

A Report of *Type Theory and Formal Proof*

Juan Pablo Royo Sales

Universitat Politècnica de Catalunya

February 19, 2021

Contents

1	Introduction	3
2	Untyped lambda calculus	3
2.1	Definition	3
2.1.1	Lambda-terms	4
2.2	Free and bound variables	4
2.2.1	Alpha conversion	5
2.3	Substitution	5
2.4	Beta reduction	5
2.5	Fixed Point Theorem	6
2.6	Exercises	6
2.6.1	1.10 Church numerals	6
2.6.2	1.11 - Successor	7
2.6.3	1.12 - If then else	8
3	Simply typed lambda calculus	9
3.1	Simple types	9
3.1.1	Remarks	9
3.2	Church-typing and Curry-typing	10
3.2.1	Typing à la Church	10
3.2.2	Typing à la Curry	10
3.3	Derivation rules for Church's $\lambda \rightarrow$	10
3.3.1	Example	11
3.4	Derivation formats	11
3.4.1	Linear format	11
3.4.2	Flag notation	11

3.5	Problems solved with judgement in Type Theory	12
3.5.1	Well-typedness in $\lambda \rightarrow$	12
3.5.2	Type Checking in $\lambda \rightarrow$	13
3.5.3	Term finding in $\lambda \rightarrow$	14
3.6	General properties of $\lambda \rightarrow$	15
3.7	Reductions and $\lambda \rightarrow$	15
3.8	Exercises	16
3.8.1	2.5 Find pre-typed terms	16
3.8.2	2.9 Type checking	16
4	Second order typed lambda calculus	17
4.1	Π -types	18
4.2	Second Order abstraction and application rules	18
4.3	The system $\lambda 2$	19
4.4	Properties of $\lambda 2$	19
4.5	Exercises	20
4.5.1	3.13 Addition and Multiplication	20
5	Types dependent on types	20
5.1	The weakening rule in $\lambda\omega$	22
5.2	Formation rule in $\lambda\omega$	23
5.3	Application and Abstraction rule in $\lambda\omega$	23
5.4	The conversion rule	24
5.5	Properties in $\lambda\omega$	24
5.6	Exercises	24
5.6.1	4.4. Flag derivations	24
5.6.2	4.5	25
6	Types dependent on terms	26
6.1	Derivation rules of λP	27
6.2	Example derivation of λP	27
6.3	Minimal predicate logic in λP	28
6.3.1	Sets	28
6.3.2	Propositions	28
6.3.3	Predicates	28
6.3.4	Implication	28
6.3.5	Universal quantification	29
6.4	Examples of derivations in λP	29
6.5	Exercises	30
6.5.1	5.4	30
6.5.2	5.9(a)	31

References

31

1 Introduction

This report is going to provide a summary over the first 6 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 $L \neq M$

- Parenthesis can be omitted
- Application is left-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; $M^{x \rightarrow y}; =_\alpha$). *Let $M^{x \rightarrow y}$ denote the result of replacing every free occurrence of x in M by y . Renaming, expressed by $=_\alpha$ is defined as: $\lambda x.M =_\alpha \lambda y.M^{x \rightarrow y}$, provided that $y \notin FV(M)$ and y is not binding in M*

Definition 2.7 (α -conversion 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 $x \neq y$
3. $(PQ)[x := N] \equiv (P[x := N])(Q[x := N])$
4. $(\lambda y.P)[x := N] \equiv \lambda z.(P^{y \rightarrow z}[x := N])$, if $\lambda z.P^{y \rightarrow z}$ is α -variant of $\lambda y.P$ such that $z \notin FV(N)$

2.4 Beta reduction

Definition 2.9 (One-step β -reduction, \rightarrow_β).

