Chapter 1

λ-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 he's effort to formalize the notion of computability, introduced a minimal symbolic language based upon function abstraction and function application, this system is now called λ-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 λ-calculus is the disctinction between extensional and intensional equality. The extensional approach to equivalence states that two functions are equivalent if they share input output pairs for every posible input. It's 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.[1] Languages like LISP, Haskell, Erlang and others share the λ-calculus as their theoretical foundation. It has has established itself as the backbone of functional programming. The introduction of type systems into the λ-calculus has allowed us to computer-verify mathematical proofs and develop programs that are correct by construction.[2]

[1] Let p be a sufficiently large prime, and let $f, g : \mathbb{Z}_p \to \mathbb{Z}_p$ 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]

[2] Principles of the λ-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 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[3]. 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 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 $\mathbf{V} = \{x, y, z, ...\}$ represents a countably infinite set of variable names.

The key takeaway of this definition is that abstraction and application together, when combined with β-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 $\Sigma = \{\lambda, ., (,), \ldots\}$, 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, BNF for short is the way to go:

- Nonterminals are enclosed in angle brackets (e.g. `<expr>`)

[3]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. "λ", ".", x)
- Productions define how nonterminals expand, written as ::=
- The vertical bar | denotes available expansions

Thus, the language for natural numbers in decimal notation is represented using:

```
 <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 $\mathbf{\Lambda}$ would be:

```
<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 [4], this grammar can be implemented in Haskell:

```haskell
data Term = Var String
          | App Term Term
          | Abs String Term
```

Listing 1.1: 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 1.1.16

Example 1.1.1. Some examples of valid λ 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:

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

Listing 1.2: Haskell interpretation of the second term

[4] 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 λ-calculus, we move on to the mechanics of computation. To this end, several equivalences among terms will be defined, namely α-equivalence, β-equivalence, and η-equivalence. Every lambda term is made up of other smaller lambda terms, so it is only natural to define the set that contains all subterms of a given term:

Definition 1.1.3. Given $T \in \mathbf{\Lambda}$, $\mathbf{Sub} : \mathbf{\Lambda} \to \mathcal{P}(\mathbf{V})$ 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)
     (Sub t2))
4 Sub a@(Abs _ t)    = Set.insert a (Sub t)
```

Listing 1.3: Haskell implementation of Sub.

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 1.1.4. 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 1.1.5. 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 $x \in \mathbf{Sub}(\lambda x.N)$.
- Free variables are those that are not bound.

Example 1.1.2. Take $M \equiv \lambda x.\, x\, y$, in this example, the x in $\lambda x.$ is binding, y is free, and x is bound. This means that y is used literally, and that x, as denoted by $\lambda x.$ is abstracted and thus subject to being replaced within the context of said abtraction. A subexample might make this explicit: $(\lambda x.\, x\, y)\, M \rightsquigarrow_\beta M\, y$. The y remained while the x was used as a placeholder for M.

Remark 3. This taxonomy is key to understanding concepts such as α-equivalence intuitively.

The set of Free Variables **FV**, those that will not get replaced during the process of reduction, is computed using:

Definition 1.1.6. 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.

```
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 1.4: Implementation of FV in Haskell.

Example 1.1.3. Computing the free variables of $\lambda x.\lambda y.yxz$. The term well-formed, 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 1.1.7. 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.

Note 1. They are called combinators since having no free variables means that all variables are bound, and thus, a closed term just combines bound variables.

We aproach the definition for α-equivalence to this end we have to introduce what it means to substitute a variable.

