Term Rewrite Systems

Let's start with something we're gonna see over and over again. Let's assume we have a set of variables names $\mathcal{V} = \{a, b, c, ...\}$ Then we will define $\mathcal{L}$ recursively.

1. $\mathcal{V} \subset \mathcal{L}$
2. If $e_1, e_2 \in \mathcal{L}$ then $e_1 e_2 \in \mathcal{L}$
3. If $e \in \mathcal{L}$ and $x \in \mathcal{V}$ then $\lambda x, e \in \mathcal{L}$

That is, we can name variables, apply functions, and define functions. These are the terms of our rewrite system, but before we can talk about the rules, we need to do some boring stuff.

In particular, we need to define *capture free substitution*. That is, we define $e[x := a]$ for e and a as lambda terms, and x as a variable.

1. $x[x := a] = a$
2. $y[x := a] = y$
3. $e_1 e_2[x := a] = (e_1[x := a])(e_2[x := a])$
4. $(\lambda x, e)[x := a] = \lambda x, e$
5. $(\lambda y, e)[x := a] = \lambda y, e[x := a]$

That is, if a name appears as a binder, ignore it. This is a quirk of how we have handled associating "holes" and "binders", not anything deep about lambda calculus. Then finally we can state our reduction rule

$$\beta : \ (\lambda x, e)a = e[x := a]$$

We will consider many kinds of rules over this set of terms, so it's useful to give the rules names. This one is known as beta. Generally you don't want to think to hard about the names of variables. In later entries of this series we will have much more to say about it.

1 How smart is the Untyped Lambda Calclus

Let's start with some basics. Let's define some useful terms, (note these aren't canonical names, just convienent for now).

$\mathrm{T} = \lambda x, \lambda y, x$

$\mathrm{F} = \lambda x, \lambda y, y$

$\mathrm{I} = \lambda x.x$

$\mathrm{R} = (\lambda x.xx)(\lambda x, xx)$

$\mathrm{Ap0} = \lambda f, \lambda x, x$

$\mathrm{Ap1} = \lambda f, \lambda x, f\, x$

$\mathrm{Ap2} = \lambda f, \lambda x, f\, (f\, x)$

We can make some observations off the bat. Firstly, all of these except R are lambda, therefore normal forms. Let's prove that R does not have a normal form. The argument is real easy: there is only one way to beta reduce R, and it produces R.

$$(\lambda x.xx)(\lambda x, xx) \to (\lambda x, xx)(\lambda x, xx)$$

This means that R is not a normal form. And every redux of R is not a normal form. So R will never normalize.

1.1 Nontermination

Ok, so we've got some non-terminating stuff. Is that really so bad? Well, one consequence is the halting problem.

Theorem 1 (Lambda Halting)**.** *There is no lambda term H such that exactly one of the following happens*

1. *(H p x) normalizes to T and (p x) normalizes*
2. *(H p x) normalizes to F and (p x) does not normalize*

Proof. Suppose not, and let H be such a lambda term. Consider

$$J = \lambda x, (Hxx)\, R\, F$$

Then what does (H J J) do? Well by assumption there are only two cases

1. H J J = T and (J J) normalizes: Then observe that

   $$JJ = (\lambda x, (Hxx)\, R\, F)J \to (H\, J\, J)\, R\, F = T\, R\, F \to^* R$$

   But (J J) can't normalize since R doesn't normalize, and there are no other paths from (J J) in the rewrite system

2. H J J = F and (J J) does not normalize: This time

   $$JJ = (\lambda x, (Hxx)\, R\, F)J \to (H\, J\, J)\, R\, F = F\, R\, F \to^* F$$

   But this is a proof that (J J) normalizes to F, contradicting the case.

Both cases lead to contradictions, so H doesn't exist. □

There are several things to notice. Firstly Lambda Calculus is not simple enough to have its own halting checker. What this proof does depend on is

1. The existence of $T = \lambda x, \lambda y, x$ as a return of H
2. The existence of $F = \lambda x, \lambda y, y$ as a return of H
3. T and F are distinct
4. The existence of a non-normalizing term (we use R)
5. The existence of a normalizing term (we use F again)
6. the existence of this J term

   $$J = \lambda H, \lambda x, (Hxx)\, R\, F$$

The existence and distinctness of T and F are really just a non-triviality condition. If there are no non-normalizing terms, then clearly $H = \lambda x, \lambda y, T$ is a halting checker. So really the only interesting requirement is the combinator. As long as such a combinator is expressible, your language will always have halting problems. This is not always a bad thing. If you want turing completeness, you don't really have a choice.

