Lecture Notes on Recursion, Binding, Substitution, and Computation

15-814: Types and Programming Languages
Ryan Kavanagh

Lecture 2
September 6, 2018

1 Introduction

Last time we saw that the set Λ of lambda terms was generated by the grammar:

$$e ::= x \mid \lambda x.e \mid e_1 e_2.$$

We worked out some programming examples involving Booleans and natural numbers. We reasoned informally $\alpha\beta\eta$-equivalence and saw that we could go wrong if we were not careful about binding and substitution.

Today we will make the notions of equivalence and substitution precise. We will also see how to capture recursion.

1.1 Warm up

To make sure we remember how to use the untyped λ-calculus, let us do a few warm-up exercises. You can find the solutions on the next page.

Exercise 1 *Define the constant function* $\mathbf{K}$ *(also known as the* $\mathbf{K}$ *combinator) that satisfies* $\mathbf{K}xy = x$ *for all* x *and* y*.*

Exercise 2 *Define a test to see if a Church numeral is zero:*

$$\mathsf{isZero}\ \overline{0} = \mathsf{true} = \lambda x.\lambda y.x$$
$$\mathsf{isZero}\ \overline{n+1} = \mathsf{false} = \lambda x.\lambda y.y$$

It is interesting to consider what happens when we apply a λ-term to itself. Self-application is captured by the term $\omega = \lambda x.xx$. This may look odd at first sight, but it is a perfectly acceptable term. For example, $\omega\mathbf{K} = \lambda y.\mathbf{K}$ and $\omega\mathbf{I} = \mathbf{I}$. More interesting is $\Omega = \omega\omega$:

$$\Omega = (\lambda x.xx)(\lambda x.xx) = [(\lambda x.xx)/x](xx) = (\lambda x.xx)(\lambda x.xx).$$

The term Ω behaves exactly like an infinite loop!

Solutions Take $\mathbf{K} = \lambda x.\lambda y.x$ and $\mathsf{isZero} = \lambda n.n\ \mathsf{true}\ (\mathbf{K}\ \mathsf{false})$. The intuition for isZero is that if $\overline{n}$ is zero, then we should return true, and otherwise, $\overline{n}$'s "successor" parameter should be a function that constantly returns false.

2 Recursion

We would like to implement the factorial function

$$\mathsf{fact}\ \overline{n} = \mathsf{if}\ (\mathsf{isZero}\ \overline{n})(\overline{1})(\mathsf{mult}\ \overline{n}\ (\mathsf{fact}\ (\mathsf{pred}\ \overline{n})))$$

assuming we already have a predecessor function pred, which you will implement on Homework 0. We might start off with

$$\mathsf{fact} = \lambda n.\mathsf{if}(\mathsf{isZero}\ n)(\overline{1})(\mathsf{mult}\ n\ (\mathsf{fact}(\mathsf{pred}\ n))),$$

but we get stuck because we have an instance of fact on both sides. Let us consider what would happen if we factored out a fact on the right:

$$\mathsf{fact} = (\lambda f.\lambda n.\mathsf{if}\ (\mathsf{isZero}\ n)(\overline{1})(\mathsf{mult}\ n\ (f(\mathsf{pred}\ n))))\mathsf{fact}.$$

Letting

$$\Phi = \lambda f.\lambda n.\mathsf{if}\ (\mathsf{isZero}\ n)(\overline{1})(\mathsf{mult}\ n\ (f(\mathsf{pred}\ n))),$$

we see that fact can be expressed as a fixed point of Φ, that is, $\Phi(\mathsf{fact}) = \mathsf{fact}$. Can we find such a fixed point?

