Turing Machines and λ-Calculus

Emmett Neyman

May 2018

About this Paper

This semester in CIS 670, we focused on improving our writing, reading and revising everything from technical papers in computer science to 19th century English literature. For my final project, I will be continuing that focus on better writing via a summary of the paper *Encoding Turing Machines into the Deterministic λ-Calculus* by Ugo Dal Lago and Beniamino Accattoli. I chose this paper because it combined topics from two different courses I took this semester: Turing machines from CIS 511 and λ-calculus from this course. I want to introduce the major concepts from this paper at a level approachable by an average undergraduate student studying computer science. I will assume only a basic knowledge of Turing machines and no knowledge of any programming language concepts such as λ-calculus. While the primary focus of this work is the writing and its accessibility, the topics are also relevant to the material covered in the course.

While I will assume readers have no background with λ-calculus, this piece is not meant to substitute a proper introduction to λ-calculus or any associated topics. I will only be covering the material necessary for the concepts covered in Dal Lago and Accattoli's paper. Additionally, I won't cover the entirety of the paper, instead focusing on the major ideas and explaining them intuitively.

Introduction

One of the first topics taught in a theoretical computer science class is the concept of a Turing machine. Each semester, new students are introduced to this abstract model of computation with an infinite tape, an alphabet of symbols, a set of discrete states, and an arbitrarily complex transition function. They're told, often without any sort of proof, that Turing machines are a *universal* model of computation; anything their laptop can do, anything a supercomputer can do, a Turing machine can do as well. This introduction is usually followed by definitions of decidability, recognizability, and reductions, beginning a dizzying foray into computation theory. Later on in a student's journey through college, they may or may not be introduced to an equivalently powerful model of computation: the λ-calculus.

There are a few reasons Turing machines are more prevalent in undergrad education than the λ-calculus, but those reasons are not the focus of this paper. Instead, I'll use this paper to introduce some of the key concepts of both Turing machines and λ-calculus, culminating in the main idea developed by Ugo Dal Lago and Beniamino Accattoli in their paper *Encoding Turing Machines into the Deterministic λ-Calculus*. In their paper, the authors describe a method of encoding Turing machines and their computation into λ-calculus, providing an intuitive and concrete proof that the λ-calculus is just as powerful as Turing machines. They also give a complete proof that there encoding is correct, showing exactly how the λ-terms generated by the encoding correctly mimic the behavior of the Turing machine. Before diving into the paper however, I will give a short recap of Turing machines and a slightly longer introduction to λ-calculus.

Turing Machines

At the University of Pennsylvania, most undergraduates are introduced to Turing machines during their first two years of college. However, before they hear a formal definition, many students hear the term "Turing machine" being casually tossed around by professors, sometimes as early as their first computer science class. Therefore, I feel it's necessary to formally define Turing machines and some related ideas before forging forward. Informally, a Turing machine (TM) is an abstract model of computation that consists of a tape that can be read from and written to. Depending on what the machine reads and its current state, it writes a new symbol onto the tape, changes its state, and then moves its head either right or left so that it can read a new cell of the tape. Formally, a TM is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where

- Q is a finite set containing all the states of the TM. Every time the machine reads a new symbol, it can change its state depending on its current state and the symbol it read.
- Σ is the input alphabet of the TM. In other words, Σ is a set of symbols that can start on the tape before the TM begins its computation. It's important to note that the blank character $\llcorner\!\lrcorner$is not in Σ.
- Γ is the tape alphabet of the TM. Usually, $\Gamma = \Sigma \cup \{\llcorner\!\lrcorner\}$ but sometimes it's useful to have symbols that the TM can write on the tape but that don't appear in the input. Formally, all that's required of Γ is that $\Sigma \subset \Gamma$ and $\llcorner\!\lrcorner \in \Gamma$.
- δ is the transition function of the TM. δ is a function from $G \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$. That is, given a current state and a just-read symbol from the tape alphabet, δ determines what the TM's next state should be, what symbol it should write onto the tape, and what direction its head should move.
- q_0 is the start state of the TM.
- q_a is the accept state of the TM. If at any time during its computation, a TM enters the state q_a it immediately halts and accepts its input.
- q_r is the reject state of the TM. If at any time during its computation, a TM enters the state q_r it immediately halts and rejects its input.

When discussing Turing machines, we often want a way to take "snapshot" of the machine during its computation on some input. To do this, we define the *configuration* of a TM. For a state q, and two strings u and v such that $u, v \in \Gamma^*$, the configuration uqv represents a TM in state q with uv on its tape and its head on the first character of v. We see that the string uqv gives a complete picture of the TM during a specific moment of its computation; we know what's on the tape, what state its in, and where its head is. We say configuration c_i *yields* configuration c_j if after one step, the TM transitions from c_i to c_j via its transition function δ.

