# 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:** dr Małgorzata Biernacka

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

~~Czerwiec '20 chyba że koronawirus~~
~~Jednak raczej wrzesień~~
Z września też nici, ale spoko

**Abstract**

We discuss how to specify, implement 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 specyfikowania, implementowania i weryfikowania funkcyjnych algorytmów i struktur danych, skupiając się bardziej na dowodach formalnych niż na asymptotycznej złożoności czy faktycznym czasie działania. Prezentujemy koncepcje 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 we look at the social context that gives rise to an interesting misconception about "algorithms and data structures".
- In section 2 we explain "functional" programming by comparing it with imperative programming.
- Section 3 is a philosophical interlude in which we give an interesting interpretation of typical algorithmic activities as living in different worlds, with links between worlds being the source of potential errors.
- In section 4 we lay out the (almost) unity of the "formally verified" algorithmic world by contrasting it with the many worlds of the previous section.
- In section 5 we describe Martin-Löf Type Theory, a formal system very close to the theoretical underpinnings of Coq, which can be seen as our model of computation.
- In section 6 we show how working "in Coq" looks like.
- In section 7 we do an ultra short literature review and discuss how our "concepts and techniques" relate to what can be found there.
- In section 8 we outline the thesis structure.

## 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 competitive 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 activities 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. The actual implementation, in contrast, 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. When pressed a bit, however, the algorithmist will 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 and each such link 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 the approach of formally verified functional algorithms, which is an attempt at getting rid of potential world mismatch errors by transferring (nearly) all of the activities into a single, unified, formal world.

## 1.4 Formal verification as a world-collapser

This formal world is Coq [12]. Coq is a proof assistant – a piece of software whose goal is helping to state and prove mathematical theorems. This help consists of providing a formal language for doing so, called Gallina, a language for automating trivial proof steps and writing proof search procedures, called Ltac, and a languages of commands, called Vernacular, which simplifies tasks like looking up theorems from

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

the standard library. The Coq ecosystem also has many useful tools, like CoqIDE, an IDE with good support for interactive proving.

Thanks to the Curry-Howard correspondence [13], all of these great things can also be interpreted from the programmer's perspective. Gallina is a dependently typed functional programming language, Ltac is a language for automatic program synthesis, and Vernacular offers a suite of facilities similar to those found in most IDEs. CoqIDE can be seen as supporting interactive program construction (and interactively proving a program correct can be seen as a very powerful form of debugging!).

Applied to algorithms, Coq gives us numerous benefits. First, we no longer need to do pen-and-paper specifications and proofs, so the world of informal mathematics gets eliminated from the picture. Second, it has very powerful abstraction capabilities, which bypass the need for pseudocode – we can directly implement the high-level idea behind the algorithm (an example of this will be provided in Chapter 2). This merges the worlds of ideas and programming into one. Third, as already mentioned, the Curry-Howard correspondence unifies functional programming and constructive mathematics, which further collapses the two already-collapsed worlds.

What we are left with are just three worlds instead of the six we had at the beginning: one for programming, proving and expressing high-level algorithmic ideas, one for the analysis (model of computation), and one for execution (hardware). Most of the links between activities now live in the first world and we can use the power of formal proofs to make sure there are no mismatches. We can prove that the algorithm satisfies the specification. If we decide to have more than one implementation (for example when we're looking for the most efficient one), we can prove they are equivalent. We can even prove that the specification is "correct", i.e. that the object it describes is unique. To sum up, we can avoid the various mismatches of previous section using formal proofs inside our unified world.

But as long as not all worlds were collapsed into one, there is still some room for error. First, because Coq is implemented in OCaml, there is no guarantee that our formal proofs are absolutely correct. In fact, as can be seen from `https://github.com/coq/coq/blob/master/dev/doc/critical-bugs`, Coq has experienced about one critical bug per year. This means if we want absolute certainty in our proofs, we need to verify Coq's implementation by hand, so the world of informal math is still somewhat alive.

The second point, related to the first, is that there is no guarantee the semantics of code executed by the hardware is the same as the semantics of the source code. There have been some attempts at formalizing various aspects of Coq in Coq [14] [15] to this effect, but they have the status of exploratory research or work-in-progress. A more serious barrier that prevents us from ever fully formalizing Coq in Coq is Gödel's incompleteness theorem.

Third, the model of computation still lives in a separate world, formally related neither to the code nor to the hardware. What's worse, there is a huge chasm between the preferred models of computation used in computational complexity (Turing machines [16]) and algorithms (RAM machines) on the one hand, and the theory of programming languages (lambda calculi [17]) on the other hand – see [18] for a brief overview.

Regarding the third point, there has been some ongoing research whose goal is to bring lambda calculus on par with other models of computation, with interesting results like [19] showing that it's not as bad as complexity theorists think, but it's still very far from entering the mainstream consciousness. Another related line of research is implicit complexity theory – an attempt to characterize complexity classes not in terms of resources needed by a machine to compute a function, but in terms of restricted, minimalistic programming languages that can be used to implement functions from these classes. See [20] [21] for an introduction. A third related body of research concerns cost semantics, a technique for specifying the resources needed by a programming language's constructs together with their operational behaviour, so that one no longer needs to consider the model of computation (or rather, the programming language becomes the model of computation). This opens up some interesting directions, like automatic complexity analyses during compilation time. See [22] for arguments why this is a viable alternative to machine-based computation models and [23] for an example paper using this technique.

We will not worry about the concerns raised, because a single formal world is still a huge win in terms of correctness. Regarding point 1, in practice, if Coq accepts a proof, then the proof is correct – chances of running into a critical bug by accident are minimal. We're also not as much interested in running algorithms and analyzing their complexity, so points 2 and 3 don't matter to us at all.[7]

## 1.5 Type theory (as a model of computation)

In this section we describe the basic principles on which Coq is founded. Coq is based on a formal system called Calculus of Constructions (CoC)[24], which later evolved into the Calculus of Inductive Constructions (CIC) [25] and finally into Predicative Calculus of Cumulative Inductive Constructions (pCuIC) [26].

To avoid drowning in minute details, we instead describe Martin-Löf Type Theory (MLTT) [27] [28], a closely related system which from now on we will call simply "type theory". In the next section, where we introduce Coq, we will point out the relevant differences from type theory as we go. The material presented in this section is standard, but in the light of previous section's closing paragraphs we encourage people familiar with type theory to read it as a description not of a formal system,

[7] This of course does not mean that we will study silly, exponentially slow algorithms...

but of a model of computation.

In type theory, as the name suggests, the basic objects of interest are types. In order to be meaningful, a type must first be formed. For example, we can always form the type $\mathbb{N}$ of natural numbers, but in general this is not the case: sometimes to form a type we must make an assumption or even several assumptions. We keep track of our assumptions using contexts. For example, if in a context $\Gamma$ we can form the types $A$ and $B$, written $\Gamma \vdash A$ and $\Gamma \vdash B$ respectively, then we can form the type of functions from $A$ to $B$, written $\Gamma \vdash A \to B$. The rules that govern type formation are called, unsurprisingly, formation rules.

After we have formed a type, we can manipulate its terms. For example, the type of natural numbers $\mathbb{N}$ has as terms, among others, 4 and $2 + 2$, written $4 : \mathbb{N}$ and $2 + 2 : \mathbb{N}$, respectively. Similarly to the case of type formation, what is a term and what is not depends on the assumptions we have made: $n + m : \mathbb{N}$ only under the assumptions that $n : \mathbb{N}$ and $m : \mathbb{N}$, which we can formally write as $n : \mathbb{N}, m : \mathbb{N} \vdash n + m : \mathbb{N}$. From now on we won't emphasize context-dependence, as everything we do in type theory depends on context.

Rules for creating terms are divided into two genres: introduction rules and elimination rules. Introduction rules for a type tell us how to create new terms of this type. For example, the introduction rules for $\mathbb{N}$ are $\Gamma \vdash 0 : \mathbb{N}$, which says that 0 is a natural number, and $\Gamma, n : \mathbb{N} \vdash \mathtt{succ}(n) : \mathbb{N}$, which says that the successor of $n$ is a natural number given that $n$ is a natural number.

Elimination rules for a type, on the other hand, tell us how to use terms of this type to get terms of other types. For example, the simplest elimination rule for $\mathbb{N}$ says that if $\Gamma \vdash z : X$ and $\Gamma \vdash s : \mathbb{N} \to X$, then $\Gamma, n : \mathbb{N} \vdash \mathtt{rec}_{\mathbb{N}}(z, s, n) : X$. This rule, alternatively called a recursor, allows us to define functions by recursion on natural numbers, but to do proofs by induction we need a stronger elimination rule. We won't state it here to keep things simple.

The next thing we are interested in is convertibility, which formalizes the notion of computation. Intuitively, two terms are convertible if they evaluate to the same result. For example, we would expect that both $2 + 2$ and 4 evaluate to 4, and this is indeed the case, so they are convertible. To state that formally, we write $\Gamma \vdash 2 + 2 \equiv 4 : \mathbb{N}$. Rules that govern the behaviour of convertibility, similarly to rules for term manipulation, come in two genres: computation rules and uniqueness rules.

