The Art of Recursion: Problem Set 4 (ver. 3)

Due Tuesday, 2 October 2012

Figure 1: Alonzo Church

Recall that terms of the *λ-calculus* are defined by the following algebra:

$\Sigma = \{\mathsf{Var}, \mathsf{Lam}, \mathsf{App}\}$

$T = \{\mathsf{Name}, \mathsf{Term}\}$

$Ty(\mathsf{Var}) = \mathsf{Name} \to \mathsf{Term}$

$Ty(\mathsf{Lam}) = \mathsf{Name} \times \mathsf{Term} \to \mathsf{Term}$

$Ty(\mathsf{App}) = \mathsf{Term} \times \mathsf{Term} \to \mathsf{Term}$

We assume there is some countably infinite set Name of names.

1. Write down a few elements of Term.

Usually, we abbreviate Terms by writing:

- x instead of $\mathsf{Var}(\mathsf{x})$;
- $\lambda \mathsf{x}.t$ instead of $\mathsf{Lam}(\mathsf{x}, t)$;
- $t_1\ t_2$ instead of $\mathsf{App}(t_1, t_2)$

We adopt the convention that App associates to the left, that is, for example,

$t_1\ t_2\ t_3\ t_4$

should be interpreted as

$(((t_1\ t_2)\ t_3)\ t_4)$,

that is, as

$\mathsf{App}(\mathsf{App}(\mathsf{App}(t_1, t_2), t_3), t_4)$.

Note also that the scope of a λ extends as far to the right as possible; for example, $\lambda \mathsf{z}.s\ t$ should be read as $\mathsf{Lam}(\mathsf{z}, (\mathsf{App}(s, t)))$, not as $\mathsf{App}(\mathsf{Lam}(\mathsf{z}, s), t)$.

2. Translate the terms you wrote down for problem 1 into this abbreviated syntax.
3. Translate

   $(\lambda \mathsf{x}.\mathsf{y})\ (\lambda \mathsf{z}.(\lambda \mathsf{q}.\mathsf{z}\ \mathsf{z}))\ ((\lambda \mathsf{f}.\mathsf{g})\ (\lambda \mathsf{y}.\mathsf{h}))$

   into a formal Term.

The idea is that $\lambda x.t$ represents a function which takes a single argument x as input and returns t as output, and App represents applying a function to an argument. For example, $\lambda x.x$ represents the "identity" function, which simply returns its argument unchanged. $\lambda x.(\lambda y.y)$ represents a function which ignores its argument and returns as output the identity function. $(\lambda x.x)\ y$ means giving the variable y as input to the identity function (and so we would expect that this is equivalent to just y). And so on.

This notion of equivalence turns out to be central. What does it mean for two λ-calculus terms to be equivalent? And what does it mean to "evaluate" a λ-calculus term to some other (equivalent) λ-calculus term?

α-equivalence

4. Can the same variable be both free and bound in the same λ-calculus term? If so, give an example; if not, explain why not.
5. Which of the following pairs of terms are α-equivalent?
   (a) $\lambda x.x$ and $\lambda y.y$
   (b) $\lambda x.y$ and $\lambda y.x$
   (c) $\lambda x.\lambda y.x$ and $\lambda z.\lambda y.z$
   (d) $(\lambda x.z)\ (\lambda y.x\ x)$ and $(\lambda q.z)\ (\lambda y.q\ q)$

Substitution and β-reduction

For *reducing* λ-calculus terms, there is only one thing that can happen: when a function is applied to an argument, substitute the argument for the function's parameter in the body of the function. Formally, we write $t_1 \longrightarrow t_2$ to indicate that t_1 *reduces to* t_2, which we define by the following three rules:

$$\frac{}{(\lambda x.b)a \longrightarrow [x \mapsto a]b}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

The notation $[\mathsf{x} \mapsto a]b$ means "substitute any occurrences of x in b with a". For example,

$$[\mathsf{x} \mapsto (\lambda \mathsf{z}.\mathsf{z})](\lambda \mathsf{y}.\mathsf{x}\ \mathsf{x}) = \lambda \mathsf{y}.(\lambda \mathsf{z}.\mathsf{z})\ (\lambda \mathsf{z}.\mathsf{z}).$$

The notation $\dfrac{foo}{bar}$ means "if *foo*, then *bar*". So the first rule says that an application of a λ to some argument always reduces, by substituting the argument for the parameter in the body of the lambda. (Note that a and b can stand for *any* λ-calculus term, not just variables.) The other two rules simply say that we can reduce subparts of an application (perhaps nested several levels deep). For example,

$$\mathsf{y}\ (((\lambda \mathsf{x}.\mathsf{x})\ \mathsf{z})\ \mathsf{y}) \longrightarrow \mathsf{y}\ (\mathsf{z}\ \mathsf{y}).$$

However, there is a wrinkle.

