

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. Implementations given will not necessarily be optimal since they are meant for educational purpose



# Chapter 1

## $\lambda$ -Calculus

In the early 20th century, mathematicians such as Bertrand Russell, David Hilbert, and Kurt Gödel were trying to set complete, consistent, and decidable foundations for mathematics. Within this context, Alonzo Church, in his effort to formalize the notion of computability, introduced a minimal symbolic language [1.1.1] based upon function abstraction and function application, this system is now called  $\lambda$ -calculus.

In 1936, Church used lambda calculus to address Hilbert's Entscheidungsproblem—whether a mechanical method could determine the truth of any first-order logic statement. He proved no such algorithm exists, establishing the undecidability of first-order logic.

One of the nuances formalized by the  $\lambda$ -calculus is the distinction between extensional and intensional equality. The extensional approach to equivalence states that two functions are equivalent if they share input output pairs for every possible input. Its intensional counterpart extends this notion of equality by requiring that the procedures that compute these pairs share complexity i.e. they take the same steps toward yielding a result.<sup>1</sup>

Languages like LISP, Haskell, Erlang and others share the  $\lambda$ -calculus as its theoretical foundation. It has established itself as the backbone of functional programming. The introduction of type systems into the  $\lambda$ -calculus has allowed us to computer-verify mathematical proofs and develop programs that are correct by construction.<sup>2</sup>

---

<sup>1</sup>Let  $p$  be a sufficiently large prime, and let  $f, g : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$  be defined by  $f(x) = x^2$  and  $g(x) = \log_a(a^{x+2})$ , where  $a \in \mathbb{Z}_p^\times$  is a fixed primitive root. Although  $f$  and  $g$  are extensionally equal, i.e., they yield the same output for all inputs in  $\mathbb{Z}_p$ , they are intensionally distinct. The function  $f$  performs a simple squaring operation, while  $g$  requires evaluating a discrete logarithm, intractable in general. ??[Pedro Bonilla]

<sup>2</sup>Principles of the  $\lambda$ -calculus and Type Theory underlie every computer assisted verification tool as well as proof assistants and kitchen table programming languages like C or Java.

## 1.1 Untyped $\lambda$ -Calculus

In order to get acquainted with the  $\lambda$ -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  $\lambda$ -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<sup>3</sup>. 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  $\lambda$ -expression is referred to as  $\beta$ -reduction, it will be covered formally later in the chapter.

**Definition 1.1.1.** The set of all 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  $\mathbf{V} = \{x, y, z, \dots\}$  represents a countably infinite set of variable names.

The key takeaway of this definition is that abstraction and application together, when combined with  $\beta$ -reduction, enable computation—thus encapsulating the meaning of function in a way that is abstract yet useful.

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

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 (BNF) ??[BNF] is the way to go:

- Nonterminals are enclosed in angle brackets (e.g.  $\langle \text{expr} \rangle$ )

---

<sup>3</sup>this is an e.g.,  $(\lambda x.x + 1)$  is not a valid lambda term, see Definition 1.1.2.

- Terminals are written literally (e.g. " $\lambda$ ", ".",  $x$ )
- 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 1.1.2.** Taking advantage of BNF notation, an alternative definition for  $\Lambda$  [1.1.1]:

```
<term> ::= <variable>
        | "<math>\lambda</math>" <variable> "." <term>
        | "(" <term> <term> ")"
<variable> ::=  $\in V$ 
```

where  $\Sigma = \{\lambda, ., (, )\} \cup V$  and  $\Lambda \subset \Sigma^*$ . Through use of algebraic types<sup>4</sup>, this grammar can be implemented in Haskell:

```
1 data Term = Var String
2           | App Term Term
3           | Abs String Term
```

Listing 1.1: Grammar implementation of the  $\lambda$ -calculus language using Haskell's algebraic types

**Remark 1.** From the definition above we extract that  $\lambda x.x + 1 \notin \Lambda$ , given this circumstance, the need to find an accurate representation for numbers within this definition arises. See Definition 1.1.15

**Example 1.1.1.** Some examples of valid  $\lambda$  terms generated using the grammar in 1.1.2:  $y$ ,  $(\lambda x.(xx))$ ,  $(\lambda x.(\lambda y.x))$ ,  $((\lambda x.(xy))(\lambda y.y))z$ . The haskell counterpart for the second term is:

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

Listing 1.2: Haskell interpretation of the second term

<sup>4</sup>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.