Computation rules describe what happens when we first apply an introduction rule and then an elimination rule. For example, the first computation rule for $\mathbb{N}$ states that given $\Gamma \vdash z : X$ and $\Gamma \vdash s : \mathbb{N} \to X$, we have $\Gamma \vdash \mathtt{rec}_{\mathbb{N}}(z, s, 0) \equiv z : \mathbb{N}$, and the second rule states that given $\Gamma \vdash z : X$, $\Gamma \vdash s : \mathbb{N} \to X$ and $\Gamma \vdash n : \mathbb{N}$ we have $\Gamma \vdash \mathtt{rec}_{\mathbb{N}}(z, s, \mathtt{succ}(n)) \equiv s(\mathtt{rec}_{\mathbb{N}}(z, s, n)) : \mathbb{N}$.

Uniqueness rules, on the other hand, describe what happens when we first use

an elimination rule and then an introduction rule. For example, the uniqueness rule for $\mathbb{N}$ states that given $\Gamma \vdash n : \mathbb{N}$, we have $\Gamma \vdash \mathtt{rec}_{\mathbb{N}}(0, \mathtt{succ}, n) \equiv n : \mathbb{N}$. Note, however, that this uniqueness rule for natural numbers is rarely included in most presentations of type theory.

The last thing to mention is how exactly contexts work and some of its consequences. First, there is the empty context, usually denoted by leaving empty space to the left of $\vdash$, for example $\vdash \mathbb{N}$ means that we can form the type $\mathbb{N}$ in the empty context. Second, if we have a context $\Gamma$ and in this context we can derive $\Gamma \vdash A$, then we can extend the context $\Gamma$ with a variable of this type, written $\Gamma, x : A$, provided that the name $x$ doesn't already occur in $\Gamma$.

The most important consequence of this is that we need a notion of convertibility for types. Because types are formed in contexts, and contexts can contain assumptions of the form $x : A$, types can depend on terms. For example, for any natural number $n$ we can form the type $\mathtt{Fin}(n)$ that has exactly $n$ elements, formally written $\Gamma, n : \mathbb{N} \vdash \mathtt{Fin}(n)$. This raises the question: is the type $\mathtt{Fin}(2+2)$ the same as the type $\mathtt{Fin}(4)$? The answer is yes – because $2+2$ and $4$ are convertible, these types are also convertible, formally written $\Gamma \vdash \mathtt{Fin}(2+2) \equiv \mathtt{Fin}(4)$. The rules that govern type convertibility aren't very interesting – they say that convertibility is an equivalence relation and that dependent types are convertible whenever the terms they depend on are convertible.

An astute reader has probably noticed that we sneaked into the above description a word that is familiar but was not defined, namely "element". The elements of a type $A$ are its closed terms that are in normal form. A term is closed if its context is empty and it is in normal form if it is the final result of evaluation,[8] i.e. it can't be evaluated any further.

For example, $\vdash 4 : \mathbb{N}$ is a closed term in normal form, $\vdash 2+2 : \mathbb{N}$ is a closed term that is not in normal form and $n : \mathbb{N} \vdash \mathtt{succ}(n) : \mathbb{N}$ is a term in normal form, but it is not closed because it depends on the assumption $n : \mathbb{N}$ (such terms are called open). It turns out that for natural numbers, the elements are precisely $0, \mathtt{succ}(0), \mathtt{succ}(\mathtt{succ}(0)), \mathtt{succ}(\mathtt{succ}(\mathtt{succ}(0))), \ldots$ which we can interpret as $0, 1, 2, 3, \ldots$ respectively. So, the elements of the type of natural numbers are precisely the natural numbers. Who would have guessed!

That's it. The basic idea behind type theory is very simple. Of course the above presentation is far from being complete – to be, we would have to list all the formation, introduction, elimination, computation and uniqueness rules for all types, and also dozens of boring technical rules whose role is to make sure everything works as expected.

The explanation of elements prompts us to change our perspective on type theory from a model of computation back to foundational and pose the following

[8] We leave the idea of evaluation informal and won't describe it in more detail.

question: what is the relation between type theory and set theory? After all, types have elements and sets have elements too. Below, we provide a short comparison.

Set theory has two basic kinds of objects – propositions and sets – living in two separate layers. Propositions live in the logical layer, which consists of first order classical logic, whereas sets live in the set-theoretical layer, which consists of set theory axioms. Type theory, on the other hand, has only one basic kind of objects, namely types, and only one layer – the type layer. Both propositions and sets can in type theory be interpreted using types.

In set theory, the membership predicate $x \in A$ is a proposition which can be proved or disproved. It is decidable internally, because of the law of excluded middle, but not externally, because there is no computer program that can tell us whether $x \in A$. In type theory, $x : A$ is not a proposition, but a judgment, which means one can establish that it holds, but it makes no sense to negate it. It neither makes sense to talk about it's internal decidability. However, it is externally decidable, which means there is a computer program that checks whether it holds.

In set theory, sets are semantic entities that need not, in general, be disjoint, so an element can belong to many sets. In type theory, types are syntactic entities that are always disjoint. Elements can belong to only one type, or, to be more precise, if an element belongs to two types, then these types are convertible.

## 1.6 Way of the Coq

In this section we give a very short Coq tutorial. Before reading it, the unfamiliar reader should install CoqIDE[9] and run the listing below[10] in interactive mode. This experience will greatly enrich the explanation.

```
Require Import List Lia.
Import ListNotations.

Print list.
(*
Inductive list (A : Type) : Type :=
    | nil : list A
    | cons : A -> list A -> list A
*)

Print app.
(*
```

[9] Available from [12].

[10] It can be found in the thesis' repository `https://github.com/wkolowski/RandomCoqCode`in the directory `Thesis/Snippets/`

```
fix app {A : Type} (l1 l2 : list A) {struct l1} : list A :=
match l1 with
  | [] => l2
  | h :: t => h :: app t l2
end
*)

Record rev_spec {A : Type} (f : list A -> list A) : Prop :=
{
    f_app   : forall l1 l2 : list A, f (l1 ++ l2) = f l2 ++ f l1;
    f_singl : forall x : A, f [x] = [x];
}.

Theorem rev_spec_unique :
  forall {A : Type} (f g : list A -> list A),
    rev_spec f -> rev_spec g ->
      forall l : list A, f l = g l.
Proof.
  intros A f g [Hfc Hfs] [Hgc Hgs].
  induction l as [| h t]; cbn.
    specialize (Hfc [] []); specialize (Hgc [] []).
      apply (f_equal (@length _)) in Hfc;
      apply (f_equal (@length _)) in Hgc.
      cbn in *. rewrite app_length in *.
      destruct (f []), (g []).
        reflexivity.
        1-3: cbn in *; lia.
    change (h :: t) with ([h] ++ t).
      rewrite Hfc, Hgc, Hfs, Hgs, IHt. reflexivity.
Qed.
```

We start by importing modules for working with lists – we will need some list functions and notations (Coq allows defining custom notations using a built-in notation mechanism), and also `lia`, a procedure for reasoning in **l**inear **i**nteger **a**rithmetic (hence the name).

The command `Print` is part of the Vernacular – a language of useful commands for looking up definitions and theorems, and other things usually provided at the IDE level. We use it to recall the definition of `list` and `app` (we modified the outputs of these commands to simplify our explanations).

`list` is a parameterized family of inductive types. This means that for each

type `A`, there is a type list `list A`, defined by giving the possible ways to construct its elements. An element of this type can be either `nil`, which represents an empty list, or `cons h t` for some `h : A` and `t : list A`, which represents a list with head `h` and tail `t`. This is very similar to the definition of lists in pseudo-Haskell that we saw in section 2.

`app` is a function that concatenates two lists. It takes as arguments a type `A` and two lists of elements of type `A`, named `l1` and `l2`, and it returns an element of type `list A` as result. The first argument is surrounded by curly braces, which means it is implicit – we won't need to write it, because Coq can infer it on its own. The other arguments are surrounded by parentheses, which means they are explicit – we need to provide them when applying the function.

`app` is defined by recursion, as indicated by the keyword `fix`, and its termination is guaranteed because this recursion is structural, as indicated by the annotation `struct l1`. The definition goes by pattern matching on the argument `l1`. if it is a `nil`, the result is `l2` and if it is `h :: t`, then the result is `h :: app t l2`. Here the double colon `::` is a handy notation for the constructor `cons`.

In section 3 we said that in a formal setting we can prove a specification correct. This is actually the first of our techniques and in the rest of the listing our task will be to demonstrate how it works. The thing we want to specify is a function for reversing lists and the specification, named `rev_spec`, starts on line 20. It takes as arguments a type `A` and a function `f : list A -> list A` and our intention is that `rev_spec f` means "`f` is a function that reverses lists".

The definition says that `rev_spec f` is a proposition[11] and that it is defined to be a record consisting of two fields. The first one is named `f_app` and intuitively means that `f` anticommutes with list concatenation – if we concatenate two lists and then reverse the result using `f`, it's the same as if we first reversed the lists and then concatenated them in the opposite order. The second field is named `f_singl` and means that on lists with only one element `f` acts like an identity function.

Now, let's take a break from Coq and make a short conceptual trip: how can we prove that a specification specifies precisely what we wanted? Of course, in general, we can't, because what we really want lives in the world of ideas, whereas the specification lives in the formal world. But if we are sure that the desired solution will meet the specification, the best sanity check we can perform is proving that the specification determines a unique object. In such a situation if we are dissatisfied with the solution, we are sure that it is not because of some bug or other kind of technical error, but simply because we were unable to express what we wanted.

