

## 1 Rewrite Rules

### 1.1 Recap— $\beta$ -reduction

Recall that  $\beta$ -reduction is the following rule:

$$(\lambda x. e) e' \xrightarrow{\beta} e\{e'/x\}.$$

An instance of the left-hand side is called a *redex* and the corresponding instance of the right-hand side is called the *contractum*. For example,

$$\lambda x. \underbrace{(\lambda y. y)x}_{\beta \text{ redex}} \xrightarrow{\beta} \lambda x. x$$

Note that in CBV,  $\lambda x. (\lambda y. y) x$  is a value (we cannot apply a  $\beta$ -reduction inside the body of an abstraction) so we cannot apply this reduction.

### 1.2 Variable capture

CBN and CBV evaluation reduce  $\beta$  redexes  $(\lambda x. e) e'$  only when the RHS  $e'$  is a closed term. In general we can try to normalize lambda terms by reducing redexes *inside* lambda abstractions, because the reductions should still preserve the equivalence of the terms. However, in this case the term  $e'$  may be an open term containing free variables. Substituting  $e'$  into  $e$  may cause *variable capture*. For example, consider the substitution  $(y (\lambda x. x y))\{x/y\}$ . If we replace all the unbound  $y$ 's with  $x$ , we'll get  $x (\lambda x. x x)$ .

In fact, this is a problem seen in many other mathematical contexts, such as in integral calculus, because like  $\lambda$ , the integral operator is a binder. For example, consider the following naive attempt to evaluate an integral, with a similar variable capture problem:

$$\int_0^x dy \cdot (1 + \int dx \cdot x) = (y + \int dx \cdot yx) \Big|_{y=0}^{y=x} = (x + \int x^2 dx) - 0 = (x + \int x^2 dx)$$

### 1.3 Free variables

In order to define substitution correctly, we first need to define the set of free variables of a term. We denote this by the function  $FV((\cdot)e)$ , which is defined recursively as follows:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(e_0 e_1) &= FV(e_0) \cup FV(e_1) \\ FV(\lambda x. e) &= FV(e) - \{x\} \end{aligned}$$

This definition is recursive, but notice that in each of the three cases, the right-hand side defines the value of  $FV(e)$  in terms of proper subterms of  $e$ . Since all terms have finite size, this means that if we were to expand out the definition of  $FV(e)$  for any given  $e$ , it would eventually bottom out at applications to variables (the first case). Technically, this is a definition of a function by *structural induction* on  $e$ . We will talk more about these kinds of definitions later.

## 1.4 Capture-avoiding Substitution

Now we write  $e_1\{e_2/x\}$  to denote the result of substituting  $e_2$  for all free occurrences of  $x$  in  $e_1$ , according to the following rules, also by structural induction on  $e$  (modulo  $\alpha$ -equivalence, which we'll talk about shortly):

$$\begin{aligned}
 x\{e/x\} &= e \\
 y\{e/x\} &= y && \text{where } y \neq x \\
 (e_1 e_2)\{e/x\} &= e_1\{e/x\} e_2\{e/x\} \\
 (\lambda x. e_0)\{e_1/x\} &= \lambda x. e_0 \\
 (\lambda y. e_0)\{e_1/x\} &= \lambda y. e_0\{e_1/x\} && \text{where } y \neq x \text{ and } y \notin FV(e_1) \\
 (\lambda y. e_0)\{e_1/x\} &= \lambda z. e_0\{z/y\}\{e_1/x\} && \text{where } y \neq x, z \neq x, z \notin FV(e_0), \text{ and } z \notin FV(e_1).
 \end{aligned}$$

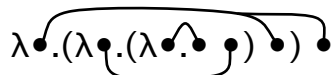
Note that the rules are applied inductively. That is, the result of a substitution in a compound term is defined in terms of substitutions on its subterms. The very last of the six rules applies when  $y \in FV(e_1)$ . In this case we can rename the bound variable  $y$  to  $z$  to avoid capture of the free occurrence of  $y$ . One might well ask: But what if  $y$  occurs free in the scope of a  $\lambda z$  in  $e_0$ ? Wouldn't the  $z$  then be captured? The answer is that it will be taken care of in the same way, but inductively on a smaller term.