Theorem 1 (Fixed point theorem) *For all $F \in \Lambda$ there exists an $X \in \Lambda$ such that $FX = X$.*

Proof: Earlier we encountered the divergent term $\Omega = (\lambda x.xx)(\lambda x.xx)$, where applying the β rule gave Ω again:

$$(\lambda x.xx)(\lambda x.xx) = [(\lambda x.xx)/x](xx) = (\lambda x.xx)(\lambda x.xx).$$

This infinite unfolding behaviour is similar to what we want in a fixed point: if X is a fixed point of F, then $X = FX = F(FX) = \cdots$. Suppose we inserted an F at the beginning of each of the function bodies:

$$(\lambda x.F(xx))(\lambda x.F(xx)) = [\lambda x.F(xx)x](F(xx)) = F((\lambda x.F(xx))(\lambda x.F(xx))).$$

Take $X = (\lambda x.F(xx))(\lambda x.F(xx))$ and we are done. □

We can abstract over the F in the above proof to get the $\mathbf{Y}$ *combinator*[1] that constructs the fixed point of any term to which it is applied:

Corollary 2 *Let* $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. *Then* $\mathbf{Y}F = F(\mathbf{Y}F)$ *for all* $F \in \Lambda$.

We can now define our factorial function: $\mathsf{fact} = \mathbf{Y}\Phi$. Unfolding the definition and using Corollary 2, we see that this is actually what we wanted:

$$\begin{aligned}
\mathsf{fact}\ \overline{n} &= \mathbf{Y}\Phi\overline{n} \\
&= \Phi(\mathbf{Y}\Phi)\overline{n} \\
&= \mathsf{if}\ (\mathsf{isZero}\ \overline{n})(\overline{1})(\mathsf{mult}\ \overline{n}\ (\mathbf{Y}\Phi(\mathsf{pred}\ \overline{n}))) \\
&= \mathsf{if}\ (\mathsf{isZero}\ \overline{n})(\overline{1})(\mathsf{mult}\ \overline{n}\ (\mathsf{fact}(\mathsf{pred}\ \overline{n}))).
\end{aligned}$$

3 Binding and substitution

We need to be careful when we substitute to ensure that we do not accidentally bind (or *capture*) free variables. We say that an occurrence of the variable x is *bound* if it is in the scope of an abstractor λx, otherwise it is *free*. The set of free variables $\mathsf{fv}(e)$ in a term e is recursively defined on the structure of the term:

$$\begin{aligned}
\mathsf{fv}(x) &= \{x\} \\
\mathsf{fv}(\lambda x.e) &= \mathsf{fv}(e) \setminus \{x\} \\
\mathsf{fv}(e_1 e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2).
\end{aligned}$$

We say that a term is *closed* or a *combinator* if it has no free variables.

For the rest of this lecture, we will use the symbol $\equiv$ to mean that two terms are syntactically equal.

[1] The startup incubator was named after this combinator.

Given a term $\lambda x.e$, a *change of bound variable* is the result of $\lambda y.[y/x]e$ when y does not appear in e. Because y does not appear in e, we do not need to worry about capture. In this case, we say that $\lambda x.e$ and $\lambda y.[y/x]e$ are α-congruent: $\lambda x.e =_\alpha \lambda y.[y/x]e$. More generally, we say that two terms e_1 and e_2 are α-congruent, $e_1 =_\alpha e_2$, if e_1 can be obtained from e_2 through a sequence changes of bound variable. For example,

$$\lambda x.x(\lambda y.yx)z =_\alpha \lambda w.w(\lambda y.yw)z \neq_\alpha \lambda z.z(\lambda y.yz)z.$$

Changing bound variables is sometimes called α-*varying*. We identify α-congruent terms, that is, we treat them as though they were syntactically equal. Thanks to this identification, we can adopt the *variable convention*: we always assume the bound variables are chosen to be different from all of the free variables. With the variable convention, we can safely substitute in the naïve manner. (We implicitly assumed the variable convention in the proof of theorem 1. Where?[2])

We can now make the definition of substitution explicit:

$$\begin{aligned}
[e/x]x &\equiv e \\
[e/x]y &\equiv y && (\text{if } x \not\equiv y) \\
[e_1/x](\lambda y.e_2) &\equiv \lambda y.[e_1/x]e_2 && (\text{if } x \not\equiv y \text{ and } y \notin \mathsf{fv}(e_1))^{3} \\
[e_1/x](e_2e_3) &\equiv ([e_1/x]e_2)([e_1/x]e_3)
\end{aligned}$$

It is a good exercise to think carefully about this definition and how it interacts with the variable convention.