The last concepts we'll introduce regarding Turing machines are the notions of *recognizability* and *decidability*. Recall that a language L over an alphabet Σ is simply a set of strings such that every string is a member of Σ^*. Some languages are regular, meaning that a string's membership in that language can be checked with a finite automaton, a very weak model of computation. For example, all finite languages are regular and so is the language `0`*`1`*. From the definition of a Turing machine we have an intuitive way to describe the language of a Turing machine, denoted $L(\mathcal{M})$ for the TM $\mathcal{M}$. We say that the language of a TM is just the set of all strings it accepts. We say a language L is *Turing-recognizable* if there exists a TM $\mathcal{M}$ such that $L(\mathcal{M}) = L$. If this is the case, we say $\mathcal{M}$ *recognizes* L. There are many equivalent terms used throughout the literature that all mean the same thing as Turing-recognizable: recursively enumerable and semi-decidable are two of the most common. Even though a language L may be recognized by a TM $\mathcal{M}$, we don't know anything about how $\mathcal{M}$ acts on an input w when $w \notin L$. $\mathcal{M}$ may halt and reject or it may loop forever, never giving an answer. We say that a TM $\mathcal{M}$ *decides* a language L if $\mathcal{M}$ recognizes L and, on all inputs that are not in L, $\mathcal{M}$ halts and rejects. If there is a TM that decides L, we say L is *Turing-decidable* or just *decidable* (the term *recursive* is equivalent).

The λ-Calculus

An equivalent, though much less taught, model of computation is the λ-calculus, a formal system that sprouted from work in combinatory logic in the 1920s and 1930s. Invented by Alonzo Church, the λ-calculus was created to be a new foundation of mathematics, an alternative to the systems developed by Bertrand Russel and Ernst Zermelo. It was quickly noticed though, that Church's system was just as powerful computationally as many other proposed models of the time, most notably Turing machines. Informally, the λ-calculus is a set of terms and a set of rules that can be applied to those terms to form new terms. There are two operations in λ-calculus that will come up: application and abstraction. The power in the λ-calculus lies in its power to encode and compute functions. Although computing functions may seem like a benign power, by composing multiple functions with multiple arguments together, it turns out one can compute anything that can be computed.

Formally, λ-terms are defined inductively as follows

- If x is a variable from some global set V, then x is a λ-term.
- If M and N are both λ-terms then (MN) is a λ-term (application).
- If M is a λ-term and x is a variable from some global set V, then $(\lambda x.M)$ is a λ-term (abstraction).

We write $M \equiv N$ if M and N are the same λ-term up to renaming variables. The convention is to write variables as lowercase letters such as x, y, z and to write λ-terms as capital letters such as L, M, N. It's also common convention to write $\lambda xyz...$ instead of $\lambda x.\lambda y.\lambda z....$ We will next define the notion of *free variables* of a λ-term M, denoted $FV(M)$. The opposite of a free variable is a *bound variable*. You've seen free and bound variables before even though you might not have realized. In calculus you saw the expression $\int y^3x^2dx$; in this expression y is a free variable and x is a bound variable. Again, we define this notion inductively.

- If $M = x$, then $FV(x) = \{x\}$.
- If $M = LN$, then $FV(M) = FV(L) \cup FV(N)$.
- If $M = \lambda x.N$, then $FV(M) = FV(N) \setminus \{x\}$.

From this definition, we see that if x is a bound variable, it must appear inside a λ. We call M a *closed term* iff $FV(M) = \emptyset$. The next thing we need to define is the idea of *substituting* a λ-term for a variable; this is a key component of the main rewrite rule of λ-calculus. We use the notation $M[x := N]$ to represent substituting N for all the free occurrences of x in some λ-term M. We define substitution formally as follows.

- $x[x := N] \equiv N$
- $y[x := N] \equiv y$ if $x \not\equiv y$
- $(M_1M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$
- $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$

Now is a good time to mention an important assumption we have been making. So far, we have assumed that if we have a collection of λ-terms $M_1, M_2, \ldots, M_k$, then the free variables and the bound variables in all these terms are different. This assumption allows us to avoid checking whether or not $y \equiv x$ and $y \in FV(N)$ when we say $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$. Now that we have defined what a λ-term is, what a free variable is, and what it means to substitute a λ-term for a free variable, we can finally define how the λ-calculus works. The main rewriting rule (axiom) of the λ-calculus, called the β-reduction rule is that

$$(\lambda x.M)N = M[x := N]$$

The symbol $\rightarrow_\beta$ is sometimes used instead of $=$ in the above rule. This rewriting rule can also be thought of as an operation, usually called λ-abstraction or just abstraction (more on this later). There are also other rules including the reflexivity, symmetry, and transitivity of equality of λ-terms and congruence/compatibility rules that allow us to substitute equal λ-terms into a larger λ-terms. For example, if $M_1 = M_2$, then $\lambda x.M_1 = \lambda x.M_2$. In some presentations of the λ-calculus, where the above variable naming rule is not assumed, there is an additional rewrite rule:

$$\lambda x.M = \lambda y.M[x := y] \text{ when } y \text{ does not occur free in } M$$

This rule is often called α-conversion. Next we'll define a few special λ-terms that are used so frequently, they've been given specific names.

