1. Introduction

The topic of my project is the Curry-Howard Isomorphism. This is a dissertation-only project. I don't have any programs to demonstrate. But I will do a 20-minute presentation.

To begin with, let us have a look at two rules. We are familiar with this very common rule of inference, the modus ponens, saying that given implication "A implies B" and proposition A, one can conclude B. Programmers frequently use function application. It says that if f is a function of type A to B and x is an argument of type A, then the result of applying f to x has type B.

Interestingly, propositions and types in these two rules have the same behaviors. It seems that there is a connection between logic and programming. This connection is known as the Curry-Howard Isomorphism.

For years, it has been extended to other mathematical systems, such as the cartesian closed categories. In this project, I mainly probed into this three-way correspondence between intuitionistic propositional logic, simply typed lambda calculus and cartesian closed categories.

2. Background

Before looking at the correspondence, we need some background.

2.1 Intuitionistic logic

The first one is intuitionistic logic. What makes it different from the classical one is that syntactically RAA is not a rule in the natural deduction system of intuitionistic logic.

Since both the law of excluded middle and double negation elimination are proved by using RAA in classical logic, neither of them is provable in intuitionistic logic.

Semantically, the judgements about statements to be true are based on the existence of a proof or a construction of that statement instead of truth value.

2.2 Simply typed lambda calculus

Lambda calculi are a family of prototype programming language. In this presentation, I'm mainly introducing the simply typed lambda calculus with only function type constructor. It is written as lambda arrow.

In lambda arrow, a type is either a ground type or a function type. A term can be a term variable, a lambda abstraction or an application. A lambda abstraction denotes a function whose input is x. A application MM applies the first M to the second M.

There are three kinds of important equivalence. The alpha equivalence says that the names of bound variables do not matter. We can replace the x here by y or other names. The beta reduction is done by substituting x by N in the function body M. The eta equivalence says that two functions are equivalent if they give the same results for the same inputs.

2.3 Cartesian closed categories

Category theory seeks to express mathematical concepts in terms of "objects" and "morphisms" independently of what they are representing.

A category C is cartesian closed if it has this structure. Firstly, it should have a terminal object, which means that for every object A there is a unique morphism from A to it. Secondly, it should have products for every pair of objects. Categorical products should satisfy these equations. Finally, it should have exponentials for every pair of objects. Similarly, categorical exponentials should satisfy these two equations. These two equations are very important in this project. That's because the first one corresponds to beta equivalence and the second one to eta equivalence, though they look very different.

3. Correspondence

Instead of listing the correspondence directly, I would like show how to encode proofs in lambda terms and how to decode lambda terms to get proofs.

3.1 Encoding proofs in lambda terms

A proof is a tree constructed by using the inference rules in the natural deduction system. In my dissertation, it has been proved that every proof leads to a type derivation.

A type derivation is a tree as well. But it is built by using the typing rules. Every derivation brings a well-typed lambda term in the bottom of the derivation tree.

As a result, every proof can be encoded in a well-typed lambda term. To put it more specific, the proofs are encoded in the terms, while the assumptions are encoded in the type context. Then the proofs of theorems (propositions without assumptions) are encoded in closed terms.

Let us have a look at an example.

This is the proof of theorem "phi to psi to phi". From top to down, we treat each proposition as a type and give it a term. In the context, we give phi a term variable x, and then we are able to conclude that x has type phi. Now since the context has another proposition psi, we should give it another term variable, say y. We can delete y from the context and construct a lambda abstraction of type psi to phi. Then we do the same thing to x in the context. We can see that it is the combinator K of type phi to psi to phi.

Therefore, we say that the proof of theorem "phi to psi to phi" is encoded in the combinator K.

3.2 Decoding well-typed lambda terms

On the inverse direction, we decode well-typed terms and obtain proof of corresponding propositions.

Firstly, every well-typed lambda term uniquely determines a type derivation. This is not trivial but its proof is given in my dissertation.

If we erase all the terms in a type derivation and consider all the types as propositions, we obtain a proof.

This example gives us the combinator K. We can rebuild a type derivation for it. By erasing all the terms in the derivation, we obtain a proof of the theorem "phi to psi to phi".

3.3 Some correspondence

Let's have a look at a brief summary.

