University of Bergen
Department of Informatics

Untitled

Author: Andreas Salhus Bakseter

Supervisors: Marc Bezem, Håkon Robbestad Gylterud

UNIVERSITETET I BERGEN

Det matematisk-naturvitenskapelige fakultet

February, 2023

Abstract

Lorem ipsum dolor sit amet, his veri singulis necessitatibus ad. Nec insolens periculis ex. Te pro purto eros error, nec alia graeci placerat cu. Hinc volutpat similique no qui, ad labitur mentitum democritum sea. Sale inimicus te eum.

No eros nemore impedit his, per at salutandi eloquentiam, ea semper euismod meliore sea. Mutat scaevola cotidieque cu mel. Eum an convenire tractatos, ei duo nulla molestie, quis hendrerit et vix. In aliquam intellegam philosophia sea. At quo bonorum adipisci. Eros labitur deleniti ius in, sonet congue ius at, pro suas meis habeo no.

Acknowledgements

Lorem ipsum

Andreas Salhus Bakseter
Thursday 16th February, 2023

Contents

1 Background **1**
1.1 Formalizing Mathematical Problems 1
1.1.1 Proofs 1
1.2 Type theory 2
1.2.1 Propositions as types 2
1.3 Proof assistants 2
1.3.1 Coq 2
1.3.2 Extraction of programs from verified proofs 2
1.3.3 Agda 3
1.3.4 Isabelle 3
1.3.5 Lean 3
1.3.6 Higher-Order Logic 3

2 Our case **4**

3 Approach & Design Choices **5**
3.1 Modeling Sets in Coq 5
3.1.1 List & ListSet 5
3.1.2 MSetWeakList 7
3.1.3 Ensembles 7
3.1.4 math-comp 7
3.2 Prop vs. Bool 7

4 Implementation **8**

5 Examples & Results **9**

6 Evaluation **10**

7 Conclusion **11**

Bibliography **12**

A Generated code from Protocol buffers **13**

List of Figures

List of Tables

Listings

A.1 Source code of something 13

Chapter 1

Background

1.1 Formalizing Mathematical Problems

1.1.1 Proofs

When solving mathematical problems, one often uses proofs to justify some claim. We can group proofs into two types; *informal* and *formal* proofs.

Informal proofs

An informal proof is often written in a natural language, and the proof is adequate if most readers are convinced by the proof [3]. As a proof grows larger and more complex, it becomes harder to follow, which can ultimately lead to errors in the proof's reasoning. This might cause the whole proof to be incorrect [2].

Formal proofs

A formal proof is written in a formal language, and can be compared to a computer program written in a programming language. Writing a formal proof is more difficult than writing an informal proof.

1.2 Type theory

Type theory groups mathematical objects with similar properties together by assigning them a "type". Similarily to data types in computer programming, we can use types to represent mathematical objects. For example, we can use the data type `nat` to represent natural numbers.

1.2.1 Propositions as types

The concept of propositions as types sees the proving of a mathematical proposition as the same process as constructing a value of that type. For example, to prove a proposition P which states "all integers are the sum of four squares", we must construct a value of the type P that shows that this is true for all integers. Proofs are mathematical objects; thus a proposition can be viewed as having the type of all its proofs (if any!). We can use this correspondance to model a proof as a typed computer program. The power of this concept comes from the fact that we can use a type checker to verify that our program is typed correctly, and thus that the corresponding proof is valid.

1.3 Proof assistants

Using a *proof assistant*, we can verify a formal proof mechanically.

1.3.1 Coq

Coq is an example of a proof assistant. Coq uses type theory to formulate and verify proofs, but can also be used as a functional programming language [4].

1.3.2 Extraction of programs from verified proofs

Coq also enables us to extract and execute programs from our proofs, once they have been verified.

