The λ-Calculus

1 Background on Computability

The history of computability stretches back a long ways, but we'll start here with German mathematician **David Hilbert** in the 1920s. Hilbert proposed a grand challenge for mathematics called *Hilbert's Program* (program in the sense of agenda, not computer program—computer programs didn't exist yet). A large part of that challenge was to create an algorithm that, given any mathematical theorem, could in finite time determine whether that theorem is true or false. This is called the *Entscheidungsproblem*, or "decision problem". It was assumed at the time that naturally such an algorithm existed, it was just a question of finding it. However, this goal turned out to be provably impossible. The three people most responsible for killing this dream are **Kurt Gödel** with his Incompleteness Theorems, **Alan Turing** with his Turing Machines, and **Alonzo Church** with his λ-calculus. We'll focus here on Turing and Church.

1.1 Alan Turing

Alan Turing was a fellow at King's College when he became interested in the decision problem. He realized that to formally address this question, we needed a formal definition of what *algorithm* meant. Heretofore people had a vague, informal notion that an algorithm was just a series of mechanical steps, like a recipe—but this notion wasn't sufficient to address the decision problem. Turing invented what later came to be known as *Turing Machines* (he didn't call them that himself) and defined an algorithm as anything that can be computed by a Turing Machine. Note that there is no way to *prove* that algorithms and Turing Machines coincide, he just made the assertion that they do. However, it turns out to be a very robust definition that is generally accepted by everyone today. Given this definition, he then proved that there exist problems that cannot be solved by any Turing Machine, e.g., the famous *Halting Problem*. By definition, then, there cannot exist any algorithm to solve those problems; these problems are called *undecidable*. It turns out that Hilbert's decision problem belongs in this class, and thus Hilbert's Program was doomed. Turing published this result in 1936.

Historical sidenote: Turing was a prolific researcher and became involved in a number of Computer Science-related problems. Many people have heard of the *Turing Test* for Artifical Intelligence, which he invented. Turing was also a significant contributor to the World War II war effort in Bletchley Park, the nerve center of the Allies' efforts to break the German secret codes. He was also gay, and despite his many contributions the British government's intolerance of his lifestyle led to his early and tragic death.

1.2 Alonzo Church

Alonzo Church was a professor at Princeton around the same time that Turing was a fellow at King's College. He was not originally interested in Hilbert's decision problem; his goal was to redefine the very foundations of mathematics. He wanted to replace *sets*, the fundamental building blocks of all mathematics, with *functions* instead. Unfortunately for him, his students later proved that his system was inconsistent and thus useless. However, he salvaged part of his system from the wreckage: the λ-calculus. Church then used the λ-calculus to address Hilbert's decision problem.

Church *also* gave a formal definition of "algorithm", in terms of the λ-calculus, and proved that there exist unsolvable problems and hence Hilbert's decision problem was impossible. He actually published his results one month before Turing, but most people only remember Turing's paper (history is full of such injustices). Turing later became Church's graduate student at Princeton. Church and Turing together proved that the λ-calculus and Turing Machines are, in fact, computationally equivalent: anything that one can compute, so can the other. Thus, whether we use Turing Machines or the λ-calculus is entirely a matter of taste and convenience. In complexity theory we usually use Turing Machines; in programming language theory we usually use the λ-calculus. The λ-calculus is also at the heart of all functional programming languages.

2 Programming Paradigms and Functional Programming

The λ-calculus forms the core of a programming paradigm called *functional programming*. Programming paradigms (*procedural programming*, *object-oriented programming*, *logic programming*, *functional programming*, etc) are different philosophies about how

to abstract computation for the programmer. Programming languages exist solely to abstract computation—to present the programmer with high-level abstractions so that they can think about problems and express solutions in terms of those abstractions, rather than than forcing them to get into the low-level details of how the underlying machine should operate (e.g., by programming in assembly language). The differences between these kinds of languages do not lie in what the programmer can *do*, because they are all equivalent in computational power. The differences lie in how the programmer *expresses* computation.

