Verified Textbook Algorithms
A Biased Survey

Tobias Nipkow and Manuel Eberl and Maximilian P. L. Haslbeck

Technische Universität München, Munich, Germany

Abstract. This article surveys the state of the art of verifying standard textbook algorithms. We focus largely on the classic text by Cormen *et al.* Both correctness and running time complexity are considered.

1 Introduction

Correctness proofs of algorithms are one of the main motivations for computer-based theorem proving. This survey focuses on the verification (which for us always means machine-checked) of textbook algorithms. Their often tricky nature means that for the most part they are verified with interactive theorem provers (*ITP*s).

We explicitly cover running time analyses of algorithms, but for reasons of space only those analyses employing ITPs. The rich area of automatic resource analysis (e.g. the work by Jan Hoffmann *et al.* [103,104]) is out of scope.

The following theorem provers appear in our survey and are listed here in alphabetic order: ACL2 [111], Agda [31], Coq [25], HOL4 [181], Isabelle/HOL [151,150], KeY [7], KIV [63], Minlog [21], Mizar [17], Nqthm [34], PVS [157], Why3 [75] (which is primarily automatic). We always indicate which ITP was used in a particular verification, unless it was Isabelle/HOL (which we abbreviate to Isabelle from now on), which remains implicit. Some references to Isabelle formalizations lead into the *Archive of Formal Proofs* (*AFP*) [1], an online library of Isabelle proofs.

There are a number of algorithm verification frameworks built on top of individual theorem provers. We describe some of them in the next section. The rest of the article follows the structure and contents of the classic text by Cormen *et al.* [49] (hereafter abbreviated by *CLRS*) fairly closely while covering some related material, too. Material present in CLRS but absent from this survey usually means that we are not aware of a formalization in a theorem prover.

Because most theorem provers are built on a logic with some kind of functional programming language as a sublanguage, many verifications we cite pertain to a functional version of the algorithm in question. Therefore we only mention explicitly if a proof deals with an imperative algorithm.

It must be emphasized that this survey is biased by our perspective and covers recent work by the authors in more depth. Moreover it is inevitably incomplete. We encourage our readers to notify us of missing related work.

2 Programming and Verification Frameworks

In this section we will describe how the various systems that appear in our survey support program verification. The theorem provers ACL2, Agda, Coq, HOL, Isabelle, Minlog, Nqthm and PVS are based on logics that subsume a functional programming language. By default that is the language algorithms must be formulated in. ACL2 and Nqthm are special in two regards: their logic contains the programming language Lisp whereas the other theorem provers typically rely on some sort of compiler into functional languages like SML, OCaml, Haskell, Scala and even JavaScript; ACL2 also supports imperative features [35] directly.

KeY and KIV are primarily program verification systems but with inbuilt provers. They support modular verification and stepwise refinement. KeY focuses on the verification of Java programs, KIV on refinement and the automatic generation of executable programs in multiple target languages.

Why3 also falls into the program verification category. It has its own programming and specification language WhyML, which is mostly functional but with mutable record fields and arrays. Verification conditions are discharged by Why3 with the help of various automated and interactive theorem provers. WhyML can be translated into OCaml but can also be used as an intermediate language for the verification of C, Java, or Ada programs.

Mizar is the odd one out: it does not have any built-in notion of algorithm and its proofs about algorithms are at an abstract mathematical level.

There are various approaches for algorithm verification. Two important categories are the following:

Explicit programming language (deep embedding). One can define a programming language – functional or imperative – with a convenient set of constructs, give it a formal semantics, and then express an algorithm as a program in this language. Additionally, a cost model can be integrated into the semantics to enable formal reasoning about running time or other resource use. The actual analysis is then typically done with some sort of program logic (e.g. a Hoare-style calculus). When embedded in a theorem prover, this approach is often referred to as a *deep embedding*.

Directly in the logic (no embedding). As was mentioned before, many ITPs offer functionality to define algorithms directly in the logic of the system – usually functionally. This approach is more flexible since algorithms can use the full expressiveness of the system's logic and not only some fixed restricted set of language constructs. One possible drawback of this approach is that it can be difficult or even impossible to reason about notions such as running time explicitly. A possible workaround is to define an explicit cost function for the algorithm, but since there is no formal connection between that function and the algorithm, one must check by inspection that the cost function really does correspond to the incurred cost. Another disadvantage is that, as was said earlier, most logics do not have builtin support for imperative algorithms.

Hybrids between these two approaches also exist (such as *shallow embeddings*). And, of course, the different approaches can be combined to reap the advantages of all of them; e.g. one can show a correspondence between the running time of a deeply-embedded algorithm and a cost function specified as a recurrence directly in the logic, so that results obtained about the latter have a formal connection to the former.

Imperative Verification Frameworks As examples of such combined approaches, both Coq and Isabelle provide frameworks for the verification of imperative algorithms that are used to verify textbook algorithms.

The CFML tool [41] allows extracting a *characteristic formula* – capturing the program behaviour, including effects and running time – from a Caml program and importing it into Coq as an axiom. The CFML library provides tactics and infrastructure for Separation Logic with time credits [42] that allow to verify both functional correctness and running time complexity.

Sakaguchi [176] presented a library in Coq that features a state-monad and extraction to OCaml with mutable arrays.

Imperative-HOL is a monadic framework with references and arrays by Bulwahn *et al.* [38] which allows code generation to ML and Haskell with references and mutable arrays. Lammich [118] presents a simplified fragment of LLVM, shallowly embedded into Isabelle, with a code generator to LLVM text. Both Imperative-HOL and Isabelle-LLVM come with a Separation Logic framework and powerful proof tools that allow reasoning about imperative programs. Zhan and Haslbeck [201] extend Imperative-HOL with time and employ Separation Logic with time credits.

Isabelle Refinement Framework There are several techniques for verifying algorithms and data structures. Many verification frameworks start in a "bottom-up" manner from a concrete implementation and directly generate verification conditions which are then discharged automatically or interactively. Another technique is to model an algorithm purely functionally, prove its correctness abstractly and then prove that it is refined by an imperative implementation. For example, Lammich [122] and Zhan [200] employ this technique for the verification of algorithms in Imperative-HOL.

A third approach is best described as "top-down": an abstract program capturing the algorithmic idea is proved correct, refined stepwise to a more concrete algorithm and finally an executable algorithm is synthesized. The Isabelle Refinement Framework [127] constitutes the top layer of a tool chain in Isabelle that follows that approach. It allows specifying the result of programs in a nondeterminism monad and provides a calculus for stepwise refinement. A special case is data refinement, where abstract datatypes (e.g. sets) are replaced by concrete datatypes (e.g. lists). Many algorithms have been formalized in this framework and we will mention some of them in the main part of this paper. Lammich provides two backends to synthesize an executable program and produce a refinement proof from an abstract algorithm: The Autoref tool [114] yields a purely functional implementation by refining abstract datatypes with data structures from the Isabelle Collections Framework [121]. The Sepref tool [118,119] synthesizes efficient imperative implementations in Imperative-HOL and LLVM. Haslbeck and Lammich [95] extended the Isabelle Refinement Framework and Sepref to reason abstractly about the running time of programs and synthesize programs that preserve upper bounds through stepwise refinement down to Imperative-HOL with time [201].

2.1 Approaches for Randomized Algorithms

There are various approaches for reasoning about randomized algorithms in a formal way. Analogously to the non-randomized setting described in Section 2, there again exists an entire spectrum of different approaches:

- fully explicit/deeply-embedded approaches
- "no embedding" approaches that model randomized algorithms directly in the logic as functions returning a probability distribution
- shallow embeddings, e.g. with shallow deterministic operations but explicit random choice and explicit "while" loops. Examples are the approaches by Petcher and Morrisett [165] in Coq and by Kaminski *et al.* [110] on paper (which was formalized by Hölzl [105]).
- combined approaches that start with a program in a deeply-embedded probabilistic programming language and then relate it to a distribution specified directly in the logic, cf. e.g. Tassarotti and Harper [188].

Next, we will explore the different approaches that exist in an ITP setting to represent probability distributions. This is crucial in the "no embedding" approach, but even in the other cases it is useful to be able to give a formal semantics to the embedded programming language, prove soundness of a program logic, etc. The first work known to us on formalizing randomized algorithms is by Hurd [109] and represented randomized algorithms as deterministic algorithms taking an infinite sequence of random bits as an additional input. However, it seems that later work preferred another approach, which we will sketch in the following.

Generally, the idea is to have a type constructor M of probability distributions, i.e. $M(\alpha)$ is the type of probability distributions over elements of type α. This type constructor, together with two monadic operations $\mathit{return} : \alpha \to M(\alpha)$ and $\mathit{bind} : M(\alpha) \to (\alpha \to M(\beta)) \to M(\beta)$, forms the *Giry monad* [84], which in our opinion has emerged as the most powerful and natural representation for randomized programs in an ITP setting.

The exact definition and scope of $M(\alpha)$ varies. The following approaches are found in popular ITPs:

- For their formalization of quicksort in Coq, Van der Weegen and McKinna [193] represented distributions as trees whose leaves are deterministic results and whose nodes are uniformly random choices. While well-suited for their use case, this encoding is not ideal for more general distributions.
- Isabelle mostly uses *probability mass functions* (PMFs), i.e. functions $\alpha \to [0,1]$ that assign a probability to each possible result (which only works for distributions with countable support). The same approach is also used by Tassarotti and Harper [187] in Coq.
- As an earlier variation of this, Audebaud and Paulin-Mohring [14] used the CPS-inspired encoding $(\alpha \to [0,1]) \to [0,1]$ of PMFs in Coq.
- Isabelle contains a very general measure theory library [106] in which distributions are functions $\mathrm{Set}(\alpha) \to [0,1]$ that assign a probability to every measurable set of results. This is the most expressive representation and allows for continuous distributions (such as Gaussians) as well. It can, however, be tedious to use due to the measurability side conditions that arise. PMFs are therefore preferred in applications in Isabelle whenever possible.

The main advantage of having probability distributions in the logic as first-class citizens is again expressiveness and flexibility. It is then even possible to prove that two algorithms with completely different structure have not just the same expected running time, but exactly the same distribution. For imperative randomized algorithms or fully formal cost analysis, one must however still combine this with an embedding, as done by Tassarotti and Harper [188].