1. (Basis) $(\lambda x.M)N \rightarrow_\beta M[x := N]$,

2. (Compatibility) If $M \rightarrow_\beta N$, then $ML \rightarrow_\beta NL$, $LM \rightarrow_\beta LN$ and $\lambda x.M \rightarrow_\beta \lambda x.N$

In 1 the left part of \rightarrow_β 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_β). $M \twoheadrightarrow_\beta N$ 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 :

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

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 $i, 0 \leq i < n$:

$$\text{either } M_i \rightarrow_\beta M_{i+1} \text{ or } M_{i+1} \rightarrow_\beta M_i$$

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:

$$M \equiv (\lambda x.L(xx))(\lambda x.L(xx)) \tag{1a}$$

$$\rightarrow_\beta L((\lambda x.L(xx))(\lambda x.L(xx))) \tag{1b}$$

$$\equiv LM \tag{1c}$$

Therefore, $LM =_\beta M$ □

2.6 Exercises

2.6.1 1.10 Church numerals

Having that:

- $zero := \lambda f x.x$
- $one := \lambda f x.f x$
- $two := \lambda f x.f(f x)$

- $add := \lambda mnfx.mf(nfx)$
- $mult := \lambda mnfx.m(nf)x$

(a). Show that: $(add\ one\ one \rightarrow_{\beta}\ two)$

Proof. Replacing by lambda expressions

$$add\ one\ one := (\lambda mnfx.mf(nfx))(\lambda fx.fx)(\lambda fx.fx) \quad (2a)$$

$$\rightarrow_{\beta} (\lambda nfx.(\lambda fx.fx)f(nfx))(\lambda fx.fx) \quad (2b)$$

$$\rightarrow_{\beta} (\lambda fx.(\lambda fx.fx)f((\lambda fx.fx)fx)) \quad (2c)$$

$$\rightarrow_{\beta} (\lambda fx.(\lambda fx.fx)f(fx)) \quad (2d)$$

$$\rightarrow_{\beta} (\lambda fx.f(fx)) \quad (2e)$$

$$:= two \quad (2f)$$

□

(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

$$mult\ one\ zero := (\lambda mnfx.m(nf)x)(\lambda fx.fx)(\lambda fx.x) \quad (3a)$$

$$\rightarrow_{\beta} (\lambda nfx.(\lambda fx.fx)(nf)x)(\lambda fx.x) \quad (3b)$$

$$\rightarrow_{\beta} (\lambda fx.(\lambda fx.fx)((\lambda fx.x)f)x) \quad (3c)$$

$$\rightarrow_{\beta} (\lambda fx.(\lambda x.((\lambda fx.x)f)x)x) \quad (3d)$$

$$\rightarrow_{\beta} (\lambda fx.(\lambda x.(\lambda x.x)x)x) \quad (3e)$$

$$\rightarrow_{\beta} (\lambda fx.(\lambda x.x)x) \quad (3f)$$

$$\rightarrow_{\beta} (\lambda fx.x) \quad (3g)$$

$$:= zero \quad (3h)$$

□

2.6.2 1.11 - Successor

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

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

Proof.

$$suc\ zero =_{\beta} (\lambda mfx.f(mfx))(\lambda fx.x) \quad (4a)$$

$$\rightarrow_{\beta} (\lambda fx.f((\lambda fx.x)fx)) \quad (4b)$$

$$\rightarrow_{\beta} (\lambda fx.f((\lambda x.x)x)) \quad (4c)$$

$$\rightarrow_{\beta} (\lambda fx.fx) \quad (4d)$$

$$:= one \quad (4e)$$

□

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

Proof.

$$suc\ one =_{\beta} (\lambda mfx.f(mfx))(\lambda fx.fx) \quad (5a)$$

$$\rightarrow_{\beta} (\lambda fx.f((\lambda fx.fx)fx)) \quad (5b)$$

$$\rightarrow_{\beta} (\lambda fx.f((\lambda x.fx)x)) \quad (5c)$$

$$\rightarrow_{\beta} (\lambda fx.f(fx)) \quad (5d)$$

$$:= two \quad (5e)$$

□

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:

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

$(\lambda x.xuv)true$.

