University of Birmingham

School of Computer Science

Final Year Project

The Curry-Howard Isomorphism

Author:
Chuangjie Xu

Supervisor:
Prof. Achim Jung

March 1, 2011

Abstract

To be finished...

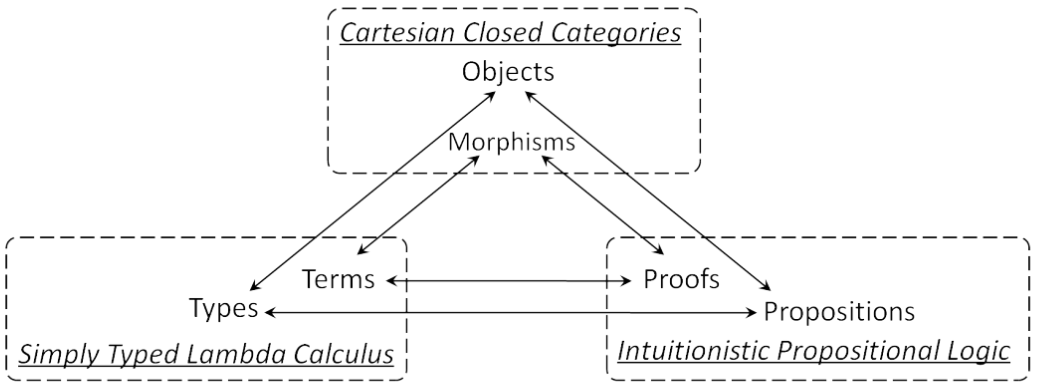
Contents

1 Introduction

Logicians must be familiar with modus ponens $P \to Q, P \vdash Q$, a very common rule of inference, saying that given implication $P \to Q$ and proposition P, we have Q. Programmers may frequently use function application: if f is a function of type $P \to Q$ and x is an argument of type P, then the application fx is of type Q. Interestingly, modus ponens behaves the same as function application. So, are proofs related to programs? Yes, there is an amazing precise correspondence between them which is described in the Curry-Howard Isomorphism.

In the 1930s, Haskell Curry observed a correspondence between types of combinators and propositions in intuitionist implicational logic. But, at that time, it was viewed as no more than a curiosity. About three decades later, William Howard extended this correspondence to first order logic by introducing dependent types. Therefore, this correspondence is called the Curry-Howard Isomorphism.

The Curry-Howard Isomorphism states a correspondence between systems of formal logic and computational calculi. For years, it has been extended to more expressive logics, e.g. higher order logic, and other mathematical systems, e.g. cartesian closed categories. In this project, I mainly probed into one of its extensions, the three-way-correspondence between intuitionistic propositional logic, simply-typed lambda calculus and cartesian closed categories, with propositions or types being interpreted as objects and proofs or terms as morphisms.


Cartesian Closed Categories
Objects
Morphisms
Terms
Types
Simply Typed Lambda Calculus
Proofs
Propositions
Intuitionistic Propositional Logic

Intuitionistic logic is a formalization of Brouwers intuitionism. As the founder of intuitionism, L. E. J. Brouwer avoided use of formal language or logic all his life. But his attitude did not stop others considering formalizations of parts of intuitionism. In the 1930s, Arend Heyting, a former student of Brouwer, produced the first complete axiomatizations for intuitionistic propositional and predicate logic. In intuitionistic logic, the law of excluded middle and double negation elimination are no longer axioms.

The lambda calculus was introduced by Alonzo Church in the early 1930s as a formal system to provide a functional foundation for mathematics. Since Churchs original system was shown to be logically inconsistent, he gave just a consistent subtheory of his original system dealing only with the functional part. Then, in 1940, Church also introduced a typed interpretation of the lambda calculus by giving each term a unique type. Today, the typed lambda calculus serves as the foundation of the modern type systems in computer science.

Category first appeared in Samuel Eilenberg and Saunders Mac Lanes paper written in 1945. It was originally introduced to describe the passage from one type of mathematical structure to another. In recent decades, category theory has found use for computer science. For instant, it has a profound influence on the design of functional and imperative programming languages, e.g. Haskell and Agda.

Looking from the historical perspective, these three different systems seem to have different origins, not related to each other. However, Joachim Lambek showed in the early 1970s that cartesian closed categories provided a formal analogy between proofs in intuitionistic propositional logic and types in combinatory logic. As a result, some people may use Curry-Howard-Lambek Isomorphism to refer to this three-way-correspondence.

2 Background

2.1 Intuitionistic Logic

Intuitionistic logic is also called constructive logic. As a formalization of intuitionism, it differs from classical logic not only in that some laws in classical logic are not axioms of the system but also in the meaning for statements to be true. The judgments about statements are based on the existence of a proof or a "construction" of that statement. This existence property makes it practically useful, e.g. provided that a constructive proof that an object exists, one can turn it into an algorithm for generating an example of the object.

One vertex in the correspondence-triangle is intuitionistic propositional logic. So the introduction to intuitionistic logic in this dissertation is up to the propositional one.

2.1.1 Syntax

The language of intuitionistic propositional logic is similar to the one of classical propositional logic. Customarily, people use $\bot, \rightarrow, \wedge, \vee$ as basic connectives and treat $\neg\varphi$ as an abbreviation for $\varphi \rightarrow \bot$.

Definition 2.1.1 (Formulas). Given an infinite set of propositional variables, the *set* Φ *of formulas* in intuitionistic propositional logic is defined by induction, represented in the following grammar:

$$\Phi \ ::= \ P \mid \bot \mid \neg\Phi \mid (\Phi \rightarrow \Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi)$$

where P is a *propositional variable*, $\bot$ is *contradiction*, $\neg$ is *negation*, $\rightarrow$ is *implication*, $\wedge$ is *conjunction*, and $\vee$ is *disjunction*.

Given a set Γ of propositions and a proposition φ, the relation $\Gamma \vdash \varphi$ says that there is a derivation with conclusion φ from hypotheses in Γ. Here, Γ is also called *context*. If Γ is empty, we write $\vdash \varphi$ and say that φ is a theorem.

For notational convenience, we use the following abbreviations:

- $\varphi_1, \varphi_2, \cdots, \varphi_n$ for $\{\varphi_1, \varphi_2, \cdots, \varphi_n\}$;
- Γ, φ for $\Gamma \cup \{\varphi\}$.

The *natural deduction system*, one kind of proof calculi, allows one to derive conclusions from premises. The logical reasoning in this system is expressed by inference rules which are closely related to the "natural" way of reasoning. The general form of an inference rule is

$$\frac{P_1, \cdots, P_n}{Q} \text{ name of the inference rule}$$

where $P_1, \cdots, P_n$ are premises and Q is the conclusion. A rule without premises is an axiom. Each inference rule is an atomic step in a derivation which shows how the relation $\vdash$ is built.

Definition 2.1.2 (Natural Deduction System). Given a set of propositional variable, the relation $\Gamma \vdash \varphi$ is obtained by using the following axiom and inference rules

- *Axiom* (*axiom*)

  $$\frac{}{\varphi \vdash \varphi}\,(axiom)$$

  Intuitively, one can conclude proposition φ since it already appears in the context. Someone may give it a more general form "$\Gamma, \varphi \vdash \varphi$". However, this form can be obtained by using $\varphi \vdash \varphi$ and weakening which is given below as another inference rule.

- *Adding hypotheses into context* (*add*)

  $$\frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}\,(add)$$

  The property captured in this rule states that hypotheses of any derived conclusions can be extended with additional assumptions. This rule is also called *weakening*.

- *→-introduction* (→I)

  $$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\,(\to\text{I})$$

  If one can derive ψ from the context with ϕ as a hypothesis, then $\varphi \to \psi$ is derivable from the same context without φ.

- *→-elimination* (→E)

  $$\frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\,(\to\text{E})$$

If both the conditional claim "if φ then ψ" and φ are provided, one can conclude ψ. As mentioned in the beginning, this is a very common inference rule which is also called *modus ponens.*

- $\wedge$-*introduction* ($\wedge$I)

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ (\wedge\mathrm{I})$$

  If both φ and ψ are derivable from Γ, $\varphi \wedge \psi$ is also derivable.

- $\wedge$-*elimination* ($\wedge$E)

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\ (\wedge\mathrm{E}_1) \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\ (\wedge\mathrm{E}_2)$$

  Provided that conjunction $\varphi \wedge \psi$ is derivable from Γ, both of its components are also derivable.

- $\vee$-*introduction* ($\vee$I)

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ (\vee\mathrm{I}_1) \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ (\vee\mathrm{I}_2)$$

  One can conclude disjunction $\varphi \vee \psi$ from either φ or ψ.

- $\vee$-*elimination* ($\vee$E)

$$\frac{\Gamma \vdash \varphi \to \rho \qquad \Gamma \vdash \psi \to \rho \qquad \Gamma \vdash \varphi \vee \psi}{\Gamma \vdash \rho}\ (\vee\mathrm{E})$$

  If ρ follows φ, ρ follows ψ and $\varphi \vee \psi$, one can conclude ρ.

- $\bot$-*elimination* $\bot$E

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ (\bot\mathrm{E})$$

  From contradiction $\bot$, we can derive any propositions. This rule is also called *principle of explosion* or *ex falso quodlibet.*

What syntactically makes it different from classical propositional logic is that the *reductio ad absurdum rule* (*RAA*) is not a rule in the natural deduction system of intuitionistic logic, which makes some theorems in classical logic unprovable in intuitionistic logic.

2.1.2 Proofs

In intuitionistic logic, the judgement about propositions is not based on truth value. We say a proposition in intuitionistic propositional logic holds if we can construct a *proof tree*, also called a *proof*, by using the inference rules in definition 2.1.2.

