The Curry-Howard Isomorphism

Student: Chuangjie Xu (1061493)

Supervisor: Achim Jung

February 1, 2011

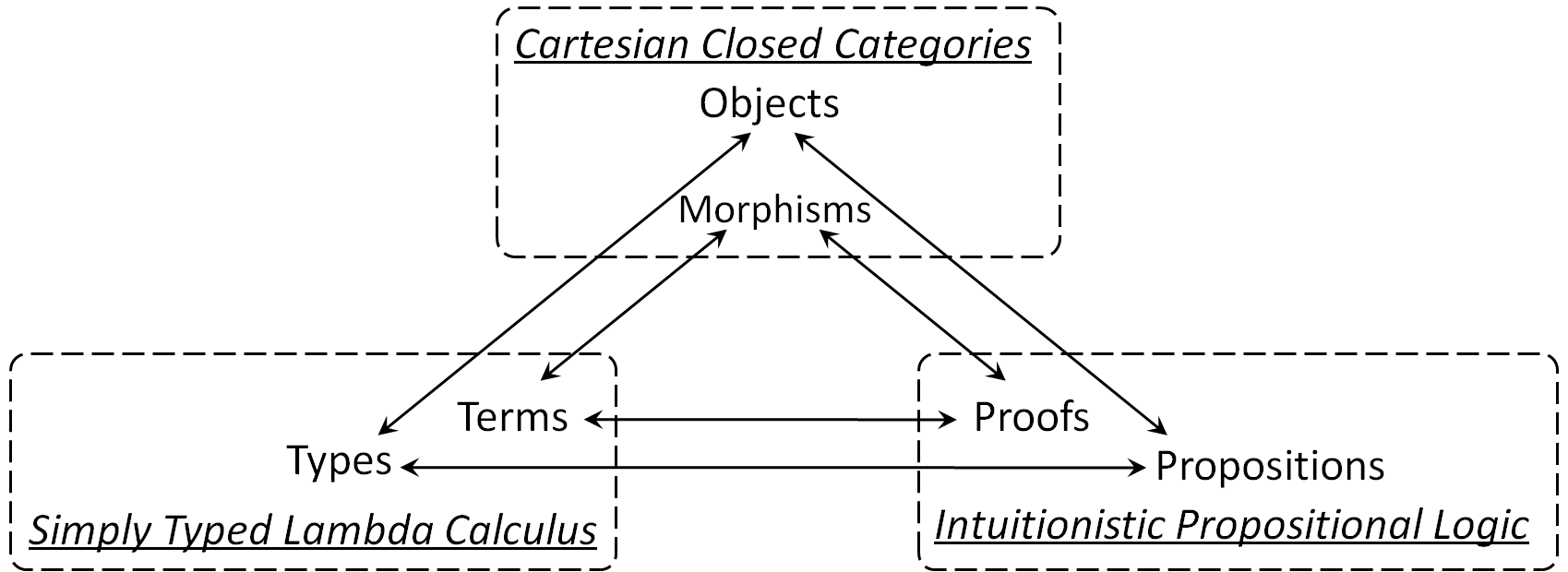
**Abstract**

**1. Introduction**

Logicians must be familiar with modus ponens , a very common rule of inference, saying that given implication and proposition , we have . Programmers may frequently use function application: if is a function of type and is an argument of type , then the application is of type . 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.



Intuitionistic logic is a formalization of Brouwer’s 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 Church’s 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 Lane’s 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**

**2.2 Lambda Calculus**

The -calculus is a family of prototype programming language. The simplest of these languages is the pure lambda calculus which studies only functions and their applicative behaviors but does not contain any constant or type.

The syntax of -terms in pure lambda calculus is simple. A -term can be a term-variable, an application, or an abstraction. *Application* is one of the primitive operations. A lambda application denotes that the function is applied to the argument . Another basic operation is *abstraction*. Let be an expression possibly containing or depending on variable . Then a lambda abstraction denotes the function .

