CS411 Untyped Lambda Calculus with Exercises

A. Demers

10 Apr – due 17 Apr

In these notes we give an overview of the untyped lambda calculus. We relate it to a subset of our example language with recursive type declarations. We argue that the untyped lambda calculus and the recursively typed programming language subset have equivalent expressive power — both are Turing complete.

A few homework exercises are interspersed.

1 Untyped Lambda Calculus

Consider the recursive type declaration

$$U \equiv \mu X.\mathbf{fun}(X)X$$

Intuitively this defines a type U that is equal to the space of functions from U to U — a "model for the untyped lambda calculus," If you don't yet know what that means, you're about to learn some elementary things about it.

Terms in the untyped lambda calculus are defined by the following simple grammar:

$$e \quad ::= \quad x \mid \lambda x.e \mid e_1(e_2)$$

That is all there is to this minimalist language. There are no "base types" like **int** or **bool**, no constants, and no typing rules. Every expression is a function, and all you can do with a function is apply it or pass it as an argument.

Semantics is usually presented as the following three rules.

$$\lambda x.e \rightarrow \lambda y.[y/x]e \quad \text{provided no capture occurs} \qquad \text{(alpha)}$$

The alpha rule allows bound variables to be renamed provided no capture occurs as a result of the renaming.

$$(\lambda x.e_1)(e_2) \quad \rightarrow \quad [e_2/x]e_1 \qquad \text{(beta)}$$
$$\text{provided no capture occurs}$$

The beta rule is essentially a "copy rule" as we discussed earlier in the course. Note that repeated uses of the alpha rule may be required to eliminate the possibility of capture and make the beta rule applicable.

$$e \quad = \quad (\lambda x.e(x)) \qquad \text{(eta)}$$
$$x \text{ not free in } e$$

Intuitively the eta rule says that the *entire* meaning of an expression is its behavior as a function — if you take an expression and explicitly turn it into a function, you do not change its meaning.

The lambda calculus is usually interpreted using lazy evaluation; but the above rules can form the basis of either a lazy or an eager semantics, determined by the order in which *redexes* – subexpressions to which the beta rule can be applied – are selected and reduced.

Except for termination, all reduction orders in the untyped lambda calculus are equivalent. That is, the system has the

Church-Rosser Property:

$$((e_1 \rightarrow^* e_2) \wedge ((e_1 \rightarrow^* e_3)) \Rightarrow$$
$$(\exists e_4)((e_2 \rightarrow^* e_4) \wedge ((e_3 \rightarrow^* e_4))$$

Consequently, all terminating reduction sequences must produce the same value.

Leftmost-outermost reduction gives lazy evaluation semantics. It is equivalent to an unrestricted copy rule — reduction of beta-redexes in any arbitrary order — and has the property that it terminates if there is *any* way to reduce the expression to normal form.

In contrast, any scheme that requires the argument expression of an application to be fully reduced before beta reducing the application will give eager semantics, sometimes diverging when lazy semantics would converge.

None of this is expected to be obvious; in fact it is rather deep and not at all trivial to prove.

Even though the untyped lambda calculus has no scalar types and no recursive function declaration capability, it is Turing complete — any computable function can be defined within it. To prove this it is sufficient to show how to simulate the natural numbers, Booleans and conditional execution, and how to do recursive function definitions, since it is clear that these features are enough to enable you to write a Turing machine simulation program. We'll do this below.

2 Translating Lambda Terms into Recursively Typed Terms.

The grammar of the untyped lambda calculus is a (small) subset of the grammar of our example language. Consequently, every lambda calculus term is an expression in our language. Of course, it might not be a *well typed* expression. It turns out that with the single recursive type U introduced above we can give a translation of any untyped lambda calculus term into an equivalent well typed term in our language. The translation uses only **lambda**, variables, application, **mu**, **abs** and **rep**. In particular there are no **let** bindings, no use of assignment, and no types other than the recursive function type U given above. It is defined by induction on lambda terms as follows:

$$\begin{array}{lcl} T[\![x]\!] & = & x \\ T[\![e_1(e_2)]\!] & = & (\mathbf{rep}_U(T[\![e_1]\!]))(T[\![e_2]\!]) \\ T[\![\lambda x.e]\!] & = & \mathbf{abs}_U(\lambda x : U.T[\![e]\!]) \end{array}$$

Proving computational equivalence of untyped lambda terms and their translations is a substantial exercise, and we won't go through the details here. However, you should convince yourself of the following partial result, which is provable by induction on evaluation derivations in the programming language:

Proposition: Suppose

$$\langle T[\![e_1]\!], \phi, \sigma\rangle \ \rightarrow \ \langle v_2, \sigma\rangle$$

then there exists an untyped lambda term e_2 such that

$$e_1 \ \rightarrow^*_{\alpha\beta} \ e_2 \qquad \text{and} \qquad T[\![e_2]\!] = v_2$$

□

The translation of untyped lambda calculus into well typed terms depends critically on recursive types. To see this, recall that in earlier exercises we considered versions of our language without loops, assignable variables, recursive functions or recursive types. For well typed terms in such a language, a *strong normalization* theorem holds — one can show by induction on type derivations that every evaluation terminates. When recursive types are introduced, however, strong normalization fails. Consider the untyped lambda term

$$(\lambda x.x(x))(\lambda x.x(x))$$

or its translated typed version

$$(\mathbf{rep}_U(E))(E)$$
$$\text{where } E \equiv \mathbf{abs}_U(\lambda x : U.(\mathbf{rep}_U(x))(x))$$

A single beta reduction on the untyped term yields

$$\begin{aligned}(\lambda x.x(x))(\lambda x.x(x)) &\rightarrow [(\lambda x.x(x))/x](x(x)) \\ &\rightarrow (\lambda x.x(x))(\lambda x.x(x))\end{aligned}$$

The term reduces nontrivially to itself, leading to an infinite nonterminating reduction sequence. It is not difficult to see that this is the *only* reduction sequence possible.

Exercise 1. Show that the typed version of this term has the same nonterminating behavior. □

The conclusion to be drawn here is that recursion fundamentally increases the expressiveness of the type system. With a single recursive type U, we can mimic the untyped lambda calculus. We are about to develop the promised result that the untyped lambda calculus is Turing complete; so in some sense recursion makes the type system as expressive as possible.

3 Lambda Calculus is Turing Complete.

As promised, we now show that the untyped lambda calculus is Turing complete. We do this by defining an encoding function $E[\![\cdot]\!]$ that maps numbers, Booleans and conditional execution, and recursive function definitions into untyped lambda terms.

3.1 Numbers.

The following construction is sometimes called the "Church numerals" after the mathematician A. Church. The basic idea is to translate a natural number n to a function of two arguments, which applies its first argument to its second n times. That is,

$$E[\![n]\!] \;=\; \lambda f.\lambda x.f^n(x)$$

For any particular n this is trivial:

$$\begin{aligned} E[\![0]\!] &= \lambda f.\lambda x.x \\ E[\![1]\!] &= \lambda f.\lambda x.f(x) \\ E[\![2]\!] &= \lambda f.\lambda x.f(f(x)) \\ &\cdots \end{aligned}$$

Given these encodings of the numbers, how might we encode the successor function? It's pretty straightforward:

$$E[\![succ]\!] \;=\; \lambda n.(\lambda f.\lambda x.f((n(f))(x)))$$

It is easy to see that a function that applies f to x exactly $n+1$ times can be constructed from a function that applies f to x exactly n times (that is, from the encoding of n) by a direct definition using no recursion.

Exercise 2. Give a nonrecursive definition for the encoding of the function add, which computes the sum of two numbers. The relation

$$(((E[\![add]\!])(E[\![n_1]\!]))(E[\![n_2]\!]) \;=\; E[\![n]\!] \quad \text{where } n = (n_1 + n_2)$$

should hold. □

3.2 Tuples.

How far can we get without conditionals and recursive function definitions? Since we're trying to show that we can *define* conditionals and recursive functions with the mechanism we have so far, the answer to this question is at best subjective. But here is a (perhaps excessively involved, but instructive) way to encode the predecessor function using an encoding of ordered pairs.

Suppose we have (the encoding of) a number n. We can use it to apply any function we want to any argument we want, n times. Could that enable us to apply a function $n-1$ times? Believe it or not, yes, by the following clever trick. Assume we have a way to construct and decompose ordered pairs. We can take any function g, and extend it to a function G on ordered pairs such that

$$G(\langle u, v\rangle) \;=\; \langle v,\; g(v)\rangle$$

Note this is not just applying g to each element of the ordered pair; rather, we are using the ordered pair to "remember" the value of the argument of g. Now the sequence

$$\langle x, x\rangle,\; G(\langle x, x\rangle),\;\; \ldots\; G^i(\langle x, x\rangle),\;\; \ldots$$

is just

$$\langle x, x\rangle,\;\; \ldots\; \langle g^{i-1}(x),\; g^i(x)\rangle,\;\; \ldots$$

and the first component of

$$G^n(\langle x, x\rangle)$$

gives us the predecessor of n.

