Formalnie zweryfikowane algorytmy w Coqu: koncepcje i techniki

Wojciech Kołowski

24 września 2021

1. Historia
2. Motywacja i efekty
3. Streszczenie pracy
4. Co dalej

Geneza pracy

- Około połowy 2017 roku postanowiłem sformalizować sobie kilka algorytmów, głównie sortowania, i innych tego typu pierdółek.
- W semestrze zimowym 2017/2018 brałem udział w kursie Algorytmy Funkcyjne i Trwałe Struktury Danych (czy jak on tam się zwał), który skłonił mnie do sięgnięcia po książkę Okasakiego i próbę sformalizowania tego, co w niej znajdę.
- Przez kolejne +- dwa i pół roku projekt ewoluował i działy się w nim różne rzeczy, takie jak badania nad dowodzeniem przez reflekcję czy memoizacją.
- Znaczna większość czasu i kodu poświęcona została strukturom danych i operującym na nich funkcjom.
- Mimo tego, największym źródłem odkryć i najbardziej istotnym dla napisania pracy elementem projektu było porównanie algorytmów sortowania, którego się podjąłem.

Backstory

That side-project of mine had two goals: first, formalize some data structures from Okasaki's book; second, see how easy (or hard) it is to formally prove correctness of functional algorithms. Even though most code (and time spent) concerns the first of these goals, most insight I gained concerns the second one. I have discovered the core of the approach described in this thesis while working on a comparison of sorting algorithms. The sheer amount of different algorithms and various versions of the same algorithm made such a highly abstract and modular approach pretty much necessary, and Coq's powerful type system made it feasible. In the end I learned that for a skilled Coq user armed with my approach, proving the algorithms correct is not much harder than implementing them in the first place and that both implementation and verification become easier when considered together.

Motywacje teoretyczne

- todo

Kontrybucje

I have stumbled upon some of the techniques presented in this thesis semi-autonomously, but I don't take any credit for inventing (or discovering) them. For example, I "knew" about (hole-) and (type-)driven development before starting this project, but I had to reinvent the wheel from scratch myself, driven purely by the needs of proof engineering, in order to really grasp, absorb and assimilate them. My only original addition to it is the idea of abstracting over holes, which I haven't seen described anywhere else, but I'm pretty sure that people more knowledgeable on the matter have also encountered this idea. I think that my greatest contribution is simply putting all of these concepts and techniques into a coherent and effective approach to functional algorithms.

Some techniques described in this thesis, like functional induction, are well-known, but lacked good expository material, which I provided in section FunInd. Others, like the Bove-Capretta method, are both well-known and widely present in the literature, but

Abstrakt

Omawiamy sposoby specyfikowania, implementowania i weryfikowania funkcyjnych algorytmów, skupiając się raczej na dowodach formalnych niż na asymptotycznej złożoności czy faktycznej wydajności. Prezentujemy koncepcje i techniki, obie często opierające się na jednej kluczowej zasadzie – reifikacji i reprezentacji, za pomocą potężnego systemu typów Coqa, czegoś, co w klasycznym, imperatywnym podejściu jest nieuchwytne, jak przepływ informacji w dowodzie czy kształt rekursji funkcji. Nasze podejście obszernie ilustrujemy na przykładzie quicksorta. Ostatecznie otrzymujemy solidną i ogólną metodę, którą można zastosować do dowolnego algorytmu funkcyjnego.

Konkluzja

In this thesis, we have described concepts and techniques useful for implementing and verifying functional algorithms. We worked in Coq, but our ideas readily transfer to other dependently typed languages, like Agda, Idris or F*. Some of the techniques (mostly those concerned with specification and implementation of the abstract template) are probably also useful in languages with weaker type systems, like Haskell, OCaml or F#.

TLDR

We have presented 15 concepts/techniques in total (numbered 0 through 14), but they neither capture everything that was described in this thesis, nor are they really unique. They also don't show the big picture – in fact, they are better thought of as subitems of the following five-item master plan which shows the steps that lead from a problem to a formally verified, user-extensible algorithm that solves it.

Find a good specification of the problem.

- #0: Good means abstract and easy to use.
- #1: Good specification determines a unique object.
- #2: Bad specifications can possibly be improved with defunctionalization.
- #3: Sometimes better specifications can be found by focusing on a different aspect of the problem.

Find an algorithm that solves the problem

- Easier said than done.

Implement a template of the algorithm that abstracts over the exact details

- #4: While implementing the template, ignore termination at first.
- #6: Types of components of your template should contain enough evidence, but not too much!
- #7: When dealing with many templates at once, make shared components into parameters. In all other cases, prefer bundled classes.
- #8: Use the Inductive Domain Method to define a better, provably terminating algorithm template.

Prove termination and correctness of the algorithm template

- #12: Use the technique of Proof by Admission.
- #9: Outline of termination proof: well-founded induction.
- #11: Outline of correctness proof: functional induction.

Provide a concrete default implementation together with all the proofs

- #5: Provide a default implementation.
- #10: Make sure that a default implementation can be run without any proof obligations, and that a provably terminating implementation can be run without having to prove correctness.
- #13: If your default implementation or its proofs are too abstract, provide a more concrete version.
- #14: When looking for default and concrete implementations, use type-driven development.

Related and further work

The approach presented in this thesis is powerful, but not complete and could be improved in many respects. Although we have seen some criteria for judging specifications and some techniques for improving them, it would be nice to have a general method for coming up with good specifications. Even though we did discuss well-founded induction, we haven't seen any techniques for constructing the most convenient well-founded relation for a given termination proof. We have also seen that the inductive domain method has some problems when dealing with functions that exhibit nested recursion or higher-order recursion.

Moreover, we have skipped the issue of actually inventing the algorithm that we want to formalize. The classical-imperative paradigm knows many techniques of algorithm design and most of them transfer easily to the functional paradigm, but it would be very interesting to see how they interact with the rest of our approach – maybe it's possible to synthesise the algorithm directly