One notable system that falls somewhat outside this classification is *Ellora* by Barthe *et al.* [19]. This is a program logic that is embedded into the EasyCrypt theorem prover [18], which is not a general-purpose ITP but still general enough to allow analysis of complex randomized algorithms.

This concludes the summary of the verification frameworks we consider. The rest of the paper is dedicated to our survey of verified textbook foundations and algorithms. We roughly follow the structure and contents of CLRS.

3 Mathematical Foundations

3.1 Basic Asymptotic Concepts

Landau symbols ("Big-O", "Big-Theta", etc.) are common in both mathematics and in the analysis of algorithms. The basic idea behind e.g. a statement such as $f(n) \in O(g(n))$ is that $f(n)$ is bounded by some multiple of $g(n)$, but different texts sometimes differ as to whether "bounded" means $f(n) \leq g(n)$ or $f(n) \leq |g(n)|$ or even $|f(n)| \leq |g(n)|$. Usually (but not always), the inequality need also only hold "for sufficiently large n". In algorithms contexts, f and g are usually functions from the naturals into the non-negative reals so that these differences rarely matter. In mathematics, on the other hand, the domain of f and g might be real or complex numbers, and the neighbourhood in which the inequality must hold is often not $n \to \infty$, but e.g. $n \to 0$ or a more complicated region.

Finding a uniform formal definition that is sufficiently general for all contexts is therefore challenging.

To make matters worse, informal arguments involving Landau symbols often involve a considerable amount of hand-waving or omission of obvious steps: consider, for instance, the fact that

$$\exp\left(\frac{x+1}{\sqrt{x}+1}\right)x^a(\log x)^b \in O\big(e^x\big)\ . \tag{1}$$

This is intuitively obvious, since the first factor on the left-hand side is "roughly" equal to $e^{x/2}$. This is exponentially smaller than the e^x on the right-hand side and therefore eclipses the other, polynomial–logarithmic factors. Doing such arguments directly in a formally rigorous way is very tedious, and simplifying this process is a challenging engineering problem.

Another complication is the fact that pen-and-paper arguments, in a slight abuse of notation, often use $O(g(n))$ as if it were one single function. The intended meaning of this is "there exists some function in $O(g(n))$ for which this is true". For example, one writes $e^{\sqrt{n+1}} = e^{\sqrt{n}+O(1/\sqrt{n})}$, meaning "there exists some function $g(n) \in O(1/\sqrt{n})$ such that $e^{\sqrt{n+1}} = e^{\sqrt{n}+g(n)}$." This notation is very difficult to integrate into proof assistants directly.

Few ITPs have support for Landau-style asymptotics at all. In the following, we list the formalizations that we are aware of:

– Avigad *et al.* [15] defined "Big-O" for their formalization of the Prime Number Theorem, including the notation $f =o\ g +o\ O(h)$ for $f(x) = g(x) + O(h(x))$ that emulates the abovementioned abuse of notation at least for some simple cases. However, their definition of O is somewhat restrictive and no longer used for new developments.
– Eberl defined the five Landau symbols from CLRS and the notion of asymptotic equivalence ("$\sim$"). These are intended for general-purpose use. The neighbourhood in which the inequality must hold is $n \to \infty$ by default, but can be modified using filters [107], which allow for a great degree of flexibility. This development is now part of the Isabelle distribution. A brief discussion of it can be found in his article on the Akra–Bazzi theorem [59].
– Guéneau *et al.* [87] (Coq) define a "Big-O"-like domination relation for running time analysis, also using filters for extra flexibility.
– Affeldt *et al.* [6] (Coq) define general-purpose Landau symbols. Through several tricks, they fully support the abovementioned "abuse of notation".

It seems that the filter-based definition of Landau symbols has emerged as the canonical one in an ITP context. For algorithm analysis, the filter in question is usually simply $n \to \infty$ so that this extra flexibility is not needed, but there are two notable exceptions:

– Multivariate "Big-O" notation is useful e.g. if an algorithm's running time depends on several parameters (e.g. the naïve multiplication of two numbers m and n, which takes $O(mn)$ time). This can be achieved with *product filters*.

– Suppose we have some algorithm that takes $O(\log n)$ time on sorted lists xs for large enough n, where $n = |xs|$ is the length of the list. This can be expressed naturally as $\text{time}(xs) \in O_F(\log |xs|)$ w.r.t. a suitable filter F. For instance, in Isabelle, this filter can be written as

$$\text{length } \textbf{going-to} \text{ at\_top } \textbf{within} \ \{xs.\ \text{sorted } xs\}\ .$$

In addition to that, Coq and Isabelle provide mechanisms to facilitate asymptotic reasoning:

– Affeldt *et al.* [6] provide a proof method called "near" in Coq that imitates the informal pen-and-paper reasoning style where in asymptotic arguments, one can assume properties as long as one can later justify that they hold *eventually*. This can lead to a more natural flow of the argument.
– Motivated by the proof obligations arising from applications of the Master theorem, Eberl [59] implemented various *simplification procedures* in Isabelle that rewrite Landau-symbol statements into a simpler form. Concretely, there are procedures to
  - cancel common factors such as $f(x)g(x) \in O(f(x)h(x))$,
  - cancel dominated terms, e.g. rewriting $f(x) + g(x) \in O(h(x))$ to $f(x) \in O(h(x))$ when $g(x) \in o(f(x))$ and
  - simplify asymptotic statements involving iterated logarithms, e.g. rewriting $x^a(\log x)^b \in O(x^c(\log\log x)^d)$ to equations/inequalities of a, b, c, d.
– Lastly, Eberl [60] provides an Isabelle proof method to prove limits and Landau-symbol statements for a large class of real-valued functions. For instance, it can solve the asymptotic problem (1) mentioned earlier fully automatically.

3.2 The Master Theorem

CLRS present the Master theorem for divide-and-conquer recurrences and use it in the running time analysis of several divide-and-conquer algorithms. They also briefly mention another result known as the Akra–Bazzi theorem and cite the streamlined version due to Leighton [130]. This result generalizes the Master theorem in several ways:

– The different sub-problems being solved by recursive calls are not required to have the same size.
– The recursive terms are not required to be exactly $f(\lfloor n/b \rfloor)$ or $f(\lceil n/b \rceil)$ but can deviate from n/b by an arbitrary sub-linear amount.
– While the "balanced" case of the original Master theorem requires $f(n) \in \Theta(n^{\log_b a}(\log n)^k)$, the Akra–Bazzi theorem also works for a much larger class of functions.

The only formalized result related to this that we are aware of is Eberl's formalization of Leighton's version of the Akra–Bazzi theorem [59]. CLRS state that the Akra–Bazzi Theorem "can be somewhat difficult to use" – probably due to its rather technical side conditions and the presence of an integral in the result. However, Eberl's formalization provides several corollaries that combine the first and second advantage listed above while retaining the ease of application of the original Master theorem.

Eberl gives some applications to textbook recurrences (mergesort, Karatsuba multiplication, Strassen multiplication, median-of-medians selection). Zhan and Haslbeck [201] also integrated Eberl's work into their work on verifying the asymptotic time complexity of imperative algorithms (namely imperative versions of mergesort, Karatsuba and median-of-medians). Rau and Nipkow [173] used Eberl's Master theorem to prove the $O(n \log n)$ running time of a closest-pair-of-points algorithm.

4 Sorting and Order Statistics

4.1 Sorting

Verification of textbook sorting algorithms was a popular pastime in the early theorem proving days (e.g. [32]) but is now more of historic interest. To show that the field has progressed, we highlight three verifications of industrial code.

The sorting algorithm TimSort (combining mergesort and insertion sort) is the default implementation for generic arrays and collections in the Java standard library. De Gouw *et al.* [86] first discovered a bug that can lead to an `ArrayIndexOutOfBoundsException` and suggested corrections. Then De Gouw *et al.* [85] verified termination and exception freedom (but not full functional correctness) of the actual corrected code using KeY.

Beckert *et al.* [20], again with KeY, verified functional correctness of the other implementation of sorting in the Java standard library, a dual pivot quicksort algorithm.

Lammich [120] verified a high-level assembly-language (LLVM) implementation of two sorting algorithms: introsort [140] (a combination of quicksort, heapsort and insertion sort) from the GNU C++ library (libstdc++) and pdqsort, an extension of introsort from the Boost C++ libraries. The verified implementations perform on par with the originals.

Additionally, we mention a classic meta result that is also presented in CLRS: Eberl [56] formally proved the $\Omega(n \log n)$ lower bound for the running time of comparison-based sorting algorithms in Isabelle.

4.2 Selection in Worst-Case Linear Time

Eberl [57] formalized a functional version of the deterministic linear-time selection algorithm from CLRS including a worst-case analysis for the sizes of the lists in the recursive calls. Zhan and Haslbeck [201] refined this to an imperative algorithm, including a proof that it indeed runs in linear time using Eberl's formalization of the Akra–Bazzi theorem (unlike the elementary proof in CLRS). However, the imperative algorithm they formalized differs from that in CLRS by some details. Most notably, the one in CLRS is in-place, whereas the one by Zhan and Haslbeck is not. Formalizing the in-place algorithm would require either a stronger separation logic framework or manual reasoning to prove that the recursive calls indeed work on distinct sub-arrays.

5 Data Structures

5.1 Elementary Data Structures

We focus again on two noteworthy verifications of actual code. Polikarpova *et al.* [167,168] verify EiffelBase2, a container library (with emphasis on linked lists, arrays and hashing) that was initially designed to replace EiffelBase, the standard container library of the Eiffel programming language [136]. A distinguishing feature is the high degree of automation of their Eiffel verifier called AutoProof [78]. The verification uncovered three bugs. Hiep *et al.* [100] (KeY) verified the implementation of a linked list in the Java Collection framework and found an integer overflow bug on 64-bit architectures.

5.2 Hash Tables

The abstract datatypes sets and maps can be efficiently implemented by hash tables. The Isabelle Collections Framework [121] provides a pure implementation of hash tables that can be realized by Haskell arrays during code generation. Lammich [122,116] also verified an imperative version with rehashing in Imperative-HOL. Filliâtre and Clochard [72] (Why3) verified hash tables with linear probing. Pottier [170] verified hash tables in CFML with a focus on iterators. Polikarpova *et al.* (see above) also verified hash tables. These references only verify functional correctness, not running times.