6. What would go wrong if we literally substituted $(\lambda \mathsf{z}.\mathsf{y}\ \mathsf{z})$ for x in $(\lambda \mathsf{y}.\mathsf{x})$?

There are many ways to get around this problem and define properly "capture-avoiding" substitution, but the simplest is this:

Before performing a substitution, first rename bound variables as necessary (resulting in an α-equivalent term) so that no free variable has the same name as any bound variable.

7. Carry out reduction of $(\lambda \mathsf{x}.(\lambda \mathsf{y}.\mathsf{x}))\ (\lambda \mathsf{z}.\mathsf{y}\ \mathsf{z})$ using this rule.

Note that we will use the notation $t_1 \longrightarrow^* t_2$ to indicate that t_1 reduces to t_2 in some finite number of steps, *i.e.*

$$t_1 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots \longrightarrow s_n \longrightarrow t_2.$$

Computing with the λ-calculus

At this point you might think that the λ-calculus seems much too simple to really do anything useful with. In fact, quite the opposite is true: it turns out that the λ-calculus is just as powerful as Turing machines! Here is just a taste.

First, we can represent the Boolean values True and False as *functions* which take two arguments and returns one of them. True is the

function which returns its first argument and ignores the second, and vice versa for False:

$\mathsf{True} = \lambda\mathsf{t}.\lambda\mathsf{f}.\mathsf{t}$

$\mathsf{False} = \lambda\mathsf{t}.\lambda\mathsf{f}.\mathsf{f}$

Normally we would write something like "if b then t else f"; now, b itself is the function which does this "if" for us! So instead we can just write $b\ t\ f$, which reduces to t if b is true, and f if b is false.

8. Define a λ-calculus term representing logical negation, that is, a term *not* such that *not* $\mathsf{True} \longrightarrow^* \mathsf{False}$, and vice versa.
9. Define λ-calculus terms representing logical conjunction and disjunction.

Next, we can represent the natural number n as a function which *iterates another function n times*. That is,

$\overline{0} = \lambda\mathsf{f}.\lambda\mathsf{z}.\mathsf{z}$

$\overline{1} = \lambda\mathsf{f}.\lambda\mathsf{z}.\mathsf{f}\ \mathsf{z}$

$\overline{2} = \lambda\mathsf{f}.\lambda\mathsf{z}.\mathsf{f}\ (\mathsf{f}\ \mathsf{z})$

$\overline{3} = \lambda\mathsf{f}.\lambda\mathsf{z}.\mathsf{f}\ (\mathsf{f}\ (\mathsf{f}\ \mathsf{z}))$

$\dots$

(If n is a natural number, we will denote the corresponding λ-calculus term by $\overline{n}$.) In general,

$$\overline{n}\ \mathsf{f}\ \mathsf{z} = \underbrace{\mathsf{f}\ (\mathsf{f}\ \dots (\mathsf{f}}_{n}\ \mathsf{z}) \dots)$$

Using this encoding, it turns out we can do everything with natural numbers we might wish.

10. Define a λ-calculus term *succ* with the property that

    $succ\ \overline{n} \longrightarrow^* \overline{n+1}$.

11. Define a λ-calculus term *add* with the property that

    $(add\ \overline{m})\ \overline{n} \longrightarrow^* \overline{m+n}$.

12. Define a λ-calculus term *mul* with the property that

    $(mul\ \overline{m})\ \overline{n} \longrightarrow^* \overline{mn}$.

We can do more... [1]

[1] In fact, though we will not show it, we can encode *any* algebra. You might enjoy thinking about how this is done in general, using the examples in the exercises for inspiration.

13. ★ Define λ-calculus terms *pair*, *fst*, and *snd* with the property that for all λ-calculus terms t_1 and t_2,

    $fst\ (pair\ t_1\ t_2) \longrightarrow^* t_1$

    and

    $snd\ (pair\ t_1\ t_2) \longrightarrow^* t_2$.

14. ★ Define a λ-calculus term *pred* such that

    $pred\ \overline{n} \longrightarrow^* \overline{n-1}$

    whenever $n > 0$. (It doesn't matter what $pred\ \overline{0}$ reduces to; you will probably find it convenient to have it reduce to $\overline{0}$.)

Recursion in the λ-calculus

At first, it might seem that there is no way to define a recursive function in the λ-calculus, since there is no way for a function to directly refer to itself—a λ term can only refer to its argument. However, despite this apparent limitation, the λ-calculus turns out to be expressive enough to encode any recursion!

The first hint that everything is not quite as it seems is the ability to define terms which continue reducing infinitely.

15. ★ Define a λ-calculus term (often called Ω) with the property that it reduces to itself, that is, $\Omega \longrightarrow \Omega$.

However, by itself Ω is just a curiosity; it can't be used to do any useful recursion.

16. ★ Define a λ-calculus term Y with the property that for any other λ-calculus term f,

$$Y\ f \longrightarrow^* f\ (Y\ f).$$