The Lambda Calculus and the Recursion Theorem

In this course, Turing Machines served as our main formalization of how a computational device behaves. However, it is only one of many formalizations of computation. Alan Turing developed his machines in 1936 to answer the Entscheidungsproblem—devise an algorithm that determines if a given statement in first-order logic is universally valid—showing that no such algorithm exists. This was proven independently by Alonzo Church with a different model of computation, the lambda calculus, and in an appendix to his original paper, Turing sketched a proof that Turing Machines and the lambda calculus were equivalent. The Church-Turing Thesis goes on to posit that Turing machines and the lambda calculus capture the set of computable functions, those that we can realize using computational methods.

In 1965, Peter Landin described a connection between the ALGOL 60 programming language and the lambda calculus, noting how the notion of function in the lambda calculus captured the notion of procedure in a programming language. Since then, the lambda calculus has become the basis for the modern-day functional programming languages such as Scheme/Racket, OCaml, Scala, and Haskell. In this reading, we introduce the lambda calculus and explore its semantics to better tie our discussion of the limits of computation to programming. We also use the lambda calculus to explore one last area of computability that we have brushed over, recursion.

The Lambda Calculus

While the lambda calculus not originally designed as such, it is easiest to think of the lambda calculus as a core programming language, a bare-bones language equipped with a minimal set of features amendable to formal analysis. In this light, the lambda calculus is, perhaps, the ultimate realization of this dream: it only contains functions. For example, here is a lambda calculus program or expression for a function:

\[ \lambda x. x \]

This is a function that, when given a value, produces that same value. It is also known as the identity function.

To invoke this function, we perform function application, applying the function to an argument. In the lambda calculus, functions are first-class values—they can be passed as arguments to functions, saved to variables, etc. Thus, we can apply the identity function to another function, for example:

\[ (\lambda x. x) (\lambda y. \lambda z. z) \]

While the argument is a complicated-looking expression, recall that the function being invoked is simply the identity function. Thus, when we evaluate this function application, we receive the following result:

\[ (\lambda x. x) (\lambda y. \lambda z. z) \rightarrow \lambda y. \lambda z. z. \]

Note that Racket has nearly identical syntax for first-class functions and function application—the construct is called a “lambda” for a reason! In racket, we would declare this function application as:
((lambda (x) x) (lambda (y) (lambda (z) z)))

Indeed, if you’d like to check the results of evaluating any particular lambda calculus term, you can simply translate the lambda calculus term into Racket code and run it in Dr. Racket!

Syntax

The lambda calculus is a stripped-down functional programming language consisting solely of a handful of expression forms. This expression forms are also called terms in the literature, although we will use “expression” to describe them to emphasize their connection to programming languages that we know. The syntax of the language is concisely described by the following grammar:

\[ e ::= x \mid \lambda x. e \mid e_1 e_2 \]

And that’s it! The language consists of exactly three forms—variables \( x \), functions \( \lambda x. e \), and application \( e_1 e_2 \). There are no other syntactic forms in the language, no numbers, booleans, strings, or classes. Indeed, we shall show shortly that these are all unnecessary as variables, functions, and function application are sufficient to encode any language feature that we desire! Of course, in practice we want to have direct access to these language features rather than use potentially obtuse encodings. But the fact that anything we’d like in a programming language is encodable in the lambda calculus reassures us that we aren’t missing anything by stripping down our language to this degree.

For example, the following are valid lambda calculus expressions:

\[
\begin{align*}
\lambda x. x & \\
\lambda f. \lambda x. f x & \\
\lambda x. \lambda y. x & \\
\lambda x. \lambda y. y & \\
(\lambda s. \lambda z. s z) (\lambda x. x) & 
\end{align*}
\]

Evaluation

Like its syntax, the lambda calculus features a relatively simple evaluation model. The only thing that a lambda calculus expression does is evaluate function applications. This is captured by the notion of a substitution of a value for a variable.

Consider our example from before, the function application:

\[(\lambda x. x) (\lambda y. \lambda z. z)\]

evaluates to its argument \( \lambda y. \lambda z. z \). Function evaluation proceeds in two steps:

1. In the body of the function being applied (the left-hand side of the application), substitute the argument of the application (the right-hand side) for every occurrence of the function’s variable in the body.
2. The result of the function application becomes the body that was substituted into.

In this case, the body of the function is the variable \( x \) and the argument is the term \( \lambda y. \lambda z. z \). The function’s variable is \( x \), so we substitute the argument for this variable. The result is the body is now \( \lambda y. \lambda z. z \). Finally, this substituted body becomes the overall result of the function application.

