Coding in Lambda Calculus

Norman Ramsey

Spring 2019

Introduction

The lambda calculus is a *universal model of computation.* What this means is the proper study of a course in theory of computation, but roughly speaking, lambda calculus is as powerful as any deterministic, sequential computer that we can imagine. Lambda calculus is equivalent to that other universal model of computation, the Turing machine.

Roughly speaking, "universal" also means "can run any program." To convince yourself that the lambda calculus really is universal, there is no better exercise than to translate code into the lambda calculus. One of the benefits of lambda calculus is that translating code into lambda calculus is relatively painless. By contrast, translating code for a Turing machine demands a compiler.

Given that high-level languages have all sorts of syntactic forms and data structures, whereas lambda calculus has only three forms of term, it's not obvious that high-level languages can be translated. This handout sketches some standard translations. Ideas about translating syntax and translating data are mixed freely.

Summary of lambda calculus

To write lambda calculus, I use the concrete syntax of the lambda interpreter in the homework. If terms are written M and N, and variables are written x and y, there three forms of term:

$$M ::= x \mid \backslash x\,.\,M \mid M_1\ M_2$$

And there is one form of definition

def ::= x `=` M `;`

(I ignore `noreduce` as a mere instruction to the interpreter.)

Lambda calculus is not evaluated in the same way that other languages are evaluated: the closest thing to evaluation is *reduction to normal form.* That said, we can *think* of lambda calculus as being evaluated like any other functional language (think Scheme or ML), *except* that the actual parameters to functions are not evaluated—instead, actual parameters are passed around in unevaluated state, and they are evaluated only when and if needed. This delay or "laziness" in evaluation is exploited by the coding of algebraic data types.

Natural numbers

A natural number n is coded using an idea of Alonzo Church's: n is coded as the capability of taking a function f and an argument x, and applying f to x n times. Examples:

```
<0> = \f.\x.x;
<1> = \f.\x.f x;
<2> = \f.\x.f (f x);
<3> = \f.\x.f (f (f x));
```

We can't write down all the natural numbers, but any natural number, no matter how large, is either zero or is the successor of some other natural number ($m + 1$). The successor of m has the ability to apply f to x m times, plus one more time, obeying this law:

```
// succ m f x = f (m f x)
```

and we can therefore define successor as follows:

```
succ = \m.\f.\x.f (m f x);
```

To add n to m, we take the successor of m, n times:

```
// plus n m = n succ m
plus  = \n.\m.n succ m;
```

To multiply n by m, we can add m to zero, n times:

```
// times n m = n (plus m) <0>
times = \n.\m.n (plus m) <0>;
```

Left as an exercise: m raised to the nth power.

Other clever tricks can be accomplished by choice of a suitable function f. For example, consider what happens if you choose a function f that always returns `true`.

Constructed data and case expressions

When thinking in lambda calculus, I code every form of data as an instance of one of these three concepts: a function, a natural number, or constructed data. (Both natural numbers and constructed data are then coded as functions.) The coding of constructed data is based, as always, on the ability to look at a constructed value and answer two questions:

- How were you made?
- From what parts?

These questions are the same questions that we can ask about value constructors and constructed data in ML, except that in lambda calculus, a value constructor doesn't take a tuple; instead, it's Curried, with zero or more arguments.[1] And each constructed datum is coded as the capability of answering these two questions. The questions are posed as follows:

- Each possible answer to "how were you made?" is coded as a continuation. That is, there is one continuation per form of data.
- Each continuation takes the "parts" as arguments.

Here's an example. There are two ways to make a list:

- `nil` makes an empty list, which does not have any parts.
- `cons` makes a nonempty list, from two parts.

Supposing I have one continuation `kn` for `nil` and another continuation `kc` for cons, the constructed data obey these algebraic laws:

```
// nil         kn kc = kn
// (cons x xs) kn kc = kc x xs
```

We therefore have these definitions:

```
nil  =        \kn.\kc.kn;
cons = \x.\xs.\kn.\kc.kc x xs;
```

What can we do with the constructed data? Pattern match! Lambda calculus doesn't have pattern matching or `case` expressions, but it can still express a limited form of pattern matching with lambdas. This form corresponds to a `case` expression in which every pattern is a value constructor applied to zero or more variables. So to translate ML code into lambda calculus, we first reduce all pattern matching into cases of this form. As an example, I translate ML's `null` into lambda calculus.