Some examples are given here to demonstrate how a proof of a proposition is built:

(1) $\varphi, \neg\varphi \vdash \bot$

$$\dfrac{\dfrac{\dfrac{}{\varphi \vdash \varphi}}{\varphi, \neg\varphi \vdash \varphi}\,(add) \qquad \dfrac{\dfrac{\dfrac{}{\neg\varphi \vdash \neg\varphi}}{\varphi, \neg\varphi \vdash \neg\varphi}\,(add)}{\varphi, \neg\varphi \vdash \varphi \to \bot}\,(\neg\varphi \text{ stands for } \varphi \to \bot)}{\varphi, \neg\varphi \vdash \bot}\,(\to\mathrm{E})$$

(2) $\vdash \varphi \to \neg\neg\varphi$

$$\dfrac{\dfrac{\dfrac{}{\varphi, \neg\varphi \vdash \bot}\,(1)}{\varphi \vdash \neg\varphi \to \bot}\,(\to\mathrm{I})}{\varphi \vdash \neg\neg\varphi}\,(\neg\neg\varphi \text{ stands for } \neg\varphi \to \bot)$$

(3) $\vdash \psi \to (\varphi \to \psi)$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\psi \vdash \psi}}{\psi, \varphi \vdash \psi}\,(add)}{\psi \vdash \varphi \to \psi}\,(\to\mathrm{I})}{\vdash \psi \to (\varphi \to \psi)}\,(\to\mathrm{I})$$

(4) $\vdash \neg\varphi \to (\varphi \to \psi)$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\varphi, \varphi \vdash \bot}\,(1)}{\neg\varphi, \varphi \vdash \psi}\,(\bot\mathrm{E})}{\neg\varphi \vdash \varphi \to \psi}\,(\to\mathrm{I})}{\vdash \neg\varphi \to (\varphi \to \psi)}\,(\to\mathrm{I})$$

(5) $\vdash (\neg\varphi \vee \psi) \to (\varphi \to \psi)$

$$\dfrac{\dfrac{(3) \qquad (4) \qquad \dfrac{}{\neg\varphi \vee \psi \vdash \neg\varphi \vee \psi}}{\neg\varphi \vee \psi \vdash \varphi \to \psi}\,(\vee\mathrm{E})}{\vdash (\neg\varphi \vee \psi) \to (\varphi \to \psi)}\,(\to\mathrm{I})$$

Though double negation introduction “$\varphi \to \neg\neg\varphi$” is a theorem in intuitionistic propositional logic, double negation elimination “$\neg\neg\varphi \to \varphi$” is unprovable. In classical logic, double negation elimination can be proven by using the rule *reductio ad absurdum*. However, the natural deduction system in intuitionistic propositional logic does not contain *RAA*. That's reason why double negation elimination cannot be proven in this system.

By the same token, the law of excluded middle "$\varphi \vee \neg\varphi$" is also unprovable. We can look at it from an intuitive view. A proof of disjunction $\varphi \vee \psi$ is either a proof of φ or a proof of ψ. For an arbitrary proposition φ, we do not know whether φ has a proof or $\neg\varphi$ has a proof. Then we cannot give $\varphi \vee \neg\varphi$ a proof; therefore, it cannot be proven in intuitionistic propositional logic.

2.2 Lambda Calculus

The λ-calculus is a family of prototype programming languages. The simplest of these languages is the pure lambda calculus which studies only functions and their applicative behavior but does not contain any constants or types. The syntax of λ-*terms* in pure lambda calculus is simple. A λ-term can be a *term-variable*, an *application*, or an *abstraction*.

Application is one of the primitive operations. A lambda application FA denotes that the function F is applied to the argument A. Another basic operation is abstraction. Let $P \equiv P[x]$ be an expression possibly containing or depending on variable x. Then the lambda abstraction $\lambda x.P$ denotes the function $x \mapsto P[x]$. Here, P is called the *scope* of the *abstractor* λx. In a term M, if variable x occurs not in the scope of λx, we say x is *free* in M and write the set of free variables in M as $FV(M)$.

There are three kinds of equivalences playing an important role in λ-calculus. The first one, α-equivalence, states that a change of bound variables in a λ-term does not change its meaning. β-equivalence shows how to evaluate an application by using substitution. And η-equivalence refers to the idea of extensionality. All of them will be discussed in more detail in 2.2.2 which is about the proof system of the simply typed lambda calculus.

2.2.1 Simply Typed Lambda Calculus $\lambda^{\rightarrow}$

Church introduced a typed interpretation of lambda calculus, now called the *simply typed lambda calculus*, by giving each λ-term a unique type as its structure. The types in simply typed lambda calculus do not contain type variables. The standard type forms include functions, products, sums, initial and terminal types.

The one with only function type constructor $\rightarrow$ is called the *simply typed lambda calculus with function types*, indicated by $\lambda^{\rightarrow}$. With products, sums and functions, we have $\lambda^{\rightarrow,\times,+}$ and so on. However, $\lambda^{\rightarrow}$ is as expressive as other versions of simply typed lambda calculus. We will have a look at $\lambda^{\rightarrow}$ first and the introduce to the other types can be found in 2.2.3.

To begin with, we give the definition of types in $\lambda^{\rightarrow}$.

Definition 2.2.1 (**Types**)**.** Assume that a set of type-constants is given. Then the types in $\lambda^{\rightarrow}$ are defined as follows:

- each type-constant b is a type, called an *atom* (or *atomic type*);
- if σ and τ are types then $\sigma \rightarrow \tau$ is a type, called a *function type*.

There are two general frameworks for describing the denotational semantics of typed lambda calculus, *Henkin models* and *cartesian closed categories*. In a Henkin model, each type expression is interpreted as a set, the set of values of that type. But we will not go into Henkin models too much. The interpretation in CCCs will be discussed in 3.3.

The syntax of $\lambda^{\rightarrow}$ is essentially that of the pure lambda calculus itself. By assigning type σ to a lambda term M, we have an expression $M : \sigma$ called a *type-assignment*, saying that term M has type σ. Here, M is called its *subject* and σ its *predicate*.

However, not every pure λ-term can be given a type. The typing constraints are context sensitive. A *type-context* is any finite set of type-assignments $\Gamma = \{x_1 : \sigma_1, \cdots, x_n : \sigma_n\}$ whose subjects are term-variables. We say a type-context is consistent if no term-variable in it is the subject of more than one assignment. If not specified, the type-contexts used in this dissertation are consistent.

Since a type-context is a set, it does not change when its members are permuted or repeated. For notational convenience, the following abbreviations are often used:

• $x_1 : \sigma_1, \cdots, x_n : \sigma_n$	for	$\{x_1 : \sigma_1, \cdots, x_n : \sigma_n\}$;
• $\Gamma, x : \sigma$	for	$\Gamma \cup \{x : \sigma\}$;
• $\Gamma - x : \sigma$	for	$\Gamma - \{x : \sigma\}$.

Given a type-context Γ, a λ-term M and a type σ, the expression $\Gamma \triangleright M : \sigma$ is called a *typing judgement*, meaning term M has type σ in context Γ. However, not all the terms of this pattern are valid. To define well-typed lambda terms of a given type, some typing rules are needed.

Definition 2.2.2 (**Typing Rules of** $\lambda^{\rightarrow}$)**.** Assume that a set of term variables is provided. The well-typed terms in $\lambda^{\rightarrow}$ are defined simultaneously using the following axioms and inference rules:

- *Axiom* (*axiom*)

  For each term-variable x and each type σ,

  $$\frac{}{x : \sigma \rhd x : \sigma}\ (axiom)$$

  It simply says that if x has type σ in the context, intuitively, then x has type σ. In other words, a variable x has any type which it is declared to have.

- *Adding variables to type context* (*add*)

  Suppose x is not free in M,

  $$\frac{\Gamma \rhd M : \tau}{\Gamma, x : \sigma \rhd M : \tau}\ (add)$$

  In words, if M has type τ in context Γ, then it has type τ in context $\Gamma, x : \sigma$, which allows one to add an additional hypothesis to the type context.

- *$\rightarrow$-introduction* ($\rightarrow$I)

  $$\frac{\Gamma \rhd M : \tau}{\Gamma - x : \sigma \rhd \lambda x : \sigma . M : \sigma \rightarrow \tau}\ (\rightarrow\mathrm{I})$$

  If a term M specifies a result of type τ for all $x : \sigma$, then the expression $\lambda x : \sigma . M$ defines a function of type $\sigma \rightarrow \tau$.

- *$\rightarrow$-elimination* ($\rightarrow$E)

  $$\frac{\Gamma \rhd M : \sigma \rightarrow \tau \qquad \Gamma \rhd N : \sigma}{\Gamma \rhd MN : \tau}\ (\rightarrow\mathrm{E})$$

  It says that by applying any function of type $\sigma \rightarrow \tau$ to an argument of type σ, we obtain a result of type τ. Therefore, it is also called *function application.*

Some literatures represent the rule ($\rightarrow$E) as

$$\frac{\Gamma_1 \rhd M : \sigma \rightarrow \tau \qquad \Gamma_2 \rhd N : \sigma}{\Gamma_1 \cup \Gamma_2 \rhd MN : \tau}\ (\rightarrow\mathrm{E})$$

However, this representation may be dangerous. Suppose $x : \sigma \in \Gamma_1$ and $x : \tau \in \Gamma_2$ where $\sigma \not\equiv \tau$, then $\Gamma_1 \cup \Gamma_2$ is inconsistent since x has type σ and type τ at the same time. And that is why the contexts of M and N are same in definition 2.2.2.

The combinators **K** and **S** are used as examples to demonstrate how a well-typed term is obtained by using the typing rules:

(1) $\emptyset \rhd \lambda x : \sigma.\lambda y : \tau.x : \sigma \to \tau \to \sigma$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \sigma \rhd x : \sigma}}{x : \sigma, y : \tau \rhd x : \sigma}\,(add)}{x : \sigma \rhd \lambda y : \tau.x : \tau \to \sigma}\,(\to\mathrm{I})}{\emptyset \rhd \lambda x : \sigma.\lambda y : \tau.x : \sigma \to \tau \to \sigma}\,(\to\mathrm{I})$$

(2) $\emptyset \rhd \lambda x : \sigma \to \tau \to \rho.\lambda y : \sigma \to \tau.\lambda z : \sigma.(xz)(yz)$

$: (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$

$$\text{(I)}\quad \dfrac{\dfrac{\dfrac{}{x : \sigma \to \tau \to \rho \rhd x : \sigma \to \tau \to \rho}}{x : \sigma \to \tau \to \rho, y : \sigma \to \tau \rhd x : \sigma \to \tau \to \rho}\,(add)}{x : \sigma \to \tau \to \rho, y : \sigma \to \tau, z : \sigma \rhd x : \sigma \to \tau \to \rho}\,(add)$$

$$\text{(II)}\quad \dfrac{\dfrac{\dfrac{}{y : \sigma \to \tau \rhd y : \sigma \to \tau}}{x : \sigma \to \tau \to \rho, y : \sigma \to \tau \rhd y : \sigma \to \tau}\,(add)}{x : \sigma \to \tau \to \rho, y : \sigma \to \tau, z : \sigma \rhd y : \sigma \to \tau}\,(add)$$

$$\text{(III)}\quad \dfrac{\dfrac{\dfrac{}{z : \sigma \rhd z : \sigma}}{x : \sigma \to \tau \to \rho, z : \sigma \rhd z : \sigma}\,(add)}{x : \sigma \to \tau \to \rho, y : \sigma \to \tau, z : \sigma \rhd z : \sigma}\,(add)$$

$$\text{(IV)}\quad \dfrac{\text{(I)} \qquad \text{(III)}}{x : \sigma \to \tau \to \rho, y : \sigma \to \tau, z : \sigma \rhd xz : \tau \to \rho}\,(\to\mathrm{E})$$

$$\text{(V)}\quad \dfrac{\text{(II)} \qquad \text{(III)}}{x : \sigma \to \tau \to \rho, y : \sigma \to \tau, z : \sigma \rhd yz : \tau}\,(\to\mathrm{E})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{(IV)} \qquad \text{(V)}}{x : \sigma \to \tau \to \rho, y : \sigma \to \tau, z : \sigma \rhd (xz)(yz) : \rho}\,(\to\mathrm{E})}{x : \sigma \to \tau \to \rho, y : \sigma \to \tau \rhd \lambda z : \sigma.(xz)(yz) : \sigma \to \rho}\,(\to\mathrm{I})}{x : \sigma \to \tau \to \rho \rhd \lambda y : \sigma \to \tau.\lambda z : \sigma.(xz)(yz) : (\sigma \to \tau) \to \sigma \to \rho}\,(\to\mathrm{I})}{\emptyset \rhd \lambda x : \sigma \to \tau \to \rho.\lambda y : \sigma \to \tau.\lambda z : \sigma.(xz)(yz)}\,(\to\mathrm{I})$$