We can codify this process into the following rule for evaluating a lambda calculus expression:

\[
(\lambda x. e_1) e_2 \rightarrow [e_2/x] e_1
\]

Like an arithmetic expression, we can write down how a lambda calculus expression evaluates step-by-step. We write \( e \rightarrow e' \) to mean that expression \( e \) steps to expression \( e' \) after one step of evaluation. The above rule says that a function application of the form \((\lambda x. e_1) e_2\) steps to the result of substituting the argument of the application into the body of the function as described above.

The substitution operation, written \([e_2/x] e_1\) says to substitute expression \( e_2 \) for all occurrences of \( x \) inside of \( e_1 \). In our example above, we would write \([\lambda y. \lambda z. z/x] x\). The result of this operation is the lambda expression \( \lambda y. \lambda z. z \).

Function application is the sole evaluation rule in the lambda calculus. However, the order in which we apply this rule is important. Consider the following complex lambda expression:

\[
(\lambda f. \lambda x. f x) (\lambda y. (\lambda q. q) (\lambda v. v))
\]

This lambda expression features three possible function applications we might choose to evaluate:

(a) The top-level application (where the function being applied is \( (\lambda f. \lambda x. f x) \)).

(b) The body of the left-hand lambda (where the function is \( f \)).

(c) The body of the right-hand lambda (where the function is \( \lambda q. q \)).

Of these, the first and third and valid applications to choose to evaluate next. While the second is indeed a function application, the left-hand side of the application is a variable, not a lambda, so we cannot evaluate it (the rule for function evaluation does not syntactically apply).

The evaluation strategy that most closely corresponds to our intuition of how programs operate is called call-by-value evaluation. We apply our evaluation rule for function application as follows:

- We only evaluate the outermost function application not contained within a lambda.
- We only evaluate this outermost function application after evaluating its argument to a value—an expression that no longer evaluates further according to our strategy.

With the example above, we note that the argument to the top-level application \( \lambda y. (\lambda q. q) (\lambda v. v) \) no longer takes any steps of evaluation—we do not evaluate the application \( (\lambda q. q) (\lambda v. v) \) because it is found inside a lambda. Therefore, we take one step of evaluation to evaluate the expression to:

\[
(\lambda f. \lambda x. f x) (\lambda y. (\lambda q. q) (\lambda v. v)) \rightarrow (\lambda y. (\lambda q. q) (\lambda v. v)) x
\]

The application of \( (\lambda q. q) (\lambda v. v) \) is still inside of a lambda, so we don’t evaluate it. This expression has no more steps of evaluation it can take under our evaluation, so it is considered a value.
Church Encodings

The lambda calculus only contains functions and function application. Nevertheless, we can encode any language feature we wish using these constructs. Here, we demonstrate these encodings as evidence that the lambda calculus is as powerful as any other programming language we can think of.

Functions of Multiple Arguments

The first thing you may have noticed with the syntax of the lambda calculus that functions can only take a single argument. We could extend the syntax of lambdas to take multiple arguments, but this is unnecessary; we can simply nest lambdas to get the same effect!

\[ \lambda f. \lambda x. f x \]

Here is a function that takes two arguments called \( f \) and \( x \). Applying this function to a single argument yields:

\[ (\lambda f. \lambda x. f x) (\lambda y. y) \rightarrow \lambda x. (\lambda y. y) x. \]

Note that the result is a function that is expecting a single argument. We can then apply a second argument to get our final result:

\[ (\lambda x. (\lambda y. y) x) (\lambda z. z) \rightarrow (\lambda y. y) (\lambda z. z) \rightarrow \lambda z. z. \]

Rather than applying one argument at a time, we can supply the original function with two arguments. The resulting expression evaluates exactly as above:

\[ (\lambda f. \lambda x. f x) (\lambda y. y) (\lambda z. z) \rightarrow (\lambda x. (\lambda y. y) x) (\lambda z. z) \rightarrow (\lambda y. y) (\lambda z. z) \rightarrow \lambda z. z. \]

We call the process of realizing a multi-argument function using nested higher-order functions currying. Curried functions can be partially applied, that is, only some of its arguments can be supplied, leaving a function expecting the remainder of the arguments.