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. Those interested in the Haskell implementation need not worry, as Haskell enforces an explicit order on constructors.

### 1.1.1 Equivalence of terms

Having defined the language of the  $\lambda$ -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 1.1.3.** Given  $T \in \mathbf{\Lambda}$ ,  $\mathbf{FV} : \mathbf{\Lambda} \rightarrow \mathcal{P}(\mathbf{V})$  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.

```

1 FV :: Term -> Set String
2 FV (Var x)      = Set.singleton x
3 FV (App t1 t2) = Set.union (FV t1) (FV t2)
4 FV (Abs x t)   = Set.delete x (FV t)
```

Listing 1.3: Implementation of FVV in Haskell.

**Example 1.1.2.** 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  $\lambda$ -abstraction.

**Definition 1.1.4.** 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.  $\mathbf{\Lambda}^0$  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 1.1.5.** Given  $T \in \mathbf{\Lambda}$ ,  $\mathbf{Sub} : \mathbf{\Lambda} \rightarrow \mathcal{P}(\mathbf{V})$  maps a term 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  $M, N \in \mathbf{\Lambda}$ .

```

1 Sub :: Term -> Set Term
2 Sub v@(Var _)      = Set.singleton v
3 Sub a@(App t1 t2) = Set.insert a (Set.union (Sub t1)
4                                     (Sub t2))
5 Sub a@(Abs _ t)    = Set.insert a (Sub t)

```

Listing 1.4: Haskell implementation of Subb.

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

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

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

**Definition 1.1.6.** 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 1.1.7.** Let  $M = \lambda x.N$  be a  $\lambda$ -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 1.1.8.** 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  $z \notin \mathbf{FV}(M) \cup \mathbf{FV}(N)$ .

---

Listing 1.5: asdasdasd

- De Bruijn
- Add insight and intuition upon the formal definition
- Add some haskell implementation details
- Maybe add code that generates  $\alpha$ -equivalent terms

**Definition 1.1.9.** Two lambda terms  $M, N \in \mathbf{\Lambda}$  are  $\alpha$ -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 possesses 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 1.1.3.**

...  $\alpha$ -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  $\beta$ -reduction tree. (find a way to say this without having introduced  $\beta$ -reduction yet)

**Definition 1.1.10.** Single-step  $\beta$ -reduction defined using inference rules:  
(remove vertical space below)



$$\begin{array}{c}
\frac{}{(\lambda x. M) N \rightsquigarrow_{\beta} M[x := N]} \text{BETA} \qquad \frac{M \rightsquigarrow_{\beta} M'}{\lambda x. M \rightsquigarrow_{\beta} \lambda x. M'} \text{ABS} \\
\\
\frac{M_1 \rightsquigarrow_{\beta} M'_1}{M_1 M_2 \rightsquigarrow_{\beta} M'_1 M_2} \text{APPL} \qquad \frac{M_2 \rightsquigarrow_{\beta} M'_2}{M_1 M_2 \rightsquigarrow_{\beta} M_1 M'_2} \text{APPR}
\end{array}$$

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  $\lambda$ -calculus. The reason for this is that Beta reduces an application of an abstraction and removes the abstraction, hence simplifying the term.