1.3.3 Agda

1.3.4 Isabelle

1.3.5 Lean

1.3.6 Higher-Order Logic

Chapter 2

Our case

We will use the Coq proof assistant to formalize parts of the proofs of the following paper, Bezem and Coquand [1]. This paper solves two problems that occur in dependent type systems where typings depend on universe-level constraints. Since this proof is complex enough that mistakes are possible it is a good candidate for formalization. We can also use this process to gain further insight into the algorithm that lies behind the proof. It might also be interesting to use Coq to extract programs from the final proofs.

Chapter 3

Approach & Design Choices

When translating an informal proof to a formal proof or specification, one often has to decide how to model certain mathematical objects and/or properties. For example, in Coq, there are several implementations of the mathematical notion of a *set*. When choosing which implementation to use, there are often tradeoffs to consider.

3.1 Modeling Sets in Coq

https://stackoverflow.com/questions/36588263/how-to-define-set-in-coq-without-defining-set-as-a-list-of-elements

3.1.1 List & ListSet

As is common in most programming languages, Coq gives us a simple inductive definition of a list; defined in the Coq standard library **List**. A list can have duplicates, and the order of the elements are preserved. This is different from how we normally define a set in mathematics, as a set in mathematics do not allow duplicates, and order is not preserved. A list is defined as either an empty list, or an element prepended to another list. Because of this definition, it is very easy to construct proofs using induction; we only need to check two cases.

The standard library also gives us a tool to combat the possibility of duplicates in a list, with `NoDup` and `nodup`. `NoDup` is an inductively defined proposition that gives evidence (?) on whether a list has duplicates or not. `nodup` is a function that takes in a list and returns a list without duplicates.

Having just the implementation of the set structure is rarely enough; we also want to do operations on the set, and reason about these. That is were the library **ListSet** comes in. This definition of a set is just an alias for list, but its library gives us some useful functions and lemmas. Most of these treat the input as a mathematical set, meaning that they try to preserve the properties of *no duplicates* and *order*. Examples of some of these functions are `set_add`, `set_mem`, `set_diff`, and `set_union`. We also get useful lemmas that prove common properties about these functions. One thing to note is that all these functions use `bool` when reasoning about if something is true or false. This makes them decidable, but it also requires the equality of the underlying type of the set to be decidable. A proof of the decidability of the underlying type must be supplied as an argument to the all the functions. An example of the proof of the decidability of the equality for the `string` type would be:

```
Lemma string_eq_dec : forall x y : string, {x = y} + {x <> y}.
Proof.
    (* proof goes here *)
Qed.
```

These proofs are often given for the standard types in Coq such as `nat`, `bool` and `string`. As such, they can just be passed to the functions as arguments. This convention of always passing the proof as an argument can be cumbersome and make the code hard to read, but it is a necessary evil to get the properties we want.

This still leaves the problem with order of elements in the list. This implementation gives us no concrete way to combat this, but there are ways to circumvent the problem. Since we often reason about if an element is in a list, or if the list has a certain length, we do not care about the order of the elements. If we construct our proofs with this in mind, `list` is a viable implementation. There might however be cases where strict equality of two lists are needed, and that is where this implementation falls short.

3.1.2 MSetWeakList

3.1.3 Ensembles

3.1.4 math-comp

3.2 Prop vs. Bool

Chapter 4

Implementation

Chapter 5

Examples & Results

Chapter 6

Evaluation

Chapter 7

Conclusion

Bibliography

[1] Marc Bezem and Thierry Coquand. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *Theoretical Computer Science*, 2022. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2022.01.017.
URL: `https://www.sciencedirect.com/science/article/pii/S0304397522000317`.

[2] Roxanne Khamsi. Mathematical proofs are getting harder to verify, 2006.
URL: `https://www.newscientist.com/article/dn8743-mathematical-proofs-getting-harder-to-verify`.

[3] Benjamin C. Pierce. *Software Foundations: Volume 1: Logical Foundations*. 2022.
URL: `https://softwarefoundations.cis.upenn.edu/current/lf-current/index.html`.

[4] The Coq Team. A short introduction to coq.
URL: `https://coq.inria.fr/a-short-introduction-to-coq`.

Appendix A

Generated code from Protocol buffers

Listing A.1: Source code of something

```
System.out.println("Hello Mars");
```