Formally verified programming with monads in Coq

(Formalnie zweryfikowane programowanie z monadami w Coqu)

Zeimer

Praca inżynierska

Promotor: dr Wpisuyashi TODO

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

czerwiec 2019

Abstract

We introduce hsCoq, a Coq library for formally verified general-purpose programming with Haskell-style abstractions: functors, applicative functors, monads, monad transformers and typeclass-based effects. We discuss the design choices we made and illustrate the working of the library with examples taken from [1].

...

Contents

Chapter 1

Introduction

In chapter 1 we motivate the need for formal verification of software and briefly describe the Coq proof assistant. In chapter 2 we discuss the problem of modeling computational effects in programming languages and compare existing approaches. In chapter 3 we present our library hsCoqand discuss its design. In chapter 4 we give some example programs and prove their properties. In chapter 5 we describe our approach to proof engineering - the formalized mathematic's equivalent of software engineering.

Chapter 2

A short introduction to Coq

This chapter briefly introduces the Coq proof assistant to those who are not familiar with it. First we present some motivation for formal verification of hardware, software and mathematics and then describe the underlying theory of Coq.

2.1 Formal verification of hardware and software

Since their invention in the 1940s, computers' significane rose at a very fast pace. They were getting applied to an ever expanding range of problems by more and more people, private companies and governments alike. It shouldn't be a considered a surprise then that we became very reliant on them for both small conveniences and large scale projects.

But significane is not the only thing that rose - another one is complexity. Exponentially growing processing speed required the complexity of chip designs to grow at a similar rate. More complex products and services require more complex software architectures and with new business models, like cloud computing, comes even more complexity in the form of virtualization, containerization and so on.

And with complexity comes, of course, the potential for bugs, which may cause a lot damage. A malfunction in software running the stock exchange can mean billions of dollars of losses; in software running a nuclear power plant - deadly radiation for thousands of people and energy shortage for millions more.

Due to these dangers a lot of effort has been put into assuring that hardware and software are correct and with great success, but here and there bugs still have crept in. Some of the most spectecular were, recently:

- Metldown, which "exploits side effects of out-of-order execution on modern processors to read arbitrary kernel-memory locations including personal data and passwords." [2]

- Spectre, which uses speculative execution and branch prediction to "leak the victim's confidential information via a side channel to the adversary." [3]
- Heartbleed [**?**]

2.2 Formal verification of mathematics

2.3 The Coq proof assistant

Coq [4] is a piece of software implementing a formal system whose slight variants go under a plethora of names: Calculus of (Inductive) Constructions, (Intensional) Martin-Löf Type Theory, Intuitionistic Type Theory, Constructive Type Theory, etc.

Thanks to the Curry-Howard correspondence [**?**] Coq can be seen as both a functional programming language and a proof assistant.

Chapter 3

Computational effects

Chapter 4

Design

Chapter 5

Examples

Chapter 6

A case study in proof engineering

Chapter 7

Conclusion

Chapter 8

TODO

1. Introduction: functional programming, formally verified programming and proving.
2. Approaches to computational effects: chaos, ML-style, monads, algebraic effects.
3. A description of the inner workings of the library: design choices, file structure, implementation.
4. Examples: some from Just Do It, maybe some custom ones.
5. Safety: some theorems and proofs.
6. Theoretical comparison of the ease of use with Haskell and Idris.
7. Practical comparison with MERC.
8. Cite some literature: some Coq papers, Moggi, Just Do It, Experimenting with Monadic Equational Reasoning in Coq
9. Technical matters:
   (a) Mention where's the implementation and put it to Coq's repository of user libraries.
   (b) Installation guide.
   (c) Tools: why no ssreflect?
   (d) Documentation (it's in the source code).
10. More: a case study in proof engineering - how do the tactics hs, monad and (maybe) the one for reflective functor simplifcation work?
11. Deficiencies, conclusion and further work.
12. Points to make: this is a library for general purpose programming, without some deep goal.

Bibliography

[1] Jeremy Gibbons and Ralf Hinze, *Just do It: Simple Monadic Equational Reasoning*, 2011

[2] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom and Mike Hamburg, *Meltdown: Reading Kernel Memory from User Space*, 2018

[3] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz and Yuval Yarom, *Spectre Attacks: Exploiting Speculative Execution*, 2019

[4] Coq Development Team, *The Coq Proof Assistant Reference Manual*, 2019