Formally verified algorithms in Coq: concepts and techniques

(Formalnie zweryfikowane algorytmy w Coqu: koncepcje i techniki)

Wojciech Kołowski

Praca magisterska

Promotor: dr Małgorzata Biernacka

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Czerwiec 2021

Abstract

We discuss how to specify, implement and verify functional algorithms, 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 on the running example of quicksort. In the end, we arrive at a robust and general method applicable to any functional algorithm.

Omawiamy sposoby specyfikowania, implementowania i weryfikowania funkcyjnych algorytmów, skupiając się raczej na dowodach formalnych niż na asymptotycznej złożoności czy faktycznej wydajności. Prezentujemy koncepcje i techniki, obie często opierające się na jednej kluczowej zasadzie – reifikacji i reprezentacji, za pomocą potężnego systemu typów Coqa, czegoś, co w klasycznym, imperatywnym podejściu jest nieuchwytne, jak przepływ informacji w dowodzie czy kształt rekursji funkcji. Nasze podejście obszernie ilustrujemy na przykładzie quicksorta. Ostatecznie otrzymujemy solidną i ogólną metodę, którą można zastosować do dowolnego algorytmu funkcyjnego.

Contents

Chapter 1

Introduction

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

- In Section 1.1 we look at the social context that gives rise to an interesting misconception about “algorithms”.
- In Section 1.2 we explain “functional” programming by comparing it with imperative programming.
- Section 1.3 is a philosophical interlude in which we give an interpretation of typical algorithmic activities as living in different worlds, with links between worlds being the source of potential errors.
- In Section 1.4 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 1.5 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 1.6 we show how working “in Coq” looks like.
- In Section 1.7 we do an ultra short literature review and discuss how our “concepts and techniques” relate to what can be found there.
- In Section 1.8 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 [2] StackOverflow question. The poor soul asking the 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 of imperative and functional algorithms are mostly a reflection of the differences between imperative and functional programming languages and only in 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 details. 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

Our 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 in providing a formal language for expressing theorems and proofs, called Gallina, a language for automating trivial proof steps and writing proof search procedures, called Ltac, and a language of commands, called Vernacular, which simplifies tasks

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

like looking up theorems from 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 3). 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 are 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] [29], 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. Our account is based mostly on Appendix A from [30] and sections 2.3 and 2.5 from [31]. The material presented in here is standard, but in the light of previous section's closing paragraphs we en-

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

courage people familiar with type theory to read it as a description not of a formal system, but of a model of computation.

As a model of computation, type theory is a bit unusual, because the most important things it is concerned with are not programs or computations – as the name suggests, its basic objects of interest are types. We can think of types as specifications that prescribe what is to be computed. Only after we have a type can we talk about the terms of that type, which are the formal counterparts of the informal notion of program/algorithm. This means that every program comes equipped with a specification which tells us what it computes.

Types and terms are manipulated using judgements, which are just assertions that may or may not be derivable.[8] The two basic kinds of judgements are the type formation judgement $\vdash A$, which asserts that A is a type, and the typing judgement $\vdash t : A$, which asserts that t is a term of type A.[9] For example, the judgement $\vdash \mathbb{N}$ asserts that $\mathbb{N}$ is a type (as we will see, it is the type of natural numbers). Similarly, the judgement $\vdash \mathbb{N} \to \mathbb{N}$ asserts that $\mathbb{N} \to \mathbb{N}$ is a type (of functions from natural numbers to natural numbers). The judgement $\vdash 2 + 2 : \mathbb{N}$ asserts that $2 + 2$ is a natural number, whereas $\vdash \lambda n.n + 5 : \mathbb{N} \to \mathbb{N}$ asserts that the function which adds 5 to its argument is a function from natural numbers to natural numbers.

The notions of type and term are not absolute. Sometimes we can assert that A is a type or that t is a term of type A only under some assumptions. For example, we can assert that $\texttt{Fin}(n)$, the type that is supposed to have exactly n distinct elements, is indeed a type, only under the assumption that n is a natural number. We write this formally as $n : \mathbb{N} \vdash \texttt{Fin}(n)$. Another example: we can assert that $n + m$ is a natural number only under the assumptions that n and m are natural numbers, which we write as $n : \mathbb{N}, m : \mathbb{N} \vdash n + m : \mathbb{N}$.

We keep track of assumptions using contexts. When a context Γ appears to the left of $\vdash$ in a judgement, it means that this judgement is made under assumptions from Γ. Such a judgement is called a hypothetical judgement. So, in their full generality, type formation judgement and typing judgement are hypothetical judgements. The type formation judgement is written $\Gamma \vdash A$ and read "in context Γ, A is a type", whereas the typing judgement is written $\Gamma \vdash t : A$ and read "in context Γ, t is of type A". What is and what is not a context is governed by a third kind of judgement, the well-formed context judgement, which is written $\Gamma \vdash$ and read "Γ is a (well-formed) context". This kind of judgement is absolute and can not depend on any assumptions.

It is important to reiterate that not all judgements are derivable. For example, the judgement $\vdash 5 : \mathbb{N} \to \mathbb{N}$ asserts that five is a function from natural numbers to natural numbers, but this is clearly nonsense. To know which judgements are derivable (and which are not), we need inference rules. An inference rule in general

[8] For judgements and in the context of type theory, "derivable" simply means "true".

[9] We can also read this as "t has type A" or "t inhabits type A".

has the form “if J_1, …, J_n, then J” and means that we can derive the judgement J provided that we have already derived all of the judgements $J_1, \ldots, J_n$. An important special case are inference rules with no premises, i.e. of the form “J”. Such a rule declares that the judgement J can be derived unconditionally. For rules with no premises, we will identify the whole rule with the judgement J. In most texts, inference rules are written with a horizontal line notation, with premises (if any) above the line and the conclusion below it, but in the following paragraphs we will state them inline using ordinary sentences.

In most varieties of type theory, the inference rules can be roughly classified into two genres:[10] the core rules and the type-directed rules. The core rules tell us about the basics of each judgement and how the judgements interact. The type-directed rules tell us how to form the given type (these are called formation rules), how to create and use terms of this type (introduction and elimination rules, respectively) and how to compute with them (computation and uniqueness rules). Additionally, there are some type-directed congruence rules which tell us how the formation, introduction and elimination rules behave with respect to computation. These congruence rules are very schematic and therefore often omitted from most presentations of type theory, but they are nonetheless necessary for the whole system to work correctly.

Let’s start with rules for contexts. First, there is the empty context, usually denoted by leaving empty space to the left of $\vdash$, and a rule which says that the empty context is a well-formed context. Second, if A is a type in context Γ, then the context $\Gamma, x : A$ (i.e. Γ extended with a variable x of type A) is a well-formed context, provided that the name x doesn’t already occur in Γ. There’s also a very important typing rule mainly concerned with contexts, called the assumption rule. It states that if we have a well-formed context Γ which contains the assumption $x : A$, then we can derive $\Gamma \vdash x : A$. These three rules are the most important core rules. We will discuss other core rules as we go along.

Moving on to the type-directed rules: for each type there is exactly one rule (called the formation rule) that tells us how this type is formed. For example, if in context Γ 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 formation rule for the natural numbers is $\Gamma \vdash \mathbb{N}$, i.e. $\mathbb{N}$ is a type in any context.

Rules for terms (i.e. for the typing judgement) can be divided into two main genres: introduction rules (also called constructors) and elimination rules (also called eliminators). Introduction rules for a given type tell us how to create terms of this type. For example, there are two introduction rules for natural numbers. The first says that $\Gamma \vdash 0 : \mathbb{N}$, i.e. that 0 is a natural number. The second says that given $\Gamma \vdash n : \mathbb{N}$, we have $\Gamma \vdash \mathtt{succ}(n) : \mathbb{N}$, i.e. that when we have a natural number n, its successor $\mathtt{succ}(n)$ is also a natural number. A type need not have any introduction

[10] We use the word “genre” (in its ordinary sense) to avoid words like “type” or “kind” which may also be used in a technical sense.

rules at all – for example, the empty type $\bot$ doesn't have introduction rules because we don't want to have any way of creating terms of this type.

Elimination rules for a given type, on the other hand, tell us how to use terms of this type to create terms of other types. For example, the simplest elimination rule for $\mathbb{N}$ says that if $\Gamma \vdash z : X$ and $\Gamma \vdash s : X \to X$ and $\Gamma \vdash n : \mathbb{N}$, then $\Gamma \vdash \texttt{rec}_{\mathbb{N}}(X, z, s, n) : X$. This rule, alternatively called a recursor, allows us to define functions with codomain X by recursion on natural numbers, but to do proofs by induction we would need a stronger elimination rule which we won't state here to keep things simple. As with introduction rules, a type needs not have any elimination rules. This is traditionally the case with the universe of types $\mathcal{U}$, although the reason behind this tradition is not clear.[11]

Now that we know how programs and their specifications work in our type-theoretical model of computation, it's time to learn about computation itself. Here too type theory is pretty unusual (as far as models of computation are concerned): instead of specifying how computation of a given program proceeds, it specifies which programs are considered equal. At first sight it may seem that equality has nothing to do with computation, but at a second glance there is an obvious link: whenever we have an equivalence relation R, we may think of its equivalence classes as preimages of elements of the codomain of some function f, and conversely we may see f as computing a canonical representative of each equivalence class of R. In our setting, the relation R is program equality, whereas the function f assigns to each program its final result. So, in the end, if we define program equality in the appropriate way, we will get the appropriate notion of computation for free. By "appropriate" we mean that programs are equal when they compute the same result, irrespective of how long it takes or in what order the computation steps are performed.

In type theory, programs are terms and their equality is realized by introducing a new judgement $\Gamma \vdash t_1 \equiv t_2 : A$ which asserts that in context Γ, t_1 and t_2 are convertible[12] terms of type A. For example, the judgement $\Gamma \vdash 2 + 2 \equiv 3 + 1 : \mathbb{N}$ asserts that $2+2$ and $3+1$ are convertible terms of type $\mathbb{N}$, whereas $\Gamma \vdash \lambda n.3 + n \equiv \lambda n.5 + n : \mathbb{N} \to \mathbb{N}$ asserts that the functions which add three and five to their arguments, respectively, are convertible. Both $2 + 2$ and $3 + 1$ should compute the result 4, so we expect the first judgement to be derivable, whereas we expect the second one not to be derivable, even if we're not entirely sure what do these functions compute to for an unspecified n.

The core inference rules that govern the convertibility judgement ensure that it is an equivalence relation. For example, there's a rule establishing that convertibility

[11] One possible answer can be that an eliminator for $\mathcal{U}$ breaks parametricity. A more modern answer [32] would be that it contradicts Univalence. For a language that allows eliminating types, see Idris 2. [33]

[12] Other common names for this judgement include "definitional equality" and "judgemental equality", but we won't use them. Perhaps a better name in this fashion would be "computational equality".

is symmetric, which means that given $\Gamma \vdash t_1 \equiv t_2 : A$, we can derive $\Gamma \vdash t_2 \equiv t_1 : A$. We omit the obvious rules for reflexivity and transitivity. Sometimes these core rules are only admissible, i.e. they are proved to be derivable even though they are not explicitly included in the definition of convertibility.

Rules that govern the behaviour of convertibility, similarly to rules for creating terms, come in two main 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 $\Gamma \vdash s : X \to X$, we have $\Gamma \vdash \mathtt{rec}_{\mathbb{N}}(X, z, s, 0) \equiv z : X$, and the second rule states that given $\Gamma \vdash z : X$, $\Gamma \vdash s : X \to X$ and $\Gamma \vdash n : \mathbb{N}$ we have $\Gamma \vdash \mathtt{rec}_{\mathbb{N}}(X, z, s, \mathtt{succ}(n)) \equiv s(\mathtt{rec}_{\mathbb{N}}(X, z, s, n)) : X$.

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 function types states that given $\Gamma \vdash f : A \to B$ we can derive $\Gamma \vdash \lambda x.f(x) \equiv f : A \to B$. Uniqueness rules are completely optional – a type need not have any. For example, almost all presentations of type theory lack a uniqueness rule for the natural numbers. In general, uniqueness rules are common for the unit type, the product type, the function type and other negative types, whereas they are very rare for positive types.[13]

Besides computation and uniqueness rules, there is also a huge number of congruence rules which ensure that convertibility behaves well with respect to introduction and elimination rules. For example, we have a rule which says that the successors of convertible natural numbers are convertible, i.e. that given $\Gamma \vdash n \equiv m : \mathbb{N}$, we have $\Gamma \vdash \mathtt{succ}(n) \equiv \mathtt{succ}(m) : \mathbb{N}$. For the elimination rules, we have a rule which says that given $\Gamma \vdash z_1 \equiv z_2 : X$ and $\Gamma \vdash s_1 \equiv s_2 : X \to X$ and $\Gamma \vdash n_1 \equiv n_2 : \mathbb{N}$, we have $\Gamma \vdash \mathtt{rec}_{\mathbb{N}}(X, z_1, s_1, n_1) \equiv \mathtt{rec}_{\mathbb{N}}(X, z_2, s_2, n_2) : X$.

This is the end of our description of convertibility of terms, but the matter of computation hasn't been settled yet. Because types can depend on terms, computation can also occur inside types, which prompts us to consider not only term equality, but also type equality. Recall the type $n : \mathbb{N} \vdash \mathtt{Fin}(n)$ that has exactly n elements. What relationships are there between instances of this type with various terms substituted[14] for n? For example, what is the relation between the types $\mathtt{Fin}(2 + 2)$ and $\mathtt{Fin}(3 + 1)$? Are they the same or different?

To even be able to ask this question formally, we need another kind of judgement, namely the type convertibility judgement $\Gamma \vdash A \equiv B$ which asserts that in

[13] The reason for not including uniqueness rules for positive types is mostly that they are hard to implement, but there also are some model-related concerns. See [34] for how to formulate a uniqueness rule for the natural numbers and for a more philosophical discussion of uniqueness rules in general. Also see [35] for a lightweight blogpost introducing the distinction between positive and negative types.

[14] We won't discuss substitution in our presentation of type theory.

context Γ the types A and B are convertible. Similarly to what was the case for term convertibility, if the inference rules for this judgement are defined in the right way, we obtain for free a notion of type computation such that two types are convertible precisely when they compute to the same result.

The most important core rule that involves type convertibility is called the subsumption rule. It says that if we can derive $\Gamma \vdash t : A$ and $\Gamma \vdash A \equiv B$, then we can also derive $\Gamma \vdash t : B$. The other rules that govern type convertibility aren't terribly interesting. Similarly to term convertibility, we have three core rules which establish that type convertibility is an equivalence relation. For example, we have a rule which says that if we can derive $\Gamma \vdash A \equiv B$ and $\Gamma \vdash B \equiv C$, then we can derive $\Gamma \vdash A \equiv C$. Just as in the case of term convertibility, these three rules may sometimes be admissible instead of being included in the definition.

Moreover, for each type formation rule we have one congruence rule which says that types formed from equal components are equal. For example, for the type $\mathtt{Fin}(n)$ we have a congruence rule which says if we can derive $\Gamma \vdash n \equiv m : \mathbb{N}$, then we can derive $\Gamma \vdash \mathtt{Fin}(n) \equiv \mathtt{Fin}(m)$. These congruence rules allow us to answer our initial question in the affirmative – because $2 + 2$ and $3 + 1$ are convertible, i.e. $\Gamma \vdash 2 + 2 \equiv 3 + 1 : \mathbb{N}$, the types $\mathtt{Fin}(2 + 2)$ and $\mathtt{Fin}(3 + 1)$ are also convertible, i.e. $\Gamma \vdash \mathtt{Fin}(2 + 2) \equiv \mathtt{Fin}(3 + 1)$.

We are mostly done describing type theory, but we will take some more time to list differences between our presentation and what may be found elsewhere. First, some presentations of type theory can include additional judgements. For example, there may be a context convertibility judgement – because convertible but not syntactically identical types like $\mathtt{Fin}(2+2)$ and $\mathtt{Fin}(3+1)$ can appear in contexts as part of an assumption, judging whether two contexts are equal may be of importance. This is especially the case in more concrete presentations aimed at implementation. See [36] and [37] for type theories that have a context convertibility judgement.

Second, some presentations of type theory may not have some of the judgements we have seen. This is most often the case for the type formation judgement and the type convertibility judgement, as they may be replaced by formulating type formation rules as introduction rules for the universe[15] and type convertibility rules as term convertibility rules for the universe. See [30] (Appendix A) for a text that takes this approach.

We didn't discuss substitution in our presentation of type theory, but there are typically two ways of doing it. The first one, called implicit substitutions, is to define a meta-level substitution function by (mutual) recursion on terms, types and contexts. The second one, called explicit substitutions, is to introduce more judgements (the well-formed substitution judgement and the substitution convertibility judgement) and more inference rules for term and type convertibility which describe

[15] We didn't discuss universes in our presentation, as their handling is somewhat technical.

the computational behaviour of these substitutions. The implicit approach is more abstract, less detailed and, in general, better suited for informal presentations. The explicit approach is more concrete, more detailed and better fit for the purpose of implementation. For presentations of type theory with explicit substitutions, see [36] and [37].

