Chapter 1

Introduction

The flow of the document, what will be covered (UTLC, STLC, DTLC, Curry-Howard, LCCCs...). The aim of the document. Things will be formal but for key constructions that are deemed to be of practical use, a simple implementation will be given, usually in haskell, lean or similar.

Chapter 2

λ-Calculus

Add some introduction. Computation and meaning, extensional vs intensional, operational vs denotational. historical note

2.1 Untyped λ-Calculus

In order to get acquainted with the λ-calculus, let us develop a simple example to familiarize ourselves before we begin with a more formal approach to this discipline. Consider the function $f(x) = x+1$, the most straightforward way to express this using λ-calculus would be $(\lambda x.x + 1)$, where the lambda denotes that x is being captured and used as a parameter to perform some computation [1]. Evaluating $f(2)$ using this newly created lambda term would look like this: $(\lambda x.x + 1)(2) \rightsquigarrow_\beta (2 + 1) \rightsquigarrow_\beta 3$. This process of reducing a λ-expression is referred to as β-reduction, it will be covered formally later in the chapter.

Definition 2.1.1. The set of lambda terms $\mathbf{\Lambda}$ is defined inductively as follows:

- (Variable) If $x \in \mathbf{V}$, then $x \in \mathbf{\Lambda}$.
- (Abstraction) If $x \in \mathbf{V}$ and $M \in \mathbf{\Lambda}$, then $(\lambda x.M) \in \mathbf{\Lambda}$.
- (Application) If $M, N \in \mathbf{\Lambda}$, then $(MN) \in \mathbf{\Lambda}$.

Where $\mathbf{V} = \{x, y, z, ...\}$ represents a countably infinite set of variable names.

The key takeaway of this definition is that abstraction and application together, encapsulate the meaning of function in a way that when combined with β-reduction **??** allows us to perform computation.

[1] this is an e.g., $(\lambda x.x + 1)$ is not a valid lambda term, see Definition 2.1.2.

Since we are dealing with a formal language, it is in our benefit to introduce a few other objects with the aim of defining a grammar to generate the set $\mathbf{\Lambda}$:

- An alphabet $\Sigma = \{\lambda, ., (,), \ldots\}$, is a finite set of symbols
- A string is a finite sequence of elements from Σ, the empty string is denoted by ε
- Σ^* denotes the set of all finite strings over Σ, $\varepsilon \in \Sigma^*$
- A language L over an alphabet Σ is a subset of Σ^*

Our aim now, to generate the set $\mathbf{\Lambda}$, to do this, we will make use of a grammar. When dealing with grammars that define programming languages i.e. context-free grammars, Backus-Naur Form is the way to go:

- Nonterminals are enclosed in angle brackets (e.g. `<expr>`)
- Terminals are written literally (e.g. `"λ"`, `"."`, x)
- Productions define how nonterminals expand, written as `::=`
- The vertical bar `|` denotes available expansions

Thus, to define the language for natural numbers in decimal notation:

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<number> ::= <digit> | <digit> <number>
```

Definition 2.1.2. Taking advantage of BNF notation, an alternative definition for $\mathbf{\Lambda}$ (2.1.1):

```
<term> ::= <variable>
         | "λ" <variable> "." <term>
         | "(" <term> <term> ")"
<variable> :: ∈ V
```

where $\Sigma = \{\lambda, ., (,)\} \cup \mathbf{V}$ and $\mathbf{\Lambda} \subset \Sigma^*$. Through use of algebraic types [2], this grammar can be implemented in Haskell:

[2] Algebraic Data Types are inductively defined types built from sums (either this or that) of products (this and that). They let you define structured data with multiple forms and support safe, exhaustive pattern matching.

```haskell
data LambdaTerm = Var String
                | App LambdaTerm LambdaTerm
                | Abs String LambdaTerm
{- variable names are implemented using strings -}
```

Listing 2.1: Grammar implementation of the λ-calculus language using Haskell's algebraic types

Remark 1. From the definition above we extract that $\lambda x.x + 1 \notin \mathbf{\Lambda}$, given this circumstance, the need to find an accurate representation for numbers within this definition arises. See Definition 2.1.13

Example 2.1.1. Some examples of valid λ terms generated using the grammar in 2.1.2: y, $(\lambda x.(xx))$, $(\lambda x.(\lambda y.x))$, $(((\lambda x.(xy))(\lambda y.y))z)$. The haskell counterpart for the second term is:

```haskell
Abs "x" (App (Var "x") (Var "x"))
```

Listing 2.2: Haskell interpretation of the second term

As a convention to avoid notational cluttering, the outermost parenthesis can be omitted e.g. $(\lambda x.x)$ can be read as $\lambda x.x$. Also, application is left-associative and binds tighter than abstraction, so MNP is parsed as $(MN)P$ and $\lambda x.MN$ means $\lambda x.(MN)$, not $(\lambda x.M)N$. Likewise, abstraction is right-associative e.g. $\lambda x.\lambda y.M$ is $\lambda x.(\lambda y.M)$ and binds more weakly than application. All these are just to keep notation straightforward, the formal definition does not leave precedence under-specified thanks to parenthesis.

2.1.1 Equivalence of terms

Having defined the language of the λ-calculus, we move on to the mechanics of computation. To this end, several equivalence relations among terms will be defined. Let us begin by introducing the set that contains all variables that are not bound to the term via an abstraction.

Definition 2.1.3. Given $T \in \mathbf{\Lambda}$, $\mathbf{FV} : \mathbf{\Lambda} \to \mathcal{P}(\mathbf{V})$ outputs the set of free variables for T:

$$\begin{aligned}
&\mathbf{FV}(x) = x \\
&\mathbf{FV}(MN) = \mathbf{FV}(M) \cup \mathbf{FV}(N) \\
&\mathbf{FV}(\lambda x.M) = \mathbf{FV}(M) \setminus \{x\}
\end{aligned}$$

Where $x \in \mathbf{V}$ and $M, N \in \mathbf{\Lambda}$, and $\mathcal{P}$ denotes the power set.

Example 2.1.2. Compute the free variables of $\lambda x.\lambda y.yxz$:
The term well-formed, thus belongs to $\mathbf{\Lambda}$, so we proceed:

$$\begin{aligned}\mathbf{FV}(\lambda x.\lambda y.yxz) &= \mathbf{FV}(\lambda y.yxz) \setminus \{x\} \\ &= \mathbf{FV}(yxz) \setminus \{x, y\} \\ &= \{x, y, z\} \setminus \{x, y\} \\ &= \{z\}\end{aligned}$$

Of course, z is the only free variable in the expression as it is the only variable that is not captured by some λ-abstraction.

Definition 2.1.4. A term $M \in \mathbf{\Lambda}$ is closed if and only if $\mathbf{FV}(M) = \emptyset$, closed lambda terms are often referred to as combinators. Λ^0 is the set of all closed lambda terms.

Since every lambda term is made up of other smaller lambda terms, it is only natural to define the set that contains all subterms of a given term:

Definition 2.1.5. Given $T \in \mathbf{\Lambda}$, $\mathbf{Sub} : \mathbf{\Lambda} \to \mathcal{P}(\mathbf{V})$ maps terms to the set of it's subterms.

$$\begin{aligned}&\mathbf{Sub}(x) = \{x\} \\ &\mathbf{Sub}(MN) = \mathbf{Sub}(M) \cup \mathbf{Sub}(N) \cup \{MN\} \\ &\mathbf{Sub}(\lambda x.M) = \mathbf{Sub}(M) \cup \{\lambda x.M\}\end{aligned}$$

Where $x \in \mathbf{V}$ and $M, N \in \mathbf{\Lambda}$.

Remark 2. Let $\preceq$ be the subterm relation on λ-terms, defined by

$$M \preceq N \iff M \in \mathbf{Sub}(N)$$

$\preceq$ is a partial order relation. This statement is follows intuitively from the definition, it can be proven using induction on the derivation tree.

Definition 2.1.6. A term $S \in \mathbf{Sub}(M),\ M \in \mathbf{\Lambda}$ is said to be proper if $S \neq M$. $\mathbf{Sub}(M) \setminus \{M\}$ would be the set of proper subterms.

Definition 2.1.7. Let $M = \lambda x.N$ be a λ-term:

- The variable x is the binding variable of the abstraction $\lambda x.N$.
- An occurrence of a variable x in M is bound if $x \in \mathbf{Sub}(\lambda x.N)$.

Definition 2.1.8. Capture-Avoiding Substitution Rules Let $M, N \in \mathbf{\Lambda}$ and $x, y, z \in \mathbf{V}$. The substitution $M[x := N]$ is defined inductively as:

$$\begin{aligned}
x[x := N] &= N \\
y[x := N] &= y && y \neq x \\
(M_1\, M_2)[x := N] &= (M_1[x := N])\,(M_2[x := N]) \\
