Last time we discussed the *β-reduction rule*, which is the main reduction rule used in the λ-calculus:

$$(\lambda x.\, e_1)\, e_2 \quad \xrightarrow{1} \quad e_1\{e_2/x\}.$$

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,

$$\lambda x.\, \underbrace{(\lambda y.\, y)\, x}_{\beta \text{ redex}} \quad \xrightarrow{1} \quad \lambda x.\, x$$

In the pure λ-calculus, β-reduction may be performed at any time to any subterm of the form $(\lambda x.\, e_1)\, e_2$.

1 Other Rewrite Rules

1.1 α-reduction

In $\lambda x.\, xz$, the name of the bound variable x does not really matter. This term is semantically the same as $\lambda y.\, yz$. Renamings like that are known as *α-reductions*. 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.

1.2 Stoy Diagrams and de Bruijn Indices

We can create a *Stoy diagram* (after Joseph E. Stoy) for a closed term in the following manner. Instead of writing a term with variable names, we replace each occurrence of a variable with a dot, then connect that dot to the binding operator that binds that variable. For example, $\lambda x.\, (\lambda y.\, (\lambda x.\, xy)\, x)\, x$ becomes the Stoy diagram shown in Fig. 1. Then two terms are α-equivalent iff they have the same Stoy diagram.

Figure 1: A Stoy diagram

A related approach is to represent variables using *de Bruijn indices* (after Nicolaas Govert de Bruijn). Here we replace each occurrence of a variable with a natural number indicating the binding operator that binds it. The variable is replaced by the number n if the binding operator that binds it has the n-th smallest scope (counting from 0) among all scopes containing that occurrence of the variable. The example above becomes $\lambda.\, (\lambda.\, (\lambda.\, 0\, 1)\, 1)\, 0$. As with Stoy diagrams, two terms are α-equivalent iff their de Bruijn terms are identical.

1.3 η-reduction

Here is another notion of equality. Compare the terms e and $\lambda x.\, ex$. 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,

$$(\lambda x.\, e_1 x)\, e_2 \xrightarrow{1} e_1\, e_2 \text{ if } x \notin FV(e_1).$$

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

This gives rise to a reduction rule called *η-reduction*:

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

The reverse operation, called *η-expansion*, is practical as well.

In practice, η-expansion is used to delay divergence by trapping expressions inside λ-terms. Such terms are sometimes called *thunks*.

1.4 Ω

Let us define an expression we will call Ω:

$$\Omega \quad \triangleq \quad (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

What happens when we try to evaluate it?

$$\Omega \quad = \quad (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \quad \xrightarrow{1} \quad (x\, x)\{(\lambda x.\, x\, x)/x\} \quad = \quad \Omega$$

We have just coded an infinite loop!

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 more difficult to define extensional 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, as illustrated in Fig. 2.

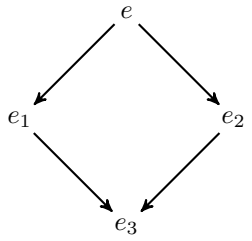

Figure 2: Confluence

It follows that normal forms are unique up to α-equivalence. 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 reduction sequence

$$(\lambda xy.\, y)\, \Omega \quad \xrightarrow{1} \quad (\lambda xy.\, y)\, \Omega \quad \xrightarrow{1} \quad \cdots$$

but also has a terminating reduction sequence, namely

$$(\lambda x.\, \lambda y.\, y)\, \Omega \quad \xrightarrow{1} \quad \lambda y.\, y.$$

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

3 Encoding Common Datatypes

Even though the pure λ-calculus consists only of λ-terms, we can represent and manipulate common data objects like integers, Boolean values, lists, and trees. All these things can be encoded as λ-terms.

3.1 Booleans

The Booleans are the easiest to encode, so let us start with them. We would like to define λ-terms to represent the Boolean constants true and false and the usual Boolean operators $\Rightarrow$ (if-then), $\wedge$ (and), $\vee$ (or), and $\neg$ (not) so that they behave in the expected way. There are many reasonable encodings. One good one is to define true and false by:

$$\mathsf{true} \triangleq \lambda xy.\, x \qquad\qquad \mathsf{false} \triangleq \lambda xy.\, y.$$

Now we would like to define a conditional test if. We would like if to take three arguments b, t, f, where b is a Boolean value (either true or false) and t, f are arbitrary λ-terms. The function should return t if $b = \mathsf{true}$ and f if $b = \mathsf{false}$.

$$\mathsf{if} \quad = \quad \lambda b\, t\, f.\, \begin{cases} t, & \text{if } b = \mathsf{true}, \\ f, & \text{if } b = \mathsf{false}. \end{cases}$$

Now the reason for defining true and false the way we did becomes clear. Since $\mathsf{true}\, t\, f \xrightarrow{1} t$ and $\mathsf{false}\, t\, f \xrightarrow{1} f$, all if has to do is apply its Boolean argument to the other two arguments:

$$\mathsf{if} \quad \triangleq \quad \lambda b\, t\, f.\, b\, t\, f$$

The other Boolean operators can be defined from if:

$$\mathsf{and} \triangleq \lambda b_1 b_2.\, \mathsf{if}\, b_1\, b_2\, \mathsf{false} \qquad \mathsf{or} \triangleq \lambda b_1 b_2.\, \mathsf{if}\, b_1\, \mathsf{true}\, b_2 \qquad \mathsf{not} \triangleq \lambda b_1.\, \mathsf{if}\, b_1\, \mathsf{false}\, \mathsf{true}$$

Whereas these operators work correctly when given Boolean values as we have defined them, all bets are off if they are applied to any other λ-term. There is no guarantee of any kind of reasonable behavior. Basically, with the untyped λ-calculus, it is *garbage in, garbage out.*

