We've seen the syntax and operational semantics for the lambda calculus (in fact, two different operational semantics). However, it may not seem at this point like a language in which you could write interesting programs. In this lecture, we'll see that many features you would expect to find in a typical programming language can be encoded into lambda calculus terms.

1 Local variables

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

let $x = e_1$ in e_2

Intuitively, we expect this expression to evaluate e_1 to some value v_1 and then to replace occurrences of x inside e_2 with v_1. In other words, it should evaluate to $e_2\{x/v_1\}$. But we can construct another term that evaluates in the same way:

$$\begin{array}{rl} & (\lambda x.\, e_2)\ e_1 \\ \longrightarrow^* & (\lambda x.\, e_2)\ v_1 \\ \longrightarrow & e_2\{x/v_1\} \end{array}$$

Therefore, we can view a let expression as simply syntactic sugar for an application of a lambda abstraction. Note that we use the notation $\longrightarrow^*$ to represent the transition through zero or more steps of the evaluation relation $\longrightarrow$.

2 Booleans

Perhaps the simplest interesting kind of value is a Boolean. We would like to define the Boolean constants *TRUE* and *FALSE* and the Boolean operators *IF*, *AND*, *OR*, *NOT*, etc. so that they behave in the expected way. There are many reasonable encodings. One good one is to define *TRUE* and *FALSE* as functions that return the first and second of their two arguments, respectively.

$$\begin{array}{rcl} \mathit{TRUE} & \triangleq & \lambda xy.\, x \\ \mathit{FALSE} & \triangleq & \lambda xy.\, y. \end{array}$$

Now we would like to define the conditional test *IF*. We would like *IF* to take three arguments b, t, f, where b is a Boolean value and t, f are arbitrary λ-terms. The function should return t if $b = \mathit{TRUE}$ and f if $b = \mathit{FALSE}$.

$$\mathit{IF} \quad = \quad \lambda b\, t f.\, \begin{cases} t, & \text{if } b = \mathit{TRUE}, \\ f, & \text{if } b = \mathit{FALSE}. \end{cases}$$

Now the reason for defining *TRUE* and *FALSE* the way we did becomes clear. Since $\mathit{TRUE}\ t\ f \to t$ and $\mathit{FALSE}\ t\ f \to f$, all *IF* has to do is apply its Boolean argument to the other two arguments:

$$\mathit{IF} \quad \triangleq \quad \lambda b\, t f.\, b\ t\ f$$

The other Boolean operators can be defined from *IF*:

$$\begin{aligned} AND &\triangleq \lambda b_1\, b_2.\, IF\, b_1\, b_2\, FALSE \\ OR &\triangleq \lambda b_1\, b_2.\, IF\, b_1\, TRUE\, b_2 \\ NOT &\triangleq \lambda b_1.\, IF\, b_1\, FALSE\, TRUE \end{aligned}$$

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 term. There is no guarantee of any kind of reasonable behavior. Evaluation will proceed and reductions will be applied, but there is no way to usefully interpret the result. With the untyped λ-calculus, it is *garbage in, garbage out*. Later on we'll talk about how to model run-time type checking that would catch erroneous uses.

3 Integers

We 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 term $\lambda f.\, f^n$, where f^n denotes the n-fold composition of f with itself:

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

We can define the successor function s as

$$s \triangleq \lambda n.\, \lambda f x.\, f(n\ f\ x).$$

That is, s 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,

$$\begin{aligned} s\, \overline{n} &= (\lambda n f x.\, f(n\ f\ x))\, \overline{n} \\ &\to \lambda f x.\, f(\overline{n}\ f\ x) \\ &\to \lambda f x.\, f(f^n\ x) \\ &= \lambda f x.\, f^{n+1}\, x \\ &= \overline{n+1}. \end{aligned}$$

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

$$ADD \triangleq \lambda m\, n\, f\, x.\, m\ f\ (n\ f\ x).$$

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

$$\begin{aligned} (\lambda m\, n\, f\, x.\, m\ f\ (n\ f\ x))\, \overline{m}\, \overline{n} &\to \lambda f\ x.\, \overline{m}\ f\ (\overline{n}\ f\ x) \\ &\to \lambda f\ x.\, f^m\ (f^n\ x) \\ &= \lambda f\ x.\, f^{m+n}\, x \\ &= \overline{m+n}. \end{aligned}$$

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

$$ADD \triangleq \lambda m\, n.\, m\ s\ n.$$

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

$$MUL \triangleq \lambda m\, n.\, m\ (ADD\ n)\ \overline{0} \qquad EXP \triangleq \lambda m\, n.\, m\ (MUL\ n)\ \overline{1}.$$

Defining subtraction, division, and other arithmetic operations is possible but more tricky.