5.3 Binary Search Trees

Unbalanced binary search trees have been verified many times. Surprisingly, the functional correctness, including preservation of the BST invariant, almost always require a surprising amount of human input (in the form of proof steps or annotations). Of course this is even more the case for balanced search trees, even ignoring the balance proofs. Most verifications are based on some variant of the following definition of BSTs: the element in each node must lie in between the elements of the left subtree and the elements of the right subtree. Nipkow [146] specifies BSTs as trees whose inorder list of elements is sorted. With this specification, functional correctness proofs (but not preservation of balancedness) are fully automatic for AVL, red-black, splay and many other search trees.

5.4 AVL and Red-Black Trees

Just like sorting algorithms, search trees are popular case studies in verification because they can often be implemented concisely in a purely functional way. We merely cite some typical verifications of AVL [74,152,172,47] and red-black trees [74,41,12,52] in various theorem provers.

We will now consider a number of search trees not in CLRS.

5.5 Weight-Balanced Trees

Weight-balanced trees were invented by Nievergelt and Reingold [143,144] (who called them "trees of bounded balance"). They are balanced by size rather than height, where the size $|t|$ of a tree t is defined as the the number of nodes in t. A tree is said to be α-balanced, $0 \leq \alpha \leq 1/2$, if for every non-empty subtree t with children l and r, $\alpha \leq \frac{|l|+1}{|t|+1} \leq 1-\alpha$. Equivalently we can require $\frac{\min(|l|,|r|)+1}{|t|+1} \leq \alpha$. Insertion and deletion may need to rebalance the tree by single and double rotations depending on certain numeric conditions. Blum and Mehlhorn [28] discovered a mistake in the numeric conditions for deletion, corrected it and gave a very detailed proof. Adams [5] used weight-balanced trees in an actual implementation (in ML) but defined balancedness somewhat differently from the original definition. Haskell's standard implementation of sets, `Data.Set`, is based on Adams's implementation. In 2010 it was noticed that deletion can break α-balancedness. Hirai and Yamamoto [102], unaware of the work by Blum and Mehlhorn, verified their own version of weight-balanced trees in Coq, which includes determining the valid ranges of certain numeric parameters. Nipkow and Dirix [149] provided a verified framework for checking validity of specific values for these numeric parameters.

5.6 Scapegoat Trees

These trees are due to Anderson [11], who called them *general balanced trees*, and Galperin and Rivest [81], who called them *scapegoat trees*. The central idea: don't rebalance every time, rebuild a subtree when the whole tree gets "too unbalanced", i.e. when the height is no longer logarithmic in the size, with a fixed multiplicative constant. Because rebuilding is expensive (in the worst case it can involve the whole tree) the worst case complexity of insertion and deletion is linear. But because earlier calls did not need to rebalance, the amortized complexity is logarithmic. The analysis by Anderson was verified by Nipkow [147].

5.7 Finger Trees

Finger trees were originally defined by reversing certain pointers in a search tree to accelerate operations in the vicinity of specific positions in the tree [88]. A functional version is due to Hinze and Paterson [101]. It can be used to implement a wide range of efficient data structures, e.g. sequences with access to both ends in amortized constant time and concatenation and splitting in logarithmic time, random access-sequences, search trees, priority queues and more. Functional correctness was verified by Sozeau [182] (Coq) and by Nordhoff *et al.* [155]. The amortized complexity of the deque operations was analysed by Danielsson [50] (Agda).

5.8 Splay Trees

Splay trees [179] are self-adjusting binary search trees where items that have been searched for are rotated to the root of the tree to adjust to dynamically changing access frequencies. Nipkow [145,146] verified functional correctness and amortized logarithmic complexity.

5.9 Braun Trees

Braun trees are binary trees where for each node the size of the left child is the same or one larger than the size of the right child [174,108]. They lend themselves to the implementation of extensible arrays and priority queues in a purely functional manner [162]. They were verified by Nipkow and Sewell [153] in great depth. McCarthy *et al.* [133] demonstrate their Coq library for running time analysis by proving the logarithmic running time of insertion into Braun trees.

6 Advanced Design and Analysis Techniques

6.1 Dynamic Programming

It is usually easy to write down and prove correct the recursive form of a dynamic programming problem, but it takes work to convert it into an efficient implementation by memoizing intermediate results. Wimmer *et al.* [196] automated this process: a recursive function is transformed into a monadic one that memoizes results, and a theorem stating the equivalence of the two functions is proved automatically. The results are stored in a so-called *state monad*. Two state monads were verified: a purely functional state monad based on search trees and the state monad of Imperative-HOL using arrays. The imperative monad yields implementations that have the same asymptotic complexity as the standard array-based algorithms. Wimmer *et al.* verify two further optimizations: bottom-up order of computation and an LRU cache for reduced memory consumption. As applications of their framework, they proved the following algorithms correct (in their recursive form) and translated them into their efficient array-based variants: Bellman-Ford, CYK (context-free parsing), minimum edit distance and optimal binary search trees. Wimmer [195] also treated Viterbi's algorithm in this manner.

Nipkow and Somogyi [154] verified the straightforward recursive cubic algorithm for optimal binary search trees and Knuth's quadratic improvement [113] (but using Yao's simpler proof [199]) and applied memoization.

6.2 Greedy Algorithms

One example of a greedy algorithm given in CLRS is Huffman's algorithm. It was verified by Théry [189] (Coq) and Blanchette [27]. For problems that exhibit a matroid structure, greedy algorithms yield optimal solutions. Keinholz

[112] formalizes matroid theory. Haslbeck *et al.* [96,95] verify the soundness and running time of an algorithm for finding a minimum-weight basis on a weighted matroid and use it to verify Kruskal's algorithm for minimum spanning trees.

7 Advanced Data Structures

7.1 B-Trees

We are aware of two verifications of imperative formulations of B-trees. Malecha *et al.* [131] used Ynot [141], an axiomatic extension to Coq that provides facilities for writing and reasoning about imperative, pointer-based code. The verification by Ernst *et al.* [64] is unusual in that it combines interactive proof in KIV with the automatic shape analysis tool TVLA [175].

7.2 Priority Queues

We start with some priority queue implementations not in CLRS. Priority queues based on Braun trees (see Section 5.9) were verified by Filliâtre [70] (Why3) and Nipkow and Sewell [153]. Two self-adjusting priority queues are the skew heap [180] and the pairing heap [77]. Nipkow and Brinkop [145,148] verified their functional correctness (also verified in Why3 [69,164]) and amortized logarithmic running times. Binomial heaps (covered in depth in the first edition of CLRS) were verified by Meis *et al.* [135] (together with skew binomial heaps [36]), Filliâtre [71] (Why3) and Appel [13] (Coq).

The above heaps are purely functional and do not provide a `decrease-key` operation. Lammich and Nipkow [124] designed and verified a simple, efficient and purely functional combination of a search tree and a priority queue, a "priority search tree". The salient feature of priority search trees is that they offer an operation for updating (not just decreasing) the priority associated with some key; its efficiency is the same as that of the update operation.

Lammich [117] verified an imperative array-based implementation of priority queues with `decrease-key`.

7.3 Union-Find

The union-find data structure for disjoint sets is a frequent case-study: it was formalized in Coq [48,192,176] and Isabelle [122]. Charguéraud, Pottier and Guéneau [42,43,87] were the first to verify the amortized time complexity $O(\alpha(n))$ in Coq using CFML. Their proof follows Alstrup *et al.* [10].

8 Graph Algorithms

8.1 Elementary Graph Algorithms

Graph-searching algorithms are so basic that we only mention a few notable ones. BFS for finding shortest paths in unweighted graphs was verified by participants of a verification competition [76] (in particular in KIV). Lammich and Sefidgar [125] verified BFS for the Edmonds–Karp algorithm. Lammich and Neumann [123] as well as Pottier [169] (Coq) verified DFS and used it for algorithms of different complexity, ranging from a simple cyclicity checker to strongly connected components algorithms. Wimmer and Lammich [198] verified an enhanced version of DFS with subsumption. Bobot [29] verified an algorithm for topological sorting by DFS in Why3.

Strongly Connected Components There are several algorithms for finding strongly connected components (SCCs) in a directed graph. Tarjan's algorithm [186] was verified by Lammich and Neumann [123,142]. Chen *et al.* verified Tarjan's algorithm in Why3, Coq and Isabelle and compared the three formalizations [46,45]. Lammich [115] verified Gabow's algorithm [79] (which was used in a verified model checker [65]), and Pottier [169] (Coq) verified the SCC algorithm featured in CLRS, which is attributed to Kosaraju.

8.2 Minimum Spanning Trees

Prim's algorithm was first verified by Abrial *et al.* [4] in B [3] and on a more abstract level by Lee and Rudnicki [129] (Mizar). Guttmann [89,90] verified a formulation in relation algebra, while Nipkow and Lammich [124] verified a purely functional version.

Kruskal's algorithm was verified by Guttmann [91] using relation algebras. Functional correctness [97] and time complexity [95] of an imperative implementation of Kruskal's algorithm were verified by Haslbeck, Lammich and Biendarra.

8.3 Shortest Paths

The Bellman-Ford algorithm was verified as an instance of dynamic programming (see Section 6.1).

Dijkstra's Algorithm Dijkstra's algorithm has been verified several times. The first verifications were conducted by Chen, Lee and Rudnicki [44,129] (Mizar) and by Moore and Zhang [138] (ACL2). While these formalizations prove the idea of the algorithm, they do not provide efficient implementations. Charguéraud [41] verifies an OCaml version of Dijkstra's algorithm parameterized over a priority queue data structure (without a verified implementation). A notable point is that his algorithm does not require a `decrease-key` operation.

Nordhoff and Lammich [156] use their verified finger trees (see Section 5.7) that support `decrease-key` to obtain a verified functional algorithm. Lammich later refined the functional algorithm down to an imperative implementation using arrays to implement the heap [117]. Zhan [200] also verified the imperative version using his auto2 tool. Finally, Lammich and Nipkow [124] used their red-black tree based priority queues that also support `decrease-key` to obtain a simple functional implementation.