First, I write the classic clausal definition:

```
fun null []      = true
  | null (x::xs) = false
```

Next, I translate it to use ML's lambda syntax (`fn`) and its `case` syntax:

```
val null = fn ys => case ys of []      => true
                             | x :: xs => false
```

Now I turn each case "arm" into a continuation:

- The arm `[] => true` becomes continuation `kn`, which takes no arguments and returns `true`.

  Continuation `kn` is just `true`.

- The arm `x :: xs => false` becomes continuation `kc`, which takes arguments `x` and `xs` and returns `false`.

  Continuation `kc` is `\x.\xs.false`.

[1]This sensible system for value constructors is also used in the programming language Haskell.

Finally, I translate the `case` expression by applying the constructed data to its continuations:

```
null = \ys . ys (true) (\x.\xs.false);
```

Here's another one:

```
singleton? = \ys. ys false (\x.\xs.null xs);
```

Here's another type and some functions:

```
datatype 'a option = NONE | SOME of 'a
(* Option.map f NONE     = NONE
   Option.map f (SOME x) = SOME (f x)

   valOf (SOME x) = x   *)
```

Here are their codings in lambda calculus

```
NONE =    \kn.\ks.kn;
SOME = \a.\kn.\ks.ks a;
Option.map = \f.\v.v NONE (\a.SOME (f a));
valOf = \v.v bot (\a.a);
```

Here `bot` is the divergent term `(\x.x x)(\x.x x)`. (In ML, `valOf NONE` results in a checked run-term error. In lambda calculus, nontermination, which is usually called "divergence," is the closest thing we have to an error.)

Booleans

We're used to Booleans being special, and they get their own special elimination form (the `if` expression), but in practice, Booleans are just constructed data:

```
datatype bool = true | false
// if P then ET else EF  ==
//    case p of true => ET | false => EF
```

We have our algebraic laws:

```
// true  ET EF = ET
// false ET EF = EF
```

and here are the definitions:

```
true = \x.\y.x;
false = \x.\y.y;
```

Pairs

Pairs are just constructed data with special syntactic support. Roughly speaking,

```
// type 'a * 'b == PAIR of 'a * 'b   // really Curried
// (e1, e2)     == PAIR e1 e2
// let val (x, y) = p in e end ==
//    (case p of (x, y) => e)
```

In other words, a pair is a single form of data (one continuation) made from two parts (two_arguments to the continuation):

```
// pair x y k = k x y
pair = \x.\y.\k.k x y;
```

It's easy and efficient to program with pairs and continuations directly, but we can also translate some ML functions:

```
fun swap (x, y) = (y, x)
fun fst  (x, y) = x
fun snd  (x, y) = y

fun swap p = case p of (x, y) => (y, x)
fun fst  p = case p of (x, y) => x
fun snd  p = case p of (x, y) => y

swap = \p.p (\x.\y.pair y x);
fst  = \p.p (\x.\y.x);
snd  = \p.p (\x.\y.y);
```

Other high-level data

Aside from natural numbers and algebraic data types, high-level languages offer other data representations. But they can be translated!

- As in C, a *character* is just a natural number. (Think Unicode "code point.")
- A *string* is a list of characters.
- An *array* is the same abstraction as a list—it just offers a different cost model (constant-time access, no growth or shrinkage). Lambda calculus, like a Turing machine, does not offer a sequence structure with constant-time access. As a result, costs relative to modern hardware may be larger by a factor polynomial in the size of the input.
- *Records* are coded in the same way as pairs and other tuples. Record names are treated as syntactic sugar.

High-level expression forms

What about translating high-level expression forms into lambda calculus? Here's a list:

- The `LITERAL` form is not needed. All values are coded as expressions, so any "literal value" is simply written directly as an expression.
- The `VAR` form is supported directly.
- The `LAMBDA` form is supported directly, but it is limited to single-argument functions. A multiple-argument function is coded by Currying. A zero-argument function is coded as just its body—thanks to the unusual evaluation model of the lambda calculus, an unprotected body behaves the same way as zero-argument function in Scheme.
- The `APPLY` form is supported directly, but is limited to applying a function to a single actual parameter. Since all functions are Curried, this works out just fine.
- The `case` form (not found in μScheme, but found in ML and in μML) is coded by applying the *scrutinee* to one or more continuations, as described above.
- The `IFX` form is coded by applying the condition to the true and false branches. It is a special case of `case`.
- The `LET` form is coded by a combination of `APPLY` and `LAMBDA`:

  ```
  (let ([x1 e1] ... [xn en]) e)
  ```

  is coded as

  ```
  ((lambda [x1 ... xn] e) e1 ... en)
  ```

- The `LETSTAR` form (and ML's `let/val` form) is syntactic sugar for a nested series of `LET` forms. The coding is described in the book.
- The `BEGIN` form can be coded using `let*`:

  ```
  (begin e1 e2 ... en)
  ```

  is coded as

  ```
  (let* ([_ e1]
         [_ e2]
         ...
         [_ en])
    _)
  ```

  where `_` stands for any name that is not free in any of the `e`'s.