$: (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$

As mentioned before, not every term can be typed. Take the term xx for example. As an application, its first component x should have a function type $\sigma \to \tau$ and then its second x should have type σ. In order to construct a derivation tree, we need two sub-trees, one with conclusion $\Gamma \rhd x : \sigma \to \tau$ and another with conclusion $\Gamma \rhd x : \sigma$, in accordance with the rule ($\to$E). Since x is a term-variable, the axiom (*axiom*) tells us both $x : \sigma \to \tau$ and $x : \sigma$ appear in context Γ, which means Γ is inconsistent. Therefore, we are unable to find a consistent context for xx and it cannot be typed.

2.2.2 Equational Proof System of $\lambda^{\rightarrow}$

To derive equations of terms in $\lambda^{\rightarrow}$ that hold in all models, we need an equational proof system for $\lambda^{\rightarrow}$. A typed equation has the form $\Gamma \vdash M = N : \sigma$ where both M and N are assumed to have type σ in context Γ. Since type assignments are included in equations, we have an equational version of the typing rules that build well-typed equations in $\lambda^{\rightarrow}$.

The simply typed lambda calculus has the same theory of α-, β- and η-equivalence as the pure lambda calculus. Since α- and β-equivalence are defined by substitution, the definition of substitution is given first.

Definition 2.2.3 (Substitution). We define $[N/x]M$ to be the result of substituting N for each free occurrence x in M and making any changes of bound variables needed to prevent variables free in N from becoming bound in $[N/x]M$. More precisely, we define for all x, N, P, Q and all $y \not\equiv x$

- $[N/x]x \equiv N$;
- $[N/x]y \equiv y$;
- $[N/x](PQ) \equiv ([N/x]P)([N/x]Q)$;
- $[N/x](\lambda x.P) \equiv \lambda x.P$;
- $[N/x](\lambda y.P) \equiv \lambda y.[N/x]P$ if $x \in FV(P)$ and $y \notin FV(N)$;
- $[N/x](\lambda y.P) \equiv \lambda z.[N/x][z/y]P$ if $x \in FV(P)$ and $y \in FV(N)$.

In the last case, z can be any variable that $z \notin FV(P)$ and $z \notin FV(N)$.

For any $N_1, \cdots, N_n$ and any distinct $x_1, \cdots, x_n$, the result of simultaneously substituting all N_i for x_i ($i = 1, \cdots, n$) in term M is defined similarly to the definition above and denoted as $[N_1/x_1, \cdots, N_n/x_n]M$.

Definition 2.2.4 (α-equivalence). Given a variable y which is not free in M, we have

$$\lambda x.M =_\alpha \lambda y.[y/x]M$$

and the act of replacing an occurrence of $\lambda x.M$ in a term by $\lambda y.[y/x]M$ is called a *change of bound variables.* M and N are *α-equivalent*, notation $M =_\alpha N$, if N results from M by a series of changes of bound variables.

Definition 2.2.5 (**β-equivalence**).

- A *β-redex* is any sub-term of the form $(\lambda x.M)N$. It can be reduced by the following rule
$$(\lambda x.M)N \rightarrow_\beta [N/x]M$$
If P contains a β-redex $(\lambda x.M)N$ and Q is the result of replacing it by $[N/x]M$, we say P *β-contracts* to Q, denoted as $P \rightarrow_\beta Q$.
- A *β-reduction* of a term P is a finite or infinite ordered sequence of β-contractions, i.e. $P \rightarrow_\beta P_1 \rightarrow_\beta P_2 \rightarrow_\beta \cdots$. A finite β-reduction is *from P to Q* if it has $n \geq 1$ contractions and $P_n =_\alpha Q$ or it is empty and $P =_\alpha Q$. If there exists a reduction from P to Q, we say P *β-reduces* to Q, denoted as $P \twoheadrightarrow_\beta Q$. We can see that α-conversions are allowed in a β-reduction.
- P and Q are *β-equivalent*, notation $P =_\beta Q$, if P can be changed to Q by a finite sequence of β-reductions and reversed β-reductions (also called *β-expansions*).

A term may be able to β-reduce to different terms at the same time. For example, the term $(\lambda x.M)((\lambda y.N)P)$ can β-reduce (in one step) to $[((\lambda y.N)P)/x]M$ by substituting $((\lambda y.N)P)$ to x in M or $(\lambda x.M)([P/y]N)$ by substituting P to y in N. It is necessary for a calculus that the result of computation is independent from the order of reduction. This property holds for all λ-terms and is described in *Church-Rosser Theorem for β*.

Definition 2.2.6 (**η-equivalence**). An *η-redex* is any term of form $\lambda x.Mx$ with $x \notin FV(M)$. It can be reduced in accordance with the following rule
$$\lambda x.Mx \rightarrow_\eta M$$
The definition of *η-contracts*, *η-reduces* ($\twoheadrightarrow_\eta$), *η-equivalence* ($=_\eta$), etc. are similar to those of the corresponding β-concepts in definition 2.2.5. However, all η-reductions are finite while β-reductions may be infinite.

Similarly, the result of computation is independent from the order of η-reduction which is described in *Church-Rosser Theorem for η*.

Now, we can define the typing rules for typed equations in $\lambda^\rightarrow$.

Definition 2.2.7 (Typing Rules for Typed Equations in $\lambda^{\to}$). The typed equations in $\lambda^{\to}$ are generated by using the following typing rules:

- *Reflexivity* (ref)

$$\frac{}{\Gamma \triangleright M = M : \sigma}\ (ref)$$

- *Symmetry* (sym)

$$\frac{\Gamma \triangleright M = N : \sigma}{\Gamma \triangleright N = M : \sigma}\ (sym)$$

- *Transitivity* $(trans)$

$$\frac{\Gamma \triangleright M = N : \sigma \qquad \Gamma \triangleright N = P : \sigma}{\Gamma \triangleright M = P : \sigma}\ (trans)$$

As an equivalence relation, the equality in $\lambda^{\to}$ should have the above three properties, reflexivity, symmetry and transitivity.

- *Adding variables to type context* (add)

  suppose x is not free in M or N,

$$\frac{\Gamma \triangleright M = N : \tau}{\Gamma, x : \sigma \triangleright M = N : \tau}\ (add)$$

- *$\to$-introduction* ($\to$I)

$$\frac{\Gamma \triangleright M = N : \tau}{\Gamma - x : \sigma \triangleright \lambda x : \sigma.M = \lambda x : \sigma.N : \sigma \to \tau}\ (\to\mathrm{I})$$

  This rule says that if M and N are equal for all values of x, then the two functions $\lambda x : \sigma.M$ and $\lambda x : \sigma.N$ are equal, i.e. lambda abstraction preserves equality.

- *$\to$-elimination* ($\to$E)

$$\frac{\Gamma \triangleright M = M' : \sigma \to \tau \qquad \Gamma \triangleright N = N' : \sigma}{\Gamma \triangleright MN = M'N' : \tau}\ (\to\mathrm{E})$$

  It says that equals applied to equals yield equals, i.e. application preserves equality.

The three rules above can be seen as the equational versions of the typing rules corresponding to the ones for well-typed terms.

- *α-equivalence* (α)

  suppose y is not free in M,

$$\frac{}{\Gamma \triangleright \lambda x : \sigma.M = \lambda y : \sigma.[y/x]M : \sigma \to \tau}\ (\alpha)$$

  It allows one to rename bound variables.

- *β-equivalence* (β)

  $$\frac{}{\Gamma \triangleright (\lambda x : \sigma.M)N = [N/x]M : \tau}\,(\beta)$$

  It shows how to evaluate a function application using substitution.

- *η-equivalence* (η)
  suppose x is not free in M,

  $$\frac{}{\Gamma \triangleright \lambda x : \sigma.Mx = M : \sigma \to \tau}\,(\eta)$$

  It says that $\lambda x : \sigma.Mx$ and M define the same function, since by (β) we have $(\lambda x : \sigma.Mx)y = My$ for any argument $y : \sigma$.

2.2.3 Other Types

Except function type, different versions of the simply typed lambda calculus may have products, sums, initial and terminal types. Their definitions and the typing rules for them are listed below.

