Verification *à la carte*: A textbook on formally verified OCaml programs

Pedro Gasparinho and Mário Pereira

NOVA LINCS, Nova School of Science and Technology, Portugal
p.gasparinho@campus.fct.unl.pt, mjp.pereira@fct.unl.pt

Abstract. Formal verification techniques have been gaining prominence in both industry and academia. Modern tools have been used to certify large and complex software. Despite the level of maturity shown empirically, documentation and learning materials are scarce for such tools. This negatively impacts the learning process of the future generations of verification engineers. In this article, we present our design process for the elaboration of a textbook on deductive verification. Our work focuses on certified algorithms and data structures using Cameleer, an automated deductive verification tool for OCaml programs with GOSPEL annotations. Currently, the textbook includes 6 chapters, with a total of 47 case studies, 18 of which are exercises, and more than 1000 lines of certified code.

Keywords: Deductive Software Verification · OCaml· Cameleer· GOSPEL

1 Introduction

The concept of *formal software verification*, *i.e.*, mathematically proving the correctness of a program, with respect to a logical specification of its behaviour, is almost as old as Computer Science. However, it was only in the last two decades that verification tools matured from simple academic artifacts to robust and scalable software capable of verifying industrial-grade systems. This "late arrival" can be partially explained by the lack of pedagogical works published concerning Formal Verification. This is further evidenced when compared to other areas of Computer Science, most notably Algorithm Design, where one can easily find a plethora of such works, each with its own characteristics.

In recent years there has been a growing adoption of formal verification tools in the industry, mostly due to the so-called *SMT revolution*. With this technology, automated proof systems became more expressive and scalable. This, in turn, broadened the spectrum of problem classes that were possible to prove, with minimal user intervention. However, almost paradoxically, in academic settings it is possible to observe that interactive tools are still regarded as the standard means to conduct research in the area of formal methods.

Part of this paradox can be explained, at least in our opinion, by the greater quantity and quality of available documentation for proof assistants. The *Software Foundation* [30] series, currently comprised by 6 volumes, is a paradigmatic example of this situation. Despite the demanding learning process associated with the Rocq system (previously known as Coq), and proof assistants in general [4], the existence of such manuals provides a more comfortable learning experience, and, overall, usage.

In this work, we clearly want to break free from this tradition. Our goal is ambitious: to write a textbook of formally verified algorithms, while also achieving fully automated proofs. Our certified programs are written in OCaml, a functional-first programming language, that also supports other paradigms. This choice allows us to organize a broad and diverse set of algorithms, several of which use the functional traits of OCaml. This variety also serves to sustain the maturity of automated verification tools.

We invite the readers of this paper to visit the textbook's website[1]. There, one is able to read the latest version of the textbook, which was made publicly available for free in PDF format. Additionally, we also provide a gallery containing all the certified algorithms and data structures found in the textbook.

2 Toolset

Functional programming is often forgotten in algorithmic textbooks. This is quite puzzling, in our opinion, as there is no shortage of interesting techniques associated with this style. Thus, not considering this paradigm would go against our objective of a diverse range of examples. Naturally, our programming language of choice must satisfy this prerequisite, in addition to having good support for automated deductive verification. There are several possible options that fit these criteria, namely verification-aware languages, such as Why3 [13], Dafny [19] and Viper [23]. This class of programming languages is traditionally meant for verification purposes only, as most are not fully executable. This limitation reduces their usability in industry for developing certified software, at least directly. Moreover, if good translation schemes do not exist, manually translating a code base or creating a translation tool may discourage companies from adopting formal methods. Due to the reasons above, our focus shifts to mainstream general-purpose programming languages. Out of these, we firmly believe that OCaml is an adequate choice for the following reasons (not exhaustive): (1) it is a multi-paradigm language (functional, imperative, and object-oriented), (2) has a state-of-the-art optimizing compiler; (3) is a type safe language with one of the most advanced type inference algorithms; and (4) has a powerful module system, which enables abstraction and modularity for comfortable large-scale software development.

To conduct our proof efforts, we rely on Cameleer [28], a deductive verification tool for OCaml programs with GOSPEL [6] annotations. GOSPEL, in turn, is a tool-agnostic specification language with a syntax based on OCaml. Internally, Cameleer uses the Why3 [13] platform by translating the annotated code to WhyML. This platform is known to support multiple external provers, both

[1] The textbook's website: https://cameleerbook.github.io/CameleerBook/

interactive and automated, as a means to guarantee more flexibility and proof capability.

As previously mentioned, we want to achieve total proof automation, *i.e.* not resorting to Why3's internal tactics[2]. This led us to choose the following state-of-the-art SMT solvers: Alt-Ergo [8], cvc5 [3], Z3 [22]. In addition, we have also used Eprover [31] to discharge a few existentially quantified expressions, most notably type invariants, where SMT solvers were not able to do so. This prover is also used in an automated manner within the Why3 platform.

3 Chapters and Scope

Our idea was clear from the start: to write a textbook about verified algorithms following a similar structure to what is found in classical Algorithm Design works. Algorithms can be greatly beneficial, in our opinion, to the teaching of Deductive Verification. Firstly, within the broader discipline of Computer Science, formal methods are regarded as an advanced topic. It requires pre-existing knowledge about programming, logic and algorithmic concepts. Thus, by selecting a collection of mostly well-known algorithms and data structures, we can reasonably expect readers to be vaguely familiar with a substantial portion of this collection. Secondly, algorithms are inherently logical artifacts, due to their nature as sets of instructions with well-studied properties. This allows them to be expressed and reasoned about independently of the selected programming and specification languages. Thirdly, algorithms and data structures are relevant across every area of computing, from academic research to large-scale industrial applications.

What was not so clear at the start, however, was which topics we wanted to include. So, we carefully analysed several famous textbooks [10,11,17,27,32,33], about algorithm design. We observed that most cover similar categories of problems, as expected. Commonly addressed topics include arithmetic, searching, ordering, data structures, optimization techniques and graphs. Out of these topics, we decided to dedicate an entire chapter to each one, except for optimization techniques and graphs. These are out of scope for the first version of the textbook. Nonetheless, we recognize their importance in classical works. This led us to discuss optimization on several occasions within other chapters. Graph algorithms, on the other hand, are reserved for future work.

