A Report of *Type Theory and Formal Proof*

Juan Pablo Royo Sales

Universitat Politècnica de Catalunya

February 12, 2021

Contents

1 Introduction

This report is going to provide a summary over the book [NG14]. Alongside the different chapters of the book I am going to describe briefly the most important parts of each chapter and, at the same time, I am going to solve 1 or 2 of the exercises proposed by the authors.

The organization of the report is going to be the same as the chapters of the book.

2 Untyped lambda calculus

In this first chapter the authors define and describe Lambda Calculus (λ-*calculus*) system which encapsulates the formalization of basic aspects of mathematical functions, in particular construction and use. In λ-*calculus* formalization system there are *typed* and *untyped* formalization of the same system. In this first case authors introduced the first basic and simple formalization which is *untyped*.

2.1 Definition

There are *two constructions principles* and *one evaluation rule*

Construction principles:

- *Abstraction:* Given an expression M and a variable x we can construct the expression: $\lambda x.M$. This is abstraction of x over M Example: $\lambda y.(\lambda x.x - y)$ Abstraction of y over $\lambda x.x - y$
- *Application:* Given 2 expressions M and N we can construct the expression: $M\ N$. This is the application of M to N. Example: $(\lambda x.x^2+1)(3)$ Application of 3 over $\lambda x.x^2 + 1$

Evaluation Rule: Formalization of this process is called Beta Reduction (β-*reduction*). β-*reduction*: An expression $(\lambda x.M)N$ can be rewritten to $M[x := N]$, which means every x should be replaced by N in M. This process is called β-*reduction* of $(\lambda x.M)N$ to $M[x := N]$.

Example: $(\lambda x.x^2 + 1)(3)$ reduces to $(x^2 + 1)[x := 3]$, which is $3^2 + 1$.

In this book, functions on λ-*calculus* notation are *Curried*.

2.1.1 Lambda-terms

Expressions in λ-*calculus* are called Lambda Terms (λ-*term*)

Definition 2.1. *The set Λ of all λ-term*

1. (Variable) If $u \in V$, then $u \in \Lambda$
   Example: x, y, z
2. (Application) If M and $N \in \Lambda$, then $(MN) \in \Lambda$
   Example: (xy), $(x(xy))$
3. (Abstraction) If $u \in V$ and $M \in \Lambda$, then $(\lambda u.M) \in \Lambda$
   Example: $(\lambda x.(xz))$, $(\lambda y.(\lambda z.x))$

Definition 2.2. *Multiset of subterms Sub*

1. (Basis) $Sub(x) = \{x\}$, for each $x \in V$
2. (Application) $Sub((MN)) = Sub(M) \cup Sub(N) \cup \{(MN)\}$
3. (Abstraction) $Sub((\lambda x.M)) = Sub(M) \cup \{(\lambda x.M)\}$

Lemma 2.1. *(1) (Reflexivity) For all λ-term M, we have $M \in Sub(M)$. (2) (Transitivity) If $L \in Sub(M)$ and $M \in Sub(N)$, then $L \in Sub(N)$.*

Definition 2.3 (Proper subterm). *L is a proper subterm of M if L is a subterm of M, but $L \not\equiv M$*

- Parenthesis can be omitted
- Application is lef-associative, MNL is $((MN)L)$
- Application takes precedence over Abstraction

2.2 Free and bound variables

Variables can be *free*, *bound* and *binding*. A variable x which is *free* in M becomes *bound* in $\lambda x.M$. M is called a *binding* variable occurrence.

Definition 2.4 (FV, set of free variables of a λ-*term*)**.**

1. (Variable) $FV(x) = \{x\}$
2. (Application) $FV(MN) = FV(M) \cup FV(N)$
3. (Abstraction) $FV(\lambda x.M) = FV(M) \setminus \{x\}$

Definition 2.5 (Closed λ-*term*; combinator; Λ^0)**.** *The λ-term M is closed if $FV(M) = \emptyset$. This is also called a combinator. The set of all closed λ-term is denoted by Λ^0*

2.2.1 Alpha conversion

It is based on the possibility of renaming bound and binding variables.

