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

The title of this thesis is "Formally verified functional algorithms and data structures in Coq: concepts and techniques". In the Introduction, we take time to carefully explain what we mean by each of these phrases:

- In section 1.1 we look at the social context that gives rise to an interesting misconception about "algorithms and data structures".
- In section 1.2 we explain "functional" programming by comparing it with imperative programming.
- In section 1.3 we lay out the unity of the "formally verified" algorithmic world by contrasting it with the many worlds which are the arenas of the imperative algorithmic struggles.
- In section 1.4 we do an ultra short literature review and discuss how our "concepts and techniques" differ from these that can be found there.
- In section 1.5 we show how working "in Coq" looks like and describe some theory and principles Coq is based on.

In the remaining chpaters we describe the actual concepts and techniques, using well-known and well-studied examples like quicksort, merge sort or binary heaps:

- TODO

1.1 The 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 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 also 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 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 immense, 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.

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

[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.

But before we do that, we spend the rest of this chapter 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 the fields 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 [6]). 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 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. [6]

```
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 [7], whereas this is not the case 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, but 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 a 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, siblings, etc. The most famous data structure whose functional, asymptotically optimal implementation is not known is union-find. [8]

The third point is that arrays, the bread-and-butter of imperative programming, provide random access read and write in $O(1)$ time, whereas equivalent functional random access structures work in $O(\log n)$ time 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 functional languages. [9]

Even though this asymptotic penalty sounds gloomy, not all hope is lost, because of two reasons. First, mutable arrays can still be used in purely[3] functional languages if the mutability is hidden behind a pure interface. An example of this is Haskell's ST monad. [10] Second, functional languages that are impure, like Standard ML or OCaml, allow using mutable arrays without much hassle, which often saves the day.

1.3 The many-worlds interpretation of imperative algorithms

We stated earlier that we intend to concentrate on formally proving correctness of purely functional algorithms. Before doing that, we take a short detour to look at how it differs from the activities and workflows of the usual algorithmic business,[4] what we embrace and what we're leaving out.

When an algorithmist encounters a problem, let's say "How do I sort a list of integers?", he will follow a path towards the solution which looks roughly like this:

- Formulate a (more) precise specification of the problem.
- Design an algorithm that solves the problem and write it down using some kind of pseudocode, keeping a good balance between generality and detail.
- Prove that the algorithm is correct. If a proof is hard to find, this may be a sign that the algorithm is incorrect.
- Analyze complexity of the algorithm, i.e. how much resources (number of steps, bits memory, bits of randomness, etc.) does the algorithm need to solve the problem of a given size. If the algorithm needs too much resources, go back and try to design another one that needs less.
- Implement the algorithm and test whether the implementation is correct.
- Run some experiments to assess the actual performance of the implementation, preferably considering a few common scenarios: random data, data that often

[3] "Pure" and "impure" are loose terms used to classify functional languages. "Pure" roughly means that a language makes organized effort to separate programs that do ordinary number crunching or data processing from those that have side effects, like throwing exceptions or connecting to a database. An "impure" languages doesn't attempt at such a separation.

[4] A person engaging in the usual algorithmic business we will call an *algorithmist*.

occurs in the real world, data constructed by an evil adversary, etc. If the performance is unsatisfying, try to find a better implementation or go back and design a better algorithm.

Of course this recipe is not stiff and some variations are possible. The two most popular ones would be:

- The extremely practical, in which the specification and proof are dropped in favour of the test suite, the pseudocode is dropped in favour of the actual implementation, and the complexity analysis is only cursory and performed on the go during the design phase, in the algorithmist's head. This approach permeates comeptitive programming, because of the time pressure, and industry, because most "algorithms" there amount to unsophisticated data processing that doesn't need specification or proof.
- The extremely theoretical, in which there is no implementation and thus no tests and no performance assessment, and the most time-consuming part becomes the complexity analysis. This approach is widespread in teaching, where the implementation part is left to students as an exercise, and in theoretical research, where real-world applicability is not always the most important goal.