4 Chapter Ordering

After selecting the contents of the textbook, a natural next step is to decide the order of the chapters. The arrangement of the topics in a textbook plays a crucial role in shaping the student's learning experience. An inadvertent chapter ordering can negatively impact their progression. To ensure a smooth and gradual learning curve, both examples and chapters should increase in complexity at

[2] Why3's tactics: https://www.why3.org/doc/technical.html#transformations.

a consistent and manageable pace. Given the scope we have disclosed above, we believe that the following chapter ordering is the most suitable for our objectives:

1. Introduction: We recognize that our tool set includes numerous technologies, each with its own installation process. Moreover, these are distributed across several different sources and platforms. As such, we consider that compiling all the necessary information in a single place would greatly streamline the process, and, overall, improve the student experience. Before diving into the case studies, we must ensure that readers are familiarized with Hoare Logic [15], and other related topics including propositional and first-order logics. This will serve to introduce the necessary theoretical background needed for this textbook. Furthermore, despite having established that basic programming and algorithmic concepts are a prerequisite to read this textbook, we cannot assume that readers are proficient with, or even knowledgeable of, the OCaml programming language. Thus, it is essential to summarize its fundamental constructs and characteristics in the first place. Additionally, this chapter is the ideal place to introduce our verification tool set. Namely, how one can use the Cameleer tool to conduct proofs inside the Why3 platform and its IDE.

2. Arithmetic: The first set of algorithms discussed in our textbook stems from arithmetical and mathematical problems. Several well-known problems of this kind have been studied for centuries. These, often, have simple and historical solutions, that make for great beginner-level programming and verification challenges. This is easily explained from the limited knowledge and computational power available at the time. In fact, most historical algorithms had to be applied manually. These characteristics turn this class of problems into the ideal candidate for the chapter after the introduction. In the context of our work, this choice is further justified by the natural transition from the mathematical definitions and logical properties to OCaml programs and GOSPEL annotations. Within this problem class, we are targeting solutions that do not include auxiliary data structures, such as arrays. Such solutions may increase the difficulty quite considerably for the first chapter, which is not our intent.

3. Searching: After arithmetical problems, and given the restriction mentioned above, we believe that the next step should exactly be to study built-in data structures, such as lists and arrays. Among the various problem classes associated with these data structures, we argue that searching stands out as the most accessible, with multiple beginner-to-intermediate-level algorithms. The searching problem consists in finding the first element that fits a given criterion, generally equality. This characteristic, alongside the fact that these algorithms do not presuppose, in principle, modifications to the contents of the data structure, makes for a smooth introduction to the techniques used to verified linear data structures. Despite our focus on linear data structures, this chapter is also a suitable place to introduce simple binary trees, *i.e.* non-ordered and non-balancing.

4. Sorting: Based on the traits of the previous chapter, we decided that the next objective would be to introduce case studies that change the contents of a data structure. This description by itself is quite vague, but a natural next step, in our opinion, is the sorting problem. It involves reorganizing a potentially unordered collection of elements into a specific, well-defined ordering, typically ascending or descending according to an established comparison criteria. From both the programming and verification perspectives, these algorithms represent a slight jump in difficulty from searching algorithms. For this chapter, we set as an objective to explore several variations of our selected case studies. Namely, different programming styles and optimization techniques. Purely functional implementations and tail recursion are important topics throughout the chapter.

5. Data Structures: Despite following a similar chapter ordering when compared to classical Algorithm Design textbooks, one notable difference lies in the placement of the data structures chapter. Although it may seem strange at first, it was a well-pondered and deliberate choice. In many classical works, data structures are introduced early on, due to necessity. The data structures there presented usually play an important role in algorithms found in later chapters. Moreover, if the data structures, themselves, are somewhat complex, then the algorithms that make use of them are likely to be even more complex. Even simple data structures can have challenging proofs. These proofs often include more advanced verification concepts, such as type invariants, refinement proofs [29], and internal views, using ghost code [12]. Once more, to ensure a smooth learning curve, we believe the previous reason alone justifies the placement of this chapter as the second to last in our textbook. Additionally, we expected data structure case studies to be considerably longer than the previous one, since these regularly include several operations to manipulate and traverse the data. Refinement proofs are an important aspect of this chapter, since we want to demonstrate that the same verified interface can have two very distinct certified modules, both implementation and proof-wise.

6. Selected Topics: As we have previously mentioned, in this textbook we concentrate on certified algorithms and data structures. However, we are not necessarily limited to these topics. There are several other interesting ideas that could be discussed in an introductory textbook about formal verification. As such, we have decided to include a final chapter dedicated to selected topics. One such idea is the collaboration between Cameleer and other verification tools. The objective of the corresponding section is to demonstrate that by combining different tools in a verification workflow, these can cover each other's weaknesses. Another topic is the verification of time complexities. Despite the mechanisms for it are limited, we nonetheless consider it a curious case study. Additionally, this chapter is also the perfect place to cover, even if briefly, topics that have been left out, or revisit problem classes that were heavily restricted. In particular, we are able to cover more complex arithmetic algorithms, for instance those that use dynamic programming techniques to achieve their goal. By doing so, we are laying the groundwork for a chapter exclusively dedicated to optimization, although it remains as future work.

5 Methodologies

An important aspect of creating a textbook, in particular one focused on being practical, is defining the methodologies to gather the case studies. In our case, this includes defining a strategy to find suitable implementations and proofs. Moreover, it is important to define the methodologies related to the educational aspects of the textbook.

Implementations: The primary goal with this work is not to reinvent the wheel, but rather to build upon classical algorithms and data structures with proofs of correctness. Accordingly, we prioritize reusing existing implementations for our selected case studies. In particular, we have preference for OCaml code from scientific works. With this in mind, our biggest source of inspiration is *Apprendre à programmer avec OCaml* [11]. This textbook teaches the OCaml language through algorithms. It is considered to be accessible to complete beginners[3]. Naturally, due to the proximity between this work and ours, many implementations of the case studies found in our textbook are adapted from here. However, it is important to note that our objective is not to fully verify every single algorithm found in that textbook, nor it is limited by it. In case we do not find an OCaml implementation of an algorithm that we want to study, we may resort to manually translating a suitable implementation in another language or pseudocode representation. For instance, *Algorithms* [32] was an important reference to find implementations in the imperative style.