There are three kinds of equivalences playing an important role in -calculus. The first one, -equivalence, states that the changes of bound variables in a -term do not change it meaning. -equivalence is about applying functions to their arguments. And -equivalence refers to the idea of extensionality. All of them will be discussed in more detail in the section about the proof system of the simply typed lambda calculus.

**2.2.1 Simply typed lambda calculus**

Church introduced a typed interpretation of lambda calculus, now called the *simply typed lambda calculus*, by giving each -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, and initial and terminal types.

The one with only function type constructor is called the *simply typed lambda calculus with function types*, indicated by . With products, sums and functions, we have and so on. However, is as expressive as other versions of simply typed lambda calculus. We will have a look at first and then introduce the other types into it.

To begin with, we give the definition of types in .

**2.2.1 Definition (Types)** Assume that a set of type-constants is given. Then the *types* in are defined as follows:

i) each type-constant is a type, called an *atom*;

ii) if and are types then 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 section 3.3.

The syntax of is essentially that of the pure lambda calculus itself. By assigning type to lambda term , we have an expression called a *type-assingment*, saying that term has type . Here, is called its *subject* and its *predicate*. However, not every pure lambda term can be given a type. The typing constraints are context sensitive.

**2.2.2 Definition (Type-context)** A *type-context* is any finite set of type-assignments  
  
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:

* for ;
* for .
* for

Given a type-context , a -term and a type , the expression is called a *term of type in context* . 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.

**2.2.3 Definition (Typing rules of )** Assume that a set of term variables of a given type is provided. The well-typed terms in are defined simultaneously using the following axioms and inference rules:

* *Axioms* *for variable* : for each term-variable of type ,  
    
  It simply says that a variable has any type which it is declared to have. In other words, if has type in the context, intuitively, then has type .
* *Adding variables to type context* : suppose is free in ,  
    
  In words, if has type in context , then it has type in context . This rule allows one to add an additional hypothesis to the type context.
* -*elimination* :  
    
  It says that by applying any function of type to an argument of type , we obtain a result of type . Therefore, it is also called *function application*.
* -*introduction* : suppose ,  
    
  If a term specifies a result of type for all , then the expression defines a function of type .

**2.2.2 Equational proof system of**

To derive equations of terms in that holds in all models, we need an equational proof system for . A typed equation has the form where both and are assumed to have type in context . Since type assignments are included in equations, we have an equational version of the typing rules that build well-typed equations in .

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

**2.2.4 Definition (Substitution)** We define to be the result of substituting for each free occurrence in and making any changes of bound variables needed to prevent variables free in from becoming bound in . More precisely, we define for all and all

* ;
* ;
* ;
* ;
* if is not free in ;
* if is free in and is not free in ;
* if is free in and is free in .

In the last case, can be any variable that is not free in and .

For any and any distinct , the result of simultaneously substituting all for ( in term is defined similarly to the definition above and denoted as .

**2.2.5 Definition (-equivalence)** Given a variable which is not free in , we have  
  
and the act of replacing an occurrence of in a term by is called a *change of bound variables*. and are -*equivalent*, notation , if results from by a series of changes of bound variables.

**2.2.6 Definition (-equivalence)**

i) A -*redex* is any term of the form . It can be reduced by the following rule  
  
If contains a -redex and is the result of replacing it by , we say -*contracts* to , denoted as .

ii) A -*reduction* of a term is a finite or infinite ordered sequence of -contractions, i.e. . A finite -reduction is *from* *to* if it has contractions and or it is empty and . If there exists a reduction from to , we say -*reduces to ,* denoted as . We can see that -conversions are allowed in a -reduction.

iii) and are -*equivalent*, notation , if can be changed to by a finite sequence of -reductions and -expansions (reversed -reductions).

A term may be able to -reduce to different terms at the same time. For example, the term can -reduce (in one step) to by substituting to in or by substituting to in . It is necessary for a calculus that the result of computation is independent from the order of reduction. This property holds for all -terms. It is described in Church-Rosser Theorem for .

**2.2.7 Definition (-equivalence)** An -*redex* is any term of form with is not free in . It can be reduced as follows:  
  
The definitions of -*contracts*, -*reduces* (), -*equivalence* (), etc. are similar to those of the corresponding -concepts in definition 2.2.6. However, all -reductions are finite while -reductions may be infinite.