As an example, consider object-oriented programming. The dominant abstraction in an object-oriented programming language (e.g., C++ and Java) is an *object*, which encapsulates a set of data and methods. The programmer is encouraged to organize their solution by thinking about how to break the problem down into distinct kinds of objects arranged in a class hierarchy, based on how these different kinds of objects relate to each other (*is-a*, *has-a*, etc). Roughly, we can think of object-oriented programming as encouraging the programmer to think in terms of *nouns* and the *relations* between nouns. For some problems this is a very convenient way to think, and classes and textbooks teaching object-oriented programming tend to focus on problems where this is true. However, not all problems conveniently break down into a static class hierarchy, and for these problems programming in an object-oriented language can be awkward and difficult.

Functional programming, in contrast, uses *functions* as the central abstraction. When we say *functions* we mean mathematical functions (a mapping from some domain to some co-domain), not the so-called "functions" of languages like C, which are more properly called *procedures*. Mathematical functions have the property that a function called with the same arguments will always yield the same result, which is sometimes called *purity* or *referential transparency* in the programming language community. Contrast this with a C-style "function", for which it is easy to come up with examples where this property does not hold. Referential transparency makes reasoning about and testing code much easier than in languages without it, and makes parallel code almost trivial get correct.

In functional programming the programmer is encouraged to organize their solution by thinking about how to break the problem down into functions and compositions of functions (given functions f and g, the composition $f \circ g$ is a function that applies g to an input and then applies f to g's result in order to get the final result). Functions allow us to decompose a problem into smaller problems and then glue the respective solutions together (higher-order functions, discussed below, are a critical part of this ability). Roughly, we can think of functional programming as encouraging the programmer to think in terms of *verbs* and the *composition* of verbs. As with object-oriented programming, for some problems this is a very convenient abstraction and for other problems it is not.

3 Some Intuition for the λ-Calculus

The λ-calculus is all about functions. Consider the function $f(x) = x^2 + 2x + 1$. To determine the value of $f(3)$, we *substitute* the argument 3 for the variable x in the body of the function, then *reduce* until we reach the final answer:

$$\begin{aligned} f(3) &= 3^2 + 2 \cdot 3 + 1 \\ &= 9 + 2 \cdot 3 + 1 \\ &= 9 + 6 + 1 \\ &= 15 + 1 \\ &= 16 \end{aligned}$$

The function has a name, f, but that name is actually superfluous; we could have called it g, *bob*, or anything else we wanted to and it would still be the same function. We can simplify the function definition by not giving it a name at all: $x \mapsto x^2 + 2x + 1$, i.e., the function with parameter x that maps to the function body $x^2 + 2x + 1$. This is called an *anonymous function*, i.e., a function with no name. Of course, that raises the question of how to call the function if it doesn't have a name. The syntax for calling anonymous functions is as follows:

$$\begin{aligned} (x \mapsto x^2 + 2x + 1)\ 3 &= 3^2 + 2 \cdot 3 + 1 \\ &= 9 + 2 \cdot 3 + 1 \\ &= 9 + 6 + 1 \\ &= 15 + 1 \\ &= 16 \end{aligned}$$

We simply juxtapose the function we're calling with its argument. In the previous examples the arguments were all numbers, but that doesn't have to be the case—arguments could also be other functions. Consider the integral or differential functions in calculus: they each take a function as an argument and return a new function as a result (the integral or differential of the argument, respectively). Functions that take other functions as arguments and/or return functions as results are called *higher-order functions*.

3.1 Environments

When evaluating a function, the act of substituting the argument for the parameter in the body of the function (e.g., substituting 3 for x in the previous examples) can be very expensive. Evaluating a function can involve multiple substitutions, and each one requires that the entire function body be iterated over to replace variables with values. Thus, using substitution makes function evaluation have $O(n^2)$ to $O(n^4)$ complexity (depending on evaluation order). We can use a simple trick to make the complexity $O(n)$ instead, using something called an environment. It turns out that substitution also requires some complicated rules to deal with nested functions and that environments handle nested functions naturally without any extra complexity, so they also simplify evaluation in that way.

