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

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

Wojciech Kołowski

Praca magisterska

Promotor: oficjalnienarazienikt

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 have their roots in lambda calculus, a formal system invented precisely to capture 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 theoretical 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 published in journals and conferences and I'm pretty sure there are at least

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

some (imperative) algorithm researchers at every computer science department in existence.

But theoretical research and higher education are not the only strongholds of imperative algorithms. They are pretty much synonymous with competitive programming, dominating most in-person and online computer science competitions like ICPC and HackerRank, respectively. They are also 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 or the like, there will be some journalists complaining that their country's education system is "falling behind". In many countries, like Poland,[2] if there's any high school level computer science education besides basic programming, it will be in imperative algorithms.

Imperative algorithms aren't just a mere field of study – they are more of a mindset and a culture; following Kuhn's [5] terminology, 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 a related field which did not yet reach the status of a paradigm – functional algorithms – 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 research focus.

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 value of a variable (and thus variables do actually vary, like the name suggests). Computation is modeled by a series of operations which change the global state.

In functional languages the basic data structures are created using the mechanism of algebraic data types – elements of each type so defined are trees whose node

[2] I believe the same is true for the rest of the former Eastern Bloc countries too and probably not much better in the West either.

labels, branching and types of values held are specified by the user. 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 arguments for the formal parameters of a function. Variables don't actually vary – they are just names for expressions.

```
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 programs 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 something which is either empty (`Nil`) or something that has a head of type `a` and a tail, which is another list of values of type `a`.

In the imperative program, we declare a variable `result` to hold the current value of the sum and then we loop over the array. We start by creating an iterator variable `i` and setting it to `0`. We successively increment it with each iteration of the loop until it gets bigger than the length of the array and the loop finishes. At each iteration, we modify `result` by adding to it the array entry we're currently looking at.

In the functional program, we pattern match on the argument of the function. In case it is `Nil` (an empty list), we declare the result to be `0`. In case it is `Cons x xs` (a list with head `x` and tail `xs`), we declare that the result is computed by adding `x` and the recursively computed sum of numbers from the list `xs`.

Even though these programs are very simple, they depict the basic differences between imperative and functional algorithms quite well. Some less obvious differences are as follows.

First, functional data structures are by default immutable and thus persistent [6], whereas this is not so for imperative data structures – they have to be explicitly designed to support persistence. This means implementing some techniques, like backtracking, is very easy in functional languages whereas it often requires much more effort in imperative languages. The price of persistence often is, however, increased memory usage.

Second, pointer juggling in imperative languages allows more efficient implementation of some operations on tree-like structures than using algebraic data types, because nodes in such trees can have pointers not only to their children, but also to parents/grandparents/siblings etc. The most famous data structure whose purely functional, asymptotically optimal implementation is not known is union-find [7].

A related point is that arrays, the bread-and-butter of imperative programming, provide random access read and write in $O(1)$ time, whereas equivalent random access functional structures work in $O(\log n)$ where n is the array size (or, at best, $O(\log i)$ where i is the accessed index). This means that algorithms relying on constant time random access will suffer an asymptotic performance penalty when implemented in a functional language. Not all hope is lost however, because arrays are not necessarily forbidden in purely functional languages – if the destructuve updates are packed up in a function so that they can't be seen from the outside, then basically this counts as purely functional.

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
Making data structures persistent, 1986

[7] Sylvain Conchon, Jean-Christophe Fillâtre,
A Persistent Union-Find Data Structure, 2007
https://www.lri.fr/~filliatr/ftp/publis/puf-wml07.pdf