Of course our presentation of type theory is far from being complete – to be, we would have to list all the formation, introduction, elimination, computation and uniqueness rules for all types. If we also diligently list all the boring congruence rules, the number of inference rules can easily surpass one hundred. But even though a full formal presentation of type theory is massive, the basic idea behind it is very simple and beautiful.

Having discussed how type theory works, we would also want to take a look at all the pleasant metatheoretical properties it enjoys. To state most of them, we will need to view computation from a different angle, by properly describing what it means that a term t computes[16] to a term t' (in a single computation step).[17] The definition of this relation (which we write as $t \rightsquigarrow t'$) is pretty simple, but very long to state precisely. The most important rules for $\rightsquigarrow$ correspond to left-to-right readings of computation rules for term convertibility. Besides, there is also a huge number of congruence rules which can easily be derived from the congruence rules for term convertibility, but do not correspond to them directly.

Let's see how this looks for natural numbers. We have $\mathtt{rec}_{\mathbb{N}}(X, z, s, 0) \rightsquigarrow z$ and $\mathtt{rec}_{\mathbb{N}}(X, z, s, \mathtt{succ}(n)) \rightsquigarrow s(\mathtt{rec}_{\mathbb{N}}(X, z, s, n))$, which correspond to the computation rules.[18] There also are the congruence rules, although this time they look differently from these for the term convertibility judgement. First, there are congruence rules for the introduction rules: given $n \rightsquigarrow n'$ we have $\mathtt{succ}(n) \rightsquigarrow \mathtt{succ}(n')$. Second, there are congruence rules for the elimination rule: given $z \rightsquigarrow z'$, we have $\mathtt{rec}_{\mathbb{N}}(X, z, s, n) \rightsquigarrow \mathtt{rec}_{\mathbb{N}}(X, z', s, n)$; given $s \rightsquigarrow s'$ we have $\mathtt{rec}_{\mathbb{N}}(X, z, s, n) \rightsquigarrow \mathtt{rec}_{\mathbb{N}}(X, z, s', n)$; and given $n \rightsquigarrow n'$ we have $\mathtt{rec}_{\mathbb{N}}(X, z, s, n) \rightsquigarrow \mathtt{rec}_{\mathbb{N}}(X, z, s, n')$. The last congruence rule we need says that given $X \rightsquigarrow X'$ we have $\mathtt{rec}_{\mathbb{N}}(X, z, s, n) \rightsquigarrow \mathtt{rec}_{\mathbb{N}}(X', z, s, n)$.

This leads us to consider the last ingredient we need in our description of computation, namely type computation. To denote that the type A reduces to A' in a single computation step, we write $A \rightsquigarrow A'$. The rules for type computation are not very interesting and consist exclusively of congruence rules.[19] There aren't any

[16] This relation is sometimes called "reduction" or "simplification", but we consider these to be misnomers, because in general computation needs not decrease any perceived measure of size or complexity.

[17] This view is called "operational semantics" and it is exceedingly rare to treat type theory in this way. See [38] for more details.

[18] For functions, we would also have $\lambda x.f(x) \rightsquigarrow f$, which corresponds to the uniqueness rule.

[19] The interesting cases of type-level computation, like computing a type by recursion on a natural number, are already covered by rules for term computation.

congruence rules for $\mathbb{N}$, but for Fin given $n \rightsquigarrow n'$ we have $\mathtt{Fin}(n) \rightsquigarrow \mathtt{Fin}(n')$, whereas for function types we have two rules: given $A \rightsquigarrow A'$ we have $A \to B \rightsquigarrow A' \to B$; and given $B \rightsquigarrow B'$ we have $A \to B \rightsquigarrow A \to B'$. We now know enough about computation to state all the interesting metatheorems.

The most important metatheoretical property of type theory, called strong normalization, is that the converse of the relation $\rightsquigarrow$ is well-founded. Stated more explicitly, strong normalization says that all sequences of computation steps of the form $t_0 \rightsquigarrow t_1 \rightsquigarrow \ldots$ are finite. Even more explicitly, this means that every term t has a normal form, i.e. after a finite number of computation steps (no matter in which order these steps are performed) it eventually computes to some term t' which doesn't compute anymore.

The next important metatheoretical property of type theory, called confluence, says that if $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$, then there exists a term t' such that $t_1 \rightsquigarrow^* t'$ and $t_2 \rightsquigarrow^* t'$. Here the relation $\rightsquigarrow^*$ denotes the reflexive-transitive closure of $\rightsquigarrow$, i.e. $t \rightsquigarrow^* t'$ means that t computes to t' after zero or more computation steps. In ordinary words, confluence is a kind of generalized determinism – the computation relation $\rightsquigarrow$ is not deterministic, but if a term computes to two different terms, these will in turn, sooner or later, compute to the same term. To paraphrase once more: confluence means that every term t has at most one normal form.

We can summarize strong normalization and confluence in one sentence by saying that "every term has a unique normal form". These normal forms can be characterized explicitly: they are generated by introduction rules[20] and elimination rules applied to variables. For example, some terms of type $\mathbb{N}$ in normal form are 0, $\mathtt{succ}(\mathtt{succ}(n))$ and $\mathtt{rec}_{\mathbb{N}}(X, z, s, n)$ (provided that X, z and s are in normal form; in both previous terms n is a variable). The situation gets even simpler for closed terms, i.e. terms in the empty context. Because there are no variables in the empty context, closed normal forms can be neither variables nor elimination rules applied to variables – they must start with an introduction rule.

There's a (very rarely mentioned) metatheoretical property related to the above characterization, called "progress": every closed term t either starts with an introduction rule or there is another term t' to which t computes. Intuitively, this means that the computation of every closed term progresses towards a normal form which is well-defined, rather than a normal form that can't compute anymore because the rules of type theory were designed badly.

Note that even though normal forms in the empty context start with an introduction rule, they aren't made only of introduction rules. The culprit here are functions, whose normal forms can look like $\lambda n.\mathtt{rec}_{\mathbb{N}}(X, z, s, n)$ (provided that X, z and s are in normal form) – they start with a lambda (the introduction rule), but then there is an elimination rule applied to a variable bound by the lambda.

[20] And also formation rules, when we're talking about types.

This observation is very important to be able to correctly state the second most important metatheoretical property of type theory, called canonicity: every closed term of base type computes to canonical form. Here, a base type is any type in the empty context besides the (dependent) function type and a canonical form is a term generated solely by introduction rules.

But how do we know which normal/canonical forms correspond to which types? This may seem obvious, but to establish the obvious we actually need two more metatheoretical properties, concerned with types. The first one is uniqueness of types: if we have $\Gamma \vdash t : A$ and $\Gamma \vdash t : A'$, then we can derive $\Gamma \vdash A \equiv A'$. Intuitively: the type of every term is unique. The second property is called either "(type) preservation" or "subject reduction", and says that if $\Gamma \vdash t : A$ and $t \rightsquigarrow t'$, then we can derive $\Gamma \vdash t' : A$. This one is also easy to grasp intuitively – it means that the computation of a term doesn't change its type.

Together with the two above properties, canonicity basically says that every closed term of type $\mathbb{N}$ computes to a numeral $(0, \texttt{succ}(0), \texttt{succ}(\texttt{succ}(0)), \ldots$ i.e. $0, 1, 2, \ldots)$, every closed term of the boolean type computes to `true` or `false` and so on. This is an extremely important property, because canonical forms are what programmers (and mathematicians) think of when they think about a given type. For example, people tend to think about the type $\mathbb{N}$ of natural numbers in terms of... well, the *actual* natural numbers $(0, 1, 2, \ldots)$, and not in terms of expressions like $2 + 2$ or $(\lambda n.n + 5)\ 37$ which only eventually *compute* to a natural number. The importance of canonicity stems from the fact that it validates this widespread intuitive way of thinking.

This is how the above metatheorems work in general – they validate intuitions widely held by programmers and mathematicians. When presented with an object, these people usually think of it as being of a particular type, which is unique (uniqueness of types).[21] When presented with $n : \mathbb{N}$, they think of it as a number and not a program that computes a number (canonicity). When programmers think of $n : \mathbb{N}$ as a program, they usually assume that it terminates[22] (strong normalization) and that its type doesn't change during computation (type preservation).[23] They also mostly think that computation happens in a fixed order and leads to a deterministic result (confluence)[24] which is meaningful and not some gibberish (progress).

An immediate corollary of these metatheoretical properties is consistency of type theory as a logic, which means that we can't construct a closed term of the empty type. Let's assume to the contrary that type theory is inconsistent, i.e. that we have some $\vdash e : \bot$. But then, by canonicity, e computes to canonical form, which

[21] The common exception in programming is subtyping, which is also present in mathematical thinking, for example when identifying a natural number with its corresponding integer.

[22] Nontermination ascends to programmers' consciousness mostly when dealing with bugs.

[23] Implicit casts, which change the type of the object on which they act, are a common source of errors in languages that have them, as programmers often overlook them.

[24] Provided they are not dealing with concurrency, randomness or other side effects.

means that e starts with one of the introduction rules for type $\bot$. But there are no such rules for $\bot$! Therefore, we have a contradiction and type theory is consistent.

Consistency is not the only corollary of these metatheoretical properties – there is more to them than just validating intuitions and expectations. In addition to that, the properties we have seen also have some more practical consequences – they guarantee that the judgements of type theory are decidable, which in turn guarantees that type theory can be efficiently implemented on a computer.

We can easily decide whether $\Gamma \vdash A$ is derivable – to do this, we need to compute A to normal form and then check if it was correctly built according to the formation rules, which is easy for types in normal form. It follows that the judgement $\Gamma \vdash$ is also decidable – it suffices to check whether all types that occur in Γ are well-formed. The typing judgement $\Gamma \vdash t : A$ is decidable too - to decide whether it holds, we need to compute both t and A to normal form, and then check whether t was built with the introduction and elimination rules that correspond to type A – again, this is easy for normal forms. We can also decide whether $\Gamma \vdash A \equiv A'$ and $\Gamma \vdash t \equiv t' : A$ are derivable. Confluence yields as a corollary the fact that two terms are convertible when they compute to the same normal form (and an analogous corollary for types), so to decide these judgements it suffices to compute both sides to normal form and then check if they are syntactically the same.

We finish our description of type theory by changing our perspective from a model of computation back to a foundational one and pose the following question: what is the relation between type theory and set theory? After all, types have terms and sets have elements, which might make both theories look more alike than really is the case. We dispel the potential confusion in advance with a short comparison below.

Set theory has two basic kinds of objects – propositions and sets – living in two separate "layers" of the theory. 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. In type theory, both propositions and sets can 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 (i.e. within set theory) thanks to the Law of Excludded Middle, but not externally (i.e. in the metatheory), 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 judgement, 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, as we have seen above, it is decidable externally.

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. As we have also seen above, terms can inhabit only one type, or, to be more precise, if a term inhabits 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[25] and run the listing below[26] in interactive mode. This experience will greatly enrich the explanation.

```
Require Export List Permutation Lia Arith.
Export ListNotations.

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

Print app.
(*
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.
```

[25] Available from [12].

[26] It can be found in the thesis' repository `https://github.com/wkolowski/coq-algs` in the directory `Thesis/Snippets/`

```
  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 some modules: one for working with lists, as we will need some list functions and notations (Coq allows defining custom notations using a built-in notation mechanism), another that contains `lia`, a procedure for reasoning in **l**inear **i**nteger **a**rithmetic (hence the name), and also some facts about Peano arithmetic and permutations of lists. For now we only need the `List` and `ListNotations` modules – the rest will be useful later. In the future we will follow the convention that at the beginning of each chapter, we import code from the previous chapter without mentioning this fact explicitly.

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 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 1.2.

`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 1.3 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[27] 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 "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 con-

[27] 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, there is a special type of propositions called Prop.

text 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.[28] 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 [39] 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 [40] 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!
- Certified Programming with Dependent Types [41] 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) [42], which is an extended version of Chris Okasaki's PhD thesis [43] (see [44] 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 [45].

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

[28] When numbering concepts and techniques, we won't distinguish between them. 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.

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 (in earlier times it was thought that amortization is contradicted by persistence) and that it is inseparably tied to lazy evaluation. This aspect of PFDS won't be important for us because, as we have already underlined many times, we are not interested in complexity – only in formal correctness.

The third big theme of the book are numerical representations. The idea is 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. In fact we think that PFDS did such a great job at developing the field of functional data structures that in this thesis we will occupy ourselves entirely with functional algorithms.

The other book that is relevant to our work is "Verified Functional Algorithms" [46], which is actually the third volume of the aforementioned Software Foundations series [39]. 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, decidable 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 it is not that much concerned with performance, just as we aren't, VFA sees the performance of an algorithm implemented in Coq as a property of the OCaml program extracted[29] from the Coq code, rather than of the Coq program itself, like we do.

1.8 Outline of the thesis

After this chapter's thorough introduction it should be clear now what we mean by "formally verified functional algorithms in Coq". In the remaining chapters of the thesis we describe the actual "concepts and techniques". The rest of the thesis is structured as follows:

- In Chapter 2 we look at techniques for verifying a specification's usefulness and for improving bad specifications.[30]
- In Chapter 3 we show how to implement an abstract template of an algorithm without worrying too much about concrete details or the termination proof.
- In Chapter 4 we describe a general method for defining recursive functions and carrying out termination proofs.
- In Chapter 5 we describe a general technique for reasoning about functions and show how to use it to formally prove the abstract algorithm template correct in a top-down fashion.
- In Chapter 6 we review all our concepts and techniques from a birds'-eye view, describe the nature of our contributions and conclude with a discussion of related and further work.

[29] 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.

[30] Sadly, there seems to be no easy way of coming up with a good specification.

Chapter 2

Specify the problem

In the previous chapter we have already seen our first concept/technique 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[1] as it requires a significant amount of mathematical creativity and insight, which can only be acquired with experience. As we will see in later chapters, 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.

The problem, whose specification we will be looking for in this chapter, is that of sorting. It is a very simple problem concerned with moving elements around in a list, the simplest and most ubiquitous data structure of functional programming. But don't let yourself get fooled by the apparent simplicity – we will see that sorting is more than enough to illustrate the various possible traps that an inexperienced practitioner of formally verified functional algorithms can fall into when trying to devise a specification.

As a guide to avoiding these traps, we present a simple heuristic for checking if a specification is good and illustrate it with a few 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 sufficiently abstract and easy to use.

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

2.1 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.

```
Fixpoint nth {A : Type} (n : nat) (l : list A) : option A :=
match l with
    | [] => None
    | h :: t =>
        match n with
            | 0 => Some h
            | 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 defining 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. Note that the use of `option` in the return type is necessary because we always need to return a result and, if the return type were just `A`, this wouldn't be possible for the empty list as we can't conjure an element of `A` out of nowhere. Moreover, in Coq pattern matching has to be exhaustive, i.e. all cases must be covered, so we can't just omit the case of empty list.[2] Neither can we throw an exception, because in Coq (and in type theory) exceptions are not allowed.

Then comes the definition of sortedness itself: the list `l` is sorted when, for any two indices `i` and `j` such that `i` comes before `j`, if 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 we will nonetheless try to give an

[2] Type theories usually don't have pattern matching, but when they do, it must be exhaustive too.

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

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 and the inductive definition of the relation `<=`. Natural numbers are defined by postulating `0 : nat` (zero is a natural number) and `S : nat -> nat` (the successor of a natural number is a natural number), whereas the relation `<=` is defined by postulating `forall n : nat, n <= n` and `forall n m : nat, n <= m -> n <= S m`. The mismatch stems from the fact that matching on `n` does not reveal in which case of `n <= m` we are and conversely, matching on a proof of `n <= m` does not reveal whether `n` is zero or a successor. This, however, is not a big problem all by itself – it's 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 in the left-hand sides 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, thus preventing us from using the equations unless we also 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 of `i <= j` would be a bit annoying. If we went for an 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.

2.2 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 #2

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 []
    | 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.

[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 [47] for a programmer-friendly introduction and [48] for a more academic work.

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

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.[6] 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.

```
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).
```

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 lists 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 of index inequality, equations between terms, 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.

2.3 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.[49]

[6] This transformation crucially relies on the fact that the usual order relation on naturals is transitive. The generalized version of this definition, which we will see in the next section, is equivalent to a generalized version of the patchwork definition only for transitive relations.