$$:= (\lambda x.xuv)(\lambda xy.x) \quad (6a)$$

$$\rightarrow_{\beta} (\lambda xy.x)uv \quad (6b)$$

$$\rightarrow_{\beta} (\lambda y.u)v \quad (6c)$$

$$\rightarrow_{\beta} u \quad (6d)$$

$$(6e)$$

□

$(\lambda x.xuv)false.$

$$:= (\lambda x.xuv)(\lambda xy.y) \quad (7a)$$

$$\rightarrow_{\beta} (\lambda xy.y)uv \quad (7b)$$

$$\rightarrow_{\beta} (\lambda y.y)v \quad (7c)$$

$$\rightarrow_{\beta} v \quad (7d)$$

$$(7e)$$

□

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 $(\sigma \rightarrow \tau) \in \mathbb{T}$

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

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* \rightarrow *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 $M : \sigma \rightarrow \tau$ and $N : \sigma$, then $MN : \tau$

- *Abstraction*: If $x : \sigma$ and $M : \tau$, then $\lambda x.M : \sigma \rightarrow \tau$

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 $\alpha \rightarrow \alpha$ and y has type $(\alpha \rightarrow \alpha) \rightarrow \beta$, then yx has type β .

If z has type β and u has type γ , then $\lambda zu.z$ has type $\beta \rightarrow \gamma \rightarrow \beta$. 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 $A \rightarrow B$, 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, $\Lambda_{\mathbb{T}}$).

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

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

$$\begin{aligned}
 (\text{var}) \quad & \Gamma \vdash x : \sigma \text{ if } x : \sigma \in \Gamma \\
 (\text{appl}) \quad & \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \\
 (\text{abst}) \quad & \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau}
 \end{aligned}$$

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

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

3.4 Derivation formats

3.4.1 Linear format

1. $y : \alpha \rightarrow \beta, z : \alpha \vdash y : \alpha \rightarrow \beta$ (*var*)
2. $y : \alpha \rightarrow \beta, z : \alpha \vdash z : \alpha$ (*var*)
3. $y : \alpha \rightarrow \beta, z : \alpha \vdash yz : \beta$ (*appl*) on 1 and 2
4. $y : \alpha \rightarrow \beta \vdash \lambda z : \alpha. yz : \alpha \rightarrow \beta$ (*abst*) on 3
5. $\emptyset \vdash \lambda y : \alpha \rightarrow \beta. \lambda z : \alpha. yz : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ (*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 below 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)	$y : \alpha \rightarrow \beta$	
(2)	$z : \alpha$	
(3)	$y : \alpha \rightarrow \beta$	(var) on (1)
(4)	$z : \alpha$	(var) on (2)
(5)	$yz : \beta$	(appl) on (3) and (4)
(6)	$\lambda z : \alpha. yz : \alpha \rightarrow \beta$	(abst) on (5)
(7)	$\lambda y : \alpha \rightarrow \beta. \lambda z : \alpha. yz : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$	(abst) on (6)

Even more succinct without *var* rule:

(1)	$y : \alpha \rightarrow \beta$	
(2)	$z : \alpha$	
(3)	$yz : \beta$	(appl) on (1) and (2)
(4)	$\lambda z : \alpha. yz : \alpha \rightarrow \beta$	(abst) on (3)
(5)	$\lambda y : \alpha \rightarrow \beta. \lambda z : \alpha. yz : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$	(abst) on (4)

3.5 Problems solved with judgement in Type Theory

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

3.5.1 Well-typedness in $\lambda \rightarrow$

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 and a type ρ such that $\Gamma \vdash M : \rho$

In our previous example of derivation if we start checking that the term $\lambda y : \alpha \rightarrow \beta. \lambda z : \alpha. yz : ?$ 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 $\lambda y : \alpha \rightarrow \beta. \lambda z : \beta. yz : ?$, we have not because $z : \beta$ cannot be applied to y .

