A Report of *Type Theory and Formal Proof*

Juan Pablo Royo Sales

Universitat Politècnica de Catalunya

February 28, 2021

Contents

1 Introduction

This report is going to provide a summary over the first chapters of the book [NG14], which are the most relevant ones. Following chapters provide a more advanced view over Type Theory concepts.

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 $\to_\beta$ 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 $i, 0 \leq i < n$:*
$M_i \to_\beta M_{i+1}$

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

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

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 $M_i \to_\beta M_{i+1}$ or $M_{i+1} \to_\beta M_i$

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:

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

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

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

□

(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*

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

□

2.6.2 1.11 - Successor

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

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

Proof.

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

□

(b). $\textit{suc one} =_\beta \textit{two}$

Proof.

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

□

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:

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

$(\lambda x.xuv)true.$

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

□

$(\lambda x.xuv)false.$

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

□

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 $\lambda \rightarrow$

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 $x : \alpha \rightarrow \beta$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 $\lambda \rightarrow$)**.**

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

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

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

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 $\Gamma \vdash M : \rho$*

3.3.1 Example

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

3.4 Derivation formats

3.4.1 Linear format

1. $y : \alpha \to \beta, z : \alpha \ \vdash \ y : \alpha \to \beta$ *(var)*
2. $y : \alpha \to \beta, z : \alpha \ \vdash \ z : \alpha$ *(var)*
3. $y : \alpha \to \beta, z : \alpha \ \vdash \ yz : \beta$ *(appl) on 1 and 2*
4. $y : \alpha \to \beta \ \vdash \ \lambda z : \alpha.yz : \alpha \to \beta$ *(abst) on 3*
5. $\emptyset \ \vdash \ \lambda y : \alpha \to \beta.\lambda z : \alpha.yz : (\alpha \to \beta) \to \alpha \to \beta$ *(abst) on 4*

3.4.2 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:

We can translate *linear format* into *flag notation*:

(1)	$\boxed{y : \alpha \to \beta}$	
(2)	$\quad\boxed{z : \alpha}$	
(3)	$\quad y : \alpha \to \beta$	(var) on (1)
(4)	$\quad z : \alpha$	(var) on (2)
(5)	$\quad yz : \beta$	(appl) on (3) and (4)
(6)	$\lambda z : \alpha.yz : \alpha \to \beta$	(abst) on (5)
(7)	$\lambda y : \alpha \to \beta.\lambda z : \alpha.yz : (\alpha \to \beta) \to \alpha \to \beta$	(abst) on (6)

Even more succinct without *var* rule:

(1)	$\boxed{y : \alpha \to \beta}$	
(2)	$\quad\boxed{z : \alpha}$	
(3)	$\quad yz : \beta$	(appl) on (1) and (2)
(4)	$\lambda z : \alpha.yz : \alpha \to \beta$	(abst) on (3)

(5) $\lambda y : \alpha \to \beta . \lambda z : \alpha . yz : (\alpha \to \beta) \to \alpha \to \beta$ (abst) on (4)

3.5 Problems solved with judgement in Type Theory

- Well-typedness in $\lambda \to$
- Type Checking in $\lambda \to$
- Term finding in $\lambda \to$

3.5.1 Well-typedness in $\lambda \to$

Find out when a term is legal:

$? \vdash \text{ term } : ?$

We want to show that a λ-*term* M is legal or not. This is done following the derivation tree and trying to find a context Γ an a type ρ such that $\Gamma \vdash M : \rho$

In our previous example of derivation if we start checking that the term $\lambda y : \alpha \to \beta . \lambda z : \alpha . yz :?$ is legal. If we check with our flag notation from bottom up in the derivation tree, we are going to find the context in which this term is legal, but for example if that term would have been $\lambda y : \alpha \to \beta . \lambda z : \beta . yz :?$, we have not because $z : \beta$ cannot be applied to y.

3.5.2 Type Checking in $\lambda \to$

It is checking the validity of a full *judgement*. Given the following:

$x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash (\lambda z : \beta . \lambda u : \gamma . z)(yx) : \gamma \to \beta$

(1) $\boxed{x : \alpha \to \alpha}$
(2) $\quad \boxed{y : (\alpha \to \alpha) \to \beta}$
$\quad\quad \vdots$
(3) $\quad (\lambda z : \beta . \lambda u : \gamma . z)(yx) : \gamma \to \beta$

The idea is to fill the dots:

(1) $\boxed{x : \alpha \to \alpha}$
(2) $\quad \boxed{y : (\alpha \to \alpha) \to \beta}$

(3)	$\lambda z : \beta.\lambda u : \gamma.z : ?_1$	
	$\vdots$	
(4)	$yx : ?_2$	
(5)	$(\lambda z : \beta.\lambda u : \gamma.z)(yx) : \gamma \to \beta$	(appl) on (3) and (4), (?)

(1)	$\boxed{x : \alpha \to \alpha}$	
(2)	$\quad\boxed{y : (\alpha \to \alpha) \to \beta}$	
(3)	$\quad\quad\boxed{z : \beta}$	
(4)	$\quad\quad\quad\boxed{u : \gamma}$	
(5)	$\quad\quad\quad z : \beta$	(var) on (3)
(6)	$\quad\quad\lambda u : \gamma.z : \gamma \to \beta$	(abst) on (5)
(7)	$\quad\lambda z : \beta.\lambda u : \gamma.z : \beta \to \gamma \to \beta$	(abst) on (6)
(8)	$\quad yx : \beta$	(appl) on (1) and (2)
(9)	$(\lambda z : \beta.\lambda u : \gamma.z)(yx) : \gamma \to \beta$	(appl) on (7) and (8), (?)

3.5.3 Term finding in $\lambda \to$

Finding an appropriated term of certain type, in a certain context. A *term* that belongs to certain type is called ***inhabitant*** of that type.

This process is constructed starting with an empty context and exploring the situation on which the type is an expression from logic: *a proposition*. Every inhabitant then codes a *proof* of this proposition, hence declaring it to be a 'true' one.

Procedure:

- Take $A \to B \to A$ as a logical expression. This is a *tautology*
- Assume A holds.
- Assume B holds, then A holds.

(1) $\boxed{x : A}$

(2) | ⋮
| $? : B \to A$
| ⋮

(3) $\ldots A \to B \to A$ (abst) on (2)

(1) $\boxed{x : A}$
(2) | $\boxed{y : B}$
| ⋮
(3) | $? : A \to A$
(4) | $\cdots : B \to A$ (abst) on (3)
(5) $\ldots A \to B \to A$ (abst) on (4)

(1) $\boxed{x : A}$
(2) | $\boxed{y : B}$
(3) | $x : A$ (var) on (1)
(4) | $\lambda y : B.x : B \to A$ (abst) on (3)
(5) $\lambda x : A.\lambda y : B.x : A \to B \to A$ (abst) on (4)

3.6 General properties of $\lambda \to$

Definition 3.6 (Domain, `dom`, subcontext, $\subseteq$, permutation, projection, $\restriction$)**.**

1. If $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$, then the *domain* of Γ or $\texttt{dom}(\Gamma)$ is the list $(x_1, \ldots, x_n)$.
2. Γ' is a subcontext of Γ, or $\Gamma' \subseteq \Gamma$, if all declarations in Γ' occurs in Γ, in the same order.
3. Γ' is a *permutation* of Γ, if all declarations in Γ' also occurs in Γ and vice versa.
4. If Γ is a context and ϕ a set of variables, the *projection* in Γ on ϕ, or $\Gamma \restriction \phi$, is the subcontext Γ' of Γ with $\texttt{dom}(\Gamma') = \texttt{dom}(\Gamma) \cap \phi$

Lemma 3.1 (Free Variables Lemma)**.** *If* $\Gamma \vdash L : \sigma$*, then* $FV(L) \subseteq \texttt{dom}(\Gamma)$

Lemma 3.2 (Thinning, Condensing, Permutation)**.**

1. (Thinning) Let Γ' and Γ'' be contexts such that $\Gamma' \subseteq \Gamma''$. If $\Gamma' \vdash M : \sigma$, then also $\Gamma'' \vdash M : \sigma$
2. (Condensing) If $\Gamma \vdash M : \sigma$, then also $\Gamma \restriction FV(M) \vdash M : \sigma$
3. (Permutation) If $\Gamma \vdash M : \sigma$, and Γ' is a permutation of Γ, then Γ' is also a context and $\Gamma' \vdash M : \sigma$