Back to Coq, the theorem `rev_spec_unique` is the fulfillment of our plan of

[11] This is actually one of the differences between type theory as presented in the last section and Coq. In type theory, we represent propositions using ordinary types. In Coq, only types of sort `Prop` are propositions, whereas types of sort `Type` are not.

"proving the specification correct". Such a theorem in Coq has the same status as an ordinary definition – the statement of the theorem is a type and to prove it we need to construct an element of that type. However, the means of construction are different from those typically used for definitions: we don't provide a proof term explicitly, like the body of `app` between `match` and `end` was provided explicitly, but instead we use Coq's tactic language called Ltac.

In Ltac, we can manipulate tactics, which can be seen as things that encapsulate a particular line of mathematical reasoning and argumentation (or, from the programmer's perspective, as techniques for synthesizing particular kinds of programs). The most basic tactics directly correspond to introduction and elimination rules of type theory as described in the previous section, and to basic features of Coq's term language, like pattern matching. More complex tactics can be created from simpler ones by using features provided by Ltac, like various tactic combinators, searching hypotheses and variables in the proof context (and more advanced features for context management), many flavours of backtracking and the ability to inspect Coq's terms as syntax.

The last of these is something that is not possible using the term language, as it would allow distinguishing between convertible terms and thus lead to contradiction. For example, given a `P : Prop`, we can't use the term language to check whether it's of the form `Q /\ R`, but we can do that using Ltac. We can use this together with other Ltac features to, for example, implement a tactic that can decide whether a proposition is an intuitionistic tautology or not.

Back to the proof of our theorem, the idea is as follows. The proof is by induction on `l`. This splits the proof into two cases: in the first one `l` is `[]` and in the second one `l` is `h :: t` for some head `h` and tail `t`. In the first case, we exploit the fact that `f [] = f [] ++ f []` (which stems from specializing one of the specification's clauses with `[]`) to argue that the length of `f []` must be zero, and thus that `f []` must equal `[]` (and analogously for `g`). In the second case, we make heavy use of equations coming from the specification and the induction hypotheses.

We won't explain how this proof idea is actually carried out, because such an explanation would be far less enlightening than just going over the proof script in CoqIDE. We will only gloss over the meaning of tactics that were used:

- `intros` lets us assume hypotheses and move universally quantified variables from the goal into the context.
- `induction` starts a proof by induction, splitting the goal into as many subgoals as there are cases.
- `cbn` performs computations, like simplifying `2 + 2` to `4`.
- `specialize` instantiates universally quantified hypotheses with particular objects.

- `apply` implements modus ponens, allowing us to transform `P` into `Q` in the context, given a proof of `P -> Q`.
- `rewrite`, given a hypothesis that is an equation, replaces the occurrences of its left-hand side with its right-hand side in the goal or another hypothesis.
- `destruct` implements reasoning by cases.
- `reflexivity` allows us to conclude that `x = x`.
- `lia` is the powerful tactic for dealing with arithmetic mentioned earlier. In our case we use it to prove that from `n + n = n` it follows that `n = 0`.
- `change` allows us to replace a term or its part with a convertible term. For example, having computed `2 + 2` to `4`, we could use `change` to "reverse" this computation, changing `4` back into `2 + 2`.

Our short tutorial on Coq has come to an end. From now on, we assume the reader is familiar with the technical workings of Coq – while discussing further code listings, we will only explain the concepts and techniques.[12] The moral of our example can be summarized as follows.

**Formally verified algorithm concept/technique #1**
Prove that the specification determines a unique object.

## 1.7 An ultra short literature review

In this section we briefly discuss the not so large body of literature relevant to our work. In case the reader still doesn't feel comfortable with Coq after last section's tutorial, we start by listing some standard learning materials on the topic:

- The first volume of Software Foundations [29] is a textbook aimed at teaching Coq fundamentals to students familiar with neither functional programming nor formal proof. It is very beginner-friendly and well-suited for self study, but it doesn't cover all Coq-related topics exhaustively.
- Coq'Art [30] is more of a reference work than a textbook – it tries to be comprehensive, covering topics like proof by reflection which are missing from Software Foundations, but it is also a bit old, having been published in 2004. Watch out for the outdated treatment of coinduction!

[12] When numbering concepts and techniques, we won't distinguish between them, as any concept, to be useful, has to be concretely realized by some technique and behind any technique there is some concept.

- Certified Programming with Dependent Types [31] is a book aimed at more advanced Coq users. It focuses on programming with dependent types and automating proofs with powerful hand-crafted tactics, but also covers topics like general recursion and dealing with axioms.

As we can see, the book-length literature on Coq is quite scarce and sometimes dated. The situation in the field of functional algorithms and data structures is even worse. In fact, there is only one significant book in the whole field, namely Purely Functional Data Structures (PFDS for short) [32], which is an extended version of Chris Okasaki's PhD thesis [33] (see [34] for the story behind the book). This book is basically a turning point which demarcates two periods in the field's history: pre-Okasaki and post-Okasaki. We won't describe these periods and their literature in detail, as a very good summary is available in this StackOverflow thread [35].

Even though the book is great, it won't be of much inspiration for us. The reasons stem directly from the book's main themes. The first one is exploring the various algorithmic uses of lazy evaluation. Even though Coq makes lazy evaluation readily available, it does so using the tactic `cbn` and command `Eval cbn in t`, whereas the term language has barely any control over the evaluation order. This means that large swaths of the book are at odds with Coq.

The second theme in the book, somewhat related to the first, are techniques for analysis of amortized complexity of lazy data structures. Okasaki was actually the first person to realize that in the purely functional world, amortization is possible (before it was though that it is contradicted by persistence) and that it is inseparably tied to lazy evaluation. This aspect of PFSD won't be important for us because, as we have already underlined many times, we are not interested in complexity, but in formal correctness.

The third big theme of the book are numerical representations. It is the idea that functional data structures are analogous to various obscure positional number systems that aren't at all used for other purposes. In this view, inserting an element into a data structure is analogous to incrementing a number, deleting an element is analogous to decrementing a number, merging two data structures is analogous to adding numbers and so on. While it is a very interesting and worthwhile idea, its purpose is coming up with new data structures – something we won't do here.

The other book that is relevant to our work is "Verified Functional Algorithms" [36], which is actually the third volume of the aforementioned Software Foundations series [29]. It is in fact much more relevant than PFDS, because it focuses on proving algorithms correct using Coq, just as we do. In our opinion however, the approach presented in VFA has many shortcomings:

- It presents the discussed algorithms in their final form without discussing how they were designed in the first place (or how they were translated from their

imperative progenitors).

- It is not abstract enough – all too often it considers only the case where the keys/values of a data structure are natural numbers and skips the general case, where they are of any type with decidable equality/partial order etc.
- It does not deal with techniques for defining general recursive functions properly. Even though it presents a method of defining general recursive functions, it doesn't discuss its principles or what to do if it fails. Often it skips the matter completely and just uses the so called "fuel recursion" which is not very hygienic.

Besides these shortcomings, there are also some other differences, which are more cosmetic in nature. First, VFA uses Coq's module system, whereas we will use the typeclass system. Second, even though just like us it is not that much concerned with performance, VFA sees the performance of an algorithm implemented in Coq as a property of the OCaml program extracted[13] from the Coq code, rather than of the Coq program itself, like we do.

## 1.8 Outline of the thesis

Having given a thorough introduction, it should now be clear what we mean by "formally verified functional algorithms and data structures in Coq". In the remaining chapters of the thesis we describe the actual concepts and techniques", using well-known and well-studied examples like quicksort, merge sort or binary heaps. The rest of the thesis is structured as follows:

- In Chapter 2 we look at quicksort inside and out, describing the bulk of our approach.
- In chapter 3, ...
- TODO: put here the remaining chapters after they're written.

[13] Coq has a mechanism called extraction, which is a form of code generation that transforms Coq code into computationally equivalent code in OCaml, Haskell or Scheme.

# Chapter 2

# A quick *tour de* sort

The field of algorithms and data structures can, in principle, be easily separated into two subfields: the algorithms and the data structures. But these two go hand in hand: all algorithms operate on data structures, no matter how primitive, and data structures exist precisely in order for algorithms to operate on them.

We have to start somewhere, however, and so we choose to break this vicious circle at algorithms. In this chapter we will look at the problem of sorting a list – one of the most basic data structures – and at quicksort, one of the fastest and most elegant solutions of this problem.

Quicksort is probably also the most researched algorithm in the whole imperative paradigm, which makes it a perfect candidate to showcase our core concepts and techniques for specifying, implementing and verifying algorithms in the functional paradigm.

- In section 2.1 we look at techniques for verifying a specification's usefulness and for improving bad specifications (sadly, there seems to be no easy way of coming up with a good specification).

- In section 2.2 we show how to implement an abstract template of an algorithm without worrying too much about concrete details or the termination proof. We also present techniques for assessing the quality of such a template.

- In section 2.3 we describe a general method for carrying out termination proofs and review techniques for reasoning about functions defined in this way.

- In section 2.4 show how to formally prove the abstract algorithm template correct in a top-down fashion without writing too many proofs.

- In section 2.5 we go through all the concepts and techniques once more to experience a higher-level view of how they fit together.

## 2.1 Specify the problem

In the previous chapter we have already seen our first concept/technique[1] and it told us to prove that our specification determines a unique object. Fair enough, but to do that we have to have some specification first and therefore it would be nice to have some technique for coming up with good specifications.

