

The fact that  $M$  reduces to  $M_n$  in some number of steps is denoted by  $M \longrightarrow_{\beta}^* M_n$ . The  $*$  superscript means the relation obtained by composing repeated  $\longrightarrow_{\beta}$  steps, but zero repetitions are also allowed. In the zero case, the reduction sequence is empty, i.e.,  $M$  is itself a normal form.

## 9. Confluence property

The confluence property of  $\beta$ -reduction is stated as follows. If a term  $M$  reduces to two possibly different terms  $N_1$  and  $N_2$ , then there is a common term  $P$  such that both  $N_1$  and  $N_2$  reduce to  $P$ . Symbolically,

$$\text{if } M \longrightarrow_{\beta}^* N_1 \text{ and } M \longrightarrow_{\beta}^* N_2 \implies \exists P. N_1 \longrightarrow_{\beta}^* P \text{ and } N_2 \longrightarrow_{\beta}^* P$$

(This property is also sometimes called the **Church-Rosser property**, even though the property proved by Church and Rosser is slightly different to confluence.)

The confluence property implies that if a term  $M$  has a normal form then the normal form is unique. (If there were two distinct normal forms, say  $N_1$  and  $N_2$ , then confluence would require that there is a common term  $P$  that both  $N_1$  and  $N_2$  reduce to  $P$ . But,  $N_1$  and  $N_2$  are supposed to be normal forms. So they are irreducible. The only way the confluence property can be satisfied is by  $N_1$  and  $N_2$  being the same.)

## 10. Termination issues

Even though confluence guarantees that all terminating reduction sequences lead to the same normal form, it says nothing about infinite reduction sequences. So, it is possible for one to make repeatedly bad choices of  $\beta$ -redexes in such a way that the normal form is never reached. Here is a simple example:

$$(\lambda y. 0) ((\lambda x. x x) (\lambda x. x x)) \longrightarrow_{\beta} 0$$

However if we were to always choose the inner  $\beta$ -redex, we do not obtain a normal form:

$$(\lambda y. 0) ((\lambda x. x x) (\lambda x. x x)) \longrightarrow_{\beta} (\lambda y. 0) ((\lambda x. x x) (\lambda x. x x)) \longrightarrow_{\beta} \cdots$$

So, the reduction strategy, i.e., the strategy of choosing the redex to reduce, is an important one.

Notice that in the above example, we have an “outer” redex and an “inner” redex. The inner redex is a subterm of the outer redex. We have noticed that the outer redex leads to a normal form but the inner redex leads to an infinite reduction sequence. This suggests that choosing the outer redexes is a better strategy.

We can explain why this happens. Functions can be constant in their arguments. For example,  $\lambda y. 0$  is a constant function. When a constant function is applied to an argument, it is pointless to reduce the argument term. Whatever the argument term might be, it will be ignored by the constant function. Choosing outer redexes allows the arguments of constant functions to be discarded early in the reduction. Therefore, it saves work, sometimes an infinite amount of work.

## 11. Normal order reduction

The reduction strategy whereby, at each step of reduction, the *leftmost outermost redex* is reduced is called the *normal order reduction strategy*. It is possible to show that if a term has a normal form, then normal order reduction leads to the normal form. In other words, if any reduction sequence terminates, then normal order reduction terminates.

Let us consider why “leftmost outermost” redexes are chosen. We have already noted that outer redexes should be preferred because they allow arguments to constant functions to be discarded early. If there is a single outermost redex then this is a clear choice.

If there are multiple outermost redexes, this can only happen if the term has an application of the form  $M_1 M_2$  at the outer level,  $M_1$  is not a  $\lambda$  abstraction, and there are redexes in  $M_1$  as well as  $M_2$ . In this case, it is clear that we should reduce the redexes in  $M_1$  first, because that might lead to  $M_1$  reducing to a  $\lambda$  abstraction, creating a  $\beta$ -redex at the outer level:

$$M_1 M_2 \longrightarrow_{\beta}^* (\lambda x. N) M_2 \longrightarrow_{\beta} N[x := M_2]$$

Since  $\lambda x. N$  can be potentially a constant function, reducing any redexes in  $M_2$  ahead of time would be wasteful. Therefore, if there are multiple outermost redexes, then the one that is leftmost is chosen for reduction.

## 12. Call by name

The normal order reduction strategy is closely related to a procedure mechanism in programming languages called **call by name**. This refers to calling procedures by passing arguments to them by “name,” i.e., by their defining expressions as opposed to their values. This is quite unlike how parameter passing works in Java and most other languages, which use the mechanism of **call by value**.

To see the difference, consider a function call of the form  $f(5 * 4)$ . In a Java-like language, the expression  $5 * 4$  is evaluated and its value 20 is passed as argument to  $f$ . In contrast, in a language with call-by-name procedures, the expression  $5 * 4$  is passed to the procedure  $f$  unevaluated. If the procedure needs the value of the argument, it will use it in some context that forces its evaluation, and the computation of  $5 * 4$  is carried out. If the procedure does not need the value of the argument, then the computation of  $5 * 4$  is never carried out.

Let us show a somewhat more interesting application of this technique. Consider an applied lambda calculus with integers, booleans and cons-pairs, and define a function called *and* as follows:

$$\text{and} = \lambda p. \lambda q. \text{if } p \text{ } q \text{ } \text{false}$$

Now, suppose we have a linked list built using cons-pairs, and we want to write an expression to test if the first element of the list is negative. The list could be potentially empty. So, we cannot assume that there is necessarily a first element in the list. The expression we write to check for the condition uses the *and* function:

$$\text{and } (\text{not } (\text{null } l)) \text{ } (< (\text{car } l) 0)$$

If *and* is a call-by-name procedure, the arguments are passed to it by “name”. The *and* procedure, as defined above, first evaluates the first argument (treating it as  $p$ ). If the value turns out to be true, i.e.,  $l$  is non-empty, then it evaluates the second argument which accesses the first element of  $l$ . If the first argument turns out to be false, i.e.,  $l$  is nil, then the *and* procedure *ignores* the second argument and returns false. This is precisely the desired behaviour of the expression.

If, on the other hand, *and* is a call-by-value procedure, both its arguments are evaluated *before* the function is called. This is not the desired behaviour: if the list  $l$  is empty, it leads to a run-time error because  $\text{car } l$  does not exist.

## 13. Applicative order reduction

The reduction strategy whereby, at each reduction step, an innermost redex is chosen, is called the applicative order reduction. This reduction corresponds to call-by-value parameter passing used in Java and most other languages.

Applicative order reduction does not have good termination behaviour. It can go on infinitely, even when there are normal forms that can be reached by normal order reduction.

However, there is an advantage to applicative order reduction. It ensures that every redex in the original term is reduced exactly once. Normal order reduction, on the other hand, can potentially make multiple copies of arguments and, hence their redexes, which leads to duplication of work. The applicative order reduction avoids duplication, but at the cost of doing unnecessary work.

A new mechanism called **call by need** or **lazy evaluation** was devised by programming language theorists to combine the advantages of normal order and applicative order reductions. In this system, the arguments are still passed by “name”. However, only one copy of the argument is maintained and all occurrences of the corresponding formal parameter refer to the same copy. This ensures that the argument term is evaluated at most once. The **Haskell** programming language uses this mechanism for evaluation.