Lemma 3.3 (Uniqueness of Types)**.** *Assume* $\Gamma \vdash M : \sigma$*, and* $\Gamma \vdash M : \tau$*, then* $\sigma \equiv \tau$

3.7 Reductions and $\lambda \rightarrow$

It is an adapted version of 2.4

$$(3)(\lambda y : \sigma.P)[x := N] \equiv \lambda z : \sigma.(P^{y \rightarrow z}[x := N]) \tag{9}$$

where $\lambda z : \sigma.P^{y \rightarrow z}$ is α-variant, such that $z \notin FV(N)$

Lemma 3.4 (Substituion Lemma)**.** *Assume* $\Gamma', x : \sigma, \Gamma'' \vdash M : \tau$ *and* $\Gamma' \vdash N : \sigma$*, then* $\Gamma', \Gamma'' \vdash M[x := N] : \tau$

Definition 3.7 (One-step β-reduction, $\rightarrow_\beta$, for $\Lambda_{\mathbb{T}}$)**.**

1. (Basis) $(\lambda x : \sigma.M)N \rightarrow_\beta M[x := N]$
2. (Compatibility) As 2

3.8 Exercises

3.8.1 2.5 Find pre-typed terms

(a)**.** $\lambda xy.x(\lambda z.y)y$

Proof. Having the following:

- Lets assume $x : \sigma \rightarrow \beta \rightarrow \gamma$, $\lambda z.y : \sigma$ and $y : \beta$
- If $z : \rho$, then $\lambda z : \rho.y : \rho \rightarrow \beta \equiv \sigma$ should hold.
- Taking the assumption $x : (\rho \rightarrow \beta) \rightarrow \beta \rightarrow \gamma$

- there is a legal term $\lambda x : (\rho \to \beta) \to \beta \to \gamma.\lambda y : \beta.x(\lambda z : \rho.y)y$ with type $((\rho \to \beta) \to \beta \to \gamma) \to \beta \to \gamma$

□

(b). $\lambda xy.x(\lambda z.x)y$

Proof. Having the following:

- Having similar assumptions as before but $\lambda z.x : \sigma$ and $y : \beta$
- If $z : \rho$, then $\lambda z : \rho.x : \rho \to \beta \to \gamma \equiv \sigma$ which does not hold.

Therefore, term is not typeable. □

3.8.2 2.9 Type checking

(a). $x : \delta \to \delta \to \alpha, y : \gamma \to \alpha, z : \alpha \to \beta \ \vdash \ \lambda u : \delta.\lambda v : \gamma.z(yv) : \delta \to \gamma \to \beta$

(1)	$\boxed{x : \delta \to \delta \to \alpha}$	
(2)	$\boxed{y : \gamma \to \alpha}$	
(3)	$\boxed{z : \alpha \to \beta}$	
(4)	$\boxed{u : \delta}$	
(5)	$\boxed{v : \gamma}$	
(6)	$yv : \alpha$	(appl) on (2) and (5)
(7)	$z(yv) : \beta$	(appl) on (3) and (6)
(8)	$\lambda v : \gamma.z(yv) : \gamma \to \beta$	(abst) on (7)
(9)	$\lambda u : \delta.\lambda v : \gamma.z(yv) : \delta \to \gamma \to \beta$	(abst) on (8)

Proof. □

(b). $x : \delta \to \delta \to \alpha, y : \gamma \to \alpha, z : \alpha \to \beta \ \vdash \ \lambda u : \delta.\lambda v : \gamma.z(xuu) : \delta \to \gamma \to \beta$

(1)	$\boxed{x : \delta \to \delta \to \alpha}$
(2)	$\boxed{y : \gamma \to \alpha}$

(3)	$\boxed{z : \alpha \to \beta}$	
(4)	$\boxed{u : \delta}$	
(5)	$\boxed{v : \gamma}$	
(6)	$xuu : \alpha$	(appl) on (1) and (4) twice
(7)	$z(xuu) : \beta$	(appl) on (3) and (6)
(8)	$\lambda v : \gamma.z(xuu) : \gamma \to \beta$	(abst) on (7)
(9)	$\lambda u : \delta.\lambda v : \gamma.z(xuu) : \delta \to \gamma \to \beta$	(abst) on (8)

Proof. □

4 Second order typed lambda calculus

What we have seen until now with *abstraction* and *application* rules in λ-*calculus* and typed-λ-*calculus*, is how ***term depends on term***. This kind of construction is called *first order* abstraction (or dependency).

In this chapter it is present another kind of dependency which is ***terms depending on types*** or *second order* operations (or dependency), giving us a new system which is called Second Order Typed λ-*calculus* (SOT λ-*calculus*)

Examples: Identity Function

- $(\lambda\alpha : *.\lambda x : \alpha.x)$ The $*$ means the Type of all Types, giving a kind to the expression. Now this term depends on a type. (Polymorphic identity function)
- $(\lambda\alpha : *.\lambda x : \alpha.x)\texttt{nat} \to_\beta \lambda x : \texttt{nat}.x$, which is the identity of `nat`
- $(\lambda\alpha : *.\lambda x : \alpha.x)\texttt{nat} \to \texttt{bool} \to_\beta \lambda x : (\texttt{nat} \to \texttt{bool}).x$, which is the identity of $\texttt{nat} \to \texttt{bool}$

Iteration

- $D_{\sigma,F}$ function mapping x in σ to $F(F(x))$
- $D_{\sigma,F}$ second iteration of $F(x)$ a.k.a $F \circ F$
- $\lambda x : \sigma.F(F(x)) \equiv D_{\sigma,F}$
- $D \equiv \lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x))$, applying Second Order.

- D is a polymorphic function.
- $D\ \mathtt{nat} \to_\beta \lambda f : \mathtt{nat} \to \mathtt{nat}.\lambda x : \mathtt{nat}.f(f(x))$

Function composition

$$\circ \equiv \lambda\alpha : *.\lambda\beta : *.\lambda\gamma : *.\lambda f : \alpha \to \beta.\lambda g : \beta \to \gamma.\lambda x : \alpha.g(f(x)) \tag{10}$$

4.1 Π-types

An expression like $\lambda\alpha : *.\lambda x : \alpha.x : * \to (\alpha \to \alpha)$, could be ambiguous if we name α with β. To avoid this ambiguity as we do in λ-*calculus* with α-conversion, we introduce a concept which is called Π-binder. In this case $\Pi\alpha : *.\alpha \to \alpha$ means that any arbitrary type α can be use in a term $\alpha \to \alpha$.

So we have $\Pi\alpha : *.\alpha \to \alpha \equiv_\alpha \Pi\beta : *.\beta \to \beta$ as an extension of α-conversion for SOT λ-*calculus*.

4.2 Second Order abstraction and application rules

Definition 4.1 (Second order abstraction rule)**.**

$$(abst_2)\ \frac{\Gamma, \alpha : *\ \vdash\ M : A}{\Gamma\ \vdash\ \lambda\alpha : *.M : \Pi\alpha : *.A}$$

Definition 4.2 (Second order application rule)**.**

$$(appl_2)\ \frac{\Gamma\ \vdash\ M : \Pi\alpha : *.A \qquad \Gamma\ \vdash\ B : *}{\Gamma\ \vdash\ MB : A[\alpha := B]}$$

4.3 The system $\lambda 2$

$\mathbb{T}2 = \mathbb{V} \mid \mathbb{T}2 \to \mathbb{T}2 \mid \Pi\mathbb{V} : *.\mathbb{T}2$

Definition 4.3 (Second order pre-typed λ-terms, $\lambda 2$-terms, $\Lambda_{\mathbb{T}2}$)**.**

$\Lambda_{\mathbb{T}2} = V \mid (\Lambda_{\mathbb{T}2}\Lambda_{\mathbb{T}2}) \mid (\Lambda_{\mathbb{T}2}\mathbb{T}2) \mid (\lambda V : \mathbb{T}2.\Lambda_{\mathbb{T}2} \mid (\lambda\mathbb{V} : *.\Lambda_{\mathbb{T}2})$

Definition 4.4 (Statement, declaration)**.**

1. Statement: Either
   - $M : \sigma$, where $M \in \Lambda_{\mathbb{T}2}$ and $\sigma \in \mathbb{T}2$
   - OR $\sigma : *$, where $\sigma \in \mathbb{T}2$
2. A Declaration is a Statement with a type variable and a term variable.