No matter the exact recipe, there is a very enlightening thing to be noticed, namely all the different worlds in which these activites take place. For example, the algorithm design process takes place in the algorithmist's mind, but after he's done, it is usually written down in some kind of comfortable pseudocode that allows skipping inessential detail. What's more, the actual implementation, in contrast to the pseudocode, will be in some concrete programming language – a very popular one among algorithmists is `C++`.

The specification and the proof are usually written in English (or whatever the language of the algorithmist), intermingled with some mathematical symbols for numbers, relations, sets and quantifiers, but that's only the surface view. If we take a closer look, it will most likely turn out that the mathematical part is based on classical first-order logic[5] and some kind of naive set theory, but if pressed a bit, the algorithmist would readily assert that the set theory could be axiomatized using, let's say, the Zermelo-Fraenkel axioms.

The next hidden world, in which the complexity analysis takes its place, is the model of computation. In most cases it is not mentioned explicitly, just like logic and set theory in the previous paragraph, but usually it's easy to guess. Most algorithms are, after all, intended to be implemented on modern computers, and the model of computation most similar to the workings of real hardware is the RAM machine.

[5]By "classical" we mean that the logic admits nonconstructive reasoning principles like proof by contradiction [11] – its use can be seen, for example, in proofs of optimality.

As we see, the typical solution of an algorithmic problem stems from a sequence (or rather, a loop) of activities which live in six different worlds: the world of abstract ideas, represented by the pseudocode, the world of programming, represented by the implementation language, the world of formal math, the world of informal math, the world of idealized execution, represented by the model of computation, and the world of concrete execution, represented by the hardware.

Even though the above many-worlds recipe and its variations work well in practice, as evidenced by the huge number of algorithms humanity put to use for solving its everyday problems, an inquisitive person interested in formal verification (or perhaps a philosopher) could ask a plethora of questions about how all these worlds fit together:

- Does the implementation correspond to the pseudocode?
- Could the informally stated specification and proof really be cast into the claimed formal system?
- Does the model of computation actually model the hardware well?
- Could the model of computation (and the complexity analysis) be formalized?
- Assuming the implementation language is compiled, how is the source program related to machine code executed by the hardware, i.e. is compilation correct?
- Does the analysis agree with the implementation language semantics?[6]

Each of these questions can be seen as pointing at a link between two worlds which is a potential source of errors – the worlds may not correspond too well. Because each pair of worlds can give rise to a potential mismatch, in theory there is a lot to worry about.

We allowed ourselves to wander into this lengthy overview of the usual algorithmic business in order to contrast it with our approach to formal verification of algorithms, which can be seen as getting rid of potential errors due to world mismatches by transfering (nearly) all of the activities into a single, unified, formal world.

This formal world is Coq, a language which can be used both for functional programming and proving theorems (we will give a more extensive overview in the next section). First, we no longer need to do pen-and-paper specifications and proofs, so the worlds of formal and informal mathematics collapse into one. Second, Coq has very powerful abstraction capabilities, which bypass the need for pseudocode – we can directly implement the high-level idea behind the algorithm.[7] This merges the worlds of ideas and programming into one.

[6] If it's the implementation that is analyzed and not the pseudocode.

[7] An example of this will be provided in 2.

1.4 An ultra short literature review

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

1.5 Way of the Coq

a type theory-based proof assistant and a dependently typed (purely) functional programming language at the same time.

Calculus of Constructions[8]

[8]Calculus of Constructions, in short CoC, was invented by Thierry Coquand. Another very similar system is Martin-Löf Type Theory[**?**]

Chapter 2

A quick *tour de* sort

Chapter 3

A man, a plan, a canal – MSc thesis

3.1 Design

- First step: specification (describe the path from intuition to formal spec).
- Second step: design (describe the path from the concrete to the abstract by tracing a stub proof of correctness).
- We shouldn't require proofs in order to run programs. Clou: packed vs unpacked classes/records/modules.

3.2 Techniques

- General recursion: Bove-Capretta method as the way to go.
- Functional induction as the way-to-go proof technique. Mention the Equations plugin.

3.3 Topics

- 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.
- Binary heaps: a case study to show the basic workflow and that it's not that obvious how to get basic stuff right.
- 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] Robert Harper, *Words Matter*, 2012
https://existentialtype.wordpress.com/2012/02/01/words-matter/

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

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

[9] Nicholas Pippenger, *Pure versus impure Lisp*, 1996
https://www.cs.princeton.edu/courses/archive/fall03/cs528/
handouts/Pure%20Versus%20Impure%20LISP.pdf

[10] John Launchbury, Simon Peyton Jones,
Lazy Functional State Threads, 1994
https://www.microsoft.com/en-us/research/wp-content/uploads/1994/
06/lazy-functional-state-threads.pdf

[11] Andrej Bauer, *Proof of negation and proof by contradiction*, 2010
http://math.andrej.com/2010/03/29/proof-of-negation-and-proof-by-contradiction/