Representing numbers and in Agda

Li Nuo

April 27, 2010

Contents

Abstract

Recent development of dependently typed languages like Coq, Agda and Epigram provide programmers as well as mathematicians to prove theorems by writing programs, or more appropirately, constructing proofs. Agda, as one of the latest functional programming languages, is a flexible and convenient proof assisstant equipped with interactive environment for writing and checking proofs. The current version of standard library which is mostly built by Danielsson has included Boolean, algebraic structures, sets, relations etc. However, to prove most of theorems for numbers, it requires more definitions of the numbers beyond natural numbers and more axioms and theorems.

To solve the problem, I start this project, in which I will define the numbers including integers, rational numbers, real numbers and complex numbers and prove the basic properties of them as the tools for theorem proving. The main motivation is my interest in mathematics. The work is also motivated by the numerous good features which gives Agda potential to be a good theorem prover. I will talk about these later along with introducing my code. Although representing numbers in a programming languages must based on the ideas in mathematics, it still has quite a few disctinctions. An interesting discovery is that I can understand the natural of numbers more deeply when I use Agda to define numbers. After that, I will discuss less abstract isssues. How to define numbers in Agda? How to prove properties? How about verification? How can we use them? The builder of Coq library has done similar work, but I represent numbers in quite different ways.

1 Introduction

Mathematics is the foundation of computer science. We could find the highly correlation between computer science and mathematics from the word "computer". In fact, as the science and technology of computer are developing faster and faster, we can see that it could contribute to mathematics more than computing.

Old chinese mathematicians using Counting rods to do computing and proving. The invention of more and more symbols for complex definitions increase the theorem proving exponentially. New inventions of tools like computer always benefit the mathematics. Before computer invented, mathematicians had to prove theorems on papers. The proofs always spread around a pile of papers. It is likely to make mistakes for proving. In order to eliminate mistakes, verification of proof is needed. Even one mathematician had spent quite a long time on the verification, no one can ensure the correctness for complicated proofs. Some proofs had been believed to be correct were found incorrect after years later. In 1879 Alfred Bray Kempe announced that he had proved the Four Colour problem. Until 11 years later, Percy John Heawood published a paper proving Kempe was wrong.

The old ways to do proving is very inefficient. Computer changed all the aspects of humans' lives. Computer scientists has created a lot of implementation of the theorem proving by computer programs. With the dependently typed language, such as Coq, Agda and Epigram, mathematicians and programmers could formally prove theorems by writing proofs as programs and the work of verification can be left to computers [2]. Computer could ensure the correctness of proofs. If we believe in the languages, we can believe in the correctness of proofs. The proof assistants has revealed their power in recent years. In 2004, Georges Gonthier and Benjamin Werner have completed their proof of the four color theorem using Coq [8]. Computer could do some automated work which seems impossible to handle by hand.

Agda is one of the proof assistants and it is the latest in a series of dependently typed programming languages [4]. Unlike tactic-based proof assistant like Coq, it provides a more flexible way of constructing proofs [2]. It has great potential in the field of theorem proving. However as Agda is at its the early stage, it requires contribution to the standard library of Agda. Some of the basic mathematical definitions has been included, such as sets, logic symbols, relations, algebraic structure. It is enough to define more concepts.

For numbers, there are only natural numbers and part of integers defined in the standard library. However to prove most of the mathematical theorems, we need more definitions for other kinds of numbers, not only the integers but also rational numbers, real numbers and complex numbers. Their basic axioms like commutativity and associativity are essential as well.

The project aims at representing the numbers in Agda. The main objective is to define those numbers in most proper ways and proving some basic properties of them that are essential for theorem proving. There are difference ways of defining different kinds of numbers, so to compare the efficiency and convenience of them is also an objective. From the aspects of numbers, to define as much as possible kinds of numbers is the primary objective.

2 Motivation

The motivations to do this project are that I am interested in how mathematics can be translated into computer science, how to use computer science to help us solve mathematical problems and how to use mathematics to solve computer science problems. The interest in functional programming is also plays a quite important role. I had learnt to use Coq which is also a proof assistant like Agda to prove theorems using software tools. I found it interesting to formalise mathematics concepts and doing more than computations such as proving or facilitating calculations. After undertaking the project I also found it is beneficial for other people as some other students from Chalmers told me they could improve their code by using the library code of numbers. Moreover to develop a part of a library is a new challenge and I also want to learn the related skills. At the same time I could also gain much insight of representing numbers and proving theorems in Agda from doing this project. It is also a good experience of doing a comparably big research project for my futher study and career lifes.

3 What is Agda?

Agda is a dependently typed functional programming language. It is designed based on Per Martin-Löf Type Theory[6].