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

Now, we can have a look at the typing rules for type equations in .

**2.2.8 Definition (Typing rules for typed equations in )** The typed equations in are generated by using the following typing rules:

* *Reflexivity* :
* *Symmetry* :
* Transitivity :  
    
  As an equivalence relation, the equality in should have the above properties, reflexivity, symmetry and transitivity.
* *Adding variables to type context* : suppose is free in and ,
* -*elimination* :  
    
  It says that equals applied to equals yield equals, i.e. application preserves equality.
* -*introduction* :  
    
  This rule says that if and are equal for all values of , then the two functions and are equal, i.e. lambda abstraction preserves equality as well.  
  The three rules above can be seen as the equational versions of the typing rules corresponding to the ones for well-typed terms.
* -equivalence : suppose is not free in ,  
    
  It allows one to rename bound variable.
* -equivalence :  
    
  It shows how to evaluate a function application using substitution.
* -equivalence : suppose is not free in ,  
    
  It says that and define the same function, since by we have for any argument .

**2.2.3 Other types**

**2.2.9 Definition (Initial and terminal types)** The *initial type*, denoted as , is a type such that for each type , there is a unique term constant  
  
The *terminal type*, denoted as , is a type such that there is only one term associated with it which is

**2.2.9 Definition (Products)** If and are types, then is a type, called a *product*. Given and , the pair has type . And the projection terms and return the first and second components of a pair. The typing rules for product are given as follows:

* -*introduction* :
* -*elimination* 1 :
* -*elimination* 2 :

**2.2.10 Definition (Sums)** If and are types, then is a type, called a *sum*. The term constants associated with sums are injections and and case . The typing rules for sums are given as follows:

* -*introduction* 1 :
* -*introduction* 2 :
* -*elimination* :

With initial type, terminal type, function types, products and sums, the type expressions in the full simply typed lambda calculus are given by the following grammar:

.

And the terms in are given by

*.*

**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 “objects” and “morphisms” independently of what they are representing. Nowadays, categories appear in most branches of mathematics and many parts of computer science. Topoi, a kind of category, can even serve as a foundation for mathematics. Cartesian closed categories can work as a framework for describing the denotational semantics of typed lambda calculus, and more generally, programming languages.

**2.3.1 Categories**

**2.3.1 Definition (Categories)** A *category* consists of:

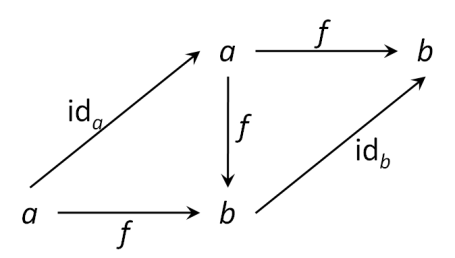
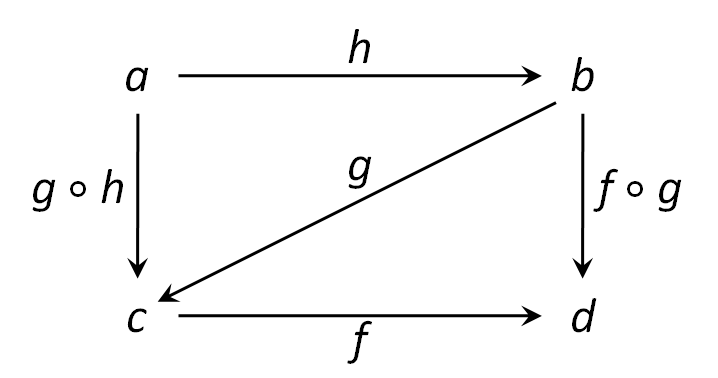
* a collection of *objects*;
* a collection of *morphisms* (also called *arrows or maps*) between objects, with two maps which give the domain and codomain of a morphism (we write to denote a morphism with and );
* a binary map “”, called composition, mapping each pair of morphisms with to a morphism such that and ;

such that the following axioms hold:

* identity: for every object , there exists a morphism , called the identity morphism for , such that for any morphism ;
* associativity: for every , and .