An *environment* is a map from variables to values. When we call a function, rather than substituting the argument for the parameter inside the function body, instead we add an entry into the environment mapping the parameter to the argument. When evaluating the function body, whenever we see a variable we look it up in the environment to get its corresponding value. For example:

Function evaluation	Environment
$(x \mapsto x^2 + 2x + 1)\ 3 = x^2 + 2x + 1$	$[x \mapsto 3]$
$= 9 + 2x + 1$	$[x \mapsto 3]$
$= 9 + 6 + 1$	$[x \mapsto 3]$
$= 15 + 1$	$[x \mapsto 3]$
$= 16$	$[x \mapsto 3]$

4 Syntax of λ-Calculus

The λ-calculus doesn't actually contain numbers and arithmetic; it only has functions, nothing else. The syntax of λ-calculus expressions is:

$$
\begin{array}{rlr}
e \in Exp ::= & x & \text{variables} \\
| & \lambda x \,.\, e & \text{function abstraction} \\
| & e_1\ e_2 & \text{function application}
\end{array}
$$

Note that we use $\lambda x \,.\, e$ to define functions instead of $x \mapsto e$; they mean the same thing. Why do we use λ? The (possibly apocryphal) story goes that Church wanted to use the notation $\hat{x} \,.\, e$ taken from Russell and Whitehead's famous book *Principia Mathematica*, but the typesetter couldn't reproduce the ˆ symbol and so used λ instead.

4.1 Example Expressions

- $\lambda f \,.\, \lambda x \,.\, x$
- $\lambda f \,.\, \lambda x \,.\, f\ x$
- $\lambda f \,.\, \lambda x \,.\, f\ (f\ x)$
- $\lambda u \,.\, \lambda f \,.\, \lambda x \,.\, f\ ((u\ f)\ x)$

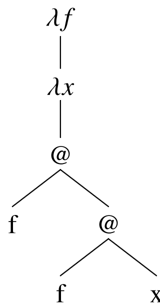
Often we'll use the shorthand notation $\lambda xy \,.\, e$ to stand for $\lambda x \,.\, \lambda y \,.\, e$; using that convention the expressions would look like this:

- $\lambda fx \,.\, x$
- $\lambda fx \,.\, f\ x$
- $\lambda fx \,.\, f\ (f\ x)$
- $\lambda ufx \,.\, f\ ((u\ f)\ x)$

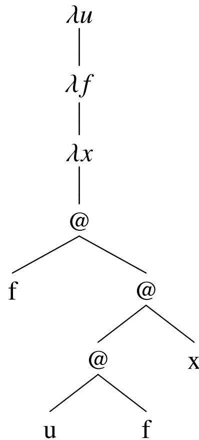
4.2 Abstract Syntax Tree (AST)

Expressions in the λ-calculus can get rather complex and difficult to read. It is often beneficial to rewrite expressions into the form of *abstract syntax trees*. These trees reveal the underlying structure of the expressions and make them easier to read. Each subtree of the AST stands for a subexpression of the entire expression; each node is either a λ (if the subexpression rooted at that node is a function definition), an '@' (if the subexpression rooted at that node is a function application; we use '@' because function application is written using juxtaposition instead of using a special symbol), or a variable (which will always be a leaf node). Here are some examples:

4.2.1 AST for $\lambda f x \,.\, f\ (f\ x)$



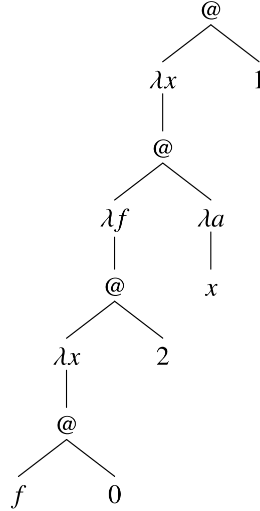
4.2.2 AST for $\lambda u f x \,.\, f\ ((u\ f)\ x)$



5 Semantics of λ-Calculus

The semantics tells us how to evaluate an expression to a value. For the λ-calculus the only things we have are functions, so expressions (which are made up of function definitions and function calls) will all evaluate to functions as their final values. We evaluate a λ-calculus expression exactly the way we did when we talked about evaluating functions using environments. In fact, λ-calculus expressions *are* mathematical functions, just using different syntax. There is one thing that we need to be careful of, though, that comes from the fact that we're dealing with *higher-order* functions using environments. The value of a function definition is not just the function itself, but also the environment that existed when the function was defined. These two things together, the function and its environment, is called a *closure*. Consider the following expression:

$$(\lambda x \,.\, (\lambda f \,.\, (\lambda x \,.\, f\ 0)\ 2)\ \lambda a \,.\, x)\ 1$$