Definition 2.6 (Renaming; $M^{x \to y}$; $=_\alpha$)**.** *Let $M^{x \to y}$ denote the result of replacing every free ocurrence of x in M by y. Renaming, expressed by $=_\alpha$ is defined as: $\lambda x.M =_\alpha \lambda y.M^{x \to y}$, provided that $y \notin FV(M)$ and y is not binding in M*

Definition 2.7 (α-convertion or α-equivalence; $=_\alpha$)**.**

1. (Renaming) same as 2.6
2. (Compatibility) If $M =_\alpha N$, then $ML =_\alpha NL$, $LM =_\alpha LN$ and, for any arbitrary z, $\lambda z.M =_\alpha \lambda z.N$
3. (Reflexivity) $M =_\alpha M$
4. (Symmetry) If $M =_\alpha N$ then $N =_\alpha M$
5. (Transitivity) If both $L =_\alpha M$ and $M =_\alpha N$, then $L =_\alpha N$

2.3 Substitution

Definition 2.8 (Substitution)**.**

1. $x[x := N] \equiv N$
2. $y[x := N] \equiv y$ if $x \not\equiv y$
3. $(PQ)[x := N] \equiv (P[x := N])(Q[x := N])$
4. $(\lambda y.P)[x := N] \equiv \lambda z.(P^{y \to z}[x := N])$, if $\lambda z.P^{y \to z}$ is α-variant of $\lambda y.P$ such that $z \notin FV(N)$

2.4 Beta reduction

Definition 2.9 (One-step β-reduction, $\to_\beta$)**.**

1. (Basis) $(\lambda x.M)N \to_\beta M[x := N]$,
2. (Compatibility) If $M \to_\beta N$, then $ML \to_\beta NL$, $LM \to_\beta LN$ and $\lambda x.M \to_\beta \lambda x.N$

In 1 the left part of $\rightarrow_\beta$ is called *redex* (reducible expression), and the right side is called *contractum* (of the redex).

Definition 2.10 (β-reduction (zero-or-more-step), $\twoheadrightarrow_\beta$)**.** *$M \twoheadrightarrow_\beta N$ if there is an $n \geq 0$ and there are terms M_0 to M_n such that $M_0 \equiv M$, $M_n \equiv N$ and for all $i, 0 \leq i < n$:*
$M_i \rightarrow_\beta M_{i+1}$

Hence, if $M \twoheadrightarrow_\beta N$, there exists a chain of single-step β-reductions, starting with M and ending with N:

$M \equiv M_0 \rightarrow_\beta M_1 \rightarrow_\beta M_2 \rightarrow_\beta \ldots \rightarrow_\beta M_{n-2} \rightarrow_\beta M_{n-1} \rightarrow_\beta M_n \equiv N$

Definition 2.11 (β-conversion, β-equality; $=_\beta$)**.** *$M =_\beta N$ if there is an $n \geq 0$ and there are terms M_0 to M_n such that $M_0 \equiv M$, $M_n \equiv N$ and for all $i, 0 \leq i < n$:*
either $M_i \rightarrow_\beta M_{i+1}$ or $M_{i+1} \rightarrow_\beta M_i$

2.5 Fixed Point Theorem

Theorem 2.1. *For all $L \in \Lambda$ there is $M \in \Lambda$ such that $LM =_\beta M$*

Proof. For given L, define $M := (\lambda x.L(xx))(\lambda x.L(xx))$ This M is a redex, so we have:

$$\begin{aligned} M &\equiv (\lambda x.L(xx))(\lambda x.L(xx)) && \text{(1a)} \\ &\rightarrow_\beta L((\lambda x.L(xx))(\lambda x.L(xx))) && \text{(1b)} \\ &\equiv LM && \text{(1c)} \end{aligned}$$

Therefore, $LM =_\beta M$ □

2.6 Exercises

2.6.1 1.10 Church numerals

Having that:

- *zero* $:= \lambda fx.x$
- *one* $:= \lambda fx.fx$
- *two* $:= \lambda fx.f(fx)$
- *add* $:= \lambda mnfx.mf(nfx)$
- *mult* $:= \lambda mnfx.m(nf)x$

(a). *Show that:* $(add\ one\ one\ \twoheadrightarrow_\beta\ two)$

Proof. Replacing by lambda expressions

$$\begin{aligned}
add\ one\ one &:= (\lambda mnfx.mf(nfx))(\lambda fx.fx)(\lambda fx.fx) && (2a)\\
&\rightarrow_\beta (\lambda nfx.(\lambda fx.fx)f(nfx))(\lambda fx.fx) && (2b)\\
&\rightarrow_\beta (\lambda fx.(\lambda fx.fx)f((\lambda fx.fx)fx)) && (2c)\\
&\rightarrow_\beta (\lambda fx.(\lambda fx.fx)f(fx)) && (2d)\\
&\rightarrow_\beta (\lambda fx.f(fx)) && (2e)\\
&:= two && (2f)
\end{aligned}$$