This is likely the hardest of all tasks that a functional algorithmist will have to perform in order to get a formally-verified algorithm[2] as it requires a significant amount of mathematical creativity and insight, which can only be acquired with experience. As we will see later in this chapter, the techniques presented there will strip the tasks of implementing the algorithm and proving it correct of most of their creativity requirements and make them quite repetitive (maybe even boring), but this is not so for the task of inventing the specification.

Nonetheless, we present a simple heuristic for checking if a specification is good and illustrate it with two failed (and one successful) attempts. We give it the number 0 to mark the fact that in the logical order of things it comes before concept/technique #1 from the previous chapter.

**Formally verified algorithm concept/technique #0**
Find a specification of the problem that is both abstract and easy to use.

### 2.1.1 Not that easy

So, how can we formally define the proposition "the list of natural numbers $l$ is sorted"? One way of doing this is to formalize the intuitive definition which says that a list is sorted if earlier elements are less than (or equal to) later elements.

```
Require Import List.
Import ListNotations.

Fixpoint nth {A : Type} (n : nat) (l : list A) : option A :=
match l with
    | [] => None
    | h :: t =>
        match n with
            | 0 => Some h
```

[1] I don't distinguish between concepts and techniques. On the one hand, every concept, to be useful, has to be operationalized, which basically means turning it into some useful technique. On the other hand, every technique is based on some underlying idea, which means it is an operationalization of some concept.

[2] Besides inventing the algorithm itself, of course, but coming up with good algorithms is not a topic of this thesis.

```
            | S n' => nth n' t
        end
end.

Definition sorted {A : Type} (l : list nat) : Prop :=
  forall i j : nat, i <= j ->
    forall n m : nat,
      nth i l = Some n -> nth j l = Some m -> n <= m.
```

This is exactly what the above definition says. We start by importing the required modules and then define an auxiliary function named `nth`, which returns the `n`-th element of the list `l` (wrapped in the `option` constructor `Some`) or `None` if the list is not long enough to have an `n`-th element. Then comes the definition of sortedness itself: the list `l` being sorted means that for any two indices `i` and `j` such that `i` comes before `j`, when the `i`-th element of `l` is `n` and the `j`-th element of `l` is `m`, then `n` is less than or equal to `m`.

This definition looks quite correct (and it is), but it won't serve us well. This is because it defies the second part of our advice #0 – it is not easy to use. The reason why will be rather elusive for people who are not Coq experts[3], but I will nonetheless try to give an appealing argument.

The problem with this definition is that it is a *patchwork* definition – it is made up of various ingredients that don't fit together perfectly. The first ingredient is universal quantification over two natural numbers. The quantifier itself is fine, as are the numbers, but problems start with the hypothesis `n <= m`.

Upon closer inspection, there's a mismatch between the inductive definition of natural numbers (using zero and successor) and the inductive definition of the relation `<=` (which is based on reflexivity and the fact that `n <= m` implies `n <= S m`). This is not a big problem all by itself, just one source of potential clumsiness.

That's not the only source of clumsiness, however. After another quantifier and two numbers, we find two equations. General equations between variables (like `n = m`) are not problematic at all, but equations with more specific terms on one or both sides are a bit clumsy, often requiring additional bookkeeping effort to use properly.

The next source of clumsiness is the fact that on the lefts of these equations we used the auxiliary function `nth`, which is defined by first matching the list argument and then matching the number argument. This is a bit clumsy because even if we reason by cases on the number, the definition of `nth` won't reduce unless we also

[3]Knowing at first sight that a definition like this one will later turn out to be quite clumsy requires precisely the kind of creativity and insight mentioned before that can't be packed into a repetitive, mindless technique.

reason by cases on the list.

All of these sources of clumsiness would add up if we attempted to prove that a list sorting function is correct. If we attempted an induction on the indices, dealing with the proof proof of `i <= j` would be a bit annoying. If we went for a induction on the proof of `i <= j`, then we would have to reason by cases on the list just to be able to use the equations. If we went for an induction on the list, we still have a lot of work with the indices.

### 2.1.2 Improving patchwork definitions

The problems are not insurmountable – with some additional bookkeeping and annoyance, they can be overcome. We could even try to dodge the problem by changing the definitions of `<=` and `nth` so that they corresponded more closely to the definition of natural numbers: `<=` can be defined using the fact that `0 <= m` and `n <= m -> S n <= S m`, whereas `nth` can be defined by recursion first on the number and only then on the list.

However, that would still yield us a patchwork definition and patchwork definitions have this worrisome property that the more you use them, the more you hate them. Therefore, let's forgo this definition of sortedness and try to find a better one.

How do we go about that? As has been said previously, in general it requires some creativity and insight, but it turns out sometimes there's a little shortcut: if we have a patchwork definition, made of parts that don't fit perfectly, we can try to restate it as an inductive definition, yielding a potential improvement.[4]

**Formally verified algorithm concept/technique #0.5**
Sometimes a patchwork definition can be improved by restating it as an inductive definition.

Let's try to restate the intuitive definition of sortedness, which says that elements with smaller indices are less than or equal to elements with bigger indices. If we (conceptually) unfold this statement a bit, we get this: the element with the smallest index is the least element, the element with the second smallest index is the second least element, etc. Unfolding a bit more: the first element is less than everything that follows it, the second element is less than everything that follows it, etc.

```
Inductive LessThanAll (n : nat) : list nat -> Prop :=
    | LessThanAll_nil : LessThanAll n []
```

[4] If the patchwork definition consists of universal quantifiers and implications, then restating it as an inductive definition is a special case of a more general transformation known as defunctionalization. See [37] for a programmer-friendly introduction and [38] for a more academic work.

```
    | LessThanAll_cons :
        forall (h : nat) (t : list nat),
          n <= h -> LessThanAll n t -> LessThanAll n (h :: t).

Inductive Sorted : list nat -> Prop :=
    | Sorted_nil : Sorted []
    | Sorted_cons :
        forall (h : nat) (t : list nat),
          LessThanAll h t -> Sorted t -> Sorted (h :: t).
```

We can easily formalize the above reformulated definition. First we define a helper relation `LessThanAll`, so that `LessThanAll n l` means that the number `n` is less than or equal to all numbers in the list `l`. Then comes the main definition: an empty list is sorted[5], and a list `h :: t` is sorted as soon as `h` is less than all elements of `t` and `t` itself is also sorted.

```
Inductive Sorted : list nat -> Prop :=
    | Sorted_nil : Sorted []
    | Sorted_singl : forall n : nat, Sorted [n]
    | Sorted_cons :
        forall (n m : nat) (l : list nat),
          n <= m -> Sorted (m :: l) -> Sorted (n :: m :: l).
```

This definition can be simplified even more if we notice that it is not necessary to compare an element with all elements coming after it, but only with the one that immediately follows it. This simplified definition reads like this: empty and singleton lists are sorted and to prove that a list with at least two elements is sorted, we need to show that the first element is less than the second element and that the tail of this list is also sorted.

Our final definition of sortedness is, in terms of usability, much better than the one we started with: it is a simple, three-clause inductive definition. It's easy to prove theorems about sorted list by induction on the proof of sortedness. It's also much easier to prove that a list is sorted, because we no longer have to deal with indices, proofs on index inequality, equations, auxiliary functions defined by recursion and so on. All of these things are now baked into the definition and thus all the parts fit together perfectly.

[5] We must remember this base case, even though it's quite implicit in the patchwork definition.

### 2.1.3 Not that abstract

Does this mean we now have a good specification? Not quite, because our improved definition still defies the first prescription of concept/technique #0: it is not abstract enough. This boils down to the fact that we defined sortedness for a list of natural numbers instead of a list of elements of a general type `A`.

There are two main reasons for seeking abstraction when solving a problem. The first one is pragmatic: a more abstract solution is more widely applicable. The second one is harder to pin down, but is sometimes known as *Inventor's paradox*. George Pólya describes it this way:

> The more ambitious plan may have more chances of success (...) provided it is not based on a mere pretension but on some vision of the things beyond those immediately present.[39]

Our initial choice has far-reaching implications, because the type of natural numbers is an object very rich in structure. Naturals form a semiring under addition and multiplication and a totally ordered set under the standard ordering, they support proof by induction, are countably infinite, embed in the reals, etc. On the other hand, a general type `A` doesn't have any of this structure – we know almost nothing about it.

At first glance this may look like a disadvantage: not having addition or countability seems to constrain us, because we can't do as much with `A` as with the naturals. At a second glance, however, we may notice that sometimes being constrained is advantageous, as we have less ways of being wrong.[6] After all, most of the rich structure of natural numbers isn't very relevant to the problem of sorting.

How do we put this into practice? It's quite simple: in our definition, we replace `nat` with `A` and the order `<=` with a general relation `R`. Both `A` and `R` become parameters.

```
Inductive Sorted {A : Type} (R : A -> A -> Prop) : list A -> Prop :=
    | Sorted_nil : Sorted R []
    | Sorted_singl : forall x : A, Sorted R [x]
    | Sorted_cons :
        forall (x y : A) (l : list A),
          R x y -> Sorted R (y :: l) -> Sorted R (x :: y :: l).
```

[6] At a third glance, this should be obvious to every proponent of strong, static typing. After all, types are a mechanism that constrains the programmer by ruling out suspicious (i.e. not well-typed) programs.