Booleans

Booleans consist of two values true and false as well as a conditional expression that uses a boolean to perform different behavior based on which boolean value is present. For example, if we extended our lambda calculus with explicit boolean constructs, we expect that:

\[ \text{if true then } \lambda x. x \text{ else } \lambda x. \lambda y. x y \rightarrow \lambda x. x. \]
We can encode these three constructs as follows:

\[
\begin{align*}
\text{true} & \equiv \lambda t. \lambda f. t \\
\text{false} & \equiv \lambda t. \lambda f. f \\
\text{if } \_ \text{ then } \_ \text{ else } \_ & \equiv \lambda b. \lambda t. \lambda f. b \, t \, f
\end{align*}
\]

Intuitively, true and false encode two possibilities \( t \) and \( f \) as function arguments and selects the first and second of those possibilities, respectively. The conditional expression simply uses this embedded “selection” mechanism to choose between its two branches. To demonstrate this, we can expand the conditional expression above using our encodings and verify that it produces the desired output:

\[
\text{if true then } (\lambda x. x) \text{ else } (\lambda x. \lambda y. x \, y)
\]

\[
\equiv (\lambda b. \lambda t. \lambda f. b \, t \, f) \, (\lambda t. \lambda f. t) \, (\lambda x. x) \, (\lambda x. \lambda y. x \, y)
\]

\[
\rightarrow (\lambda t. \lambda f. (\lambda t. \lambda f. t) \, t \, f) \, (\lambda x. x) \, (\lambda x. \lambda y. x \, y)
\]

\[
\rightarrow (\lambda f. (\lambda t. \lambda f. t) \, (\lambda x. x) \, f) \, (\lambda x. \lambda y. x \, y)
\]

\[
\rightarrow (\lambda t. \lambda f. t) \, (\lambda x. x) \, (\lambda x. \lambda y. x \, y)
\]

\[
\rightarrow (\lambda f. \lambda x. x) \, (\lambda x. \lambda y. x \, y)
\]

\[
\rightarrow \lambda x. x
\]

All of our encodings in the lambda calculus take this form. We first identify for a given language feature the different syntactic forms introduced by that feature. We then give an encoding for each syntactic form using the syntax of the lambda calculus. The encodings are designed to emulate the behavior of the features in question in a particular way, emulating the “fold” operation over that particular data type. Such encodings are called **Church encodings**.

**Natural Numbers**

Natural numbers introduce an infinite amount of constants 0, 1, 2, . . . as well as various operations over these numbers such as addition and subtraction. First, we define the constants using the following schema:

\[
\begin{align*}
0 & \equiv \lambda s. \lambda z. z \\
1 & \equiv \lambda s. \lambda z. s \, z \\
2 & \equiv \lambda s. \lambda z. s \, (s \, z) \\
3 & \equiv \lambda s. \lambda z. s \, (s \, (s \, z))
\end{align*}
\]

A natural number is encoded as a two-argument function, the first is a “successor” function and the second is a base value. Zero is encoded as simply returning the base value. Non-zero values are encoded as applying the successor function to the base value, applying the successor function
to that result, and so forth, \( n \) times.

With this encoding, the addition function can be defined as:

\[
\text{plus} \ _\ _ \ \overset{\text{def}}{=} \lambda x. \lambda y. \lambda s. \lambda z. \ x \ s \ (y \ s \ z)
\]

The addition function takes two natural numbers as arguments, \( x \) and \( y \) and produces the natural number (of the form \( \lambda s. \lambda z. \ldots \)) where \( s \) is applied \( y \) times and then \( z \) times, starting with \( z \). It is worthwhile to convince yourself that this encoding works by stepping through the evaluation of \( \text{plus} \ 1 \ 1 \ \rightarrow 2 \).

The Recursion Theorem

In our discussion of the complexity classes \( \text{PSPACE} \) and \( L \), we appealed to recursive algorithms to show that languages were in these classes. However, we never discussed how recursion might be performed by a Turing machine. Indeed, if you think about it, the problem is non-trivial: how does a Turing machine invoke itself recursively? Is such an operation valid?

It turns out that we did nothing wrong as the Recursion theorem tells us that such an operation is possible (Sipser 6.3):

**Theorem 1 (Recursion theorem).** Let \( T \) be a Turing machine that computes a function \( t : \Sigma^* \times \Sigma^* \rightarrow \Sigma^* \). There is a Turing machine \( R \) that computes a function \( r : \Sigma^* \rightarrow \Sigma^* \) where for every \( w \):

\[
r(w) = t(\langle R \rangle, w).
\]

Intuitively, \( T \) is a function that takes two arguments—a Turing machine and a string. The Recursion theorem says that we can create a Turing machine \( R \) that behaves like \( T \) but “patches in” the definition of \( R \) into the first argument for \( T \). The proof of this theorem requires that we are first able to construct a Turing machine that can output its own description. We can use this process to create the desired Turing machine \( R \).