Definition 4.5 ($\lambda 2$-context; domain; $\texttt{dom}$)**.**

1. $\emptyset$ is a $\lambda 2$-context. $\texttt{dom}(\emptyset) = ()$
2. Γ is a $\lambda 2$-context, $\alpha \in \mathbb{V}$ and $\alpha \notin \texttt{dom}(\Gamma)$, then $\Gamma, \alpha : *$ is a $\lambda 2$-context. $\texttt{dom}(\Gamma, \alpha : *) = (\texttt{dom}(\Gamma), \alpha)$
3. Γ is a $\lambda 2$-context, if $\rho \in \texttt{T2}$, $\alpha \in \texttt{dom}(\Gamma)$ for all free typed variables in ρ and if $x \notin \texttt{dom}(\Gamma)$, then $\Gamma, x : \rho$ is a $\lambda 2$-context; $\texttt{dom}(\Gamma, x : \rho) = (\texttt{dom}(\Gamma), x)$

Definition 4.6 (Var-rule for $\lambda 2$)**.**

(var) $\Gamma \vdash x : \sigma$ if Γ is a $\lambda 2$-context and $x : \sigma \in \Gamma$

Definition 4.7 (Formation rule)**.**

(form) $\Gamma \vdash B : *$ if Γ is a $\lambda 2$-context, $B \in \mathbb{T}2$ and all free type variables in B are declared in Γ

4.4 Properties of $\lambda 2$

Definition 4.8 (α-conversion or α-equivalence, extended)**.**

1. (Renaming of term variable) $\lambda x : \sigma.M =_\alpha \lambda y : \sigma.M^{x \to y}$ if $y \notin FV(M)$ and y is not binding in M
2. (Renaming of type variable)
   $\lambda \alpha : *.M =_\alpha \lambda \beta : *.M[\alpha := \beta]$ if β does not occur in M,
   $\Pi \alpha : *.M =_\alpha \Pi \beta : *.M[\alpha := \beta]$ if β does not occur in M
3. (Compatibility, Reflexivity, Symmetry and Transitivity) equal to *first order*

4.5 Exercises

4.5.1 3.13 Addition and Multiplication

(a)**.** *Having* $\texttt{Add} \equiv \lambda m, n : \texttt{nat}.\lambda \alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \texttt{nat}.m \alpha f (n \alpha f x)$ *Show that simulates addition*

Proof. We have:

$$\texttt{Add One One} \equiv (\lambda m, n : \texttt{nat}.\lambda \alpha \ldots)(\lambda \alpha : * \ldots \lambda x : \alpha.fx) \ldots \tag{11a}$$
$$\twoheadrightarrow_\beta \lambda \alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(fx) \tag{11b}$$
$$\equiv \texttt{Two} \tag{11c}$$

□

(b). *Find λ2-term Mult that simulates multiplication on Nat.*

Proof. We have:

- $\texttt{Mult} \equiv \lambda m, n : \texttt{nat}.\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \texttt{nat}.m\alpha(n\alpha f)x$
- By statement of exercise 3.12 we know encoding of `One` and `Two`.
- So, $\texttt{Mult One Two} \twoheadrightarrow_\beta \lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \texttt{nat}.f(f(x)) \equiv \texttt{Two}$

□

5 Types dependent on types

Construct *generalized* types $\lambda\alpha : *.\alpha \to \alpha$. This is a function with a *type* as a value, what is called *type constructor*.

When we apply the type, we obtain the real types.

- $(\lambda\alpha : *.\alpha \to \alpha)\beta \quad \to_\beta \quad \beta \to \beta$
- $(\lambda\alpha : *.\alpha \to \alpha)\gamma \quad \to_\beta \quad \gamma \to \gamma$
- $(\lambda\alpha : *.\alpha \to \alpha)(\gamma \to \beta) \quad \to_\beta \quad (\gamma \to \beta) \to (\gamma \to \beta)$

More elaborated type constructors can be built like $\lambda\alpha : *.\lambda\beta : *.\alpha \to \beta$

The type of $\lambda\alpha : *.\alpha \to \alpha$ is $* \to *$, so $\lambda\alpha : *.\alpha \to \alpha : * \to *$

Our type variable also can be of the form $\alpha : * \to *$, so these are some valid constructors:

- $\lambda\alpha : * \to *.\alpha\gamma : (* \to *) \to *$ if $\gamma : *$
- Having identity type $\lambda\beta : *.\beta : * \to *$, we can construct $(\lambda\alpha : * \to *.\alpha\gamma)(\lambda\beta : *.\beta) : *$
- $\lambda\alpha : * \to *.\alpha : (* \to *) \to (* \to *)$

This system is called $\lambda\underline{\omega}$ or ***types depending on types***.

Abstract syntax tree: $\mathbb{K} = * \mid (\mathbb{K} \to \mathbb{K})$

New symbol for ***type of all kinds*** which is $\square$. For example $* : \square$, $* \to * : \square$

If $\mathcal{K}$ is a kind, each M of type $\mathcal{K}$ is called a type constructor or constructor.

Proper constructors are constructors which are not types.

Definition 5.1 (Constructor, proper constructor, sort)**.**

1. If $\mathcal{K} : \Box$ and $M : \mathcal{K}$, then M is a constructor. If $\mathcal{K} \not\equiv *$, then M is a proper constructor.
2. The set of ***sort*** is $\{*, \Box\}$

Definition 5.2 (Levels)**.**

1. Level 1: Terms
2. Level 2: Constructors (Proper and types)
3. Level 3: kinds
4. Level 4: $\Box$

Example: $t : \sigma : * \to * : \Box$ (Level 1 to 4 from left to right)

Definition 5.3 (Sort-rule)**.** *(sort)* $\emptyset \ \vdash \ * : \Box$

s will represent ***sort*** rule. So s is $*$ or $\Box$.

Definition 5.4 (Var-rule)**.**

$$(var) \ \frac{\Gamma \ \vdash \ A : s}{\Gamma, x : A \ \vdash \ x : A} \quad \text{If } x \notin \Gamma$$

To declare x we need to define A which is a type or a kind.

Example

	$s \equiv \Box$		$s \equiv *$	
$A : s$	$* : \Box$	$* \to * : \Box$	$\alpha : *$	$\alpha \to \beta : *$
$x : A$	$\alpha : *$	$\beta : * \to *$	$x : \alpha$	$y : \alpha \to \beta$

$$\dfrac{\dfrac{(1) \ \emptyset \ \vdash \ * : \Box}{(2) \ \alpha : * \ \vdash \ \alpha : *}\,(\text{var})}{(3) \ \alpha : *, x : \alpha \ \vdash \ x : \alpha}\,(\text{var})$$

5.1 The weakening rule in $\lambda\omega$

This rule allow us to ***weaken*** the context of a judgement.

Definition 5.5 (Weakening rule)**.**

$$(weak) \quad \frac{\Gamma \ \vdash \ A : B \qquad \Gamma \ \vdash \ C : s}{\Gamma, x : C \ \vdash \ A : B} \quad \text{if } x \notin \Gamma$$

We are adding an extra declaration $C : s$ to the context Γ, assuming that we already provided something for that context.

Thinning in Type Theory is inserting new declarations in a given list. ***Weakening*** is extending that list at the end.