Note that we don't assume anything about the relation – `R` can be any relation whatsoever, not necessarily a total order relation. This is the first glimpse of a concept we will meet later: don't assume what is not needed. To state the definition we don't need `R` to have any properties, so we don't assume any.

### 2.1.4 Just about right... or is it? Staying on the right track

So, we're done, right? Not quite. If you're an attentive reader, you should remember that we set out to give a specification of a sorting function, whereas it's easy to notice that so far we have only given a definition of what it means for a list to be sorted.

That's a significant mismatch, because a sorting function is not one that just returns a sorted list as a result. Consider the function that always returns an empty list. It certainly returns a list that is sorted, but it isn't a sorting function – the process of sorting shouldn't remove (or add) any elements, only shuffle the existing ones. What we are missing is a condition saying that the output of the function is a permutation of the input.

This is a minor but important lesson for us: while devising a specification, we must not get drown in minute details of improving patchwork definitions with inductive types, but we have to always bear in mind our final goal.

Having said that, how do we define what it means for two lists to be permutations of each other? The definition that can be found in Coq's standard library looks as follows.

```
Inductive Permutation {A : Type} : list A -> list A -> Prop :=
  | perm_nil1 :
      Permutation [] []
  | perm_skip :
      forall (x : A) (l1 l2 : list A),
        Permutation l1 l2 -> Permutation (x :: l1) (x :: l2)
  | perm_swap :
      forall (x y : A) (l : list A),
        Permutation (y :: x :: l) (x :: y :: l)
  | perm_trans :
      forall l1 l2 l3 : list A,
        Permutation l1 l2 -> Permutation l2 l3 -> Permutation l1 l3.
```

After all the praise inductive definitions got up to now, we might be tempted to declare victory and go on to implement the algorithm already, but that would be a bit hasty: this definition fails the "easy to use" criterion. This is even harder to see in advance than the clumsiness of our initial definition of sortedness – most

people would probably discover it as an empirical fact as soon as they tried to prove anything nontrivial about quicksort. The reason is that quicksort moves list elements around in very different ways than allowed by the definition of `Permutation`. This is yet another reminder that finding good specifications is more an art than a science.

How to find a better definition of `Permutation`? Let's try the following advice.

**Formally verified algorithm concept/technique #0.75**
Sometimes we can find an alternative specification by changing the focus from one aspect to another.

This one is admittedly quite murky. What is meant by "aspect" here? The best way to understand it is to immediately apply it to our case. In the inductive definition of `Permutation`, the main idea is that of rearranging elements in a list. What this concept/technique is trying to tell us is to define `Permutation` using some other crucial idea that completely characterizes what it means to be a permutation.

```
Fixpoint count {A : Type} (p : A -> bool) (l : list A) : nat :=
match l with
    | [] => 0
    | h :: t => (if p h then 1 else 0) + count p t
end.

Definition Permutation {A : Type} (l1 l2 : list A) : Prop :=
  forall p : A -> bool, count p l1 = count p l2.
```

Such an idea is *counting decidable properties*: two lists are permutations of one another just when, for all decidable properties `p`, they have the same number of elements satisfying `p`.[7] This definition may seem surprising at first, but it can be derived from the one we started with: instead of rearranging elements in a list, we can just as well count the number of occurrences of every element. If we get the same result for all `x : A`, then the two lists are permutations – we don't need to worry about moving the elements around.

The definition we get as a result is not abstract enough, because not all types support decidable equality, so the the last step is to generalize from counting occurrences of elements to counting elements satisfying a general decidable property. This is what was meant by "changing the focus from one aspect to another".

[7] The proof of equivalence of these two definitions of `Permutation` can be found in the thesis' repository. I won't give a precise location here, because it's likely to be out of date soon.

```
Class Sort
  {A : Type} (R : A -> A -> Prop) (f : list A -> list A) : Prop :=
{
    isSorted : forall l : list A, Sorted R (f l);
    isPermutation : forall l : list A, Permutation l (f l)
}.
```

The final specification of a sorting function that we have arrived at through our considerations is shown in the listing. A function `f` is a sorting function (according to some relation `R`, of which we think as the order) if the output is a sorted permutation of the input. We pack these notions into a class named `Sort` using a combination of parameters and fields - a distinction we will revisit later.

To sum up our struggles, we once more stress the fact that inventing specifications is an art that one gets better at with experience. We learned a useful criterion for telling when a specification is bad, but to apply it we must first have something to apply it to. We also saw two hints that can help us improve a bad specification, but they don't always work: sometimes by "inductivizing" a definition we can gain a lot, but sometimes inductive definitions are a path to oblivion and more patchwork-y definitions reign supreme.

## 2.2 Abstract the algorithm

Having found a good specification of the problem, the next step in the process is to implement its solution.[8] The usual high-level idea behind quicksort (agnostic as to whether the thing being sorted is a list or an array) is usually presented somewhat like this:

- If the list/array is empty, we're done.
- Otherwise, choose a pivot, which can be any element from the list/array.
- Partition the list/array (with pivot removed) into two lists/arrays, one of which contains elements less than the pivot, while the other contains elements greater than or equal to the pivot.
- Sort both lists/arrays recursively and put them together with pivot in the middle.

[8] If the problem were novel, the next step would of course be coming up with a solution, but because we're dealing with a well-researched problem with widely known solutions, we omit this step. Another thing is that, just like for specifications, there aren't very many bulletproof techniques for coming up with good algorithms – only heuristics.

When you first saw the famous two-line implementation of quicksort in Haskell, you likely realized that the above idea is pretty simple and easy to implement correctly (if you have never seen it, take a look at listing 3). But whenever you see the imperative implementation (a typical example of which is shown in listing 4), you probably aren't very convinced of this fact.[9]

The main difference between these two implementations is a discrepancy in their levels of abstraction. The first[10] and second[11] bullet points are realized at roughly the same level. The crux of the matter is the partitioning. In the Haskell version, it is expressed as `filter (< h) t` and `filter (>= h) t`, which can be directly read as "all elements of `t` which are less than `h`" and "all elements of `t` which are greater than or equal to `h`", respectively. In the C version, things are much more non-obvious, but if we squint hard enough, we can see that the first `while` loop says "let `a[l]` be the first element on the left of the pivot that is greater than the pivot", the second one says "let `a[h]` be the first element on the right (counting from the end) that is less than the pivot" and the `if` clause means "swap `a[l]` and `a[h]`", and the `do...while` loop carries this on until the array is fully partitioned.

```
qs :: Ord a => [a] -> [a]
qs [] = []
qs (h:t) = qs (filter (< h) t) ++ [h] ++ qs (filter (>= h) t)
```

Listing 3: The famous two-line (three if we count the type signature) quicksort implementation for lists, written in Haskell.

[9] If you don't believe me, try figuring out whether the implementation from listing 4 is correct – I didn't check!

[10] `qs [] = []` in Haskell, `if (lo < hi)` in C

[11] `int p = a[hi];` in C. In Haskell, the pivot is implicitly chosen to be `h`.

```c
// To sort array a[] of size n: qsort(a, 0, n - 1)
void qsort(int a[], int lo, int hi)
{
    if (lo < hi)
    {
        int l = lo;
        int h = hi;
        int p = a[hi];

        do
        {
            while (l < h && a[l] <= p)
                l = l + 1;
            while (h > l && a[h] >= p)
                h = h - 1;

            if (l < h)
            {
                int t = a[l];
                a[l] = a[h];
                a[h] = t;
            }
        }
        while (l < h);

        a[hi] = a[l];
        a[l] = p;

        qsort(a, lo, l - 1);
        qsort(a, l + 1, hi);
    }
}
```

Listing 4: A typical quicksort implementation for sorting arrays, wrriten in C. This one is adapted from `https://wiki.haskell.org/Introduction`

We have to admit that the presentation is a bit skewed in favor of Haskell, because the definition of `filter` is omitted. However, `filter` is another two-liner, so even if it was included, the Haskell way of partitioning is still much more abstract than the C way. This is also mitigated by the fact that the second part of the last bullet point, "putting the recursively sorted parts together" is skewed in favor of C, because it's already included in the partitioning, whereas in Haskell, if we wanted to be fully explicit, we would have to inline the definition of `++`, another two-liner.

The above details aren't that important, however. What matters is the conclusion we wanted to illustrate: the Haskell version looks much simpler (and it's way easier to judge its correctness) because it's much more abstract, in some sense. The C version, being much more concrete, isn't as friendly-looking.[12]

Our formally correct implementation in Coq will necessarily look much scarier and be several times longer than the typical imperative implementation (because of all the correctness proofs being much longer than the algorithm itself), but will in the end be just as simple and beautiful as the Haskell implementation. The key to achieving this is a technique derived from the above observations on abstractness and concreteness.

**Formally verified algorithm concept/technique #2**

Implement an abstract template that captures the idea of the algorithm. You can ignore the matter of termination at first.

In our case, the "idea of the algorithm" corresponds to the four-point description of quicksort given at the beginning of the current section, but we will modify it a bit. We change the first bullet point to "if the input list is short, sort it using some ad hoc method" (to allow various kinds of optimizations, like switching to insertion sort for short lists) and in the third bullet point, we will partition the list into three sublists whose elements are less than, equal and greater than the pivot (so that the user can choose whether he wants 2-way or 3-way quicksort).