3.5.2 Type Checking in $\lambda \rightarrow$

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

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

$$\begin{array}{l|l} (1) & x : \alpha \rightarrow \alpha \\ (2) & \boxed{y : (\alpha \rightarrow \alpha) \rightarrow \beta} \\ & \vdots \\ (3) & (\lambda z : \beta. \lambda u : \gamma. z)(yx) : \gamma \rightarrow \beta \end{array}$$

The idea is to fill the dots:

$$\begin{array}{l|l} (1) & x : \alpha \rightarrow \alpha \\ (2) & \boxed{y : (\alpha \rightarrow \alpha) \rightarrow \beta} \\ (3) & \lambda z : \beta. \lambda u : \gamma. z : ?_1 \\ & \vdots \\ (4) & yx : ?_2 \\ (5) & (\lambda z : \beta. \lambda u : \gamma. z)(yx) : \gamma \rightarrow \beta \quad (\text{appl on (3) and (4), (?)}) \end{array}$$

$$\begin{array}{l|l} (1) & x : \alpha \rightarrow \alpha \\ (2) & \boxed{y : (\alpha \rightarrow \alpha) \rightarrow \beta} \\ (3) & \boxed{z : \beta} \\ (4) & \boxed{u : \gamma} \\ (5) & z : \beta \quad (\text{var on (3)}) \\ (6) & \lambda u : \gamma. z : \gamma \rightarrow \beta \quad (\text{abst on (5)}) \end{array}$$

- | | | |
|-----|---|----------------------------|
| (7) | $\lambda z : \beta. \lambda u : \gamma. z : \beta \rightarrow \gamma \rightarrow \beta$ | (abst) on (6) |
| (8) | $yx : \beta$ | (appl) on (1) and (2) |
| (9) | $(\lambda z : \beta. \lambda u : \gamma. z)(yx) : \gamma \rightarrow \beta$ | (appl) on (7) and (8), (?) |

3.5.3 Term finding in $\lambda \rightarrow$

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

- | | | |
|-----|---|---------------|
| (1) | <div style="border: 1px solid black; display: inline-block; padding: 2px 5px;">$x : A$</div> | |
| | \vdots | |
| (2) | $? : B \rightarrow A$ | |
| | \vdots | |
| (3) | $\dots A \rightarrow B \rightarrow A$ | (abst) on (2) |

- | | | |
|-----|---|---------------|
| (1) | <div style="border: 1px solid black; display: inline-block; padding: 2px 5px;">$x : A$</div> | |
| (2) | <div style="border: 1px solid black; display: inline-block; padding: 2px 5px;">$y : B$</div> | |
| | \vdots | |
| (3) | $? : A \rightarrow A$ | |
| (4) | $\dots : B \rightarrow A$ | (abst) on (3) |
| (5) | $\dots A \rightarrow B \rightarrow A$ | (abst) on (4) |

- $$\begin{array}{ll}
(1) & \boxed{x : A} \\
(2) & \boxed{\boxed{y : B}} \\
(3) & \boxed{x : A} \quad (\text{var}) \text{ on } (1) \\
(4) & \boxed{\lambda y : B. x : B \rightarrow A} \quad (\text{abst}) \text{ on } (3) \\
(5) & \lambda x : A. \lambda y : B. x : A \rightarrow B \rightarrow A \quad (\text{abst}) \text{ on } (4)
\end{array}$$

3.6 General properties of $\lambda \rightarrow$

Definition 3.6 (Domain, dom , subcontext, \subseteq , permutation, projection, \upharpoonright).