Proofs: Our aim extends beyond solely adapting, translating and compiling existing formal proofs. We also intend to contribute to the scientific community by proving algorithms that, to the best of our knowledge, have not yet been verified using automated tools. Nonetheless, given the introductory nature of our study, this goal remains secondary. Prioritizing a set of completely original proofs would certainly lead to the exclusion of pedagogically interesting examples. Instead, we have opted to find a balance between well-known proofs, such as iterative Fibonacci and fast exponentiation, and newly developed proofs. One possible way to achieve these newly developed proofs is by exploring different variations of an algorithm, that may not yet have been formally verified. Such variations may include using a different paradigm, and/or optimization techniques, such as tail-recursion. Additionally, there have been situations where we expanded upon the existing proof, for example, new certified operations within the context of a data structure. Our biggest sources of inspiration in this matter are proof repositories associated with our tools, namely: Cameleer's example directory found in its source code[4], and a public repository with several proofs in Why3[5].

[3] List of OCaml books: https://ocaml.org/books
[4] Cameleer's gallery: https://github.com/ocaml-gospel/cameleer/tree/master/examples
[5] Why3's proof repository: https://toccata.gitlabpages.inria.fr/toccata/gallery/why3.en.html

Education: As previously mentioned, our textbook takes the difficulty between case studies and chapters, as a whole, in consideration to ensure a smooth learning experience. This is to be expected in introductory-level works. Another important aspect of our methodologies, concerning education, is its focus on practicality. When discussing practicality within the context of our work, it is important to dissociate the discussions about Formal Verification theory, and the theoretical contextualization necessary when presenting a given algorithm. In that sense, case studies, on an individual level, may feature lengthy theoretical discussions, in order to thoroughly explain and motivate their specifications. However, the former is best not presented all at once. Instead, theoretical concepts are discussed as needed, and include small illustrative examples. We believe that this makes for a more pleasant reading experience, compared to presenting all the theory found in the textbook upfront. For instance, refinement proofs are only used from the fifth chapter onwards. Hence, in our opinion, the best time to introduce the theory behind refinement proofs is at the start of that chapter.

Exercises: Exercises are a fundamental part of textbooks. Students are encouraged to explore interesting problems on their own with the knowledge learned from reading the textbook. The benefits of having exercises are undoubtable. However, the inclusion of solutions is a much more controversial debate. In our opinion solutions can be of benefit to the reader. Regarding our exercise design process, these are, generally, placed at the end of each chapter, and are meant to be open-ended exercises that contemplate both implementing and verifying. To remain loyal to our motivations, these exercises must be comprised of algorithmic problems, generally variations of the discussed topics, with at least one automated solution. By providing our own set of candidate solutions, students who come up with their own solutions can see and learn from the difference between them. Moreover, the process of verifying a program can be quite frustrating, or if an exercise is unclear to a reader (we do try to avoid such scenarios), solutions may help ease less positive experiences. However, it is up to each individual to use the solutions responsibly and positively. Solutions can also be of great help to lecturers preparing their lab classes, lecture notes and lectures. As previously mentioned, our exercises are, mainly, composed of open-ended problems that also contemplate implementing the program at hand. This choice can be justified by our objective of letting students freely explore the problem first and gain experience with the OCaml programming language.

6 Case Study Selection

In this section, we dive into several case studies found in the textbook. To accompany the case studies, we display illustrative code, at times, to reinforce our points. The explanations here presented are directed towards functional programming experts, the target audience of this article. These may not necessarily reflect the discussions found in the textbook, which are directed towards beginners. Furthermore, we are not able to discuss every case study in the same depth as in the textbook. Instead, we focus on the design and educational aspects behind the selection of algorithms and data structures. At times, some case studies may be grouped and presented simultaneously for conciseness.

6.1 Arithmetic

Extended Euclidean Algorithm: Choosing the first case study in a textbook is not an easy task. It must balance simplicity and complexity quite carefully, as it sets the baseline difficulty for the rest of the work. Additionally, we also believe that it should not be too long, in order to ensure that the reader does not get lost in the new concepts. As such, the algorithm that best fits this description, in our opinion, is the extended version of the Euclidean algorithm. From a verification perspective, using the extended version simplifies the proof substantially. This is due to the use of Bézout's identity[6] as a post-condition, rather the concept of greatest common divisor, which would require defining such concept as an auxiliary logical function. Bézout's identity is a strong property, and its coefficients can be easily obtained using recursion:

GOSPEL + OCaml

```
let rec extended_gcd x y =
  if y = 0 then (1, 0, x)
  else
    let q = x / y in
    let (a, b, d) = extended_gcd y (x - q * y) in
    (b, a - q * b, d)
(*@ (a, b, d) = extended_gcd x y
    requires x <> 0
    variant abs y
    ensures d = a*x + b*y *)
```

GOSPEL annotations are expressed inside special comments, ignored by the compiler, in the form `(*@ ... *)`. The `requires` and `ensures` clauses are used to express pre-conditions and post-conditions, respectively. The `variant` clause is used, in this case, to prove the termination of `extended_gcd`.

As observed, the extended Euclidean algorithm can be implemented and proven in considerably small number of lines. This aligns with our goal of not having a case study that is too long, despite having the necessity to explain the mathematical reasoning on how to obtain the Bézout's coefficients and why does Bézout's identity work as a post-condition. Fortunately, that can be achieved in a couple of pages.

McCarthy's 91 Function: While our definition of algorithm can be a bit broad at times, this function[7] serves the purpose of showing a tricky termination proof, due to nested recursion, as seen in the mathematical definition and OCaml implementation below:

[6] Bézout's identity: https://en.wikipedia.org/wiki/B%C3%A9zout%27s_identity.
[7] McCarthy's 91 Function: https://en.wikipedia.org/wiki/McCarthy_91_function

$$M(n) = \begin{cases} M(M(n+11)), & \text{if } n \le 100 \\ n - 10, & \text{if } n > 100 \end{cases}$$

```
let rec M n =                                        OCaml
  if n <= 100 then M(M (n + 11))
  else n - 10
```

Additionally, with this example, we demonstrate two possible strategies to deal with conditional properties:

```
let rec M n =            GOSPEL + OCaml
  (* ... *)
(*@ r = M n
    variant 101 - n
    ensures n <= 100 -> r = 91
    ensures n > 100 -> r = n - 10 *)
```

```
let rec M n =            GOSPEL + OCaml
  (* ... *)
(*@ r = M n
    variant 101 - n
    ensures if n <= 100 then r = 91
            else r = n - 10 *)
```

The specification on the left side uses the logical consequence operator, while the one on the right side uses an if-expression.

