Formally verified algorithms and data structures in Coq: concepts and techniques

(Formalnie zweryfikowane algorytmy i struktury danych w Coqu: koncepty i techniki)

Wojciech Kołowski

Praca magisterska

Promotor: narazienikt

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Czerwiec '20 chyba że koronawirus

Abstract

We discuss how to design, implement, specify and verify functional algorithms and data structures, concentrating on formal proofs rather than asymptotic complexity or actual performance. We present concepts and techniques, both of which often rely on one key principle – the reification and representation, using Coq's powerful type system, of something which in the classical-imperative approach is intangible, like the flow of information in a proof or the shape of a function's recursion. We illustrate our approach using rich examples and case studies.

Omawiamy sposoby projektowania, implementowania, specyfikowania i weryfikowania funkcyjnych algorytmów i struktur danych, skupiając się bardziej na dowodach formalnych niż na asymptotycznej złożoności czy faktycznym czasie działania. Prezentujemy koncepty i techniki, obie często opierające 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 bogato ilustrujemy przykładami i studiami przypadku.

Contents

Chapter 1

Introduction

TODO: after completing the thesis, write a short overview of what each chapter is about.

1.1 An overarching paradigm

The Free Dictionary says [1] that an algorithm is

> A finite set of unambiguous instructions that, given some set of initial conditions, can be performed in a prescribed sequence to achieve a certain goal and that has a recognizable set of end conditions.

The purpose of this entry is to explain the concept to a lay person, but it likely sounds just about right to the imperative programmer's ear too. To a functional ear, however, talking about sequences of instructions most certainly sounds as unfunctional as it possibly could. It is no surprise then that some people wonder if it is even possible for algorithms to "exist" in a functional programming language, as exemplified by this StackOverflow question [2]. The poor soul asking this question had strongly associated algorithms with imperative languages in his head, even though functional languages are based on lambda calculus, which was invented precisely to formalize what an algorithm is.

This situation is not uncommon and rather easy to explain. Imperative algorithms and data structures [1] form one of the oldest, biggest, most widespread and prestigious fields of computer science. They are taught to every student in every computer science programme at every university. There's a huge amount of books and textbooks, with classics such as [3] [4] known to pretty much everybody, at least by title. There's an even huger and still growing mass of research articles and confer-

[1] From now on when we write "algorithms" we will mean "algorithms and data structures"

ences and I'm pretty sure there are at least some (imperative) algorithm researchers at every computer science department in existence.

Imperative algorithms are pretty much synonymous with competitive programming, dominating most in-person and online computer science competitions like ICPC and HackerRank, respectively. They are seen as the thing that gifted high school students interested in computer science should pursue – each time said students don't win medals in the International Olympiad in Informatics, there will be some journalists and politcians complaining about the state of a country's education system.

Imperative algorithms don't form a mere field of study – they are more of a mindset and a culture than a field; in short, following Kuhn [5], they can be said to form a paradigm. Because this paradigm is so vast, so powerful and so entrenched, we feel free to completely disregard it and devote ourselves and this thesis to studying another field (which did not yet reach the status of a paradigm) – functional algorithms and data structures, focusing on proving their formal correctness.

But before we do that, we spend the rest of this chapter on comparing the imperative and functional approaches to algorithms and briefly reviewing available literature on functional algorithms.

1.2 Two flavours of algorithms

The differences between imperative and functional algorithms are mostly a reflection of the differences between imperative and functional programming languages and only in a small part a matter of differences in focus. For example, it's rare but possible to implement persistent data structures in imperative languages, whereas it's the default for functional data structures [6], all caused by built-in language features. An example of the second kind of difference may be the bigger focus on lazy evaluation for functional algorithms.

The basic data structure in imperative languages is the array, which abstracts a contiguous block of memory holding values of a particular type. More advanced data structures are usually records that hold values and pointers/references to other (or even the same) kinds of data structures. The basic control flow primitives for traversing these structures are various loops (`while`, `for`, `do ... while`) and branch/jump statements (`if`, `switch`, `goto`). The most important operation is assignment, which changes the state held in a variable. Computation is modeled by a series of operations which change the global state.

In functional languages both basic and advanced data structures amount to algebraic data types (whose values are various kinds of trees with labeled nodes that can hold values of specified types). The basic control flow primitives are pattern matching (checking the label and values of the tree's root) and recursion. The

most important operation is function composition, which allows building complex functions from simpler ones. Computation is modeled by substitution of values for the formal parameters of a function.

```
int sum(int[] a)
{
    int result = 0;
    for(int i = 0; i < a.length; ++i)
    {
        result += a[i];
    }
    return result;
}
```

Listing 1: A simple program for summing all integers stored in an array, written in an imperative pseudocode that resembles Java.

```
data List a = Nil | Cons a (List a)

sum : List Int -> Int
sum Nil = 0
sum (Cons x xs) = x + sum xs
```

Listing 2: A simple program for summing all integers stored in a (singly-linked) list, written in a functional pseudocode that resembles Haskell.

The two above listings showcase the relevant differences in practice. In both cases we want a function that sums integers stored in the most basic data structure. In the case of our pseudo-Java, this is a built-in array, whereas in our pseudo-Haskell, this is a list defined, using the mechanism of algebraic data types, as sometihng which is either empty (`Nil`) or something that contains a value of type `a` and then another list.

1.3 Complexity, performance and correctness

Differences between performance-oriented design and formal-correctness-oriented design.

1.4 An overview of available literature

Literature review, Okasaki is old and bad for Coq, SF3 is shallow.

Chapter 2

Binary search trees – an extended case study

Binary search trees: a case study to show the basic workflow and that it's not that obvious how to get basic stuff right.

Chapter 3

A man, a plan, a canal – MSc thesis

3.1 Things to write about

- Design: we shouldn't require proofs in order to run programs. Ways of doing general recursion and, connected with it, functional induction as the way-to-go proof technique. Maybe something about the equations plugin. A word about classes, records and modules.
- Quicksort: in functional languages we have so powerful abstractions that we can actually implement algorithms and not just programs.
- Braun mergesort: in order not to waste resources, we sometimes have to reify abstract patterns, like the splitting in mergesort.
- Cool data structures: ternary search trees, finger trees.

Bibliography

[1] https://www.thefreedictionary.com/algorithm

[2] *Do "algorithms" exist in Functional Programming?*,
https://stackoverflow.com/questions/25940327/
do-algorithms-exist-in-functional-programming

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein,
Introduction to Algorithms,
http://ressources.unisciel.fr/algoprog/s00aaroot/aa00module1/res/
%5BCormen-AL2011%5DIntroduction_To_Algorithms-A3.pdf

[4] Donald Knuth, *The Art of Computer Programming*

[5] Thomas S. Kuhn, *The Structure of Scientific Revolutions*

[6] Driscoll JR, Sarnak N, Sleator DD, Tarjan RE, 1986
Making data structures persistent