Chapter 1

λ-Calculus

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 [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 1.1.1. The set of lambda terms Λ is defined inductively as follows:

- (Variable) If $x \in V$, then $x \in \Lambda$.

- (Abstraction) If $x \in V$ and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$.

- (Application) If $M, N \in \Lambda$, then $(MN) \in \Lambda$.

Where $V = \{x, y, z, ...\}$ 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 1.1.2.

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 Λ:

- 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 Λ, 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 1.1.2. Taking advantage of BNF notation, an alternative definition for Λ (1.1.1):

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

where $\Sigma = \{\lambda, ., (,)\} \cup V$ and $\Lambda \subset \Sigma^*$.

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 1.1.9

Example 1.1.1. Some examples of valid λ terms generated using the grammar in 1.1.2: y, $(\lambda x.(xx))$, $(\lambda x.(\lambda y.x))$, $(((\lambda x.(xy))(\lambda y.y))z)$.

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 straight-forward, the formal definition does not leave precedence under-specified thanks to parenthesis.

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

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

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

Example 1.1.2. Compute the free variables of $\lambda x.\lambda y.yxz$:
    The term is indeed well-formed and thus belongs to Λ, allowing us to 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 1.1.4. A term $M \in \Lambda$ 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 1.1.5. Given $T \in \Lambda$, $\mathbf{Sub} : \Lambda \to \mathcal{P}(V)$ 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 $x \in \mathbb{V}$ and $M, N \in \Lambda$.

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

Definition 1.1.7. 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 1.1.8. Capture-Avoiding Substitution Rules Let $M, N \in \Lambda$ and $x, y, z \in V$. 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 $z \notin \mathbf{FV}(M) \cup \mathbf{FV}(N)$.

Beta reduction

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:

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

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

1.1.2 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.

Definition 1.1.9. Representation of the set of all naturals, usually referred to as Church numerals:

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

Remark 3. We defined the church numerals similarly to the peano axioms: by bringing into existance an initial element, and using the succesor function to define the rest of them. Let us apply $\texttt{SUCC}$ a few times to illustrate:

$$0 \equiv \lambda f.\lambda x.x$$
$$1 \equiv \lambda f.\lambda x.fx$$
$$2 \equiv \lambda f.\lambda x.ffx$$
$$3 \equiv \lambda f.\lambda x.fffx$$
$$\texttt{n} \equiv \lambda f.\lambda x.f^n x$$

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:

$$\texttt{ADD} \equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$
$$\texttt{MUL} \equiv \lambda f.\lambda x.\lambda f.\lambda x.m(nf)x$$
$$\texttt{ISZERO} \equiv \lambda n.n\,(\lambda x.\texttt{FALSE})\,\texttt{TRUE}$$

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 1.1.10. Boolean truth values:

$$\texttt{TRUE} \equiv \lambda t.\lambda f.t$$
$$\texttt{FALSE} \equiv \lambda t.\lambda f.f$$

Remark 4. Note how $\mathtt{TRUE}\ M\ N \leadsto_\beta M$ and $\mathtt{FALSE}\ M\ N \leadsto_\beta N$

Definition 1.1.11. Conditional, in other words *"if statement"*.

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

The reality of the $\mathtt{IF}$ 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 1.1.3. On the correct behabiour of $\mathtt{IF}$ i.e. check that it implements branching. $M, N \in \Lambda$ and $\mathtt{IF}, \mathtt{TRUE}, \mathtt{FALSE}$ are as usual:

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

$$\begin{aligned}
&\mathtt{IF}\ \mathtt{FALSE}\ M\ N \\
&\leadsto^*_\beta \mathtt{FALSE}\ M\ N \leadsto_\beta N
\end{aligned}$$

So the $\mathtt{IF}$ 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.

How can we build every logic gate thus every logical function using if true and false. add reference to normalisation of logical propositions

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

Definition 1.1.12. $\mathtt{PAIR} \equiv \lambda x.\lambda y.\lambda f.f\,x\,y$
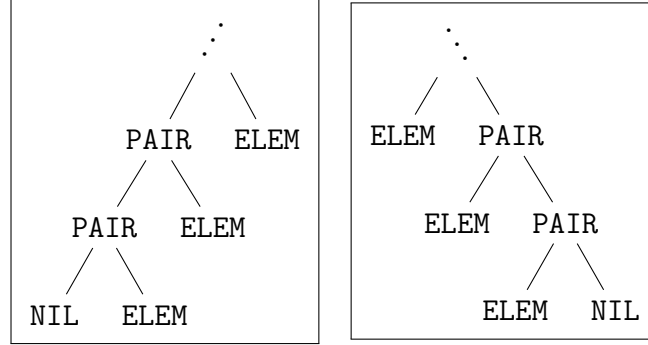
From this definition follows that $\mathtt{FST}$ and $\mathtt{SND}$ can be used to retrieve the first and second elements of the tuple:

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

A key property of $\mathtt{PAIR}$'s is:

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

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

Definition 1.1.13.

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

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

left blank

Recursion:

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

1.1.3   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

1.2   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 $\neq$
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