1. If $\Gamma \equiv x_1 : \sigma_1, \dots, x_n : \sigma_n$, then the *domain* of Γ or $\text{dom}(\Gamma)$ is the list (x_1, \dots, 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 $\Gamma \upharpoonright \phi$, is the subcontext Γ' of Γ with $\text{dom}(\Gamma') = \text{dom}(\Gamma) \cap \phi$

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

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 $\Gamma \upharpoonright FV(M) \vdash M : \sigma$
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

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

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

Lemma 3.4 (Substitution 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_β , 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 $\lambda x : (\rho \rightarrow \beta) \rightarrow \beta \rightarrow \gamma.\lambda y : \beta.x(\lambda z : \rho.y)y$ with type $((\rho \rightarrow \beta) \rightarrow \beta \rightarrow \gamma) \rightarrow \beta \rightarrow \gamma$

□

(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 $\lambda z : \rho.x : \rho \rightarrow \beta \rightarrow \gamma \equiv \sigma$ which does not hold.

Therefore, term is not typeable.

□

3.8.2 2.9 Type checking

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

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

(3)			$z : \alpha \rightarrow \beta$	
(4)			$u : \delta$	
(5)			$v : \gamma$	
(6)			$yv : \alpha$	(appl) on (2) and (5)
(7)			$z(yv) : \beta$	(appl) on (3) and (6)
(8)			$\lambda v : \gamma. z(yv) : \gamma \rightarrow \beta$	(abst) on (7)
(9)			$\lambda u : \delta. \lambda v : \gamma. z(yv) : \delta \rightarrow \gamma \rightarrow \beta$	(abst) on (8)

Proof.

□

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

(1)			$x : \delta \rightarrow \delta \rightarrow \alpha$	
(2)			$y : \gamma \rightarrow \alpha$	
(3)			$z : \alpha \rightarrow \beta$	
(4)			$u : \delta$	
(5)			$v : \gamma$	
(6)			$xuu : \alpha$	(appl) on (1) and (4) twice
(7)			$z(xuu) : \beta$	(appl) on (3) and (6)
(8)			$\lambda v : \gamma. z(xuu) : \gamma \rightarrow \beta$	(abst) on (7)
(9)			$\lambda u : \delta. \lambda v : \gamma. z(xuu) : \delta \rightarrow \gamma \rightarrow \beta$	(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)
- $(\lambda\alpha : *. \lambda x : \alpha. x) \text{nat} \rightarrow_{\beta} \lambda x : \text{nat}. x$, which is the identity of nat
- $(\lambda\alpha : *. \lambda x : \alpha. x) \text{nat} \rightarrow \text{bool} \rightarrow_{\beta} \lambda x : (\text{nat} \rightarrow \text{bool}). x$, which is the identity of $\text{nat} \rightarrow \text{bool}$

Iteration

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

Function composition

$$\circ \equiv \lambda\alpha : *. \lambda\beta : *. \lambda\gamma : *. \lambda f : \alpha \rightarrow \beta. \lambda g : \beta \rightarrow \gamma. \lambda x : \alpha. g(f(x)) \quad (10)$$

4.1 Π -types

An expression like $\lambda\alpha : *. \lambda x : \alpha. x : * \rightarrow (\alpha \rightarrow \alpha)$, 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 $\Pi\alpha : *. \alpha \rightarrow \alpha$ means that any arbitrary type α can be use in a term $\alpha \rightarrow \alpha$.

So we have $\Pi\alpha : *. \alpha \rightarrow \alpha \equiv_{\alpha} \Pi\beta : *. \beta \rightarrow \beta$ 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).

$$(abt_2) \frac{\Gamma, \alpha : * \vdash M : A}{\Gamma \vdash \lambda \alpha : *. M : \Pi \alpha : *. A}$$

Definition 4.2 (Second order application rule).

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

4.3 The system $\lambda 2$

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

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; dom).

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

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

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

Definition 4.7 (Formation rule).