Imperative Euclidean Division & Iterative Fibonacci: The first two case studies were purely functional. Given our objective of having a set of diverse examples in terms of programming styles, we believe that the imperative paradigm should also be introduced early on. Firstly, we introduce `while` loops and loop invariants with the Euclidean division algorithm. Secondly, we discuss an iterative implementation, using the `for` loop, of the Fibonacci sequence. This example also serves as an introduction to logical functions, in order to define the concept of Fibonacci number, which can be defined and annotated, in GOSPEL, as follows:

```
(*@ function rec fib (n: int) : int =                              GOSPEL
    if n <= 1 then n
    else fib (n-1) + fib (n-2) *)
(*@ requires n >= 0
    variant n *)
```

Imperative Extended Euclidean Algorithm: The iterative way to calculate Bézout's coefficients is considerably more complex than the recursive approach. It requires a total of 4 auxiliary variables to keep track of the coefficients. These variables can be annotated as ghost variables, as a means to introduce the concept of ghost code. In this case study, we also use the existential quantifier for the first time:

```
let gcd (x:int) (y:int) = (* ... *)                         GOSPEL + OCaml
(*@ r = gcd x y
    requires x >= 0
    requires y >= 0
    ensures exists a,b. r = a*x+b*y *)
```

This version represents a jump in difficulty. However, we believe that by having already explored this example previously, it eases the difficulty. Moreover, it is one of the approachable examples using ghost code and existential quantifiers, in our opinion.

Fast Exponentiation: The fast exponentiation algorithm is another common example of logical functions. This time around, we need to define the power function in GOSPEL. Although, the real educational value of this algorithm, in our textbook, is the need to define and prove a lemma-function:

GOSPEL + OCaml

```
let[@lemma] rec power_lemma (x: int) (n: int) =
  if n > 1 then power_lemma x (n-2)
(*@ requires n >= 0
    variant n
    ensures mod n 2 = 0 -> power x n = (power (x * x) (div n 2))
    ensures mod n 2 = 1 -> power x n = x * (power (x * x) (div n 2)) *)
```

This lemma is used to prove the fundamental mathematical property that allows fast exponentiation to exist, that being:

$$\text{For } n \geq 0, \text{ then: } x^n = \begin{cases} (x^2)^{\frac{n}{2}}, & \text{if } n \bmod 2 = 0 \\ x * (x^2)^{\frac{n-1}{2}}, & \text{if } n \bmod 2 = 1 \end{cases}$$

For the implementation of this algorithm, we choose an imperative version. Despite its functional counterpart being on the same level of difficulty, we decided to leave that version to the exercises. We believe that students are, generally, more used to the imperative paradigm, hence, we encourage them to practice and experience the beauty of functional programming, by themselves.

Exercises: There are several interesting arithmetic problems that students could tackle, by themselves. However, we decided to be more conservative, at least for now, with the exercises in this chapter. Most are various of the previous case studies: (1) Euclidean division that supports negative numbers, (2) functional Euclidean division, (3) iterative factorial function, (4) tribonacci sequence, (5) functional fast exponentiation.

6.2 Searching

Linear Search: This algorithm represents the first use of data structures in our textbook, namely by searching for a value in an array. It also introduces exceptions, on a programmatic level, as a means to stop iteration early in OCaml.

Binary Search & Ternary Search: After presenting the linear search algorithm, a natural next step is discussing binary search. This algorithm is used to briefly discuss optimization, when certain conditions are known, in this case if the array is sorted. This case study marks the first use of a predicate in GOSPEL, that being the concept of an array being sorted. Moreover, we also display different ways to encode values.

Ternary search is a variation of binary search that divides the array in three roughly equal parts. From a verification perspective, this algorithm has interesting experimental value. The same exact specification used to prove both binary search and ternary search. In fact, it is applicable to the n-ary family of searching algorithms. After this experiment, we also introduce a second implementation of ternary search that raises an exception when the value was not found. This example illustrates on how to deal with exceptions on a logical level.

Depth-first search for Binary Trees: This chapter focuses primarily on searching algorithms applied to arrays. So, we felt the need to introduce a functional example, and one that strays away from linear data structures. Thus, the first algorithm that came to mind was exactly DFS for (simple) Binary Trees.

Exercises: We recognize that finding exercises for this chapter has been rather difficult, and is one of our top priorities to expand upon. For now, we only have two exercises: (1) backwards linear search, and (2) recursive binary search.

6.3 Sorting

Small Verification Library: To successfully verify a sorting algorithm, it is often necessary to define the logical concept of a sorted collection of elements, as well as the concept of being a permutation of another collection. Moreover, one may need to define multiple lemmas on top of those logical definitions. This led us define a common verification library, with the necessary logical objects needed for the various algorithms that we address:

GOSPEL + OCaml

```
let[@logic][@ghost] rec occ v =
  (* number of occurrences omitted *)

(*@ predicate permut (l1 l2: int list) =
      forall x. occ x l1 = occ x l2 *)

(*@ predicate rec sorted (l: int list) =
      match l with
      | [] | _::[] -> true
      | x::(y::ls) -> x <= y && sorted (y::ls) *)
(*@ variant l *)

(* Lemmas are omitted *)
```

In particular, for the first part of the sorting chapter, the library and the following algorithms are geared towards focus on integer lists. This is due to technical restrictions, on Why3's side, that limit the direct usage of polymorphism in the presence of the equality operator (=). This operator is not polymorphic in Why3, when used in non-logical environments.

Functional Selection Sort: Optimization was a valued detail that we wanted to add in this chapter, namely tail-recursion. For now, we have considered four sorting algorithms: insertion, selection, merge and quick. Out of these four, we wanted to start this chapter with one that had a "standard" tail recursive version. This excluded both merge sort and quick sort, since these have two recursive calls. Between selection sort and insertion sort, we find the functional version of the former to be quite interesting. Thus, we decided to discuss selection sort first. Before actually presenting its tail recursive version, we first verify the standard functional version, and then compare both.

Functional Merge Sort: While it is possible to achieve a tail recursive merge sort implementation using either continuation-passing style or defunctionalization, neither was possible to present in this chapter. The former makes use of higher-order logic, which is not available in Why3, and consequently Cameleer. Proofs involving the latter are quite difficult and not adequate for this chapter. Thus, we have decided to ignore tail-call optimizations for the merge sort algorithm, focusing only on its standard version.

