Logic in Computer Science

1 Introduction

The main subject of the study of this semester is logic in computer science. Logic is a subject that has been closely connected to computer science and plays an important part in the development of computer science. Logical tools are great for building consistent models and establishing properties, which means a lot to the field of computer science. As abstraction and reasoning about it is also a large part of studying in computer science.

There has been interesting examples of how two field previously thought separately could be linked together in some way, like Descartes's coordinates, which links geometry to algebra. Curry-Howard correspondence has been a recurring concept throughout my study in this semester. It relates the field of logic to the field of computer science. It is also this idea that lies in the foundation of Martin-Löf's intuitionistic type theory[4].

Computer science has also made important contributions to logic. Proof assistant has been one of them, which is a tool that helps constructing proofs for logical propositions. Automated theorem prover is another. Logic programming languages also contribute a lot. Coq[1] is a well-known example of a proof assistant with applications like CompCert compiler certification project[3].

In this writeup, I will give my understanding of the basic idea and thoughts of some selected concepts and topics I have studied, and try to talk about why it is important. The selected concepts include: Coq, Curry-Howard correspondence, language formalization, judgmental reconstruction of modal logic, linear logic.

2 Concepts/Topics Studied

2.1 Curry-Howard Correspondence

The basic idea of Curry-Howard correspondence is that logic world and computation world are similar. More specifically, Curry-Howard correspondence is about constructive logic, a specific branch of logic, corresponds to functional languages. Constructive logic focuses on the verification. It is constructive in a sense that there is always direct evidence when something is true or false. Propositions in constructive logic can be viewed as types in computation and proofs can be viewed as programs. Other correspondences are listed below:

```
propositions  ~  types
proof objects ~  programs
proof reduction ~ term reduction
proof normal form ~ term normal form
```

The idea became clear when the observation, that intuitionistic natural deduction and typed lambda calculus are the same system that has different interpretations, was made by Curry and Howard. And the idea can be applied to a variety of logics and computation models. This shows that there this type of formal systems that can be interpreted in both ways. The correspondence has led to discovery of new concepts like dependent types, which correspond to universal and existential quantifiers. It also provides interesting evidence that shows these systems were discovered rather than devised.

2.2 Coq

Coq[1] is a proof assistant for helping to write formal proofs. Using proof assistant to write formal proofs even though is different from writing proofs on paper, has the advantage that it can eliminates human error. Proof assistant translates proofs into programs that machines is able to check.

Curry-Howard correspondence is an important notion in Coq and probably a motivation behind proof assistants in general. Proof is almost the same thing as program and proposition is almost the same thing as data type. And both can be written in the fashion of the other. As a programming language, Coq have features similar to most functional languages, like higher-order functions, algebraic data types and pattern matching.

Coq has a lot of usages in computer science such as platform for modeling programming languages and softwares and proving properties about them.

2.2.1 Induction Principles

Given that a lot of the interesting propositions are inductive, induction is probably the most commonly used technique. In Coq, for every inductive definition, there is an induction principle. And the principle will be applied when doing induction on the definition.

2.2.2 Logic in Coq

There are two different ways of encoding logical facts in Coq. One of them is using proposition, the other is using boolean. For example, evenness can be written as both function and proposition. And to prove some number is even, we can either run function with the number and check if the output is true, or try to use Coq's tactics to prove the proposition holds. I suppose this is a consequence of Curry-Howard correspondence. This is also a flexibility the correspondence brings. If one way is difficult, there is always the other one.

2.3 Language Formalization

2.3.1 Computational and Relational Definition

In Software Foundation[6], an example of using Coq to formalize a language is Imp, a simple imperative programming language. The book presents two ways of giving definition of a language in Coq, computational and relational definition. The steps to do computational definition and use the definition can be summarized as follow:

- First, give definition of what the language look like in form of a type according to syntax.
- Then, show how to evaluate a program using previous definition in form of a function.
- Finally, give evaluation function the program, and the result is the output.

Relational definition comes with thinking about evaluation as relation. How does a boolean expression matches the value, for example. Instead of showing how to evaluate, it gives what the relation is when the expression and the result matches. Relational definition is different but both definition has the same ability to express a language. Using relational definition can be summarized as follow:

- First, give definition of what the language look like in form of a type according to syntax.
- Then, show how program would match the result in form of a proposition.

If both were done right, it can be proved that the two definition is equivalent. Although the ability to express is different in Coq. Because Coq doesn't allow functions that doesn't guarantee termination, the computational definition fails at modeling languages with loop that doesn't guarantee termination. But relational definition isn't restricted by this rule.

In addition, using computational definition the evaluation process could be done automatically, while relational definition couldn't. Because Coq, as a proof assistant, helps people to prove proposition by themselves, not doing proof automatically.