Examples:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash * : \Box}}{\alpha : * \vdash \alpha : *}\,(var) \quad \dfrac{\dfrac{}{\emptyset \vdash * : \Box}}{\alpha : * \vdash \alpha : *}\,(var)}{\alpha : *, x : \alpha \vdash \alpha : *}\,(weak)$$

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash * : \Box}}{\alpha : * \vdash \alpha : *}\,(var) \quad \dfrac{\dfrac{}{\emptyset \vdash * : \Box} \quad \dfrac{}{\emptyset \vdash * : \Box}}{\alpha : * \vdash * : \Box}\,(weak)}{\alpha : *, \beta : * \vdash \alpha : *}\,(weak)$$

5.2 Formation rule in $\lambda\omega$

This rule enables to form types and kinds

Definition 5.6 (Formation rule)**.**

$$(form) \quad \dfrac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s}$$

Examples:

$$\dfrac{\dfrac{\dots}{\alpha : *, \beta : * \vdash \alpha : *} \qquad \dfrac{\dots}{\alpha : *, \beta : * \vdash \beta : *}}{\alpha : *, \beta : * \vdash \alpha \to \beta : *}\,(form)$$

$$\dfrac{\dfrac{\dots}{\alpha : * \vdash * : \Box} \qquad \dfrac{\dots}{\alpha : * \vdash * : \Box}}{\alpha : * \vdash * \to * : \Box}\,(form)$$

5.3 Application and Abstraction rule in $\lambda\omega$

$$(appl) \quad \dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$(abst) \quad \dfrac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x : A.M : A \to B}$$

Example:

	$\vdots$	
(1)	$\boxed{\beta : *}$	
(2)	$* : \Box$	(weak) on (1) and (1)
(3)	$\boxed{\alpha : *}$	
(4)	$\alpha : *$	(var) on (3)
(5)	$\alpha \to \alpha : *$	(form) on (2) and (2)
(6)	$* \to * : \Box$	(form) on (3) and (3)
(7)	$\lambda\alpha : *.\alpha \to \alpha : * \to *$	(abst) on (5) and (6)
(8)	$\beta : *$	(var)
(9)	$(\lambda\alpha : *.\alpha \to \alpha)\beta : *$	(appl) on (7) and (8)

5.4 The conversion rule

Recalling example:

$(\lambda\alpha : *.\alpha \to \alpha)\beta \to_\beta \beta \to \beta$

We want to be able to derive something like:

If M has type B and $B =_\beta B'$, then M also has type B'.

As it is, the system is to weak to derive that and because of that we need to define the conversion rule.

Definition 5.7 (Conversion rule)**.**

$$\textit{(conv)} \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \textit{if} \quad B =_\beta B'$$

Example

$$\frac{\Gamma \vdash x : (\lambda\alpha : *.\alpha \to \alpha)\beta \qquad \Gamma \vdash \beta \to \beta : *}{\Gamma \vdash x : \beta \to \beta}\ \textit{(conv)}$$

5.5 Properties in $\lambda\omega$

All the properties of other system $\lambda \rightarrow$ and $\lambda 2$ applies here but a modification on Uniqueness property needs to be made:

Lemma 5.1 (Uniqueness of Types up to Conversion)**.** *If* $\Gamma \vdash A : B_1$ *and* $\Gamma \vdash A : B_2$*, then* $B_1 =_\beta B_2$

5.6 Exercises

5.6.1 4.4. Flag derivations

(a)**.** $\alpha.*, \beta : * \rightarrow * \vdash \beta(\beta\alpha) : *$

(1)	$\boxed{\alpha : *}$	
(2)	$\quad\boxed{\beta : * \rightarrow *}$	
(3)	$\quad\quad \beta\alpha : *$	(appl)
(4)	$\quad\quad \beta(\beta\alpha) : *)$	(appl)

(b)**.** $\alpha.*, \beta : * \rightarrow *, x : \beta(\beta\alpha) \vdash \lambda y : \alpha.x : \alpha \rightarrow \beta(\beta\alpha)$

(1)	$\boxed{\alpha : *}$	
(2)	$\quad\boxed{\beta : * \rightarrow *}$	
(3)	$\quad\quad\boxed{y : \alpha}$	
(4)	$\quad\quad\quad \beta\alpha : *$	(appl)
(5)	$\quad\quad\quad \beta(\beta\alpha) : *$	(appl)
(6)	$\quad\quad\quad\boxed{x : *}$	
(7)	$\quad\quad\quad\quad x : \beta(\beta\alpha)$	(conv)
(8)	$\quad\quad\quad\quad \lambda y : \alpha.x : \alpha \rightarrow \beta(\beta\alpha)$	(abst)

5.6.2 4.5

Provide derivation in flag format of following judgement:

$\alpha : *, x : \alpha \vdash \lambda y : \alpha.x : (\lambda\beta : *.\beta \rightarrow \beta)\alpha$

(1)	$\boxed{\alpha : *}$	
(2)	$\quad\boxed{x : \alpha}$	
(3)	$\quad\quad\boxed{y : \alpha}$	
(4)	$\quad\quad x : \alpha$	(weak) on (2)
(5)	$\quad\quad \lambda y : \alpha . x : \alpha \to \alpha$	(abst) on (4)
(6)	$\quad\quad\boxed{\beta : *}$	
(7)	$\quad\quad \beta \to \beta : *$	(form) on (6) twice
(8)	$\quad\quad \lambda \beta : * . \beta \to \beta : * \to *$	(abst) on (7)
(9)	$\quad\quad (\lambda \beta : * . \beta \to \beta)\alpha : *$	(appl) on (8) and (1)
(10)	$\quad\quad \lambda y : \alpha . x : (\lambda \beta : * . \beta \to \beta)\alpha$	(conv) on (5) and (9)

6 Types dependent on terms

Until now in the previous systems described in the previous chapters we have seen:

- $\lambda \to$: *terms depending on terms*
- $\lambda 2$: *terms depending on terms + terms depending on types*
- $\lambda \underline{\omega}$: We added *types depending on types*

Now it is time to add to the system ***types depending on terms*** which is going to be called as λP, where P stands from predicate.

The format of a λP is: $\lambda x : A.M$, where M is a type and x a term-variable which type is A. This is known as a *type-valued* function or *type constructor*.

Examples:

- $(\lambda n : \texttt{nat}.S_n)3$ where S_n is the Set S_n for each $n : \texttt{nat}$
- $(\lambda n : \texttt{nat}.P_n)3$ where SP_n is the Proposition P_n for each $n : \texttt{nat}$

In both cases represents a type depending on a term. After *β-reduction* first case S_3 is the Set of all non negative multiple of 3. In the second case P_3 the proposition *3 is a prime number*.

6.1 Derivation rules of λP

$$\textit{(sort)} \quad \emptyset \vdash * : \Box$$

$$\textit{(var)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad \text{if } x \notin \Gamma$$

$$\textit{(weak)} \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \quad \text{if } x \notin \Gamma$$

$$\textit{(form)} \quad \frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s}$$

$$\textit{(appl)} \quad \frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$\textit{(abst)} \quad \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.B : s}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$$

$$\textit{(conv)} \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{if } B =_\beta B'$$

The main difference with $\lambda\underline{\omega}$ system is the introduction of Π-types in *(form), (appl) and (abst).*

- *(form)* the context is extended with x, so $\Gamma, x : A$
- *(appl)* proper type is chosen $B[x := N]$

6.2 Example derivation of λP

(1)	$* : \Box$	(sort)
(2)	$\boxed{A : *}$	
(3)	$A : *$	(var) on (1)
(4)	$* : \Box$	(weak) on (1) and (1)
(5)	$\boxed{x : A}$	
(6)	$* : \Box$	(weak) on (3) and (4)

(7) | $A \to * : \square$ (form) on (3) and (6)

6.3 Minimal predicate logic in λP

In this system we can build a *minimal predicate logic*, which only contains *implication* and *universal quantification* as logical constructs. Entities are *propositions, sets and predicates over sets.*

With this we have *propositions-as-types* and *proof-as-terms.* Both are abbreviated as **PAT**-interpretation.

- If a term b inhabits type B ($b : B$), in which B is interpreted as a proposition, then b is the *proof* of B. b is the proof object.
- Where no inhabitant of B exists, there is no *proof* of B, so B is *false.*

Definition 6.1 (PAT-interpretation)**.**

Proposition B is inhabited $\iff$ B is true;
Proposition B is not inhabited $\iff$ B is false.

6.3.1 Sets

Sets as types $S : *$, elements as terms. If a is an element of S then $a : S$.

$\texttt{nat} : *$, $\texttt{nat} \to \texttt{nat} : *$, $3 : \texttt{nat}$, $\lambda n : \texttt{nat}.n : \texttt{nat} \to \texttt{nat}$

6.3.2 Propositions

If A is a proposition, then $A : *$. A term p inhabits A is a proof of A.