- The `WHILE` form can be coded using a recursive function. You did something similar for homework:

  ```
  (while e1 e2)
  ```

  is coded as

  ```
  (letrec
     ([loop (lambda ()
              (if e1 (begin e2 (loop)) #f))])
    (loop))
  ```

  where `loop` is a name that is not free in `e1` or `e2`.

This is almost everything you might find in a high-level language. We're left with `LETREC`, which is treated below, and `SET`.

Mutation (`SET`) doesn't play well with lambda calculus—in lambda calculus, as in ML and Haskell, a name stands for a value, not a mutable location.[2] If you want to simulate mutation in lambda calculus, I don't know a better way than just simulating the operational semantics of a mutable store. (What happens in practice is that we learn to write algorithms that use `let*` instead

[2]In Haskell, the real story is more complicated, as a name may stand for a "thunk" containing an unevaluated expression, but a name definitely does *not* stand for a location that user code can mutate.

of `set`, and formal parameters instead of mutable variables. This trick works well.)

Recursion

Lambda calculus has nothing like `letrec` or `define`. There is no recursion in the lambda calculus, and when we use the (only) definition form, all variables on the right-hand side have to be defined in the environment already (so they can be substituted). So the lambda calculus might look like a dead end—how can we write anything interesting with no loops or recursion?—but there is an amazing trick. The trick is more than just a trick, however: it is based in the ultimate truth about how we think about and define recursions. This method, the method of *successive approximations*, is where we begin.

Approximations (mathy)

This section defines, technically, what an approximation is. It's mathy, and you can think about skipping it.

All approximations begin with a divergent term. I use

```
noreduce bot = (\x.x x)(\x.x x);
```

Term `bot` beta-reduces to itself in one step, so any attempt to normalize it runs away in an endless sequence of reductions. (This behavior is called *divergence*.) This behavior makes `bot` a really bad *approximation* to factorial. Approximation is defined on both functions and lambda terms, but the place to start is with functions:

> Function f approximates g ($f \sqsubseteq g$) if g is defined on at least as many inputs as f is, and where they are both defined, they agree.

Getting super pedantic, we can adapt this definition for lambda terms:

> Term M_1 approximates term M_2 if for every term N, the following property holds:
>
> - If $M_1\ N$ has a normal form, then $M_2\ N$ also has a normal form, and furthermore, these two normal forms are equivalent up to alpha-renaming.

Therefore, `bot` approximates factorial, because for every term N, `bot` N doesn't have a normal form, and the property holds trivially. (The property makes it an approximation, and the triviality makes it a *bad* approximation.)

Approximating factorial

We'd really like to define factorial recursively:

```
// WRONG!
// fact = \n.(zero? n) <1> (* n (fact (pred n)));
```

This idea won't work: we can imagine that `zero?`, `1`, `*`, and `pred` are all defined, but there is no way that `fact` is defined. However, since `fact` is all we're missing, we can slot in an approximation: on the right-hand side, instead of using `fact`, we can use `bot`:

```
noreduce fact0 = \n.(zero? n) <1> (* n (bot (pred n)));
```

And we can usefully apply this to zero, but not one:

```
-> fact0 <0>;
\f.f
-> fact0 <1>;
Failed to normalize after 100000 reductions
DIVERGENT TERM fact0 <1>
fact0 <1>
->
```

But now we can use `fact0` to define a better approximation: we go back to our recursive definition, and on the right-hand side, we plug in `fact0` in place of the recursive call:

```
noreduce fact1 = \n.(zero? n) <1> (* n (fact0 (pred n)))
```

If we pass 0 to `fact1`, it works just like it did before. But now if we pass 1 to `fact1`, it takes the predecessor to get 0, passes 0 to `fact0`, and it *still* works:

```
-> fact1 <1>;
\f.f
```

More things that work:

```
noreduce fact2 = \n.(zero? n) <1> (* n (fact1 (pred n)))
noreduce fact3 = \n.(zero? n) <1> (* n (fact2 (pred n)))
```