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:

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

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

Listing 1.5: asdasdasd

- Add insight and intuition upon the formal definition
- Add some haskell implementation details

Definition 1.1.9. 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 is an equivalence relation:

$$
\begin{array}{ll}
(Reflexivity) & M =_\alpha M \\
(Symetry) & M =_\alpha N \Rightarrow N =_\alpha M \\
(Transitivity) & M =_\alpha N \text{ and } N =_\alpha P \Rightarrow M =_\alpha P
\end{array}
$$

Example 1.1.4. A simple equivalence class and two samples of not α-equivalent pairs.

$$\lambda x.\lambda y.\, y =_\alpha \lambda a.\lambda b.\, b =_\alpha \lambda z.\lambda x.\, x$$

$$\lambda x.\lambda y.\, x \neq_\alpha \lambda x.\lambda y.\, y \quad \lambda x.\, x\, y \neq_\alpha \lambda x.\, x\, z$$

Remember terms are α-equivalence only when bounded names are properly substituted.

Remark 4. α-equivalence comes naturally once substitution is defined, since the names of bound variables are just placeholders whose only purpuse is to get replaced.

Naming variables can be a nightmare, which is why there exists a way of methodically assigning numbers to variables in a way that eliminates the need for α-equivalence. These naming system works by using numeric indices that indicate how many binders away a variable's binder is. These indices are called De Bruijn indeces.

$$\lambda x.\, x \Rightarrow \lambda.\, 0 \quad \lambda x.\, \lambda y.\, x \Rightarrow \lambda.\, \lambda.\, 1 \quad \lambda x.\, \lambda y.\, y \Rightarrow \lambda.\, \lambda.$$

It turns out we are better at naming variables than at working with numbers, which is why we will stick to symbols and leave De Bruijn indices to computers. Programs like automated theorem provers do use this naming strategy to simplify their work.

Definition 1.1.10. Single-step β-reduction $\rightsquigarrow_\beta$ is defined using the following inference rules:

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

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

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

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

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

Definition 1.1.11. Zero or more step β-reduction, also called 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.[5]

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

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 $M_i \rightsquigarrow_\beta M_{i+1}$.

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

Definition 1.1.12. β-equivalence relation, equivalence of λ-terms through β-reductions.

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

Using definition 1.1.11:

$$M =_\beta N \iff \begin{cases} \exists M_0, \ldots, M_n \text{ s.t. } M \equiv M_0 \rightsquigarrow_\beta M_1 \rightsquigarrow_\beta \cdots \rightsquigarrow_\beta M_n \equiv N \\ \exists N_0, \ldots, N_n \text{ s.t. } N \equiv N_0 \rightsquigarrow_\beta N_1 \rightsquigarrow_\beta \cdots \rightsquigarrow_\beta N_n \equiv M \end{cases}$$

Remark 5. In the context of equivalence of programs, β-equivalence can be understood as means for intensional equivalece, the kind of equivalence that requires programs to be equivalent in a step-by-step fashion.

We now explore a key aspect of the λ-calculus, in the jargon referred to as first-class citizenship, or high-order, a property of terms that restrains them from being exclusively classified as either function or argument. The example below sheds some light on the matter:

Example 1.1.5. High Order by example. Let us β-reduce the λ-term $A\,B\,C$ and inspect the consequences:

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

[5]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. $(\lambda f.\lambda x.\, x)\, M\, N \rightsquigarrow_\beta^2 \text{N}$. Instead of $\rightsquigarrow_\beta^n$ I use $\rightsquigarrow_\beta^*$ whenever the number of steps is undefined.

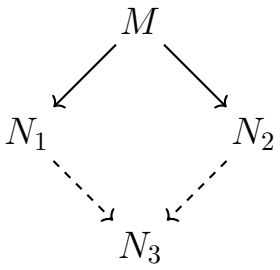
Taking a closer look, A, applies f twice to x, as a consequence, B is being applied twice to C. And so, very easily, we have implemented the $x+2$ function via a sequential application of $x+1$ to C.

Remark 6. As seen above, we can pass what we understand as conventional functions as arguments since they are treated alike. One can intuitively apreciate the computational expresivenes this brings with it, and how this sintactic-semantic homogeneity sets the λ-calculus apart from the classical set-theoretic approach to functions.