For any objects and of a category , the collection of all morphisms is called a *hom-set* and denoted as . A category is determined by its hom-sets.

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

** **

A common example of a category is the category **.** It is the category whose objects are sets and whose morphisms are functions. The identity of object in is the identity function . The composition of morphisms is the composition of functions. As a category, it satisfies the two category axioms:

i) , for every ;

It follows by using the definition of composite function and identity function:  
 and

ii) , for every , and .

This follows from the fact that composition of functions is associative:

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 whose objects are posets and whose morphisms are monotone functions. It will be discussed later as an example of a CCC.

**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 of these is the notion of initial object and its dual, terminal object.

**2.3.2 Definition (Initial and Terminal Objects)** Let be a category. An object of is *initial* if, for any object in , there is a unique morphism from to . An object in is *terminal* if, for any object in , there is a unique morphism from to .

In this dissertation, terminal objects are denoted as and, for object , the unique morphism is denoted as .

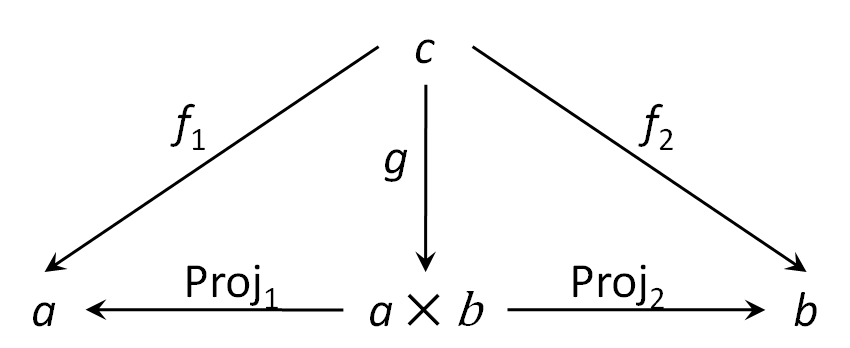
In , the initial object is the empty set , and the unique morphism with for its source is the empty function whose graph is empty. Any singleton set is terminal in since for any set , there is exactly one function from to this singleton set.

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).

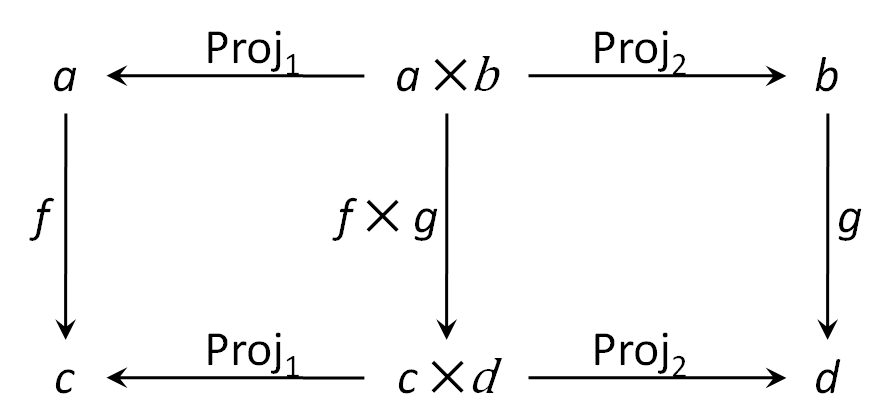
**2.3.3 Definition (Products)** Let and be objects of a category . The *product* of and is an object together with two morphisms and , and for every object in , an operation such that for all morphisms , , and , the following equations hold:

* ;
* .

Since equations in category theory can be represented by commutative diagrams, we can give another definition of categorical products based on diagrams: Let and be objects of category . The *product* of and is an object together with two morphisms and , and for every object in , an operation such that for every and , the morphism is the unique satisfying



The cartesian product construction for morphisms can also be given a categorical definition. Given morphisms and the product is defined by whose correspondent commutative diagram is the following:



**Proposition** Let be a category with products. Given , , and , we have .

**Proof**

Similarly, we have .