(\lambda x.M)[x := N] &= \lambda x.M \\
(\lambda y.M)[x := N] &= \lambda y.(M[x := N]) && y \neq x, y \notin \mathbf{FV}(N) \\
(\lambda y.M)[x := N] &= \lambda z.(M[y := z][x := N]) && y \neq x, y \in \mathbf{FV}(N)
\end{aligned}$$

where $z \notin \mathbf{FV}(M) \cup \mathbf{FV}(N)$.

- Add insight and intuition upon the formal definition
- Add some haskell implementation details
- Maybe add code that generates α-equivalent terms

Definition 2.1.9. Two lambda terms $M, N \in \mathbf{\Lambda}$ are α-equivalent, written $M =_\alpha N$, if they are structurally identical except for the names of bound variables. Formally:

$$\lambda x.\, M =_\alpha \lambda y.\, M[x := y]$$

Alpha-equivalence posseses the following properties:

$$\begin{aligned}
&M =_\alpha M \\
&M =_\alpha N \Rightarrow N =_\alpha M \\
&M =_\alpha N \text{ and } N =_\alpha P \Rightarrow M =_\alpha P
\end{aligned}$$

And thus, is an equivalence relation.

- Examples

  Example 2.1.3.

  ... α-equivalence relations

- This relation is the most horizontal of the ones that show up in the text. By this I mean it just relates terms with equivalent terms at the same level in the β-reduction tree. (find a way to say this without having introduced β-reduction yet)

Definition 2.1.10. Single-step β-reduction defined using inference rules: (remove vertical space below)

$$\frac{}{(\lambda x.\, M)\, N \rightsquigarrow_\beta M[x := N]}\ \textsc{Beta} \qquad \frac{M \rightsquigarrow_\beta M'}{\lambda x.\, M \rightsquigarrow_\beta \lambda x.\, M'}\ \textsc{Abs}$$

$$\frac{M_1 \rightsquigarrow_\beta M_1'}{M_1\, M_2 \rightsquigarrow_\beta M_1'\, M_2}\ \textsc{AppL} \qquad \frac{M_2 \rightsquigarrow_\beta M_2'}{M_1\, M_2 \rightsquigarrow_\beta M_1\, M_2'}\ \textsc{AppR}$$

where $M, N \in \mathbf{\Lambda}$. For those not so familiar with inference rules here is an alternative definition where L also belongs to $\mathbf{\Lambda}$:

- $(\lambda x.\, M)\, N \rightsquigarrow_\beta M[x := N]$
- If $M \rightsquigarrow_\beta N$, then: $M\, L \rightsquigarrow_\beta N\, L; L\, M \rightsquigarrow_\beta L\, N; \lambda x.\, M \rightsquigarrow_\beta \lambda x.\, N$

Here, The Beta rule abstracts the concept of reduction, that encapsulates the idea of computation within the λ-calculus. The reason for this is that Beta reduces an application of an abstraction and removes the abstraction, hence simplifying the term.

Single β-reductions can be applied succesively, which induces the following definition

Definition 2.1.11. Zero or more step β-reduction, or as in the literature, *reflexive-transitive-closure*:

We write $M \rightsquigarrow_\beta^* N$ if and only if there exists a sequence of one-step β-reductions starting from M and ending at N. [3]

$$M \equiv M_0 \rightsquigarrow_\beta M_1 \rightsquigarrow_\beta M_2 \rightsquigarrow_\beta \cdots \rightsquigarrow_\beta M_{n-1} \rightsquigarrow_\beta M_n \equiv N$$

That is, there exists an integer $n \geq 0$ and a sequence of terms $M_0, M_1, \ldots, M_n$ such that: $M_0 \equiv M$ and $M_n \equiv N$ for all i with $0 \leq i < n$, we have $M_i \rightsquigarrow_\beta M_{i+1}$.

This process of b reducing a term leads raises the question of whether all terms can be reduced indefinitly or whether all terms have some kind of normal form that cant be reduced. (obv not since variables in V cant be reduced), so we are left with: Do all terms reduce to a final form where the only possible bred is M bred M? See recursion Y combinator...