The Floyd–Warshall Algorithm Early verifications by Paulin-Mohring [161] (Coq), Berger *et al.* [22] (Minlog), and Berghofer [23] relied on program extraction from a constructive proof and only targeted the Warshall algorithm for computing the transitive closure of a relation. Filliâtre and Clochard [73] (Why3) verified an imperative implementation of the Warshall algorithm. Wimmer [194] verified the functional correctness of the Floyd–Warshall algorithm for the APSP problem including detection of negative cycles. The main complication is to prove that destructive updates can be used soundly. This and the detection of negative cycles are left as an exercise to the reader in CLRS. The resulting functional implementation (with destructive updates) was later refined to an imperative implementation by Wimmer and Lammich [197].

8.4 Maximum Network Flow

The first verification of the Ford–Fulkerson method, at an abstract level, was by Lee [128] (Mizar). Lammich and Sefidgar [125] verified the Ford–Fulkerson method and refined it down to an imperative implementation of the Edmonds–Karp algorithm. They proved that the latter requires $O(|V| \cdot |E|)$ iterations. On randomly generated networks, their code is competitive with a Java implementation by Sedgewick and Wayne [177]. In further work [126] they verified the generic push–relabel method of Goldberg and Tarjan and refined it to both the relabel-to-front and the FIFO push–relabel algorithm. They also performed a running time analysis and benchmarked their algorithms against C and C++ implementations.

Haslbeck and Lammich [95] provided a proper running time analysis of the Edmonds–Karp algorithm and proved the complexity $O(|V| \cdot |E| \cdot (|V| + |E|))$.

8.5 Matching

Edmonds' famous blossom algorithm [62] for finding maximum matchings in general graphs was verified by Abdulaziz [2].

Hamid and Castelberry [92] (Coq) verified the Gale–Shapley algorithm [80] for finding stable marriages.

9 Selected Topics

9.1 Matrix Operations

Palomo-Lozano *et al.* formalized Strassen's algorithm for matrix multiplication in ACL2 [158], but only for square matrices whose size is a power of two. Dénès *et al.* formalized a slightly more efficient variant of it known as Winograd's algorithm in Coq [51] for arbitrary matrices. Garillot *et al.* formalized the LUP decomposition algorithm from CLRS in Coq [83].

9.2 Linear Programming

The simplex algorithm was formalized by Allamigeon and Katz [9] (Coq) and by Spasić and Marić [183,132]. The latter was repurposed into an incremental algorithm that can emit unsatisfiable cores by Bottesch *et al.* [30]. Parsert and Kaliszyk [159] extended this to a full solver for linear programming.

9.3 Polynomials and FFT

The recursive Fast Fourier Transform was formalized in various systems. We are aware of the formalizations in ACL2 by Gamboa [82], in Coq by Capretta [39], in HOL4 by Akbarpour and Tahar [8] and in Isabelle by Ballarin [16].

9.4 Number-Theoretic Algorithms

Most of the basic number theory shown in CLRS (GCDs, modular arithmetic, Chinese remainder theorem) is available in the standard libraries of various systems and we will therefore not give individual references for this and focus entirely on the algorithms.

Hurd formalized the Miller–Rabin test [109] in HOL4. Stüwe and Eberl [185] formalized Miller–Rabin and some other related tests (Fermat[1] and Solovay–Strassen) in Isabelle. In all cases, what was shown is that a prime is always correctly classified as prime and that a composite is correctly classified with probability at least $\frac{1}{2}$. The running time analysis is not particularly interesting for these algorithms, and although they are randomized algorithms, the randomness is of a very simple nature and thus not very interesting either.

Beyond the primality-testing algorithms in CLRS, Chan [40] gave a HOL4 formalization of the correctness and polynomial running time of the AKS, which was the first deterministic primality test to be proved to run in polynomial time.

9.5 String Matching

The Knuth–Morris–Pratt algorithm was verified by Filliâtre in Coq and Why3 [68,67]. Hellauer and Lammich [99] verified a functional version of this algorithm and refined it to Imperative-HOL. Lammich [118] synthesized verified LLVM code. The Boyer–Moore string searching algorithm [33] was covered in the first edition of CLRS. Boyer and Moore [34] (Nqthm) and Moore and Martinez [137] (ACL2) verified different variants of this algorithm; Toibazarov [190] verified the preprocessing phase of the variant considered by Moore and Martinez. Besta and Stomp [26] (PVS) verified the preprocessing phase of the original algorithm.

[1] The Fermat test is called Pseudoprime in CLRS.

9.6 Computational Geometry

Convex hull algorithms have been popular verification targets: Pichardie and Bertot [166] (Coq) verified an incremental and a package wrapping algorithm, Meikle and Fleuriot [134] verified Graham's scan and Brun *et al.* [37] (Coq) verified an incremental algorithm based on hypermaps. Dufourd and Bertot [53,24] (Coq) verified triangulation algorithms based on hypermaps.

Rau and Nipkow [173] verified the divide-and-conquer closest pair of points algorithm and obtained a competitive implementation.

9.7 Approximation and Online Algorithms

Stucke [184] (Coq and Isabelle) verified an approximation algorithm for vertex colouring in relation algebra. Eßmann *et al.* [66] verified three classic algorithms and one lesser-known one for vertex cover, independent set, load balancing and bin packing. Haslbeck and Nipkow [98] formalized online algorithms and verified several deterministic and randomized algorithms for the list update problem.

9.8 Randomized Algorithms

In addition to the randomized algorithms from CLRS, we will also list some from the classic textbook *Randomized Algorithms* by Motwani and Raghavan [139]. All work was done using PMFs unless stated otherwise (refer to Section 2.1 for a discussion of the various approaches).

The first work on a non-trivial randomized algorithm in an ITP was probably Hurd's [109] previously-mentioned formalization of the Miller–Rabin primality test in HOL (using an infinite stream of random bits to encode the randomness). The primality tests formalized by Stüwe and Eberl [185] are technically also randomized algorithms, but the probabilistic content is very small.

The expected running time of the coupon collector problem was treated by Kaminski *et al.* [110] using their Hoare-style calculus for the pGCL language (on paper). Hölzl [105] formalized their approach in Isabelle.

Barthe *et al.* analyzed several probabilistic textbook problems using a program logic called *Ellora* [19], which is embedded into the EasyCrypt system [18]:

– expected running time of the coupon collector problem
– tail bounds on the running time of Boolean hypercube routing
– probability of incorrectly classifying two different polynomials as equal in probabilistic polynomial equality checking

The correctness of CLRS's RANDOMIZE-IN-PLACE, also known as the Fisher–Yates shuffle, was verified by Eberl [54].

The correctness and expected running time of randomized quicksort was formalized by Van der Weegen and McKinna [193] (Coq) using their "decision tree" approach mentioned earlier and by Eberl [61,58]. Both additionally treated the case of average-case deterministic quicksort: Van der Weegen and McKinna proved that its expected running time is $O(n \log n)$, whereas Eberl additionally proved that it has exactly the same distribution as randomized quicksort.

Eberl [55,61] proved that random binary search trees (BSTs into which elements are inserted in random order) have logarithmic expected height and internal path length. He also proved that the distribution of the internal path length is precisely the same as that of the running time of randomized quicksort.

Haslbeck *et al.* [61,94] formalized randomized treaps [178] and proved that their distribution is precisely equal to that of random BSTs, regardless of which order the elements are inserted. The analysis is particularly noteworthy because it involves continuous distributions of trees, which require a non-trivial amount of measure theory.

Haslbeck and Eberl [93] also defined skip lists [171] and formally analyzed two of the most algorithmically interesting questions about them, namely the expected height and the expected path length to an element.

Tassarotti and Harper [187] developed a general cookbook-style method for proving tail bounds on probabilistic divide-and-conquer algorithms in Coq. They applied this method to the running time of randomized quicksort and the height of random BSTs. Later [188] they used a hybrid approach that combines a program logic for a deeply embedded imperative language with high-level reasoning in Coq to analyze skip lists (restricted to two levels for simplicity).

Acknowledgements We thank Andrew Appel, Gilles Barthe, Arthur Charguéraud, Cyril Cohen, Gidon Ernst, Jean-Christophe Filliâtre, Walter Guttmann, Reiner Hähnle, Peter Lammich, J Moore, Joseph Tassarotti, Laurent Théry, René Thiemann and Simon Wimmer for their help in compiling this survey and Jasmin Blanchette for nitpicking.

This research is supported by DFG Koselleck grant NI 491/16-1.

References

1. Archive of Formal Proofs, `http://www.isa-afp.org`
2. Abdulaziz, M., Mehlhorn, K., Nipkow, T.: Trustworthy graph algorithms (invited talk). In: Rossmanith, P., Heggernes, P., Katoen, J. (eds.) 44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019. LIPIcs, vol. 138, pp. 1:1–1:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), `https://doi.org/10.4230/LIPIcs.MFCS.2019.1`
3. Abrial, J.: The B-book - Assigning Programs to Meanings. Cambridge University Press (1996), `https://doi.org/10.1017/CBO9780511624162`
4. Abrial, J., Cansell, D., Méry, D.: Formal derivation of spanning trees algorithms. In: Bert, D., Bowen, J.P., King, S., Waldén, M. (eds.) ZB 2003: Formal Specification and Development in Z and B. LNCS, vol. 2651, pp. 457–476. Springer (2003), `https://doi.org/10.1007/3-540-44880-2_27`
5. Adams, S.: Efficient sets - A balancing act. J. Funct. Program. **3**(4), 553–561 (1993), `https://doi.org/10.1017/S0956796800000885`
6. Affeldt, R., Cohen, C., Rouhling, D.: Formalization techniques for asymptotic reasoning in classical analysis. Journal of Formalized Reasoning **11**(1), 43–76 (2018), `https://doi.org/10.6092/issn.1972-5787/8124`

7. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, LNCS, vol. 10001. Springer (2016), https://doi.org/10.1007/978-3-319-49812-6
8. Akbarpour, B., Tahar, S.: A methodology for the formal verification of FFT algorithms in HOL. In: Hu, A.J., Martin, A.K. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2004. LNCS, vol. 3312, pp. 37–51. Springer (2004), https://doi.org/10.1007/978-3-540-30494-4_4
9. Allamigeon, X., Katz, R.D.: A formalization of convex polyhedra based on the simplex method. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving, ITP 2017. LNCS, vol. 10499, pp. 28–45. Springer (2017), https://doi.org/10.1007/978-3-319-66107-0_3
10. Alstrup, S., Thorup, M., Gørtz, I.L., Rauhe, T., Zwick, U.: Union-find with constant time deletions. ACM Trans. Algorithms **11**(1), 6:1–6:28 (2014), https://doi.org/10.1145/2636922
11. Andersson, A.: Improving partial rebuilding by using simple balance criteria. In: Dehne, F.K.H.A., Sack, J., Santoro, N. (eds.) Algorithms and Data Structures, WADS '89. LNCS, vol. 382, pp. 393–402. Springer (1989), https://doi.org/10.1007/3-540-51542-9_33
12. Appel, A.W.: Efficient verified red-black trees (2011), https://www.cs.princeton.edu/~appel/papers/redblack.pdf
13. Appel, A.W.: Verified Functional Algorithms (Aug 2018), https://softwarefoundations.cis.upenn.edu/vfa-current/index.html
14. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in Coq. Sci. Comput. Program. **74**(8), 568–589 (2009), https://doi.org/10.1016/j.scico.2007.09.002
15. Avigad, J., Donnelly, K., Gray, D., Raff, P.: A formally verified proof of the prime number theorem. ACM Trans. Comput. Logic **9**(1) (12 2007). https://doi.org/10.1145/1297658.1297660
16. Ballarin, C.: Fast Fourier Transform. Archive of Formal Proofs (Oct 2005), http://isa-afp.org/entries/FFT.html, Formal proof development
17. Bancerek, G., Bylinski, C., Grabowski, A., Kornilowicz, A., Matuszewski, R., Naumowicz, A., Pak, K., Urban, J.: Mizar: State-of-the-art and beyond. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) Intelligent Computer Mathematics, CICM 2015. LNCS, vol. 9150, pp. 261–279. Springer (2015), https://doi.org/10.1007/978-3-319-20615-8_17
18. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.: Easycrypt: A tutorial. In: Aldini, A., López, J., Martinelli, F. (eds.) Foundations of Security Analysis and Design, FOSAD 2012/2013. Lecture Notes in Computer Science, vol. 8604, pp. 146–166. Springer (2013), https://doi.org/10.1007/978-3-319-10082-1_6
19. Barthe, G., Espitau, T., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.: An assertion-based program logic for probabilistic programs. In: Ahmed, A. (ed.) Programming Languages and Systems, ESOP 2018. LNCS, vol. 10801, pp. 117–144. Springer (2018), https://doi.org/10.1007/978-3-319-89884-1_5
20. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK's dual pivot quicksort correct. In: Paskevich and Wies [160], pp. 35–48, https://doi.org/10.1007/978-3-319-72308-2_3
21. Berger, U., Miyamoto, K., Schwichtenberg, H., Seisenberger, M.: Minlog - A tool for program extraction supporting algebras and coalgebras. In: Corradini, A.,

Klin, B., Cîrstea, C. (eds.) Algebra and Coalgebra in Computer Science, CALCO 2011. LNCS, vol. 6859, pp. 393–399. Springer (2011), https://doi.org/10.1007/978-3-642-22944-2_29

22. Berger, U., Schwichtenberg, H., Seisenberger, M.: The Warshall algorithm and Dickson's lemma: Two examples of realistic program extraction. J. Autom. Reasoning **26**(2), 205–221 (2001), https://doi.org/10.1023/A:1026748613865
23. Berghofer, S.: Program extraction in simply-typed higher order logic. In: Geuvers, H., Wiedijk, F. (eds.) Types for Proofs and Programs, TYPES 2002. LNCS, vol. 2646, pp. 21–38. Springer (2002), https://doi.org/10.1007/3-540-39185-1_2
24. Bertot, Y.: Formal verification of a geometry algorithm: A quest for abstract views and symmetry in Coq proofs. In: Fischer, B., Uustalu, T. (eds.) Theoretical Aspects of Computing - ICTAC 2018. LNCS, vol. 11187, pp. 3–10. Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_1
25. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004), https://doi.org/10.1007/978-3-662-07964-5
26. Besta, M., Stomp, F.A.: A complete mechanization of correctness of a string-preprocessing algorithm. Formal Methods Syst. Des. **27**(1-2), 5–17 (2005), https://doi.org/10.1007/s10703-005-2243-0
27. Blanchette, J.C.: Proof pearl: Mechanizing the textbook proof of Huffman's algorithm. J. Autom. Reasoning **43**(1), 1–18 (2009), https://doi.org/10.1007/s10817-009-9116-y
28. Blum, N., Mehlhorn, K.: On the average number of rebalancing operations in weight-balanced trees. Theor. Comput. Sci. **11**, 303–320 (1980), https://doi.org/10.1016/0304-3975(80)90018-3
29. Bobot, F.: Topological sorting (2014), http://toccata.lri.fr/gallery/topological_sorting.en.html, formal proof development
30. Bottesch, R., Haslbeck, M.W., Thiemann, R.: Verifying an incremental theory solver for linear arithmetic in isabelle/hol. In: Herzig, A., Popescu, A. (eds.) Frontiers of Combining Systems, FroCoS 2019. LNCS, vol. 11715, pp. 223–239. Springer (2019), https://doi.org/10.1007/978-3-030-29007-8_13
31. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda - A functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer (2009), https://doi.org/10.1007/978-3-642-03359-9_6
32. Boyer, R.S., Moore, J S.: Proving theorems about LISP functions. In: Nilsson, N. (ed.) Int. Joint Conference on Artificial Intelligence. pp. 486–493. William Kaufmann (1973), http://ijcai.org/Proceedings/73/Papers/053.pdf
33. Boyer, R.S., Moore, J S.: A fast string searching algorithm. Commun. ACM **20**(10), 762–772 (1977), https://doi.org/10.1145/359842.359859
34. Boyer, R.S., Moore, J S.: A computational logic handbook, Perspectives in computing, vol. 23. Academic Press (1979)
35. Boyer, R.S., Moore, J S.: Single-threaded objects in ACL2. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) Practical Aspects of Declarative Languages, PADL 2002. LNCS, vol. 2257, pp. 9–27. Springer (2002), https://doi.org/10.1007/3-540-45587-6_3
36. Brodal, G.S., Okasaki, C.: Optimal purely functional priority queues. J. Funct. Program. **6**(6), 839–857 (1996), https://doi.org/10.1017/S095679680000201X

37. Brun, C., Dufourd, J., Magaud, N.: Designing and proving correct a convex hull algorithm with hypermaps in Coq. Comput. Geom. **45**(8), 436–457 (2012). https://doi.org/10.1016/j.comgeo.2010.06.006
38. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) Theorem Proving in Higher Order Logics, TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer (2008), https://doi.org/10.1007/978-3-540-71067-7_14
39. Capretta, V.: Certifying the fast fourier transform with coq. In: Boulton, R.J., Jackson, P.B. (eds.) Theorem Proving in Higher Order Logics, TPHOLs 2001. LNCS, vol. 2152, pp. 154–168. Springer (2001), https://doi.org/10.1007/3-540-44755-5_12
40. Chan, H.L.J.: Primality Testing is Polynomial-time: A Mechanised Verification of the AKS Algorithm. Ph.D. thesis, Australian National University (2019), https://openresearch-repository.anu.edu.au/bitstream/1885/177195/1/thesis.pdf
41. Charguéraud, A.: Program verification through characteristic formulae. In: Hudak, P., Weirich, S. (eds.) Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010. pp. 321–332. ACM (2010), https://doi.org/10.1145/1863543.1863590
42. Charguéraud, A., Pottier, F.: Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving, ITP 2015. LNCS, vol. 9236, pp. 137–153. Springer (2015), https://doi.org/10.1007/978-3-319-22102-1_9
43. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. J. Autom. Reasoning **62**(3), 331–365 (2019), https://doi.org/10.1007/s10817-017-9431-7
44. Chen, J.C.: Dijkstra's shortest path algorithm. Formalized Mathematics **11**(3), 237–247 (2003), http://fm.mizar.org/2003-11/pdf11-3/graphsp.pdf
45. 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. LIPIcs, vol. 141, pp. 13:1–13:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), https://doi.org/10.4230/LIPIcs.ITP.2019.13
46. Chen, R., Lévy, J.: A semi-automatic proof of strong connectivity. In: Paskevich and Wies [160], pp. 49–65, https://doi.org/10.1007/978-3-319-72308-2_4
47. Clochard, M.: Automatically verified implementation of data structures based on AVL trees. In: Giannakopoulou, D., Kroening, D. (eds.) Verified Software: Theories, Tools and Experiments, VSTTE 2014. LNCS, vol. 8471, pp. 167–180. Springer (2014), https://doi.org/10.1007/978-3-319-12154-3_11
48. Conchon, S., Filliâtre, J.: A persistent union-find data structure. In: Russo, C.V., Dreyer, D. (eds.) Workshop on ML, 2007. pp. 37–46. ACM (2007). https://doi.org/10.1145/1292535.1292541, https://doi.org/10.1145/1292535.1292541
49. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009), http://mitpress.mit.edu/books/introduction-algorithms

50. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008. pp. 133–144. ACM (2008), `https://doi.org/10.1145/1328438.1328457`
51. Dénès, M., Mörtberg, A., Siles, V.: A refinement-based approach to computational algebra in Coq. In: Beringer, L., Felty, A.P. (eds.) Interactive Theorem Proving, ITP 2012. LNCS, vol. 7406, pp. 83–98. Springer (2012), `https://doi.org/10.1007/978-3-642-32347-8_7`
52. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: Barrett, C.W., Davies, M., Kahsai, T. (eds.) NASA Formal Methods, NFM 2017. LNCS, vol. 10227, pp. 68–83 (2017), `https://doi.org/10.1007/978-3-319-57288-8_5`
53. Dufourd, J., Bertot, Y.: Formal study of plane Delaunay triangulation. In: Interactive Theorem Proving, ITP 2010. pp. 211–226 (2010). https://doi.org/10.1007/978-3-642-14052-5_16
54. Eberl, M.: Fisher–yates shuffle. Archive of Formal Proofs (Sep 2016), `http://isa-afp.org/entries/Fisher_Yates.html`, Formal proof development
55. Eberl, M.: Expected shape of random binary search trees. Archive of Formal Proofs (Apr 2017), `http://isa-afp.org/entries/Random_BSTs.html`, Formal proof development
56. Eberl, M.: Lower bound on comparison-based sorting algorithms. Archive of Formal Proofs (Mar 2017), `http://isa-afp.org/entries/Comparison_Sort_Lower_Bound.html`, Formal proof development
57. Eberl, M.: The median-of-medians selection algorithm. Archive of Formal Proofs (Dec 2017), `http://isa-afp.org/entries/Median_Of_Medians_Selection.html`, Formal proof development
58. Eberl, M.: The number of comparisons in quicksort. Archive of Formal Proofs (Mar 2017), `http://isa-afp.org/entries/Quick_Sort_Cost.html`, Formal proof development
59. Eberl, M.: Proving divide and conquer complexities in Isabelle/HOL. Journal of Automated Reasoning **58**(4), 483–508 (4 2017). https://doi.org/10.1007/s10817-016-9378-0
60. Eberl, M.: Verified real asymptotics in Isabelle/HOL. In: Proceedings of the International Symposium on Symbolic and Algebraic Computation. ISSAC '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3326229.3326240
61. Eberl, M., Haslbeck, M.W., Nipkow, T.: Verified analysis of random binary tree structures. In: Journal of Automated Reasoning (2020), `https://doi.org/10.1007/s10817-020-09545-0`
62. Edmonds, J.: Paths, trees, and flowers. Canadian Journal of Mathematics **17**, 449â–467 (1965). https://doi.org/10.4153/CJM-1965-045-4
63. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: overview and verifythis competition. Int. J. Softw. Tools Technol. Transf. **17**(6), 677–694 (2015), `https://doi.org/10.1007/s10009-014-0308-3`
64. Ernst, G., Schellhorn, G., Reif, W.: Verification of B$^+$ trees by integration of shape analysis and interactive theorem proving. Software and Systems Modeling **14**(1), 27–44 (2015), `https://doi.org/10.1007/s10270-013-0320-1`
65. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification, CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer (2013), `https://doi.org/10.1007/978-3-642-39799-8_31`

66. Eßmann, R., Nipkow, T., Robillard, S.: Verified approximation algorithms. In: Peltier and Sofronie-Stokkermans [163], pp. 291–306, https://doi.org/10.1007/978-3-030-51054-1_17
67. Filliâtre, J.C.: Knuth-Morris-Pratt string searching algorithm, http://toccata.lri.fr/gallery/kmp.en.html, formal proof development
68. Filliâtre, J.: Proof of imperative programs in type theory. In: Altenkirch, T., Naraschewski, W., Reus, B. (eds.) Types for Proofs and Programs, TYPES '98. LNCS, vol. 1657, pp. 78–92. Springer (1998), https://doi.org/10.1007/3-540-48167-2_6
69. Filliâtre, J.C.: Skew heaps (2014), http://toccata.lri.fr/gallery/skew_heaps.en.html, formal proof development
70. Filliâtre, J.C.: Purely applicative heaps implemented with Braun trees (2015), http://toccata.lri.fr/gallery/braun_trees.en.html, formal proof development
71. Filliâtre, J.C.: Binomial heaps (2016), http://toccata.lri.fr/gallery/binomial_heap.en.html, formal proof development
72. Filliâtre, J.C., Clochard, M.: Hash tables with linear probing (2014), http://toccata.lri.fr/gallery/linear_probing.en.html, formal proof development
73. Filliâtre, J.C., Clochard, M.: Warshall algorithm (2014), http://toccata.lri.fr/gallery/warshall_algorithm.en.html, formal proof development
74. Filliâtre, J., Letouzey, P.: Functors for proofs and programs. In: Schmidt, D.A. (ed.) Programming Languages and Systems, ESOP 2004. LNCS, vol. 2986, pp. 370–384. Springer (2004), https://doi.org/10.1007/978-3-540-24725-8_26
75. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems, ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer (2013), https://doi.org/10.1007/978-3-642-37036-6_8
76. Filliâtre, J., Paskevich, A., Stump, A.: The 2nd verified software competition: Experience report. In: Proc. 1st Int. Workshop on Comparative Empirical Evaluation of Reasoning Systems. pp. 36–49 (2012), http://ceur-ws.org/Vol-873/papers/paper_6.pdf
77. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The pairing heap: A new form of self-adjusting heap. Algorithmica **1**(1), 111–129 (1986), https://doi.org/10.1007/BF01840439
78. Furia, C.A., Nordio, M., Polikarpova, N., Tschannen, J.: Autoproof: auto-active functional verification of object-oriented programs. Int. J. Softw. Tools Technol. Transf. **19**(6), 697–716 (2017), https://doi.org/10.1007/s10009-016-0419-0
79. Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Inf. Process. Lett. **74**(3-4), 107–114 (2000), https://doi.org/10.1016/S0020-0190(00)00051-X
80. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. The American Mathematical Monthly **69**(1), 9–15 (1962)
81. Galperin, I., Rivest, R.L.: Scapegoat trees. In: Ramachandran, V. (ed.) Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms. pp. 165–174. ACM/SIAM (1993), http://dl.acm.org/citation.cfm?id=313559.313676
82. Gamboa, R.: The correctness of the Fast Fourier Transform: A structured proof in ACL2. Formal Methods Syst. Des. **20**(1), 91–106 (2002), https://doi.org/10.1023/A:1012912614285

83. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer (2009), https://doi.org/10.1007/978-3-642-03359-9_23
84. Giry, M.: A categorical approach to probability theory. In: Categorical Aspects of Topology and Analysis, Lecture Notes in Mathematics, vol. 915, pp. 68–85. Springer (1982). https://doi.org/10.1007/BFb0092872, http://dx.doi.org/10.1007/BFb0092872
85. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Autom. Reasoning **62**(1), 93–126 (2019), https://doi.org/10.1007/s10817-017-9426-4
86. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification, CAV 2015, Proceedings, Part I. LNCS, vol. 9206, pp. 273–289. Springer (2015), https://doi.org/10.1007/978-3-319-21690-4_16
87. Guéneau, A., Charguéraud, A., Pottier, F.: A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In: Ahmed, A. (ed.) Programming Languages and Systems, ESOP 2018. LNCS, vol. 10801, pp. 533–560. Springer (2018), https://doi.org/10.1007/978-3-319-89884-1_19
88. Guibas, L.J., McCreight, E.M., Plass, M.F., Roberts, J.R.: A new representation for linear lists. In: Hopcroft, J.E., Friedman, E.P., Harrison, M.A. (eds.) Proceedings of the 9th Annual ACM Symposium on Theory of Computing. pp. 49–60. ACM (1977), https://doi.org/10.1145/800105.803395
89. Guttmann, W.: Relation-algebraic verification of Prim's minimum spanning tree algorithm. In: Sampaio, A., Wang, F. (eds.) Theoretical Aspects of Computing, ICTAC 2016. LNCS, vol. 9965, pp. 51–68 (2016), https://doi.org/10.1007/978-3-319-46750-4_4
90. Guttmann, W.: An algebraic framework for minimum spanning tree problems. Theor. Comput. Sci. **744**, 37–55 (2018), https://doi.org/10.1016/j.tcs.2018.04.012
91. Guttmann, W.: Verifying minimum spanning tree algorithms with stone relation algebras. J. Log. Algebraic Methods Program. **101**, 132–150 (2018), https://doi.org/10.1016/j.jlamp.2018.09.005
92. Hamid, N.A., Castleberry, C.: Formally certified stable marriages. In: Proceedings of the 48th Annual Southeast Regional Conference. ACM SE '10, ACM (2010), https://doi.org/10.1145/1900008.1900056
93. Haslbeck, M.W., Eberl, M.: Skip lists. Archive of Formal Proofs (Jan 2020), http://isa-afp.org/entries/Skip_Lists.html, Formal proof development
94. Haslbeck, M.W., Eberl, M., Nipkow, T.: Treaps. Archive of Formal Proofs (Feb 2018), http://isa-afp.org/entries/Treaps.html, Formal proof development
95. Haslbeck, M.P.L., Lammich, P.: Refinement with time — refining the run-time of algorithms in Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) Interactive Theorem Proving, ITP 2019. LIPIcs, vol. 141, pp. 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), https://doi.org/10.4230/LIPIcs.ITP.2019.20
96. Haslbeck, M.P.L., Lammich, P., Biendarra, J.: Kruskal's algorithm for minimum spanning forest. Archive of Formal Proofs (Feb 2019), http://isa-afp.org/entries/Kruskal.html, Formal proof development