6.3.3 Predicates

Predicate P is a function from a set S to the set of all propositions. $P : S \to *$. Also for each $a : S$, then $Pa : *$

6.3.4 Implication

Logical implication $A \implies B$ corresponds in types to $A \to B$.

- $A \implies B$ is true
- If A is true, then also B is true.
- If A is inhabited, then also B is inhabited.

- there is a function mapping inhabitants of A to inhabitants of B, $f : A \to B$
- $A \to B$ is inhabited

6.3.5 Universal quantification

Having $\forall_{x \in S}(P(x))$ of some predicate P:

- $\forall_{x \in S}(P(x))$ is true;
- for each $x \in S$, $P(x)$ is true;
- for each $x \in S$, Px is inhabited;
- there is a function mapping each $x \in S$ to an inhabitant of Px with type $\Pi x : S.Px$
- $f : \Pi x : S.Px$
- $\Pi x : S.Px$ is inhabited

Therefore we can encode $\forall$ in type theory with Π-type

Minimal predicate logic	The type theory of λP
S is a set A is a proposition	$S : *$ $A : *$
$a \in S$ p proves A	$a : S$ $p : A$
P is a predicate on S	$P : S \to *$
$A \implies B$ $\forall_{x \in S}(P(x))$	$A \to B$ $(= \Pi x : A.B)$ $\Pi x : S.Px$
$(\implies\text{-}elim)$ $(\implies\text{-}intro)$	*(appl)* *(abst)*
$(\forall\text{-}elim)$ $(\forall\text{-}intro)$	*(appl)* *(abst)*

6.4 Examples of derivations in λP

$\forall_{x \in S} \forall_{y \in S}(Q(x, y)) \implies \forall_{u \in S}(Q(u, u))$

(1) $\boxed{Assume : \forall_{x \in S} \forall_{y \in S}(Q(x, y))}$

(2) $\quad \boxed{Let : u \in S}$

(3)	$\forall_{y \in S}(Q(u,y))$	($\forall$-elim) on (1) and (2)
(4)	$Q(u,u)$	($\forall$-elim) on (3) and (2)
(5)	$\forall_{u \in S}(Q(u,u))$	($\forall$-intro) on (4)
(6)	$\forall_{x \in S}\forall_{y \in S}(Q(x,y)) \implies \forall_{u \in S}(Q(u,u))$	($\implies$-intro) on (5)

Coding this into λP we need to find an inhabitant for this $? : \Pi x : S.\Pi y : S.Qxy \to \Pi u : S.Quu$

(1)	$S : *$	
(2)	$Q : S \to S \to *$	
(3)	$z : \Pi x : S.\Pi y : S.Qxy$	
(4)	$u : S$	
(5)	$zu : \Pi y : S.Quy$	(appl) on (3) and (4)
(6)	$zuu : Quu$	(appl) on (5) and (4)
(7)	$\lambda u : S.zuu : \Pi u : S.Quu$	(abs) on (6)
(8)	$\lambda z : (\Pi x : S.\Pi y : S.Qxy).\lambda u : S.zuu :$ $\Pi x : S.\Pi y : S.Qxy \to \Pi u : S.Quu$	(abs) on (7)

6.5 Exercises

6.5.1 5.4

. *Prove that $*$ is the only legal kind in λP*

Proof. Lets think we can build a kind $* \to *$

- In system λP is $\Pi x : *.*$
- This should be construct with *(form)*
- Having as premise $\Gamma \vdash * : *$
- But $\Gamma \vdash * : B$ is not possible if we don't have $* : \Box$ as premise
- So, $\Gamma \vdash * : *$ is not possible
- Therefore $* \to * : \Box$ cannot be derived in Γ

□

6.5.2 5.9(a)

$\forall_{x \in S}(Q(x)) \implies \forall_{y \in S}(P(y) \implies Q(y))$

(1)	$\boxed{\forall_{x \in S}(Q(x))}$	
(2)	$\quad\boxed{y \in S}$	
(3)	$\quad Q(y)$	($\forall$-elim) on (1) nad (2)
(4)	$\quad P(y) \implies Q(y)$	($\implies$-intro) on (3) and (2)
(5)	$\forall_{y \in S}(P(y) \implies Q(y))$	$\forall$-intro on (4) and (2)
(6)	$\forall_{x \in S}(Q(x)) \implies \forall_{y \in S}(P(y) \implies Q(y))$	($\implies$-intro) on (5)

(1)	$\boxed{S : *}$
(2)	$\quad\boxed{P, Q : S \to *}$
(3)	$\quad\quad\boxed{u : \Pi x : S.Qx}$
(4)	$\quad\quad\quad\boxed{v : \Pi z : S.(Pz \to Qz)}$
(5)	$\quad\quad\quad uy : Qy$
(6)	$\quad\quad\lambda y : S.vy : \Pi y : S.Py \to Qy$
(7)	$\quad\lambda u : (\Pi x : S.Qx).\lambda y : S.vy : \Pi y : S.Py \to Qy$

7 The Calculus of Constructions

7.1 System λC

In this system we combine all the previous systems having all the possible choices *terms/types depending on terms/types*. This is called *Calculus of Constructions*, λC or λ-Coquand, after *Th. Coquand*. Letter 'c' also refer to λ-cube.

$\lambda C = \lambda 2 + \lambda \underline{\omega} + \lambda P$

The main difference in the *(form)* rule:

In λP we have

$$(form)_{\lambda P} \quad \frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s}$$

This mean that we have a type $B : s$ depending on a term $x : A$. So we can have only types or terms depending on terms only. If we relax the restriction we can have terms/types depending on terms/types. So we can put $A : s$ instead of $A : *$, but we need another s to distinguish from the other one.

$$(form)_{\lambda C} \quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}$$

Both s_1 and s_2 can be chosen independently from each other. Final type is s_2 which is inherit from the body.

- If B is a type, then the generalized type $\Pi x : A.B$ is a type as well
- If B is a kind, then $\Pi x : A.B$ should be a kind.

$x : A : s_1$	$b : B : s_2$	(s_1, s_2)	$\lambda x : A.b$	form
$*$	$*$	$(*, *)$	term-depending-on-term	$\lambda \rightarrow$
$\Box$	$*$	$(\Box, *)$	term-depending-on-type	$\lambda 2$
$\Box$	$\Box$	$(\Box, \Box)$	type-depending-on-type	$\lambda \underline{\omega}$
$*$	$\Box$	$(*, \Box)$	type-depending-on-term	λP

7.2 The λ-cube

There are 3 possible extensions to $\lambda \rightarrow$

- Terms depending on types: $\lambda 2$
- Types depending on types: $\lambda \underline{\omega}$
- Types depending on terms: λP

All three extension together forms λC. There are other possibilities of combining $\lambda \rightarrow$ with only 2 of them. All these possible combinations are called λ-cube or ***Barendregt cube*** [Bar93]

system:	combinations (s_1, s_2) allowed			
$\lambda\to$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\Box,*)$		
$\lambda\underline{\omega}$	$(*,*)$		$(\Box,\Box)$	
λP	$(*,*)$			$(*,\Box)$
$\lambda\omega$	$(*,*)$	$(\Box,*)$	$(\Box,\Box)$	
$\lambda P2$	$(*,*)$	$(\Box,*)$		$(*,\Box)$
$\lambda P\underline{\omega}$	$(*,*)$		$(\Box,\Box)$	$(*,\Box)$
$\lambda P\omega = \lambda C$	$(*,*)$	$(\Box,*)$	$(\Box,\Box)$	$(*,\Box)$