- $\mathsf{I} \equiv \lambda x.x$
- $\mathsf{K} \equiv \lambda xy.x$

- $\mathsf{K}_* \equiv \lambda xy.y$
- $\mathsf{S} \equiv \lambda xyz.xz(yz)$

I is called the identity combinator since, given any λ-term M, $\mathsf{I}M = M$. K and K_* are combinators that when applied to two λ-terms, return the first and second term respectively. Lastly, S, given λ-terms L, M, N, returns $ML(NL)$. Already we're seeing the true power of the λ-calculus in its ability to compute arbitrary functions. However, the real power comes when we introduce *fixpoints*, combinators that allow us to represent recursive functions. There are many fixpoint combinators, but the most popular is the Y combinator (name sound familiar?). Formally, we let

$$\mathsf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

While this term may look confusing at first glance, consider what happens when we apply Y to a λ-term F. We see that

$$\begin{aligned}\mathsf{Y}F &= (\lambda x.F(xx))(\lambda x.F(xx))\\ &= F(\lambda x.F(xx))(\lambda x.F(xx))\\ \mathsf{Y}F &= F(\mathsf{Y}F)\end{aligned}$$

From the equation above, it's easy to see how $\mathsf{Y}F$ can be used to define recursive functions, a tool that gives the λ-calculus much of its computational power. So far we've demonstrated how we can use the λ-calculus to represent and compute functions but we haven't shown how these functions can compute on any useful operands. Next, we'll do just that, finishing our section on the λ-calculus before moving on to our main topic. The *Church numerals* are a set of λ-terms that are used to represent the natural numbers $\mathbb{N}$. Before defining the Church numerals, we introduce some new notation. Let F and M be any λ-terms. Then $F^n(M)$ is defined inductively such that $F^0(M) \equiv M$ and $F^{n+1}(M) \equiv F(F^n(M))$. Using this notation, we can define the nth Church numeral c_n to be $\lambda fx.f^n(x)$. Church numerals allow us to start using the λ-calculus almost like a real programming language. J.B. Rosser, one of Church's students discovered the following three λ-terms that "play nicely" with Church numerals.

- $\mathsf{A}_+ \equiv \lambda xypq.xp(ypq)$
- $\mathsf{A}_* \equiv \lambda xyz.x(yz)$
- $\mathsf{A}_{\exp} \equiv \lambda xy.yx$

As hinted at by the names, these λ-terms act like operators on Church numerals.

$$\mathsf{A}_+ c_i c_j = c_{i+j},\ \mathsf{A}_* c_i c_j = c_{i*j},\ \text{and}\ \mathsf{A}_{\exp} c_i c_j = c_{i^j}\ \text{when}\ j \neq 0$$

These λ-terms are complicated and it's a good exercise to work through each of these terms to achieve the equalities shown above.

With our new knowledge of the λ-calculus under our belts, we can now proceed with the exhibition of the paper *Encoding Turing Machines into the Deterministic λ-Calculus.* That is not to say however, that we have given a complete introduction to the λ-calculus. There is so much more to cover including reduction strategies, values/normal forms, and η-conversion. Not to mention all the typed variants of λ-calculus that have been developed. But for now, we have laid a strong enough framework in order to move on.

Deterministic λ-Calculus

Before diving into the meat of the paper, we first need to define a variant of the λ-calculs called the Deterministic λ-Calculus. While the word "variant" is technically correct, it's better to think of the deterministic λ calculus as a "fragment" of the entire formal system. Intuitively, this variant introduces the constraint that every λ-term must have at most one possible equal λ-term (using the notion of equality defined above by the β reduction). In other words, this means that if we have a term MN where M and N are both λ-terms, then N must either be an abstraction or a variable; it can't be further reduced via the β reduction rule. The rewrite rule (β reduction) is the same as we saw above. For terms t and s: the term $(\lambda x.t)s = t[x := s]$. For the rest of this paper, I will begin to use the symbol $\to_\beta$ instead of the $=$ we have been using so far and begin to use the word "reduce" instead of "equals".

The authors chose to encode Turing machines into this fragment of λ-calculus since it has the advantage that every term deterministically reduces to at most one other term. This is important because Turing machines are inherently deterministic machines. Another reason that the authors chose the deterministic λ-calculus was that they showed that simulating a Turing machine in this system takes a linear amount of reductions with respect to the number of transitions made by the Turing machine. The authors prove the following lemma in their paper: if t is a term in the deterministic λ-calculus, then there is at most one other term s such that $t \to_\beta s$, and in that case t is an application.