Definition 2.2.8 (Initial and Terminal Types). The *initial type*, denoted as *null*, is an empty type and, for each type σ, there is a unique term constant

$$\text{Zero}^{\sigma} : \mathit{null} \to \sigma$$

The *terminal type*, denoted as *unit*, is a type such that there is only one term associated with it, $* : \mathit{unit}$, and, for each type σ, there is a unique term constant

$$\text{One}^{\sigma} : \sigma \to \mathit{unit}$$

Definition 2.2.9 (Products). If σ and τ are types then $\sigma \times \tau$ is a type, called the *product* of σ and τ. Given $M : \sigma$ and $N : \tau$, the pair $\langle M, N\rangle$ has type $\sigma \times \tau$. The projection terms $\text{Proj}_1^{\sigma,\tau} : \sigma \times \tau \to \sigma$ and $\text{Proj}_2^{\sigma,\tau} : \sigma \times \tau \to \tau$ are the coordinate functions that return the first and second components in a pair. The typing rules for products are given as follows:

- *$\times$-introduction* ($\times$I)

  $$\frac{\Gamma \triangleright M : \sigma \qquad \Gamma \triangleright N : \tau}{\Gamma \triangleright \langle M, N\rangle : \sigma \times \tau}\,(\times\text{I})$$

- $\times$-*elimination* ($\times$E)

$$\frac{\Gamma \triangleright M : \sigma \times \tau}{\Gamma \triangleright \mathrm{Proj}_1^{\sigma,\tau} M : \sigma}\,(\times \mathrm{E}_1) \qquad \frac{\Gamma \triangleright M : \sigma \times \tau}{\Gamma \triangleright \mathrm{Proj}_2^{\sigma,\tau} M : \tau}\,(\times \mathrm{E}_2)$$

Definition 2.2.10 (Sums). If σ and τ are types then $\sigma + \tau$ is a type, called the *sum* of σ and τ. The term constants associated with sums are injections $\mathrm{Inleft}^{\sigma,\tau} : \sigma \to \sigma + \tau$ and $\mathrm{Inright}^{\sigma,\tau} : \tau \to \sigma + \tau$ that construct terms of type $\sigma + \tau$ from a term of type either σ or τ, and $\mathrm{Case}^{\sigma,\tau,\rho} : (\sigma + \tau) \to (\sigma \to \rho) \to (\tau \to \rho) \to \rho$. The typing rules for sums are given as follows:

- $+$-*introduction* ($+$I)

$$\frac{\Gamma \triangleright M : \sigma}{\Gamma \triangleright \mathrm{Inleft}^{\sigma,\tau} M : \sigma + \tau}\,(+\mathrm{I}_1) \qquad \frac{\Gamma \triangleright M : \tau}{\Gamma \triangleright \mathrm{Inright}^{\sigma,\tau} M : \sigma + \tau}\,(+\mathrm{I}_2)$$

- $+$-*elimination* ($+$E)

$$\frac{\Gamma \triangleright M : \sigma + \tau \qquad \Gamma \triangleright N : \sigma \to \rho \qquad \Gamma \triangleright P : \tau \to \rho}{\Gamma \triangleright \mathrm{Case}^{\sigma,\tau,\rho} MNP : \rho}\,(+\mathrm{E})$$

With initial type, terminal type, function types, products and sums, the type expressions in the *full simply typed lambda calculus* $\lambda^{null,unit,\to,\times,+}$ are given by the following grammar

$$\sigma \;::=\; b \mid null \mid unit \mid \sigma \to \sigma \mid \sigma \times \sigma \mid \sigma + \sigma$$

where b is the type constant.

The λ-terms in $\lambda^{null,unit,\to,\times,+}$ are given by

$$\begin{aligned} M \;::=\; & \mathrm{Zero} \mid * \mid \mathrm{One} \mid M\ M \mid \lambda x : \sigma.M \mid \langle M, M\rangle \mid \mathrm{Proj}_1\ M \mid \\ & \mathrm{Proj}_2\ M \mid \mathrm{Inleft}\ M \mid \mathrm{Inright}\ M \mid \mathrm{Case}\ M\ M\ M \end{aligned}$$

2.3 Categories

As a relatively young branch of mathematics, category theory studies in an abstract way the properties of particular mathematical structures. It seeks to express all mathematical concepts in terms of "object" and "morphisms" independently of what they are representing. Nowadays, categories appear in most branches of mathematics and many parts of computer science. For instance, topoi, a kind of category, can even serve as a foundation for mathematics. Cartesian closed categories, as another example, can work as a framework for describing the denotational semantics of typed lambda calculus, and more generally, programming languages.

2.3.1 Categories

The formal definition of categories is given first.

Definition 2.3.1 (Categories). A *category* $\mathcal{C}$ consists of

- a collection C^o of objects;
- a collection C^m of morphisms (also called arrows or maps) between objects, with two maps $dom, cod : C^m \to C^o$ which give the domain and codomain of a morphism (we write $f : A \to B$ to denote a morphism f with $dom(f) = A$ and $cod(f) = B$);
- a binary map "$\circ$", called composition, mapping each pair f, g of morphisms with $cod(f) = dom(g)$ to a morphism $g \circ f$ such that $dom(g \circ f) = dom(f)$ and $cod(g \circ f) = cod(g)$;

such that the following axioms hold:

- identity: for every object A, there exists a morphism $\mathrm{id}_A : A \to A$, called the identity morphism for A, such that $f = f \circ \mathrm{id}_A$ and $g = \mathrm{id}_A \circ g$ for any morphisms f and g with $dom(f) = cod(g) = A$;
- associativity: $h \circ (g \circ f) = (h \circ g) \circ f$ for every $f : A \to B$, $g : B \to C$, and $h : C \to D$.

For any objects A and B in a category $\mathcal{C}$, the collection of all morphisms $f : A \to B$ is called a *hom-set* and denoted as $\mathrm{Hom}_{\mathcal{C}}(A, B)$. A category is determined by its hom-sets.

Categorists use *diagrams* to express equations. In a diagram, a morphism $f : A \to B$ is represented as an arrow form point A to B, labeled f. A diagram *commutes* if the composition of the morphism along any path between two fixed objects is equal. The identity and associative laws in definition 2.3.1 can be represented by the commutative diagrams in figure 2.1.

$$\begin{tikzcd}
& A \arrow[r, "f"] & B \\
C \arrow[ur, "g"] \arrow[r, "g"'] & A \arrow[u, "\mathrm{id}_A"'] \arrow[ur, "f"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g \circ f"'] & B \arrow[d, "h \circ g"] \arrow[dl, "g"'] \\
C \arrow[r, "h"'] & D
\end{tikzcd}$$

Figure 2.1: Diagrams of identity and associativity

A common example of a category is SET which is the category whose objects are sets and whose morphisms are functions. The identity of object S in SET is the identity function $\mathrm{id}_S : S \to S$ such that $\mathrm{id}_S(s) = s$ for all $s \in S$. The composition of morphisms is the composition of functions, i.e. $(g \circ f)(x) = g(f(x))$. As a category, it satisfies the two category axioms:

i) $f = f \circ \mathrm{id}_A = \mathrm{id}_B \circ f$ for every $f : A \to B$
   The identity follows by using the definitions of composite functions and identity functions:
   $(f \circ \mathrm{id}_A)(a) = f(\mathrm{id}_A(a)) = f(a)$ and $(\mathrm{id}_B \circ f)(a) = \mathrm{id}_B(f(a)) = f(a)$.

ii) $h \circ (g \circ f) = (h \circ g) \circ f$ for every $f : A \to B$, $g : B \to C$, and $h : C \to D$
   The associativity follows from the fact that composition of functions is associative:
   $(h \circ (g \circ f))(a) = h((g \circ f)(a)) = h(g(f(a)))$ and
   $((h \circ g) \circ f)(a) = (h \circ g)(f(a)) = h(g(f(a)))$.

One typical use of categories is to consider categories whose objects are sets with mathematical structure and whose morphisms are functions that preserve that structure. One of the common examples is the category POS whose objects are posets and whose morphisms are monotone functions. It will be discussed later as an example of a CCC in 2.3.3.

2.3.2 Categorical Constructions

There are many categorical constructions, i.e. particular objects and morphisms that satisfy a given set of axioms, which enrich the language of Category Theory. When studying constructions, one observes that all concepts are defined by their relations with other objects, and these relations are established by the existence and the equality of particular morphisms. In this dissertation, the following fundamental categorical constructions will be considered.

The simplest among these is the notion of initial object and its dual, terminal object.

Definition 2.3.2 (**Initial and terminal objects**). Let $\mathcal{C}$ be a category. An object A in $\mathcal{C}$ is *initial* if, for any object B in $\mathcal{C}$, there is a unique morphism from A to B. An object A in $\mathcal{C}$ is *terminal* if, for any object B in $\mathcal{C}$, there is a unique morphism from B to A.

In this dissertation, terminal objects are denoted as *unit* and, for object A, the unique morphism is denoted as $\text{One}^A : A \to \mathit{unit}$.

In Set, the initial object is the empty set $\emptyset$, and the unique morphism with $\emptyset$ for its source is the empty function whose graph is empty. Any singleton set is terminal in Set since for any set S, there is exactly one function from S to this singleton set.

In set theory, we can form a cartesian product of two sets and define coordinate functions for it. Then we can even form a product function of two given functions which have the same domain. This motivates a general definition of categorical products (within a category).

Definition 2.3.3 (**Products**). Let A and B be objects in a category $\mathcal{C}$. The *product* of A and B is an object $A \times B$ together with two morphisms $\text{Proj}_1^{A,B} : A \times B \to A$ and $\text{Proj}_2^{A,B} : A \times B \to B$, and for every object C in $\mathcal{C}$, an operation $\langle \cdot, \cdot \rangle : \text{Hom}(C, A) \times \text{Hom}(C, B) \to \text{Hom}(C, A \times B)$ such that for every $f_1 : C \to A$, $f_2 : C \to B$, and $g : C \to A \times B$, the following equations hold:

- $\mathrm{Proj}_i \circ \langle f_1, f_2 \rangle = f_i$;
- $\langle \mathrm{Proj}_1 \circ g, \mathrm{Proj}_2 \circ g \rangle = g$.

Since equations in category theory can be represented by commutative diagrams, we can give another definition of categorical products based on diagrams: Let A and B be objects in a category $\mathcal{C}$. The product of A and B is an object $A \times B$ together with two morphisms $\mathrm{Proj}_1^{A,B} : A \times B \to A$ and $\mathrm{Proj}_2^{A,B} : A \times B \to B$, and for every object C in $\mathcal{C}$, an operation $\langle \cdot, \cdot \rangle : \mathrm{Hom}(C, A) \times \mathrm{Hom}(C, B) \to \mathrm{Hom}(C, A \times B)$ such that for every $f_1 : C \to A$ and $f_2 : C \to B$, the morphism $\langle f_1, f_2 \rangle : C \to A \times B$ is the unique g that makes the diagram in figure 2.2 commute.

$$\begin{tikzcd}
& C \arrow[dl, "f_1"'] \arrow[d, "g"'] \arrow[dr, "f_2"] & \\
A & A \times B \arrow[l, "\mathrm{Proj}_1"'] \arrow[r, "\mathrm{Proj}_2"] & B
\end{tikzcd}$$

Figure 2.2: Diagram of product $A \times B$

The cartesian product construction for morphisms can also be given a categorical definition. Given morphisms $f : A \to C$ and $g : B \to D$ the product $f \times g : A \times B \to C \times D$ is defined by $f \times g = \langle f \circ \mathrm{Proj}_1^{A,B}, g \circ \mathrm{Proj}_2^{A,B} \rangle$ whose correspondent commutative diagram is given in figure 2.3.

$$\begin{tikzcd}
A \arrow[d, "f"'] & A \times B \arrow[l, "\mathrm{Proj}_1"'] \arrow[r, "\mathrm{Proj}_2"] \arrow[d, "f \times g"'] & B \arrow[d, "g"] \\
C & C \times D \arrow[l, "\mathrm{Proj}_1"'] \arrow[r, "\mathrm{Proj}_2"] & D
\end{tikzcd}$$

Figure 2.3: Diagram of product of morphisms

Proposition 2.3.4. Let $\mathcal{C}$ be a category with products. Given $f_1 : A \to B$, $g_1 : B \to C$, $f_2 : A' \to B'$ and $g_2 : B' \to C'$, the equation $(g_1 \times g_2) \circ (f_1 \times f_2) = (g_1 \circ f_1) \times (g_2 \circ f_2)$ holds.