[3] The convention in the literature is to use $\rightarrow_\beta$ for β-reductions $\twoheadrightarrow_\beta$ for multi-step β-reductions. I prefer to use $\rightsquigarrow_\beta$ and $\rightsquigarrow_\beta^n$ since this allows me to use the superscript to note the number of steps in the process β-reduction e.g. $\texttt{FALSE}\, M\, N \rightsquigarrow_\beta^2$ N. Instead of $\rightsquigarrow_\beta^n$ I use $\rightsquigarrow_\beta^*$.

Definition 2.1.12. β-equivalence relation add more detail and complete the definition

$$M =_\beta N \iff M \rightsquigarrow_\beta^* N \vee N \rightsquigarrow_\beta^* M$$

Put in other words:

$$M =_\beta N \iff \exists M_0, \ldots, M_n \text{ s.t. } M \equiv M_0 \leftrightarrow_\beta M_1 \leftrightarrow_\beta \cdots \leftrightarrow_\beta M_n \equiv N$$

where $\leftrightarrow_\beta$ denotes a bidirectional $\rightsquigarrow_\beta$.

Having covered β-reduction, we now explore a key aspect of the λ-calculus, in the jargon referred to as first-class citizenship, or more formally high-order. When we use high-order to refer to a λ-term, we refer to the fact that both functions and arguments are treated indistiguishably, the example below sheds some light on the matter:

$$\begin{aligned}\overbrace{(\lambda f.\lambda y.f\,f\,y)}^{A}\overbrace{(\lambda x.x+1)}^{B}\overbrace{(2)}^{C} &\rightsquigarrow_\beta (\lambda y.(\lambda x.x+1)(\lambda x.x+1)y)(2)\\ &\rightsquigarrow_\beta (\lambda x.x+1)(\lambda x.x+1)(2)\\ &\rightsquigarrow_\beta^* (\lambda x.x+1)(3)\\ &\rightsquigarrow_\beta^* (4)\end{aligned}$$

Taking a closer look, A, applies f twice to x, as a consequence, B is being applied twice to C. And so, very easily, we have implemented the $x+2$ function via a sequential application of $x+1$ to C. One can intuitively apreciate the computational expresivenes this brings with it, and how the sintactic-semantic homogeneity sets the λ-calculus apart from the classical set-theoretic approach to functions.

normalization + church rosser i.e. confluence

2.1.2 Some Important Constructs

Within the λ-calculus there exist a number of conventional constructions that represent key ideas in programming (numbers, booleans, branches and lists among others) In the following we define a few of the most important ones.

We begin defining $\mathbb{N}$ for no particular reason other than the natural numbers are the foundation of most of mathematics. The notion of natural numbers in the λ-calculus follows from the Peano axioms:

- (Zero) $0 \in \mathbb{N}$
- (Successor) $\forall x \in \mathbb{N}, s(x) \in \mathbb{N}$

- (Initial) $\forall x \in \mathbb{N}, s(x) \neq 0$
- (Injectivity) $\forall x, y \in \mathbb{N}, s(x) = s(y) \Rightarrow x = y$
- (Induction) $P(0) \wedge \forall (P(x) \Rightarrow P(s(x))) \Rightarrow \forall x, P(x)$

Maybe include here a graph to make the intuition behind the axioms explicit

Say for instance we mean to represent 3 using Peano notation, we would come up with $s(s(s(0)))$. To add two numbers one can just use the set of rules:

$$\begin{cases} n + s(m) = s(n + m) \\ n + 0 = n \end{cases}$$

A recursive rule that performs the addition, and a base case to stop the recursion for n + 0, since there is no such number that has 0 as succesor 2.1.2.

Definition 2.1.13. Representation of the set of all naturals, usually referred to as Church numerals:

$$\begin{aligned} \mathtt{0} &\equiv \lambda f.\lambda x.x \\ \mathtt{SUCC} &\equiv \lambda n.\lambda f.\lambda x.f(nfx) \end{aligned}$$

Remark 3. The church numerals have been defined similarly to the peano axioms, that is, by bringing into existance an initial element and using the succesor function to define the rest of them. Let us apply SUCC a few times to illustrate:

$$\begin{aligned} \mathtt{0} &\equiv \lambda f.\lambda x.x \\ \mathtt{1} &\equiv \lambda f.\lambda x.fx \\ \mathtt{2} &\equiv \lambda f.\lambda x.ffx \\ \mathtt{3} &\equiv \lambda f.\lambda x.fffx \\ \mathtt{n} &\equiv \lambda f.\lambda x.f^n x \end{aligned}$$

where n denotes the number of times we append f.

Now that we have a construction for the naturals, we can define some arithmetic operations:

$$\begin{aligned} \mathtt{ADD} &\equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx) \\ \mathtt{MUL} &\equiv \lambda f.\lambda x.\lambda f.\lambda x.m(nf)x \\ \mathtt{ISZERO} &\equiv \lambda n.n\,(\lambda x.\mathtt{FALSE})\,\mathtt{TRUE} \end{aligned}$$

This are left as a sidequest since they will not be used in the rest of the text.

The next construction in line are the truth values of boolean logic, "*true*" and "*false*" for short.

Definition 2.1.14. Boolean truth values:

$$\begin{aligned}\texttt{TRUE} &\equiv \lambda t.\lambda f.t\\ \texttt{FALSE} &\equiv \lambda t.\lambda f.f\end{aligned}$$

Remark 4. Note how $\texttt{TRUE}\ M\ N \rightsquigarrow_\beta M$ and $\texttt{FALSE}\ M\ N \rightsquigarrow_\beta N$

Definition 2.1.15. Conditional, in other words *"if statement"*.

$$\texttt{IF} \equiv \lambda b.\lambda t.\lambda e.b\,t\,e$$

The reality of the IF is that it is just a combinator that happens to behave as a branching operator whenever we pass a church encoded boolean as a first argument.

Example 2.1.4. On the correct behabiour of IF i.e. check that it implements branching. $M, N \in \mathbf{\Lambda}$ and $\texttt{IF}, \texttt{TRUE}, \texttt{FALSE}$ are as usual:

$$\begin{aligned}&\texttt{IF TRUE}\ M\ N\\ \equiv\ &(\lambda b.\lambda t.\lambda e.\,b\,t\,e)\ \texttt{TRUE}\ M\ N\\ \rightsquigarrow_\beta\ &(\lambda t.\lambda e.\,\texttt{TRUE}\ t\,e)\ M\ N\\ \rightsquigarrow_\beta\ &(\lambda e.\,\texttt{TRUE}\ M\,e)\ N\\ \rightsquigarrow_\beta\ &\texttt{TRUE}\ M\ N \rightsquigarrow_\beta^* M\end{aligned}$$

$$\begin{aligned}&\texttt{IF FALSE}\ M\ N\\ \rightsquigarrow_\beta^*\ &\texttt{FALSE}\ M\ N \rightsquigarrow_\beta N\end{aligned}$$

So the IF just combines b, t, e in such a way that whenever b is a church encoded boolean either t or e is dropped depending on the truth value of b. See Remark 4. From this example we extract that $\lambda x.x$ is equivalent to IF it terms of behaviour. Thus, we could interchangeably define $\texttt{IF} \equiv \lambda x.x$.

Remark 5. Using the IF we can implement an universal set of logic gates:

$$\begin{aligned}\texttt{NOT} &\equiv \texttt{IF}\,b\,\texttt{FALSE}\,\texttt{TRUE}\\ \texttt{AND} &\equiv \texttt{IF}\,a\,(\texttt{IF}\,b\,\texttt{TRUE}\,\texttt{FALSE})\,\texttt{FALSE}\end{aligned}$$

This allows us to simulate a nand gate, known to be universal ?? universality, conjunctive normal form. Thus, the lambda calculus is at least equivalent to propositional logic ?? Something on turing completeness.

On the same track, another must-have for programmers is lists. Sice they are a bit more involved, we require tuples to define them. Note that the `IF` is a tuple that contains two elements, we query the first element using a `TRUE` i.e. the *"then"* branch, and the sencond using a `FALSE` i.e. the *"else"* branch. We will be using this alternative definition for convenience:

Definition 2.1.16. $\texttt{PAIR} \equiv \lambda x.\lambda y.\lambda f.f\, x\, y$

Remark 6. This is more convenient since $\texttt{PAIR}\, M\, N \rightsquigarrow_\beta^* \lambda f.f\, M\, N$, which allows us to query the first element using $(\lambda f.f\, M\, N)\, \texttt{TRUE}$ and the second by $(\lambda f.f\, M\, N)\, \texttt{FALSE}$, this motivates the definition below:

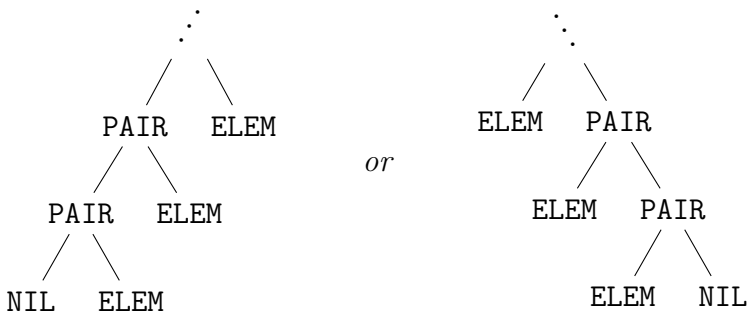