By the equation in definition 2.3.3,

According to the definition of products of morphisms,

.

Therefore, the equation holds.

□

The products in are the cartesian product of sets. Let and be two sets. The cartesian product is the set of pair with and , together with the coordinate functions and such that and . Given two functions and , the function is defined by for all . Then, given , and , the equations in definition 2.3.3 are satisfied:

i)

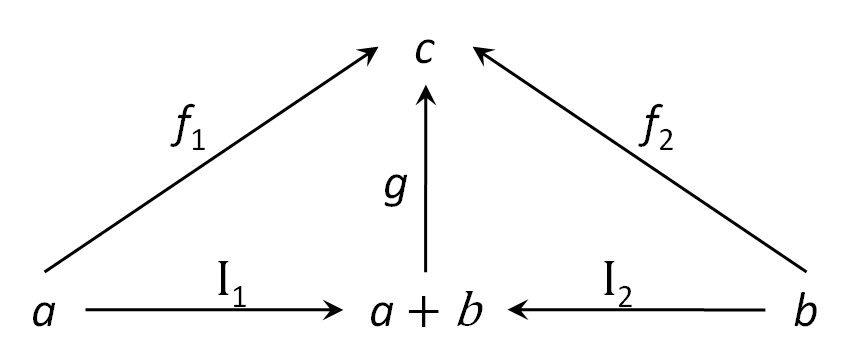
ii)

for all .

**2.3.4 Definition (Coproducts)** Let and be objects of a category . The *coproduct* of and is an object together with morphisms and , and for every object in an operation such that for every , and , , the following equations hold:

* ;
* .

The corresponding commutative diagram to the equations above is shown as follows:



where is the unique .

The coproducts in are the disjoint unions of sets. Let and be two sets. The disjoint union of them is defined by with two injection functions that takes all in to in and that takes all in to in . Given two functions and , the function is defined by for all . Given , and , the equations in the definition above are satisfied:

i)

ii)

for all .

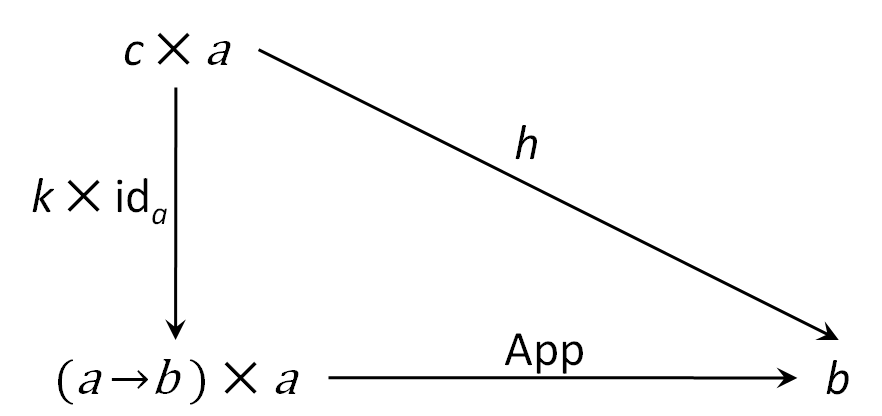
One can form a set of functions which have the same domain and codomain. Similarly, the hom-set of morphisms may form an object. This idea brings our last basic construction, exponentials.

**2.3.5 Definition (Exponentials)** Let be a category with products for all objects, and and be objects of . The *exponential*, also called *function object*, of and is an object together with a morphism , and for every object in , an operation such that for every and , the following equations hold:

* ;
* .

Using product of morphisms , the two equations above can be rewritten as and .

The commutative diagram representing the equations in the definition is shown as follows:



where the morphism is the unique .

In , the exponent set of and is the set of functions from to . The function is given by for all and . Given a function , the function is defined by for all and . Given and , the two equations hold:

i)

ii)

for all and .