Single  $\beta$ -reductions can be applied succesively, which induces the following definition

**Definition 1.1.11.** Zero or more step  $\beta$ -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  $\beta$ -reductions starting from  $M$  and ending at  $N$ .<sup>5</sup>

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

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

The process of succesively  $\beta$ -reducing a term leads raises the question of whether all terms can be reduced indefinitely. We know the answer to this, since for instance, variables in  $\mathbf{V}$  cannot be reduced. The subsequent question then is, Do all all terms have some kind of normal form that cannot be reduced further? This is answered in Section [1.1.3].

---

<sup>5</sup>The convention in the literature is to use  $\rightarrow_{\beta}$  for  $\beta$ -reductions  $\rightarrow_{\beta}$  for multi-step  $\beta$ -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  $\beta$ -reduction e.g.  $\text{FALSE } M N \rightsquigarrow_{\beta}^2 N$ . Instead of  $\rightsquigarrow_{\beta}^n$  I use  $\rightsquigarrow_{\beta}^*$  whenever the number of steps is undefined.

**Definition 1.1.12.**  $\beta$ -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:

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

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

Having covered  $\beta$ -reduction, we now explore a key aspect of the  $\lambda$ -calculus, in the jargon referred to as first-class citizenship, or more formally high-order. When we use high-order to refer to a  $\lambda$ -term, we refer to the fact that both functions and arguments are treated indistinguishably, 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 appreciate the computational expressiveness this brings with it, and how the syntactic-semantic homogeneity sets the  $\lambda$ -calculus apart from the classical set-theoretic approach to functions.

**normalization + church rosser i.e. confluence**

### 1.1.2 Some Important Constructions

Within the  $\lambda$ -calculus there exist a number of conventional constructions that represent key ideas in programming (numbers, booleans, branching and lists among others) In the following we define a few of the most important ones. These constructions can be used to prove relevant statements concerning the expressiveness of the  $\lambda$ -calculus.

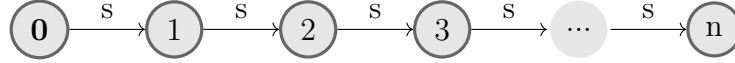
We begin defining  $\mathbb{N}$  for no particular reason other than the natural numbers are the foundation of most of computer science. The notion of natural numbers in the  $\lambda$ -calculus follows from the Peano axioms:

**Definition 1.1.13.** Peano Axioms, the set of natural numbers  $\mathbb{N}$

- (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)  $P(0) \wedge \forall (P(x) \Rightarrow P(s(x))) \Rightarrow \forall x, P(x)$

Graphically:



**Example 1.1.4.** Say for instance we mean to represent 3 using Peano notation, using the graph, we would come up with  $s(s(s(0)))$ , since it is 3 steps away from 0.

**Definition 1.1.14.** To add and multiply natural numbers one can just use following the sets of rules:

$$n + m = \begin{cases} n & \text{if } m = 0 \\ s(n + m') & \text{if } m = s(m') \end{cases} \quad a \cdot b = \begin{cases} 0 & \text{if } b = 0 \\ a \cdot b' + a & \text{if } b = s(b') \end{cases}$$

$a + b$ : 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 successor 1.1.13.

$a \cdot b$ : Using addition we define multiplication in a similar way, adding  $a$  as many times as the function  $s$  is applied to 0 in  $b$

**Definition 1.1.15.** Representation of the set of all naturals, usually referred to as Church numerals:

$$0 \equiv \lambda f. \lambda x. x \\ \text{SUCC} \equiv \lambda n. \lambda f. \lambda x. f(nfx)$$

**Example 1.1.5.** The church numerals have been defined similarly to the peano axioms, that is, by bringing into existence an initial element and using the successor function to define the rest of them. Let us apply **SUCC** a few times to illustrate:

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

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:

$$\begin{aligned}\text{ADD} &\equiv \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x) \\ \text{MUL} &\equiv \lambda f. \lambda x. \lambda f. \lambda x. m (n f) x \\ \text{ISZERO} &\equiv \lambda n. n (\lambda x. \text{FALSE}) \text{TRUE}\end{aligned}$$

This proves that the  $\lambda$ -calculus is capable of arithmetic, we leave them as a sidequest since we will not be extending or using them or in the rest of the text.