Approximations keep getting better!

```
-> fact2 <3>;
Failed to normalize after 100000 reductions
DIVERGENT TERM fact2 <3>
fact2 <3>
-> fact3 <3>;
\f.\x.f (f (f (f (f (f x)))))
```

Plugging things in by hand grows tiresome. Fortunately, beta reduction gives us a tool for plugging things in. I can define an "approximation builder" that takes an approximation as a parameter, then plugs it in, leaving me with a better approximation. Because it's a "factorial builder," I call it `fb`:

```
fb = \f.\n.(zero? n) <1> (* n (f (pred n)));

noreduce fact4 = fb fact3;
noreduce fact5 = fb fact4;
noreduce fact6 = fb fact5;
```

Now we can start computing some decent-sized factorials:

```
-> fact6 <4>;
\f.\x.f (f (f (f (f (f (f (f (f (f (f (f
        (f (f (f (f (f (f (f (f (f (f (f x)
        )))))))))))))))))))))))
```

How an approximation works

Let's sketch how those calls actually work. To begin, let's look more closely at the structure of `fact6`:

```
fact6 = fb fact5 = fb (fb fact4)
  = fb (fb (fb fact3))
  = fb (fb (fb (fb fact2)))
  = fb (fb (fb (fb (fb fact1))))
  = fb (fb (fb (fb (fb (fb bot)))))
```

By itself this term does nothing interesting—it just diverges. But when we apply it to the Church numeral for 4, we suddenly have some more interesting reductions available. To see how things evolve, I'll exploit the algebraic laws for Booleans and for Church numerals, plus a few more:

```
// fb f n = (zero? n) <1> (* n (f (pred n)))
// zero? <0> = true
// zero? (succ n) = false
```

Let's calculate![3]

```
fact6 4
  = { definition of `fact6` above }
fb (fb (fb (fb (fb (fb bot))))) 4
  = { law of `fb` }
(zero? 4) 1 (* 4 (fb (fb (fb (fb (fb bot)))) (pred 4)))
  = { 2nd law of `zero?` }
false 1 (* 4 (fb (fb (fb (fb (fb bot)))) (pred 4)))
  = { law of `false` }
(* 4 (fb (fb (fb (fb (fb bot)))) (pred 4)))
  = { law of `pred` }
(* 4 (fb (fb (fb (fb (fb bot)))) 3))
  = { law of `fb` }
(* 4 ((zero? 3) 1 (* 3 (fb (fb (fb (fb bot))) (pred 3)))))
  = { 2nd law of `zero?` }
(* 4 (false 1 (* 3 (fb (fb (fb (fb bot))) (pred 3)))))
  = { law of `false` }
(* 4 (* 3 (fb (fb (fb (fb bot))) (pred 3))))
  = { law of `pred` }
(* 4 (* 3 (fb (fb (fb (fb bot))) 2)))
  = { law of `fb` }
(* 4 (* 3 ((zero? 2) 1 (* 2 (fb (fb (fb bot)) (pred 2))))))
  = { 2nd law of `zero?` }
(* 4 (* 3 (false 1 (* 2 (fb (fb (fb bot)) (pred 2))))))
  = { law of `false` }
(* 4 (* 3 (* 2 (fb (fb (fb bot)) (pred 2)))))
  = { law of `pred` }
(* 4 (* 3 (* 2 (fb (fb (fb bot)) 1))))
  = { law of `fb` }
(* 4 (* 3 (* 2 ((zero? 1) 1 (* 1 (fb (fb bot) (pred 1)))))))
  = { 2nd law of `zero?` }
(* 4 (* 3 (* 2 (false 1 (* 1 (fb (fb bot) (pred 1)))))))
  = { law of `false` }
(* 4 (* 3 (* 2 (* 1 (fb (fb bot) (pred 1))))))
  = { law of `pred` }
(* 4 (* 3 (* 2 (* 1 (fb (fb bot) 0)))))
  = { law of `fb` }
(* 4 (* 3 (* 2 (* 1 ((zero? 0) 1 (fb bot (pred 0)))))))
  = { at long last, the first law of `zero?` }
(* 4 (* 3 (* 2 (* 1 (true 1 (fb bot (pred 0)))))))
  = { law of `true` }
(* 4 (* 3 (* 2 (* 1 1))))
```

[3]To make the calculation fit, I've taken the angle brackets off all the numerals.