**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.

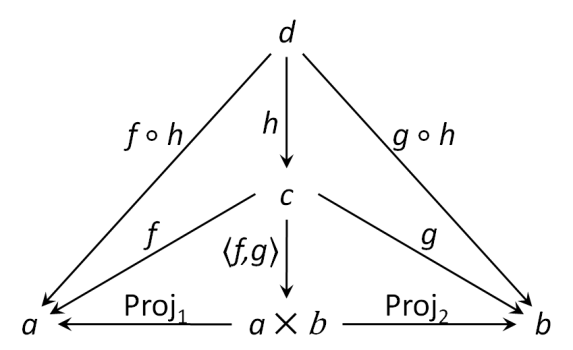
**2.3.6 Definition (Cartesian Closed Categories)** A category is *cartesian closed* iff

* It contains a terminal object ;
* For all objects and in , there is a product;
* For all objects and in , there is an exponential.

**Proposition** The following two useful identities hold in all cartesian closed categories:

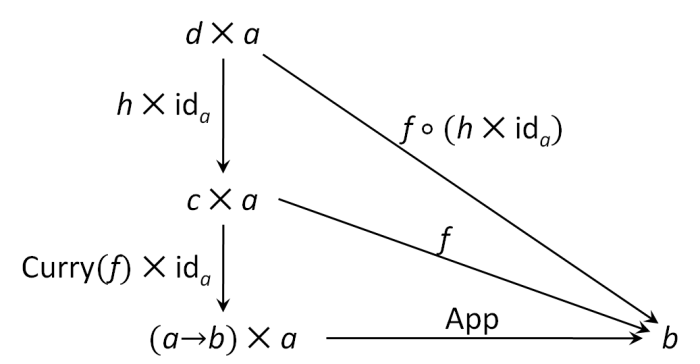
i)

where , and ;

**

ii)

where and .



**Proof**

These two equations have been proved by the diagrams following them. But they can also be proved by the equations given in definition 2.3.3 and definition 2.3.5:

i)

Since we have , then holds.

ii) (by )

(by )

(by )

□

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

(1)

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 and is the cartesian product of and ;
* The exponential of sets and is the set of functions from to .

(2)

is the category whose objects are posets and whose morphisms are monotone maps. The identity of object in is the identity map such that for all . The composition of morphisms is the composition of maps which still preserves the monotonicity property.

is cartesian closed:

* Any singleton poset can be the terminal object . The map from any poset in to this singleton poset is unique.
* The product of posets and is the cartesian product of set and with the ordering which is defined by iff and . The projections are the coordinate functions and . Given two monotone maps and , the map is defined by for all . Suppose and , then and . According to the definition of , we have , therefore, the map is monotone. In the same way as , satisfies the equations in definition 2.3.3.
* The object is a poset consisting of the set of monotone maps from to and a ordering defined by, given , iff for all . Given , the map is defined by for all and . Suppose , and , we have because ; therefore, the map is monotone. The map is defined by for all and . Suppose and with and , then by the definition of and ; therefore, is also monotone. satisfies the equations in definition 2.3.5. The proof is carried out in the same way of .

(3)

is same as the category except that each poset has a least element . We construct its identity, composition, terminal object, product, exponential in the same way of **.** Since the categorical constructions of morphism still preserve the monotonicity of morphisms; therefore, is cartesian closed.

(4) – non-example

is the category whose objects are posets with least element and whose morphisms are monotone maps with the property that in one poset is mapped to in another one, i.e. for any monotone map , .

However, is not cartesian closed. It has terminal object. But it does not have products and exponentials. Suppose that the products in are constructed in the same way of **.** (the monotonicity cannot be preserved)

**3. Correspondences**

**3.1 Every type-derivation in leads to a proof in intuitionistic implicational logic**

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

**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 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 without product type constructor is as expressive as . Therefore, both the type expressions and well-typed lambda terms in can be interpreted in any CCC, and this interpretation should be sound and complete.

**3.3.1 Definition (The Interpretation of Terms)** Given a typed lambda calculus and a cartesian closed category , we choose an object of for each type constant and a morphism for each term constant, and then all type expressions and typing contexts can be interpreted as objects and well-typed terms as morphisms. For notational simplicity, is omitted to denote the interpretation of type expressions, typing contexts and terms of in :

(1) The interpretation of type expression is defined as follows:

* , given as an object constant in ;
* .