97. Haslbeck, M.P.L., Lammich, P., Biendarra, J.: Kruskal's algorithm for minimum spanning forest. Arch. Formal Proofs **2019** (2019), `https://www.isa-afp.org/entries/Kruskal.html`
98. Haslbeck, M.P.L., Nipkow, T.: Verified analysis of list update algorithms. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016. LIPIcs, vol. 65, pp. 49:1–49:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.FSTTCS.2016.49, `https://doi.org/10.4230/LIPIcs.FSTTCS.2016.49`
99. Hellauer, F., Lammich, P.: The string search algorithm by knuth, morris and pratt. Archive of Formal Proofs (Dec 2017), `http://isa-afp.org/entries/Knuth_Morris_Pratt.html`, Formal proof development
100. Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M.C.J.D., de Gouw, S.: Verifying OpenJDK's `LinkedList` using KeY. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020, Part II. LNCS, vol. 12079, pp. 217–234. Springer (2020), `https://doi.org/10.1007/978-3-030-45237-7_13`
101. Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. J. Funct. Program. **16**(2), 197–217 (2006), `https://doi.org/10.1017/S0956796805005769`
102. Hirai, Y., Yamamoto, K.: Balancing weight-balanced trees. J. Funct. Program. **21**(3), 287–307 (2011), `https://doi.org/10.1017/S0956796811000104`
103. Hoffmann, J.: Types with potential: polynomial resource bounds via automatic amortized analysis. Ph.D. thesis, Ludwig Maximilians University Munich (2011), `http://edoc.ub.uni-muenchen.de/13955/`
104. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. **34**(3), 14:1–14:62 (2012), `https://doi.org/10.1145/2362389.2362393`
105. Hölzl, J.: Formalising semantics for expected running time of probabilistic programs. In: Blanchette, J.C., Merz, S. (eds.) Interactive Theorem Proving, ITP 2016. pp. 475–482. Springer (2016), `https://doi.org/10.1007/978-3-319-43144-4_30`
106. Hölzl, J., Heller, A.: Three chapters of measure theory in Isabelle/HOL. In: van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Interactive Theorem Proving, ITP 2011. LNCS, vol. 6898, pp. 135–151. Springer (2011), `https://doi.org/10.1007/978-3-642-22863-6_12`
107. Hölzl, J., Immler, F., Huffman, B.: Type classes and filters for mathematical analysis in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving, ITP 2013. LNCS, vol. 7998, pp. 279–294. Springer (2013), `https://doi.org/10.1007/978-3-642-39634-2_21`
108. Hoogerwoord, R.R.: A logarithmic implementation of flexible arrays. In: Bird, R.S., Morgan, C., Woodcock, J. (eds.) Mathematics of Program Construction. LNCS, vol. 669, pp. 191–207. Springer (1992), `https://doi.org/10.1007/3-540-56625-2_14`
109. Hurd, J.: Verification of the Miller–Rabin probabilistic primality test. J. Log. Algebraic Methods Program. **56**(1-2), 3–21 (2003), `https://doi.org/10.1016/S1567-8326(02)00065-6`
110. Kaminski, B.L., Katoen, J.P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run-times of probabilistic programs. In: Programming Languages and Systems, ESOP 2016. pp. 364–389. Springer (2016), `http://dx.doi.org/10.1007/978-3-662-49498-1_15`

111. Kaufmann, M., Moore, J S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
112. Keinholz, J.: Matroids. Archive of Formal Proofs (Nov 2018), `http://isa-afp.org/entries/Matroids.html`, Formal proof development
113. Knuth, D.E.: Optimum binary search trees. Acta Inf. **1**, 14–25 (1971), `https://doi.org/10.1007/BF00264289`
114. Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving, ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer (2013), `https://doi.org/10.1007/978-3-642-39634-2_9`
115. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving, ITP 2014. LNCS, vol. 8558, pp. 325–340. Springer (2014), `https://doi.org/10.1007/978-3-319-08970-6_21`
116. Lammich, P.: Refinement to Imperative/HOL. In: Urban and Zhang [191], pp. 253–269, `https://doi.org/10.1007/978-3-319-22102-1_17`
117. Lammich, P.: Refinement based verification of imperative data structures. In: Avigad, J., Chlipala, A. (eds.) Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs. pp. 27–36. ACM (2016), `https://doi.org/10.1145/2854065.2854067`
118. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) Interactive Theorem Proving, ITP 2019. LIPIcs, vol. 141, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), `https://doi.org/10.4230/LIPIcs.ITP.2019.22`
119. Lammich, P.: Refinement to Imperative HOL. J. Autom. Reasoning **62**(4), 481–503 (2019), `https://doi.org/10.1007/s10817-017-9437-1`
120. Lammich, P.: Efficient verified implementation of Introsort and Pdqsort. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning, IJCAR 2020, Part II. LNCS, vol. 12167, pp. 307–323. Springer (2020), `https://doi.org/10.1007/978-3-030-51054-1_18`
121. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, ITP 2010. LNCS, vol. 6172, pp. 339–354. Springer (2010), `https://doi.org/10.1007/978-3-642-14052-5_24`
122. Lammich, P., Meis, R.: A Separation Logic Framework for Imperative HOL. Archive of Formal Proofs (Nov 2012), `http://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html`, Formal proof development
123. Lammich, P., Neumann, R.: A framework for verifying depth-first search algorithms. In: Leroy, X., Tiu, A. (eds.) Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015. pp. 137–146. ACM (2015), `https://doi.org/10.1145/2676724.2693165`
124. Lammich, P., Nipkow, T.: Proof pearl: Purely functional, simple and efficient priority search trees and applications to Prim and Dijkstra. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) Interactive Theorem Proving, ITP 2019. LIPIcs, vol. 141, pp. 23:1–23:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), `https://doi.org/10.4230/LIPIcs.ITP.2019.23`
125. Lammich, P., Sefidgar, S.R.: Formalizing the Edmonds-Karp algorithm. In: Blanchette, J.C., Merz, S. (eds.) Interactive Theorem Proving, ITP2016. LNCS, vol. 9807, pp. 219–234. Springer (2016), `https://doi.org/10.1007/978-3-319-43144-4_14`

126. Lammich, P., Sefidgar, S.R.: Formalizing network flow algorithms: A refinement approach in Isabelle/HOL. J. Autom. Reasoning **62**(2), 261–280 (2019), https://doi.org/10.1007/s10817-017-9442-4
127. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A.P. (eds.) Interactive Theorem Proving, ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer (2012), https://doi.org/10.1007/978-3-642-32347-8_12
128. Lee, G.: Correctnesss of Ford-Fulkerson's maximum flow algorithm. Formalized Mathematics **13**(2), 305–314 (2005), http://fm.mizar.org/2005-13/pdf13-2/glib_005.pdf
129. Lee, G., Rudnicki, P.: Correctness of Dijkstra's shortest path and Prim's minimum spanning tree algorithms. Formalized Mathematics **13**(2), 295–304 (2005), http://fm.mizar.org/2005-13/pdf13-2/glib_004.pdf
130. Leighton, T.: Notes on better master theorems for divide-and-conquer recurrences (MIT lecture notes) (1996), https://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf
131. Malecha, J.G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010. pp. 237–248. ACM (2010), https://doi.org/10.1145/1706299.1706329
132. Marić, F., Spasić, M., Thiemann, R.: An incremental simplex algorithm with unsatisfiable core generation. Archive of Formal Proofs (Aug 2018), http://isa-afp.org/entries/Simplex.html, Formal proof development
133. McCarthy, J.A., Fetscher, B., New, M.S., Feltey, D., Findler, R.B.: A Coq library for internal verification of running-times. In: Kiselyov, O., King, A. (eds.) Functional and Logic Programming, FLOPS 2016. LNCS, vol. 9613, pp. 144–162. Springer (2016), https://doi.org/10.1007/978-3-319-29604-3_10
134. Meikle, L.I., Fleuriot, J.D.: Mechanical theorem proving in computational geometry. In: Hong, H., Wang, D. (eds.) Automated Deduction in Geometry, ADG 2004. LNCS, vol. 3763, pp. 1–18. Springer (2004). https://doi.org/10.1007/11615798_1
135. Meis, R., Nielsen, F., Lammich, P.: Binomial heaps and skew binomial heaps. Archive of Formal Proofs (Oct 2010), http://isa-afp.org/entries/Binomial-Heaps.html, Formal proof development
136. Meyer, B.: Eiffel: The Language. Prentice-Hall (1991), http://www.eiffel.com/doc/#etl
137. Moore, J S., Martinez, M.: A mechanically checked proof of the correctness of the Boyer-Moore fast string searching algorithm. In: Broy, M., Sitou, W., Hoare, T. (eds.) Engineering Methods and Tools for Software Safety and Security. pp. 267–284. IOS Press (2009)
138. Moore, J S., Zhang, Q.: Proof pearl: Dijkstra's shortest path algorithm verified with ACL2. In: Hurd, J., Melham, T.F. (eds.) Theorem Proving in Higher Order Logics, TPHOLs 2005. LNCS, vol. 3603, pp. 373–384. Springer (2005), https://doi.org/10.1007/11541868_24
139. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, USA (1995)
140. Musser, D.R.: Introspective sorting and selection algorithms. Softw. Pract. Exp. **27**(8), 983–993 (1997), https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-%23

141. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: dependent types for imperative programs. In: Hook, J., Thiemann, P. (eds.) Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008. pp. 229–240. ACM (2008), https://doi.org/10.1145/1411204.1411237
142. Neumann, R.: CAVA — A Verified Model Checker. Ph.D. thesis, Technische Universität München (2017), http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20170616-1342881-1-9
143. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. In: Fischer, P.C., Zeiger, H.P., Ullman, J.D., Rosenberg, A.L. (eds.) Proceedings of the 4th Annual ACM Symposium on Theory of Computing. pp. 137–142. ACM (1972), https://doi.org/10.1145/800152.804906
144. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. SIAM J. Comput. **2**(1), 33–43 (1973), https://doi.org/10.1137/0202005
145. Nipkow, T.: Amortized complexity verified. In: Urban and Zhang [191], pp. 310–324, https://doi.org/10.1007/978-3-319-22102-1_21
146. Nipkow, T.: Automatic functional correctness proofs for functional search trees. In: Blanchette, J.C., Merz, S. (eds.) Interactive Theorem Proving, ITP 2016. LNCS, vol. 9807, pp. 307–322. Springer (2016), https://doi.org/10.1007/978-3-319-43144-4_19
147. Nipkow, T.: Verified root-balanced trees. In: Chang, B.E. (ed.) Programming Languages and Systems, APLAS 2017. LNCS, vol. 10695, pp. 255–272. Springer (2017), https://doi.org/10.1007/978-3-319-71237-6_13
148. Nipkow, T., Brinkop, H.: Amortized complexity verified. J. Autom. Reasoning **62**(3), 367–391 (2019), https://doi.org/10.1007/s10817-018-9459-3
149. Nipkow, T., Dirix, S.: Weight-balanced trees. Archive of Formal Proofs (Mar 2018), http://isa-afp.org/entries/Weight_Balanced_Trees.html, Formal proof development
150. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer (2014), https://doi.org/10.1007/978-3-319-10542-0, http://www.concrete-semantics.org/
151. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002), https://doi.org/10.1007/3-540-45949-9
152. Nipkow, T., Pusch, C.: AVL trees. Archive of Formal Proofs (Mar 2004), http://isa-afp.org/entries/AVL-Trees.html, Formal proof development
153. Nipkow, T., Sewell, T.: Proof pearl: Braun trees. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020. pp. 18–31. ACM (2020), https://doi.org/10.1145/3372885.3373834
154. Nipkow, T., Somogyi, D.: Optimal binary search trees. Archive of Formal Proofs (May 2018), http://isa-afp.org/entries/Optimal_BST.html, Formal proof development
155. Nordhoff, B., Körner, S., Lammich, P.: Finger trees. Archive of Formal Proofs (Oct 2010), http://isa-afp.org/entries/Finger-Trees.html, Formal proof development
156. Nordhoff, B., Lammich, P.: Dijkstra's shortest path algorithm. Archive of Formal Proofs (Jan 2012), http://isa-afp.org/entries/Dijkstra_Shortest_Path.html, Formal proof development

