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 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 by formalizing examples taken from [1].

Prezentujemy *hsCoq*, Coqową bibliotekę do formalnie zweryfikowanego programowania z abstrakcjami w stylu Haskella: funktorami, funktorami aplikatywnymi, monadami, transformatorami monad oraz efektami opartymi o klasy typów. Omawiamy nasze decyzje projektowe i przedstawiamy działanie biblioteki na przykładach z [1].

Contents

Chapter 1

Introduction

This chapter gives some motivations for why formal verification of hardware, software and mathematics is useful and then briefly introduces the Coq proof assistant – a tool for such verification – to those who are not familiar with it.

The rest of the thesis is structured as follows:

- 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 *hsCoq*and discuss its design.
- In chapter 4 we give some example effectful programs and prove their properties.
- In chapter 5 we describe our approach to proof engineering (the formalized mathematic's equivalent of software engineering) in our library.
- In chapter 6 we conclude and give some possible directions for further work.

1.1 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, which "allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet" [4]

DeepSpec [5] is a Coq-based project that tries to eliminate both hardware and software bugs and security vulnerabilities by creating a web of formally verified hardware, operating systems, compilers, web servers, cryptography libraries etc.

1.2 Formal verification of mathematics

Hardware and software are not the only things in need of formal verification – mathematics is also one of them.

The four colour theorem is a problem posed in 1852. It states that any planar map can be coloured with only four colours so that no two regions sharing a boundary are assigned the same colour. It became famous for resisting many proof attempts by many famous mathematicians for more than a century until it was finally proved by Appel and Haken in 1976. Its importance stems from the proof method – it was the first major theorem proven using a computer program, whose job was to make sure a very large case analysis was exhaustive.

Thomas Tymoczko, a philosopher of mathematics, criticised this proof by labeling it with a term he invented just for this purpose – "non-surveyable". He considered a proof to be non-surveyable when its verification cannot be performed by human mathematicians competent in the relevant field. Appel and Haken's proof certainly did fail the surveyability criterion – the program was written in IBM 370 assembly, a language graph theorists very likely didn't understand.

This was the perfect theorem to let formal proofs and formally verified programs shine by dispeling Tymoczko's and other mathematician's doubts. This is what

indeed happened in 2005, when Georges Gonthier presented a proof of the theorem formalized in Coq [6] [7].

But big theorems with difficult proofs are not the only call for formalized mathematics. Another one is the mere fallibility of humans, especially their limited memory and reasoning skills and the tendency to follow authority. As unmathematical as it sounds, these are the main reasons cited by Vladimir Voevodsky, a field medalist mathematician turned a fan of formal proofs, in one of his talks [8].

The fields he refers to are homotopy theory, higher category theory and motivic cohomology – all of them containing many layers of abstraction, tons of concepts and definitions, and rather shaky foundations.

As an example, noticing an error in one of his papers took 7 years and repairing the mistake another 6 years. In another, more extreme case, after publishing a paper in 1989, an alleged counterexample was found in 1998 by anoter expert in the field, but it was too difficult for them to agree on whether it really was a counterexample and Voevodsky only realized he was wrong in 2013. All of this put him in search of formalized foundations of mathematics, and he chose Coq to purse them.

1.3 The Coq proof assistant

Coq [9] is a proof assistant that was started in the late 1980's in France and still under active development. It consists of three languages:

- Gallina, the term language, implements 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. We can use it to express specifications, programs, theorems and proofs.
- Vernacular, the command language, is a language of commands, which allow things like looking up useful theorems in the environment or creating modules.
- Ltac, the tactic language, is a language which facilitates writing proofs – these can in principle be written using the term language, but it's unwieldy even for simple proofs.

The basic objects of our interest in Coq (and in any kind of the aforementioned formal systems) are types. Their role is to classify terms – for example, `21 + 21` is a term of type `nat`, written `21 + 21 : nat`. Types are syntactical entities, which means that the judgement `x : A` can always be checked algorithmically.

Thanks to the the Curry-Howard correspondence [10], types can seen both as specifications of programs and as statements of theorems. All programming and proving may be then conceptualized as manipulating a few kinds of rules:

- Formation rules tell us what types are there.
- Introduction rules tell us how to construct inhabitants of a given type.
- Elimination rules tell us how to use an inhabitant of some type to construct inhabitants of other types.
- Computation rules tell us how an elimination rule acts on an introduction rule – computation happens when we first build something and then take it apart.
- Uniqueness rules tell us how an introduction rule acts on an elimination rule – if we first take something apart and then rebuild it, we should get the same thing we initially had.