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 second glance, however, we may notice that sometimes being constrained is advantageous, as we have fewer ways of being wrong. After all, most of the rich structure of natural numbers isn't very relevant to the problem of sorting. At third glance, this should be obvious to every proponent of strong, static typing – the type system is precisely a mechanism that constrains the programmer by ruling out suspicious (i.e. not well-typed) programs.[7]

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).
```

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.

2.4 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.

[7] A very good source of examples of Inventor's paradox are proofs by induction. Sometimes proving a theorem with a stronger conclusion can be easier than proving a theorem with a weaker conclusion, because a strong induction hypothesis is much more valuable than a weak induction hypothesis.

That's a significant mismatch, because a sorting function is not any function 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_nil :
      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 something that satisfies the specification, but we won't be this hasty – there's one more concept that can be useful for validating and improving specifications. Let's say we have a feeling that this definition will fail the "easy to use" criterion at some point in the future during a proof of correctness of some nontrivial sorting algorithm. We think this could be so because the algorithm moves list elements around in ways that don't play too well with the above definition of `Permutation`. How to find a better definition in such a case?

Formally verified algorithm concept/technique #3

Sometimes we can find an alternative specification by changing its focus.

This one is admittedly quite murky. What is meant by "focus" here and how can we change it? The best way to understand this concept is to immediately apply it to our case. In the above definition of Permutation, the main idea is that of simultaneously building up two lists, starting from two []s, using constructors that preserve the fact of being a permutation. So, we can say that this definition focuses on *generating* permutations.[8] If we want to apply the above concept/technique, we should try 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*: two lists are permutations of one another when they contain the same number of occurrences of each x : A. Note that, because we count elements, this new definition only applies to types that have decidable equality. This restriction is not a big problem in practice, because the need to sort functions, streams, or other such infinite objects lacking decidable equality, is pretty rare.

But the resulting definition by counting is *not abstract enough*, because it *assumes what is not needed*: to use this definition for lists that contain elements of type A, we need to prove that A has decidable equality. We can improve the definition with one last step of abstraction, by generalizing from counting occurrences of elements to counting the number of elements satisfying a general decidable property p : A -> bool. This definition is clearly equivalent, but we don't need to prove anything to use it.

```
Inductive Transposition {A : Type} : list A -> list A -> Prop :=
    | Transposition' :
        forall (x y : A) (l1 l2 l3 : list A),
          Transposition (l1 ++ x :: l2 ++ y :: l3)
                        (l1 ++ y :: l2 ++ x :: l3).

Inductive Permutation {A : Type} : list A -> list A -> Prop :=
    | Permutation_refl :
        forall l : list A, Permutation l l
```

[8] Of course all inductive definitions are about generating objects. The sense in which we use the word "generating" will become clearer in the next paragraphs, after looking at ways of defining permutations that have a different focus.

```
  | Permutation_step_trans :
      forall l1 l2 l3 : list A,
        Transposition l1 l2 ->
          Permutation l2 l3 -> Permutation l1 l3.
```

Counting doesn't exhaust the pool of ideas around which we can focus a definition of permutation. Yet another such idea is *permuting* (that is, rearranging) the elements in a list. This idea is arguably the one most apparent in the name "permutation" and it corresponds to the well-known theorem of classical mathematics that a permutation can be represented as a product of transpositions.

The corresponding definition is shown in the listing above. First we define transpositions, the meaning of `Transposition l1 l2` being that we can take any two elements of `l1` and switch their places, getting `l2` as a result. Then a permutation is defined effectively as a sequence of such transpositions, where the constructor `Permutation_refl` represents no transpositions at all, and the constructor `Permutation_step_trans` means that to permute `l1` into `l3` we first need to transpose two elements in `l1` to obtain `l2` and then permute the elements of `l2` to get `l3`.

So far we have seen three different definitions of `Permutation`, but there's still more to it, as each of the central ideas can be put into practice in slightly different ways. Generation can be realized by repeatedly inserting the same element into two lists at possibly different positions. The patchwork-y definition by counting could be inductivized a little, which would also generalize it to types without decidable equality. And permuting may be implemented by seeing a permutation as a sequence of *adjacent* transpositions, i.e. transpositions that apply only to neighbouring elements in a list.

The focal ideas of *generating*, *counting* and *permuting* yield definitions which are pretty different, but it is crucial to observe that the precise definition matters only if we are starting from scratch and have to prove all the necessary lemmas ourselves. In such a case different definitions are fit for different purposes and vary greatly in their strengths and weaknesses, but in case all the properties of permutations we need are readily available, the definition doesn't matter.[9] This is indeed the case for the original definition we gave (the one based on generating), because Coq's standard library is replete with lemmas that we can use in our proofs, which guarantees that this definition is easy to use. Because it is also abstract enough, we will stick with it.

[9] Another way not to worry about how we define permutations is to abstract over this definition by taking "permutation" to mean any relation `P : forall A : Type, list A -> list A -> Prop` that satisfies the properties we will need it to satisfy. This is a glimpse of the concepts and techniques we will see in the chapters to come.

The final specification of a sorting function that we have arrived at through our considerations is this: 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, whose definition is shown in the listing below.

```
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)
}.
```

To sum up our struggles: in this chapter 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 learned two hints that can help us improve a bad specification, but they are no silver bullets: sometimes we can improve a patchwork definition by turning it into an inductive definition, and sometimes we can find a new definition by focusing on a different aspect of the object we are defining, but it is hard to tell in advance whether we should take such steps. We once more stress the fact that inventing specifications is an art that one gets better at with experience.

Chapter 3

Abstract the algorithm

Now that we have found a good specification of the problem, the next step in the process is to come up with its solution, but we will not concern ourselves with that because the concepts and techniques that aid in inventing algorithms (like divide and conquer, dynamic programming or backtracking) are well-known and covered in all standard textbooks.

Rather than invent algorithms from scratch, in the chapters to come we will show how to formalize and prove them correct. Our running example will be quicksort, one of the fastest and most elegant solutions of the problem of sorting. 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.

3.1 Top-down is better than bottom-up

The biggest temptation faced by an algorithmist when implementing an algorithm is to get drowned in unnecessary details too quickly – to mistake the trees for the forest, that is. The best way of combating this temptation is to constantly keep in mind a high-level idea that underlies the algorithm. For quicksort (agnostic as to whether the thing being sorted is a list or an array), it usually goes 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.

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

The famous two-line[1] implementation of quicksort in Haskell, shown in the listing above, clearly shows that the idea behind quicksort is pretty simple and easy to implement correctly. But whenever you see an imperative implementation (a typical example of which is shown in the listing below), you probably aren't very convinced of this fact.[2]

```
// 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);
    }
}
```

[1] Three if we count the type signature.

[2] If you don't believe me, try figuring out whether the C implementation is correct – I didn't check it!

The main difference between these two implementations is a discrepancy in their levels of abstraction. The first[3] and second[4] 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.

We have to admit that the presentation is a bit biased 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 biased 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 is, in some sense, much more abstract and modular. The C version, being much more concrete, isn't as friendly-looking.[5]

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 #4

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

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

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

[5] 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.

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 tailored 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 we can express are necessarily terminating. This raises the problem of termination checking: how does Coq know whether a function we're implementing is terminating? In many 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 (how to do it) and the 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 (T : Type) : Type :=
{
    short : list T -> bool;
    adhoc : list T -> list T;
    choosePivot : list T -> T * list T;
    partition : T -> list T -> list T * list T * list T;
}.

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[6] 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 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`).

3.2 User experience: concrete algorithm as a sanity check

Are we done with the template? Not quite – how do we know that our template makes sense? There's the obvious criterion of conforming to 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 template?

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

[6] 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.

our case things are a bit different, however: even though correctness is even more taken for granted, execution speed will never be satisfying[7]. 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 #5

Instantiate your template in the most naive way. See what happens and use that observation to improve the template.

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

[7] Coq users make various jokes along the lines of "Since discovering Coq, I don't run my programs anymore, 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.

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 because this number does not appear in the list (which is 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 T 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 (T : Type) : Type :=
{
    short : list T -> option (T * list T);
    adhoc : list T -> list T;
    choosePivot : T -> list T -> T * list T;
    partition : T -> list T -> list T * list T * list T;
}.

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 QSA_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 QSA_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 `T -> list T -> T * list T`, 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 T -> option (T * list T)`, 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.

3.3 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 we will call *evidence-based programming*.[8] 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

[8] This is a term I coined for the purpose of this thesis. Of course it is a pun on all the "evidence-based X", like evidence-based medicine or evidence-based policy.

disease that often afflicts programmers transitioning 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 ideas, like the *small-scale reflection* proof methodology or *intrinsic and extrinsic typing* in lambda calculus.

"Boolean blindness"[9] 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 [51] to name the erroneous view, held by some imperative programmers and classical mathematicians, that booleans and propositions are the same thing.[10] However, it is most often applied in quite different situations (see [52] for a more software-engineering-oriented presentation and [53] 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-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`),

[9] The term was coined by Dan Licata in his lecture notes for an introductory functional programming class [50].

[10] 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 [30].

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.[11] A piece of information is explicit when it is backed by evidence[12] 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.[13] In the `cons` case, we explicitly know that

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

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

[13] Recall that convertibility was explained in section 1.5.

`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 [54]. 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*[14], a general-purpose proof engineering methodology most famously applied to the first fully formal proof of the four colour theorem [58] [59]. 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.

[14] See [55] for an implementation of a proof language for Coq which is based on this methodology and [56] for its documentation. Also see [57], a book about a library of formalized mathematics done in Coq which uses the small scale reflection methodology.

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 others 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 #6

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.

3.4 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 class as a field. “Unbundling”, on the other hand, is realizing a component of a record or class as a parameter.

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

Coercion T : 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, because the type was a parameter. The listing above shows the alternative realization as a fully bundled class in which T is a field instead. We declare T 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 A, it takes only one, A : QSArgs, and we can use it to write list A just as if 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.[15][16] Why bother then?

[15] Our original choice, with T : 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.

[16] A pragmatically important thing to bear in mind when deciding between bundled and unbundled classes is that field names in Coq are treated like ordinary top-level names. This often leads to a polluted global context and interferes with other mechanisms, like automatic hypothesis naming in proofs. Parameters avoid this problem, because they are ordinary local names. Therefore, if you

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 #7

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.

want to name your type `T`, it might be smarter to make it a parameter rather than a field.

Chapter 4

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. In this section we will develop a better template that, when it is instantiated, guarantees that the concrete algorithm necessarily terminates. We will first learn how to do it manually (while hoping that we won't ever have to use this knowledge) and then we'll see a Coq feature which is just as good as the technique while being much more automated and easy to use.

4.1 The inductive domain method

How do we prove that our algorithm terminates? Before we tackle this question, let's first recall *why* we have to prove it. The only form of recursion allowed in Coq is *structural recursion*, i.e. one in which recursive calls are made on subterms of the principal argument. Because terms are finite,[1] this guarantees that all structurally recursive programs terminate and termination guarantees the consistency of Coq's logic.[2] Thus, on the one hand, the restriction to structural recursion protects us from contradiction (and from other evils too), but on the other hand it can be quite inconvenient from the pragmatic programmer's point of view.

[1] To be more precise, they are trees of finite height, but their width can be arbitrary.

[2] "Termination" is an informal term which means the same thing as "strong normalization". Recall that we proved that type theory is consistent in section 1.5.

Structural recursion is to be contrasted with *general recursion*, found in most programming languages, in which arguments of recursive calls can be arbitrary. This is very convenient for the pragmatic programmer, because it doesn't constrain him at all, but it also renders the language inconsistent as a logic and thus absolutely useless for formal reasoning.[3] For example, consider the proposition `False`, which in Coq also doubles as the empty type, because it has no constructors and thus no direct way to be proven. Using general recursion we could easily define a nonterminating function `f : True -> False` by setting `f x := f x`, which we could then call with a proof of `True` to obtain an proof of `False`.

Formally verified algorithm concept/technique #8

Define a predicate that represents the shape of recursion in the algorithm and use it to implement a better template.

Back to our (almost) initial question: we need to prove termination because only structural recursion is allowed and our algorithm is not structurally recursive. As for the truly initial question of how to do it, the answer is provided by the above technique which we will call the *inductive domain method*. It might look like a revelation at first, but in hindsight it couldn't have been more obvious. This technique basically tells us to represent general recursion on some type by structural recursion on a special type crafted precisely for this purpose. By doing that we get a new template which is guaranteed to terminate when filled in. There's a caveat, however: we changed the type of our algorithm, so what we got is not what we initially wanted. To implement the original algorithm, we will have to do some more work.

```
Inductive QSDom (A : QSArgs) : list A -> Type :=
  | Short :
    forall l : list A, short l = None -> QSDom A l
  | Long :
    forall {l : list A},
     forall {h : A} {t : list A},
      short l = Some (h, t) ->
     forall (pivot : A) {rest : list A},
      choosePivot h t = (pivot, rest) ->
     forall (eq : list A) {lt gt : list A},
      partition pivot rest = (lt, eq, gt) ->
     QSDom A lt -> QSDom A gt -> QSDom A l.

Fixpoint qs'
 {A : QSArgs} {l : list A} (d : QSDom A l) : list A :=
```

[3] Note, however, that an inconsistent language can still be useful for the purpose of *rigorous* reasoning. See [60] for details.

```
match d with
    | Short _ _ _ => adhoc l
    | Long _ _ pivot _ eq _ ltd gtd =>
      qs' ltd ++ pivot :: eq ++ qs' gtd
end.
```

The above listing shows how the inductive domain method looks like in practice. We define `QSDom`, an inductive family of domain predicates,[4] one for each parameter `A : QSArgs`. `QSDom A l` can be read aloud as "`l` belongs to the domain of quicksort" and an element of `QSDom A l` represents the complete call graph of our previous implementation of `qs` applied to `l`.

There are two constructors. `Short` takes as argument a proof of `short l = None`, so it represents the case in which the list `l` is short. It can be read as "if `l` is short, then it belongs to the domain of quicksort". `Long` takes, as one of its arguments, a proof of `short l = Some (h, t)`, so it represents the recursive case in which `l` is not short. Its other arguments are proofs of `choosePivot h t = (pivot, rest)` and `partition pivot rest = (lt, eq, gt)`, which represent the calls to these functions that were present in our previous implementation of `qs`. Lastly, it takes as arguments the proofs that `lt` and `gt` belong to the domain, which represent the recursive calls. Therefore, the constructor `Long` can be read as "`l` belongs to the domain if `lt` and `gt` belong to the domain".

Given such a domain predicate, the implementation of `qs'` (a helper function that we will use to implement the new version of `qs`) is trivial. Both `A` and `l` become implicit arguments and the new principal argument is `d : QSDom A l`. We pattern match on `d` and in the first case we sort the list `l` ad hoc, whereas in the second case we recursively sort the elements lesser and greater than the pivot and put them together with the pivot and the elements equal to the pivot in the middle. Note that we don't have to call `choosePivot` or `partition` – we can retrieve everything we need from the arguments of `Long`.

Having seen an example use of the method, let's restate it in more abstract terms: the idea is to represent a general recursive function of type `A -> B` as a function of type `forall a : A, D a -> B`, where `D : A -> Type` is the domain predicate whose elements represent the call graph of the principal argument, i.e. the tree of all the recursive calls that have to be performed in order to compute the result. To obtain the original function of type `A -> B` it suffices to prove that all `a : A` satisfy the domain predicate.

[4] `QSDom A` is in fact a type family, because its codomain is `Type`, but we'll ignore this little naming inconsistency. In a later section we will see a variant of the inductive domain method where the domain predicate really is a predicate.

In the literature, the inductive domain method is better known as the "Bove-Capretta method", but this name, derived from the surnames of its inventors, is neither very descriptive nor catchy, so we avoid it. It is also sometimes called the *ad-hoc predicate* method, but this name is even worse. The method and related ideas were first developed in a string of papers between 2001 and 2009 [61] [62] [63] [64] [65] [66] [67] [68] [69] in a form that vaguely resembles what we have seen in this section. In the meantime, [40] proposed a more complicated variant which makes use of sort `Prop` to improve the method's fitness for use with Coq's extraction mechanism – we will see it in section 4.5.

We have seen only the basic version of the inductive domain method because quicksort's recursion scheme is rather basic. The method is pretty malleable and can be accomodated to fit mutual recursion (i.e. two or more general recursive functions that call each other; see [63]), nested recursion (i.e. a situation where the result of a recursive call is the argument of another recursive call, like in McCarthy's famous f91 function [70]; see [62]) and possibly even higher-order recursion (i.e. a situation where recursive calls may be partially applied; see [66]).

The inductive domain method can be used not only to implement general recursive functions, but also to represent partial recursive functions [68]. If a partial recursive function terminates for some input, we can construct the corresponding call graph and use it to run the function's representation. Of course in general we won't be able to prove that a partial recursive function terminates for all inputs, so we won't be able to recover the desired partial function of type `A -> B` from its representation of type `forall a : A, D a -> B`. However, as a means of representing partial recursive functions, the inductive domain method has some problems, for example when one wants to compose partial functions. See [71] for the most recent take on partial recursive functions and computability in (homotopy) type theory.

Another application of the inductive domain method, even more general than representing partial functions, is extracting an executable function definition from a specification defined as an inductive relation (see [78]). Such a relation, when seen as a function, can not only be partial in the sense of not terminating for certain arguments, but can, on the one hand, be altogether undefined for certain arguments and, on the other hand, be nondeterministic, i.e. return more than one result for a given argument. In such cases the proof that an argument belongs to the domain not only certifies that the "function" is well-defined and terminating for this particular argument, but also encodes an evaluation strategy that determines which result should be returned in cases where more than one is possible. Of course in this case the same caveat as for ordinary partial functions applies, but the problems are likely a lot more severe. If one is working in a setting which allows quotient inductive-inductive types, a possibly much better solution would be to use the partiality monad described in [72].

4.2 Well-founded induction

We now proceed to prove termination of the new quicksort template. The perfect (and, in some sense, the only) way to do this is *well-founded induction.* A relation on type A is well-founded if all elements of A are accessible and an element of A is accessible if all elements below it are accessible.[5] Well-founded induction is then just structural induction on the proof of accessibility of an element of A. And conversely, structural induction is nothing more than just well-founded induction with the well-founded relation being "is a subterm of". This is why well-founded induction and structural induction are, in a sense, equivalent.

In classical mathematics well-founded induction also goes by the names of *strong induction*, *complete induction*, *Noetherian induction* and many others. There it is defined in one of two ways: the first definition states that a relation is well-founded when it has no infinite descending chains; the second says a relation on A is well-founded when all subsets of A have a minimal element. Classically all definitions of well-foundedness are equivalent, but from the point of view of constructive mathematics the classical ones are clearly inferior to the accessibility-based definition given in the previous paragraph, because the latter is constructive, whereas the former are not.

For example, the relation "is shorter than" on `list A`, which we will use in the listing below, is well-founded because all lists are accessible. This should be intuitively clear: the empty list is accessible because all shorter lists (of which there are none) are accessible; all singleton lists are accessible, because all shorter lists (i.e. just the empty list) are accessible; all lists of length two are accessible, because all lists of length 0 and 1 are accessible and so on.