Each time we call a function (the '@' nodes in the AST) we push a new mapping onto the environment: first $x \mapsto 1$, then $f \mapsto \lambda a\,.\,x$, and finally $x \mapsto 2$. Now consider what happens when we evaluate the function call $f\ 0$. We know that f is the function $\lambda a\,.\,x$, i.e., the function that returns the value of x no matter what argument we pass in. But what is the value of x? It should be 1, the value of x when we defined the function $\lambda a\,.\,x$, but the current environment when we evaluate the function call maps x to 2 instead. The only way to remember what the value of x was when we defined the function is to save the environment along with the function, i.e., a closure. Thus the environment should have $f \mapsto (\lambda a\,.\,x, [x \mapsto 1])$ instead of just $f \mapsto \lambda a\,.\,x$. Then when we evaluate the call $f\ 0$, we look up f in the environment to get the closure and evaluate the closure's function using the closure's environment, *not* the current environment.

6 Encoding Higher-Level Abstractions

The λ-calculus is elegant in its simplicity, but it isn't very convenient for actual programming. We can get around this by showing how to encode higher-level abstractions using the λ-calculus (e.g., numbers, arithmetic, etc) and then building them directly into the language as a convenience. We'll end up with the following extended version of the λ-calculus:

$$x \in \mathit{Variable} \qquad n \in \mathbb{N} \qquad b \in \mathit{Bool}$$

$$\begin{aligned} e \in \mathit{Exp} ::= {} & n \mid b \mid x \mid \lambda x\,.\,e \mid e_1\ e_2 \\ & \mid (e_1, e_2) \mid \textbf{fst}(e) \mid \textbf{snd}(e) \mid \textbf{inl}(e) \mid \textbf{inr}(e) \\ & \mid \textbf{case}\ e_1\ \textbf{of inl}\ x \Rightarrow e_2\ ,\ \textbf{inr}\ x \Rightarrow e_3 \end{aligned}$$

Where (e_1, e_2) builds a pair of values; $\textbf{fst}(e)$ takes a pair and extracts the first value; $\textbf{snd}(e)$ takes a pair and extracts the second value; $\textbf{inl}(e)$ injects a value into the left side of a union; $\textbf{inr}(e)$ injects a value into the right side of a union; and $\textbf{case}\ e_1\ \textbf{of inl}\ x \Rightarrow e_2\ ,\ \textbf{inr}\ x \Rightarrow e_3$ pattern-matches on e_1 (which should be a union), evaluating e_2 if the union is on the left side and e_3 if it's on the right side. We'll also assume that the language has builtin functions on numbers and booleans $+, -, \times, \div, =, <, \wedge, \vee, \neg$.

6.1 Arithmetic

We can encode the natural numbers using λ-calculus in a number of ways, but here is a very simple way that lends itself to defining arithmetic:

$$\begin{aligned} 0 \in \mathbb{N} &\equiv \lambda sz\,.\,z \\ n \in \mathbb{N} &\equiv \lambda sz\,.\,s^n\ z \end{aligned}$$

Where $s^n\ z$ means to apply function s to z a total of n times, e.g., $s^3\ z = s\ (s\ (s\ z))$. In this encoding, 0 is a function that takes two arguments and returns the second one, and any natural number n is the function that takes two arguments and applies the first one to the second one n times. We can then define arithmetic on natural numbers as follows:

$$\textbf{succ} \equiv \lambda nsz\,.\,s\ (n\ s\ z)$$
$$\textbf{add} \equiv \lambda mn\,.\,m\ \textbf{succ}\ n$$
$$\textbf{mul} \equiv \lambda mn\,.\,m\ (\textbf{add}\ n)\ 0$$

The successor function **succ** takes a number and adds 1 to it. The addition function **add** takes two numbers and returns the sum. The multiplication function **mul** takes two numbers and returns the product. These definitions work as we would expect arithmetic to work because of the way that we have encoded numbers as functions. We could encode subtraction, division, integers, and more in a similar way.

Historical sidenote: The successor, addition, and multiplication functions were simple to encode, but not all encodings are as obvious. Church and his students spent a long time trying to figure out how to encode the predecessor function (the function that substracts 1 from a number), which is needed to define subtraction and division. Finally, Church's student Kleene went to the dentist for an operation, and as he was being anesthetized with laughing gas he had a flash of inspiration and figured out the solution.