In the two-line Haskell implementation of quicksort, the head of the list is chosen as pivot and partitioning is done using `filter`. In our case, we will implement an "abstract template" in which pivot choice, partitioning and the ad hoc sorting of short lists, are all input arguments of the algorithm. In this way, the user can craft a version of quicksort tailed precisely to his needs.

The second part of the concept/technique tells us that we can "ignore the matter of termination at first". At first glance this looks rather unrelated to the part about "abstract template", but after closer inspection it's an important special case.

Coq is a total language, which means that all recursive functions necessarily terminate. This raises the problem of termination checking: how does Coq know whether a function we're implementing is terminating? In most cases a simple syntactical check is enough, but in the general case the problem is undecidable and we have to prove termination manually.

Because of this fact, an implementation of a general recursive function is usually weaved of two closely knit parts: the algorithmic part (what to do and how) and the

[12] I don't mean to imply that all imperative implementations of quicksort must be somehow inferior to all functional implementations. I think that given language features such as higher-order functions and interfaces, which are missing from C, the imperative implementation could match the functional one in simplicity and elegance.

logical part (why doing it terminates). These parts are intertwined: the algorithmic part is littered with proofs and the termination proof of course strongly depends on the algorithm's properties.

In the next section we will see a technique that allows us to deal with these two parts separately and a Coq language feature that can automate this technique, but at present, in accordance with concept/technique #2, we will ignore the matter of termination. A nice way to do this, recently introduced into Coq, is the command `Unset Guard Checking`, which can temporarily turn the termination checker off.

```
Class QSArgs (A : Type) : Type :=
{
    short : list A -> bool;
    adhoc : list A -> list A;
    choosePivot : list A -> A * list A;
    partition : A -> list A -> list A * list A * list A;
}.

Unset Guard Checking.
Fixpoint qs
  {A : Type} (args : QSArgs A) (l : list A) {struct l} : list A :=
    if short l
    then adhoc l
    else
      let '(pivot, rest) := choosePivot l in
      let '(lt, eq, gt)  := partition pivot rest in
        qs args lt ++ pivot :: eq ++ qs args gt.
Set Guard Checking.
```

Here's our first attempt at the abstract template. We begin by defining a class[13] whose fields are the user-provided subroutines that we will use in our template. Then comes the template itself. It is a direct translation of the (modified) four-point description of quicksort from the beginning of this section: if the list is short, sort it using some ad hoc sorting procedure; otherwise choose a pivot, partition the rest of the list into three parts, sort the parts containing smaller and greater elements recursively and join all the parts together with the pivot in the middle.

Note that the definition is placed between commands `Unset Guard Checking` and `Set Guard Checking` in order to temporarily disable the termination checker. Without these, Coq would reject the definition: it is declared to be structurally

[13] Coq's classes are like a cross of Haskell's typeclasses and records, but much more powerful because of dependent types – the types of later fields in a class can depend on the values of earlier fields.

recursive (by the annotation {`struct l`}), but there is nothing to guarantee that recursive calls are made on structural subterms of `l` (indeed, in almost all cases either `lt` or `gt` won't be a subterm of `l`).

### 2.2.1 Foreseeing the user experience

Are we done with the template? Not quite – how do we know that our template makes sense? There's the obvious criterion of conforming with the informal description of the algorithm, but it's quite vague and thus fulfilling it is a rather weak guarantee of success – it's too easy to sneak in errors or inconveniences that will later turn out to be problematic.

A better criterion is user experience. It's not the first thing that comes to mind when thinking about either algorithms or anything formally verified, but nonetheless it's important to get it right. What does user experience mean in our case? Who are the users and what will they want to do with our algorithm?

In typical algorithmic settings the answer is not very enlightening: the user is anybody who runs our algorithm and user experience means that the algorithm is fast (or at least fast enough for his purposes; correctness is taken for granted). In our case things are a bit different, however: even though correctness is even more taken for granted, execution speed will never be satisfying[14]. What matters to us is that the template is very abstract and, unless provided with some defaults, the user will have to fill it out in order to get a concrete algorithm running. Thence comes a better advice for ensuring quality of our abstract template:

**Formally verified algorithm concept/technique #2.5**
Instantiate your template in the most naive way. See what happens and use that observation to improve the template. Provide a default concrete algorithm so that the user doesn't need to fill the template himself.

```
Instance QS_nat : QSArgs nat :=
{
    short l :=
      match l with
          | [] => true
          | _ => false
      end;
    adhoc _ := [];
    choosePivot l :=
```

[14] Coq users make various jokes along the lines of "I don't run my programs, I only prove them correct.". It is possible to extract Coq programs into Haskell, OCaml or Scheme and get performance typical for these languages, but we won't be interested in performance of the extracted code.

```
      match l with
          | [] => (42, []) (* Wut? *)
          | h :: t => (h, t)
      end;
    partition p l :=
      (filter (fun x => leb x p) l,
       [],
       filter (fun x => negb (leb x p)) l)
}.

Compute qs QS_nat [5; 4; 3; 2; 1; 0].
(* ===> = [0; 1; 2; 3; 4; 5]
        : list nat *)
```

Here's what happens when we try this advice for our quicksort template. Let's say a typical user, after stumbling upon our template, will want to build the simplest quicksort variant to sort some natural numbers, and so we define: a list is short when it's empty; we can sort a short (i.e. empty) list ad hoc by returning an empty list; and we can partition by filtering twice, just like in the Haskell version.

But how do we choose the pivot? When the list has head and tail, we choose the head to be the pivot, but in case the list is empty we run into a problem, because the choice is not obvious. We actually chose the number `42`, as shown in the listing, but this choice is arbitrary, as this number does not appear in the list (since it's empty).

When we run this particular version of quicksort on a reverse-sorted example list, it gives the correct answer, but this shouldn't deceive us. If we wanted to be a bit more general and define a similar, but more polymorphic version of quicksort (that works, let's say, on any type with a decidable linear order), we couldn't, because we can't conjure an element of a general type `A` out of thin air. Our template is, therefore, seriously flawed, as any user who attempted to do just that would find the task impossible.

```
Class QSArgs (A : Type) : Type :=
{
    short : list A -> option (A * list A);
    adhoc : list A -> list A;
    choosePivot : A -> list A -> A * list A;
    partition : A -> list A -> list A * list A * list A;
}.
```

```
Unset Guard Checking.
Fixpoint qs
  {A : Type} (args : QSArgs A) (l : list A) {struct l} : list A :=
    match short l with
        | None => adhoc l
        | Some (h, t) =>
            let '(pivot, rest) := choosePivot h t in
            let '(lt, eq, gt)  := partition pivot rest in
              qs args lt ++ pivot :: eq ++ qs args gt
    end.
Set Guard Checking.

Instance QS_nat : QSArgs nat :=
{
    short l :=
      match l with
          | [] => None
          | h :: t => Some (h, t)
      end;
    adhoc _ := [];
    choosePivot h t := (h, t);
    partition p l :=
      (filter (fun x => leb x p) l,
       [],
       filter (fun x => negb (leb x p)) l)
}.

Compute qs QS_nat [5; 4; 3; 2; 1; 0].
(* ===> = [0; 1; 2; 3; 4; 5]
        : list nat *)
```

Improving our flawed template turns out to be pretty easy. We can change the type of `choosePivot` to `A -> list A -> A * list A`, whose domain we can interpret as guaranteeing that the input list is not empty. But what if the input list *is* empty? This shouldn't happen: we call `choosePivot` only in case the input list is not short and the empty list, being the shortest of all lists, certainly should be considered short. We can therefore modify the type of `short` to `list A -> option (A * list A)`, with the possible results being either `None` (which means that the list is short) or `Some (h, t)` (which means it is not short and guarantees that it is not empty).

With these improved types our template doesn't change a lot, but the user

experience becomes much better: defining the naivest concrete quicksort on natural numbers is easier than before, because we got rid of the problematic empty list case, which also enables us to easily generalize to a more polymorphic version of basic quicksort.

### 2.2.2 Boolean blindness vs evidence-based programming

The new template, thoroughly tested for user experience, will soon turn out to be good enough. But user experience is not the only way of checking whether the template makes sense. In our case, we could have arrived at the same conclusions from a completely different angle, which I will call *evidence-based programming*.[15] It roughly corresponds to the part of our abstract about "representing flow of information in a proof using types", but we will consider the flow of information in a program, not a proof.

Before we see how to use this new concept to improve our original template, we will first have to meet the problem it solves, called *boolean blindness*, a dangerous disease that often afflicts programmers trasitioning from imperative to functional. We will also develop a conceptual framework which allows to understand these two in a deeper sense and link them with other related things, like the *small-scale reflection* proof methodology or *intrinsic and extrinsic typing* in lambda calculus.

"Boolean blindness"[16] is an informal term used in the functional programming blogosphere to describe a few interrelated programming misconceptions and antipatterns. Robert Harper used it on his blog [41] to name the erroneous view, held by some imperative programmers and classical mathematicians, that booleans and propositions are the same thing.[17] However, it is most often applied in quite different situations (see [43] for a more software-engineering-oriented presentation and [44] for a perspective going beyond the booleans).

One of these is a bad programming practice, in which any concept that can take one of two values is represented using the boolean type. This decreases code readability, because in a call like `f true` it's not obvious what the `true` stands for, and one has to consult documentation or the function's definition to learn that the first argument of `f` is named, for example, `isAdmin`, which clarifies the meaning of the call `f true`.