Functional Quick Sort: There are several possible quick sort optimizations. On one hand, the previous techniques also apply to quick sort, but are not adequate to our verification purposes. On the other hand, there are other simpler techniques that we could use. Out of those techniques, we highlight two, using a better pivot, namely the median of three, and the largest half tail-call. Given that we are using lists in this section of the sorting chapter, accessing any other element other than the head is not advisable due to the time complexity being $O(n)$ in the worst case. By contrast, the largest half tail-call is feasible, in this context. This technique amounts to first checking which of the two sub-lists, obtained from splitting the list based on its pivot, is the largest. Then, that sub-list is called last, so that it can be the tail-call. This achieves logarithmic stack space, since only half or less of the elements, are being used in a non-tail recursive call. Similarly, to the selection sort case study, we first present the non-optimized version, and the compare both.

Polymorphic Selection Sort: As previously mentioned, due to technical limitation, we are not able to directly use polymorphism in the presence of the equality operator (=), in non-logical environments. Instead, we can devise a verified signature (that does not need to be implemented) to encode polymorphism:

GOSPEL + OCaml

```
module type OrderedType = sig
  type t

  val eq: t -> t -> bool [@@logic]
  (*@ b = eq x y
      ensures b <-> x = y *)

  (*@ function le: t -> t -> bool *)
  (*@ axiom reflexive : forall x. le x x *)
  (*@ axiom total : forall x y. le x y \/ le y x *)
  (*@ axiom transitive: forall x y z. le x y -> le y z -> le x z *)
  val leq: t -> t -> bool [@@logic]
  (*@ b = leq x y
      ensures b <-> le x y *)
end
```

Type `t` can be seen as a generic type, containing an equality test and a (non-strict) total order relation, akin to $\leq$. This signature can be used in a parameterized module, as such:

```
module SelectionSort (O: OrderedType) = struct                        OCaml
  type elt = O.t
  (* Omitted *)
end
```

The remainder of the case study amounts to replacing instances of `=` and `<=` by `E.eq` and `E.leq`, respectively, in the desired algorithm and the verification library. For illustration and simplicity purposes, we have chosen selection sort.

Imperative Insertion Sort: Sorting arrays is substantially different from sorting lists, even on a logical level. This is mostly due to mutability and in-place sorting. Consequently, it requires completely redesigning the verification library. In this textbook, we present the approach we have taken to verify a swap-based imperative insertion sort. To achieve this it is crucial to have a strong postcondition on the swapping function:

```
let swap (arr: int array) i j =                               GOSPEL + OCaml
  let v = arr.(i) in
  arr.(i) <- arr.(j);
  arr.(j) <- v
(*@ requires 0 <= i < Array.length arr
    requires 0 <= j < Array.length arr
    ensures exchange arr (old arr) i j
    ensures permut arr (old arr) *)
```

Namely, the exchange predicate, that states that an array is equal to another array in every single position except two, which must have their values swapped:

```
(*@ predicate exchange (a1 a2: int array) (i j: int) =        GOSPEL + OCaml
    Array.length a1 = Array.length a2 &&
    0 <= i < Array.length a1 &&
    0 <= j < Array.length a1 &&
    a1[i] = a2[j] &&
    a1[j] = a2[i] &&
    (forall k. 0 <= k < Array.length a1 && k <> i -> k <> j ->
    a1[k] = a2[k]) *)
```

Exercises: For now, we have devised several exercises based on variations of the case studies: (1) Functional insertion sort, (2) tail recursive insertion sort, (3) polymorphic functional insertion sort, (4) further optimizations on tail recursive selection sort (Using (::) instead of (@), and reversing at the end), (5) imperative selection sort (swap-based), (6) polymorphic functional merge sort, (7) un-optimized tail recursive quick sort (tail call always goes the same side), and (8) polymorphic functional quick sort.

6.4 Data Structures

Zippers: The zipper [16] is an iteration technique that can be applied to purely functional data structures for more fine-grained traversal. This technique, usually consists in finding an alternative representation to recursive data types. Such types do not offer the possibility to go backwards, which may not be adequate for some use cases. Hence, the zipper offers the possibility to revisit previous nodes and explore other paths (when applied to trees, for instance). In this chapter, we explore both the zipper applied to lists and to trees. The first serves to introduce the concept of proofs by refinement and type invariants. Meanwhile, the second is used to solve the *same fringe* problem using a modular approach.

Resizeable Array: With the first two case studies of the data structures chapter being based around the functional paradigm, we decided to introduce an imperative case study, this being resizeable arrays. Additionally, this example introduces the concept of witness, and how to find and prove simple witnesses. Witness are more of a technicality from the Why3 platform. Providing a witness in WhyML is quite comfortable with the `by` construct. Whereas, in GOSPEL, such construct is not available, and is much less comfortable to prove them. Hence, in the following examples, the respective verification conditions were ignored.

Persistent Queue & Circular Queue: The queue abstract data type can be implemented in several distinct manners. For instance, one may implement a queue with purely functional techniques, to achieve amortized time complexities, or achieve efficient time and space complexities by limiting the number of elements to a fixed number of memory blocks, using arrays. These two implementations could not be more different, yet are tied to the same set of operations, which makes them look similar from an outside perspective (not exactly equal due to the fixed size of circular queues). This common behaviour can be described logically with almost identical certified signatures. These signatures only differ in more concrete aspects, such as typing information on OCaml's side and to check invalid accesses on the circular queue. We believe that this similarity is enough to demonstrate that the "same" certified interface can be used to prove by refinement two completely opposite implementations.

Binary Search Tree & List: The penultimate case study in our data structures chapter is the binary search tree; the last is the list. Compared to the previous examples, binary search trees are often considered more complex, hence, to ensure a gradual increment in difficult, it earns its place as the second to last case study. Regarding the list, while its placement might seem strange, we have greatly emphasized optimization in its implementation, via tail recursion. Moreover, it is quite possibly our longest case study, since it adapts a realistic set of operations in a list module. These factors combined make it a considerably challenging proof.

Exercises: This chapter currently includes three exercises on the simpler side. It shall be revisited on a later date. For now, the exercises are: (1) Queue using a linked list, (2) Stack using a linked list, and (3) Set using a linked list. Operations (mem, add and check if is subset).

