Formalnie zweryfikowane algorytmy w Coqu: koncepcje i techniki

Wojciech Kołowski

27 września 2021

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 znajdę.
- Przez kolejne $+-$ dwa i pół roku projekt ewoluował i działy się w nim różne rzeczy, takie jak badania nad spamiętywaniem czy dowodzeniem przez reflekcję.

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

- Poznawszy 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 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.

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 te techniki 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 kwantyfikacji uniwersalnej, implikacji i funkcji).
- #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.

Zaimplementuj abstrakcyjny szablon algorytmu (rozdział 3)

- #4: Przy implementacji pierwszego szablonu nie przejmuj się terminacją.
- #6: Typy parametrów szablonu powinny zawierać wystarczająco *evidence* (żeby np. nie popaść w ślepotę boolowską), ale nie za dużo.
- #7: Jeżeli implementujesz wiele szablonów na raz (wielu powiązanych algorytmów), ich współdzielone komponenty powinny być parametrami szablonu. Pozostałe komponenty powinny być jego polami.

Udowodnij terminację i poprawność (rozdziały 4 i 5)

- #8: Użyj Metody Induktywnej Dziedziny (to mój rebranding metody Bove-Capretta) żeby zdefiniować lepszy szablon algorytmu, którego terminację łatwo udowodnić.
- #12: W dowodzeniu niesamowicie przydatna jest technika *proof by admission*, czyli formalny odpowiednik dowodu przez machanie rękami. W skrócie: jeżeli nie potrafisz czegoś udowodnić, załóż że to prawda. Później udowodnij lemat, który wypełni tę lukę, albo przyjmij dodatkowe założenie.
- #9: Dowód terminacji sprowadza się do indukcji dobrze ufundowanej – wystarczy znaleźć jakąś rozsądną relację dobrze ufundowaną lub miarę.
- #11: Dowód poprawności szablonu sprowadza się do indukcji funkcyjnej.

Dostarcz domyślną implementację (rozdziały 5 i 3)

- #5: Dostarcz domyślną implementację.
- #10: Upewnij się, że domyślna implementacja może zostać uruchomiona bez dowodzenia czegokolwiek (terminacji/poprawności) i że terminująca implementacja może zostać uruchomiona bez dowodzenia poprawności.
- #13: Jeżeli twoja domyślna implementacja jest zbyt abstrakcyjna, dostarcz bardziej konkretną wersję.
- #14: Do zdefiniowana domyślnej i konkretnej implementacji użyj type-driven development.

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 dotychczas były nieopisane lub były opisane słabo.
- 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 oficjalnej nazwy (ja nazwałem to *proof by admission*), a jego związki z *Hole-driven development* są nie do końca oczywiste.

Co dalej - ogólnie

- Zaprezentowane w pracy podejście do algorytmów funkcyjnych nie jest kompletne i można je rozszerzyć i poprawić.
- Nie znam żadnej ogólnej metody wymyślania specyfikacji.
- Pominięte zostały metody wymyślania/projektowania algorytmów, gdyż mieleniem ich zajmują się wszystkie możliwe kursy i książki, np. klasyczna książka Okasakiego i niedawno wydana *Algorithm Design with Haskell*.
- Analiza złożoności mogłaby także nieco zyskać na formalnym podejściu, choćby dlatego że możemy formalnie dowodzić zależności rekurencyjnych. Z drugiej strony analiza komplikuje się przez problemy związane np. z obecnością predykatu dziedzinowego.

Co dalej - terminacja

- Wspomniałem o problemach standardowej Metody Induktywnej Dziedziny z funkcjami z zagnieżdżoną rekursją i rekursją wyższego rzędu, i jak sobie z tym poradzić, ale nie pokazałem, jak dokładnie należy to zrobić.
- Nie omówiłem żadnych metod konstruowania relacji dobrze ufundowanych.
- Nie omówiłem też alternatywnych metod dowodzenia terminacji jako HORPO (*Higher Order Recursive Path Ordering*) czy *Size-Change Principle*.
- Milczałem też o algorytmach korekurencyjnych i dowodzeniu ich produktywności, choć to temat raczej badawczy niż dydaktyczno-folklorystyczny.

Co dalej - bujanie w chmurach

- Zdaje się, że istnieje linia badań nad "obliczaniem" algorytmu ze specyfikacji za pomocą rozumowań równaniowych. W połączeniu z techniką wymyślania specyfikacji byłoby to naprawdę potężne combo.
- Ciekawą rzeczą jest Kombinatoryka Analityczna, która pozwala obliczyć liczbę struktur (np. drzew) danego rodzaju z samej definicji typu induktywnego, dzięki czemu stanowi pomost między specyfikacją a analizą złożoności.
- Testowanie może się wydawać zbędne, jednak *property-based testing* jest bardzo dobry w znajdowaniu kontrprzykładów, co może znacznie skrócić czas od popełnienia błędu do wykrycia go – moglibyśmy zrobić to już na etapie domyślnej implementacji, czyli niedługo po zaimplementowaniu pierwszego szablonu, zamiast dopiero na końcu, przy dowodzie poprawności.