(2) The interpretation of typing context is defined by induction on the length of the context:

* ;
* .

(3) The interpretation of a well-typed term is a morphism from to which is defined by induction on the proof of the typing judgement :

* ;
* , given as a morphism constant in ;
* ;
* ;
* where and contains all the free variables of .

In this definition, is an *m*,*n*-function. If is an ordered type-context of length , the ordered type-context of length is defined by . If is a well-typed term and contains all the free variables of , then the interpretation of can be related to the one of by using a combination of pairing and projection functions.

Before giving the definition of the combination, we need a few notational conventions. If is a morphism from object to , for , then we write for the morphism and for a composition of projection morphisms and so that .

Now, given an *m*,*n*-function , we can define by

.

The terminal object is included in the type of since the interpretation of any type-context contains .

Some lemmas are needed in the proof of soundness and completeness of the interpretation defined in 3.3.1.

The first one is the substitution lemma, which will be used in the proof of soundness later.

**3.3.2 Lemma (CCC Substitution)** If and are well-typed terms, then .

**Proof**

The proof is carried out by induction on typing derivation. The base case is the one whose term is a term variable. The inductive steps have two cases: application and abstraction.

**Base case**

(by)

**Inductive steps**

**(1) Application**

Induction hypothesis:

and

.

**(2) Abstraction**

Induction hypothesis:

(by)

□

**3.3.3 Theorem (Soundness)** Given any well-typed terms and with , then the interpretations of them are same, i.e. , in every CCC.

**Proof**

**(1) -equivalence**

During the interpretation, we can see that the names of term variables never appear in the interpretation; therefore, -equivalence should hold in the interpretation. Here, an equational proof of a stronger form of -equivalence is given.

If and are well-typed, with , then

The proof can be carried out by induction on the term . The base case is the one when is a variable while the inductive steps contains application and abstraction.

Base case –

Inductive steps:

(a) Application –

with and .

Induction hypothesis:

and

(b) Abstraction –

with

Induction hypothesis:

Therefore, .

**(2) -equivalence**

We can represent -equivalence in the following form:

(by)

(by)

(by Substitution Lemma)

**(3) -equivalence**

We can represent -equivalence in the following form:

(by and )

(by)

□

**3.3.4 Theorem (Completeness)** Given any well-typed terms and , there exists a CCC such that if, then .

**Proof**

The category is generated by in the following way:

The objects of are sequences of type expressions. To be specific, the empty sequence is the terminal object and a sequence is the product of these types. For notational convenience, we write for a sequence of variables, similarly for a sequence of type expressions, and for typing context .

The morphisms from to are given by -tuples of terms over free variables. To put it more specifically, a morphism is an equivalence class of tuples of terms,

.

Composition in is defined by substitution. Given morphisms and , the composition of them is

**(1) is cartesian closed**

The cartesian closed structure of is obtained as follows:

**i)** Terminal object with unique morphism

As mentioned above, the empty sequence of types is the terminal object . According to the definition of morphisms above, a morphism from an object to is given by -tuples of terms. Then, the morphism from an object to , the empty sequence, should be given by the empty tuple. For every object , the morphism is defined by the empty tuple, i.e. . Clearly, it is unique for every object .

**ii)** Product objects with projection morphisms and function

Given two objects and , the product of them is obtained by their concatenation, . Then, the projection morphisms is given by

And, given and , the product function is defined by the concatenation of tuples:

The equations in definition 2.3.3 are satisfied. For any

,

, and

,

we have

Similarly, we have

**iii)** Exponentials with function and morphism

The exponential of objects and is the sequence of types of the following pattern .

Given any , the morphism is defined by

For any and , the morphism is defined by

The equations in definition 2.3.5 should hold. For any  
 and   
,

we have

Therefore, is cartesian closed.

**(2)**

Given any well-typed term in , its interpretation in obtained in (1) is the equivalence class of itself. Precisely,

Then, if and have the same interpretation, they should be in the same equivalence class, i.e. **.**

Therefore, given any well-typed terms and , there exists a CCC such that if, then .

□

**References**

1