Theorem 1.1.1. The Church-Rosser Property states that $\forall M, N_1, N_2 \in \mathbf{\Lambda}$:

$$M \rightsquigarrow_\beta^* N_1 \quad \text{and} \quad M \rightsquigarrow_\beta^* N_2 \quad \Rightarrow \quad \exists N_3 \mid N_1 \rightsquigarrow_\beta^* N_3 \wedge N_2 \rightsquigarrow_\beta^* N_3$$

If a lambda term M has a normal form, then every reduction sequence starting from M eventually reduces to that same normal form, up to alpha-equivalence. The diagram below is the reason this theorem is often referred to as the diamond property.



The proof to this theorem is a quite lengthy, we would be miss the point of this document if we were to introduce it [ChurchRosser].

Corollary 1.1.2. Normalization: If a λ-term M reduces to two normal forms N_1 and N_2, then they are alpha-equivalent:

$$M \rightarrow_\beta^* N_1 \quad \text{and} \quad M \rightarrow_\beta^* N_2 \quad \text{and } N_1, N_2 \text{ are normal} \quad \Rightarrow \quad N_1 =_\alpha N_2$$

we introduced beta reductions i.e. intensional equivalence, what about extensionality?

The last and indeed the least important notion of equivalence:

Definition 1.1.13. Two lambda terms M and N are said to be η-equivalent, written $M =_\eta N$, if:

$$\lambda x.\,(M\ x) =_\eta M \quad x \notin \mathbf{FV}(M)$$

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

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

The abstraction $\lambda x.\, (M\ x)$ performs the same operation as M, so if x is not used freely within M, the abstraction is redundant.

η-equivalence represents weak extensionality, since it does not encode observational equivalence. Two terms are observationally equivalent if they cannot be distinguished by any context that can be papplied to them.

1.1.2 Some Important Constructions

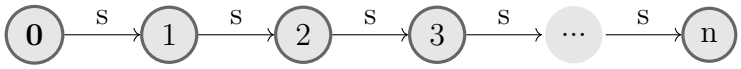
Within the λ-calculus there exist a number of conventional constructions that represent key ideas in programming (numbers, booleans, branching and lists among others) In the following we define a few of the most important ones. These constructions can be used to prove relevant statements concerning the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

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

Definition 1.1.14. Peano Axioms, the set of natural numbers $\mathbb{N}$

- (Zero) $0 \in \mathbb{N}$
- (Successor) $\forall n \in \mathbb{N}, S(n) \in \mathbb{N}$
- (Initial) $\forall n \in \mathbb{N}, S(n) \neq 0$
- (Injectivity) $\forall n, m \in \mathbb{N}, S(n) = S(m) \Rightarrow n = m$
- (Induction) $P(0) \wedge \forall (P(n) \Rightarrow P(S(n))) \Rightarrow \forall n, P(n)$

The first four axioms can be summarized using a graph:



The last axiom is concerned with proving properties for all naturals given a precedent that satisfies the property and proof that if the property is satisfied for n then it is satisfied for $n + 1$, then the property is true of all naturals.

Example 1.1.6. 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 1.1.15. To add and multiply natural numbers one can just use following the sets of rules:

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

- $\underline{a + b}$: 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 succesor.
- $\underline{a \cdot b}$: 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 1.1.16. Representation of the set of all naturals in the realm of λ-calculus, usually referred to as Church numerals:

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

Example 1.1.7. 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.

Remark 7. Since λ-terms are first-class citizens, and f and x are abstracted away, the reality is that numbers are a quite general construct that has many distinct applications. Recall Example 1.1.5 for instance.