□

(b). *Show that:* $(add\ one\ one\ \neq_\beta\ mult\ one\ zero)$

Proof. We need to reduce $(mult\ one\ zero)$ and show that is not two

$$\begin{aligned}
mult\ one\ zero &:= (\lambda mnfx.m(nf)x)(\lambda fx.fx)(\lambda fx.x) && (3a)\\
&\rightarrow_\beta (\lambda nfx.(\lambda fx.fx)(nf)x)(\lambda fx.x) && (3b)\\
&\rightarrow_\beta (\lambda fx.(\lambda fx.fx)((\lambda fx.x)f)x) && (3c)\\
&\rightarrow_\beta (\lambda fx.(\lambda x.((\lambda fx.x)f)x)x) && (3d)\\
&\rightarrow_\beta (\lambda fx.(\lambda x.(\lambda x.x)x)x) && (3e)\\
&\rightarrow_\beta (\lambda fx.(\lambda x.x)x) && (3f)\\
&\rightarrow_\beta (\lambda fx.x) && (3g)\\
&:= zero && (3h)
\end{aligned}$$

□

2.6.2 1.11 - Successor

Having that $suc := \lambda mfx.f(mfx)$. Check the following

(a). $suc\ zero =_\beta one$

Proof.

$$\begin{aligned}
suc\ zero &=_\beta (\lambda mfx.f(mfx))(\lambda fx.x) && (4a)\\
&\rightarrow_\beta (\lambda fx.f((\lambda fx.x)fx)) && (4b)\\
&\rightarrow_\beta (\lambda fx.f((\lambda x.x)x)) && (4c)\\
&\rightarrow_\beta (\lambda fx.fx) && (4d)\\
&:= one && (4e)
\end{aligned}$$

□

(b). *suc one* $=_\beta$ *two*

Proof.

$$\begin{aligned} suc\ one\ &=_\beta (\lambda mfx.f(mfx))(\lambda fx.fx) && (5a)\\ &\rightarrow_\beta (\lambda fx.f((\lambda fx.fx)fx)) && (5b)\\ &\rightarrow_\beta (\lambda fx.f((\lambda x.fx)x)) && (5c)\\ &\rightarrow_\beta (\lambda fx.f(fx)) && (5d)\\ &:= two && (5e) \end{aligned}$$

□

2.6.3 1.12 - If then else

The term 'If x then u else v' is represented by $\lambda x.xuv$. Check this by calculating β-normal forms of $(\lambda x.xuv)$true and $(\lambda x.xuv)$false, having that:

- *true* $:= \lambda xy.x$
- *false* $:= \lambda xy.y$

$(\lambda x.xuv)$*true.*

$$\begin{aligned} &:= (\lambda x.xuv)(\lambda xy.x) && (6a)\\ &\rightarrow_\beta (\lambda xy.x)uv && (6b)\\ &\rightarrow_\beta (\lambda y.u)v && (6c)\\ &\rightarrow_\beta u && (6d)\\ & && (6e) \end{aligned}$$

□

$(\lambda x.xuv)$*false.*

$$\begin{aligned} &:= (\lambda x.xuv)(\lambda xy.y) && (7a)\\ &\rightarrow_\beta (\lambda xy.y)uv && (7b)\\ &\rightarrow_\beta (\lambda y.y)v && (7c)\\ &\rightarrow_\beta v && (7d)\\ & && (7e) \end{aligned}$$

□

3 Simply typed lambda calculus

In this chapter authors introduce ***Types*** to λ-*calculus* Formalization system. When we are acting on mathematical functions, the natural thing is to restrict over some domain, both the image and the pre-image. The addition of types to the formalization system prevents some anomalies that are present in the regular λ-*calculus* model.

3.1 Simple types

It is done adding type *variables* with an infinite set $\mathbb{V} = \{\alpha, \beta, \gamma, \dots\}$

Definition 3.1 (The set $\mathbb{T}$ of all simple types).