```
Lemma QSDom_all :
  forall (A : QSArgs) (l : list A), QSDom A l.
Proof.
  intro A.
  apply well_founded_induction_type with (R := ltof _ (@length A)).
    apply well_founded_ltof.
    intros l IH. destruct (short l) as [[h t] |] eqn: Hshort.
      2: constructor; assumption.
      destruct (choosePivot h t) as [pivot rest] eqn: Hpivot;
      destruct (partition pivot rest) as [[lt eq] gt] eqn: Hpartition.
      econstructor 2; try eassumption.
        apply IH; red. apply le_lt_trans with (length rest).
          admit.
          admit.
        apply IH; red. apply le_lt_trans with (length rest).
```

[5] The definition of accessibility should be interpreted inductively.

```
        admit.
        admit.
Abort.
```

Our first attempt to prove termination is shown in the listing above. We proceed by well-founded induction on `l`, the well-founded relation being "is shorter than" described in the previous paragraph. The main part of the proof starts after we have convinced Coq that this relation really is well-founded (using the lemma `well_founded_ltof` from the standard library).

We split cases on whether `l` is short or not. If it is not, then `l` belongs to the domain thanks to the constructor `Short`. In the other case, we use the constructor `Long` and it suffices to prove that `lt` and `gt` belong to the domain. We consider only the `lt` case, because the other one is analogous.

To prove `QSDom A lt` we use the induction hypothesis `IH` and so we only need to prove that `lt` is shorter than `l`. Since `lt` was obtained by partitioning `rest`, it should be the case than `lt` is not longer than `rest`. We therefore reason by transitivity and it suffices to prove `length lt <= length rest` and `length rest < length l`.

As for the second inequality, `rest` was obtained from `h :: t` by choosing a pivot, so it should be of the same length as `t`. When it comes to `t`, it is returned by `short` along with `h` as evidence that `l` is not empty, so that `l = h :: t` (or, for more exotic implementations of `short`, `l` and `h :: t` may be just permutations) and therefore `t` is shorter than `l`.

And so we proceed by... well, we don't proceed anymore. We are stuck because we have no evidence to back up our reasoning from the previous one and a half paragraphs. The reason is simple: there's nothing in `QSArgs` guaranteeing that `short`, `choosePivot` and `partition` don't produce gibberish.

Formally verified algorithm concept/technique #9

When proving termination by well-founded induction, assume every property and lemma required for the proof to go through by adding them to the algorithm template.

But the beauty of mathematics in general and of our approach to algorithms in particular is that we can just assume everything we need and call it a day! Therefore it suffices to introduce a new variant of `QSArgs` which will provide us with all the lemmas we need – its definition is shown in the listing below.

```
Class TerminatingQSArgs : Type :=
{
    args :> QSArgs;

    short_len :
      forall {l : list T} {h : T} {t : list T},
        short l = Some (h, t) -> length t < length l;

    choosePivot_len :
      forall {h : T} {t : list T} {pivot : T} {rest : list T},
        choosePivot h t = (pivot, rest) ->
          length rest = length t;

    partition_len_lt :
      forall {pivot : T} {rest lt eq gt : list T},
        partition pivot rest = (lt, eq, gt) ->
          length lt <= length rest;

    partition_len_gt :
      forall {pivot : T} {rest lt eq gt : list T},
        partition pivot rest = (lt, eq, gt) ->
          length gt <= length rest;
}.

Coercion args : TerminatingQSArgs >-> QSArgs.
```

Coming up with the proper definition of `TerminatingQSArgs` is straightforward. Start with a class that has a single field which accomodates the original `QSArgs`. Don't forget to declare a coercion, so that elements of `TerminatingQSArgs` can be used in all places where an element of `QSArgs` is expected. Then attempt to prove termination. Each time you are stuck, augment the definition of `TerminatingQSArgs` with a lemma that will make the proof go through. After you are done, try to minimize the number of assumptions by deriving some of these lemmas from others.

```
Lemma QSDom_all :
  forall (A : TerminatingQSArgs) (l : list A), QSDom A l.
Proof.
  intro A.
  apply well_founded_induction_type with (R := ltof _ (@length A)).
    apply well_founded_ltof.
    intros l IH. destruct (short l) as [[h t] |] eqn: Hshort.
```

```
   2: constructor; assumption.
   destruct (choosePivot h t) as [pivot rest] eqn: Hpivot;
   destruct (partition pivot rest) as [[lt eq] gt] eqn: Hpartition.
   econstructor 2; try eassumption.
    apply IH; red. apply le_lt_trans with (length rest).
     apply (partition_len_lt Hpartition).
     rewrite (choosePivot_len Hpivot). apply (short_len Hshort).
    apply IH; red. apply le_lt_trans with (length rest).
     apply (partition_len_gt Hpartition).
     rewrite (choosePivot_len Hpivot). apply (short_len Hshort).
Defined.

Definition qs (A : TerminatingQSArgs) (l : list A) : list A :=
 qs' (QSDom_all A l).
```

Having all the lemmas we need, the proof proceeds exactly as described above. `qs`, our provably terminating implementation of a quicksort template, is a result of feeding the termination proof into `qs’`. Note that it is crucial to that we have ended the proof with the command `Defined` instead of `Qed`. If we didn't, we wouldn't be able to run `qs` – it would get instantly stuck. This is because the command `Qed` results in the definition being opaque, i.e. the body of the definition gets erased. This would prevent `qs` from computing anything, because the helper function `qs’` was defined by structural recursion on `d : QSDom A l`, which in the case of `qs` comes from the termination proof.

4.3 User experience: default implementation as a sanity check

In section 3.2 we learned an important sanity check we can easily perform: filling the template in to get a concrete algorithm. Carrying out that check turned out to be very beneficial – it might have happened, after all, that our template is so silly that it isn't possible to fill it in!

Our present situation is quite similar: our termination "proof" is not really a proof, but a combination of a bunch of unverified assumptions that we made when our first proof attempt failed. If we made a mistake in our initial reasoning, it could have lead us to making too strong or even contradictory[6] assumptions, which would render our algorithm template useless. We might be tempted to extend our previous

[6] Our assumptions can't be too weak, however – if they were, the termination proof wouldn't have gone through.

sanity check by providing a concrete termination proof for the concrete algorithm that it spawned. There is, however, a smarter thing to do.

Formally verified algorithm concept/technique #10

Make sure that users can run a default implementation of the algorithm without having to prove anything.

The rationale behind the technique is quite simple – it is a kind of *principle of least effort*. A typical user will want to see the algorithm's results and running time immediately, without having to fill in the template. If he does decide to fill it in, he will want to get a feel of his custom algorithm without having to invest his precious time in proving anything. If he does decide to prove termination, he will want to use his provably-terminating algorithm as a building block of some bigger development without having to prove it correct. Only when he wants to prove the whole thing correct should he be forced into providing a concrete correctness proof.

By using the above technique, we can hit two birds with one stone: instead of a concrete termination proof for a quicksort for sorting lists of naturals according to the standard order, we can define and prove termination of a default implementation of quicksort that works for any type `A` and any comparison function `p : A -> A -> bool`. The effort required is almost the same, but the result is much more generally applicable.

```
Instance QSA_default
  (A : Type) (p : A -> A -> bool) : QSArgs :=
{
    T := A;
    short l :=
      match l with
          | [] => None
          | h :: t => Some (h, t)
      end;
    adhoc _ := [];
    choosePivot h t := (h, t);
    partition pivot rest :=
      (filter (fun x => p x pivot) rest,
       [],
       filter (fun x => negb (p x pivot)) rest)
}.

#[refine]
Instance TQSA_default
  (A : Type) (p : A -> A -> bool) : TerminatingQSArgs :=
{
    args := QSA_default A p;
}.
Proof.
  all: cbn.
    destruct l; inversion 1; cbn. apply le_refl.
    inversion 1; subst. reflexivity.
    inversion 1; subst. apply len_filter.
    inversion 1; subst. apply len_filter.
Defined.

Compute
  qs
    (TQSA_default _ (fun l1 l2 => leb (length l1) (length l2)))
    [[1; 2; 3]; [2; 2]; []; [4; 4; 4; 42]].
(* ===> = [[]; [2; 2]; [1; 2; 3]; [4; 4; 4; 42]] *)
```

As we can see, the definition of `TQSA_default` is almost the same as that of `QSA_nat`. The proofs aren't very hard, so we have passed the sanity check – we now have a provably terminating quicksort template, together with a default implementation, and, thanks to the last check, it looks like it produces correct results. But before we move on to prove our template's correctness, let's see what we could have done differently with regards to the termination proof.

4.4 A slight variation on a theme

The first thing we could have done differently is the relation between the provably terminating template and the possibly nonterminating template. We have implemented these separately, with the termination checker turned off so that the latter gets accepted, but we could have chosen a different design in which `qs` is defined using the inductive domain method and it is the termination proof (i.e. `QSDom_all`) for which we turn off the termination checker. In this way we derive the possibly nonterminating version of `qs` from the template we used to define the provably terminating version. The code is shown in the listing below.

```
Unset Guard Checking.
Fixpoint QSDom_all
  (A : QSArgs) (l : list A) {struct l} : QSDom A l.
Proof.
  destruct (short l) as [[h t] |] eqn: Hshort.
```

```
    2: constructor; assumption.
    destruct (choosePivot h t) as [pivot rest] eqn: Hpivot,
             (partition pivot rest) as [[lt eq] gt] eqn: Hpartition.
      econstructor 2; try eassumption.
        apply QSDom_all.
        apply QSDom_all.
Defined.
Set Guard Checking.
```

It is not clear if we should choose this variant over the original one (and so we won't). On the one hand it seems to be "more uniform" (one template, two termination proofs) than the original (two templates, one termination proof), but on the other hand the definition of `qs` with termination checker turned off is much more readable than the above proof of `QSDom_all`, which is virtually the same, but implemented with tactics.

The only argument in favor of this variant that comes to my mind is that it would be simpler to prove it correct without having to prove termination – we have to do the proofs only once, whereas in the original design this would require two correctness proofs (one for the possibly nonterminating template, another one for the provably terminating template).

4.5 A more irrelevant variant of the inductive domain method

Another thing we could have done differently is the particular form of the inductive domain method we used. Our domain "predicate" was not really a predicate, but a type family (i.e. a function `A -> Type`). As long as we stay inside Coq this is perfectly fine, but it starts being problematic when we leave Coq by using the mechanism of *extraction*.

Extraction is a way to translate a program written in Coq into Haskell, OCaml or Scheme[7]. Because Coq is more of a bare typechecker than a full-blown programming language, it lacks any mechanism to interact with the outside world,[8] and thus extraction is often the only way to use Coq in a real-world project. In such a case it is essential to write Coq code that produces the highest quality extracted code.

[7] There are some attempts to support other target languages. See `https://github.com/pirapira/coq2rust` for a project which tries to provide extraction of Coq programs into Rust.

[8] However, there are some attempts at adding IO to Coq. See `http://coq.io/` and `https://github.com/Lysxia/coq-simple-io` for more information.

Very relevant to this goal is the distinction between the sorts[9] Type and Prop, one of the main differences between Coq's theory and Martin-Löf Type Theory which we discussed in the introduction. For the purpose of this section we can think that Type is the universe of computationally relevant program specifications that are inhabited by programs, whereas Prop is the universe of computationally irrelevant propositions that are inhabited by proofs.

A term is computationally relevant if its actual definition matters and irrelevant otherwise. Programs are relevant, because different programs may produce different results and we do care what the results actually look like. Proofs, on the other hand, are irrelevant, because we don't care *how* propositions are proved, only *that* they are proved.

This distinction between Type and Prop is realized in Coq by a rule which forbids defining a program by pattern matching on a proof of a proposition. One consequence of it is that for propositions we can assume the Axiom of Proof Irrelevance which asserts that any two proofs of a given proposition are equal, i.e. `forall (P : Prop) (p1 p2 : P), p1 = p2`. We cannot do so for types, however, because it would lead to contradiction (as, for example, the numbers 0 and 1 are different).

Another consequence is the behaviour of the extraction mechanism: because proofs of propositions are computationally irrelevant (and because none of the three target languages of extraction supports theorem proving), they are erased during extraction. Programs, on the other hand, are of course not erased, but translated into target language programs that compute the same results in the same way.

```
Extraction Language Haskell.

Extraction QSDom.
(* ===>
  data QSDom =
   | Short (List T)
   | Long (List T) T (List T) T (List T) (List T)
          (List T) (List T) QSDom QSDom
*)

Extraction QSDom_all.
(* ===>
  qSDom_all :: TerminatingQSArgs -> (List T) -> QSDom
  qSDom_all a =
    well_founded_induction_type (\l iH ->
      let {o = short (args a) l} in
      case o of {
```

[9] Sorts are also called universes.

```
       Some p ->
        case p of {
         Pair h t ->
          let {p0 = choosePivot (args a) h t} in
          case p0 of {
           Pair pivot rest ->
            let {p1 = partition (args a) pivot rest} in
            case p1 of {
             Pair p2 x ->
              case p2 of {
               Pair lt eq -> Long l h t pivot rest eq lt x
                (iH lt __) (iH x __)}}}};
       None -> Short l})
*)

Extraction qs'.
(* ===>
  qs' :: QSArgs -> (List T) -> QSDom -> List T
  qs' a l d =
    case d of {
     Short _ -> adhoc a l;
     Long _ _ _ pivot _ eq lt gt ltd gtd ->
      app (qs' a lt ltd) (Cons pivot (app eq (qs' a gt gtd)))}
*)
```

The above listing shows how `qs'`, its domain predicate `QSDom` and the termination proof `QSDom_all` are extracted when the codomain of `QSDom` is `Type`. There's quite a bit of code and that should worry us, because this code is an artifact stemming from the fact that we had to use the inductive domain method to convince Coq that `qs'` terminates. We certainly don't want the termination proof to leave Coq and slip into Haskell, because the proof is not necessary here – Haskell allows general recursion. In fact, we would like the extracted code to resemble our original implementation of `qs'` from chapter 3 with the termination checker turned off. But when we look at the result of extraction for `qs'`, we clearly see that this is not the case.

There is a variant of the inductive domain method (arguably somewhat more complex than the one we already know) which we can use to achieve this goal. Luckily, the aforementioned rule that disallows defining programs by pattern matching on proofs has an exception: we can pattern match on a proof if only a single case is possible.[10]

```
Inductive QSDom (A : QSArgs) : list A -> Prop :=
    | Short :
        forall l : list A, short l = None -> QSDom A l
    | Long :
        forall {l : list A},
          forall {h : A} {t : list A},
            short l = Some (h, t) ->
          forall (pivot : A) {rest : list A},
            choosePivot h t = (pivot, rest) ->
          forall (eq : list A) {lt gt : list A},
            partition pivot rest = (lt, eq, gt) ->
          QSDom A lt -> QSDom A gt -> QSDom A l.
```

To get what we want, it therefore suffices to change the codomain of QSDom into Prop and use the exception to the rule for making recursive calls. The former task, whose results are shown in the listing above, is a piece of cake, but the latter has some technical overhead – Coq is not smart enough to just let us match on the proof, so we will need to pack the pattern matching into special lemmas. One of these is shown in the listing below.

```
Lemma Long_inv_lt :
  forall
    {A : QSArgs} {l : list A} (d : QSDom A l),
      forall {h : A} {t : list A},
        short l = Some (h, t) ->
      forall {pivot : A} {rest : list A},
        choosePivot h t = (pivot, rest) ->
      forall {lt eq gt : list A},
        partition pivot rest = (lt, eq, gt) ->
          QSDom A lt.
Proof.
  destruct 1; intros.
    congruence.
    {
      rewrite H in H2; inversion H2; subst.
      rewrite H0 in H3; inversion H3; subst.
      rewrite H1 in H4; inversion H4; subst.
```

[10] Or, if no cases at all are possible – this is the canonical way to construct an element of any type A from a proof of False.