6.2 Booleans

Booleans are simple to encode:

$$\textbf{true} \equiv \lambda tf\,.\,t$$
$$\textbf{false} \equiv \lambda tf\,.\,f$$

The value **true** is a function that takes two arguments and returns the first one; the value **false** is a function that takes two arguments and returns the second one. We can use these encodings to define the standard boolean operators:

$$\textbf{and} \equiv \lambda ab\,.\,a\ b\ \textbf{false}$$
$$\textbf{or} \equiv \lambda ab\,.\,a\ \textbf{true}\ b$$
$$\textbf{not} \equiv \lambda a\,.\,a\ \textbf{false}\ \textbf{true}$$

6.3 Pairs

So far we've encoded primitive values (numbers and booleans) and operations on those primitive values. Now we'll show how to encode data structures, specifically pairs of values:

$$\textbf{pair} \equiv \lambda xyb\,.\,b\ x\ y$$
$$\textbf{fst} \equiv \lambda p\,.\,p\ \textbf{true}$$
$$\textbf{snd} \equiv \lambda p\,.\,p\ \textbf{false}$$

The pair constructor is a nested function definition that takes three arguments. Giving it two arguments (the values to store in the pair) results in a function that takes a single argument and applies it to the original two arguments (i.e., x and y). The **fst** function takes a pair as an argument and applies it to the function **true**, while the **snd** function does the same except applies it to **false**. Recall that **true** is a function that takes two arguments and returns the first one, while **false** is a function that takes two arguments and returns the second. Thus, **fst** will return the first value given to **pair** while **snd** will return the second value given to **pair**.

6.4 Unions

The second data structure we'll encode is unions of two values. A union is a value that is tagged as being either a *left* value or a *right* value; what "left" and "right" mean is up to how the programmer interprets them. For example, we could return a union value as the result of a function where a "left" value means the function had an error and the value is an error message, while a "right" value means the function operated correctly and the value is the function's result. We create a "left" value using the **inl** (inject left) function, and a "right" value using the **inr** (inject right) function. Think of **inl** and **inr** as creating a pair where the first element is **true** or **false** (meaning left or right) and the second element is the actual value:

$$\textbf{inl}(e) \equiv (\textbf{true}, e)$$
$$\textbf{inr}(e) \equiv (\textbf{false}, e)$$

We can pattern-match on unions to determine whether they are “left” or “right” values based on how they are tagged; the **case** statement takes a union value and two cases: one is an expression to evaluate if the union is a “left” value, the other is an expression to evaluate if the union is a “right” value:

$$\textbf{case}\ e_1\ \textbf{of inl}\ x \Rightarrow e_2\ ,\ \textbf{inr}\ x \Rightarrow e_3\ \equiv\ \textbf{fst}(e_1)\ ((\lambda x\,.\,e_2)\ \textbf{snd}(e))\ ((\lambda x\,.\,e_3)\ \textbf{snd}(e))$$

$\textbf{fst}(e_1)$ extracts the first element of the union pair, which will be either **true** or **false** (recall that they are functions that take two arguments and return either the first or second one, respectively). The two arguments we pass are the cases for left and right. The first one calls $\lambda x\,.\,e_2$ passing in the second element of the pair (the actual value of the union), and the second one does the same except it calls $\lambda x\,.\,e_3$. So if the union is a “left” value then **case** will return the result of evaluating e_2 on the union’s value, and if the union is a “right” value then **case** will return the result of evaluating e_3 on the union’s value

6.5 Example Expressions

- $(\lambda x\,.\,x + x)\quad ((\lambda y\,.\,y + 1)\quad 6)$
- $((\lambda f g x\,.\,g\ (f\ x))\quad (\lambda y\,.\,y + 1)\quad (\lambda z\,.\,z * z))\quad 2$
- $\textbf{snd}((\lambda x\,.\,(\textbf{snd}(x), \textbf{fst}(x)))\quad (1, 2))$
- $(\lambda z\,.\,\textbf{case}\ z\ \textbf{of inl}\ x \Rightarrow x + 1\ ,\ \textbf{inr}\ x \Rightarrow (\textbf{fst}(x) + 1, \textbf{snd}(x) + 1))\quad \textbf{inr}((\lambda w\,.\,(w, w))\quad 1)$