The last piece of the deterministic λ-calculus Dal Lago and Accattoli define in their paper is a new fixpoint operator. Above, we defined the **Y** combinator, discovered by Haskell Curry, as one such fixpoint operator in the λ-calculus. The authors chose to use another fixpoint combinator, Alan Turing's θ combinator. The combinator $\theta = tt$ where t is the term $\lambda xy.y(\lambda z.xxyz)$. Written out fully, we have

$$\theta \equiv (\lambda xy.y(\lambda z.xxyz))(\lambda xy.y(\lambda z.xxyz))$$

Like the **Y** combinator, we see that θs eventually reduces to $s(\theta s)$. For this reduction, we need the η-conversion rule which we didn't introduce above. For this reason, we won't go into the entire derivation showing that θs reduces to $s(\theta s)$. The authors use this fixpoint operator to encode the transition function of the Turing machine since it needs to be called repeatedly until the machine stops.

Encoding a Turing Machine

As we noted above, after encoding a machine into λ-calculus, the number of reductions needed to simulate the machine is linear with respect to the number of transitions made by the machine during its computation. The authors also show that the length of the encoding of some TM $\mathcal{M}$ on input w is linear with respect to the size of the input $|w|$. We consider the size of the alphabet $|\Sigma|$ and the number of states $|Q|$ to be constants. The authors break down the encoding into two main sections

1. Encoding the strings the machine works with
2. Encoding the machine itself (its states and transitions)

The majority of Dal Lago and Accattoli's paper deals with defining the exact encoding for each part of a Turing machine and then presenting the proofs of correctness for each part of the encoding. For the sake of brevity, clarity, and accessibility, I will omit these proofs from this work as well as omitting the exact encodings for each of the parts. Instead, I will give a high-level overview of their strategy and explain some of the decisions they make.

Encoding Strings

There are three main concerns that the authors put forward regarding their encoding of strings. The first is how to encode a string s with respect to a specific alphabet Σ. They note that their method of encoding a string is dependent on the alphabet of which that string's characters are from. So, two identical strings s and s' might have different encodings if s is encoded with respect to Σ and s' is encoded with respect to another alphabet Σ'. The second concern they address is the need for a λ-term that efficiently appends a character onto a string. When simulating a Turing machine, new symbols are continually being written to the tape. We can think of this action as continually appending a new character onto the string to the right or left of the tape head. To address this issue, they define a λ-term $\texttt{append}_\Sigma^a$ that takes a string and a character and appends them together. The last concern they raised was that with Turing machines, the input alphabet Σ does not equal the tape alphabet Γ. This presents a problem since our encoding of strings is alphabet dependent. To solve this problem, the authors present two λ-terms, $\texttt{lift}_\Sigma$ and $\texttt{flat}_\Gamma$, that can translate between a string encoded with respect to Σ and a string encoded with respect to Γ.

Encoding the Machine's Behavior

Encoding the behavior of a Turing machine can be broken down into four parts. The first is encoding a configuration of a Turing machine. The second is translating the input string into the initial configuration.

To do this, the authors introduce a λ-term $\texttt{init}^{\mathcal{M}}$ that takes an input string s and returns the initial configuration of the machine. The third part is actually simulating the transition function of the Turing machine. This is accomplished by defining a λ-term $\texttt{trans}^{\mathcal{M}}$ that basically encodes the transition function δ as a giant table. The final part of simulating the machine's behavior is to generate the output string from the encoding of the final configuration of the machine. This is done via the λ-term $\texttt{final}^{\mathcal{M}}$ which takes a configuration and returns a string in Σ^*.

Conclusion

Two of the most well known models of computation are the Turing machine and the λ-calculus. In this paper, I tried to give a solid background of both at a level accessible by an average undergraduate student studying computer science. While most students are introduced to Turing machines in an introductory computer science theory course, some students go the entirety of their undergraduate curriculum without being exposed to the λ-calculus. After introducing both models, I briefly summarized a recent paper by Ugo Dal Lago and Beniamino Accattoli in which they show how a Turing machine can be encoded into and simulated by a fragment of the λ-calculus called the deterministic λ-calculus. Not only is this topic a satisfying combination of Turing machines and the λ-calculus, it also gives an intuitive argument as to why the λ-calculus is just as powerful as a Turing machine.

References

History of Lambda-calculus and Combinatory Logic by Felice Cardone and J. Roger Hindley (2006)
Introduction to the Theory of Computation, Third Edition by Michael Sipser (2013)
Introduction to Lambda Calculus by Henk Barendregt and Erik Barendsen (2000)
Encoding Turing Machines into the Deterministic λ-Calculus by Ugo Dal Lago and Beniamino Accattoli (2017)