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

**Definition 1.1.16.** Boolean truth values:

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

**Remark 3.** Note how  $\text{TRUE } M \ N \rightsquigarrow_{\beta} M$  and  $\text{FALSE } M \ N \rightsquigarrow_{\beta} N$

**Definition 1.1.17.** Conditional, in other words “*if statement*”.

$$\text{IF} \equiv \lambda b. \lambda t. \lambda e. b t e$$

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

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

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 3. From this example we extract that  $\lambda x. x$  is equivalent to IF in terms of behaviour, and thus, we could use  $\text{IF} \equiv \lambda x. x$  interchangeably.

**Remark 4.** Using the IF we can implement an universal set of logic gates:

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

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. Since 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 second using a **FALSE** i.e. the “*else*” branch. We will be using this alternative definition for convenience:

**Definition 1.1.18.**  $\text{PAIR} \equiv \lambda x.\lambda y.\lambda f.f\ x\ y$

**Remark 5.** This is more convenient since  $\text{PAIR}\ M\ N \rightsquigarrow_{\beta}^* \lambda f.f\ M\ N$ , which allows us to query the first element using  $(\lambda f.f\ M\ N)\ \text{TRUE}$  and the second by  $(\lambda f.f\ M\ N)\ \text{FALSE}$ , this motivates the definition below:

**Definition 1.1.19.**

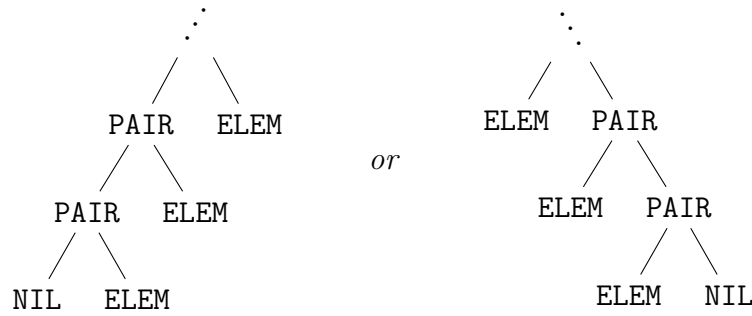
$$\text{FST} \equiv \lambda p.p\ (\lambda x.\lambda y.x)$$

$$\text{SND} \equiv \lambda p.p\ (\lambda x.\lambda y.y)$$

**Proposition 1.1.1.** A key property of **PAIR**’s is:

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

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



**Definition 1.1.20.** We define **NIL** in the same way we defined **0** and **FALSE** before:  $\text{NIL} \equiv \lambda f.\lambda z.z$ ; **NIL** represents the terminating element of the inductive definition for lists. To build lists, we cons lists together using:

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

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 1.1.21.** Relevant list operators:

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

$$\text{HEAD} \equiv \lambda l.l (\lambda h.\lambda t.h) \text{undef}$$

### 1.1.3 Fixed Point Combinators and Recursion

The most straight forward way to test this, is to check whether there exists some term that reduces to itself, effectively creating a loop in the  $\beta$ -reduction chain so to speak, that is  $M \rightsquigarrow_{\beta}^n M$  with  $n > 0$ . It turns out  $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$  fullfills this requirement. Let us inspect what goes on as we keep  $\beta$ -reducing this term:

$$\Omega \equiv (\lambda x.xx)(\lambda x.xx) \rightsquigarrow_{\beta} (\lambda x.xx)(\lambda x.xx) \rightsquigarrow_{\beta}^* (\lambda x.xx)(\lambda x.xx) \equiv \Omega$$

This so called  $\Omega$ -combinator, proves that not all terms have a normal form, and shines a beam of light onto how recursion can be achieved in the  $\lambda$ -calculus.

