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: $(\lambda x.x + 1)(2) \rightsquigarrow (2 + 1) \rightsquigarrow 3$. 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.

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:

[1] this is an e.g., $(\lambda x.x + 1)$ is not a valid lambda term, see Definition 1.1.2.

- 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. From the definition above we extract that $\lambda x.x + 1 \notin \Lambda$, and thus, we need to find an accurate representation for numbers within this definition. See Definition **??**

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.

Having defined the language of the λ-calculus, we move on to the mechanics of computation. We will encounter results concerning the equivalence of λ-terms. 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:

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

Where $x \in V$ 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.

a

A key aspect of the λ-calculus, is what in the jargon is 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:

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

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.

1.1.1 Lambda Terms

Definition 1.1.5. (Sub; Multiset of Subterms): Since every lambda term is made up of other smaller lambda terms, it is only natural to define the set of 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 $M, N \in \Lambda$.

Definition 1.1.6. (Proper Subterm): 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 multiset of proper subterms.

Sean Var un conjunto contable de variables $x, y, z, \ldots$. La sintaxis del cálculo lambda no tipado se define como sigue:

$$M ::= x \mid (M\ N) \mid \lambda x.M$$

El conjunto de variables libres $\mathrm{FV}(M) \subseteq \mathsf{Var}$ se define por:

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

La conversión alfa (α-conversión) permite renombrar variables ligadas. Si $y \notin \mathrm{FV}(M)$, entonces:

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

La relación de equivalencia alfa $\equiv_\alpha$ es la menor relación de equivalencia cerrada por:

[label=–]$x \equiv_\alpha x$ $M\ N \equiv_\alpha M'\ N'$ si $M \equiv_\alpha M'$ y $N \equiv_\alpha N'$ $\lambda x.M \equiv_\alpha \lambda y.M[x := y]$ si $y \notin \mathrm{FV}(M)$

La sustitución $M[x := N]$ se define por inducción estructural en M, asegurando evitación de captura:

$$\begin{aligned} x[x := N] &= N \\ y[x := N] &= y \quad \text{si } y \neq x \\ (M_1\ M_2)[x := N] &= M_1[x := N]\ M_2[x := N] \\ (\lambda y.M')[x := N] &= \begin{cases} \lambda y.M' & \text{si } y = x \\ \lambda y.M'[x := N] & \text{si } y \neq x \text{ y } y \notin \mathrm{FV}(N) \\ \lambda z.(M'[y := z])[x := N] & \text{si } y \neq x,\ y \in \mathrm{FV}(N),\ z \notin \mathrm{FV}(M') \cup \mathrm{FV}(N) \end{cases} \end{aligned}$$

La reducción beta se define por la regla:

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

Y se extiende por clausura contextual:

$$\begin{aligned} M \to_\beta M' &\Rightarrow M\ N \to_\beta M'\ N \\ N \to_\beta N' &\Rightarrow M\ N \to_\beta M\ N' \\ M \to_\beta M' &\Rightarrow \lambda x.M \to_\beta \lambda x.M' \end{aligned}$$

La clausura reflexiva y transitiva de $\to_\beta$ se denota $\to_\beta^*$.

Como ejemplo, sea el término $(\lambda x.\lambda y.x)\ y$. Aplicamos reducción beta:

$$(\lambda x.\lambda y.x)\ y \to_\beta (\lambda y.x)[x := y]$$

Existe riesgo de captura, ya que y es variable ligada. Realizamos conversión alfa:

$$\lambda y.x \equiv_\alpha \lambda z.x$$

Ahora la sustitución es segura:

$$(\lambda z.x)[x := y] = \lambda z.y$$

El resultado correcto es:

$$(\lambda x.\lambda y.x)\ y \to_\beta \lambda z.y$$

Finalmente, resumimos las definiciones en la siguiente tabla:

•

Concepto	**Definición Formal**
Renombramiento	$\lambda x.M \to \lambda y.M[x := y]$, con $y \notin \mathrm{FV}(M)$
-conversión	$\lambda x.M \equiv_\alpha \lambda y.M[x := y]$
Sustitución	$M[x := N]$: recursiva, libre de captura
-reducción	$(\lambda x.M)\ N \to_\beta M[x := N]$

1.1.2 Some Important Constructs

Church numerals:

$$\begin{aligned}0 &\equiv \lambda f.\lambda x.x\\ 1 &\equiv \lambda f.\lambda x.f\,x\\ 2 &\equiv \lambda f.\lambda x.f\,(f\,x)\\ n &\equiv \lambda f.\lambda x.f^n\,x\end{aligned}$$

Booleans:

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

Conditional:

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

Pairs:

$$\begin{aligned}\texttt{PAIR} &\equiv \lambda x.\lambda y.\lambda f.f\,x\,y\\ \texttt{FIRST} &\equiv \lambda p.p\,(\lambda x.\lambda y.x)\\ \texttt{SECOND} &\equiv \lambda p.p\,(\lambda x.\lambda y.y)\end{aligned}$$

Lists:

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

Arithmetic:

$$\begin{aligned}\texttt{SUCC} &\equiv \lambda n.\lambda f.\lambda x.f\,(n\,f\,x)\\ \texttt{ADD} &\equiv \lambda m.\lambda n.\lambda f.\lambda x.m\,f\,(n\,f\,x)\\ \texttt{MUL} &\equiv \lambda m.\lambda n.\lambda f.m\,(n\,f)\\ \texttt{ISZERO} &\equiv \lambda n.n\,(\lambda x.\texttt{FALSE})\,\texttt{TRUE}\\ \texttt{PRED} &\equiv \lambda n.\lambda f.\lambda x.n\,(\lambda g.\lambda h.h\,(g\,f))\,(\lambda u.x)\,(\lambda u.u)\\ \texttt{SUB} &\equiv \lambda m.\lambda n.n\,\texttt{PRED}\,m\end{aligned}$$

Recursion:

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

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

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

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.

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