157. Owre, S., Shankar, N.: A brief overview of PVS. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) Theorem Proving in Higher Order Logics, TPHOLs 2008. LNCS, vol. 5170, pp. 22–27. Springer (2008), https://doi.org/10.1007/978-3-540-71067-7_5
158. Palomo-Lozano, F., Inmaculada Medina-Bulo, J.A.A.J.: Certification of matrix multiplication algorithms. In: Theorem Proving in Higher Order Logics, Supplemental Proceedings, TPHOLs 2001 (2001)
159. Parsert, J., Kaliszyk, C.: Linear programming. Archive of Formal Proofs (Aug 2019), http://isa-afp.org/entries/Linear_Programming.html, Formal proof development
160. Paskevich, A., Wies, T. (eds.): Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers, LNCS, vol. 10712. Springer (2017), https://doi.org/10.1007/978-3-319-72308-2
161. Paulin-Mohring, C.: Extraction de programmes dans le Calcul des Constructions. (Program Extraction in the Calculus of Constructions). Ph.D. thesis, Paris Diderot University, France (1989), https://tel.archives-ouvertes.fr/tel-00431825
162. Paulson, L.C.: ML for the Working Programmer. Cambridge University Press, 2nd edn. (1996)
163. Peltier, N., Sofronie-Stokkermans, V. (eds.): Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II, LNCS, vol. 12167. Springer (2020), https://doi.org/10.1007/978-3-030-51054-1
164. Pereira, M.: Pairing heaps (2016), http://toccata.lri.fr/gallery/pairing_heap.en.html, formal proof development
165. Petcher, A., Morrisett, G.: The foundational cryptography framework. In: Focardi, R., Myers, A.C. (eds.) Principles of Security and Trust, POST 2015. LNCS, vol. 9036, pp. 53–72. Springer (2015), https://doi.org/10.1007/978-3-662-46666-7_4
166. Pichardie, D., Bertot, Y.: Formalizing convex hull algorithms. In: Boulton, R.J., Jackson, P.B. (eds.) Theorem Proving in Higher Order Logics, TPHOLs 2001. LNCS, vol. 2152, pp. 346–361. Springer (2001). https://doi.org/10.1007/3-540-44755-5_24
167. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. In: Bjørner, N., de Boer, F.S. (eds.) FM 2015: Formal Methods. LNCS, vol. 9109, pp. 414–434. Springer (2015), https://doi.org/10.1007/978-3-319-19249-9_26
168. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. Formal Asp. Comput. **30**(5), 495–523 (2018), https://doi.org/10.1007/s00165-017-0435-1
169. Pottier, F.: Depth-first search and strong connectivity in Coq. In: Journées Francophones des Langages Applicatifs (JFLA) (Jan 2015), http://gallium.inria.fr/~fpottier/publis/fpottier-dfs-scc.pdf
170. Pottier, F.: Verifying a hash table and its iterators in higher-order separation logic. In: ACM SIGPLAN Conference on Certified Programs and Proofs (CPP). pp. 3–16 (Jan 2017). https://doi.org/https://doi.org/10.1145/3018610.3018624, http://gallium.inria.fr/~fpottier/publis/fpottier-hashtable.pdf
171. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990), https://doi.org/10.1145/78973.78977

172. Ralston, R.: ACL2-certified AVL trees. In: Eighth International Workshop on the ACL2 Theorem Prover and Its Applications. pp. 71–74. ACL2 '09, ACM (2009), https://doi.org/10.1145/1637837.1637848
173. Rau, M., Nipkow, T.: Verification of closest pair of points algorithms. In: Peltier and Sofronie-Stokkermans [163], pp. 341–357, https://doi.org/10.1007/978-3-030-51054-1_20
174. Rem, M., Braun, W.: A logarithmic implementation of flexible arrays (1983), memorandum MR83/4. Eindhoven University of Techology
175. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002), https://doi.org/10.1145/514188.514190
176. Sakaguchi, K.: Program extraction for mutable arrays. In: Gallagher, J.P., Sulzmann, M. (eds.) Functional and Logic Programming, FLOPS 2018. LNCS, vol. 10818, pp. 51–67. Springer (2018). https://doi.org/10.1007/978-3-319-90686-7_4, https://doi.org/10.1007/978-3-319-90686-7_4
177. Sedgewick, R., Wayne, K.: Algorithms, 4th Edition. Addison-Wesley (2011)
178. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica **16**(4/5), 464–497 (1996), https://doi.org/10.1007/BF01940876
179. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM **32**(3), 652–686 (1985), https://doi.org/10.1145/3828.3835
180. Sleator, D.D., Tarjan, R.E.: Self-adjusting heaps. SIAM J. Comput. **15**(1), 52–69 (1986), https://doi.org/10.1137/0215004
181. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) Theorem Proving in Higher Order Logics, TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer (2008), https://doi.org/10.1007/978-3-540-71067-7_6
182. Sozeau, M.: Program-ing finger trees in Coq. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007. pp. 13–24. ACM (2007), https://doi.org/10.1145/1291220.1291156
183. Spasić, M., Marić, F.: Formalization of incremental simplex algorithm by stepwise refinement. In: Giannakopoulou, D., Méry, D. (eds.) Formal Methods, FM 2012. LNCS, vol. 7436, pp. 434–449. Springer (2012), https://doi.org/10.1007/978-3-642-32759-9_35
184. Stucke, I.: Reasoning about cardinalities of relations with applications supported by proof assistants. In: Höfner, P., Pous, D., Struth, G. (eds.) Relational and Algebraic Methods in Computer Science, RAMiCS 2017. Lecture Notes in Computer Science, vol. 10226, pp. 290–306 (2017), https://doi.org/10.1007/978-3-319-57418-9_18
185. Stüwe, D., Eberl, M.: Probabilistic primality testing. Archive of Formal Proofs (Feb 2019), http://isa-afp.org/entries/Probabilistic_Prime_Tests.html, Formal proof development
186. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972), https://doi.org/10.1137/0201010
187. Tassarotti, J., Harper, R.: Verified tail bounds for randomized programs. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving. Springer (2018)
188. Tassarotti, J., Harper, R.: A separation logic for concurrent randomized programs. Proc. ACM Program. Lang. **3**(POPL), 64:1–64:30 (2019), https://doi.org/10.1145/3290377
189. Théry, L.: Formalising Huffman's algorithm. Research report, Università degli Studi dell'Aquila (2004), https://hal.archives-ouvertes.fr/hal-02149909

190. Toibazarov, E.: An ACL2 proof of the correctness of the preprocessing for a variant of the Boyer-Moore fast string searching algorithm. Honors thesis, Computer Science Dept., University of Texas at Austin (2013), see www.cs.utexas.edu/users/moore/publications/toibazarov-thesis.pdf
191. Urban, C., Zhang, X. (eds.): Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings, LNCS, vol. 9236. Springer (2015), https://doi.org/10.1007/978-3-319-22102-1
192. Vafeiadis, V.: Adjustable references. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving ITP 2013. LNCS, vol. 7998, pp. 328–337. Springer (2013). https://doi.org/10.1007/978-3-642-39634-2_24, https://doi.org/10.1007/978-3-642-39634-2_24
193. van der Weegen, E., McKinna, J.: A machine-checked proof of the average-case complexity of quicksort in Coq. In: Berardi, S., Damiani, F., de'Liguoro, U. (eds.) Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers. LNCS, vol. 5497, pp. 256–271. Springer (2008). https://doi.org/10.1007/978-3-642-02444-3_16, https://doi.org/10.1007/978-3-642-02444-3_16
194. Wimmer, S.: Formalized timed automata. In: Blanchette, J.C., Merz, S. (eds.) Interactive Theorem Proving, ITP 2016. LNCS, vol. 9807, pp. 425–440. Springer (2016), https://doi.org/10.1007/978-3-319-43144-4_26
195. Wimmer, S.: Hidden Markov models. Archive of Formal Proofs (May 2018), http://isa-afp.org/entries/Hidden_Markov_Models.html, Formal proof development
196. Wimmer, S., Hu, S., Nipkow, T.: Verified memoization and dynamic programming. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving, ITP2018. LNCS, vol. 10895, pp. 579–596. Springer (2018), https://doi.org/10.1007/978-3-319-94821-8_34
197. Wimmer, S., Lammich, P.: The Floyd-Warshall algorithm for shortest paths. Archive of Formal Proofs (May 2017), http://isa-afp.org/entries/Floyd_Warshall.html, Formal proof development
198. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018, Part I. LNCS, vol. 10805, pp. 61–78. Springer (2018), https://doi.org/10.1007/978-3-319-89960-2_4
199. Yao, F.F.: Efficient dynamic programming using quadrangle inequalities. In: Miller, R.E., Ginsburg, S., Burkhard, W.A., Lipton, R.J. (eds.) Proceedings of the 12th Annual ACM Symposium on Theory of Computing. pp. 429–435. ACM (1980), https://doi.org/10.1145/800141.804691
200. Zhan, B.: Efficient verification of imperative programs using Auto2. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018. LNCS, vol. 10805, pp. 23–40. Springer (2018), https://doi.org/10.1007/978-3-319-89960-2_2
201. Zhan, B., Haslbeck, M.P.L.: Verifying asymptotic time complexity of imperative programs in Isabelle. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Automated Reasoning, IJCAR 2018. LNCS, vol. 10900, pp. 532–548. Springer (2018), https://doi.org/10.1007/978-3-319-94205-6_35