The process of creating such a Turing machine seems rather arbitrary and it isn’t entirely clear that we aren’t “cheating” during the process in some way. A more convincing and enlightening proof of the Recursion theorem comes by studying the lambda calculus because it’s evaluation model is very simple—function application—and it more closely resembles a traditional (functional) programming language.

First, we note that recursion in the lambda calculus appears to be difficult for the simple reason that we aren’t able to “name” a lambda. However, we can overcome this difficulty in a similar manner to a Turing machine—a recursive function gains an additional argument a function invokable as the “recursive call”. After this function is “patched up”, it receives itself as that argument.

To demonstrate how to achieve this behavior, we first note a curious lambda expression, commonly called \( \omega \):

\[
\omega \overset{\text{def}}{=} (\lambda x. \ x \ x) \ (\lambda x. \ x \ x)
\]

If we evaluate \( \omega \) we obtain something surprising:

\[
(\lambda x. \ x \ x) \ (\lambda x. \ x \ x) \rightarrow [(\lambda x. \ x \ x)/x] \ x \ x = (\lambda x. \ x \ x) \ (\lambda x. \ x \ x)
\]
Evaluating \( \omega \) any number of steps results in \( \omega \) again! \( \omega \) is an example of a non-terminating program in the lambda calculus, a program that can always take a step of evaluation. The way that \( \omega \) achieves this is by simply outputting itself—this is precisely the Turing machine that can output its own description described in the proof sketch of the Recursion theorem above!

We can generalize \( \omega \) into an lambda calculus expression that explicitly performs the construction of the Recursion theorem. Recall that the recursion theorem takes a Turing machine \( T \) of two arguments—another Turing machine and a string—and produces a Turing machine \( R \) of one argument that behaves like \( T \) except that it patches in the first argument with its own description. The following function:

\[
\text{fix}_\_ \equiv \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))
\]

Takes as input a function of the form \( \lambda f. \lambda x. \ldots \) where \( f \) is the recursive function and \( x \) is the function’s actual argument. It then produces a patched function that only expects the actual argument and the recursive function invocations call itself. This function, the fixed-point combinator, allows us to perform recursion inside of the lambda calculus.

Let’s imagine running the fixed-point combinator on the following function:

\[
F \equiv \lambda f. \lambda x. f x
\]

which applies the recursive function directly to its argument. Note that this is equivalent to writing the following recursive Racket function:

\[
(\text{define} \ (\text{loop} \ x) \ (\text{loop} \ x))
\]

which produces an infinite loop when run. Nevertheless, let’s see how the fixed-point combinator deals with the program (throughout, we use the shorthand names for \( \text{fix} \) and \( F \) to stand in for their actual lambda terms to avoid clutter):

\[
\text{fix} \ F \rightarrow (\lambda x. F (\lambda y. x x y)) (\lambda x. F (\lambda y. x x y))
\]

\[
\rightarrow F (\lambda y. (\lambda x. F (\lambda y. x x y)) (\lambda x. F (\lambda y. x x y)) y)
\]

\[
\rightarrow \lambda x. (\lambda y. (\lambda x. F (\lambda y. x x y)) (\lambda x. F (\lambda y. x x y)) y) x
\]

Note that the body of the top-most lambda contains a copy of the result of evaluating \( \text{fix} \ F \) one step. Therefore, the final program is equivalent to:

\[
\lambda x. (\text{fix} \ F) x.
\]

Essentially, the fixed-point combinator “unrolls” its recursive definition one step. Providing the function with an argument begins the process of continuing the unroll the recursive definition a single step until the recursion bottoms out at its base case. If there is no base case, like with \( F \), then the function unrolls continuously, producing an infinite loop.

With \( \text{fix} \), we can easily prove the Recursion theorem specialized to lambda calculus expressions:
**Theorem 2 (Recursion Theorem).** Let $f$ be a lambda calculus function that takes two arguments—a function and a value. Then there exists a lambda calculus function $F$ such that for all values $x$:

$$(F f) x \equiv (f f) x$$

**Proof.** Our construction of fix fulfills this property for $F$ as demonstrated above. 