```
      exact d1.
    }
Defined.
```

Even though when matching on the proof of `QSDom A l` there are two possible cases, `Short` and `Long`, we will match it only when we know the list is not short, which reduces the number of possible cases to one. The above inversion lemma, which retrieves a proof of `QSDom A lt` from the proof of `QSDom A l`, formalizes this simple intuition. Its proof, however, is a bit involved because of all the equational hypotheses which need to be rewritten at some point. We also need an analogous inversion lemma for `QSDom A gt` – it is not shown in the listing because it is completely analogous. Note that we end the proof with `Defined`, not `Qed` – this is crucial, because Coq will need to be able to inspect the resulting proofterm.

```
Fixpoint qs'
  (A : QSArgs) (l : list A) (d : QSDom A l) {struct d} : list A :=
match
  short l as x return short l = x -> list A
with
| None => fun _ => adhoc l
| Some (h, t) => fun eq1 =>
    match
      choosePivot h t as y
    return
      choosePivot h t = y -> list A
    with
    | (pivot, rest) => fun eq2 =>
        match
          partition pivot rest as z
        return
          partition pivot rest = z -> list A
        with
        | (lt, eq, gt) => fun eq3 =>
            qs' A lt (Long_inv_lt d eq1 eq2 eq3)
            ++ pivot :: eq ++
            qs' A gt (Long_inv_gt d eq1 eq2 eq3)
        end eq_refl
    end eq_refl
end eq_refl.
```

Armed with both lemmas, we can now implement `qs'` – the implementation is shown in the listing above. The first important point to notice is that the principal argument is `d : QSDom A l`, as can be seen from the annotation {`struct d`}, but we don't match on `d` – the definition proceeds like for our initial `qs'` from chapter 3, when the termination checker was turned off.

The second important thing to notice is that, unlike previously, we don't just simply match on `short l` or use `let`s to pattern match on the results of `choosePivot` or `partition`. This time in each of these cases we have to use the full power of Coq's dependent pattern matching in order to generalize the return types. This is because we need to feed equations like `short l = Some (h, t)` into our inversion lemmas.

Doing this is simple, but the code looks a bit scary. For example, consider the fragment `match short l as x return short l = x -> list A with ... end eq_refl`. It works like this: we match on `short l` and use `x` to name the matched value (i.e. either `Some (h, t)` or `None`). Then we say that the return type of the whole match is `short l = x -> list A`, but after the `end` we have to apply the function returned by the `match` to `eq_refl`, because we want our final result to be just `list A`. This works, because Coq now knows that `eq_refl : short l = short l` is convertible with `eq_refl : short l = x`, because `x` in the branches reduces to either `None` or `Some (h, t)` and these are convertible with `short l` because a match was performed. We have to repeat a similar maneuver three times, once for each of the equations we need for our inversion lemmas. This way of generalizing `match` results with an equality is sometimes called "the convoy pattern". See chapter "MoreDep" from Adam Chlipala's CPDT [41] for a more thorough description.

The last important thing to notice is how the recursive calls look like. We use the lemma `Long_inv_lt` applied to the proof of `QSDom A l` and the three equations and it retrieves for us a proof of `QSDom A lt` which we use in the recursive call of `qs'`. Coq can see from the body of `Long_inv_lt` that it is a structurally smaller subterm of `d` and thus the recursive call is valid.

```
Definition qs (A : TerminatingQSArgs) (l : list A) : list A :=
  qs' A l (QSDom_all A l).

Compute qs (TQSA_default nat leb) [4; 3; 2; 1].
(* ===> = [1; 2; 3; 4] *)

Extraction qs'.
(* ===>
  qs' :: QSArgs -> (List T) -> List T
  qs' a l =
    case short a l of {
     Some p ->
      case p of {
       Pair h t ->
        case choosePivot a h t of {
         Pair pivot rest ->
          case partition a pivot rest of {
           Pair p0 gt ->
            case p0 of {
             Pair lt eq ->
              app (qs' a lt) (Cons pivot (app eq (qs' a gt)))}}}};
     None -> adhoc a l}
*)
```

The termination proof `QSDom_all` doesn't change at all – the fact that the codomain of `QSDom` is now `Prop` doesn't affect it in the slightest. The same can be said of `qs` and the default instance `TQSA_default` – their definitions are the same as before. When we run `qs` on an exemple list of naturals, can see the list also gets sorted just like before.

What changes drastically is the Haskell code extracted from `qs'`. Even though it will look very foreign to Haskellers (we would need to do some manual tweaking to make it look like idiomatic Haskell), it's clear that its structure is exactly what we wanted it to be – a mirror image of our original definition of `qs'` from chapter 3. We have achieved our goal.

Chapter 5

Verificatio ex nihilo

We are now ready to make the last step in our grand quest for formally verified algorithms – in this chapter, we will prove correctness of our template. Our approach to this task will be pretty similar to what we did when proving termination – we will assume all the properties and lemmas we need and then put them together to get the correctness proof, leaving the concrete proofs to the users of our algorithm. But as a conveniece for the users (and a sanity check for ourselves), we will provide a nice correctness proof for the default implementation.

5.1 One induction to rule them all – functional induction

Before we make the last step of our quest, we have to ask ourselves: once we have all the lemmas we need, how exactly are we going to use them? What will the outline of the proof look like? This question may seem trivial: we defined quicksort by recursion, so the proof will go by induction. But as everyone who has ever done a formal proof by induction in Coq knows, it's not that simple.

If the recursion scheme of the function is different from the induction scheme in the induction principle used for the proof, problems may arise. For example, the standard induction principle for natural numbers, which is concerned only with 0 and $n+1$, is sometimes not sufficient for proofs about the Fibonacci numbers, which are defined by recursion by considering the cases of 0, 1 and $n+2$.

A tentative answer to this problem may be to use well-founded induction, just as we did for the termination proof. But again, as anybody who has ever done a proof by well-founded induction in Coq knows, this would be a bit messy and inconvenient. Because well-founded induction is so general that it can be used for any function whatsoever, it doesn't fit any single recursion scheme particularly well and this produces some overhead.

Formally verified algorithm concept/technique #11

Define an inductive relation that represents the graph of your algorithm, prove that it really is the graph and use its induction principle to derive the functional induction principle of your algorithm.

Fortunately, there is a relatively cheap ultimate solution of the problem of inductive proofs: functional induction. This is a form of induction in which a tailor-made induction principle that perfectly fits the function's recursion scheme is used to prove all properties of the function. Thanks to this technique the proofs become pretty easy[1] and also easy to automate, and the derivation of the functional induction principle is very schematic and easy to automate itself. The above technique outlines how to accomplish it.

The idea behind functional induction is pretty simple. First, we want to separate the implementation of the function (its intensional aspect) from reasoning about its properties (its extensional aspect). Second, we want to capture the reasoning process in a single standalone lemma that we can reuse. To achieve this, we need to think about functions extensionally: two functions are equal whenever they give equal results for equal arguments.[2] Thus if we know all relationships between the inputs and outputs of a function, we know everything about the function. A relation that embodies all input-output relationships of a function is called a graph of that function. Therefore, we can deduce all properties of a function from its graph.

We say that a binary relation `R : A -> B -> Prop` is a graph of the function `f : A -> B` when `R a b <-> f a = b`. We called `R` "a" graph, but it can be proven that any function has exactly one graph, so we can just as well call it "the" graph. But even though all definitions of the graph of a given function are equivalent, not all of them are equivalently useful for our purposes – the usefulness of their intensional characteristics may vary.

For example, consider the function `div2` which divides a natural number by two, rounding down if the argument is odd. We can define its graph in two extensionally equivalent ways, either by saying that `R a b := a = 2 * b ∨ a = 1 + 2 * b` or with an inductive definition that closely follows the recursive definition of `div2` (which basically says that `div2 0 = 0`, `div2 1 = 0` and `div2 (S (S n)) = S (div2 n)`). Only the latter, inductively defined graph relation is useful for deriving the functional induction principle.

```
Inductive QSGraph (A : QSArgs) : list A -> list A -> Prop :=
    | ShortG :
        forall l : list A, short l = None -> QSGraph A l (adhoc l)
```

[1] Problems may arise only when we compose the function in question with others – in such a case even the functional induction principle may be not enough to guarantee a smooth proof.

[2] Sadly, this principle (called functional extensionality) is not provable in Coq and needs to be assumed as an axiom if we want to use it.

```
  | LongG :
      forall {l : list A},
        forall {h : A} {t : list A},
          short l = Some (h, t) ->
        forall (pivot : A) {rest : list A},
          choosePivot h t = (pivot, rest) ->
        forall (eq : list A) {lt gt : list A},
          partition pivot rest = (lt, eq, gt) ->
        forall lt' gt' : list A,
          QSGraph A lt lt' -> QSGraph A gt gt' ->
            QSGraph A l (lt' ++ pivot :: eq ++ gt').
```

The inductive graph relation for our quicksort template is shown in the listing above. It is very similar to the definition of QSDom and also closely matches our original definition of qs from chapter 3. The parameter A and the first index of type list A represent quicksort's arguments, whereas the second index of type list A represents its result. In the constructor LongG, the inductive occurrences of QSGraph mimic the recursive calls.

```
Lemma QSGraph_correct :
  forall (A : TerminatingQSArgs) (l : list A),
    QSGraph A l (qs A l).
Proof.
  intros. unfold qs. generalize (QSDom_all A l).
  induction q; cbn; econstructor; eassumption.
Qed.
```

The next step is to prove that QSGraph is "correct" with respect to qs. In less jargony words, this corresponds to one of the implications from the definition of a function's graph, namely `f a = b -> R a b`. The statement of the lemma is a bit different, however: it is the equivalent of saying `R a (f a)` instead. The proof is easy: we introduce assumptions to context, unfold the definition of qs and generalize the termination proof `QSDom_all A l` to a general `q : QSDom A l`. The rest is a simple induction on q, followed by using the appropriate constructor and feeding it with the arguments it wants.

```
Lemma QSGraph_det :
  forall (A : QSArgs) (l r1 r2 : list A),
    QSGraph A l r1 -> QSGraph A l r2 -> r1 = r2.
```

```
Proof.
  intros until 1. revert r2.
  induction H; inversion 1; subst.
    reflexivity.
    congruence.
    congruence.
    {
      rewrite H in H5. inversion H5; subst.
      rewrite H0 in H6. inversion H6; subst.
      rewrite H1 in H7; inversion H7; subst.
      repeat f_equal.
        apply IHQSGraph1. assumption.
        apply IHQSGraph2. assumption.
    }
Qed.

Lemma QSGraph_complete :
  forall (A : TerminatingQSArgs) (l r : list A),
    QSGraph A l r -> r = qs A l.
Proof.
  intros. apply QSGraph_det with l.
    assumption.
    apply QSGraph_correct.
Qed.
```

We proceed to proving that `QSGraph` is “complete” with respect to `qs`. This is the other implication from the definition of a function’s graph, namely `R a b -> f a = b`. But if we tried to prove it directly, it would turn out to be very hard – we would have to reason about `qs`, but we haven’t derived the functional induction principle yet! Luckily, there is a very easy way to get this done: since we already know that `QSGraph` is correct, it suffices to prove that it is also deterministic, i.e. that `R a b1 -> R a b2 -> b1 = b2`. If we combine correctness with being deterministic, we get completeness.

The proof of `QSGraph_det` is as follows: given `r1` and `r2`, two potential results for `l`, and the corresponding derivations `QSGraph A l r1` and `QSGraph A l r2`, we proceed by induction on the first derivation, generalizing the other assumptions as necessary to get the most general induction hypothesis, and then we invert the second derivation.

In case both derivations are of the form `ShortG` the proof is trivial. In case they were derived using different constructors, it’s an easy contradiction. The last case, when both were constructed using LongG, is just as easy, but a bit more involved – we have to perform various rewritings and inversions to convince Coq that using the inductive hypotheses is ok.

With the lemma at hand, proving completeness is trivial: first we use the lemma QSGraph_det and then in one case we use an assumption, whereas in the other we use the correctness lemma.

```
Lemma qs_ind :
  forall (A : TerminatingQSArgs) (P : list A -> list A -> Prop),
    (forall l : list A, short l = None -> P l (adhoc l)) ->
    (
      forall (l : list A) (h : A) (t : list A),
        short l = Some (h, t) ->
      forall (pivot : A) (rest : list A),
        choosePivot h t = (pivot, rest) ->
      forall lt eq gt : list A,
        partition pivot rest = (lt, eq, gt) ->
          P lt (qs A lt) -> P gt (qs A gt) ->
            P l (qs A lt ++ pivot :: eq ++ qs A gt)
    ) ->
      forall l : list A, P l (qs A l).
Proof.
  intros A P Hshort Hlong l.
  apply QSGraph_ind; intros.
    apply Hshort. assumption.
    apply QSGraph_complete in H2. apply QSGraph_complete in H4.
      subst. eapply Hlong; eassumption.
    apply QSGraph_correct.
Qed.
```

At last we can formulate the functional induction principle and prove it. The principle looks similar to the induction principle QSGraph_ind that Coq generated automatically for QSGraph. The differences are that we assume the induction hypothesis holds for qs A lt and qs A gt instead of lt' and gt' (which come from derivations of QSGraph A lt lt' and QSGraph A gt gt'). Another difference is that there is no mention of QSGraph in the conclusion.

The proof is straightforward. We introduce assumptions to context and apply the induction principle of QSGraph. In the first case we use one of the assumptions and in the second case we use the other one, with two uses of the completeness lemma to make everything typecheck. When asked for a proof of QSGraph A l (qs

A l), we use the correctness lemma.

```
Lemma qs_eq :
  forall (A : TerminatingQSArgs) (l : list A),
    qs A l =
    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.
Proof.
  intros A l. apply qs_ind; intros.
    rewrite H. reflexivity.
    rewrite H, H0, H1. reflexivity.
Qed.
```

The last thing to do is proving the recursive equation, shown in the listing above. In theory it should not be necessary – the functional induction principle should always suffice – but in practice it is sometimes handy to have it. The purpose of the recursive equation is to be able to "unfold" the function's definition, replacing it with the desired, clean body we wanted to have at the very beginning instead of the clumsy actual implementation, which involves hairy details like the termination proof.

The proof is very easy – we introduce assumptions into context and proceed by functional induction, using the principle we just derived. This needs to be done with apply, because induction doesn't understand our custom principle. In both cases we finish by rewriting some hypotheses. Note that the induction hypotheses were not necessary - a case analysis would be sufficient, but we don't have a "functional case analysis principle" and it's not worth to derive one.

There is also yet another principle for reasoning about recursive functions that is not worth deriving, but which is nonetheless very easy to use in our current setup – functional inversion. The need for functional inversion arises when we know the function's result and want to infer what arguments could have produced that result. Of course, when $f(x) = y$, the space of all possible xs that could have produced y is $f^{-1}(y)$, the preimage of y under f. But this is neither very useful nor what we usually want, particularly when dealing with recursive functions – often we only want to know some basic facts about x, like which constructor it was made with.

Functional inversion comes in handy in precisely these kinds of situations. Formulating a standalone functional inversion principle for a given function is a rather daunting task, wordy and clumsy (and it would most likely only obfuscate the matter at hand), but using such a principle is very simple. When we have a hypothesis `H : f x = y` and `R` is the inductively defined graph relation of `f`, we need to convert `H` to `H' : R x y` by using the correctness lemma of the graph and then perform an `inversion` on `H'`. This results in a kind of "taking one step back in the evaluation of `f`", which exposes all the possible forms of `x` that could have made `f x` compute to `y` directly (i.e. in one step).

Deriving the functional induction principle turned out to be pretty easy and it should be just as easy when applied with the variant of inductive domain method from section 4.5. Moreover, derivation of the principle can be easily automated (together with the the inductive domain method) in most situations, as we will see in section 5.3.

But not all functions are as tame as quicksort – some tougher classes of functions require slight modifications to the approach presented above. One such class are the functions that exhibit *nested recursion*, i.e. a situation in which the result of a recursive call is an argument of another recursive call. These functions are problematic because the domain predicate can't be defined in the usual way: whether an element belongs to the domain might depend on whether the function's result for this argument belongs to the domain. Luckily, we can deal with such functions by first defining the inductive graph relation and then using it to represent the recursive calls in the definition of the domain predicate and then proceed as usual.

Another not-so-tame class of recursive functions that do not easily submit to our technique for deriving the functional induction principle are functions that exhibit *higher-order recursion*, i.e. a situation in which the recursive calls are partially applied.[3] This time the problem is defining the inductive graph itself: we either have to define an additional inductive graph relation for each of the partially applied recursive calls[4] or at least an additional inductive graph relation for each function which takes the recursively defined function as argument. Either way this means a lot of additional work to be done, most of which is writing boring boilerplate code. In practice the situation is even worse than it sounds, because the automation we will see in section 5.3 often can't generate these definitions automatically.

Before we finish, let's try to trace the history of functional induction. The term originated as a name for a Coq tactic in [73] and it isn't widely known. Coq is most likely the only place where it is used – searching Google for "functional induction" returns mostly Coq-related papers and Coq documentation pages intermixed with results in which the words "functional" and "induction" appear together but have

[3] Coq can recognize basic functions of this kind (like `map` for a tree that can have a `list` of subtrees) as structurally recursive without any help from the user, but this relies on some heuristics in the termination checker and is not very reliable.

[4] This is pretty much equivalent to the inductive graph of a variant of the function in which the higher-order recursive calls were replaced by first-order helper functions.

unrelated meanings. The technique itself seems not to have a standard name and trying to look it up on Wikipedia is also hopeless.

The core idea, i.e. separating the concrete implementation of a function from reasoning about its input-output behaviour, was first described[5] by Burstall in [74], but this work is not very well-known. Since then it was independently reinvented many times by many people, for example in [75], [76] and [77], which shows that the idea is quite intuitive and natural. In the context of Coq, the technique derived from the idea arose slowly in works that dealt with automating the definition of and reasoning about general recursive functions and culminated as an important part of the `Function` command [78], a tool designed precisely for this purpose (although variants of the technique used there may differ from the one presented in this thesis).

5.2 Don't do this at home

Before we proceed to see the automation, we will try to explore another way of deriving the functional induction principle – a way which, at first sight, looks much easier and more appealing than the one we have already seen, but upon closer inspection leads down a rabbit hole. That way is to omit the inductive graph altogether and try to derive the functional induction principle from the induction principle of the domain predicate. The first step in such an attempt is to prove an induction principle for the helper function `qs'`, shown in the listing below.