Another consequence stemming from this practice is lowered code extensibility: when the business logic changes (e.g. users can now have one of three statuses: "admin", "premium", "regular"; instead of the previous two: "admin" and "non-

[15] This is a term I coined for the purpose of this thesis.

[16] The term was coined by Dan Licata in his lecture notes for an introductory functional programming class [40].

[17] Note that it is only wrong to think that they are *a priori* the same. If we assume classical logic and the Univalence Axiom, it is indeed the case that `bool = Prop`. For details see exercise 3.9 in [42].

admin"), it requires a huge refactoring (which can, if performed badly, lead to the third kind of boolean blindness described below) instead of small changes to a custom data type and a few functions that use it.

"Boolean blindness" is also used to name a horrible antipattern in which boolean tests (i.e. if-then-else) are used instead of pattern matching. An example in Coq would be writing `if isSome x then f (unwrapSome x) else y` instead of `match x with | Some x' => f x' | None => y end`, where `unwrapSome : forall A : Type, option A -> A`. If Coq accepted such code, we could easily derive a contradiction, because `unwrapSome False None` is a proof of `False`. Fortunately it is not possible to write such code in Coq, because Coq won't allow us to implement `unwrapSome`.

A less ominous example would be writing `if isZero n then x else f (pred n)` instead of `match n with | 0 => x | S n' => f n' end`, where `pred : nat -> nat` is the predecessor function on naturals. This example does not lead to contradiction, but decreases performance: the former expression contains two implicit matches on `n` (one in the definition of `isZero`, the other in the definition of `pred`), whereas the latter matches `n` only once and explicitly.

The clou of this last kind of boolean blindness is a lack of explicitly represented evidence, stemming from the use of boolean tests instead of pattern matching, which distorts, or sometimes even blocks, the flow of information in the program.

In the example, in the else branch we know that `n` is not zero, but the evidence for that fact, which is the predecessor of `n`, is missing. Because of this the flow of information is blocked, i.e. we can't explicitly pass the predecessor to `f`. This forces us to call `pred`, which recovers the predecessor and thus restores the flow of information. When seen from this angle, using if-then-else instead of pattern matching is not only very inefficient, but also extremely silly.

In Coq the difference between `bool` and `Prop` is very clear and hard to miss, so the first kind of boolean blindness is unlikely to afflict Coq users. Because not much ordinary software development is carried out in Coq, the second kind is also unlikely. It is therefore the third kind that can most easily sneak into our Coq code and thankfully only in the milder form (the `isZero` and `pred` example), because Coq shields us from the worst mistakes (the `unwrapSome` example).

This is what has indeed happened to our initial template. We declared the return type of `short` to be `bool` and then branched on it in the definition of `qs`. This causes the information on shortness to not be present in the else branch, so that the only sensible domain for `choosePivot` is `list A`, which leads to problems with filling the template that we have already seen.

How do we avoid making such mistakes in the future? To learn that, we have to look at the problem from a more general perspective first. One such perspective, which also encompasses many other phenomena, stems from making a distinction

between implicit and explicit information.[18] A piece of information is explicit when it is backed by evidence[19] that and implicit otherwise. This distinction can be likened to that between language and metalanguage or theory and metatheory: explicit information is information internal to the system (in our case, available inside Coq), whereas implicit information is information external to the system (in our case, available to us, but not necessarily to Coq).

An example: the lists `[1; 2; 3]` and `[2; 3; 1]` are obviously permutations of each other, but given only this, the information stays implicit. If we also have a proof of `Permutation [1; 2; 3] [2; 3; 1]`, the information becomes explicit, with the proof being our evidence. Another example: if we pattern match on a list `l`, then in the `nil` branch we explicitly know that `l` is empty and the evidence is the fact that `l` and `nil` are convertible. In the `cons` case, we explicitly know that `l` is not empty and the evidence is the fact that `l` is convertible with `h :: t` where `h` is its head and `t` its tail.

Boolean blindness (of the third kind) can be summarized by saying that it occurs when not enough explicit information is present to guarantee a smooth information flow in the program, but we can apply the distinction between explicit and implicit information to understand many more phenomena.

For example, there are two versions of simply typed lambda calculus: extrinsically typed and intrinsically typed [45]. In the intrinsic variant all terms are well-typed (so, for example, $\lambda x.xx$ is not a term). In the extrinsic variant terms need only conform to the grammar and types are assigned separately, so $\lambda x.xx$ is a term, but it is not well-typed. It is easy to see that the difference is all about typing information: in the intrinsic variant it is explicit (present in the term) and in the extrinsic variant it is implicit (not present in the term, but outside it, in a separate typing relation). Other conclusions follow: in the extrinsic variant type inference is a method of reconstructing explicit evidence for the implicit typing information present in terms.

Lambda calculus is not the only place where a distinction between intrinsic and extrinsic typing information makes sense – the same holds for almost any data structure. Consider binary search trees, for example. We can define the underlying type of trees in two ways: in the first case all trees are binary search trees (intrinsic typing, explicit information); in the second case, the trees are just ordinary binary trees and the binary search trees are only those that satisfy an extrinsic specification (extrinsic "typing", implicit information).

From this perspective we can also understand *small-scale reflection*[20], a general-

[18] The terms and the whole perspective are mine. I don't know if it appears anywhere in the literature.

[19] For the purpose of the current paragraph, "evidence" is basically synonymous with judgments of type theory, as presented in section 1.5

[20] See [46] for an implementation of a proof language for Coq which is based on this methodology

purpose proof engineering methodology most famously applied to the first fully formal proof of the four colour theorem [49] [50]. At its heart lie conversions between explicit and implicit information, which are performed systematically, depending on which one is more convenient in a given situation, in order to maximize ease of proving theorems, speed of execution, memory usage or some other goal.

Let's say we have a predicate `Even : nat -> Prop` which represents the property of a natural number being even. It explicitly contains all the information on evenness, but it also takes a lot of memory and computing with it is rather slow. Small-scale reflection is a way of improving this situation by defining a function `even : nat -> bool`, which "reflects" the predicate `Even` – `Even n` holds if and only if `even n = true`. Using `even` we can compute much faster and the proofs of `even n = true` occupy a negligible amount of memory. When we need the explicit information about `n` being even, we can recover it using `even`'s specification and use `Even n` as if we had it from the very beginning.

Explicit and implicit information are not absolute concepts. Between these two extremes there are various degrees and shades of mixedness, with some (pieces of) information being explicit and other implicit. We can use these subtle shades to clearly see the, otherwise murky, purpose of the so-called "hybrid types" in Coq. An example is the type `sumor A P` (with `A : Type` and `P : Prop`), whose elements can be: either of the form `inleft a` for `a : A`, which we can interpret as successful computation results that don't carry any explicit information; or of the form `inright p` for `p : P`, which we can interpret as failures carrying explicit information – a proof of a proposition which states what went wrong.

Having come so far, we can now explain what evidence-based programming is: it is programming that incorporates considerations about information flow in the program into the design process. We can apply it to our algorithmic template as follows:

**Formally verified algorithm concept/technique #2.75**
Make sure that types in your abstract template contain enough evidence, so that information flow in the algorithm is smooth, but also make sure that it's not cluttered with unnecessary evidence.

### 2.2.3 Remarks on (un)bundling and (lack of) sharing

We could have defined our template in a slightly different manner, namely with a more bundled class for holding the arguments. By "bundling", in the context of Coq, we will mean realizing a component of a record or a class as a field. "Unbundling", on the other hand, is realizing a component of a class or record as a parameter.

and [47] for its documentation. Also see [48], a book about a library of formalized mathematics done in Coq which uses the small scale reflection methodology.

```
Class QSArgs : Type :=
{
    A : Type;
    short : list A -> option (A * list A);
    adhoc : list A -> list A;
    choosePivot : A -> list A -> A * list A;
    partition : A -> list A -> list A * list A * list A;
}.

Coercion A : QSArgs >-> Sortclass.

Unset Guard Checking.
Fixpoint qs (A : QSArgs) (l : list A) {struct l} : list A :=
match short l with
    | None => adhoc l
    | Some (h, t) =>
        let '(pivot, rest) := choosePivot h t in
        let '(lt, eq, gt)  := partition pivot rest in
          qs A lt ++ pivot :: eq ++ qs A gt
end.
Set Guard Checking.
```

Our first definition of `QSArgs` was unbundled with respect to `A : Type`, because it was a parameter. The listing above shows the alternative realization as a fully bundled class in which `A` is a field instead. We declare `A` as a coercion from `QSArgs` to `Sortclass`, so that we can use instances of `QSArgs` as if they were types. This is quite handy in the new definition of `qs` – instead of two arguments, `A : Type` and `args : QSArgs`, we take only one, `A : QSArgs`, and we can use it to write `list A` just as if it `A` were a type. Besides these cosmetic differences, the new definition of `qs` is the same as before.

The difference between bundled and unbundled versions of a class does not matter in theory – we can derive each one from the other. If we have an unbundled class, we can define a new class whose fields correspond to the parameters of the original class and an instance of this class itself. In the other direction, given a bundled class we can define a new unbundled class, with parameters corresponding to some fields of the original class, whose fields are an instance of the original class itself and equality proofs that constrain the instance's fields to match the parameters of the new class.

