

Formalnie zweryfikowane algorytmy w Coqu: koncepcje i techniki

Wojciech Kołowski

27 września 2021

- 1 Historia
- 2 Streszczenie pracy
- 3 Podsumowanie i co dalej

Geneza pracy

- Około połowy 2017 roku postanowiłem sformalizować sobie kilka algorytmów i struktur danych. Tak powstał projekt żyjący pod adresem <https://github.com/wkolowski/coq-algs>
- 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 znajduję.
- Przez kolejne $+\frac{1}{2}$ dwa i pół roku projekt ewoluował i działał w nim różne rzeczy, takie jak badania nad spamiętywaniem czy dowodzeniem przez refleksję.

Skąd odkrycia?

- Pewnego razu postanowiłem sformalizować pokaźny wachlarz przeróżnych algorytmów sortowania.
- Podjąłem się go częściowo z nudów, a częściowo żeby nieco systematyczniej zbadać, jak trudno (lub łatwo) formalizuje się algorytmy w Coqu i dowodzi ich poprawności.
- Wcześniejsze formalizacje algorytmów w tym projekcie nie dawały dobrej odpowiedzi na to pytanie, bo dotyczyły przypadkowo dobranych problemów, a rozwiązania pochodziły głównie z książki Okasakiego.

Algorytmy sortowania motorem napędowym nauki

- Spora liczba algorytmów i jeszcze większa mnogość ich wariantów bardzo szybko wymusiły wysoce abstrakcyjne i modularne podejście, a potężny system typów Coqa umożliwił (niemal) bezproblemową realizację tej wizji.
- W miarę postępów formalizacji poznałem też całą masę uniwersalnych prawidłowości, które następnie przekułem w pracy na tytułowe koncepcje i techniki.
- Ostatecznie przekonałem się, że dla wprawnego użytkownika Coqa, uzbrojonego w moje koncepcje i techniki, dowodzenie poprawności algorytmów jest niewiele trudniejsze niż sama ich implementacja (choć jest dużo bardziej pracochłonne).
- Co więcej, zarówno implementacja jak i weryfikacja stają się łatwiejsze, gdy rozważamy je razem.

Pareto wiecznie żywy

- Porównanie algorytmów sortowania okazało się więc być strzałem w dziesiątkę.
- Zadziałała też niezawodna zasada Pareto: mimo, że znaczna większość czasu i kodu poświęcona została podstawowym strukturom danych takim jak kolejki, sterty czy samobalansujące się drzewa wyszukiwań, to największym źródłem odkryć i najbardziej istotnym dla napisania pracy elementem projektu było właśnie porównanie algorytmów sortowania.

Motywacje

- Poznaawszy wspaniałe techniki radzenia sobie z formalizacją algorytmów stosowalne we wszystkich w zasadzie językach z typami zależnymi, postanowiłem podzielić się nimi ze światem.
- Głównym celem pracy było bardzo przyjazne i dostępne dla zwykłych śmiertelników opisanie tego wszystkiego, w formie tutorialowej zbliżonej stylem do najlepszych dydaktycznie znanych mi postów na blogach.

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 Coq, 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.

Jak sformalizować funkcyjny algorytm?

- Praca prezentuje 15 technik przydatnych przy formalizacji algorytmów funkcyjnych.
- Liczba 15 jest trochę naciągana, zresztą nie o ilość chodzi.
- Najlepiej rozumieć je jako podpunkty uniwersalnego planu, który prowadzi nas od problemu do formalnie zweryfikowanego algorytmu, który go rozwiązuje.

Znajdź specyfikację problemu (sekcja 1.6 i rozdział 2)

- #0: Dobra specyfikacja jest abstrakcyjna i prosta w użyciu.
- #1: Dobra specyfikacja określa unikalny obiekt.
- #2: Patchworkowa specyfikacja może zostać ulepszona przy użyciu defunkcjonalizacji (czyli przy użyciu definicji induktywnych zamiast kwantyfikatora uniwersalnego).
- #3: Czasem można znaleźć lepszą specyfikację skupiając się na innym aspekcie zagadnienia.

Znajdź algorytm, który rozwiązuje problem

- Łatwiej powiedzieć niż zrobić.
- Praca nie porusza tematyki wymyślania czy projektowania algorytmów - zakładamy, że formalizujemy znany już algorytm.
- Ostatnio wyszła ciekawa książka o projektowaniu algorytmów funkcyjnych, ale jeszcze nie czytałem: Algorithm Design with Haskell

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.

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#.

Główna kontrybucja pracy

- Główną kontrybucją pracy jest systematyzacja wielu technik, ich synteza w spójną metodę i zastosowanie tej metody konkretnie do problemu formalizacji algorytmów funkcyjnych.
- Uwaga: nie przypisuję sobie odkrycia w zasadzie żadnej z opisanych przeze mnie koncepcji i technik – część była znana już wcześniej, część ma status pewnego folkloru, a część jest na tyle nieuchwytna, że trzeba je wymyślić na nowo, żeby porządnie je zrozumieć.

Pomniejsze kontrybucje

- Udało mi się w przyjazny i zrozumiały sposób opisać także wiele znanych już technik, które jednak dotychczas
- Indukcja funkcyjna, elegancka metoda dowodzenia właściwości funkcji rekurencyjnych, nie została nigdzie opisana w sposób przyjazny dla początkujących.
- Metoda Bove-Capretta (co za okropna nazwa!) została opisana w wielu pracach, ale nie są one przyjazne dla początkujących.
- Type-driven development został przyjaźnie opisany, ale książka jest płatna i trochę przeterminowana.
- Spamowanie taktyką `admit` nie zostało nigdzie opisane jako systematyczna metoda dowodzenia i nie ma nawet nazwy, a jego związki z *Hole-driven development* są nie do końca oczywiste.

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