We can find three key elements in the definition of Agda, the "functional programming language", "dependently typed" and "Per Martin-Löf Type Theory".

- *Functional programming language.* As the name indicates that, functional programming languages emphasizes the application of functions rather than changing data in the imperative style like C++ and Java. The base of functional programming is lambda calculus. The key motivation to develop functional programmming language is to eliminating the side effects which means we can ensure the result will be the same no matter how many times we input the same data. There are several generations of functional programming languages, for example Lisp, Erlang, Haskell etc. Most of the applications of them are currently in the academic fields, however as the functional programming developed, more applications will be explored.

- *Dependent type.* Dependent types are types that depends on values of other types [3]. It is one of the most important features that makes Agda a proof assistant. In Haskell and other Hindley-Milner style languages, types and values are separated clearly [5], In Agda, we can define types depending on values. To illustrate what this means, the most common example is **Vector A n**. It is a data type which represents a vector containing elements of type **A** and has a given length of **n**. Here the type **Vector A n** depends on value **n** which is a natural numbers. With the type checker of Agda, we can set more constraints in the type so that

type-unmatched problems will always be detected by complier. Therefore we could define the function more precisely as there more partitions of types. For instance, to use the dependently typed vector, it could avoid defining a function which will cause exceptions like out of bounds in Java.

- *Per Martin-Löf Type Theory.* It has different names like Intuitionistic type theory or Constructive type theory and is developed by Per Martin-Löf in 1980s. It associated functional programs with proofs of mathematical propositions written as dependent types. That means we can now represent propositions we want to prove as types in Agda by dependent types and Curry-Howard isomorphism [4]. Then we only need to construct a program of the corresponding type to prove that propostion. For example:

  ```
  n+0+0≡n : ∀ {n} → n + 0 + 0 ≡ n
  n+0+0≡n {zero} = refl
  n+0+0≡n {suc n} = suc n+0+0≡n
  ```

  The do theorem proving in Agda we can follow the steps: First we give the name of the proposition and encode it as the type. Then we can gradually refine the goal to formalise a type-correct program namely the proof. There are no tactics like in Coq. However it is more flexible to construct a proof. The process of building proofs is very similar to the process of constructing proofs in regular mathematics. The logic behind it is that if we could construct an instance of the type (proposition), we prove it. It is actually the Curry-Howard isomorphism.

As Agda is primarily used to undertake theorem proofs tasks, the designer enhanced it to be more professional proof assistant. There are several beneficial features facilitating theorem proving,

- *Emacs interface.* It has a Emacs-based interface for interactively writing and verifying proofs by type-checker. It allows programmers leaving part of the proof unfinished so that the type-checker will provide useful information of what is missing [4]. Therefore programmers could gradually refine their incomplete proofs of correct types.

- *Unicode support.* It supports Unicode identifiers and keywords like: ∀, ∃ etc. It also supports mixfix operators like: + , − etc. The benefits are obvious. Firstly we could define symbols which look the same and behave the same in mathematics. These are the expressions of commutativity for natural numbers:

  ```
  Maths: ∀ a b, a + b = b + a
  Code in Agda: ∀ a b → a + b = b + a
  ```

  Secondly we could use symbols to replace some common-used properties to simply the proofs a lot. Finally, we could use some other languages to define functions for example using Chinese characters.

- *Code navigation.* We can simply navigate to the definition of functions from our current code. It is a wonderful features for programmers as it alleviate a great deal of work to look up the library.

4 What are numbers?

5 How to define numbers in Agda?

6 What to define in Agda?

7 Related work

8 Proving using Agda

9 The detailed explanation of code

10 How to use the code

11 Problems and changes

I have rewritten nearly all of the code to improve the efficiency to use. The time spent to type check the definition with real number costs several minutes, but the newer version can pass within 10 seconds. I give up to use the ring solver which is less efficient than construct proof manually. Also I define some symbols to make the proof looks much simpler and it is much shorter using the reasoning structure.

The new function for rational number is 'Inverse'. In mathematics, to get a inverse we just need to swap the numerator and the denominator. But there are two points to notice when defining the inverse function: 1. 0 has no inverse 2. Because the types of the numerator and denominator are different, we have to keep the sign on the top, project $\mathbb{Z}$ on the top to $\mathbb{N}$ then pos$\mathbb{N}$ and inject pos$\mathbb{N}$ on the bottom to $\mathbb{Z}$. When we inject $\mathbb{N}$ to pos$\mathbb{N}$, we need to give the proof that it is not zero which can be transformed from the premise using the 'inverse-lem'.

12 Evaluation

13 Summary

14 Future work