Definition 1.1.17. Now that we have a construction for the naturals, we can define some arithmetic operations:

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

Example 1.1.8.

$$\begin{aligned} \mathtt{ADD\ 2\ 3} &= (\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x))\ (\lambda f.\lambda x.f\ f\ x)\ (\lambda f.\lambda x.f\ f\ f\ x) \\ &\rightsquigarrow^*_\beta \lambda f.\lambda x.\ (\lambda f.\lambda x.f\ f\ x)\ f\ ((\lambda f.\lambda x.f\ f\ f\ x)\ f\ x) \\ &\rightsquigarrow^*_\beta \lambda f.\lambda x.\ (\lambda x.f\ f\ x)\ ((\lambda x.f\ f\ f\ x)\ x) \\ &\rightsquigarrow_\beta (\lambda f.\lambda x.\ f\ f\ f\ f\ f\ x) \\ &\equiv \mathtt{5} \quad \text{Can I use alpha equiv o is it confusing?} \end{aligned}$$

Multiplication works similarly to addition. As for `ISZERO`, the intuition behind it is that if the number is zero, then $\lambda x.\,\mathtt{FALSE}$ is dropped, and the result is `TRUE`; otherwise, it is applied at least once, which immediately yields `FALSE`.

Remark 8. This proves that the λ-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 constructions in line are the truth values of boolean logic, "*true*" and "*false*" for short.

Definition 1.1.18. Boolean truth values:

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

Remark 9. Note how $\texttt{TRUE}\ M\ N \rightsquigarrow_\beta M$ and $\texttt{FALSE}\ M\ N \rightsquigarrow_\beta N$

Definition 1.1.19. 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 similarly to a branching statement whenever we pass a church encoded boolean as a first argument.

Example 1.1.9. 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 9. From this example we extract that $\lambda x.x$ is equivalent to IF it terms of behaviour, and thus, we could use $\texttt{IF} \equiv \lambda x.x$ interchangeably.

Remark 10. 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 1.1.20. $\texttt{PAIR} \equiv \lambda x.\lambda y.\lambda f.f\ x\ y$

Remark 11. This is more convenient since $\texttt{PAIR}\, M\, N \twoheadrightarrow_\beta^* \lambda f.f\, M\, N$, 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:

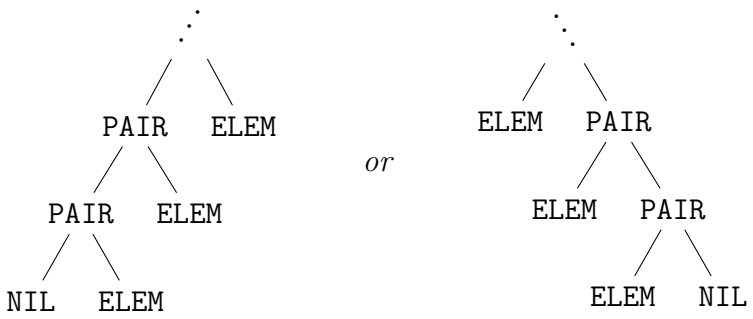
Definition 1.1.21.

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

Proposition 1.1.3. A key property of PAIR's is:

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

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



Definition 1.1.22. 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 2. This notion of inductively defined lists is lays the foundations of the **LISP** family of programming languages, hence the name (LISt Processing).

Definition 1.1.23. Relevant list operators:

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

Turing completeness of the λ-calculus

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

Recursion:

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

1.1.3 Fixed Point Combinators and Recursion

Reintroduce the question from before From the Church Rosser Property of confluence, we know that whenever a term has a normal form, this normal form must be unique. We still have not given an answer to whether all λ-terms have a normal form.