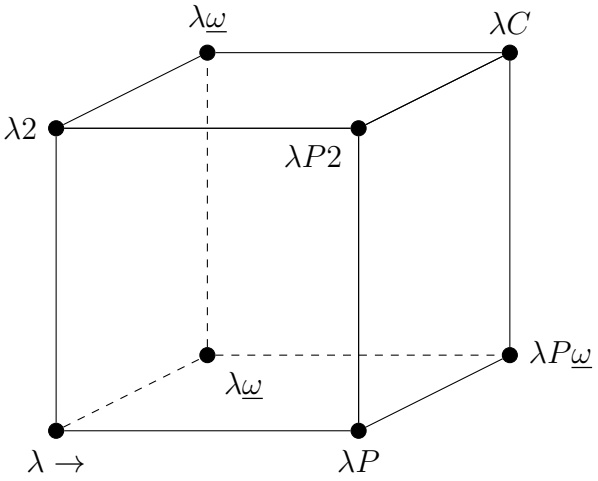

Figure 1: The λ-cube or Barendregt cube

The result investigations on [Bar93] is that all the eight systems can be described with only one set of derivations rules

$$\textit{(sort)} \quad \emptyset \vdash * : \Box$$

$$\textit{(var)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad \text{if } x \notin \Gamma$$

$$\textit{(weak)} \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \quad \text{if } x \notin \Gamma$$

$$\textit{(form)} \quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}$$

$$\textit{(appl)} \quad \frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$\textit{(abst)} \quad \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.B : s}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$$

$$\textit{(conv)} \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{if } B =_\beta B'$$

Table 20: Derivation rules for the systems of λ-cube

7.3 Properties of λC

Definition 7.1 (Expressions of λC, $\mathcal{E}$)**.** *The set of $\mathcal{E}$ of λC is defined by:*

$\mathcal{E} = V \mid \Box \mid * \mid (\mathcal{E}\mathcal{E}) \mid (\lambda V : \mathcal{E}.\mathcal{E}) \mid (\Pi V : \mathcal{E}.\mathcal{E})$

Lemma 7.1 (Free Variable Lemma)**.**

If $\Gamma \vdash A : B$, then $FV(A), FV(B) \subseteq \texttt{dom}(\Gamma)$

Definition 7.2 (Well-formed context)**.**

A context Γ is well-formed if there are A and B such that $\Gamma \vdash A : B$

Lemma 7.2 (Thinning, Permutation and Condesing Lemma)**.**

1. *(Thinning)*: Let Γ' and Γ'' be contexts such that $\Gamma' \subseteq \Gamma''$. If $\Gamma' \vdash A : B$ and Γ'' is well-formed, then also $\Gamma'' \vdash A : B$.
2. *(Permutation)*: Let Γ' and Γ'' be contexts such that Γ'' is a Permutation of Γ'. If $\Gamma' \vdash A : B$ and Γ'' is well-formed, then also $\Gamma'' \vdash A : B$.

3. *(Condensing)*: If $\Gamma', x : A, \Gamma'' \vdash B : C$ and x does not occur in Γ'', B or C , then also $\Gamma, \Gamma'' \vdash B : C$.

Lemma 7.3 (Generation Lemma)**.**

1. If $\Gamma \vdash x : C$, there exists a sort s and an expression B such that $B =_\beta C, \Gamma \vdash B : s$ and $x : B \in \Gamma$.
2. If $\Gamma \vdash MN : C$, then M has a Π-type, i.e. there exists an expression A and B such that $\Gamma \vdash M : \Pi x : A.B$; moreover, N fits in this Π-type $\Gamma \vdash N : A$ and finally $C =_\beta B[x := N]$
3. If $\Gamma \vdash \lambda x : A.b : C$, then there are a sort s and an expression B such that $C =_\beta \Pi x : A.B$ where $\Gamma \vdash \Pi x : A.B : s$ and moreover $\Gamma, x : A \vdash b : B$
4. If $\Gamma \vdash \Pi x : A.B : C$ then there are s_1 and s_2 such that $C \equiv s_2$, and moreover $\Gamma \vdash A : s_1$ and $\Gamma, x : A \vdash B : s_2$

Definition 7.3. *An expression M in λC is legal if there exists Γ and N such that $\Gamma \vdash M : N$ or $\Gamma \vdash N : M$ (typeable or inhabited)*

Lemma 7.4 (Subexpression lemma)**.** *If M is legal, then every subexpression of M is also legal.*

Lemma 7.5 (Uniqueness of Types up to Conversion)**.** *If $\Gamma \vdash A : B_1$ and $\Gamma \vdash A : B_2$, then $B_1 =_\beta B_2$*

Lemma 7.6 (Substitution Lemma)**.**

Let $\Gamma', x : A, \Gamma'' \vdash B : C$ and $\Gamma' \vdash D : A$.

Then $\Gamma', \Gamma''[x := D] \vdash B[x := D] : C[x := D]$

Theorem 7.1 (Church-Rosser Theorem; CR; Confluence)**.** *The Church-Rosser property holds for λC.*

Theorem 7.2 (Strong Normalization Theorem or Termination Theorem)**.** *Every legal M is strongly normalizing.*

Theorem 7.3 (Decidability of Well-typedness and Type Checking)**.** *In λC and its subsystems, the questions of Well-typedness and Type Checking are decidable.*

The question of *Term finding* it is decidable for $\lambda \rightarrow$ and $\lambda\underline{\omega}$, but undecidable for the rest. All the proofs can be found on [Bar93].

7.4 Exercises

7.4.1 6.6

Given $M \equiv \lambda S : *.\lambda P : S \to *.\lambda x : S.(Px \to \bot)$.

((a)). *Which is the smallest system in the λ-cube in which M may occur?*

The smallest is λC. Taking the eight possibilities

- $\bot$ needs $(\Box, *)$
- $\lambda x : S.(Px \to \bot)$ needs $(*, \Box)$
- $\lambda P : S \to *.\lambda x : S.(Px \to \bot)$ needs $(\Box, \Box)$

((b)). *Proof M is legal*

(1) $\boxed{S : *}$

(2) $\quad \boxed{P : S \to *}$

(3) $\quad\quad \boxed{x : S}$

(4) $\quad\quad\quad Px : S$

(5) $\quad\quad\quad \bot : *$

(6) $\quad\quad\quad Px \to \bot : *$

(7) $\quad\quad \lambda x : S.(Px \to \bot) : S \to *$

(8) $\quad \lambda P : S \to *.\lambda x : S.(Px \to \bot) : (S \to *) \to S \to *$

(9) $\lambda S : *.\lambda P : S \to *.\lambda x : S.(Px \to \bot) : S \to (S \to *) \to S \to *$

((c)). *How could you interpret M if $A \to \bot$ encodes $\neg A$?*

It is a function that maps S and P to $\overline{P}$, something that is true if P does not hold, therefore it is impossible.

8 The encoding of logical notions in λC

Until now in λP system we have been seen in PAT-interpretation how encode $\Longrightarrow$ and $\forall$ through *introduction* and *elimination* rules.

This chapter presents the formalization for encoding $\neg, \wedge, \vee$ in Type Theory.

8.1 Absurdity and negation in type theory

Negation '$\neg$', can be seen as an implication $A \implies \bot$, where $\bot$ is *absurdity* or *contradiction*. In that sense $\neg A$ means 'A implies absurdity'

Absurdity

If $\bot$ is true, then every proposition is true. In Type Theory words *If $\bot$ is inhabited, then all propositions A are inhabited.*

If M is an inhabitant of $\bot$, there exist a function $f : \Pi\alpha : *.\alpha$. By *(appl)* rule $fA : \alpha[\alpha := A]$, this holds for general A, so fB inhabits B, etc.