Church numerals only encode natural numbers, but they can be used to build more complex kinds of numbers such as integers and floating point numbers.

4 Pairing and Projection

Logic and arithmetic are good places to start, but we still have no useful data structures. For example, consider ordered pairs. We expect to have a pairing constructor *PAIR* with projection operations *FIRST* and *SECOND* that obey the following equational specification,

$$\begin{aligned} FIRST\ (PAIR\ e_1\ e_2) &= e_1 \\ SECOND\ (PAIR\ e_1\ e_2) &= e_2 \\ PAIR\ (FIRST\ p)\ (SECOND\ p) &= p, \end{aligned}$$

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:

$$PAIR \triangleq \lambda ab.\, \lambda f.\, f\ a\ b.$$

Thus $PAIR\ e_1\ e_2 \to \lambda f.\, f\ e_1\ e_2$. To get e_1 back out, we can just apply this to *TRUE*: $(\lambda f.\, f\ e_1\ e_2)\ TRUE \to TRUE\ e_1\ e_2 \to e_1$, and similarly applying it to *FALSE* extracts e_2. Thus we can define

$$FIRST \triangleq \lambda p.\, p\ TRUE \qquad SECOND \triangleq \lambda p.\, p\ FALSE.$$

Again, if p isn't a term constructed using *PAIR*, expect the unexpected.

5 Recursion and the Y Combinator

With an encoding for *IF*, we have some control over the flow of a program. We can also write simple `for` loops using the Church numerals $\overline{n}$, by applying the Church numeral to the loop body expressed as a function. However, we do not yet have the ability to write an unbounded `while` loop or a recursive function.

In ML, we can write the factorial function recursively as

$$\mathsf{rec\ fact(n) = if\ n \leq 1\ then\ 1\ else\ n*fact(n\text{-}1)}$$

But how can we write this in the λ-calculus, where all the functions are anonymous? We must somehow construct a term *FACT* that satisfies following equation, assuming appropriate definitions for *SUB* and a less-than-or-equal comparison *LEQ*:

$$FACT = \lambda n.\, IF\ (LEQ\ n\ 1)\ 1\ (MUL\ n\ (FACT\ (SUB\ n\ 1))) \tag{1}$$

We can't just define *FACT* with this equation, because it appears on the right-hand side. But it turns out we can define a term that acts in the way we want.

5.1 Recursion through self-application

Suppose we break up the recursion into two steps. Since *FACT* can't refer to itself directly, we need some way to pass *FACT* into itself so it can be used for the recursion. We start by defining a function that adds an extra argument f for passing in something like *FACT*:

$$F \triangleq \lambda f.\, \lambda n.\, IF\ (LEQ\ n\ 1)\ (1)\ (MUL\ n\ (f\ (SUB\ n\ 1)))$$

Now, if we just had *FACT* available, we could pass it to this function and get the *FACT* we want. Of course, that's begging the question. But we do have something rather similar to *FACT*: this function itself. That won't quite work either, because this function expects to receive the extra argument f. So we just change the function definition so it calls f with an extra argument: f. Call this new function $FACT'$:

$$FACT' \triangleq \lambda f.\, \lambda n.\, IF\ (LEQ\ n\ 1)\ (1)\ (MUL\ n\ (\Big(f\ f\Big)(SUB\ n\ 1)))$$

$FACT'$ has the property that if a function just like it is passed to it, it will do what *FACT* should. Since $(FACT'\ FACT')$ should behave as we want *FACT* to, we define *FACT* as exactly this self-application:

$$FACT \triangleq FACT'\ FACT'$$

We can now see the recursion working:

$$\begin{aligned}
FACT(4) &= (FACT'\ FACT')\ 4 \\
&= (\lambda n.\, IF\ (LEQ\ n\ 1)\ 1\ (MUL\ n\ ((FACT'\ FACT')\ (SUB\ n\ 1))))\ 4 \\
&= IF\ (LEQ\ 4\ 1)\ 1\ (MUL\ 4\ (FACT\ (SUB\ 4\ 1))) \\
&= (MUL\ 4\ (FACT\ (SUB\ 4\ 1))) \\
&= (MUL\ 4\ (FACT\ 3)) \\
&\ldots
\end{aligned}$$

5.2 Fixed points and the Y combinator

If the function F is passed *FACT* as an argument, it returns a function that also acts like *FACT*.