Despite the importance of substitution, it was not until the mid-1950's that a completely satisfactory definition of substitution was given, by Haskell Curry. Previous mathematicians, from Newton to Hilbert to Church, worked with incomplete or incorrect definitions. It is the last of the rules above that is hard to get right, because it is easy to forget one of the three restrictions on the choice of  $y'$ , or to falsely convince yourself that they are not needed.

In the pure  $\lambda$ -calculus, we can start with a  $\lambda$ -term and perform  $\beta$ -reductions on subterms in any order, using the full substitution rule to avoid variable capture when the substituted term is open.

## 1.5 Stoy diagrams

We can create a *Stoy diagram* (after Joseph Stoy) for a closed term in the following manner. Instead of writing a term with variable names, we write dots to represent the variables and connect variables with the same binding with edges. Then  $\alpha$ -equivalent terms have the same Stoy-diagram. For example, the term  $\lambda x. (\lambda y. (\lambda x. x y) x) x$  has the following Stoy diagram:



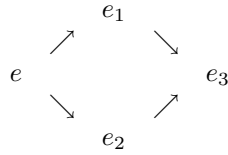
Another way to formalize the lambda calculus is to define its terms as the *equivalence classes* of the syntactic terms given, with respect to the  $\alpha$ -equivalence relation. (The equivalence classes of a set are just the subsets that are equivalent with respect to some equivalence relation.) Since all terms in the equivalence class have the same Stoy diagram, we can understand the terms as Stoy diagrams, and the reductions as operating on these diagrams. This approach is often taken in theoretical programming language work, even when the presentation appears to be using explicit variable names. A related approach is to represent variables using *de Bruijn indices*, which replace variables with natural numbers that indicate the binding site.

## 2 Confluence

In the classical  $\lambda$ -calculus, no reduction strategy is specified, and no restrictions are placed on the order of reductions. Any redex may be chosen to be reduced next. A  $\lambda$ -term in general may have many redexes, so the process is nondeterministic. We can think of a reduction strategy as a mechanism for resolving the nondeterminism, but in the classical  $\lambda$ -calculus, no such strategy is specified. A *value* in this case is just a term containing no redexes. Such a term is said to be in *normal form*.

This makes it very difficult to define equality. One sequence of reductions may terminate, but another may not. It is even conceivable that different terminating reduction sequences result in different values. Luckily, it turns out that the latter cannot happen.

It turns out that the  $\lambda$ -calculus is *confluent* (also known as the *Church–Rosser* property) under  $\alpha$ - and  $\beta$ -reductions. Confluence says that if  $e$  reduces by some sequence of reductions to  $e_1$ , and if  $e$  also reduces by some other sequence of reductions to  $e_2$ , then there exists an  $e_3$  such that both  $e_1$  and  $e_2$  reduce to  $e_3$ .



It follows that up to  $\alpha$ -equivalence, normal forms are unique. For if  $e \Downarrow v_1$  and  $e \Downarrow v_2$ , and if  $v_1$  and  $v_2$  are in normal form, then by confluence they must be  $\alpha$ -equivalent. Moreover, regardless of the order of previous reductions, it is always possible to get to the unique normal form if it exists.

However, note that it is still possible for a reduction sequence not to terminate, even if the term has a normal form. For example,  $(\lambda x. \lambda y. y)\Omega$  has a nonterminating CBV reduction sequence

$$(\lambda x y. y)\Omega \xrightarrow{\beta} (\lambda x y. y)\Omega \xrightarrow{\beta} \dots$$

but a terminating CBN reduction sequence, namely

$$(\lambda x. \lambda y. y)\Omega \xrightarrow{\beta} \lambda y. y.$$

It may be difficult to determine the most efficient way to expedite termination. But even if we get stuck in a loop, the Church–Rosser theorem guarantees that it is always possible to get unstuck, provided the normal form exists.

In *normal order*, the leftmost redex is always reduced first. This strategy is also called *normal order*. This is closely related to CBN evaluation, but also reduces inside lambdas. Like CBN, it finds a normal form if one exists, albeit not necessarily in the most efficient way. Call-by-value (CBV) is correspondingly related to *applicative order*, where arguments are reduced first.