3.2 Natural Numbers

We will encode natural numbers $\mathbb{N}$ using *Church numerals.* This is the same encoding that Alonzo Church used, although there are other reasonable encodings. The Church numeral for the number $n \in \mathbb{N}$ is denoted

$\overline{n}$. It is the λ-term $\lambda fx.\, f^n\, x$, where f^n denotes the n-fold composition of f with itself:

$$\begin{array}{rcll}
\overline{0} & \triangleq & \lambda fx.\, f^0 x & = \lambda fx.\, x \\
\overline{1} & \triangleq & \lambda fx.\, f^1 x & = \lambda fx.\, fx \\
\overline{2} & \triangleq & \lambda fx.\, f^2 x & = \lambda fx.\, f(fx) \\
\overline{3} & \triangleq & \lambda fx.\, f^3 x & = \lambda fx.\, f(f(fx)) \\
 & \vdots & & \\
\overline{n} & \triangleq & \lambda fx.\, f^n x & = \lambda fx.\, \underbrace{f(f(\ldots(f}_{n}\, x)\ldots))
\end{array}$$

We can define the successor function succ as

$$\mathsf{succ} \triangleq \lambda nfx.\, f\,(n\, f\, x).$$

That is, succ on input $\overline{n}$ returns a function that takes a function f as input, applies $\overline{n}$ to it to get the n-fold composition of f with itself, then composes that with one more f to get the $(n+1)$-fold composition of f with itself. Then

$$\begin{array}{rcl}
\mathsf{succ}\,\overline{n} & = & (\lambda nfx.\, f(nfx))\,\overline{n} \\
 & \xrightarrow{1} & \lambda fx.\, f(\overline{n}fx) \\
 & \xrightarrow{1} & \lambda fx.\, f(f^n x) \\
 & = & \lambda fx.\, f^{n+1} x \\
 & = & \overline{n+1}.
\end{array}$$

We can perform basic arithmetic with Church numerals. For addition, we might define

$$\mathsf{add} \triangleq \lambda mnfx.\, mf(nfx).$$

On input $\overline{m}$ and $\overline{n}$, this function returns

$$\begin{array}{rcl}
(\lambda mnfx.\, mf(nfx))\,\overline{m}\,\overline{n} & \xrightarrow{1} & \lambda fx.\, \overline{m}f(\overline{n}fx) \\
 & \xrightarrow{1} & \lambda fx.\, f^m(f^n x) \\
 & = & \lambda fx.\, f^{m+n} x \\
 & = & \overline{m+n}.
\end{array}$$

Here we are composing f^m with f^n to get f^{m+n}.

Alternatively, recall that Church numerals act on a function to apply that function repeatedly, and addition can be viewed as repeated application of the successor function, so we could define

$$\mathsf{add} \triangleq \lambda mn.\, m\,\mathsf{succ}\, n.$$

Similarly, multiplication is just iterated addition, and exponentiation is iterated multiplication:

$$\mathsf{mul} \triangleq \lambda mn.\, m\,(\mathsf{add}\, n)\,\overline{0} \qquad \mathsf{exp} \triangleq \lambda mn.\, m\,(\mathsf{mul}\, n)\,\overline{1}.$$

3.3 Pairing and Projections

Logic and arithmetic are good places to start, but we still are lacking any useful data structures. For example, consider ordered pairs. It would be nice to have a pairing function pair with projections first and second that obeyed the following equational specifications:

$$\mathsf{first}\,(\mathsf{pair}\,e_1\,e_2) \;=\; e_1 \qquad \mathsf{second}\,(\mathsf{pair}\,e_1\,e_2) \;=\; e_2 \qquad \mathsf{pair}\,(\mathsf{first}\,p)\,(\mathsf{second}\,p) \;=\; p,$$

provided p is a pair. We can take a hint from if. Recall that if selects one of its two branch options depending on its Boolean argument. pair can do something similar, wrapping its two arguments for later extraction by some function f:

$$\mathsf{pair} \;\stackrel{\triangle}{=}\; \lambda abf.\, fab.$$

Thus $\mathsf{pair}\,e_1\,e_2 \;\to\; \lambda f.\, fe_1e_2$. To get e_1 back out, we can just apply this to true: $(\lambda f.\, fe_1e_2)\,\mathsf{true} \;\to\; \mathsf{true}\,e_1\,e_2 \to e_1$, and similarly applying it to false extracts e_2. Thus we can define

$$\mathsf{first} \;\stackrel{\triangle}{=}\; \lambda p.\, p\,\mathsf{true} \qquad \mathsf{second} \;\stackrel{\triangle}{=}\; \lambda p.\, p\,\mathsf{false}.$$

Again, if p is not a term of the form $\mathsf{pair}\,a\,b$, expect the unexpected.

3.4 Lists

One can define lists $[x_1; \ldots; x_n]$ and list operators corresponding to the OCaml `::`, `List.hd`, and `List.tl` in the λ-calculus. We leave these constructions as exercises.

3.5 Local Variables

One feature that seems to be missing is the ability to declare local variables. For example, in OCaml, we can introduce a new local variable with the let expression:

$$\mathsf{let}\; x = e_1 \;\mathsf{in}\; e_2$$

Intuitively, we expect this expression to evaluate e_1 to some value v and then to replace occurrences of x inside e_2 with v. In other words, it should evaluate to $e_2\{v/x\}$. But we can construct a λ-term that behaves the same way:

$$(\lambda x.\, e_2)\,e_1 \;\to\; (\lambda x.\, e_2)\,v \;\stackrel{1}{\to}\; e_2\{v/x\}.$$

We can thus view a let expression as syntactic sugar for an application of a λ-abstraction.