Then we can write equation (1) simply as

$$FACT = F(FACT)$$

This shows that what we are looking for is a *fixed point* (sometimes shortened to *fixpoint*) of the function F: an argument *FACT* that when passed to F, results in the same value *FACT*. Any solution of (1) will do; different solutions may disagree on non-integers, but one can show inductively that any solution of (1) will yield $\overline{n!}$ on input $\overline{n}$. Thus it is only a question of existence.

So the question is, can we in general find a fixed point of an arbitrary function F? And can we do it within the confines of the lambda calculus? It is not immediately obvious that it is possible. The cardinality of the possible functions from lambda calculus expressions to other lambda calculus expressions is larger than that of the lambda calculus expressions themselves, so the (computable) functions expressible in the lambda calculus are a vanishingly small subset of the possible functions. And we know that some functions, like the halting problem, are in fact not computable and therefore cannot be written as lambda calculus expressions.

It turns out that we can use the same self-application trick to construct a term that finds fixed points for us. The key observation is that we can turn F into $FACT'$ by applying it to a self-application term $(x\ x)$, and abstracting over the same x:

$$\lambda x.\, F\ (x\ x) = \lambda x.\, \lambda n.\, IF\ (LEQ\ n\ 1)\ 1\ (MUL\ n\ ((x\ x)\ (SUB\ n\ 1))) \tag{2}$$

Using this, we can define a term that when applied to F, gives us exactly $(FACT'\ FACT')$. This term is called the Y combinator or fixed-point combinator.

$$\begin{aligned} Y &\triangleq \lambda f.\,(\lambda x.\,f\ (x\ x))\ (\lambda x.\,f\ (x\ x)) \\ YF &= (\lambda x.\,F\ (x\ x))\ (\lambda x.\,F\ (x\ x)) \\ &= FACT'\ FACT' = FACT \end{aligned}$$

The Y combinator is very helpful for producing recursive functions, because it finds fixed points. We also can see this by considering the term we get when we apply Y to an arbitrary function F:

$$(\lambda x.\,F\ (x\ x))\ (\lambda x.\,F\ (x\ x)).$$

This is a fixed point of F, since

$$(\lambda x.\,F\ (x\ x))\ (\lambda x.\,F\ (x\ x)) \quad \rightarrow \quad F\ ((\lambda x.\,F\ (x\ x))\ (\lambda x.\,F\ (x\ x))).$$

Curiously, although *every* term is a fixed point of the identity map $\lambda x.\,x$, the Y combinator produces a particularly unfortunate one, namely the divergent term Ω introduced in Lecture 2.

5.3 A call-by-value Y combinator

The Y combinator shown works perfectly in call-by-name evaluation, but in call-by-value evaluation, it produces divergent functions. The problem is in step 2, where we passed a self-application term $(x\ x)$ that can diverge. If that term diverges, the left-hand side of the equation will diverge when applied to an argument, even though the right-hand side wouldn't. When the Y combinator is used in a CBV setting, it tries to fully unroll the recursive function definition before any use of the function, leading to divergence.

The call-by-value divergence problem can be fixed by wrapping the self-application term in another lambda abstraction: $\lambda z.\,(x\ x)\ z$. This term yields the same result as $(x\ x)$ when applied to any argument, but is a value, and therefore will only be evaluated when it is applied. The effect of wrapping the term is to delay evaluation as long as possible, simulating what would happen in call-by-name evaluation.

So the call-by-value Y combinator is:

$$Y_{\mathrm{CBV}} \triangleq \lambda f.\,(\lambda x.\,f\ (\lambda z.\,(x\ x)\ z))\ (\lambda x.\,f\ (\lambda z.\,(x\ x)\ z))$$

5.4 Other fixed-point combinators

There other fixed-point combinators. For example, we can observe that since $Y\ F$ is supposed to be a fixed point of F, we are trying to solve the equation $Y\ F = F(Y\ F)$, for any F. Therefore $Y = \lambda f.\,f\ (Y\ f)$, which is a recursive function definition. Directly applying the self-application trick of Section 5.1 to this function definition, we obtain a fixed-point combinator discovered by Alan Turing:

$$\Theta \triangleq (\lambda y.\lambda f.f\ (y\ y\ f))\ (\lambda y.\lambda f.f\ (y\ y\ f))$$

In fact, there are infinitely many fixed-point combinators!