6.5 Selected Topics

Advanced Arithmetic: Since arithmetic is the first problem class tackled in the textbook, we are heavily restricted when selecting algorithms, in order to ensure a difficulty level adequate to beginners. Therefore, we have decided to cover a few more complex algorithms here. The two examples covered are Delannoy numbers and binomial coefficients. In particular, we have used dynamic programming approaches in these implementations. This is also in line with our objective of covering optimization techniques, even if briefly.

Tools In collaboration: Each programming language has its own advantages and disadvantages. Verification tools are no different in that regard. So, what if we were able to explicitly combine multiple verification tools to prove one or more programs within the context of a single case study? That is the question we pose in this section of our Selected Topics chapter. Naturally, for this to be possible, there has to be good intra-tool support. Fortunately, that is the case within the OCaml verification sphere. It is possible to create a module in OCaml with GOSPEL annotations that can be imported in WhyML. This opens the door for many interesting applications. One such application, is to combine our insertion sort implementation, from 6.3, with a binary search implementation in WhyML:

WhyML

```
module Client
  use int.Int
  use int.ComputerDivision
  use ref.Ref
  use seq.Seq
  use ocamlstdlib.Stdlib
  use import insertion_sort.Insertion_sort as IS

  let binary_search (a: array int) (v: int) : int
    requires { IS.sorted a }
    ensures { (* Omitted *) }
  = (* Omitted *)

  let main (a: array int) (v: int) : bool
    ensures { result <-> Array.mem v a }
  = IS.in_sort a;
  0 <= binary_search a v < Array.length a

end
```

Let us assume that we have a file named `insertion_sort.ml`, containing the insertion sort implementation, and a file named `client.mlw`, with the complete code from the previous (abbreviated) listing, in the same directory. Then, `insertion_sort.ml` can be treated as a library in WhyML. To import this file, in WhyML, one should write: `use import insertion_sort.Insertion_sort` (Note that letter capitalization is important). The `insertion_sort` before the dot (.), corresponds to the name of the file, without the extension. The following `Insertion_sort` corresponds to an implicit module defined of the same name of the file, without extension, except that the first letter is always capitalized. Moreover, when dealing with arrays in OCaml, the `use ocamlstdlib.Stdlib` import is necessary. In the context we have described, one may use the following command in the terminal, when starting the Why3 IDE, to link the library: `why3 ide client.mlw -L .` .

By analysing the previous client, one may see that the `main` function applies our OCaml implementation of insertion sort to a potentially unsorted array, which can then be applied to the binary search operation to find a given value `v`. The `binary_search` function uses the `sorted` predicate defined in GOSPEL as a precondition to check if the array is, in fact, sorted.

Time Complexity: Verifying against time complexity is possible in Cameleer. Although there are no specific mechanisms for it, one can simulate time complexity analysis by comparing a counter variable to the reference values. This counter variable should be updated after significant operation $\Theta(1)$. Exact bounds (Θ) can be calculated by using the equality (=) operator, while lower (Ω) and upper bounds (O), can be obtained from using inequality operators.

7 Benchmarking

Previously, we have stated that our proofs are fully automatic, in the sense that we do not use Why3 tactics. As such, it is important to demonstrate Cameleer's performance on our several case studies. To do so, we present below a table containing benchmarking information regarding the different case studies and exercises found in our textbook. This table is composed by four columns: (1) the respective case study, (2) the number of verification conditions (VC) generated, (3) the number of code, specification and ghost [12] lines, (4) average proof time.

The term ghost code refers to a commonly available feature in deductive verification tools. Certain fragments of code, such as variables or functions, can be declared as ghost. These fragments of code are meant for logical use only. This means that they can not directly or indirectly impact the result of non-ghost code. However, ghost code can be initialized and updated with non-ghost code. Moreover, it can be used to simplify certain proofs, for instance, certain existentially quantified expressions or type invariants. Given the logical and programmatic nature of ghost code, we decided to separate the line counting into three categories: code, specification and ghost. If we were to omit the ghost category, then a line of ghost code would have been counted as a line of code and specification simultaneously.

Case Study	#VCs	LoC/ LoS / LoG	Time (s)
Arithmetic			
`Functional Extended GCD`	1	5 / 4 / 0	0.774
`McCarthy's 91 Function`	2	6 / 7 / 0	0.770
`Imperative Euclidean Division`	1	8 / 8 / 0	0.664
`Fibonacci Sequence`	3	11 / 14 / 0	0.744
`Imperative Extended GCD`	1	14 / 9 / 6	0.744
`Imperative Fast Exponentiation`	3	12 / 15 / 2	7.762
`Exercise 1 (Eudiv w/neg. numbers)`	2	11 / 11 / 0	0.759
`Exercise 2 (Functional Eudiv)`	2	3 / 12 / 0	0.766
`Exercise 3 (Factorial)`	2	5 / 9 / 0	0.764
`Exercise 4 (Tribonacci Sequence)`	2	14 / 9 / 4	0.757
`Exercise 5 (Functional Fast Exp.)`	3	7 / 13 / 2	7.768
Searching			
`Linear Search`	1	8 / 4 / 0	0.791
`Binary Search`	1	13 / 11 / 0	0.733
`Ternary Search`	1	15 / 11 / 0	0.771
`Ternary Search (2nd version)`	1	15 / 10 / 0	0.741
`DFS Binary Tree`	2	3 / 8 / 0	0.943
`Exercise 1 (Backwards L. Search)`	1	9 / 6 / 0	0.786
`Exercise 2 (Recursive B. Search)`	1	9 / 11 / 0	0.990
Sorting			
`Functional Selection Sort`	25	31 / 38 / 16	7.828
`Tail Recursive Selection Sort`	28	29 / 42 / 16	5.243
`Functional Merge Sort`	37	32 / 42 / 16	6.321
`Functional Quick Sort`	26	28 / 38 / 16	3.787
`Optimized Tail Rec. Quick Sort`	68	35 / 51 / 16	51.459
`Polymorphic Selection Sort`	16	39 / 45 / 16	3.066
`Imperative Insertion Sort`	3	16 / 34 / 0	1.342
`Exercise 1 (Func. Insertion Sort)`	15	13 / 28 / 4	3.283
`Exercise 2 (Opt. Tail Ins. Sort)`	30	30 / 41 / 21	22.191
`Exercise 3 (Poly. Insertion Sort)`	6	22 / 36 / 4	1.864
`Exercise 4 (Opt. Tail Sel. Sort)`	29	34 / 43 / 21	6.191
`Exercise 5 (Imp. Sel. Sort)`	3	17 / 33 / 0	1.397
`Exercise 6 (Poly. Merge Sort)`	29	40 / 50 / 16	5.950
`Exercise 7 (Tail. Quick Sort)`	37	29 / 48 / 16	24.845
`Exercise 8 (Poly. Quick Sort)`	15	36 / 49 / 16	3.586
Data Structures			
`Persistent Queue`	17	53 / 48 / 4	3.507
`Circular Queue`	24	72 / 96 / 17	3.191
`List Zipper`	18	50 / 76 / 4	2.493
`Tree Zipper (Same Fringe)`	6	31 / 26 / 0	1.063
`Binary Search Tree`	15	52 / 102 / 0	4.127
`Resizeable Array`	16	49 / 50 / 3	1.891
`Linked List`	10	41 / 91 / 0	1.425
`Exercise 1 (Queue List)`	13	27 / 36 / 0	1.979
`Exercise 2 (Stack List)`	13	27 / 36 / 0	1.580