```
Lemma qs'_ind :
  forall (A : TerminatingQSArgs) (P : list A -> list A -> Prop),
    (forall l : list A, short l = None -> P l (adhoc l)) ->
    (
      forall (l : list A) (h : A) (t : list A),
        short l = Some (h, t) ->
      forall (pivot : A) (rest : list A),
        choosePivot h t = (pivot, rest) ->
      forall
        (lt eq gt : list A)
        (ltd : QSDom A lt) (gtd : QSDom A gt) ,
          partition pivot rest = (lt, eq, gt) ->
            P lt (qs' ltd) -> P gt (qs' gtd) ->
              P l (qs' ltd ++ pivot :: eq ++ qs' gtd)
    ) ->
      forall (l : list A) (d : QSDom A l), P l (qs' d).
Proof.
  induction d; cbn.
```

[5] This is the earliest paper I could find.

```
    apply H. assumption.
    eapply H0; eassumption.
Qed.
```

Even though deriving an induction principle for the helper function is very easy, transfering the principle to `qs` is impossible. The reason is that the induction hypotheses for `qs` hold only for specific proofs of `QSDom A l` (namely `QSDom_all A lt` and `QSDom_all A gt`, where `lt` and `gt` are the list of elements smaller and greater than the pivot, respectively), whereas the induction hypotheses for `qs'` are required to hold for all possible proofs of `QSDom A lt` and `QSDom A gt`. Compare the above listing with the listing below, which shows an attempt at proving the functional induction principle for `qs`.

```
Lemma qs_ind_bad :
  forall (A : TerminatingQSArgs) (P : list A -> list A -> Prop),
    (forall l : list A, short l = None -> P l (adhoc l)) ->
    (
      forall (l : list A) (h : A) (t : list A),
        short l = Some (h, t) ->
      forall (pivot : A) (rest : list A),
        choosePivot h t = (pivot, rest) ->
      forall lt eq gt : list A,
        partition pivot rest = (lt, eq, gt) ->
          P lt (qs A lt) -> P gt (qs A gt) ->
            P l (qs A lt ++ pivot :: eq ++ qs A gt)
    ) ->
      forall l : list A, P l (qs A l).
Proof.
  unfold qs. intros A P Hshort Hlong l.
  apply qs'_ind.
    apply Hshort.
    intros.
Abort.
```

A smart Coq user could try to salvage the above proof by attempting to prove that any two `d : QSDom A l` are equal. This too, however, is impossible, as can be seen in the listing below (or rather, by going over the below listing interactively in CoqIDE). We start by induction on `d1` and then perform case analysis on `d2`. When they are constructed with different constructors, we have an easy contradiction (`short l` is both `None` and `Some` at the same time). When both are `Short`, we are required us to prove that any two proofs of `short l = None` are equal. This is possible, but pretty involved – techniques from Homotopy Type Theory are necessary[6] – and thus we skip it with the tactic `admit`.

```
Lemma isProp_QSDom :
  forall (A : TerminatingQSArgs) (l : list A) (d1 d2 : QSDom A l),
    d1 = d2.
Proof.
  induction d1; destruct d2.
    f_equal. admit.
    congruence.
    congruence.
      assert (h = h0) by congruence.
      assert (t = t0) by congruence.
      assert (pivot = pivot0) by congruence.
      assert (rest = rest0) by congruence.
      assert (lt = lt0) by congruence.
      assert (eq = eq0) by congruence.
      assert (gt = gt0) by congruence.
      subst. f_equal.
Abort.
```

The real problem is that in the last case, in which both `d1` and `d2` were made with the constructor `Long`, after establishing some initial equalities we have to prove that two proofs of `short l = Some (h, t)` are equal (and this is just the first of three similar goals). This time however the goal is impossible to achieve: Homotopy Type Theory tells us[7] that the properties of equality proofs in type `option (A * list A)` depend on the properties of equality proofs in type `A`. The bottom line is that in Coq, without assuming additional axioms we can't prove that an arbitrary type `A` satisfies UIP[8] and therefore there is no chance of proving that any two proofs of `short l = Some (h, t)` are equal.

Assuming that all types satisfy UIP is consistent with Coq, but an axiom-dependent way of deriving functional induction principles is clearly inferior to an axiom-free one. Similarly, the variant of inductive domain method which uses `Prop` as the codomain also isn't viable, as it would require us to assume Proof Irrelevance.

[6] The appropriate technique is called the *encode-decode method*. See sections 2.12 and 2.13 of the HoTT Book [30] for easy examples and section 8.1 for an application to the fundamental group of the circle. Also see [81] [82] for newer ideas that relate the encode-decode method to functional induction principles.

[7] See sections 2.5—2.14 of the HoTT Book. [30]

[8] UIP stands for *Uniqueness of Identity Proofs*. A type `A` satisfies UIP when for all `x y : A`, any two proofs `p q : x = y` are equal.

The ultimate moral of this section's story is that even though the approach to deriving functional induction principles based on inductive graph relations presented in the last section is quite boilerplate-heavy, there is no way around it... except for some automation![9]

5.3 Automating the boring stuff

When we are given a source function, i.e. a function defined the way we saw in section 3.1, without worrying too much about termination, we can automate generation of the boilerplate that sets up the inductive domain method and establishes the functional induction principle:

- To define the graph, repeat the source function's definition, with each `match` branch turned into a constructor, each `let` turned into a bunch of universally quantified variables together with the appropriate equality hypothesis, and each recursive call turned into an inductive occurrence of the relation being defined.

- To define the domain predicate, take the graph and erase the last index, which corresponds to erasing the function's result. Also change the codomain from `Prop` to `Type` (or not, but then automating the procedure gets a little harder).

- To define the auxiliary function (the one with the modified domain), repeat the target function's definition, with an additional argument corresponding to the proof of the domain predicate. Replace the original `match` with a `match` on this proof, the recursive calls with recursive calls on subproofs, etc.

- To define the terget function (i.e. the one equivalent to the source function, but with termination taken care of), feed the termination proof to the auxiliary function.

- To prove the target function correct with respect to the graph, an `induction` on the proof of the domain predicate together with `eauto` should suffice.

- To prove the graph deterministic, `induction` on the first hypothesis and `inversion` on the second. The hairy rewritings can (most likely) be automated with some `Ltac` code.

- To prove the graph complete, use the determinism and correctness lemmas.

- To derive the functional induction principle, repeat the graph's induction principle, with occurrences of the graph relation replaced with the property being

[9] Well, there is. If we need to do just a few quick proofs and don't care about separating the reasoning from the implementation, we can just directly use the domain predicate's induction principle.

proved. Prove it using the graph's induction principle and previously proved lemmas.

- To derive the recursive equation, an application of the functional induction principle together with `eauto` should suffice.

Luckily, we don't need to automate this process ourselves. It has already been done – the standard Coq tool for defining and reasoning about general recursive functions is the `Function` command [78] [79]. We won't describe its internal workings here – interested readers should refer to section 3 of [78], especially for an idea of how they differ from the approach presented above. Instead, we will content ourselves with a quick tutorial showing how to apply the tool to our terminating quicksort template.

```
Require Import Recdef.

Function qsf
  (A : TerminatingQSArgs) (l : list A) {measure length 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
          qsf A lt ++ pivot :: eq ++ qsf A gt
end.
Proof.
  intros; subst. apply le_lt_trans with (length rest).
    apply (partition_len_gt teq2).
    rewrite (choosePivot_len teq1). apply (short_len teq).
  intros; subst. apply le_lt_trans with (length rest).
    apply (partition_len_lt teq2).
    rewrite (choosePivot_len teq1). apply (short_len teq).
Defined.

Compute qsf (TQSA_default nat leb) [4; 3; 2; 1].
(* ===> = [1; 2; 3; 4] *)
```

To use the `Function` command we first need to `Require Import Recdef` (or `Require Import FunInd`). `Function` is not really a built-in Coq command, but more like a plugin, and without importing the appropriate module we will get a parse error when trying to use it.

The definition of our quicksort template is stunningly similar to the one from section 3.2. The only differences are that the definition starts with `Function` instead of `Fixpoint`, that we use `TerminatingQSArgs` instead of `QSArgs` (because we need to prove termination for the definition to be accepted by Coq) and that we use the annotation `measure length l` instead of `struct l`. The function body is exactly the same as back then and we don't need to clutter it with any auxiliary constructions.

The termination proof is much shorter than the one we saw back in section 4.2. This is in part because `Function` frees us from the bureaucracy of instantiating the well-founded induction principle ourselves – we only need to provide the annotation `measure length l` and all the details get inferred automatically.

Here `measure` means that we want to perform well-founded induction using a well-founded relation constructed by mapping its arguments to natural numbers and then using the standard order $<$ on natural numbers, `length` specifies which function we want to use for the mapping and `l` says the length of which list is going to decrease. This is exactly the same thing we did back insection 4.2 – internally it even uses the same function `ltof` and lemma `well_founded_ltof` that we used.

The remaining part of the proof is also much simpler than in section 4.2. When using `Function` we don't need anymore to manually perform `destructs`, apply the induction hypothesis or decide which constructor of the domain predicate to use when. The only proofs that are left to us are those establishing that the length of the argument descreases in the recursive calls. To sum up, the termination proof using `Function` is only 5 lines long, whereas the one from section 4.2 was 12 lines – we got rid of almost 60% of the work, all of which was just bookkeeping.

But we shouldn't be too happy: automation won't dispel all our problems with defining functions and proving termination, because the `Function` command can't handle some kinds of recursion. The offending classes of recursive functions are the same ones that give us trouble with manual definition using the inductive domain method, namely those that exhibit *nested recursion* and *higher-order recursion*. Two examples of such functions are shown in the listings below.

```
Fail Function weird_id (n : nat) {measure id n} : nat :=
match n with
    | 0 => 0
    | S n' => S (weird_id (weird_id n'))
end.
(* ===> The command has indeed failed with message:
        on expr : weird_id n' check_not_nested: failure weird_id *)
```

The first one, called `weird_id`, is a very weird way of defining the identity function on natural numbers (if this function were accepted, an ordinary induction on `n` would suffice to prove that it is indeed the identity). This function exhibits nested recursion because of the expression `weird_id (weird_id n')`: first we call `weird_id` with the argument `n'`, which is a structurally recursive call on a subterm of `n`, and everything is fine so far, but then we call `weird_id` with the argument `weird_id n'`, i.e. the result of the previous call, which is not a structurally recursive call. We annotate the function with `measure id n`, in order to argue about its termination using well-founded induction (when we measure a natural number with the identity function, the resulting well-founded relation is simply $<$). But this maneuver doesn't work and we fail miserably – because of nested recursion, the `Function` command literally refuses to even consider our function and gives us an error message.

```
Inductive Tr (A : Type) : Type :=
    | N : A -> list (Tr A) -> Tr A.

Arguments N {A} _ _.

Function tmap {A B : Type} (f : A -> B) (t : Tr A) : Tr B :=
match t with
    | N x l => N (f x) (map (tmap f) l)
end.
(* ===> tmap is defined
        tmap is recursively defined (guarded on 4th argument)
        Cannot define graph(s) for tmap [...]
        Cannot build inversion information [...] *)
```

The second function, called `tmap`, is way of mapping its argument `f` over every node of a non-empty tree with finite branching. This function exhibits higher-order recursion because of the expression `map (tmap f) l`: the recursive call to `tmap` is partial, i.e. not fully applied, and in this form it is passed to `map` as argument. This scheme of recursion is less problematic than nested recursion and therefore this time the `Function` command doesn't refuse to cooperate. Moreover, we don't even need to argue about termination – the function is accepted after passing the ordinary guardedness check, which means that Coq considers it structurally recursive. But there's still a huge problem – after the function is defined, we get two warning messages telling us that `Function` couldn't generate definition of the graph relation and that it couldn't generate the functional inversion principle. And without the graph relation, of course, we don't get any automation at all.

But not all hope is lost! There exists another, less popular but newer Coq plugin that can assist us with reasoning about termination and derive the functional induction principle for us, called `Equations` [80]. Its internal workings are vastly different from those of `Function` (and it also has a very distinct, Haskell-like syntax), but the end effect is pretty similar. The strength of `Equations` is that it seems to handle nested recursion just fine – in the case of the function `weird_id`, we don't even need to argue about termination, because `Equations` sees through it immediately. On the other hand, when it comes to higher-order recursion, we are still fundamentally out of luck.

To sum up, all of the boilerplate that we saw in sections 4.1 and 5.1 can be easily automated using either the `Function` command or the `Equations` package, provided that we avoid higher-order recursion and, in case of `Function`, also nested recursion. For these two kinds of recursion schemes, the most robust way of defining functions, proving termination and deriving the functional induction principle is to manually apply the inductive domain method. However, since simple examples of higher-order recursion are considered structurally recursive by Coq, it may be a good alternative idea to just use ordinary induction for the proofs.

5.4 Hole-driven development and proof by admission

We have defined our quicksort template, proved it terminating and derived the functional induction principle, which will allow us to prove it correct. But before we start proving, we will spend some time to put our approaches to software engineering and proof engineering[10] in a wider context.

We started our development by giving a specification. This is reminiscent of test-driven development, except that tests are replaced by formal proofs. Such an approach has been dubbed *proof-driven development* [83], but we don't attach any emotional value to this label.

A much more important tenet of our approach could be called *hole-driven development.*[11] The idea here is that we construct programs in stages, by refining earlier, incomplete versions of the program. Each missing piece of the program is marked with a hole and refining an incomplete program is done by filling (i.e. replacing) the hole with a program of the appropriate type, which can also possibly be incomplete and contain further holes. This process is iterated until we reach a complete program.

The most well-known languages that support holes are Agda, Idris and Haskell [84]. In Idris, a hole looks like `?h : A`, where `h` is the name of the hole and `A` is its type – holes are intrinsically typed, just like all other terms in the language. Coq, sadly, does not support holes, but it supports *existential variables* [85] which

[10] "Proof engineering" is a fancy name for proving theorems using a proof assistant like Coq. It is usually applied to large and complex proofs, just like the term "software engineering" is usually applied to large and complex software systems and not to hello-world-sized programs.

[11] Sadly, there is no solid literature on the topic, mostly just blog posts that introduce the idea.

are somewhat similar. When we combine existential variables with tactic-based definitions[12] using the tactic `refine`, we got something that vaguely resembles the hole-driven development available in Idris.

But wait! What we did when defining the quicksort template looked nothing like the hole-driven development approach that we just described. That's right, and for two separate reasons. First, we used holes quite implicitly, because quicksort is too easy an algorithm to pose serious problems when defining it. Second and more importantly, we filled our implicit holes with subprograms that we abstracted over (by putting them into the definition of `QSArgs)`, effectively postponing the hole-filling until the time when a user actually wants to run the algorithm.

What we did when defining the terminating quicksort template also didn't look like hole-driven development as described above, but it was. Recall that in section 4.2, after having defined the domain predicate, we unsuccessfully tried to prove by well-founded induction that all lists belong to the domain. The reason of our failure was that some termination-related properties of our subprograms were lacking and thus our proof had a few gaps.[13] We then assumed that the desired properties indeed hold (by putting them into the definition of `TerminatingQSArgs`), and used them to fill the gaps in our proof.

Formally verified algorithm concept/technique #12

When proving correctness of the algorithm template (or when proving anything else, really), use the tactic `admit` (and/or `cut`) for any subgoals that you can't immediately prove. After you finish the proof, go back and turn these `admit`ted subgoals into lemmas or, when they are not provable, into additional assumptions of your proof.

We see that our approach, based on filling holes with things we abstract over, is very powerful and lends itself well to both writing programs and proving theorems. But we don't need to treat all holes this way – we can still, of course, fill holes with concrete terms, like in the traditional hole-driven development. This is indeed how we will prove our quicksort correct, as shown in the above concept/technique: we will reason by functional induction and use the tactic `admit` for any parts of the theorems that we can't prove right away. Then we will step back and try to turn these `admit`ted gaps into lemmas. When it's clear we couldn't possibly prove some lemma, we will simply assume that it holds (by putting it into the definition of

[12] Coq's tactics are not just for proving! Because they are just a different way of constructing terms, they can be used for defining programs too (we only need to remember to close such definitions with `Defined` instead of `Qed`).

[13] Interestingly, mathematicians tend to use the word "gap" for a missing part of a proof. This is very similar to our use of "hole" for a missing part of a program. But there's also a significant difference between gaps and holes: holes in programs come into existence by being deliberately put there, whereas in the mathematical world, gaps usually arise from mistakes and errors. However, see [86] [87] [88] for more on proof gaps which are intentional or acceptable.

VerifiedQSArgs, the ultimate incarnation of QSArgs which allows the correctness proof to go through) and use this assumption to fill the gap in the main proof.

The last thing that remains to be done is to name the above technique for easier reference in the future. It could be called "verificatio ex nihilo", like in the title of this chapter, but Latin names aren't too catchy in computer science. Another possible name would be "gap-driven development", referring to gaps we leave in proofs to be filled later, but the technique is more about proof engineering than software development. Therefore, from now on we will refer to this technique as *proof by admission*, after the tactic admit.[14]