Summary: Lambda calculus cannot, internally, distinguish normalizing from non-normalizing terms.

1.2 Confluence

Thankfully lambda calculus is confluent. Which largely justifies our intuition that a term is a "program" and its reduction is it's evaluation. The proof of confluence is a bit tricky. We won't go through all the details, but the approach isn't too bad. The main problem is that when evaluating a term like

$$(\lambda x, x\, x)((\lambda x, x)\, a) \to (\lambda x, xx)\, a \to a\, a$$

Or via

$$(\lambda x, x\, x)((\lambda x, x)\, a) \to ((\lambda x, x)\, a)((\lambda x, x)\, a) \to ((\lambda x, x)\, a)\, a \to a\, a$$

Even though things normalize, the choice of where to apply a rule changes the length of the path to its normal form. The proof is in three steps.

1. Define a new rule, a "parellel" beta which applies every possible beta.
2. prove that this systems has the same paths in the term rewriting system
3. prove that parellel-beta resolve in one-step, that is, if $a \to b$ and $a \to c$, then there exists a d with $b \to d$ and $c \to d$. This is a proof by induction on lambda terms.

1.3 Equality

Equality of terms can be quite subtle. We discussed briefly how to turn a term rewrite system into an equational theory. Consider the rewrite system graph: forget the direction of the edges. Then two terms are equal when they're in the same graph component. E.G. there is a path between them.

Continuing the idea of *can we make lambda terms think for us*; Lambda calculus doesn't know how to distinguish normalizing and non-normalizing things. Fine. What about equal things? Whatever $=$ means, it should have the property that

$$x = y \to \forall e, ex = ey$$

Well, let's agree, as usual, that T and F are not equal. Then how might we show that Ap0 and Ap1 are not equal? One way to do this is find a term X which behaves differently on them. See

$$X = \lambda x, x\, (\lambda y, F)\, T \qquad X\, Ap0 = T \qquad X\, Ap1 = (\lambda y, F)\, T = F$$

So X witnesses the difference between $Ap0$ and $Ap1$. In our greediest fantasy, we have a term E with

1. if x and y are normal forms, then $E\, x\, y = T \leftrightarrow x$ and y are (syntactically) distinct
2. if x and y are normal forms, then $E\, x\, y = F \leftrightarrow x = y$ (syntactically)

In other words, does lambda calculus compute its own equality?

The answer is no for several reasons. One reason is rather frustrating. Observe that

$$\lambda x, f\, x \quad \neq_\beta \quad f$$

since they are both normal forms, and not syntactically equal. But for any term e,

$$(\lambda x, f\, x)e \quad =_\beta \quad f\, e$$

So what would $E\,(\lambda x, f\,x)f$ do? Turns out no such E will exist. We don't go into that proof, it's not fun or insightful. Basically you have to apply the terms, and you can't tell if you applied an f or a $\lambda x, f\,x$. But we know we can *sometimes* find inequality witnesses. So how do we fix our goal? Turns out the right approach is to add a new rewrite rule,

$$\eta:\ \lambda x, fx \to f$$

This will give us the rather convenient "extensional equality"

$$\forall e, f\,e =_{\beta\eta} g\,e \to f =_{\beta\eta} g$$

So again, we can ask is there a lambda term E where

1. if x and y are $\beta\eta-$normal forms, then $E\,x\,y = T \leftrightarrow x$ and y are (syntactically) distinct

2. if x and y are $\beta\eta-$normal forms, then $E\,x\,y = F \leftrightarrow x = y$ (syntactically)

Surprisingly, the answer is yes! This is *Bohm's Theorem*, and it provides an algorithm for encoding $\beta\eta-$normal forms into a nice tree. We don't go over the proof here since it's a bit technical.

Summary: Lambda calculus can, internally, distinguish $\beta\eta-$normal forms from each other. Of course deciding if you are a $\beta\eta-$normal form cannot be done in general!