To define ordered pairs, we first define *projection functions*

$$\mathrm{proj}_1 \;=\; \lambda x.\lambda y.x \qquad \mathrm{proj}_2 \;=\; \lambda x.\lambda y.y$$

Next, we arrange to represent an ordered pair $\langle u, v\rangle$ by something that applies a projection function to u and v:

$$\mathrm{mkpr} \;=\; \lambda x.\lambda y.(\lambda r.(r(u))(v))$$

Clearly

$$((\mathrm{mkpr}(u))(v))(\mathrm{proj}_1) \;=\; u \qquad \text{and} \qquad ((\mathrm{mkpr}(u))(v))(\mathrm{proj}_2) \;=\; v$$

So we are able to construct and select components of ordered pairs.

Exercise 3. Complete the development of the predecessor function encoding. First define functions fst and snd that extract the first and second components of an ordered pair. Then define a function extend that extends functions to ordered pairs as described above; it should satisfy

$$(\text{extend}(g))(\text{mkpr}(u,v)) \quad = \quad \text{mkpr}(v, g(v))$$

Finally, make use of your new collection of "macro definitions" to give a nonrecursive definition of the predecessor function. By the way, we conventionally define the predecessor of zero to be zero; you'll find this works out conveniently. □

Exercise 4. Now that we have shown how to define the predecessor function, give a nonrecursive definition for the encoding of the function *sub*, which computes the difference between two numbers. Your definition should satisfy

$$\begin{array}{rcl} (((E[\![sub]\!])(E[\![n_1]\!]))(E[\![n_2]\!]) & = & E[\![n]\!] \\ \quad \text{where } n = (n_1 - n_2) & & \end{array}$$

if $n_1 > n_2$, and

$$(((E[\![sub]\!])(E[\![n_1]\!]))(E[\![n_2]\!]) \quad = \quad E[\![0]\!]$$

if $n_1 \leq n_2$. □

We haven't quite finished with the natural numbers — we need predicates $<$, $=$, ..., which compare numbers. Of course, we can't define those until we've shown how to encode Boolean values.

3.3 Booleans and Conditionals.

Having understood the ordered pair encoding used above, you should have little trouble with Booleans and conditionals. We simply encode the two truth values as the binary projection functions; that is,

$$\begin{array}{rcl} E[\![\textbf{true}]\!] & = & \lambda x.\lambda y.x \\ E[\![\textbf{false}]\!] & = & \lambda x.\lambda y.y \end{array}$$

With this interpretation it is easy to encode the usual Boolean operators.

$$\begin{array}{rcl} E[\![\textbf{not}]\!] & = & \lambda u.(\lambda x.\lambda y.(u(y))(x)) \\ E[\![\textbf{or}]\!] & = & \lambda u.\lambda v.(\lambda x.\lambda y.(u(x))((v(x))(y))) \end{array}$$

It is easy (if tedious) to verify that these definitions have the correct behavior for all possible combinations of **true** and **false** arguments u and v.

It is tempting to encode **if** expressions simply by

$$E[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] \quad = \quad ((E[\![e_1]\!])(E[\![e_2]\!]))(E[\![e_3]\!])$$

That is, $E[\![e_1]\!]$ is the encoding of a Boolean, and thus always returns a binary projection function. If that projection function is $E[\![\textbf{true}]\!]$, then the result is e_2; otherwise the result is e_3. In fact this intuitive approach works correctly in the lambda calculus with lazy evaluation. It fails if eager evaluation is used, sometimes diverging when it should not. For example, suppose e_2 converges and e_3 diverges. Clearly $E[\![e_3]\!]$ must also diverge, or the encoding E would be incorrect. The expression

$$\textbf{if true then } e_2 \textbf{ else } e_3$$

should converge, since e_2 converges. But the expression

$$((E[\![\textbf{true}]\!])(E[\![e_2]\!]))(E[\![e_3]\!])$$

diverges under eager evaluation rules, since evaluation of the argument $E[\![e_3]\!]$ diverges. Fortunately, this problem is not difficult to fix.

Exercise 5. Give an encoding of conditional expressions that works with eager evaluation rules. Argue convincingly that your encoding is correct. □

As we remarked above, we haven't quite finished with the natural numbers — we need predicates to compare them. With the machinery we have built up this is easy.