Proof.

$$\begin{array}{rl} & \mathrm{Proj}_1 \circ ((g_1 \times g_2) \circ (f_1 \times f_2)) \\ = & (\mathrm{Proj}_1 \circ (g_1 \times g_2)) \circ (f_1 \times f_2) \\ = & (g_1 \circ \mathrm{Proj}_1) \circ (f_1 \times f_2) \\ = & g_1 \circ (\mathrm{Proj}_1 \circ (f_1 \times f_2)) \\ = & g_1 \circ (f_1 \circ \mathrm{Proj}_1) \\ = & (g_1 \circ f_1) \circ \mathrm{Proj}_1 \end{array}$$

Similarly, we have $\mathrm{Proj}_2 \circ ((g_1 \times g_2) \circ (f_1 \times f_2)) = (g_2 \circ f_2) \circ \mathrm{Proj}_2$.

By the equation $\langle \mathrm{Proj}_1 \circ g, \mathrm{Proj}_2 \circ g \rangle = g$ in definition 2.3.3,

$$\begin{array}{rl} & (g_1 \times g_2) \circ (f_1 \times f_2) \\ = & \langle \mathrm{Proj}_1 \circ ((g_1 \times g_2) \circ (f_1 \times f_2)), \mathrm{Proj}_2 \circ ((g_1 \times g_2) \circ (f_1 \times f_2)) \rangle \\ = & \langle (g_1 \circ f_1) \circ \mathrm{Proj}_1, (g_2 \circ f_2) \circ \mathrm{Proj}_2 \rangle \end{array}$$

According to the definition of products of morphisms,

$$\begin{array}{rl} & (g_1 \circ f_1) \times (g_2 \circ f_2) \\ = & \langle (g_1 \circ f_1) \circ \mathrm{Proj}_1, (g_2 \circ f_2) \circ \mathrm{Proj}_2 \rangle \end{array}$$

Therefore, the equation $(g_1 \times g_2) \circ (f_1 \times f_2) = (g_1 \circ f_1) \times (g_2 \circ f_2)$ holds. □

The products in Set are the cartesian product of sets. Let A and B be two sets. The cartesian product $A \times B$ is the set of pair $\langle a, b \rangle$ with $a \in A$ and $b \in B$, together with the coordinate functions $\mathrm{Proj}_1 : A \times B \to A$ and $\mathrm{Proj}_2 : A \times B \to B$ such that $\mathrm{Proj}_1(\langle a, b \rangle) = a$ and $\mathrm{Proj}_2(\langle a, b \rangle) = b$. Given two functions $f_1 : C \to A$ and $f_2 : C \to B$, the function $\langle f_1, f_2 \rangle : C \to A \times B$ is defined by $\langle f_1, f_2 \rangle(c) = \langle f_1(c), f_2(c) \rangle$ for all $c \in C$. Then, given every $f_1 : C \to A$, $f_2 : C \to B$ and $g : C \to A \times B$, the equations in definition 2.3.3 are satisfied:

i) $\mathrm{Proj}_i \circ \langle f_1, f_2 \rangle = f_i$

$$\begin{array}{rl} & (\mathrm{Proj}_i \circ \langle f_1, f_2 \rangle)(c) \\ = & \mathrm{Proj}_i(\langle f_1, f_2 \rangle(c)) \\ = & \mathrm{Proj}_i(\langle f_1(c), f_2(c) \rangle) \\ = & f_i(c) \end{array}$$

for any $c \in C$.

ii) $\langle \mathrm{Proj}_1 \circ g, \mathrm{Proj}_2 \circ g \rangle = g$

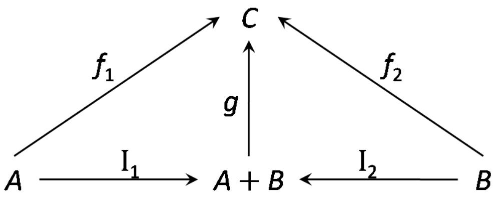
$$\begin{aligned} & \langle \mathrm{Proj}_1 \circ g, \mathrm{Proj}_2 \circ g \rangle(c) \\ = \; & \langle (\mathrm{Proj}_1 \circ g)(c), (\mathrm{Proj}_2 \circ g)(c) \rangle \\ = \; & \langle \mathrm{Proj}_1(g(c)), \mathrm{Proj}_2(g(c)) \rangle \\ = \; & g(c) \end{aligned}$$

for any $c \in C$.

Definition 2.3.5 (Coproducts). Let A and B be objects in a category $\mathcal{C}$. The *coproduct* of A and B is an object $A + B$ together with morphisms $\mathrm{I}_1^{A,B} : A \to A + B$ and $\mathrm{I}_2^{A,B} : B \to A + B$, and for every object C in $\mathcal{C}$ an operation $\langle \cdot | \cdot \rangle : \mathrm{Hom}(A, C) \times \mathrm{Hom}(B, C) \to \mathrm{Hom}(A + B, C)$ such that for every $f_1 : A \to C$, $f_2 : B \to C$ and $g : A + B \to C$, the following equations hold:

- $\langle f_1 | f_2 \rangle \circ \mathrm{I}_i = f_i$;
- $\langle g \circ \mathrm{I}_1 | g \circ \mathrm{I}_2 \rangle = g$.

The corresponding commutative diagram to the equations above is shown in figure 2.4, where $\langle f_1 | f_2 \rangle$ is the unique g.



$$\begin{tikzcd} & C & \\ A \arrow[ur, "f_1"] \arrow[r, "\mathrm{I}_1"] & A+B \arrow[u, "g"] & B \arrow[ul, "f_2"'] \arrow[l, "\mathrm{I}_2"'] \end{tikzcd}$$

Figure 2.4: Diagram of coproduct $A + B$

The coproducts in SET are the disjoint unions of sets. Let A and B be two sets. The disjoint union of them is defined by $A + B = \{\langle a, 1 \rangle | a \in A\} \cup \{\langle b, 2 \rangle | b \in B\}$ with two injection functions $\mathrm{I}_1 : A \to A + B$ that takes all a in A to $\langle a, 1 \rangle$ in $A + B$ and $\mathrm{I}_2 : B \to A + B$ that takes all b in B to $\langle b, 2 \rangle$ in $A + B$. Given two functions $f_1 : A \to C$ and $f_2 : B \to C$, the function $\langle f_1, f_2 \rangle : A + B \to C$ is defined by $\langle f_1, f_2 \rangle(\langle x, i \rangle) = f_i(x)$ for all $\langle x, i \rangle \in A + B$. Given any $f_1 : A \to C$, $f_2 : B \to C$ and $g : A + B \to C$, the equations in definition 2.3.5 are satisfied:

i) $\langle f_1|f_2\rangle \circ \mathrm{I}_i = f_i$

$$\begin{aligned} & (\langle f_1|f_2\rangle \circ \mathrm{I}_i)(x) \\ = \ & \langle f_1|f_2\rangle(\mathrm{I}_i(x)) \\ = \ & \langle f_1|f_2\rangle(\langle x, i\rangle) \\ = \ & f_i(x) \end{aligned}$$

for any $\langle x, i\rangle \in A + B$.

ii) $\langle g \circ \mathrm{I}_1|g \circ \mathrm{I}_2\rangle = g$

$$\begin{aligned} & \langle g \circ \mathrm{I}_1|g \circ \mathrm{I}_2\rangle(\langle x, i\rangle) \\ = \ & (g \circ \mathrm{I}_i)(x) \\ = \ & g(\mathrm{I}_i(x)) \\ = \ & g(\langle x, i\rangle) \end{aligned}$$

for any $\langle x, i\rangle \in A + B$.

One can form a set of functions which have the same domain and codomain. Similarly, the hom-set of morphisms may form an object. This idea brings our last basic construction, exponentials.

Definition 2.3.6 (**Exponentials**)**.** Let $\mathcal{C}$ be a category with products for all objects, and A and B be objects in $\mathcal{C}$. The *exponential*, also called *function object*, of A and B is an object $A \to B$ together with a morphism $\mathrm{App} : (A \to B) \times A \to B$, and for every object C in $\mathcal{C}$, an operation $\mathrm{Curry} : \mathrm{Hom}(C \times A, B) \to \mathrm{Hom}(C, A \to B)$ such that for every $h : C \times A \to B$ and $k : C \to (A \to B)$, the following equations hold:

- $\mathrm{App} \circ \langle \mathrm{Curry}(h) \circ \mathrm{Proj}_1, \mathrm{Proj}_2\rangle = h$;
- $\mathrm{Curry}(\mathrm{App} \circ \langle k \circ \mathrm{Proj}_1, \mathrm{Proj}_2\rangle) = k$.

Using the definition of products of morphisms, $f \times g = \langle f \circ \mathrm{Proj}_1, g \circ \mathrm{Proj}_2\rangle$, the two equations in definition 2.3.6 can be rewritten as $\mathrm{App} \circ (\mathrm{Cutty}(h) \times \mathrm{id}) = h$ and $\mathrm{Curry}(\mathrm{App} \circ (k \times \mathrm{id})) = k$.

The commutative diagram representing the equations in the definition is shown in figure 2.5,where the morphism $\mathrm{Curry}(h) : C \to (A \to B)$ is the unique k.