If such an f exists, then we can make all propositions $(A, B, \dots)$ true; which is **absurd**.

$$\bot \text{ is inhabited} \iff \Pi\alpha : *.\alpha \text{ is inhabited.}$$

Negation

$\neg A \equiv A \to \bot$, where $A \to \bot$ is an abbreviation of $\Pi x : A.\ \bot$

$$(\bot\text{-intro}) \quad \frac{A \qquad A \implies \bot}{\bot}$$

Introduction and elimination rule compared with Propositional logic

$$(\neg\text{-intro}) \quad \frac{\begin{array}{c} A \\ \vdots \\ \bot \end{array}}{\neg A} \qquad (\implies\text{-intro}) \quad \frac{\begin{array}{c} A \\ \vdots \\ B \end{array}}{A \implies B}$$

$$(\neg\text{-elim}) \quad \frac{\neg A \qquad A}{\bot} \qquad (\implies\text{-elim}) \quad \frac{A \implies B \qquad A}{B}$$

8.2 Conjunction and disjunction in type theory

Conjunction

$A \wedge B \equiv \Pi C : *.(A \to B \to C) \to C$, which is:

- $\Pi C : *.(A \to B \to C) \to C$: For all C, (A implies (B implies C)) implies C
- Or *if A and B together imply C, C holds*

Introduction and elimination rule for *second order encodings* compared with Propositional logic:

$$(\wedge\text{-intro}) \quad \frac{A \qquad B}{A \wedge B} \qquad (\wedge\text{-intro-sec}) \quad \frac{A \qquad B}{\Pi C : *(A \to B \to C) \to C}$$

$$(\wedge\text{-elim-left}) \quad \frac{A \wedge B}{A} \qquad (\wedge\text{-elim-left-sec}) \quad \frac{\Pi C : *(A \to B \to C) \to C}{A}$$

$$(\wedge\text{-elim-right}) \quad \frac{A \wedge B}{B} \qquad (\wedge\text{-elim-right-sec}) \quad \frac{\Pi C : *(A \to B \to C) \to C}{B}$$

Derivable rules for this under PAT-interpretation, assuming that $\Gamma \equiv A : *, B : *$

$$(\wedge\text{-intro-sec-tt}) \quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash ?_1 : \Pi C : *.(A \to B \to C) \to C}$$

$$(\wedge\text{-elim-left-sec-tt}) \quad \frac{\Gamma \vdash c : \Pi C : *.(A \to B \to C) \to C}{\Gamma \vdash ?_2 : A}$$

$$(\wedge\text{-elim-right-sec-tt}) \quad \frac{\Gamma \vdash c : \Pi C : *.(A \to B \to C) \to C}{\Gamma \vdash ?_3 : B}$$

In flag derivation *(∧-intro-sec)* format to find $?_1, ?_2, ?_3$ we have:

(1)	$\boxed{A : *}$	
(2)	$\quad\boxed{B : *}$	
(3)	$\quad\quad\boxed{x : A}$	
(4)	$\quad\quad\quad\boxed{y : B}$	
(5)	$\quad\quad\quad\quad\boxed{C : *}$	
(6)	$\quad\quad\quad\quad\quad\boxed{z : A \to B \to C}$	
(7)	$\quad\quad\quad\quad\quad\quad zx : B \to C$	(appl) on (6) and (3)
(8)	$\quad\quad\quad\quad\quad\quad zxy : C$	(appl) on (7) and (4)
(9)	$\quad\quad\quad\quad\quad \lambda z : A \to B \to C.zxy : (A \to B \to C) \to C$	(abst) on (8)
(10)	$\quad\quad\quad\quad \lambda C : *.\lambda z : A \to B \to C.zxy :$ $\Pi C : *.(A \to B \to C) \to C$	(abst) on (9)

Disjunction

$A \vee B \equiv \Pi C : *.(A \to B) \to (B \to C) \to C$, which is:

- $\Pi C : *.(A \to B) \to (B \to C) \to C$: For all C, ($A \to C$ implies that ($B \to C$ implies C)
- Or *if A implies C and also B implies C, C holds on its own*

In natural deduction rules:

($\vee$-intro-left) $$\frac{A}{A \vee B}$$

($\vee$-intro-right) $$\frac{B}{A \vee B}$$

($\forall$-elim) $$\frac{A \vee B \qquad A \implies C \qquad B \implies C}{C}$$

In Type Theory second order version:

($\vee$-intro-left-sec) $$\frac{A}{\Pi C : *.(A \to B) \to (B \to C) \to C}$$

($\vee$-intro-right-sec) $$\frac{B}{\Pi C : *.(A \to B) \to (B \to C) \to C}$$

($\forall$-elim-sec) $$\frac{\Pi D : *.(A \to D) \to (B \to D) \to D \qquad A \to C \qquad B \to C}{C}$$

In flag derivation *($\vee$-intro-sec)*

(1)	$A : *$	
(2)	$B : *$	
(3)	$C : *$	
(4)	$x : (\Pi D : *.(A \to D) \to (B \to D) \to D)$	
(5)	$y : A \to C)$	
(6)	$z : B \to C)$	
(7)	$xC : (A \to C) \to (B \to C) \to C$	(appl) on (4) and (3)
(8)	$xCy : (B \to C) \to C$	(appl) on (7) and (5)
(9)	$xCyz : C$	(appl) on (8) and (6)

The Three encodings $\neg, \wedge, \vee$ are not formally complete because A, B are free variables and we need something to define formally that $A, B : *$.

- $\neg \equiv \lambda\alpha : *.(\alpha \to \bot)$
- $\wedge \equiv \lambda\alpha : *.\lambda\beta : *.\Pi\gamma : *.(\alpha \to \beta \to \gamma) \to \gamma$
- $\vee \equiv \lambda\alpha : *.\lambda\beta : *.\Pi\gamma : *.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$

With this encoding we have everything in place to derive the expressions, for example:

$\neg A \equiv (\lambda\alpha : *.(\alpha \to \bot))A \to \bot \quad \to_\beta \quad A \to \bot$

8.3 Example of propositional logic in λC

Lets start with the tautology $(A \vee B) \implies (\neg A \implies B)$:

We need a type theoretical proof that the previous encoding works:

$$\underbrace{(\Pi C : *.(A \to C) \to (B \to C) \to C)}_{A \vee B} \to \underbrace{(A \to \bot)}_{\neg A} \to B$$

(1)	$A : *$	
(2)	$B : *$	
(3)	$x : (\Pi C : *.(A \to C) \to (B \to C) \to C)$	
(4)	$y : A \to \bot$	
(5)	$xB : (A \to B) \to (B \to B) \to B$	(appl) on (2) and (3)
(6)	$u : A$	
(7)	$yu : \bot$	(appl) on (6) and (4)
(8)	$yuB : B$	(appl) on (7) and (2)
(9)	$\lambda u : A.yuB : A \to B$	(abst) on (8)
(10)	$xB(\lambda u : A.yuB) : (B \to B) \to B$	(appl) on (5) and (9)
(11)	$v : B$	
(12)	$v : B$	(var) on (11)
(13)	$\lambda v : B.v : B \to B$	(abst) on (12)

(14)	$xB(\lambda u : A.yuB)(\lambda v : B.v) : B$	(appl) on (10) and (13)
(15)	$\lambda y : A \to \bot\ .xB(\lambda u : A.yuB)(\lambda v : B.v) :$ $(A \to perp) \to B$	(abst) on (14)
(16)	$\lambda x : (\Pi C : *.(A \to C) \to (B \to C) \to C).$ $\lambda y : A \to \bot\ .xB(\lambda u : A.yuB)(\lambda v : B.v) :$ $(\Pi C : *.((A \to C) \to (B \to C) \to C)) \to (A \to \bot) \to B$	(abst) on (15)

Derivation 24: A derivation of logical tautology $(A \vee B) \implies (\neg A \implies B)$

8.4 Classical logic in λC

Until now the correspondence between type theory and Logic was focused on *Constructive* logic, which is less powerful than *Classical*, for example the law of *exclude the third* (ET) where $A \vee \neg A$ holds for any A, and *double negation* law (ND) where $\neg\neg A \implies A$ also holds for any A.

In order to encode the extension of *constructive* logic to *classical* we only need to add either ET or ND to *constructive*, because the one we dont add can be derived from the other. This addition on λC it is just having something that always holds. This can be done adding an assumption at the top of the context.

For example, adding ET:

$\boxed{i_{ET} : \Pi\alpha : *.\alpha \vee \neg\alpha}$

We call this the addition of an *axiom*.

Now we can derive from our *constructive* logic plus ET, ND:

(1)	$\boxed{i_{ET} : \Pi\alpha : *.\alpha \vee \neg\alpha}$
(2)	$\boxed{\beta : *}$
(3)	$\boxed{x : \neg\neg\beta}$
(4)	$i_{ET}\beta : \beta \vee \neg\beta$
(5)	$i_{ET}\beta : \Pi\gamma : *.(\beta \to \gamma) \to (\neg\beta \to \gamma) \to \gamma$
(6)	$i_{ET}\beta\beta : (\beta \to \beta) \to (\neg\beta \to \beta) \to \beta$
(7)	$\lambda y : \beta.y : \beta \to \beta$

(8) $i_{ET}\beta\beta(\lambda y : \beta.y) : (\neg\beta \to \beta) \to \beta$

(9) $\boxed{z : \neg\beta}$

(10) $\quad xz :\bot$

(11) $\quad xz\beta : \beta$

(12) $\lambda z : \neg\beta.xz\beta : \neg\beta \to \beta$

(13) $i_{ET}\beta\beta(\lambda y : \beta.y)(\lambda z : \neg\beta.xz\beta) : \beta$

Derivation 25: A derivation of ND from *constructive* logic plus ET

8.5 Predicate logic λC

Now we are going top encode *Predicate logic*. For that we need to encode quantifiers $\forall, \exists$.