Exercise 6. Give the encoding of a function leq, which compares two numbers. It should satisfy

$$\begin{array}{rcl}(E[\![leq]\!](E[\![n_1]\!]))(E[\![n_2]\!])) & = & E[\![b]\!] \\ \quad \text{where } b & = & (n_1 \leq n_2)\end{array}$$

With the Boolean operations above this gives us all the necessary predicates on natural numbers. □

3.4 Recursion.

At last we get to general recursive functions. Assume there is an explicit construct for recursive function expressions

$$e \ ::= \ (\textbf{recfun}\ f\ \sim\ \textbf{lambda}\ x\ \textbf{dot}\ e)$$

where the recursive function name f may (and presumably does) occur free in e. Assume, as discussed in lecture, the meaning of a **recfun** expression is the least fixed point of the corresponding functional

$$F \ \equiv \ \textbf{lambda}\ f\ \textbf{dot}\ (\textbf{lambda}\ x\ \textbf{dot}\ e)$$

Finally, assume inductively that we can encode F — it contains fewer instances of **recfun** than the original expression did. Thus, our encoding for the recursive function expression is

$$E[\![(\textbf{recfun}\ f\ \sim\ \textbf{lambda}\ x\ \textbf{dot}\ e)]\!] \ = \\ Y(\ E[\![\textbf{lambda}\ f\ \textbf{dot}\ (\textbf{lambda}\ x\ \textbf{dot}\ e]\!])\ )$$

and our remaining task is to give an untyped lambda expression Y that computes a least fixed point.

To get some intuition for this, recall a lambda expression we presented above

$$(\lambda x.(x(x)))(\lambda x.(x(x)))$$

which reduces nontrivially to itself. By modifying the body of the individual lambda terms in this expression we can make it reduce nontrivially to some function of itself. Since we seek a fixed point of F, the obvious modification is to introduce an application of F. That is,

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

This suggests that the lambda expression

$$Y \ \equiv \ \lambda f.(\lambda x.f(x(x)))(\lambda x.f(x(x)))$$

should act as a fixed point operator. (It is perhaps not obvious why it should compute a *least* fixed point; we'll return to this issue later).

Here is a situation we have encountered before. In the lazy lambda calculus, the above definition of Y works as desired. If eager evaluation is used, then naturally $Y(F)$ diverges for any F.

Exercise 7. Consider the following **recfun** expression with no recursion:

$$\textbf{recfun}\ f\ \sim\ \textbf{lambda}\ x\ \textbf{dot}\ x$$

What is the corresponding functional F? Show that $(Y(F))(zero)$ converges (to the correct answer) using lazy evaluation — that is, give a terminating reduction sequence for this expression. □

Fortunately, it is not too difficult to come up with a fixed point function Y_e that works with eager evaluation. The eta rule given above asserts that any expression e is equal to $\lambda x.(e(x))$, (at least as far as its behavior as a function is concerned). But the two expressions are not equivalent for the purposes of eager evaluation — one of them $(\lambda x.(e(x)))$ can always be passed as an argument, even if the other (e) is divergent. We can exploit this behavior to derive Y_e.

To fill in the details, recall we used

$$Y(F) \quad = \quad F(Y(F))$$

to define what it means for $Y(F)$ to be a fixed point of F. Suppose we modify this definition slightly by applying the eta rule to the right hand side, to obtain

$$Y_e(F) \quad = \quad \lambda z.((F(Y_e(F)))(z))$$

as a proposed fixed point definition for Y_e. An expression satisfying this equation can be derived by the same heuristic we used to derive Y above — slightly modifying the body of our archetypal divergent expression

$$(\lambda x.x(x))(\lambda x.x(x))$$

Specifically, let

$$Y_e \quad \equiv \quad \lambda f.((\lambda x.\lambda z.(f(x(x))(z)))(\lambda x.\lambda z.(f(x(x))(z))))$$

A single outermost beta reduction on $Y_e(F)$ yields

$$\begin{aligned} Y_e(F) \quad &\rightarrow \quad ((\lambda x.\lambda z.(F(x(x))(z)))(\lambda x.\lambda z.(F(x(x))(z)))) \\ &\rightarrow \quad \lambda z.(F((\lambda x.\lambda z.(F(x(x))(z)))(\lambda x.\lambda z.(F(x(x))(z)))))(z) \\ &\equiv \lambda z.(F(Y_e(F)))(z) \end{aligned}$$

as required. Note that z is not free in

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

so no capture occurs in the above substitutions, as the beta rule requires.

Exercise 8. Repeat Exercise 7 above using Y_e with eager evaluation (where a term of the form $\lambda x.e$ is irreducible) rather than Y with lazy evaluation. □