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 Capture-avoiding Substitution

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. The rules are defined by induction on the height of the AST for e_1, following the convention that the height of a variable is 0. In addition, by induction on the height of e_1, we show that the height of $e_1\{e_2/x\}$ is at most the height of e_1 plus the height e_2.

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

The rules are defined inductively. That is, the result of a substitution in a compound term is defined in terms of substitutions on its subterms. Recall that the function $FV(e)$ gives the set of free variables of e. The very last of the six rules applies when $y \in FV(e)$, so we need to avoid variable capture. 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'? 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.

Each of the rules preserves the property that the height of the resulting tree is no greater than the sum of the heights of e_1 and e_2, assuming that the property holds for all smaller-height terms. In particular, this means that the result of the substitution $e'\{z/y\}$ in the final rule has the same height as e' itself, making the entire inductive definition well-founded.

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 to avoid variable capture when the substituted term is open.

1.4 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. Another way to achieve a canonical representation for alpha-equivalent terms is to represent variables using *de Bruijn indices*, which replace variables with natural numbers that indicate how many levels up in the AST the binding site is found. Thus, the term $\lambda x.\,x$ becomes $\lambda 0$ and $\lambda x.\,\lambda y.\,x$ becomes $\lambda\lambda 1$.

2 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:

$$\frac{}{v \Downarrow v}\ (\textsc{Val}) \qquad \frac{e_0 \Downarrow \lambda x.\,e \quad e_1 \Downarrow v' \quad e\{v'/x\} \Downarrow v}{e_0\;e_1 \Downarrow v}\ (\textsc{CBVApp})$$

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 β 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 β reduction:

$$\frac{}{v \Downarrow v}\ (\textsc{Val}) \qquad \frac{e_0 \Downarrow \lambda x.\, e \quad e\{e_1/x\} \Downarrow v}{e_0\ e_1 \Downarrow v}\ (\textsc{CBNApp})$$

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 exaple, suppose that we wanted to add a *nondeterministic choice* operator $e_1 \Box e_2$ 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:

$$\frac{}{e_1 \Box e_2 \longrightarrow e_1}\ (\textsc{SChooseLeft}) \qquad \frac{}{e_1 \Box e_2 \longrightarrow e_2}\ (\textsc{SChooseRight})$$

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

$$\frac{e_1 \Downarrow v}{e_1 \Box e_2 \Downarrow v}\ (\textsc{BChooseLeft}) \qquad \frac{e_2 \Downarrow v}{e_1 \Box e_2 \Downarrow v}\ (\textsc{BChooseRight})$$

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 $e \Box \Omega$ 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.