In the case of $\forall$ we already provided the encoding for $\forall_{x\in S}(P(x))$ as $\Pi x : S.Px$.

In the case of $\exists$ we need to provide something for $\exists_{x\in S}(P(x)) \equiv \neg\forall_{x\in S}(\neg P(x))$ having:

$\exists_{x\in S}(P(x))$ as $\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha)$

Which means *For all α, if we know that for all $x \in S$ it holds that Px implies α, then α holds.*

We got there with *($\exists$-elim)* and *($\exists$-intro)* rules:

Elimination rule

($\exists$-elim) $$\frac{\exists_{x\in S}P(x) \qquad \forall_{x\in S}(P(x) \implies A)}{A}$$

In type theory:

($\exists$-elim-sec) $$\frac{\Pi\alpha : *.((\Pi x : S.(Px \to \alpha) \to \alpha) \qquad \Pi x : S.(Px \to A)}{A}$$

(1) $\boxed{S : *}$

(2) $\quad \boxed{P : S \to *}$

(3) $\quad\quad \boxed{A : *}$

(4) $\boxed{y : \Pi\alpha : *.((\Pi x : S.(Px \to \alpha) \to \alpha)}$

(5) $\quad\boxed{z : \Pi x : S.(Px \to A)}$

(6) $\quad\quad yA : (\Pi x : S.(Px \to A)) \to A$

(7) $\quad\quad yAz : A$

Derivation 26: A derivation of *(∃-elim-sec)*

Introduction rule

(∃-intro) $\dfrac{a \in S \qquad P(a)}{\exists_{x \in S}(P(x))}$

In type theory:

(∃-intro-sec) $\dfrac{a : S \qquad Pa}{\Pi\alpha : *.((\Pi x : S.(Px \to \alpha) \to \alpha)}$

(1) $\boxed{S : *}$

(2) $\quad\boxed{P : S \to *}$

(3) $\quad\quad\boxed{a : S}$

(4) $\quad\quad\quad\boxed{u : Pa}$

(5) $\quad\quad\quad\quad\vdots$

(6) $\quad\quad\quad\quad ? : \Pi\alpha : *.((\Pi x : S.(Px \to \alpha) \to \alpha)$

Derivation 27: A derivation of *(∃-intro-sec)*

8.6 Example of predicate logic in λC

Given the following proposition:

$\neg\exists_{x \in S}(P(x)) \implies \forall_{y \in S}(\neg P(y))$

In second order λC-encoding this is:

$$\underbrace{\underbrace{(\underbrace{(\Pi\alpha : *.(\Pi x : S.(Px \to \alpha)) \to \alpha)}_{\exists_{x \in S}(P(x))} \to \bot) \to \underbrace{\Pi y : S.(Py \to \bot)}_{\forall_{y \in S}(P(y))}}_{\neg\exists_{x \in S}(P(x))}}_{\implies}$$

(1)	$\boxed{S : *}$	
(2)	$\boxed{P : S \to *}$	
(3)	$\boxed{u : \neg(\exists x : S.Px)}$	
(4)	$\boxed{y : S}$	
(5)	$\boxed{v : Py}$	
(6)	$\lambda\alpha : *.\lambda w : (\Pi x : S.(Px \to \alpha)).wyv : \exists x : S.Px$	
(7)	$u(\lambda\alpha : *.\lambda w : (\Pi x : S.(Px \to \alpha)).wyv) : \bot$	(appl) on (3) and (6)
(8)	$\cdots : \neg(Py)$	(abst) on (7)
(9)	$\cdots : \forall y : S.\neg(Py)$	(abst) on (8)
(10)	$\cdots : \neg(\exists x : S.Px) \implies \forall y : S.\neg P(y)$	(abst) on (9)

Derivation 28: A derivation of $\neg\exists_{x\in S}(P(x)) \implies \forall_{y\in S}(\neg P(y))$

8.7 Exercises

8.7.1 7.2

(a). *Formulate a double negation law (DN) as an axiom in λC*

Formulation. $\boxed{L_{dn} : \Pi\gamma : *.\neg\neg\gamma \to \gamma}$ □

(b). *Verify the following expression is a tautology in classical logic by givin a corresponding flag-style derivation in λC). Use (DN).* $(\neg A \implies A) \implies A$

Derivation. $\neg A \implies A \equiv \neg\neg A \vee A$
So, we have the following expression: $(\neg\neg A \vee A) \implies A$

(1)	$\boxed{L_{dn} : \Pi\gamma : *.\neg\neg\gamma \to \gamma}$
(2)	$\boxed{A : *}$
(3)	$\boxed{x : \Pi A : *.((\neg\neg A \to A) \to (A \to A) \to A)}$
(4)	$\boxed{y : A}$
(5)	$L_{dn}A : \neg\neg A \to A$

(6) $xL_{dn}A : (A \to A) \to A$

(7) $\lambda y : A.y : A \to A$

(8) $xL_{dn}Ay : A$

□

9 Conclusion

I have been working professionally in Functional Programming (FP) for the last 5 years, and in particular in ***Haskell*** for the last 2. Reading this book allowed me to understand the strong relationship between ***Types*** and ***Programs***, in a more formal and mathematical way. This concept in fact was first introduced in [How80] with the name of *Proposition as types*, but here the full formalization of the concept in the framework of *Type Theory* and in particular in $\lambda \to$ system, has been done in a deep and contextual manner. Learning the formalization on how a correspondence between *Logic* and *Type Theory* can be established, it is a great tool for understanding the need of ***Strong Typed Checked Languages***, which give us a powerful tool where we can write programs as a mathematical proof of some propositions which are the types.

Another important and fundamental construct of the book I have learned is the different Lambda Calculus Systems describe from Chapter 1 up to 6. Before this book, my knowledge was limited to Untyped Lambda Calculus in general and some construct what is formalized here as $\lambda \to$ system. This is because in most of the Functional Programming Books or Online resources, only Untyped Lambda Calculus is introduced as a way to understand the formalization roots of Paradigm, but a lot of details are left for the reader. Reading and going deeper on the exercises allowed me to understand that there are more systems involved like $\lambda 2$, $\lambda \underline{\omega}$, λP and λC and how they are related. Basically each system contains the previous with more formalization to support more encodings. This have been summarized clearly in what it is called λ-cube 7.2, until we reached to a formalization which allows us to encode ***Propositional logic and Classical***.

Having discovering this it was a novelty for me because i understood finally the roots of the Automated Theorem Proofers like ***Agda and Coq*** which now i am eager to explore deeper.

Regarding the exercises I would say that are doable and manageable because

in each chapter the author presents from the easiest ones to the complex. I have liked and enjoyed very much doing the *flag derivation* exercises which i consider pretty challenging and close to natural deduction as well.

In regards the practical exercises and description of *flag derivation* or deductions, I think there is a lack of explanations from the authors of the book on some specific steps and how they are applied. For example, there is a part that is explained here 5.1, where there is a *(var)* rule applied twice. This is something that is also applied in *flag derivation* and it is not clearly explain from the authors and I had to research on my own. Although the rest of the explanations are quite formal and deep and they claimed in the Preface that the book is for *advanced readers*, I think they should have done a better explanation of the derivations and how to do this step by step, moreover in advanced system like λP or $\lambda\underline{\omega}$ which are not so cover.

In General the book is a great and complete guideline to *Type Theory*, its formalization and how this is a applied in *Formal Proof* in Mathematics. As I've claimed before, even for my field which is *Computer Science*, this is a great material for understanding in deep the roots of the *Type Systems* behind *Compilers and Strong typed languages*.

References

[Bar93] H. P. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993.

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

[How80] W. Howard. The formulas-as-types notion of construction. *Seldin & Hindley*, pages 479–490, 1980.

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