1. (Type variable) If $\alpha \in \mathbb{V}$, then $\alpha \in \mathbb{T}$
2. (Arrow type) If $\sigma, \tau \in \mathbb{T}$, then $(\sigma \to \tau) \in \mathbb{T}$

Also, $\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$.

Parenthesis in *arrow types* are right-*associative*

3.1.1 Remarks

- *Type variable* represent simple types like *Nat, Lists, etc.*
- *Arrow types* represent functions such as $nat \ \to \ real$
- *'term M has type σ'* (typing statement) is represented as $M : \sigma$
- *'variable x has type σ'* is represented as $x : \sigma$
- If $x : \sigma$ and $x : \tau$ then $\sigma \equiv \tau$
- *Application*: If $M : \sigma \to \tau$ and $N : \sigma$, then $MN : \tau$
- *Abstraction*: If $x : \sigma$ and $M : \tau$, then $\lambda x.M : \sigma \to \tau$

3.2 Church-typing and Curry-typing

3.2.1 Typing à la Church

Unique type for each variable upon its introduction [Chu40].

Example: If x has type $\alpha \to \alpha$ and y has type $(\alpha \to \alpha) \to \beta$, then yx has type β.

If z has type β and u has type γ, then $\lambda zu.z$ has type $\beta \to \gamma \to \beta$. Therefore application $(\lambda zu.z)(yx)$ is permitted.

3.2.2 Typing à la Curry

Not give the types of variables, leave them *implicit*, therefore is called *implicit typing*.

Example: Suppose we have $M \equiv (\lambda zu.z)(yx)$ but types are not given. Guessing we have $\lambda zu.z$ should have some type $A \to B$, so (yx) must be of type A, then M is of type B. If we continue with the guessing assigning type variables after replacing we end up with the same expression as explicit typing.

Most of the book use *Typing a la Church* because in math and logic types are usually fixed and known beforehand.

3.3 Derivation rules for Church's $\lambda \to$

Definition 3.2 (Pre-typed λ-*term*, $\Lambda_{\mathbb{T}}$).

$$\Lambda_{\mathbb{T}} = V \mid (\Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}}) \mid (\lambda V : \mathbb{T}.\Lambda_{\mathbb{T}}) \tag{8}$$

We want to express things like 'λ-*term* M has type σ' relative to context Γ

Definition 3.3 (Statement, declaration, context, judgement).

1. **Statement**: $M : \sigma$, where $M \in \Lambda_{\mathbb{T}}$ and $\sigma \in \mathbb{T}$. M is called *subject* and σ *type*
2. **Declaration**: Is a statement with a *variable* as subject. Example $x : \alpha \to \beta$ç
3. **Context**: List of Declarations with different subjects
4. **Judgement**: $\Gamma \vdash M : \sigma$, where Γ is a *Context* and $M : \sigma$ is a *Statement*.

Definition 3.4 (Derivation rules for $\lambda \to$).

(var) $\Gamma \vdash x : \sigma$ if $x : \sigma \in \Gamma$

$$(appl)\ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$(abst)\ \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau}$$

This rules are ***universal***.

Definition 3.5 (Legal $\lambda\rightarrow$-terms)**.** *A pre-typed term M in $\lambda\rightarrow$ is called **legal** if there exist a context Γ and type ρ such that $\Gamma \vdash M : \rho$*

3.4 Flag notation

Flag notation is a succinct and useful way to represent Derivation rules on Typed-λ-*calculus*. It is represented using a *flag* (rectangular box) as a declaration, and everything that is bellow and attached to this *flag* are statements that belong to it. This is also called *flag pole*. Lets see an example of derivation:

Having the following derivation:

$$\cfrac{\cfrac{\cfrac{y:\alpha\rightarrow\beta, z:\alpha\vdash y:\alpha\rightarrow\beta \qquad y:\alpha\rightarrow\beta, z:\alpha\vdash z:\alpha}{y:\alpha\rightarrow\beta, z:\alpha\vdash yz:\beta}}{y:\alpha\rightarrow\beta\vdash\lambda z:\alpha.yz:\alpha\rightarrow\beta}}{\emptyset\vdash\lambda y:\alpha\rightarrow\beta.\lambda z:\alpha.yz:(\alpha\rightarrow\beta)\rightarrow\alpha\rightarrow\beta}$$

References

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[NG14] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof*. Cambridge University Press, Cambridge CB2 8BS, United Kindom, 2014.