In practice, however, the difference is very annoying. Each direction of the conversion is rather clumsy and impractical to perform, with the bundled-to-unbundled

more so than the other way around. It is therefore best to avoid having to perform the conversion and decide ahead of time which version to use. This is mainly a question of sharing, i.e. a situation in which we want two or more classes to have equal parameters/fields. It is easy to set parameters of two classes to be the same (a matter of application, which is well-behaved), but hard to ensure two classes have equal fields (a matter of equality constraints, which are clumsy), so when we need to share, we better use unbundled classes which realize the shared things as parameters.

In case of our template's `QSArgs` the difference doesn't matter very much and we could have used either the bundled or the unbundled version.[21] Why bother then? A situation in which problems could potentially arise is when we are concurrently developing templates of many algorithms that share some components. In such cases it's a good idea to make the shared components into parameters. Otherwise we may stick to the bundled version (which we will do in the rest of this chapter).

**Formally verified algorithm concept/technique #2.8901234567**
If you are developing many algorithm templates at once, make components shared between them into parameters and other components into fields. In case of a single template, use a bundled class by default.

## 2.3 Prove termination

The algorithmic part of our journey has come to and end – we now have a working quicksort template. If we fill it in with concrete details, we get a variant of quicksort that we can run to get a sorted list back... or not. So far, there's no guarantee that the result will be sorted, because the proof of correctness is missing, but an even more pressing concern is that there's no guarantee that we will get any answer whatsoever – if we fill the template with gibberish, the resulting concrete algorithm need not terminate. This is because, in accordance with concept/technique #2, we ignored the matter of termination by turning Coq's termination checker off.

**Formally verified algorithm concept/technique #3**
Implement a better template using a type that represents the shape of the algorithm's recursion. Prove termination using well-founded induction.

Bove-Capretta method, at first crudely using sort Type, then maybe Prop but not necessarily.

Mention functional induction here and how to derive it.

[21] Our original choice, with `A : Type` realized as a parameter, is probably a cultural heritage of Haskell, in which most typeclasses have a single type parameter and any number of non-type fields.

### 2.3.1 A less relevant variant of the inductive domain method

### 2.3.2 Automating the termination proof... a little

Coq's `Function` command.

### 2.3.3 Foreseeing user experience 2

**Formally verified algorithm concept/technique #4**
Make sure the concrete algorithm is runnable without having to prove anything about it.

## 2.4 Verificatio ex nihilo

**Formally verified algorithm concept/technique #5**
Prove your template correct "by fiat": find out what properties and lemmas are required for each step of the proof to go through and assume they are part of your template.

### 2.4.1 Get your hands dirty – concrete proofs

**Formally verified algorithm concept/technique #6**
Prove the concrete algorithm correct by filling out the proofs.

## 2.5 Summary

## 2.6 Exercise

There's a common abstraction behind both quicksort and mergesort: split the list into parts, sort them recursively, and then merge these parts. In quicksort, the real sorting is done during the "split" phase, i.e. during partitioning, and the "merge" phase is trivial list concatenation. In mergesort it's the other way around: the "split" phase is more or less trivial and the sorting is done during the "merge" phase.

Design a sorting algorithm that generalizes both quicksort and mergesort. Using concepts and techniques from this chapter, give a specification of this algorithm, implement it and prove it correct.

# Chapter 3

# Conclusion

Mention thesis' repository: `wkolowski.github.io/RandomCoqCode/Thesis/`

# 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/

[12] https://coq.inria.fr/

[13] Morten Heine Sørensen, Paweł Urzyczyn,
*Lectures on the Curry-Howard Isomorphism*, 2006,
http://disi.unitn.it/~bernardi/RSISE11/Papers/curry-howard.pdf

[14] Bruno Barras, Benjamin Werner, *Coq in Coq*, 1997,
http://www.lix.polytechnique.fr/Labo/Bruno.Barras/publi/coqincoq.pdf

[15] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, Théo Winterhalter,
*Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq*, 2020,
https://www.irif.fr/~sozeau/research/publications/drafts/Coq_Coq_Correct.pdf

[16] Alan Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, 1936
https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

[17] Alonzo Church, *An Unsolvable Problem of Elementary Number Theory*, 1936
https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/church.pdf

[18] Guy Blelloch, Robert Harper,
*λ-Calculus: The Other Turing Machine*, 2015
https://www.cs.cmu.edu/~rwh/papers/lctotm/cs50.pdf

[19] Ugo Dal Lago, Simone Martini,
*The Weak Lambda Calculus as a Reasonable Machine*, 2008
https://www.di.unito.it/~deligu/CDR60_TCS/Martini.pdf

[20] Ugo Dal Lago,
*A Short Introduction to Implicit Computational Complexity*, 2010
http://cs.unibo.it/~dallago/FICQRA/esslli.pdf

[21] Ugo Dal Lago, *Machine-Free Complexity*, 2019
https://caleidoscope.sciencesconf.org/data/DalLago_caleidoscopeslides.pdf

[22] Robert Harper, *Languages and Machines*, 2015
https://existentialtype.wordpress.com/2011/03/16/languages-and-machines/

[23] Norman Danner, Daniel R. Licata, Ramyaa Ramyaa
*Denotational Cost Semantics for Functional Languages with Inductive Types*
https://dlicata.wescreates.wesleyan.edu/pubs/dlr15inductive/dlr15inductive.pdf

[24] Thierry Coquand, Gérard Huet,
*The calculus of constructions*, 1984,
https://www.sciencedirect.com/science/article/pii/0890540188900053

[25] Christine Paulin-Mohring,
*Introduction to the Calculus of Inductive Constructions*, 2015
https://hal.inria.fr/hal-01094195/document

[26] Amin Timany, Matthieu Sozeau,
*Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)*, 2018
https://hal.inria.fr/hal-01615123v2/document

[27] Per Martin-Löf,
*An intuitionistic theory of types*, 1972
https://archive-pml.github.io/martin-lof/pdfs/
An-Intuitionistic-Theory-of-Types-1972.pdf

[28] Per Martin-Löf,
*Intuitionistic Type Theory*, 1984
https://archive-pml.github.io/martin-lof/pdfs/
Bibliopolis-Book-retypeset-1984.pdf

[29] Benjamin C. Pierce, Andrew W. Appel et al.,
*Software Foundations*, 2019,
https://softwarefoundations.cis.upenn.edu/

[30] Yves Bertot and Pierre Castéran,
*Interactive Theorem Proving and Program Development*
*Coq'Art: The Calculus of Inductive Constructions*, 2004,
https://www.labri.fr/perso/casteran/CoqArt/

[31] Adam Chlipala, *Certified Programming with Dependent Types*,
http://adam.chlipala.net/cpdt/

[32] Chris Okasaki, *Purely Functional Data Structures*, 1998

[33] Chris Okasaki, *Purely Functional Data Structures* (PhD thesis), 1996
https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf

[34] Chris Okasaki, *Ten Years of Purely Functional Data Structures*, 2008
https://okasaki.blogspot.com/2008/02/ten-years-of-purely-functional-data.
html

[35] *What's new in purely functional data structures since Okasaki?*,
https://cstheory.stackexchange.com/questions/1539/
whats-new-in-purely-functional-data-structures-since-okasaki

[36] Andrew W. Appel, *Verified Functional Algorithms*, 2018
https://softwarefoundations.cis.upenn.edu/vfa-current/index.html

[37] James Koppel,
*The Best Refactoring You've Never Heard Of*, 2019,
http://www.pathsensitive.com/2019/07/the-best-refactoring-youve-never-heard.html

[38] Olivier Danvy and Lasse R. Nielsen, *Defunctionalization at Work*, 2001,
https://www.brics.dk/RS/01/23/BRICS-RS-01-23.pdf

[39] George Pólya, *How to Solve It*, 1945

[40] https://www.cs.cmu.edu/~15150/previous-semesters/2012-spring/resources/lectures/09.pdf

[41] Robert Harper, *Boolean Blindness*, 2011,
https://existentialtype.wordpress.com/2011/03/15/boolean-blindness/

[42] The Univalent Foundations Program,
*Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013,
https://homotopytypetheory.org/book/

[43] Jeremy Fairbank, *Solving the Boolean Identity Crisis*, 2019,
https://medium.com/swlh/solving-the-boolean-identity-crisis-33eeccde2c96

[44] Harrison Ainsworth, *'Boolean-blindness' is about types*, 2013,
http://www.hxa.name/notes/note-hxa7241-20131124T0927Z.html

[45] John C. Reynolds, *The Meaning of Types: from Intrinsic to Extrinsic Semantics*, 2000,
https://www.cs.cmu.edu/afs/cs/user/jcr/ftp/typemeaning.pdf

[46] Georges Gonthier, Assia Mahboubi, Enrico Tassi
*A Small Scale Reflection Extension for the Coq system*, 2016,
https://hal.inria.fr/inria-00258384v17/document

[47] https://coq.inria.fr/refman/proof-engine/ssreflect-proof-language.html

[48] Assia Mahboubi and Enrico Tassi with contributions by Yves Bertot and Georges Gonthier,
*Mathematical Components*, 2018,
https://math-comp.github.io/mcb/

[49] Georges Gonthier, *A computer-checked proof of the Four Colour Theorem*, 2005
https://www.cl.cam.ac.uk/~lp15/Pages/4colproof.pdf

[50] Georges Gonthier, *Formal Proof – The Four-Color Theorem*, 2008, `http://www.ams.org/notices/200811/tx081101382p.pdf`