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

4.4 Properties of $\lambda 2$

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

1. (Renaming of term variable) $\lambda x : \sigma. M =_\alpha \lambda y : \sigma. M^{x \rightarrow y}$ 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* $\text{Add} \equiv \lambda m, n : \mathbf{nat}. \lambda \alpha : *. \lambda f : \alpha \rightarrow \alpha. \lambda x : \mathbf{nat}. m \alpha f (n \alpha f x)$
Show that simulates addition

Proof. We have:

$$\text{Add One One} \equiv (\lambda m, n : \mathbf{nat}. \lambda \alpha : *. \lambda f : \alpha \rightarrow \alpha. \lambda x : \mathbf{nat}. m \alpha f (n \alpha f x)) \dots \quad (11a)$$

$$\rightarrow_\beta \lambda \alpha : *. \lambda f : \alpha \rightarrow \alpha. \lambda x : \mathbf{nat}. f (f x) \quad (11b)$$

$$\equiv \text{Two} \quad (11c)$$

□

(b). *Find* $\lambda 2$ -term *Mult* that simulates multiplication on *Nat*.

Proof. We have:

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

□

5 Types dependent on types

Construct *generalized* types $\lambda \alpha : *. \alpha \rightarrow \alpha$. 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.

- $(\lambda \alpha : *. \alpha \rightarrow \alpha) \beta \rightarrow_\beta \beta \rightarrow \beta$

- $(\lambda\alpha : *. \alpha \rightarrow \alpha)\gamma \rightarrow_\beta \gamma \rightarrow \gamma$
- $(\lambda\alpha : *. \alpha \rightarrow \alpha)(\gamma \rightarrow \beta) \rightarrow_\beta (\gamma \rightarrow \beta) \rightarrow (\gamma \rightarrow \beta)$

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

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

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

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

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

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

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

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 $\mathcal{K} : \square$ and $M : \mathcal{K}$, then M is a constructor. If $\mathcal{K} \neq *$, then M is a proper constructor.
2. The set of **sort** is $\{*, \square\}$

Definition 5.2 (Levels).

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

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

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

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

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

	$s \equiv \square$		$s \equiv *$	
$A : s$	$* : \square$	$* \rightarrow * : \square$	$\alpha : *$	$\alpha \rightarrow \beta : *$
$x : A$	$\alpha : *$	$\beta : * \rightarrow *$	$x : \alpha$	$y : \alpha \rightarrow \beta$

$$\frac{\frac{(1) \emptyset \vdash * : \square}{(2) \alpha : * \vdash \alpha : *} (var)}{(3) \alpha : *, x : \alpha \vdash x : \alpha} (var)$$

5.1 The weakening rule in $\lambda\omega$

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

Definition 5.5 (Weakening rule).

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

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:

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

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

5.2 Formation rule in $\lambda\omega$

This rule enables to form types and kinds

Definition 5.6 (Formation rule).

$$(form) \quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash A \rightarrow B : s}$$

Examples:

$$\frac{\begin{array}{c} \dots \\ \alpha : *, \beta : * \vdash \alpha : * \end{array} \quad \begin{array}{c} \dots \\ \alpha : *, \beta : * \vdash \beta : * \end{array}}{\alpha : *, \beta : * \vdash \alpha \rightarrow \beta : *} (form)$$

$$\frac{\begin{array}{c} \dots \\ \alpha : * \vdash * : \square \end{array} \quad \begin{array}{c} \dots \\ \alpha : * \vdash * : \square \end{array}}{\alpha : * \vdash * \rightarrow * : \square} (form)$$

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

$$(appl) \quad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

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

Example:

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

$$(9) \quad \left| \quad (\lambda\alpha : *. \alpha \rightarrow \alpha)\beta : * \quad (\text{appl}) \text{ on (7) and (8)} \right.$$

5.4 The conversion rule

Recalling example:

$$(\lambda\alpha : *. \alpha \rightarrow \alpha)\beta \rightarrow_{\beta} \beta \rightarrow \beta$$

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 too weak to derive that and because of that we need to define the conversion rule.

Definition 5.7 (Conversion rule).