In Set, the exponent set $A \to B$ of A and B is the set of functions from A to B. The function $\mathrm{App} : (A \to B) \times A \to B$ is given by $\mathrm{App}(\langle f, a\rangle) = f(a)$

$$\begin{tikzcd}
C \times A \arrow[d, "k \times \mathrm{id}_A"'] \arrow[dr, "h"] & \\
(A \to B) \times A \arrow[r, "\mathrm{App}"] & B
\end{tikzcd}$$

Figure 2.5: Diagram of exponential $A \to B$

for all $f : A \to B$ and $a \in A$. Given a function $g : C \times A \to B$, the function $\mathrm{Curry}(g)$ is defined by $((\mathrm{Curry}(g))(c))(a) = g(\langle c, a \rangle)$ for all $c \in C$ and $a \in A$. Given any $h : C \times A \to B$ and $k : C \to (A \to B)$, the two equations in definition 2.3.6 hold:

i) $\mathrm{App} \circ \langle \mathrm{Curry}(h) \circ \mathrm{Proj}_1, \mathrm{Proj}_2 \rangle = h$

$$\begin{aligned}
& (\mathrm{App} \circ \langle \mathrm{Curry}(h) \circ \mathrm{Proj}_1, \mathrm{Proj}_2 \rangle)(\langle c, a \rangle) \\
= \ & \mathrm{App}(\langle \mathrm{Curry}(h) \circ \mathrm{Proj}_1, \mathrm{Proj}_2 \rangle(\langle c, a \rangle)) \\
= \ & \mathrm{App}(\langle (\mathrm{Curry}(h) \circ \mathrm{Proj}_1)(\langle c, a \rangle), \mathrm{Proj}_2(\langle c, a \rangle) \rangle) \\
= \ & \mathrm{App}(\langle \mathrm{Curry}(h)(\mathrm{Proj}_1(\langle c, a \rangle)), a \rangle) \\
= \ & \mathrm{App}(\langle (\mathrm{Curry}(h))(c), a \rangle) \\
= \ & ((\mathrm{Curry}(h))(c))(a) \\
= \ & h(\langle c, a \rangle)
\end{aligned}$$

for any $a \in A$ and $c \in C$.

ii) $\mathrm{Curry}(\mathrm{App} \circ \langle k \circ \mathrm{Proj}_1, \mathrm{Proj}_2 \rangle) = k$

$$\begin{aligned}
& ((\mathrm{Curry}(\mathrm{App} \circ \langle k \circ \mathrm{Proj}_1, \mathrm{Proj}_2 \rangle))(c))(a) \\
= \ & (\mathrm{App} \circ \langle k \circ \mathrm{Proj}_1, \mathrm{Proj}_2 \rangle)(\langle c, a \rangle) \\
= \ & \mathrm{App}(\langle k \circ \mathrm{Proj}_1, \mathrm{Proj}_2 \rangle(\langle c, a \rangle)) \\
= \ & \mathrm{App}(\langle (k \circ \mathrm{Proj}_1)(\langle c, a \rangle), \mathrm{Proj}_2(\langle c, a \rangle) \rangle) \\
= \ & \mathrm{App}(\langle k(\mathrm{Proj}_1(\langle c, a \rangle)), a \rangle) \\
= \ & \mathrm{App}(\langle k(c), a \rangle) \\
= \ & (k(c))(a)
\end{aligned}$$

for any $a \in A$ and $c \in C$.

2.3.3 Cartesian Closed Categories

Both products and exponentials have special importance for theories of computation. A two-argument function can be reduced to a one-argument function yielding a function from the second argument to the result. This passage is called *currying*. And exponentials give a categorical interpretation to the notion of currying. Therefore, categories with products and exponentials for every pair of objects are important enough to deserve a special name.

Definition 2.3.7 (**Cartesian Closed Categories**)**.** A category $\mathcal{C}$ is *cartesian closed* iff

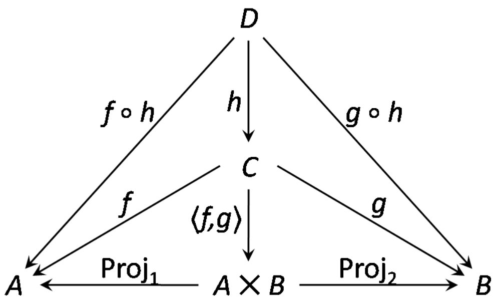
- it contains a terminal object *unit*;
- for every pair of objects A and B in $\mathcal{C}$, there is a product;
- for every pair of objects A and B in $\mathcal{C}$, there is an exponential.

Proposition 2.3.8. The following two identities hold in every cartesian closed category:

i) $\langle f, g\rangle \circ h = \langle f \circ h, g \circ h\rangle$
   where $f : C \to A$, $g : C \to B$ and $h : D \to C$;

ii) $\text{Curry}(f) \circ h = \text{Curry}(f \circ (h \times \text{id}))$
   where $f : A \times B \to C$ and $h : D \to A$.

Proof. These two equations have been proved by the diagrams in figures 2.6 and 2.7. But they can also be proved by using the equations given in definition 2.3.3 and 2.3.6 .



$$\begin{tikzcd}
 & D \arrow[dl, "f \circ h"'] \arrow[d, "h"] \arrow[dr, "g \circ h"] & \\
 & C \arrow[dl, "f"'] \arrow[d, "\langle f,g\rangle"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\text{Proj}_1"'] \arrow[r, "\text{Proj}_2"] & B
\end{tikzcd}$$

Figure 2.6: Diagram of $\langle f, g\rangle \circ h = \langle f \circ h, g \circ h\rangle$

$$\begin{tikzcd}
D \times A \arrow[d, "h \times \mathrm{id}_A"'] \arrow[ddr, "f \circ (h \times \mathrm{id}_A)"] & \\
C \times A \arrow[d, "\mathrm{Curry}(f) \times \mathrm{id}_A"'] \arrow[dr, "f"] & \\
(A \to B) \times A \arrow[r, "\mathrm{App}"] & B
\end{tikzcd}$$

Figure 2.7: Diagram of $\mathrm{Curry}(f) \circ h = \mathrm{Curry}(f \circ (h \times \mathrm{id}))$

i) $\mathrm{Proj}_1 \circ (\langle f, g\rangle \circ h) = (\mathrm{Proj}_1 \circ \langle f, g\rangle) \circ h = f \circ h$
$\mathrm{Proj}_2 \circ (\langle f, g\rangle \circ h) = (\mathrm{Proj}_2 \circ \langle f, g\rangle) \circ h = g \circ h$
$\langle \mathrm{Proj}_1 \circ (\langle f, g\rangle \circ h), \mathrm{Proj}_2 \circ (\langle f, g\rangle \circ h)\rangle = \langle f \circ h, g \circ h\rangle$
Since we have $\langle \mathrm{Proj}_1 \circ g, \mathrm{Proj}_2 \circ g\rangle = g$ (in definition 2.3.3), then $\langle f, g\rangle \circ h = \langle f \circ h, g \circ h\rangle$ holds.

ii) $f = \mathrm{App} \circ (\mathrm{Curry}(f) \times \mathrm{id}))$ (by $\mathrm{App} \circ (\mathrm{Cutty}(h) \times \mathrm{id}) = h$ in definition 2.3.6)

$$\begin{aligned}
& \mathrm{Curry}(f \circ (h \times \mathrm{id})) \\
= \; & \mathrm{Curry}((\mathrm{App} \circ (\mathrm{Curry}(f) \times \mathrm{id}))) \circ (h \times \mathrm{id})) \\
= \; & \mathrm{Curry}(\mathrm{App} \circ ((\mathrm{Curry}(f) \times \mathrm{id})) \circ (h \times \mathrm{id}))) \\
& \text{(by proposition 2.3.4 } (g_1 \times g_2) \circ (f_1 \times f_2) = (g_1 \circ f_1) \times (g_2 \circ f_2)) \\
= \; & \mathrm{Curry}(\mathrm{App} \circ ((\mathrm{Curry}(f) \circ h) \times (\mathrm{id} \circ \mathrm{id}))) \\
= \; & \mathrm{Curry}(\mathrm{App} \circ ((\mathrm{Curry}(f) \circ h) \times \mathrm{id})) \\
& \text{(by } \mathrm{Curry}(\mathrm{App} \circ (k \times \mathrm{id})) = k \text{ in definition 2.3.6)} \\
= \; & \mathrm{Curry}(f) \circ h
\end{aligned}$$

□

The following gives some examples and one non-example of CCCs.

(1) Set

Set has been already given as an example of each categorical construction. It is cartesian closed since it satisfies the three conditions:

- Any singleton set can be the terminal object (it does not matter which one it exactly is since all the singleton sets are isomorphic);

- The product of sets A and B is the cartesian product of A and B;
- The exponential of sets A and B is the set of functions from A to B.

(2) Pos

Pos is the category whose objects are posets and whose morphisms are monotone maps. The identity of object $\mathcal{A} = (A, \leq_A)$ in Pos is the identity map $\text{id}_A : A \to A$ such that $\text{id}_A(a) = a$ for all $a \in A$. The composition of morphisms is the composition of maps which still preserves the monotonicity property.

Pos is cartesian closed:

- Any singleton poset can be the terminal object *unit*. Then the map $\text{One}^A : A \to unit$ from any poset $\mathcal{A}$ to this singleton poset is unique.
- The product $\mathcal{A} \times \mathcal{B} = (A \times B, \leq_{A\times B})$ of posets $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ is the cartesian product of sets A and B with the ordering $\leq_{A\times B}$ which is defined by $\langle a, b\rangle \leq_{A\times B} \langle a', b'\rangle)$ iff $a \leq_A a'$ and $b \leq_B b'$.
  The projections are the coordinate functions $\text{Proj}_1 : A \times B \to A$ and $\text{Proj}_2 : A \times B \to B$ which just return one of the two components in the pair. So they preserve the monotonicity.
  Given two monotone maps $f_1 : A \to B$ and $f_2 : A \to C$, the map $\langle f_1, f_2\rangle : A \to B \times C$ is defined by $\langle f_1, f_2\rangle(a) = \langle f_1(a), f_2(a)\rangle$ for all $a \in A$. Suppose $a, a' \in A$ and $a \leq_A a'$, then $f_1(a) \leq_B f_1(a')$ and $f_2(a) \leq_C f_2(a')$. According to the above definition, we have $\langle f_1, f_2\rangle(a) \leq_{B\times C} \langle f_1, f_2\rangle(a')$. Hence, the map $\langle f_1, f_2\rangle$ is monotone.
  In the same way as Set, Pos satisfies the equations in definition 2.3.3.
- The exponential $\mathcal{A} \to \mathcal{B} = (A \to B, \leq_{A\to B})$ of posets $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ is the set of monotone maps from A to B and a ordering $\leq_{A\to B}$ defined by, given $f, f' : A \to B$, $f \leq_{A\to B} f'$ iff $f(a) \leq_B f'(a)$ for all $a \in A$.
  Given a monotone map $g : A \times B \to C$, the map $\text{Curry}(g) : A \to$

$(B \to C)$ is defined by $((\text{Curry}(g))(a))(b) = g(\langle a, b\rangle)$ for all $a \in A$ and $b \in B$. Suppose $a, a' \in A$, $b \in B$ and $a \leq_A a'$, we have $(\text{Curry}(g))(a) \leq_{B\to C} (\text{Curry}(g))(a')$ because $((\text{Curry}(g))(a))(b) = g(\langle a, b\rangle) \leq_C g(\langle a', b\rangle) = ((\text{Curry}(g))(a'))(b)$ for any $b \in B$; therefore, Curry(g) is monotone.

The map $\text{App} : (B \to C)\times B \to C$ is defined by $\text{App}(\langle h, b\rangle) = h(b)$ for all $h : B \to C$ and $b \in B$. Suppose $h, h' : B \to C$ and $b.b' \in B$ with $h \leq_{B\to C} h'$ and $b \leq_B b'$, we have $\langle h, b\rangle \leq_{(B\to C)\times B} \langle h', b'\rangle$ by the definition of $\leq_{(B\to C)\times B}$ and then $\text{App}(\langle h, b\rangle) = h(b) \leq_C h(b') \leq_C h'(b') = \text{App}(\langle h', b'\rangle)$; therefore, App is also monotone.