Case Study	#VCs	LoC/ LoS / LoG	Time (s)
Data Structures (Cont.)			
`Exercise 3 (Set List)`	10	24 / 43 / 0	2.303
Selected Topics			
`Delannoy Numbers`	19	10 / 11 / 0	1.793
`Binomial Coefficients`	2	10 / 14 / 0	1.673
`Tools in Collaboration (Client)`	2	8 / 14 / 0	0.968
`Time Complexity (Linear Search)`	1	9 / 5 / 7	0.775

Table 1: Case Studies Metrics

The time results found in table 1 were measured using the following configuration: 3 warm-up runs, followed by 17 concrete runs, of which the minimum and maximum values were removed to calculate the average of the remaining 15 concrete runs. The data was obtained using the hyperfine tool applied to the `why3 replay` command. These samples were collected on a Linux Mint 22.1 Cinnamon machine, with an Intel Core i7-13620H CPU, 16GB of RAM memory, and the `6.8.0-87-generic` Linux kernel. The tool versions include: OCaml 4.14.1, Why3 1.8.1, Alt-Ergo 2.6.2, cvc5 1.1.0, Z3 4.15.1, and Eprover 2.0 Turzum.

8 Related Work

The idea of a textbook about verified algorithms is a recent topic. In 2020, Nipkow *et al.* performed a study [24,26] to discover which algorithms from the famous textbook *Introduction to Algorithms* [10] had already been verified, either automatically or interactively. The results were remarkably positive, with the majority having already been proven. The previous study triggered the creation of the textbook *Functional Data Structures and Algorithms* [25], using the *Isabelle* proof assistant. Similarly, the third volume of the *Software Foundations* [2] series focuses on proving algorithms in the functional style with the Rocq proof assistant (previously known as Coq). This family of deductive tools is notorious, despite their powerful proving capabilities, for the strenuous learning process associated [4]. We believe that new verification students and industry users of formal methods will benefit from first learning an automated deductive tool. Moreover, the space for a textbook about verified algorithm using proof assistants has already been filled by these two extremely competent works.

By contrast, that is not the case, in our opinion, within the ecosphere of automated tools. The closest work we are aware of to textbook about verified algorithms is *Program Proofs* [20]. It covers several algorithms, namely from the searching and sorting problem class. However, it separates functional programming from imperative and object-oriented. This decision, regarding the structure of the textbook, feels more akin to studying the Dafny language in a paradigm-by-paradigm basis, or, at times, from feature to feature. This contrasts with the structure that we have envisioned. In particular, we wanted to follow the classical approach of Algorithm Design works. Moreover, Dafny is a verification-aware programming language, and, while this certainly has its merits and advantages, we preferred to use OCaml, a mainstream general-purpose programming language, as justified in section 2. Other well-regarded works using automated tools include *Guide to Software Verification with Frama-C* [18] and *Deductive Software Verification - The KeY Book* [1]. A common characteristic between these works and ours is the use of mainstream general-purpose programming languages, however, these textbooks are very complete, extensive, and, at times, technical in the teaching of their respective tools, which contrasts with our practice-oriented and algorithmic approach.

9 Conclusions and Future Work

In this article, we have presented our design process to write an introductory textbook about certified algorithms and data structures with automated deductive verification tools, in particular Cameleer. In this section, we describe possible future applications of our textbook and lines of work to expand it. We also briefly reflect on the state of our work and the Deductive Verification field.

Applications: Undoubtably, the most obvious application of our work is in educational contexts. It can either be used as a self-study manual or be a significant reference for courses on Formal Verification. As our textbook is nearing the end of its initial development phase, the next step is to put it to test. Ideally, we want to use it in a classroom environment to test its effectiveness. This would allow us to see how students react to the different chapters and case studies. Additionally, during this phase, we expect to receive substantial feedback and new ideas, as well as identifying potential typos. If we overlook the educational aspect of our work, then it can also be seen as a standard library of certified algorithms. Software projects with high security and fidelity thresholds can use these independent components as a foundation. Moreover, projects with more lenient requirements may also integrate these algorithms and data structures for additional reliability. With these applications in mind, alongside the growing interest in formal methods seen in industry and academia, we firmly believe that our work presents itself as a serious contribution to this field, as well as the Cameleer and OCaml ecospheres.

Reflections: Writing a textbook is far from being an easy and quick task. In fact, it is a "journey of a thousand miles". As of now, we have just arrived at the halfway point, which was having a tangible first version of the book with a significant set of examples and exercises. During these "five hundred miles", we were able to study all sorts of interesting problems and programming techniques. Even if proving such problems felt, at times, like an unsurmountable challenge. Despite the hardships, we consider that the collection of examples we have devised cover a substantial subset of the available functionalities of the

OCaml fragment supported by Cameleer. It includes its module system, recursion, mutability, and much more. This demonstrates, once more, the maturity of automated deductive tools.