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

Example

$$\frac{\Gamma \vdash x : (\lambda\alpha : *. \alpha \rightarrow \alpha)\beta \quad \Gamma \vdash \beta \rightarrow \beta : *}{\Gamma \vdash x : \beta \rightarrow \beta} (conv)$$

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

$$\begin{array}{lcl} (1) & \boxed{\alpha : *} & \\ (2) & \boxed{\beta : * \rightarrow *} & \\ (3) & \beta\alpha : * & (\text{appl}) \\ (4) & \beta(\beta\alpha) : * & (\text{appl}) \end{array}$$

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

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

6 Types dependent on terms

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

- $\lambda \rightarrow$: *terms depending on terms*
- $\lambda 2$: *terms depending on terms + terms depending on types*
- $\lambda \omega$: 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:

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

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

<i>(sort)</i>	$\emptyset \vdash * : \square$
<i>(var)</i>	$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad \text{if } x \notin \Gamma$
<i>(weak)</i>	$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \quad \text{if } x \notin \Gamma$
<i>(form)</i>	$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A. B : s}$
<i>(appl)</i>	$\frac{\Gamma \vdash M : \Pi x : A. B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$
<i>(abst)</i>	$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x : A. B : s}{\Gamma \vdash \lambda x : A. M : \Pi x : A. B}$
<i>(conv)</i>	$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{if } B =_{\beta} B'$

The main difference with $\lambda\omega$ 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)	$* : \square$	(sort)
(2)	$A : *$	
(3)	$A : *$	(var) on (1)
(4)	$* : \square$	(weak) on (1) and (1)
(5)	$x : A$	
(6)	$* : \square$	(weak) on (3) and (4)

$$(7) \quad \left| \quad A \rightarrow * : \square \quad (\text{form}) \text{ on (3) and (6)} \right.$$

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 $a : S$.

$\text{nat} : *, \text{nat} \rightarrow \text{nat} : *, 3 : \text{nat}, \lambda n : \text{nat}. n : \text{nat} \rightarrow \text{nat}$

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. $P : S \rightarrow *$. Also for each $a : S$, then $Pa : *$

6.3.4 Implication

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

- $A \implies B$ 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 , $f : A \rightarrow B$
- $A \rightarrow B$ 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	$S : *$
A is a proposition	$A : *$
$a \in S$	$a : S$
p proves A	$p : A$
P is a predicate on S	$P : S \rightarrow *$
$A \implies B$ $\forall_{x \in S}(P(x))$	$A \rightarrow B (= \Pi x : A.B)$ $\Pi x : S.Px$
$(\implies -elim)$ $(\implies -intro)$	$(appl)$ $(abst)$
$(\forall -elim)$ $(\forall -intro)$	$(appl)$ $(abst)$

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) $Assume : \forall_{x \in S} \forall_{y \in S}(Q(x, y))$
- (2) $Let : u \in S$

(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 $? : \Pi x : S. \Pi y : S. Qxy \rightarrow \Pi u : S. Quu$

(1)	$S : *$	
(2)	$Q : S \rightarrow S \rightarrow *$	
(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)	$\lambda z : (\Pi x : S. \Pi y : S. Qxy). \lambda u : S. zuu :$	(abs) on (7)
	$\Pi x : S. \Pi y : S. Qxy \rightarrow \Pi u : S. Quu$	

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 $* \rightarrow *$

- 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 $* \rightarrow * : \Box$ 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) | $\forall_{x \in S}(Q(x))$ | |
| (2) | $y \in S$ | |
| (3) | $Q(y)$ | (\forall -elim) on (1) nad (2) |
| (4) | $P(y) \implies Q(y)$ | (\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) | $S : *$ |
| (2) | $P, Q : S \rightarrow *$ |
| (3) | $u : \Pi x : S. Qx$ |
| (4) | $v : \Pi z : S. (Pz \rightarrow Qz)$ |
| (5) | $uy : Qy$ |
| (6) | $\lambda y : S. vy : \Pi y : S. Py \rightarrow Qy$ |
| (7) | $\lambda u : (\Pi x : S. Qx). \lambda y : S. vy : \Pi y : S. Py \rightarrow Qy$ |

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.