1 Rewrite Rules

1.1 Recap—β-reduction

Recall that *β-reduction* is the following rule:

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

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,

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

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

1.2 Variable capture

CBN and CBV evaluation reduce β 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 λ, 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 $FV(()e)$, 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 α-equivalence, which we'll talk about shortly):

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

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 λz 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 λ-calculus, we can start with a λ-term and perform β-reductions on subterms in any order, using the full substitution rule avoid to avoid variable capture when the substituted term is open.

1.5 α-reduction

In $\lambda x.\, x\ z$ the name of the bound variable x doesn't really matter. This term is semantically the same as $\lambda y.\, y\ z$. A renaming like this is known as an *α-reduction*, *α-conversion*, or just *α-renaming*. In an α-reduction, the new bound variable must be chosen so as to avoid capture. If a term α-reduces to another term, then the two terms are said to be *α-equivalent*. This defines an equivalence relation on the set of terms, denoted $e_1 =_\alpha e_2$.

Recall the definition of free variables $FV(e)$ of a term e. In general we have

$$\lambda x.e \quad =_\alpha \quad \lambda y.e\{y/x\} \text{ if } y \notin FV(e).$$

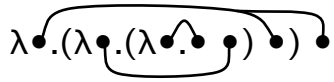
The proviso $y \notin FV(e)$ is to avoid the capture of a free occurrences of y in e as a result of the renaming.

When writing a λ-interpreter, the job of looking for α-renamings doesn't seem all that practical. However, we can use them to improve our earlier definition of equality:

$$\text{If } e_1 \Downarrow v_1,\ e_2 \Downarrow v_2, \text{ and } v_1 =_\alpha v_2, \text{ then } e_1 = e_2.$$

1.6 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 α-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 α-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.

1.7 η-reduction

Here is another notion of equality. Compare the terms e and $\lambda x.\, e\ x$. If these two terms are both applied to an argument e', then they will both reduce to $e\, e'$, provided x has no free occurrence in e. Formally,

$$(\lambda x.e_1 x)e_2 \xrightarrow{\beta} e_1 e_2 \text{ if } x \notin FV(e_1).$$

This says that e and $\lambda x.\, e\ x$ behave the same way as functions and should be considered equal. Another way of stating this is that e and $\lambda x.\, e\ x$ behave the same way in all contexts of the form $[[\,\cdot\,]]\, e'$.

This gives rise to the rule for *η-reduction*:

$$\lambda x.\, e\ x \quad \xrightarrow{\eta} \quad e \quad \text{if } x \notin FV(e).$$

The η rule may not be sound with respect to our earlier notion of equality, depending on our reduction strategy. For example, $\lambda x.\, e\ x$ is a value in CBV, but reductions might be possible in e and it might diverge.

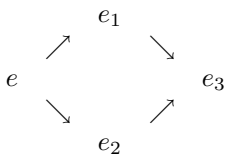
The reverse operation, called *η-expansion*, can be useful as well. In practice, η-expansion is used to delay divergence by trapping expressions inside λ abstraction terms.

2 Confluence

In the classical λ-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 λ-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 λ-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 λ-calculus is *confluent* (also known as the *Church–Rosser* property) under α- and β-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 α-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 α-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

$$(\lambda xy.y)\Omega \quad \xrightarrow{\beta} \quad (\lambda xy.y)\Omega \quad \xrightarrow{\beta} \quad \cdots$$

but a terminating CBN reduction sequence, namely

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

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.