Definition 2.1.17.

$$\texttt{FST} \equiv \lambda p.p\, (\lambda x.\lambda y.x)$$
$$\texttt{SND} \equiv \lambda p.p\, (\lambda x.\lambda y.y)$$

Proposition 2.1.1. A key property of `PAIR`'s is:

$$\texttt{PAIR}\, A\, B \equiv_\beta \texttt{PAIR}\, (\texttt{FST}\, A\, B)\, (\texttt{SND}\, A\, B)$$

Using pairs, we now can define linear lists, we can achieve this using either left linear trees or right linear trees:



Definition 2.1.18. We define `NIL` in the same way we defined `0` and `FALSE` before: $\texttt{NIL} \equiv \lambda f.\lambda z.z$; `NIL` represents the terminating element of the inductive definition for lists. To build lists, we cons lists together using:

$$\texttt{CONS} \equiv \lambda h.\lambda t.\lambda f.\lambda z.f\, h\, (t\, f\, z)$$

that binds a new element h to to an pre-existing list or to `NIL`.

Note 1. This notion of inductively defined lists is lays the foundations of the **LISP** family of programming languages, hence the name (LISt Processing).

Definition 2.1.19. Relevant list operators:

$$\texttt{ISNIL} \equiv \lambda l.l\,(\lambda h.\lambda t.\texttt{FALSE})\,\texttt{TRUE}$$
$$\texttt{HEAD} \equiv \lambda l.l\,(\lambda h.\lambda t.h)\,\text{undef}$$

left blank

$$\texttt{CONS} \equiv \lambda h.\lambda t.\lambda f.\lambda z.f\ h\ (t\ f\ z)$$
$$\texttt{IS\_NIL} \equiv \lambda l.l\ (\lambda h.\lambda t.\texttt{FALSE})\ \texttt{TRUE}$$
$$\texttt{HEAD} \equiv \lambda l.l\ (\lambda h.\lambda t.h)\ \text{undef}$$
$$\texttt{TAIL} \equiv \lambda l.\,\texttt{FIRST}\ (l\ (\lambda p.\lambda h.\,\texttt{PAIR}\ (\texttt{SECOND}\ p)\ (\texttt{CONS}\ h\ (\texttt{SECOND}\ p)))\ (\texttt{PAIR}\ \texttt{NIL}\ \texttt{NIL}))$$

Recursion:

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

2.1.3 Fixed Point Combinators and Recursion

Y combinator.

$$Y \triangleq \lambda f.\,(\lambda x.\,f\ (x\,x))\ (\lambda x.\,f\ (x\,x))$$

Turing combinator

$$\Theta \triangleq (\lambda x\ f.\,f\ (x\,x\,f))\ (\lambda x\ f.\,f\ (x\,x\,f))$$

Z combinator

$$Z \triangleq \lambda f.\,(\lambda x.\,f\ (\lambda v.\,x\,x\,v))\ (\lambda x.\,f\ (\lambda v.\,x\,x\,v))$$

Turing completeness of the λ-calculus

2.2 Simply Typed λ-Calculus

rewrite this pls ==>
The Untyped Lambda Calculus in computationally equivalent to a Turing machine. However, with great computational power comes limited decidability of properties, leading to non-termination, or expressions such as $x(\lambda y.y)$, whose meaning is unclear.
A classic example of non-termination is the Y combinator, however, the Simply Typed Lambda Calculus does not allow such expressions, as its type system is unable to assign a valid type to them.
if there is a proof of turing completeness, add how if we remove combinators then we remove turing completeness i.e. finite tape ≠ turing complete
To understand why, consider the role of function types: in the world of functions, a function maps values from a domain to a range. The

Simply Typed Lambda Calculus enforces this structure explicitly, ensuring that every function application is well-typed and preventing self-application patterns that would lead to paradoxes or infinite loops. Having grasped the untyped lambda calculus's Turing completeness and ability to compute all computable functions, we now seek properties related to decidability. To this end, we introduce the simply typed lambda calculus. Although it possesses less computational power than its untyped counterpart, it offers attractive features regarding decidability that will be useful later on.

Chapter 3

The Curry-Howard Correspondence

A Primer on Logic

build up to first order logic