Perfect approximations via fixed points

The key thing making the calculation `fact6 4` work is that we never get to `bot`: we have enough applications of `fb` so that eventually, we stop the recursion. But a finite number of applications is not always good enough: `fact6` can compute the factorials of numbers up to 6, and no more. But there is an amazing trick, hinted at by the observation that we don't actually use `bot`: suppose we replace `bot` with a *different* diverging term, let's call it `F`, so that when it "reduces," it actually expands to give us more applications of `fb`. That is, we're looking for

```
F = fb F = fb (fb F) = fb (fb (fb F))
```

such an `F`, if it exists, is called a *fixed point* of `fb`. The shocking thing is that there is a lambda term that, when applied to `fb`, *is* a fixed point of `fb`. There are actually many such terms; the one we use is called the "Y combinator":

```
noreduce Y = \f.(\x.f(x x))(\x.f(x x));
```

Let's calculate!

```
Y fb
  = { definition of `Y` }
(\f.(\x.f(x x))(\x.f(x x))) fb;
  = { beta-reduction }
(\x.fb(x x))(\x.fb(x x));
  = { beta-reduction }
fb ((\x.fb(x x))(\x.fb(x x)));
  = { first two steps in this calculation, backward }
fb (Y fb)
```

And we didn't use any properties of `fb`! The Y combinator works for *any* function.

Calculating with the Y combinator

We've already done factorial. Let's do `length`. We just define a function `lb` ("length builder"), and instead of the recursive call, we plug in the parameter:

```
noreduce lb = \f.\xs.null? xs <0> (succ (f (cdr xs)));
// lb f xs = null? xs <0> (succ (f (cdr xs)))
noreduce length = Y lb;
```

And let's calculate:

```
length (cons A (cons B nil))
  = { definition of `length` }
Y lb (cons A (cons B nil))
  = { law of `Y` }
lb (Y lb) (cons A (cons B nil))
  = { law of `lb` }
null? (cons A (cons B nil)) <0> (succ ((Y lb) (cdr (cons
  = { null-cons law }
false <0> (succ ((Y lb) (cdr (cons A (cons B nil)))))
  = { law of `false` }
succ ((Y lb) (cdr (cons A (cons B nil))))
  = { cdr-cons law }
succ ((Y lb) (cons B nil))
  = { KEY STEP: law of `Y` }
succ (lb (Y lb) (cons B nil))
  = { law of `lb` }
succ (null? (cons B nil) <0> (succ ((Y lb) (cdr (cons B nil)))))
  = { null-cons law }
succ (false <0> (succ ((Y lb) (cdr (cons B nil)))))
  = { law of `false` }
succ (succ ((Y lb) (cdr (cons B nil))))
  = { cdr-cons law }
succ (succ ((Y lb) nil))
  = { KEY STEP: law of `Y` }
succ (succ (lb (Y lb) nil))
  = { law of `lb` }
succ (succ (null? nil <0> (succ (Y lb) (cdr nil))))
  = { null-nil law }
succ (succ (true <0> (succ (Y lb) (cdr nil))))
  = { law of `true` }
succ (succ <0>)
```

As expected, the length is 2. Each "KEY STEP" in the calculation is an expansion of `Y lb` to `lb (Y lb)`. Each one corresponds to one recursive call of `length`.

Replacing recursion

Every `define` form can be replaced with a `val` form that uses `Y`:

```
// (define f (x) e) becomes
// val f = Y (\f.\x.e);
```

The same ideas work on `letrec`:

```
// (letrec ([f (lambda (x) e)]) body)
//   becomes
// (\f.body) (Y (\f.\x.e))
```

When `letrec` defines multiple functions, you can put them into (nested) pairs, and use the fixed-point operator to approximate the pairs.

Summary

To translate any code into lambda calculus, choose from these transformations:

- Replace `while` with recursion and `set` with let-binding or function parameters. Replace `begin` with `let*`.
- Desugar `let*` into nested `let` expressions.
- Desugar `let` expressions into applied `lambda`s.
- Code signed integers and floating-point numbers in software, using natural numbers.
- Code natural numbers, unsigned integers, and characters using Church numerals.
- Code constructed data using continuations.
- Code sequences using lists.
- Curry all functions, and change applications to match.
- Translate conditionals into `case` expressions.
- Translate all pattern matching (`case`, `fun`, `let`) into `case` expressions.
- Un-nest patterns in `case` expressions, turning nested patterns into nested `case` expressions.
- Translate `case` into continuation-passing style.
- Translate recursion using the Y combinator .

Many of these transformations are standard compiler transformations.