Pos also satisfies the equations in definition 2.3.6. The proof is carried out in the same way of Set.

(3) $\text{Pos}_\perp$

$\text{Pos}_\perp$ is same as the category Pos except that each poset has a least element $\perp$. We construct its identity, composition, terminal object, product, exponential in the same way of Pos.

However, the least elements in products and exponential should be defined particularly. If $\perp_A$ and $\perp_B$ are least elements in posets $\mathcal{A}$ and $\mathcal{B}$, the least element in product $\mathcal{A} \times \mathcal{B}$ is $\perp_{A\times B} = \langle \perp_A, \perp_B\rangle$. The least element $\perp_{A\to B}$ in exponential $\mathcal{A} \to \mathcal{B}$ is the map converting any $a \in A$ to $\perp_B$, i.e. $\perp_{A\to B} = \lambda a : A.\perp_B$.

Since the definition of morphisms in $\text{Pos}_\perp$ is still the same as the one in Pos, $\text{Pos}_\perp$ should be cartesian closed.

(4) $\text{Pos}_\perp!$

In category $\text{Pos}_\perp!$, the objects are posets with least elements and the morphisms are monotone maps with a property that the least element $\perp$ in one poset is mapped to $\perp$ in another one, i.e. for every monotone map $f : A \to B$, $f(\perp_A) = \perp_B$.

Though it looks similar to $\text{Pos}_\perp$, $\text{Pos}_\perp!$ is not cartesian closed. Assume $\text{Pos}_\perp!$ has exponentials, then for any morphism $f : A \times B \to C$ there should be a unique $g = \text{Curry}(f) : A \to (B \to C)$ which is isomorphic to f. Since $g(\perp_A) = \perp_{B\to C}$ and $\perp_{B\to C}(b) = \perp_C$ for any $b \in B$, we have $(g(\perp_A))(b) = \perp_C$. However, f dose not map all $\langle \perp_A, b\rangle$ to $\perp_C$, which means that $f \ncong g$. Therefore, $\textsc{Pos}_\perp!$ does not have exponentials and is not cartesian closed.

3 Correspondences

3.1 Every type-derivation in $\lambda^{\to}$ leads to a proof in intuitionistic implicational logic

The proof trees in 2.1.1 have a similar shape to the type-derivation trees in 2.2.1. In fact, the rules in natural deduction system to construct relations $\Gamma \vdash \varphi$ work in the same way as the typing rules to build well-typed λ-terms $\Gamma \rhd x : \sigma$. Let us have a look at the simplest example first, the type-derivations in simply typed lambda calculus $\lambda^{\to}$ and the proofs in intuitionistic implicational logic.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\sigma \rhd x:\sigma}}{x:\sigma, y:\tau \rhd x:\sigma}\,(add)}{x:\sigma \rhd \lambda y:\tau.x : \tau \to \sigma}\,(\to\mathrm{I})}{\emptyset \rhd \lambda x:\sigma.\lambda y:\tau.x : \sigma \to (\tau \to \sigma)}\,(\to\mathrm{I})$$

The tree above is a type-derivation of λ-term $\lambda x : \sigma.\lambda y : \tau.x : \sigma \to (\tau \to \sigma)$. Let us erase the terms in the tree. If we consider every type-symbol as a propositional variable and the relation "$\rhd$" as "$\vdash$", then the tree in the right side is exactly a proof tree, which means we obtain a proof of the conclusion represented by the type of the term in the bottom of the tree. Here, the proven proposition is $\sigma \to (\tau \to \sigma)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cancel{x:}\,\sigma \rhd \cancel{x:}\,\sigma}}{\cancel{x:}\,\sigma, \cancel{y:}\,\tau \rhd \cancel{x:}\,\sigma}\,(add)}{\cancel{x:}\,\sigma \rhd \cancel{\lambda y:\tau.x:}\,\tau \to \sigma}\,(\to\mathrm{I})}{\emptyset \rhd \cancel{\lambda x:\sigma.\lambda y:\tau.x:}\,\sigma \to (\tau \to \sigma)}\,(\to\mathrm{I}) \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\sigma \rhd \sigma}}{\sigma, \tau \rhd \sigma}\,(add)}{\sigma \rhd \tau \to \sigma}\,(\to\mathrm{I})}{\emptyset \rhd \sigma \to (\tau \to \sigma)}\,(\to\mathrm{I})$$

Proposition 3.1.1. Every type-derivation in $\lambda^{\to}$ leads to a proof in intuitionistic implicational logic.

Proof. This proposition can be proven by induction on the height of type-derivation trees. In order to make the proof more readable, the types are denoted by using the propositional variable symbols $(\varphi, \psi ...)$.

The base case is the one when the height of the tree is 1, i.e. a type-derivation without hypotheses which is the typing rule (*axiom*). By erasing the term and replacing the relation symbol "$\triangleright$" by "$\vdash$", we obtain the axiom in the natural deduction system (the right tree).

$$\frac{}{x : \varphi \triangleright x : \varphi}\,(axiom) \quad \Longrightarrow \quad \frac{}{\varphi \vdash \varphi}\,(axiom)$$

Suppose for very type-derivation tree of height at most n, we get a corresponding proof by erasing the terms in the tree. Then, we want to show that the tree of height $n+1$ also leads to a corresponding proof. According to the typing rules of $\lambda^{\to}$, there are two inductive cases, one with lambda abstraction and another with application:

(1) Lambda abstraction

Assume that the following type-derivation of height n leads to a proof on its right side,

$$\frac{\vdots}{\Gamma, x : \varphi \triangleright M : \psi} \quad \Longrightarrow \quad \frac{\vdots}{\Gamma', \varphi \vdash \psi}$$

Since the proof tree is obtained by erasing the terms in the derivation tree, the context Γ' should contain all the types in Γ but nothing else. By using introduction rule for function type on $\Gamma, x : \varphi \triangleright M : \psi$, we have a type-derivation of term $\lambda x : \varphi.M$

$$\cfrac{\cfrac{\vdots}{\Gamma, x : \varphi \triangleright M : \psi}}{\Gamma \triangleright \lambda x : \varphi.M : \varphi \to \psi}\,(\to\mathrm{I})$$

By using introduction rule for implication on $\Gamma', \varphi \vdash \psi$, we have the following proof

$$\cfrac{\cfrac{\vdots}{\Gamma', \varphi \vdash \psi}}{\Gamma' \vdash \varphi \to \psi}\,(\to\mathrm{I})$$

It is obvious that $\Gamma' \vdash \varphi \to \psi$ can also be obtained by deleting the terms in Γ which is Γ' and the term $\lambda x : \varphi.M$. Hence, the derivation of $\Gamma \triangleright \lambda x : \varphi.M : \varphi \to \psi$ leads to a proof of $\Gamma' \vdash \varphi \to \psi$.

(2) Application

Assume the following two derivation-trees, one of which has height n and another has height at most n, lead to two proofs

$$\frac{\vdots}{\Gamma \triangleright M : \varphi \to \psi} \quad \Longrightarrow \quad \frac{\vdots}{\Gamma' \vdash \varphi \to \psi}$$

$$\dfrac{\vdots}{\Gamma \rhd N : \varphi} \quad \Longrightarrow \quad \dfrac{\vdots}{\Gamma' \vdash \varphi}$$

Similarly, Γ' contains all the types in Γ. Then, according to function application, we obtain a derivation of MN

$$\dfrac{\dfrac{\vdots}{\Gamma \rhd M : \varphi \to \psi} \qquad \dfrac{\vdots}{\Gamma \rhd N : \varphi}}{\Gamma \rhd MN : \psi}\ (\to\text{E})$$

We also get a proof of ψ by applying *modus ponens* on $\varphi \to \psi$ and φ

$$\dfrac{\dfrac{\vdots}{\Gamma' \vdash \varphi \to \psi} \qquad \dfrac{\vdots}{\Gamma' \vdash \varphi}}{\Gamma' \vdash \psi}\ (\to\text{E})$$

Then we can see that if we erase the terms in Γ and term MN, we also have $\Gamma' \vdash \psi$. Hence, the derivation of $\Gamma \rhd MN : \psi$ leads to a proof of $\Gamma' \vdash \psi$.

Therefore, every type-derivation in $\lambda^{\to}$ leads to a proof in intuitionistic implicational logic. □

3.2 Every proof in intuitionistic propositional logic can be encoded by a lambda term

Proposition 3.1.1 in last section tells us that, from every type-derivation, one can obtain a proof. In this section, we will have a look at the inverse direction and extend this connection to intuitionistic propositional logic and full simply typed lambda calculus.

$$\cfrac{\cfrac{\cfrac{\overline{\varphi \vdash \varphi}}{\varphi, \tau \vdash \varphi}\,(add)}{\varphi \vdash \tau \to \varphi}\,{\to}\mathrm{I}}{\vdash \varphi \to (\tau \to \varphi)}\,{\to}\mathrm{I}$$

Given the above proof of proposition $\varphi \to (\tau \to \varphi)$, we construct a type-derivation tree in the following steps:

(1) Treat each proposition in the context as a type and assign a term variable to it. Atomic propositions are considered as atomic types while implications, conjunctions, disjunctions and contradiction correspond to function types, product types, sum types and initial type, respectively. The same term variables are assigned to the same propositions in all the contexts through out the whole tree. Each term variable is assigned to one proposition only in order to make the type context consistent.

$$\cfrac{\cfrac{\cfrac{\overline{x : \varphi \vdash \varphi}}{x : \varphi, y : \tau \vdash \varphi}\,(add)}{x : \varphi \vdash \tau \to \varphi}\,{\to}\mathrm{I}}{\vdash \varphi \to (\tau \to \varphi)}\,{\to}\mathrm{I}$$

(2) Use the type assignments in the context Γ to construct a proper term M of type φ from the top of the tree. Then a typing judgement $\Gamma \vdash M : \varphi$ is obtained. If the proposition (or type) on the right side of $\vdash$ already exists in the context, then the term of that type is the term variable which has been assign to that type in the context.

$$\cfrac{\cfrac{\cfrac{\overline{x : \varphi \vdash x : \varphi}}{x : \varphi, y : \tau \vdash x : \varphi}\,(add)}{x : \varphi \vdash ? : \tau \to \varphi}\,{\to}\mathrm{I}}{\vdash ? : \varphi \to (\tau \to \varphi)}\,{\to}\mathrm{I}$$

In the third typing judgement (counted from the top of the tree), the proposition $\tau \to \varphi$ does not appear in the context. So the term of type $\tau \to \varphi$ is not a term variable but a lambda abstraction where the bound variable in the abstractor has type τ and the scope has type φ. But τ does not appear in the context, either. We need to go up to look for the variable which is assigned to τ in the context and the term which has type φ. From the second typing judgement, we know y is assigned to τ and term x has type φ. Therefore, we construct the term in the third typing judgement as $\lambda y : \tau.x : \tau \to \varphi$. The last typing judgement is obtained in the same way.

$$\cfrac{\cfrac{\cfrac{\overline{x : \varphi \vdash x : \varphi}}{x : \varphi, y : \tau \vdash x : \varphi}\,(add)}{x : \varphi \vdash \lambda y : \tau.x : \tau \to \varphi}\,{\to}\mathrm{I}}{\vdash \lambda x : \varphi.\lambda y : \tau.x : \varphi \to (\tau \to \varphi)}\,{\to}\mathrm{I}$$

(3) Replace the relation "$\vdash$" by "$\rhd$".

$$\cfrac{\cfrac{\cfrac{\overline{x : \varphi \rhd x : \varphi}}{x : \varphi, y : \tau \rhd x : \varphi}\,(add)}{x : \varphi \rhd \lambda y : \tau.x : \tau \to \varphi}\,{\to}\mathrm{I}}{\rhd \lambda x : \varphi.\lambda y : \tau.x : \varphi \to (\tau \to \varphi)}\,{\to}\mathrm{I}$$

Since every lambda term determines a type-derivation, rather than saying that every proof can be obtained from a type-derivation, we can say that every proof can be encoded in a lambda term.