Future Work: As future work, we want, in the first place, to continue expanding our textbook with more case studies and exercises. A chapter entirely dedicated to graph algorithms is exactly our next step. Despite the existing bibliography concerning certified graph algorithms [7], the reality is that, in general, this class of problems remains difficult to the current capacities of automated tools. We believe, once more, that the secret to success resides in the design of modular specifications and proofs, with a clear separation between different aspects of graph algorithm and data structures. We intend, in particular, to follow the approach presented by the OCamlGraph library [9]. Finally, we have narrowed ourselves, so far, to the choice and implementation of case studies that fit inside the OCaml fragment supported by the Cameleer tool. In particular, this decision does not allow us to tackle the verification of case studies that resort to dynamically allocated memory. To surpass this limitation, we anticipate two possible lines of future work: (1) contribute to the expansion of Cameleer in order to reason about separation logic [14]; (2) combine Cameleer, in a collaborative effort, with other OCaml verification tools, such as CFML [5] or Iris [21].

Acknowledgments. We thank Ion Chirica for his helpful comments during the development of both this paper and textbook.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P., Ulbrich, M.: Deductive Software Verification – The KeY Book: From Theory to Practice. Lecture Notes in Computer Science, Springer International Publishing (2016), https://books.google.pt/books?id=vdLDDQAAQBAJ
2. Appel, A.W.: Verified Functional Algorithms, Software Foundations, vol. 3. Electronic textbook (2025), version 1.5.5, http://softwarefoundations.cis.upenn.edu
3. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (2022), https://doi.org/10.1007/978-3-030-99524-9_24
4. Brain, M., Polgreen, E.: A pyramid of (formal) software verification. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14934, pp. 393–419. Springer (2024), https://doi.org/10.1007/978-3-031-71177-0_24
5. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the

16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 418–430. ACM (2011), https://doi.org/10.1145/2034773.2034828

6. Charguéraud, A., Filliâtre, J., Lourenço, C., Pereira, M.: GOSPEL - providing ocaml with a formal specification language. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 484–501. Springer (2019), https://doi.org/10.1007/978-3-030-30942-8_29
7. Chen, R., Cohen, C., Lévy, J., Merz, S., Théry, L.: Formal proofs of tarjan's strongly connected components algorithm in why3, coq and isabelle. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA. LIPIcs, vol. 141, pp. 13:1–13:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), https://doi.org/10.4230/LIPIcs.ITP.2019.13
8. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories. Oxford, United Kingdom (Jul 2018), https://inria.hal.science/hal-01960203
9. Conchon, S., Filliâtre, J., Signoles, J.: Designing a generic graph library using ML functors. In: Morazán, M.T. (ed.) Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007. Trends in Functional Programming, vol. 8, pp. 124–140. Intellect (2007)
10. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, fourth edition. MIT Press (2022), https://books.google.pt/books?id=HOJyzgEACAAJ
11. Filliâtre, J., Conchon, S.: Apprendre à programmer avec OCaml: Algorithmes et structures de données. Noire, Eyrolles (2014), https://books.google.pt/books?id=aTy9BAAAQBAJ
12. Filliâtre, J., Gondelman, L., Paskevich, A.: The spirit of ghost code. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 1–16. Springer (2014), https://doi.org/10.1007/978-3-319-08867-9_1
13. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013), https://doi.org/10.1007/978-3-642-37036-6_8
14. Gros, C., Pereira, M.: Le chameau et le serpent rentrent dans un bar: vérification quasi-automatique de code ocaml en logique de séparation. In: 36es Journées Francophones des Langages Applicatifs (JFLA 2025) (2025)
15. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969), https://doi.org/10.1145/363235.363259
16. Huet, G.P.: The zipper. J. Funct. Program. **7**(5), 549–554 (1997), https://doi.org/10.1017/s0956796897002864
17. Knuth, D.E.: The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition. Addison-Wesley (1997), https://www.worldcat.org/oclc/312910844
18. Kosmatov, N., Prevosto, V., Signoles, J.: Guide to Software Verification with Frama-C: Core Components, Usages, and Applications. Computer Science Founda-

tions and Applied Logic, Springer International Publishing (2024), https://books.google.pt/books?id=lD0TEQAAQBAJ

19. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
20. Leino, K., Leino, K.: Program Proofs. MIT Press (2023), https://books.google.pt/books?id=98p3EAAAQBAJ
21. Mével, G., Jourdan, J., Pottier, F.: Cosmo: a concurrent separation logic for multicore ocaml. Proc. ACM Program. Lang. **4**(ICFP), 96:1–96:29 (2020), https://doi.org/10.1145/3408978
22. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
23. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Pretschner, A., Peled, D., Hutzelmann, T. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 50, pp. 104–125. IOS Press (2017). https://doi.org/10.3233/978-1-61499-810-5-104, https://doi.org/10.3233/978-1-61499-810-5-104
24. Nipkow, T.: Teaching algorithms and data structures with a proof assistant (invited talk). In: Hritcu, C., Popescu, A. (eds.) Certified Programs and Proofs, CPP 2021. ACM (2021)
25. Nipkow, T., Blanchette, J., Eberl, M., Lammich, A.G.P., Sternagel, C., Wimmer, S., Zhan, B.: Functional data structures and algorithms. https://functional-algorithms-verified.org/, accessed: 2025-06-07
26. Nipkow, T., Eberl, M., Haslbeck, M.P.L.: Verified textbook algorithms - A biased survey. In: Hung, D.V., Sokolsky, O. (eds.) Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12302, pp. 25–53. Springer (2020), https://doi.org/10.1007/978-3-030-59152-6_2
27. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
28. Pereira, M., Ravara, A.: Cameleer: A deductive verification tool for ocaml. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 677–689. Springer (2021), https://doi.org/10.1007/978-3-030-81688-9_31
29. Pereira, M.J.P.: Tools and Techniques for the Verification of Modular Stateful Code. (Outils et techniques pour la vérification de programmes impératives modulaires). Ph.D. thesis, University of Paris-Saclay, France (2018), https://tel.archives-ouvertes.fr/tel-01980343
30. Pierce, B.C., de Amorim, A.A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Logical Foundations, Software Foundations, vol. 1. Electronic textbook (2025), version 6.9.0, http://softwarefoundations.cis.upenn.edu
31. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) Proc. of the 27th CADE, Natal, Brasil. pp. 495–507. No. 11716 in LNAI, Springer (2019)
32. Sedgewick, R., Wayne, K.: Algorithms. Addison-Wesley (2011), https://books.google.pt/books?id=MTpsAQAAQBAJ

33. Skiena, S.: The Algorithm Design Manual. Texts in Computer Science, Springer International Publishing (2021), https://books.google.pt/books?id=G22jzgEACAAJ