```
Theorem Permutation_qsf_first_try :
  forall (A : TerminatingQSArgs) (l : list A),
    Permutation l (qsf A l).
Proof.
  intros. functional induction (qsf A l).
    admit.
    eapply Permutation_trans with (lt ++ pivot :: eq ++ gt).
      admit.
      apply Permutation_app.
        assumption.
        constructor. apply Permutation_app.
          reflexivity.
          assumption.
Admitted.
```

Recall that our specification consists of two properties. First, the output of the sorting function has to be a permutation of the input. Second, it has to be sorted. These properties are separate, but it will turn out that we need the first one in order to prove the second, so it's better to deal with Permutation first.

We start by introducing everything into the context and performing functional induction. This results in two cases. In the first case, the list is short and so it gets sorted with adhoc. Here we encounter the first opportunity to admit something without proof – because adhoc is abstract, we have no way to prove that its output is a Permutation of l.

In the second case the list is not short, and the goal is Permutation l (qsf A lt ++ pivot :: eq ++ qsf A gt). From our induction hypotheses we know that qsf A lt is a Permutation of lt and similarly for gt, so it's best to reason by transitivity. This again yields two subgoals.

[14] Proof by admission can also be seen as the formalized counterpart of a very powerful proof technique often used by mathematicians, called *proof by handwaving*.

The second one is easy and boils down mostly to the fact that `Permutation` is preserved by list concatenation, followed by uses of induction hypotheses. However the first subgoal, which reads `Permutation l (lt ++ pivot :: eq ++ gt)`, is hopeless. We know that `lt`, `eq` and `gt` are outputs of `partition`, whose inputs are the outputs of `choosePivot`, whose inputs are the outputs of `short`, but there is nothing certifiying that the outputs of these functions are `Permutation`s of their inputs. This is the perfect place to use `admit` and finish the proof (with the command `Admitted`).

```
Theorem Sorted_qsf_first_try :
  forall (A : TerminatingQSArgs) (R : A -> A -> Prop) (l : list A),
    Sorted R (qsf A l).
Proof.
  intros. functional induction (qsf A l).
    admit.
    cut (Sorted R (qsf A lt) /\
         Forall (fun x => R pivot x) (qsf A lt) /\
         Sorted R (pivot :: eq ++ qsf A gt)).
      admit.
      repeat split.
        assumption.
        apply (Permutation_Forall (Permutation_qsf_first_try A lt)).
          admit.
        cut (Sorted R (pivot :: eq) /\
             Forall (fun x => R pivot x) (qsf A gt)).
          admit.
          split.
            cut (Forall (fun x => x = pivot) eq).
              admit.
              admit.
            apply (Permutation_Forall (Permutation_qsf_first_try A gt)).
              admit.
Admitted.
```

The proof of the second property starts the same as the first one: we introduce everything into the context and use functional induction, which again yields two subgoals. In the first, we sort `l` using `adhoc`, but we don't actually know that `adhoc` is a valid sorting function! Thus comes the first `admit`.

The second subgoal reads `Sorted R (qsf A lt ++ pivot :: eq ++ qsf A gt)`. Intuitively, it should be pretty easy: everything in `qsf A lt` is less than the pivot and everything in `qsf A gt` is greater than the pivot, so it suffices to somehow break the goal into subgoals along these lines... and thus we encounter another hurdle, because there's nothing we can use. We nonetheless allow ourselves to reason in this way by using the tactic `cut` and `admit`ting the missing lemma – we will prove it later.

We're thus left with three subgoals. In the first, we use an induction hypothesis to prove that `qsf A lt` is sorted. In the second subgoal, we need to prove that all elements of `qsf A lt` are less than the pivot. This should be quite easy – we already know that the output of `qsf` is a permutation of its input, and we know that `Forall` respects `Permutation`, thanks to a lemma from the standard library called `Permutation_Forall`. But the proof won't go through – we're missing the fact that all elements of `lt` are less than the pivot, and so we have to `admit`.

In the third subgoal, we are left with `Sorted R (pivot :: eq ++ qsf A gt)`. This should be provable with reasoning similar to the previous one, namely it should suffice to know that all elements of `qsf A gt` are greater than the pivot. But yet again, we lack this knowledge and so we use `cut` together with `admit`.

This leaves us with two more subgoals. First, we have to prove `Sorted R (pivot :: eq)`. It looks quite easy, as all elements of `eq` are supposed to be equal to `pivot`, but we know neither that nor have we any evidence for the fact that a list in which all elements are equal is sorted. This results in another `cut` and two more `admit`s. The last subgoal, where we have to show that all elements of `qsf A gt` are greater than the `pivot`, is completely analogous to what we described two paragraphs above, and so we have to `admit` it too.

We are now done with the proofs (modulo the gaps) and well-acquainted with proof by admission. The technique can be used not only for algorithm correctness, but for pretty much any kind of proof which is tedious, but not complex. The difference between these two is whether the outline of the proof is known.

Tedious proofs have a known outline – in our case it was functional induction, but it can also be ordinary induction, coinduction, rewriting, or anything else – but the details might be gory. Complex proofs, on the other hand, have no known outline – an example could be the normalization of simply typed lambda calculus. If you don't know how it's usually proved and have to invent the proof yourself, it turns out to be harder than filling some gaps with lemmas and assumptions!

```
Class VerifiedQSArgs : Type :=
{
    T : TerminatingQSArgs;

    Permutation_adhoc :
      forall {l : list T},
        short l = None ->
          Permutation l (adhoc l);
    Permutation_short :
      forall {l : list T} {h : T} {t : list T},
        short l = Some (h, t) ->
          Permutation l (h :: t);

    Permutation_choosePivot :
      forall {h : T} {t : list T} {pivot : T} {rest : list T},
        choosePivot h t = (pivot, rest) ->
          Permutation (h :: t) (pivot :: rest);

    Permutation_partition :
      forall {pivot : T} {rest lt eq gt : list T},
        partition pivot rest = (lt, eq, gt) ->
          Permutation (pivot :: rest) (lt ++ pivot :: eq ++ gt);

    R : T -> T -> Prop;

    R_refl :
      forall x : T, R x x;

    Sorted_adhoc :
      forall {l : list T},
        short l = None ->
          Sorted R (adhoc l);

    partition_pivot_lt :
      forall {pivot : T} {rest lt eq gt : list T},
        partition pivot rest = (lt, eq, gt) ->
          Forall (fun x => R x pivot) lt;

    partition_pivot_eq :
      forall {pivot : T} {rest lt eq gt : list T},
        partition pivot rest = (lt, eq, gt) ->
          Forall (fun x => pivot = x) eq;

    partition_pivot_gt :
      forall {pivot : T} {rest lt eq gt : list T},
        partition pivot rest = (lt, eq, gt) ->
          Forall (fun x => R pivot x) gt;
}.

Coercion T : VerifiedQSArgs >-> TerminatingQSArgs.
Coercion R : VerifiedQSArgs >-> Funclass.
```

We can now progress to fill the gaps, of which there are two kinds: lemmas to be proved and additional properties to be assumed. Thankfully, they are quite easy to distinguish: in our proofs, uses of the tactic `cut` (or, to be more precise, `cut` followed by `admit`) correspond to lemmas and all other uses of `admit` correspond to additional assumptions we need to make (some correspond to more than one assumption).

We will start with the additional assumptions we need, collecting them into a new class called `VerifiedQSArgs`. We want this class to be kind of an "extension" of the class `TerminatingQSArgs`, similarly to how we defined `TerminatingQSArgs` as an extension of `QSArgs`. Therefore, the first field of the class, called `T`, is an instance of `TerminatingQSArgs`.

The four fields that follow correspond to the two `admits` from the proof of `Permutation_qsf_first_try`. They are evidence for the fact that all components of our algorithm template return permutations of their inputs and thus, when put together, are evidence that the algorithm as a whole also returns a permutation of its input.

Next comes a binary relation `R` on (the underlying type of) `T`, which is the type of elements that are being sorted. We will need it to even state what we mean when we say that `qsf` is a sorting function. It may come as a surprise to you, but we came so far in our formalization without mentioning any kind of order at all! If you are a careful reader, you might have noticed that when we tried to prove `Sorted_qsf_first_try`, we actually had to assume that the order relation is given as an assumption.

The next field of `VerifiedQSArgs` ensures that this order relation `R` is reflexive. This fact doesn't correspond to any of our `admits`, but we would have discovered that it is needed when trying to prove that a list of all elements equal to the pivot is sorted.

The remaining fields correspond to `admits` from the proof of `Sorted_qsf_first_try`. The first `admit` was used because we didn't know that `adhoc` is a sorting function and the remaining ones that did not immediately follow uses of `cut` were used because we didn't know what is the relation between the pivot and the outputs of `partition`.

At the end, we declare some coercions – this is the simplest way to simulate inheritance/subtyping in Coq. First, we declare `T` to be a coercion from `VerifiedQSArgs` to `TerminatingQSArgs`, which ensures that we can use an instance of the former class wherever an instance of the latter class is expected. Second, we declare R to be a coercion from VerifiedQSArgs to Funclass, which means that we can use an instance x : VerifiedQSArgs to refer to the order relation R x.

```
Global Hint Constructors Sorted : core.

Lemma Sorted_app :
  forall
    {A : Type} {R : A -> A -> Prop}
    {pivot : A} {lt rest : list A},
      Forall (fun x => R x pivot) lt ->
      Sorted R lt ->
      Sorted R (pivot :: rest) ->
        Sorted R (lt ++ pivot :: rest).
Proof.
  induction 1; try inversion 1; subst; cbn; auto.
Qed.

Lemma Sorted_app_eq :
  forall
    {A : Type} {R : A -> A -> Prop}
    {pivot : A} {eq gt : list A},
      Forall (fun x => pivot = x) eq ->
      Forall (fun x => R pivot x) gt ->
      (forall x : A, R x x) ->
      Sorted R gt ->
        Sorted R (pivot :: eq ++ gt).
Proof.
  induction 1; cbn; inversion 1; subst; auto.
Qed.
```

To fill the remaining gaps, we will need some lemmas, shown in the listing above. Note that, even though we used the tactic cut three times, only two lemmas suffice – the last two cuts got merged and appear as the second lemma. The lemmas aren't terribly interesting, so we will not discuss them at length. Note only that we don't use VerifiedQSArgs to state these lemmas – this is because it's easier for automation to find the proofs for us when all premises are explicit. All automation that is actually needed is on the first line – we need to tell the tactic auto that it might need to use constructors of Sorted.

```
Theorem Permutation_qsf :
  forall (A : VerifiedQSArgs) (l : list A),
    Permutation l (qsf A l).
Proof.
  intros.
  functional induction (qsf A l).
    eapply Permutation_adhoc. assumption.
    {
      apply Permutation_short in e.
      apply Permutation_choosePivot in e0.
      apply Permutation_partition in e1.
      rewrite e, e0, e1.
      apply Permutation_app.
        assumption.
        constructor. apply Permutation_app.
          reflexivity.
          assumption.
    }
Qed.

Theorem Sorted_qsf :
  forall (A : VerifiedQSArgs) (l : list A),
    Sorted A (qsf A l).
Proof.
  intros.
  functional induction (qsf A l).
    apply Sorted_adhoc. assumption.
    apply Sorted_app.
      apply (Permutation_Forall (Permutation_qsf A lt)).
        eapply partition_pivot_lt; eassumption.
      assumption.
      apply Sorted_app_eq.
        eapply partition_pivot_eq; eassumption.
        apply (Permutation_Forall (Permutation_qsf A gt)).
          eapply partition_pivot_gt; eassumption.
        apply R_refl.
        assumption.
Qed.

#[refine]
Instance Sort_qsf
  (A : VerifiedQSArgs) : Sort A (qsf A) := {}.
Proof.
  apply Sorted_qsf.
  apply Permutation_qsf.
Defined.
```

The finished correctness proofs are shown in the listing above. They are not terribly different from the initial versions. The only exceptions are that they are now stated using `VerifiedQSArgs`, which makes them shorter, and they no longer contain any gaps. Also, the structure of the second proof changed a bit to reflect the fact that only two lemmas are needed now (as opposed to three `cut`s previously).

5.5 Sanity check: default correctness proofs

We are almost done with our formally correct quicksort. The last step that separates us from proclaiming total success is yet another sanity check – we have to provide concrete correctness proofs for the default implementation from section 4.3.

```
#[refine]
Instance VQSA_default
  {A : Type} (p : A -> A -> bool)
  (Hrefl : forall x : A, p x x = true)
  (Htotal : forall x y : A, p x y = false -> p y x = true)
  : VerifiedQSArgs :=
{
    T := TQSA_default A p;
    R x y := p x y = true;
    R_refl := Hrefl
}.
Proof.
  destruct l; inversion 1. constructor.
  destruct l; inversion 1. reflexivity.
  inversion 1. reflexivity.
  inversion 1; subst; clear H. Print VerifiedQSArgs.
    induction rest as [| h t]; cbn in *.
      reflexivity.
      destruct (p h pivot); cbn.
        rewrite perm_swap, <- IHt; reflexivity.
        {
          rewrite Permutation_app_comm.
          rewrite <- !app_comm_cons.
          rewrite perm_swap.
          rewrite (@perm_swap _ h pivot).
          constructor.
          rewrite app_comm_cons.
          rewrite Permutation_app_comm.
          assumption.
        }
  constructor.
  inversion 1; subst; clear H.
    induction rest as [| h t]; cbn.
      constructor.
      destruct (p h pivot) eqn: Hp.
        constructor; assumption.
        assumption.
  inversion 1; subst; clear H. constructor.
  inversion 1; subst; clear H.
    induction rest as [| h t]; cbn.
      constructor.
      destruct (p h pivot) eqn: Hp; cbn.
        assumption.
        constructor.
          apply Htotal in Hp. assumption.
          assumption.
Defined.
```

Our concrete proofs are shown in the listing above. We want to prove that `TQSA_default`, the default implementation we defined in section 4.3, is correct. To do this, we define an instance of `VerifiedQSArgs` that extends `TQSA_default`. Because the latter is parameterized with a comparison function `p : A -> A -> bool`, we need the same parameters for `VQSA_default`.

We can then define the order relation with respect to which the output will be sorted as `R x y := p x y = true` – we are esentially turning our comparison function into a relation. Next, we need to provide a proof that our relation is reflexive, but functions of type `A -> A -> bool` in general don't need to have this property, so we need to assume it. Last but not least, it only makes sense to sort using a comparison function which guarantees that the order is total, i.e. `p x y = true` or `p y x = true`, which we express in a more convenient way as `p x y = false -> p y x = true`. Interestingly, we didn't need to assume totality during our proof by admission, but it turns out to be needed for one of the concrete proofs.

The proofs themselves are not terribly interesting, so we will not describe them in detail. In most cases it suffices to reason by inversion on the first hypothesis (which is always an equation like `short l = None`). If this is not enough, we reason by induction on the input list. When proving Permutation_partition, we need to do some rewriting with standard library lemmas concerning Permutation. In the last proof we need to use the hypothesis Htotal at the right moment.

Formally verified algorithm concept/technique #13
If your "concrete" correctness proof for the default implementation is actually not that concrete, i.e. if it is parameterized with some proofs, make sure it's possible to fill them in and get something useful.

We are now done with the correctness proofs for the default implementation, but we shouldn't feel completely satisfied with them. They were, after all, supposed to be concrete, whereas they turned out to be parameterized by proofs of reflexivity and totality. This isn't very dangerous, because finding an order which is reflexive and total isn't very hard, but in general it might be the case that our assumptions are contradictory and it isn't possible to retrieve a concrete verified algorithm from our template. The solution to this problem, suggested by the above concept/technique, is the same as before: we should try to instantiate our template to get something concrete out of it. In our case, let's try to derive a formally verified counterpart of the quicksort on naturals from section 3.2.