There is one caviat to all of this, namely that classically recursion is achieved through naming, take for instance this definition for the gcd:

$$\underline{\text{gcd}}(a, b) = \begin{cases} a & b = 0 \\ \underline{\text{gcd}}(b, a \bmod b) & b \neq 0 \end{cases}$$

Here, when  $b \neq 0$  the name gcd is used to refer to the same definition again. In the  $\lambda$ -calculus such self-referencing definitions are not permitted since the language does not allow names. See [1.1.2]. To bypass this setback the notion of fixed point is of particular interest.

**Definition 1.1.22.** Let  $f : X \rightarrow X$  be a function. An element  $x \in X$  is called a fixed point of  $f$  if  $f(x) = x$ .

**Example 1.1.7.** Some simple functions that illustrate what fixed points are about:

<i>Function</i>	<i>Fixed Point(s)</i>
$f(x) = c$	$x = c$
$g(x) = x$	$\forall x, x \in \mathbb{R}$
$h(x) = x^2$	$x, x \in \{0, 1\}$

This harmless concept satisfies an interesting property, namely that  $x = f(x) = f(f(x)) = f^n(x)$  effectively creating a chain every element  $f^{n+1}(x) = f(f^n(x))$ , thus attaining self-reference without naming. This idempotent behaviour under application is important, since it showcases the ability of fixed points to induce recursion. Our quest now, is to find a term that simulates this behaviour. To do this, let us translate the definition for fixed point into the  $\lambda$ -world. **maybe use circ (i.e function composition) in the chain since it might have some meaning in the context of categories.**

**Definition 1.1.23.** Given a lambda term  $L \in \mathbf{\Lambda}$ , a term  $M \in \mathbf{\Lambda}$  is called a fixed point of  $L$  if  $M =_\beta LM$ .

**Theorem 1.1.2.** For every  $L \in \mathbf{\Lambda}$ , there exists  $M \in \mathbf{\Lambda}$  such that  $M$  is a fixed point for  $M$  i.e.  $M =_\beta LM$ . Referred to as the Fixed Point Theorem in the literature.

*Proof.* Given  $L \in \mathbf{\Lambda}$ , define  $M$  as below:

$$M := (\lambda x. L(x x))(\lambda x. L(x x))$$

$M$  is a reducible expression, redex for short:

$$\begin{aligned} M &\equiv (\lambda x. L(x x))(\lambda x. L(x x)) \\ &\rightsquigarrow_\beta L((\lambda x. L(x x))(\lambda x. L(x x))) \\ &\equiv LM. \end{aligned}$$

Hence,  $M =_\beta LM$ , as required.  $\square$

See how  $LM$  is the application of  $M$  to  $L$  and that using APPR we can keep  $\beta$ -reducing  $M$  the same way we did for  $x = f(x) = f(f(x)) \dots$  only this time using lambda terms:

$$M \rightsquigarrow_\beta LM \rightsquigarrow_\beta LLM \rightsquigarrow_\beta^* L^n M.$$

Now that nameless self-reference has been proven to exist within the realm of the  $\lambda$ -calculus, we take a step forward and introduce a few of the most important fixed point combinators and explore their different natures.

**Definition 1.1.24.** *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  $\lambda$ -calculus

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

$$\text{IS\_NIL} \equiv \lambda l. l (\lambda h. \lambda t. \text{FALSE}) \text{TRUE}$$

$$\text{HEAD} \equiv \lambda l. l (\lambda h. \lambda t. h) \text{undef}$$

$$\text{TAIL} \equiv \lambda l. \text{FIRST} (l (\lambda p. \lambda h. \text{PAIR} (\text{SECOND } p) (\text{CONS } h (\text{SECOND } p)))) (\text{PAIR NIL NIL})$$

Recursion:

$$Y \equiv \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

## 1.2 Simply Typed $\lambda$ -Calculus

The Untyped Lambda Calculus is 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  $\neq$  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 2

# The Curry-Howard Correspondence

A Primer on Logic

build up to first order logic