In C, the order of evaluation of arguments is not defined by the language; it is implementation-specific. Because of this, and the fact that C has side effects, C is not confluent. For example, the value of the expression  $(x = 1) + x$  is 2 if the left operand of  $+$  is evaluated first,  $x + 1$  if the right operand is evaluated first. This makes writing correct C programs more challenging!

The absence of confluence in concurrent imperative languages is why concurrent programming is difficult. In the lambda calculus, confluence guarantees that reduction can be done in parallel without fear of changing the result.

### 3 Big-step semantics

So far we have seen a style of structural operational semantics called small-step semantics, in which each step of the evaluation is modeled as a rewrite rule operating on a redex. The original SOS was, instead, a *big-step* (also called *natural*) semantics, in which the rules give the final result of evaluation rather than the next step. Instead of defining a relation  $e \longrightarrow e'$ , the big-step semantics defines a relation  $e \Downarrow v$ , where  $v$  must be a value (that is, a lambda term). Thus, the relation ( $\Downarrow$ ) is a subset of **Term**  $\times$  **Value**.

We can define both CBV and CBN evaluation using big-step semantics. Let's start with CBV. We need just two rules:

$$\frac{}{v \Downarrow v} \text{ (VAL)} \qquad \frac{e_0 \Downarrow \lambda x. e \quad e_1 \Downarrow v' \quad e\{v'/x\} \Downarrow v}{e_0 e_1 \Downarrow v} \text{ (CBVAPP)}$$

We can read the (APP) rule as saying that to evaluate an application, we first evaluate  $e_0$ , then  $e_1$ , and then complete evaluation starting from the usual  $\beta$  contractum. This translates rather directly into a recursive interpreter for the lambda calculus, along the following lines:

```

type term = Var of string | Lambda of string * term | App of term * term

let interpret term =
  match term with
  | Var x | Lambda (x,e) -> term
  | App e0 e1 ->
    match interpret e0 with
    | Lambda (x, e) ->
      let v = interpret e1 in
      interpret(subst e v x)

```

The rules for call-by name are similar, but even simpler because we don't need to evaluate the argument to an application before  $\beta$  reduction:

$$\frac{}{v \Downarrow v} \text{ (VAL)} \qquad \frac{e_0 \Downarrow \lambda x. e \quad e\{e_1/x\} \Downarrow v}{e_0 e_1 \Downarrow v} \text{ (CBNAPP)}$$

Comparing the big-step and small-step semantics, we expect that they should agree with each other on all terminating computations.

The big-step semantics has some advantages for describing such computations. The big-step semantics is intrinsically more extensional because it describes the entire computation of a program. It also has fewer rules.

However, the big-step semantics has nothing to say about programs that diverge, whereas the small-step semantics can describe program executions that diverge. The small-step rules are also simpler than the big-step rules, which is good. Perhaps the key advantage of the small-step approach is that it is inherently more compositional. It can handle language features such as nondeterminism and concurrency better than big-step semantics can.

For example, suppose that we wanted to add a *nondeterministic choice* operator  $e_1 \square e_2$  to the lambda calculus. The idea is that it chooses to either evaluate  $e_1$  or  $e_2$ . In the small-step setting, this is easy to specify:

$$\frac{}{e_1 \square e_2 \longrightarrow e_1} \text{ (SCHOOSLEFT)} \qquad \frac{}{e_1 \square e_2 \longrightarrow e_2} \text{ (SCHOOSRIGHT)}$$

If we try to write similar rules for the big-step semantics, there is a subtle problem:

$$\frac{e_1 \Downarrow v}{e_1 \square e_2 \Downarrow v} \text{ (BCHOOSLEFT)} \qquad \frac{e_2 \Downarrow v}{e_1 \square e_2 \Downarrow v} \text{ (BCHOOSRIGHT)}$$

The big-step rules allow either  $e_1$  or  $e_2$  to be used to construct a valid execution. But there is no representation of the fact that choosing the “wrong” one might cause the computation to diverge. The big-step meaning of  $e \square \Omega$  is the same as for  $e$  alone. Thus, the big-step rules encode what is known as *angelic nondeterminism*, in which an oracle (or angel) tells the evaluation which way to go whenever it has a choice. This is a serious problem for modeling systems, such as concurrent systems, where nondeterministic choices matter.