```
Lemma leb_total :
  forall x y : nat, leb x y = false -> leb y x = true.
Proof.
  intros.
  apply leb_correct.
  apply leb_complete_conv in H.
  lia.
Qed.

Compute
  qsf (VQSA_default leb Nat.leb_refl leb_total) [1; 2; 666; 42; 0].
(* ===> = [0; 1; 2; 42; 666] *)
```

We succeed, and so our formally verified quicksort is really correct. Well done!

5.6 User experience: type-driven development

Before we conclude, we will take our time to put all our sanity checks into a broader perspective. We will be especially interested in how they are related to hole-driven development and proof by admission as described in section 5.4.

The crucial idea to be mentioned here is *type-driven development* [89], sometimes also called *type-first development* [90] or *typeful programming* [91]. Type-driven development is, as the name suggests, a way of using the type system to guide the programmer towards an implementation of the desired program. It can be contrasted with the usual way of utilizing the type system, in which the programmer writes the program first and only after finishing he asks the typechecker to help him find errors in the program.

Type-driven development is championed by Idris and described in the book [89] by the language's creator, Edwin Brady. There it is realised by a kind of interaction between the programmer and the language's tooling, where the programmer talks to the system by writing types and program fragments, and the system responds by synthesizing obvious programs that fit the given type (which is reminiscent of Coq's proof automation), refining the program (e.g. performing case splits) or providing the programmer with suggestions when there are more possible implementations, and typechecking the program fragments provided by the user.

In Coq we can also use this approach as described above, at least as long as we stay in the proof mode. Outside the proof mode, support for type-driven development seems missing, but it's still possible to practice it, even though without tooling support. In fact, what we did in section 3.2, when we defined the default concrete implementation of our algorithm, was pretty similar in nature to type-driven development. The most noticeable difference is that in Idris, type-driven development is usually practiced using precise, dependently typed specifications, whereas our quicksort template consists only of simply typed components.

Formally verified algorithm concept/technique #14

To ensure that your formally verified algorithm template provides the best user experience, use type-driven development to define and prove correct another (preferably quite different) default/concrete implementation.

However, we (and the users of our algorithm template) are not confined to the simply typed `QSArgs` – we also have additional guidance from the termination properties in `TerminatingQSArgs` and correctness properties in `VerifiedQSArgs`. The above concept/technique suggests that we should consider them all at once, instead of defining the concrete implementation in stages. What we end up with in such a case is precisely just an exercise in type-driven development – as soon as we define any term of the appropriate type (i.e. `VerifiedQSArgs`), we are done.

So, it turns out most of the sanity checks we did and most of our user experience quality assurance can be summarized as "do some type-driven development and make sure it goes well". In the end, if we were still worried about the quality of our work, we could do some more type-driven development to define another concrete or default implementation and see how it goes. We won't do this, however, as quicksort is too simple an algorithm to be worth putting so much effort into – our concrete correctness proofs from the previous section are the ultimate evidence that we did a good job.

To sum up our findings from the present chapter: hole-driven development, together with our idea of abstracting some holes away, naturally leads to programs that can be proven correct using proof by admission, and then finally we can recover concrete, fully correct implementations using type-driven development. These three techniques are intrinsically linked, yielding a perfect harmony.

Chapter 6

Conclusion

In this thesis, we have described concepts and techniques useful for implementing and verifying functional algorithms. We worked in Coq, but our ideas readily transfer to other dependently typed languages, like Agda, Idris or F*. Some of the techniques (mostly those concerned with specification and implementation of the abstract template) are probably also useful in languages with weaker type systems, like Haskell, OCaml or F#.

Too long, didn't read! We have presented 15 concepts/techniques in total (numbered 0 through 14), but they neither capture everything that was described in this thesis, nor are they really unique. They also don't show the big picture – in fact, they are better thought of as subitems of the following five-item master plan which shows the steps that lead from a problem to a formally verified, user-extensible algorithm that solves it.

- Find a good specification of the problem.

  #0: Good means abstract and easy to use.

  #1: Good specification determines a unique object.

  #2: Bad specifications can possibly be improved with defunctionalization.

  #3: Sometimes better specifications can be found by focusing on a different aspect of the problem.

- Find an algorithm that solves the problem.

- Implement a template of the algorithm that abstracts over the exact details.

  #4: While implementing the template, ignore termination at first.

  #6: Types of components of your template should contain enough evidence, but not too much!

  #7: When dealing with many templates at once, make shared components into parameters. In all other cases, prefer bundled classes.

#8: Use the Inductive Domain Method to define a better, provably terminating algorithm template.

- Prove termination and correctness of the algorithm template.

  #12: Use the technique of Proof by Admission.

  #9: Outline of termination proof: well-founded induction.

  #11: Outline of correctness proof: functional induction.

- Provide a concrete default implementation together with all the proofs.

  #5: Provide a default implementation.

  #10: Make sure that a default implementation can be run without any proof obligations, and that a provably terminating implementation can be run without having to prove correctness.

  #13: If your default implementation or its proofs are too abstract, provide a more concrete version.

  #14: When looking for default and concrete implementations, use type-driven development.

Technical details. The source code of the most up-to-date version of this thesis can be found at `https://github.com/wkolowski/coq-algs/tree/master/Thesis`, together with all code snippets and a script that compiles the thesis to PDF (it requires latexmk and pygments, particularly the package minted). The thesis was born out of my side-project, which can be found at `https://github.com/wkolowski/coq-algs`.

Backstory. That side-project of mine had two goals: first, formalize some data structures from Okasaki's book; second, see how easy (or hard) it is to formally prove correctness of functional algorithms. Even though most code (and time spent) concerns the first of these goals, most insight I gained concerns the second one.[1] I have discovered the core of the approach described in this thesis while working on a comparison of sorting algorithms.[2] The sheer amount of different algorithms and various versions of the same algorithm made such a highly abstract and modular approach pretty much necessary, and Coq's powerful type system made it feasible. In the end I learned that for a skilled Coq user armed with my approach, proving the algorithms correct is not much harder than implementing them in the first place and that both implementation and verification become easier when considered together.

Contributions. I have stumbled upon some of the techniques presented in this thesis semi-autonomously, but I don't take any credit for inventing (or discovering)

[1] This looks like yet another example of the famous Pareto principle.

[2] This probably explains why quicksort became the running example of the thesis.

them. For example, I "knew" about (hole-) and (type-)driven development before starting this project, but I had to reinvent the wheel from scratch myself, driven purely by the needs of proof engineering, in order to really grasp, absorb and assimilate them. My only original addition to it is the idea of abstracting over holes, which I haven't seen described anywhere else, but I'm pretty sure that people more knowledgeable on the matter have also encountered this idea. I think that my greatest contribution is simply putting all of these concepts and techniques into a coherent and effective approach to functional algorithms.

Some techniques described in this thesis, like functional induction, are well-known, but lacked good expository material, which I provided in section 5.1. Others, like the Bove-Capretta method, are both well-known and widely present in the literature, but scattered across different papers, some of which are paywalled, most of which are jargony and directed at insiders in the field. I rectified the situation by giving this method a more friendly name – "the inductive domain method" – and providing a beginner-friendly introduction in section 4.1. Yet other techniques, like proof by admission, are kind of folklore. On the one hand, I am not aware of any tutorials or other expository texts on it. On the other hand, I'm pretty sure it's very well-known, but it is hard to check this because it lacks a standard name. This is a serious problem, because it is very hard to refer to nameless things, hard to think about them and hard to teach them.

Related and further work. The approach presented in this thesis is powerful, but not complete and could be improved in many respects. Although we have seen some criteria for judging specifications and some techniques for improving them, it would be nice to have a general method for coming up with good specifications. Even though we did discuss well-founded induction, we haven't seen any techniques for constructing the most convenient well-founded relation for a given termination proof. We have also seen that the inductive domain method has some problems when dealing with functions that exhibit nested recursion or higher-order recursion.

Moreover, we have skipped the issue of actually inventing the algorithm that we want to formalize. The classical-imperative paradigm knows many techniques of algorithm design and most of them transfer easily to the functional paradigm, but it would be very interesting to see how they interact with the rest of our approach – maybe it's possible to synthesise the algorithm directly from the specification, or maybe the abstract template is easier to implement for greedy algorithms than for dynamic programming?

It would also be worth exploring how other algorithmic activities described in section 1.3 fit into our approach. An interesting link between complexity analysis and inductive types is Analytic Combinatorics [92], whose slogan is "If you can specify it, you can analyze it". Testing is also relevant to formally verified algorithms. Although, as Dijkstra famously put it, "Testing can be used to show the presence of bugs, but never to show their absence!", property-based testing is extremely efficient at generating counterexamples and can shorten the feedback loop between implementation and bug detection by many orders of magnitude.[3] There has already been some work on property-based testing in Coq [93].

[3] When formalizing algorithms without tests, we usually detect implementation bugs when we can't prove the algorithm correct, but before that happens we waste a lot of time pondering the basic dilemma: is it really incorrect or are we just incompetent at proving it correct?

Bibliography

[1] `https://www.thefreedictionary.com/algorithm`,
WaybackMachine copy

[2] *Do "algorithms" exist in Functional Programming?*,
WaybackMachine copy

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein,
Introduction to Algorithms, 1990

[4] Donald Knuth, *The Art of Computer Programming*, 1968

[5] Thomas S. Kuhn, *The Structure of Scientific Revolutions*, 1962

[6] Robert Harper, *Words Matter*, 2012,
WaybackMachine copy

[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

[9] Nicholas Pippenger, *Pure versus impure Lisp*, 1996

[10] John Launchbury, Simon Peyton Jones,
Lazy Functional State Threads, 1994

[11] Andrej Bauer, *Proof of negation and proof by contradiction*, 2010,
WaybackMachine copy

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

[13] Morten Heine Sørensen, Paweł Urzyczyn,
Lectures on the Curry-Howard Isomorphism, 2006

[14] Bruno Barras, Benjamin Werner, *Coq in Coq*, 1997

[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

[16] Alan Turing,
On Computable Numbers, with an Application to the Entscheidungsproblem, 1936

[17] Alonzo Church, *An Unsolvable Problem of Elementary Number Theory*, 1936

[18] Guy Blelloch, Robert Harper, *λ-Calculus: The Other Turing Machine*, 2015

[19] Ugo Dal Lago, Simone Martini,
The Weak Lambda Calculus as a Reasonable Machine, 2008

[20] Ugo Dal Lago,
A Short Introduction to Implicit Computational Complexity, 2010

[21] Ugo Dal Lago, *Machine-Free Complexity*, 2019

[22] Robert Harper, *Languages and Machines*, 2015,
WaybackMachine copy

[23] Norman Danner, Daniel R. Licata, Ramyaa Ramyaa
Denotational Cost Semantics for Functional Languages with Inductive Types, 2015

[24] Thierry Coquand, Gérard Huet, *The calculus of constructions*, 1984

[25] Christine Paulin-Mohring,
Introduction to the Calculus of Inductive Constructions, 2015

[26] Amin Timany, Matthieu Sozeau,
Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC), 2018

[27] Per Martin-Löf, *An intuitionistic theory of types*, 1972

[28] Per Martin-Löf, *Intuitionistic Type Theory*, 1984

[29] Martin Hofmann, *Syntax and Semantics of Dependent Types*, 1997

[30] The Univalent Foundations Program,
Homotopy Type Theory: Univalent Foundations of Mathematics, 2013

[31] Jesper Cockx, *Overlapping and order-independent patterns in type theory*, 2013

[32] Michael Shulman,
Higher inductive-recursive univalence and type-directed definitions, 2014,
WaybackMachine copy

[33] Idris 2 documentation, *Docs » A Crash Course in Idris 2 » Multiplicities*, 2021,
WaybackMachine copy

[34] Ansten Klev, *Eta-rules in Martin-Löf type theory*, 2019

[35] Robert Harper, *Polarity in Type Theory*, 2012,
WaybackMachine copy

[36] Nils Anders Danielsson,
A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family, 2006

[37] James Chapman, *Type Theory Should Eat Itself*, 2008

[38] Healfdene Goguen, *A Typed Operational Semantics for Type Theory*, 1994

[39] Benjamin C. Pierce, Andrew W. Appel et al., *Software Foundations*, 2019

[40] Yves Bertot and Pierre Castéran,
Interactive Theorem Proving and Program Development
Coq'Art: The Calculus of Inductive Constructions, 2004

[41] Adam Chlipala, *Certified Programming with Dependent Types*, 2013

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

[43] Chris Okasaki, *Purely Functional Data Structures* (PhD thesis), 1996

[44] Chris Okasaki, *Ten Years of Purely Functional Data Structures*, 2008,
WaybackMachine copy

[45] *What's new in purely functional data structures since Okasaki?*, 2010,
WaybackMachine copy

[46] Andrew W. Appel, *Verified Functional Algorithms*, 2018

[47] James Koppel, *The Best Refactoring You've Never Heard Of*, 2019

[48] Olivier Danvy and Lasse R. Nielsen, *Defunctionalization at Work*, 2001

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

[50] Daniel Licata's lecture notes, 2012,
WaybackMachine copy

[51] Robert Harper, *Boolean Blindness*, 2011,
WaybackMachine copy

[52] Jeremy Fairbank, *Solving the Boolean Identity Crisis*, 2019,
WaybackMachine copy

[53] Harrison Ainsworth, *'Boolean-blindness' is about types*, 2013,
WaybackMachine copy

[54] John C. Reynolds,
The Meaning of Types: from Intrinsic to Extrinsic Semantics, 2000

[55] Georges Gonthier, Assia Mahboubi, Enrico Tassi
A Small Scale Reflection Extension for the Coq system, 2016

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

[57] Assia Mahboubi and Enrico Tassi with contributions by Yves Bertot and Georges Gonthier,
Mathematical Components, 2018

[58] Georges Gonthier, *A computer-checked proof of the Four Colour Theorem*, 2005

[59] Georges Gonthier, *Formal Proof – The Four-Color Theorem*, 2008

[60] Nils Anders Danielsson, John Hughes, Patrik Jansson, Jeremy Gibbons,
Fast and Loose Reasoning is Morally Correct, 2006

[61] Ana Bove, *Simple general recursion in type theory*, 2001

[62] Ana Bove and Venanzio Capretta,
Nested General Recursion and Partiality in Type Theory, 2001

[63] Ana Bove, *Mutual General Recursion in Type Theory*, 2002

[64] Ana Bove, *General Recursion in Type Theory*, 2003

[65] Ana Bove and Venanzio Capretta,
Modelling general recursion in type theory, 2004

[66] Ana Bove and Venanzio Capretta,
Recursive Functions with Higher Order Domains, 2005

[67] Ana Bove and Venanzio Capretta, *Computation by Prophecy*, 2007

[68] Ana Bove and Venanzio Capretta, *A Type of Partial Recursive Functions*, 2008

[69] Ana Bove, *Another Look at Function Domains*, 2009

[70] Donald Knuth, *Textbook Examples of Recursion*, 1991

[71] Cory Knapp, *Partial Functions and Recursion in Univalent Type Theory*, 2018

[72] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus,
Partiality, Revisited
The Partiality Monad as a Quotient Inductive-Inductive Type, 2017

[73] G. Barthe and P. Courtieu,
Efficient Reasoning about Executable Specifications in Coq, 2002

[74] R. M. Burstall, *Proving properties of programs by structural induction*, 1968

[75] Konrad Slind, *Reasoning about terminating functional programs*, 1999

[76] Conor McBride, James McKinna, *The view from the left*, 2004

[77] Matthieu Sozeau,
Un environnement pour la programmation avec types dépendants, 2008

[78] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu,
Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant, 2006

[79] Coq Reference Manual, *Functional induction*

[80] Matthieu Sozeau, Cyprien Mangin,
Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq, 2019

[81] James McKinna and Fredrik Nordvall Forsberg,
The Encode-Decode Method, Relationally (Abstract), 2015

[82] James McKinna and Fredrik Nordvall Forsberg,
The Encode-Decode Method, Relationally (Slides), 2015

[83] B. Goodspeed, *Proof-Driven Development*, 2015

[84] *GHC/Typed holes - Haskell wiki*,
WaybackMachine copy

[85] *Existential variables - Coq 8.13.1 documentation*,
WaybackMachine copy

[86] Don Fallis, *Intentional Gaps in Mathematical Proofs*, 2003

[87] Yacin Hamami,
Mathematical Rigor, Proof Gap and the Validity of Mathematical Inference, 2014

[88] Line Edslev Andersen, *Acceptable gaps in mathematical proofs*, 2020

[89] Edwin Brady, *Type-Driven Development with Idris*, 2017

[90] Tomas Petricek, *Why type-first development matters*, 2012,
WaybackMachine copy

[91] Luca Cardelli, *Typeful Programming*, 1991

[92] Philippe Flajolet and Robert Sedgewick, *Analytic Combinatorics*, 2011,
also see the accompanying booksite `https://ac.cs.princeton.edu/home/`

[93] Leonidas Lampropoulos and Benjamin C. Pierce,
Software Foundations Volume 4. QuickChick: Property-Based Testing in Coq, 2021