Proposition 3.2.1. Every proof in intuitionistic propositional logic can be encoded by a lambda term in full simply typed lambda calculus.

Proof. Its proof is also carried out by induction on the height of proof tree. In this proof, the type symbols $(\sigma, \tau, \rho \cdots)$ are adopted as proposition symbols in order to make the proof easier to read.

The base case is the a proof of height 1 which is the axiom in the natural deduction system. In $(axiom)$, the proposition in the right side of $\vdash$ also appears in the context. Both of them are assigned to a same term variable. By replacing $\vdash$ by $\rhd$, we obtain the typing rule $(axiom)$.

$$\cfrac{}{\sigma \vdash \sigma}\,(axiom) \implies \cfrac{}{x : \sigma \rhd x : \sigma}\,(axiom)$$

In the inductive step, let us assume that all the proofs of height at most n can be encoded in a corresponding lambda term. Then what we want to prove is that the proof of height $n+1$ can be encoded in a lambda term as well. Except the axiom, the natural deduction system of intuitionisitic propositional logic has several other inference rules for all the propositional connectives, $\to$, $\wedge$, $\vee$, and $\perp$. Each them is an inductive case.

(1) ($\to$I)

The induction hypothesis here is that the proof of height n with conclusion $\Gamma, \sigma \vdash \tau$ can be encoded in $\Gamma', x : \sigma \rhd M : \tau$ where Γ' is a consistent type context whose types are propositions in Γ.

$$\frac{\vdots}{\Gamma, \sigma \vdash \tau} \quad \Longrightarrow \quad \frac{\vdots}{\Gamma', x : \sigma \rhd M : \tau}$$

By using the introduction rule of implication ($\to$I), we get the following proof whose height is $n+1$.

$$\cfrac{\cfrac{\vdots}{\Gamma, \sigma \vdash \tau}}{\Gamma \vdash \sigma \to \tau}\ (\to\text{I})$$

According to the typing rule of introducing function type ($\to$I), we get another typing judgement as follows.

$$\cfrac{\cfrac{\vdots}{\Gamma', x : \sigma \rhd M : \tau}}{\Gamma' \rhd \lambda x : \sigma . M : \sigma \to \tau}\ (\to\text{I})$$

Since all the types in type context Γ' are the propositions in Γ and the term $\lambda x : \sigma.M$ has type $\sigma \to \tau$, the proof of $\Gamma \vdash \sigma \to \tau$ is encoded in $\Gamma' \rhd \lambda x : \sigma.M : \sigma \to \tau$.

(2) ($\to$E)

In this case, we assume the two proofs below, one of which has height n and another has height at most n, can be encoded in the terms next to them.

$$\frac{\vdots}{\Gamma \vdash \sigma \to \tau} \quad \Longrightarrow \quad \frac{\vdots}{\Gamma' \rhd M : \sigma \to \tau}$$

$$\frac{\vdots}{\Gamma \vdash \sigma} \quad \Longrightarrow \quad \frac{\vdots}{\Gamma' \rhd N : \sigma}$$

We use modus ponens ($\to$E) and then have a proof of $\Gamma \vdash \tau$.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \sigma \to \tau} \quad \dfrac{\vdots}{\Gamma \vdash \sigma}}{\Gamma \vdash \tau}\ (\to\mathrm{E})$$

We use function application ($\to$E) and then have a type derivation with conclusion $\Gamma' \triangleright MN : \tau$.

$$\dfrac{\dfrac{\vdots}{\Gamma' \triangleright N : \sigma} \quad \dfrac{\vdots}{\Gamma' \triangleright N : \sigma}}{\Gamma' \triangleright MN : \tau}\ (\to\mathrm{E})$$

Γ' encoded all the propositions in Γ and MN has type τ; therefore, the proof of $\Gamma \vdash \tau$ is encoded in $\Gamma' \triangleright MN : \tau$.

(3) ($\wedge$I)

Assume the two following proofs, one of which has height n and another has height at most n, can be encoded in the terms next to them.

$$\dfrac{\vdots}{\Gamma \vdash \sigma} \quad \Longrightarrow \quad \dfrac{\vdots}{\Gamma' \triangleright M : \sigma}$$

$$\dfrac{\vdots}{\Gamma \vdash \tau} \quad \Longrightarrow \quad \dfrac{\vdots}{\Gamma' \triangleright N : \tau}$$

By introduction rule of conjunction ($\wedge$I), we get a proof of $\Gamma \vdash \sigma \wedge \tau$.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \sigma} \quad \dfrac{\vdots}{\Gamma \vdash \tau}}{\Gamma \vdash \sigma \wedge \tau}\ (\wedge\mathrm{I})$$

By introduction rule of product type ($\times$I), we get a type derivation of $\Gamma' \triangleright \langle M, N \rangle : \sigma \times \tau$.

$$\dfrac{\dfrac{\vdots}{\Gamma' \triangleright M : \sigma} \quad \dfrac{\vdots}{\Gamma' \triangleright N : \tau}}{\Gamma' \triangleright \langle M, N \rangle : \sigma \times \tau}\ (\wedge\mathrm{I})$$

Then, we can see that the proof of $\Gamma \vdash \sigma \wedge \tau$ can be encoded in $\Gamma' \triangleright \langle M, N \rangle : \sigma \times \tau$.

(4) ($\wedge$E)

Assume the following proof of height n can be encoded in the term next to it.

$$\dfrac{\vdots}{\Gamma \vdash \sigma \wedge \tau} \quad \Longrightarrow \quad \dfrac{\vdots}{\Gamma' \triangleright M : \sigma \times \tau}$$

By the elimination rule of conjunction ($\wedge$E), we get a proof of $\Gamma \vdash \sigma$ and another one of $\Gamma \vdash \tau$.

$$\frac{\dfrac{\vdots}{\Gamma \vdash \sigma \wedge \tau}}{\Gamma \vdash \sigma}(\wedge\mathrm{E}_1) \qquad \frac{\dfrac{\vdots}{\Gamma \vdash \sigma \wedge \tau}}{\Gamma \vdash \tau}(\wedge\mathrm{E}_2)$$

By the elimination rule of product type ($\times$E), we get a type derivation of $\Gamma' \triangleright \mathrm{Proj}_1 M : \sigma$ and another one of $\Gamma' \triangleright \mathrm{Proj}_2 M : \tau$.

$$\frac{\dfrac{\vdots}{\Gamma' \triangleright M : \sigma \times \tau}}{\Gamma' \triangleright \mathrm{Proj}_1 M : \sigma}(\times\mathrm{E}_1) \qquad \frac{\dfrac{\vdots}{\Gamma' \triangleright M : \sigma \times \tau}}{\Gamma' \triangleright \mathrm{Proj}_2 M : \tau}(\times\mathrm{E}_2)$$

Hence, the proof of $\Gamma \vdash \sigma$ can be encoded in $\Gamma' \triangleright \mathrm{Proj}_1 M : \sigma$ and the proof of $\Gamma \vdash \tau$ in $\Gamma' \triangleright \mathrm{Proj}_2 M : \tau$.

(5) ($\vee$I)

...

(6) ($\vee$E)

...

(7) ($\bot$E)

...

Every proof in intuitionistic propositional logic can be encoded by a lambda term in full simply typed lambda calculus. □

3.3 Every well-typed lambda term can be interpreted as a morphism in a CCC

As mentioned in the previous section, cartesian closed categories can work as a more general but also more abstract framework for describing the denotational semantics of typed lambda calculus.

The lambda calculus $\lambda^{unit,\times,\rightarrow}$ has terminal type, product types and function types. Correspondently, a CCC has terminal object, products and exponentials. A closed relation between them seems to be obvious. However, the lambda calculus $\lambda^{\rightarrow}$ with only function type constructor is as expressive as $\lambda^{unit,\times,\rightarrow}$. Therefore, both the type expressions and well-typed lambda terms in $\lambda^{\rightarrow}$ can be interpreted in any CCC, and this interpretation should be sound and complete.

References

[1] A. Asperti and G. Longo. *Categories, Types, and Structures.* The MIT Press, 1991.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier, revised edition, 1985.

[3] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice Hall, 2nd edition, 1995.

[4] J. R. Hindley. *Basic Simple Type Theory.* Cambridge University Press, 1997.

[5] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2nd edition, 2004.

[6] A. Jung. A short introduction to the lambda calculus, March 2004. Notes from the second year course *Models of Computation*, taught at the University of Birmingham, 98/99 to 03/04.

[7] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*, chapter Cartesian closed categories and λ-calculus. Cambridge University Press, 1986.

[8] J. C. Mitchell. *Foundations fro Programming Languages.* The MIT Press, 1996.

[9] B. C. Pierce. *Basic Category Theory for Computer Science.* The MIT Press, 1991.

[10] A. M. Pitts. *Handbook of Logic in Computer Science*, volume 5, chapter Categorical Logic. Clarendon Press, 2000.

[11] D. E. Rydeheard and R. M. Burstall. *Computational category Theory.* Prentice Hall, 1988.

[12] M. H. B. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 1998.

[13] D. van Dalen. *Logic and Structure*. Springer, 3rd edition, 1997.

[14] D. van Dalen. *Philosophical Logic*, chapter Intuitionistic Logic. Blackwell, 2001.