4 Reduction and computation

So far we have treated the λ-calculus as an equational theory. This is not a satisfactory notion of computation, because we have no notion of making progress or of termination (of knowing when we have reached a "value").

To capture the idea of making some form of directed "progress", we use *reductions*. β-*reduction* is the least relation $\to_\beta$ on Λ satisfying for all $e_1, e_2 \in \Lambda$:

- $(\lambda x.e_1)e_2 \to_\beta [e_2/x]e_1$,
- if $e_1 \to_\beta e_1'$, then $e_1e_2 \to_\beta e_1'e_2$, $e_2e_1 \to_\beta e_2e_1'$, and $\lambda x.e_1 \to_\beta \lambda x.e_1'$.

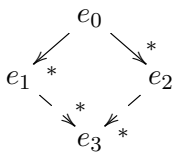
[2] We implicitly assumed $x \notin \mathsf{fv}(F)$.
[3] These conditions are redundant by the variable convention.

Let $\rightarrow_\beta^*$ be the reflexive, transitive closure of $\rightarrow_\beta$, i.e., the least relation on Λ inductively defined by:

- $e \rightarrow_\beta^* e$ for all e,
- if $e_1 \rightarrow_\beta e_1'$ and $e_1' \rightarrow_\beta^* e_2$, then $e_1 \rightarrow_\beta^* e_2$.

We say that M is in β-normal from if M cannot be β-reduced.

β-reduction satisfies the *confluence* property that we foreshadowed last time, from which we can deduce that every λ-term has a unique β-normal form. A relation $\rightarrow$ is confluent if whenever $e_0 \rightarrow^* e_1$ and $e_0 \rightarrow^* e_2$, there exists an e_3 such that $e_1 \rightarrow^* e_3$ and $e_2 \rightarrow^* e_3$. Pictorially,



Theorem 3 (Church-Rosser) *β-reduction is confluent.*

However, β-reduction is not what we want as a notion of computation. The reason is that β-reduction behaves a bit like equality: it can be applied anywhere in a term. As a result, operationally it is highly non-deterministic. Depending on how you apply β-reduction, you could either reach a β-normal form or fail to ever terminate. Consider for example the λ-term $(\lambda x.y)\Omega$. Applying β-reduction on the outermost β-redex gives $(\lambda x.y)\Omega \rightarrow_\beta y$. In contrast, if we repeatedly apply β-reduction to Ω, we never reach a β-normal form: $(\lambda x.y)\Omega \rightarrow_\beta (\lambda x.y)\Omega \rightarrow_\beta (\lambda x.y)\Omega \rightarrow_\beta \cdots$.

To make reduction deterministic, we use *reduction strategies*. The simplest of these is call-by-name (CBN) reduction, $\rightarrow_{CBN}$, defined to be the least relation on Λ satisfying for all $e_1, e_2 \in \Lambda$:

- $(\lambda x.e_1)e_2 \rightarrow_{CBN} [e_2/x]e_1$, and
- if $e_1 \rightarrow_{CBN} e_1'$, then $e_1 e_2 \rightarrow_{CBN} e_1' e_2$.

The intuition is that we eagerly reduce as far to the left as possible. Observe that this reduction strategy is deterministic: if $e_1 \rightarrow_{CBN} e_2$ and $e_1 \rightarrow_{CBN} e_2'$, then $e_2 \equiv e_2'$.

Theorem 4 *If $e_1 \rightarrow_{CBN} e_2$, then $e_1 \rightarrow_\beta e_2$. The converse is false.*

Proof: The first part is obvious. The term $(\lambda x.y)\Omega$ is a counter-example to the converse: $(\lambda x.y)\Omega \rightarrow_\beta (\lambda x.y)\Omega$, but $(\lambda x.y)\Omega \not\rightarrow_{CBN} (\lambda x.y)\Omega$. □

References

[Bar12] Henk P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 40 of *Studies in Logic*. College Publications, 2012.