How does this play out in practice? Let's take a look at an example development, which illustrates the basic workings of Coq. At this point, we strongly encourage the unfamiliar reader to install CoqIDE (a dedicated IDE for Coq, available from [9]) and run this snippet (which can be found in the thesis sources in the directory `snippets/`) in interactive mode – this experience will be better than any explanation.

```
Print nat.
(*
Inductive nat : Set :=
    | O : nat
    | S : nat -> nat.
*)

Inductive Tree (A : Type) : Type :=
    | Leaf : A -> Tree A
    | Node : Tree A -> Tree A -> Tree A.

Arguments Leaf {A} _.
Arguments Node {A} _ _.

Fixpoint label
  {A : Type} (t : Tree A) (n : nat) : nat * Tree (A * nat) :=
match t with
    | Leaf x => (n, Leaf (x, n))
    | Node l r =>
        let (n', l') := label l n in
        let (n'', r') := label r (S n') in
          (n'', Node  l' r')
end.

```

```
25 Definition lbl {A : Type} (t : Tree A) : Tree (A * nat) :=
26   snd (label t 0).
27
28 Compute lbl (Node (Node (Leaf true) (Leaf true)) (Leaf false)).
29 (* = Node (Node (Leaf (true, 0)) (Leaf (true, 1))) (Leaf (false, 2))
30    : Tree (bool * nat) *)
31
32 Fixpoint size {A : Type} (t : Tree A) : nat :=
33 match t with
34     | Leaf _ => 1
35     | Node l r => size l + size r
36 end.
37
38 Require Import Arith.
39
40 Lemma label_size :
41   forall (A : Type) (t : Tree A) (n n' : nat) (t' : Tree (A * nat)),
42     label t n = (n', t') -> S n' = n + size t.
43 Proof.
44   induction t as [| l IHl r IHr].
45     cbn. intros. inversion H. rewrite <- plus_comm. cbn. reflexivity.
46     cbn. intros.
47       case_eq (label l n); intros m1 t1 H1.
48       case_eq (label r (S m1)); intros m2 t2 H2.
49       specialize (IHl _ _ _ H1). specialize (IHr _ _ _ H2).
50       rewrite H1, H2 in H. inversion H; subst.
51       rewrite IHr, IHl. rewrite plus_assoc. reflexivity.
52 Qed.
```

A Coq file consists of a series of commands, which define types and terms, import and export modules, state or look up theorems etc. In this file we want to implement a function that labels the leaves of a tree with natural numbers starting from 0.

For this, we will need the type `nat`, which is provided by the standard library. We can print it's definition using the command `Print`. What we get is an inductive definition of a type with two constructors, `O` and `S`. `O` is supposed to represent 0 and `S` is supposed to represent the successor operation, so that `S O` represents 1, `S (S O)` represents 2 and so on.

We will also need a type `Tree` that represents trees. The definition will be inductive, just as for natural numbers, but this time is has a parameter A : `Type`

that tells us the type of elements that can be stored in the trees. This can be seen as defining infinitely many types at once – one for each choice of A. There are two constructors: `Leaf`, which represents a tree with one element, and `Node`, which represents a tree with two subtrees. Thus our trees are always nonempty and they contain elements only in their leaves.

The next two commands, `Arguments`, control the use of implicit arguments. Normally, we would have to write `Leaf nat` 42 for the tree containing only the value 42, but this is redundant, since given a term like `42`, we acan infer its type to be `nat`. We can thus write this as `Leaf _ 42`, but this is still redundant. Thanks to the first of these two commands, we can write simply `Leaf 42`.

The function `label` is the clou of the whole development. It's type says that given a type `A` (which we don't need to give explicitly, since it's an implicit argument), a tree `t` and a number `n` which acts as the counter, it returns a pair consisting of a natural number (the new counter) and a tree contaning elements of type `A * nat`.

The definition is recursive (marked by the keyword `Fixpoint`).

Chapter 2

Computational effects

Chapter 3

Design

Chapter 4

Examples

Chapter 5

A case study in proof engineering

Chapter 6

Conclusion

Bibliography

[1] Jeremy Gibbons and Ralf Hinze,
Just do It: Simple Monadic Equational Reasoning, 2011
`http://www.cs.ox.ac.uk/jeremy.gibbons/publications/mr.pdf`

[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
`https://meltdownattack.com/meltdown.pdf`

[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
`https://spectreattack.com/spectre.pdf`

[4] `https://heartbleed.com`, 2019

[5] `https://deepspec.org`, 2019

[6] Georges Gonthier, *A computer-checked proof of the Four Colour Theorem*, 2005
`https://www.cl.cam.ac.uk/~lp15/Pages/4colproof.pdf`

[7] Georges Gonthier, *Formal Proof – The Four-Color Theorem*, 2008,
`http://www.ams.org/notices/200811/tx081101382p.pdf`

[8] Vladimir Voevodsky, *UNIVALENT FOUNDATIONS*, slides for a talk given at IAS on 26 March 2014,
`http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2014_IAS.pdf`

[9] `https://coq.inria.fr/`

[10] Morten Heine Sørensen, Paweł Urzyczyn,
Lectures on the Curry-Howard Isomorphism, 2006