Propositions correspond to types. Since propositional connectives correspond to type constructors. If we take the set of propositional variables equal to the set of type variables, then the set of propositional formulas is identical to the set of simple types.

Proofs correspond to terms. That's because the natural deduction system works in the same way as the typing rules. One is used for building proof and the other one for building well-typed terms.

Some typical questions in these two fields correspond to each other as well, such as proof checking and type checking, provability and inhabitation.

3.4 Interpreting well-typed lambda terms in CCCs

The correspondence between CCCs and the other two systems is a little bit different with the one we just see. The CCCs are used for describing the denotational semantics of the other two systems. As a result, we can interpret the lambda calculus in CCCs. And the interpretation is sound and complete.

3.4.1 Which lambda calculus

The connection between lambda unit, times, arrow and CCC seems to be obvious. But lambda arrow, the simply typed lambda calculus with only function types, is as expressive as this one. All the literature I have read gives the interpretation of this lambda calculus in CCCs but not lambda arrow. In the project, I defined the interpretation of lambda arrow in CCCs and show that it is sound and complete. And this part is my own contribution as well as the main part of the project.

3.4.2 Interpretations

Generally, both types and type contexts are interpreted as objects. Ground types are interpreted as object constants and function types as exponentials. The interpretation of a type context is the product of the interpretations of types in it. Specially, the empty context is interpreted as the terminal object.

Given a term M having type sigma in context gamma, it is interpreted as a morphism from the interpretation of context gamma to the interpretation of its type sigma. Specifically, the interpretation of a term variable is a projection morphism. An application is interpreted as the composition of the evaluation morphism and the pair of the interpretations of the two terms. The interpretation of an abstraction is the currying of the interpretation of its scope.

The last rule is used to get rid of non-free variables in the context. Here, x is not free in M. The morphism chi is a combination of projection morphisms. Its full definition can be found in my dissertation.

3.4.4 Soundness and completeness

The proof of soundness and completeness is so long that the time is not enough for me to explain it. Here I just try to give an outline.

3.4.4.1 Soundness

If two terms are equivalent ant this implies that their interpretations are equal, then we

say the interpretation is sound. Here, we just consider the alpha, beta and eta equivalence and then the proof of the soundness theorem is split into three parts.

Since no term variable name appears in the calculation, alpha-equivalence should hold in the interpretation. My dissertation also provides an equational proof of the stronger form of alpha equivalence.

In the proof of beta equivalence, two important rules are needed. The first one is the equation in the definition of exponentials, the one I just mentioned in the previous slide. Another one is the substitution lemma since beta reduction is done by substitution.

Similarly, another equation mentioned in the same slide is used in the proof of eta equivalence.

3.4.4.2 Completeness

Conversely, if two terms having the same interpretation implies that they are equivalent, we say the interpretation is complete.

In order to prove it, we need to construct a category C from lambda arrow. In this category, the objects are sequences of types. Empty sequence is the terminal objects. Non-empty one is the product of its elements. We write it as vector sigma m. The morphisms are given by n tuples of equivalence classes of terms over m free variables. These terms have the same type context containing m variables. So the domain is vector sigma m. There are n equivalence classes in the tuple. So the codomain is vector tau n.

Then we need to prove that this category is cartesian closed. This may be the most difficult proof in my dissertation. We need to go through the three conditions in the definition of CCCs to show that this category has the cartesian closed structure.

The rest proof is very straightforward. The interpretation of a term is a singleton tuple of its equivalence class. If two terms have the same interpretations, they should belong to the same equivalence class. In other words, they are equivalent.

4. Reflections

4.1 Different subjects

Intuitionistic logic, lambda calculus and categories come from very diverse subjects. But an amazing connection between them can be found.

4.2 Extensions

The Curry-Howard isomorphism can be extended to stronger logic, more expressive languages and richer categories. For instance, first order logic corresponds to dependent types. The category of domains with continuous functions can be used for describing the denotational semantics of programming computable functions.

4.3 Helps

Once the correspondence is established, the Curry-Howard isomorphism helps to prove theorems in each field. Here is an example. How can we prove that the inhabitation problem in dependent types is undecidable? According to the Curry-Howard isomorphism,

inhabitation corresponds to provability and dependent types correspond to first order logic. If we know that provability in first order logic is undecidable, then the inhabitation problem in dependent types should be undecidable as well.