2.3.2 Big-step and Small-step

Operational semantics are ways of describing how to interpret a program in the language. Big-step semantics specifies how a given expression can be evaluated to its final value. However, it's not a natural way to think about concurrent programming languages, where the intermediate states matters. Another problem is if we want to allow different types of variables in the same expression, then evaluation might get stuck for 2 reasons: because there is an infinite loop and because there is an undefined expression.

Small-step semantics specifies how to reduce an expression one step at a time. This give us the ability to reason program behaviors we previously can't. Using small-step allows us to distinguish nontermination from stuck states. Small-step also helps when proving important properties like progress and type preservation.

2.3.3 Type System

In programming languages, programs deal with multiple kinds of data like integer or boolean. And not all operation make sense, for example, addition of integer and boolean. Type system helps check this kind of term. Using this tool, we can prove 3 critical properties.

- Progress: Well-typed normal forms are not stuck, it's either a value or can take a step.
- Type Preservation: When a well-typed term takes a step, the result is also a well-typed term
- Type Soundness (previous 2 combine): well-typed term never stuck, it just reduced to normal form and becomes a value

When proving properties we need to give definition of how a term is matched to a type. This can be done by using inductive defined propositions or a function that takes a term and returns a type. This is similar to computation and relational definition when formalizing a language.

However, type system also has application other than proving properties of a language and making sure there is no ill-typed term, for example, program synthesis.

2.3.4 Case Study: Linear Meld paper

Linear Meld is a logic programming language for graph-based problem designed by Flavio Cruz et al[2]. In this section, my focus is on their methodology of formalizing a language.

In the Linear Meld paper, the proof theory section talked about how the new aggregate and comprehension features would fit in to the logical definition of the language. It first introduces a new logical connective to relate the two features in logic. Next, the paper gives the operational semantics of how the programs with new connectives would be evaluated in steps. Although, because there is an nondeterministic element in the logical definition of the two features, the paper proposes a low level semantics to deal with this.

The method can be summarized as follows:

- First, introduces the new logical connectives to the system if necessary.
- Then, gives the operational semantics.
- Finally, reason with the system, look at properties that could be proved.

2.4 Modal Logic paper

In this section, I will give a summary of the paper, the judgmental reconstruction of modal logic by Pfenning and Davies[5].

Modal logic contains judgments such as A is valid and A is possible. The paper starts with the separation of judgments and proposition according to Martin-Löf. This separation helps formalizing statements and avoid ambiguities. Then the paper goes on to introduce hypothetical judgment, validity and possibility and eventually gives a formal system of modal logic and a corresponding type system.

With modal operator, modal logic can model a lot of problems more clearly. Modal operators can also be interpreted in different ways which has given rises to epistemic logic, temporal logic and so on.

2.5 Linear Logic

To quote Philip Wadler, "traditional logic is about truth and linear logic is about food"[7]. Linear logic has a restricted use of contraction and weakening. Focusing on the idea that views facts as resources, makes linear logic very suitable for reasoning about problem with dynamically changing states. Some examples like, session types, which corresponds linear logic to pi calculus can be used to describe contract and communication.

3 Conclusions

Over the course of this semester, I have gotten familiar with the idea that logic and computer science, programming languages in particular, shares a deep connection. The correspondence between the two provides an insight and interesting direction for looking for research topics. Computer is a powerful tool. Logic programming and proof assistants provide ways to utilize this tool. And logic gives programming languages a different perspective and tools to argue about themselves. I have also learned to work with proof assistant Coq and how to formalize a programming language in it. Language formalization are tools which allows people to reason about languages properties and doing so in proof assistant can provide proofs that can be checked by computers. The modal logic paper and Philip Wadler's paper[7] demonstrate how a constructive logic is built using natural deduction.

Reference

[1] Jos Carlos Bacelar Almeida, Jorge Sousa Pinto and Simo Melo de Sousa. Coq proof assistant.

[2] Flavio Cruz, Ricardo Rocha, Seth Copen Goldstein and Frank Pfenning. A linear logic programming language for concurrent programming over graph structures. 2014/5/14.

[3] Xavier Leroy. The compcert verified compiler. *Documentation and user's manual.INRIA Paris-Rocquencourt*, , 2012.

[4] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Bibliopolis,, Napoli, c. 1984.

[5] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(04):511–540, 2001.

[6] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjoberg and Brent Yorgey. Software foundations. *Webpage: http://www.cis.upenn.edu/bcpierce/sf/current/index.html*, 2010.

[7] Philip Wadler. A taste of linear logic. Pages 185–210. Springer, Berlin, Heidelberg, 1993/8/30.