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 β-reduction chain so to speak. In more formal terms this means there exists some M such that $M \rightsquigarrow_\beta^n M$ with $n > 0$. It turns out $\Omega \equiv (\lambda x.\, x\, x)(\lambda x.\, x\, x)$ fullfills this requirement. Let us inspect what goes on while β-reducing this term:

$$\Omega \equiv (\lambda x.\, x\, x)(\lambda x.\, x\, x) \rightsquigarrow_\beta (\lambda x.\, x\, x)(\lambda x.\, x\, x) \rightsquigarrow_\beta^* (\lambda x.\, x\, x)(\lambda x.\, x\, x) \equiv \Omega$$

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

There is one caveat to this, that classically, recursion is achieved through naming, take for instance this definition for the gdc:

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

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

Definition 1.1.24. Let $f : X \to X$ be a function. An element $x \in X$ is called a fixed point of f if $f(x) = x$.

Example 1.1.10. Some simple functions that illustrate what fixed points are:

Function	*Fixed Point(s)*
$f(x) = c$	$x = c$
$g(x) = x$	$\{x \mid x \in \mathbb{R}\}$
$h(x) = x^2$	$x = 0,\ x = 1$

This harmless idea 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 λ-world. maybe use circ (i.e function composition) in the chain since it might have some meaning in the context of categories.

Definition 1.1.25. Given a lambda term $L \in \mathbf{\Lambda}$, a term $M \in \mathbf{\Lambda}$ is called a fixed point of L if $M =_\beta LM$.

No surprises here, the definition is close to a one-to-one mapping of its functional counterpart seen previously. Interestingly enough, it turns out that every lambda term has a fixed point, as stated in the theorem proven below:

Theorem 1.1.4. 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:

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

Hence, $M =_\beta LM$, as required. □

Remark 12. See how LM is the application of M to L and that using AppR we can keep β-reducing M the same way we did for $x = f(x) = f(f(x)) \ldots$ only this time using lambda terms:

$$M \rightsquigarrow_\beta L\,M \rightsquigarrow_\beta L\,L\,M \rightsquigarrow_\beta^* L^n M.$$

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

Definition 1.1.26. Y Combinator. The term that given some other term returns its fixed point.

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

Note 3. $Y \equiv \lambda L.M$ M defined previously. Improve this, find some other way to put it? note it is an improved Ω

Remark 13. Even though the Y combinator is of theoretical interest since it allows us to find recursion within the λ-calculus, it is of little practical use since it always runs into non-termination e.g.

$$\begin{aligned} Y\,F &\equiv (\lambda f.\,(\lambda x.\,f\,(x\,x))(\lambda x.\,f\,(x\,x)))F \\ &\rightsquigarrow_\beta (\lambda x.\,F\,(x\,x))(\lambda x.\,F\,(x\,x)) \\ &\rightsquigarrow_\beta F((\lambda x.\,F\,(x\,x))(\lambda x.\,F\,(x\,x))) \\ &\rightsquigarrow_\beta F(F((\lambda x.\,F\,(x\,x))(\lambda x.\,F\,(x\,x)))) \\ &\rightsquigarrow_\beta \cdots \end{aligned}$$

This is due to its lazy nature, this means that the argument F is not evaluated until Y is done unravelling i.e. never.

To work arround this issue, several other combinators exist, one of the most notorious, the Z combinator is defined next:

Definition 1.1.27. Z combinator

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

Example 1.1.11.

$$\begin{aligned} Z\,F &\equiv (\lambda f.\,(\lambda x.\,f\,(\lambda v.\,x\,x\,v))\,(\lambda x.\,f\,(\lambda v.\,x\,x\,v)))\,F \\ &\rightsquigarrow_\beta (\lambda x.\,F\,(\lambda v.\,x\,x\,v))\,(\lambda x.\,F\,(\lambda v.\,x\,x\,v)) \\ &\rightsquigarrow_\beta F\,(\lambda v.\,(\lambda x.\,F\,(\lambda v.\,x\,x\,v))\,(\lambda x.\,F\,(\lambda v.\,x\,x\,v))\,v) \\ &\equiv F\,(\lambda v.\,Z\,F\,v) \end{aligned}$$

1.2 Simply Typed λ-Calculus

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 $\neq$ 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.