Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Formally Proving Size Optimality of Sorting Networks

Luís Cruz-Filipe · Kim S. Larsen · Peter Schneider-Kamp

Received: / Accepted:

Abstract Recent successes in formally verifying increasingly larger computer-generated proofs have relied extensively on (a) using oracles, to find answers for recurring subproblems efficiently, and (b) extracting formally verified checkers, to perform exhaustive case analysis in feasible time.

In this work we present a formal verification of optimality of sorting networks on up to 9 inputs, making it one of the largest computer-generated proofs that has been formally verified. We show that an adequate pre-processing of the information provided by the oracle is essential for feasibility, as it improves the time required by our extracted checker by several orders of magnitude.

Keywords interactive theorem proving, large-scale proofs, program extraction, sorting networks

Mathematics Subject Classification (2000) 68T15,68Q60,68N18,68Q25,68Q17

1 Introduction

Although it was not the first computer-assisted proof, the proof of the four color theorem from 1976 [1,3] was the first to generate broad awareness of a new area of mathematics, sometimes dubbed "experimental" or "computational" mathematics, where computers play an essential role. Since then, numerous theorems in mathematics and computer science have been established via computer-assisted and computer-generated proofs.

Besides obvious philosophical debates about what constitutes a mathematical proof, concerns about the validity of such proofs have been raised often since. In particular,

Partially supported by the Danish Council for Independent Research, Natural Sciences, grant DFF-1323-00247.

Department of Mathematics and Computer Science
University of Southern Denmark
Campusvej 55
5230 Odense M
E-mail: {lcf,kslarsen,petersk}@imada.sdu.dk

proofs based on exhausting the solution space were originally met with skepticism [2]; criticism target in particular the correctness and verifiability of the computer program involved.

Nevertheless, computer-assisted proofs have become more and more present. Especially during the last decade, we have seen an increasing use of verified proof assistants to create formally verified computer-generated proofs. This has been a success story, and it has resulted in a plethora of formalizations of mathematical proofs – including a formal verification of the four color theorem from 2005 [22]. A common element to many of these proofs is that they include an exhaustive case analysis too large to be handled manually. The original proof of the four color theorem required analyzing 1,936 different "maps", obtained by reducing around 1 billion possibilities – a number of cases that is substantially surpassed by more recent proofs, e.g. [30,38].

Outside the world of formal proofs, computer-generated proofs are flourishing, too, and growing to tremendous sizes. The proof of Erdős' discrepancy conjecture for $C = 2$ from 2014 [28] has been touted as one of the largest mathematical proofs, and produced approximately 13 GB of proof witnesses. Another recent example, by two of the authors of this article, is the computer-generated proof of the optimality of 25 comparisons when sorting 9 inputs [10,11], which analyzed trillions of cases and resulted in over 70 million proof witnesses, totaling 27 GB of data.

Such large-scale computer-generated proofs are extremely challenging for formal verification. Given the current state of theorem provers and computing equipment, it is prohibitive due to memory and run-time constraints to use approaches such as Claret et al.'s [9] of importing an *oracle* based on the proof witnesses into the theorem prover Coq [4] for the scale of proofs we consider. For this reason, *untrusted oracles* have appeared in recent years. Here, the verified proof tool is relegated to a checker of the computations of the untrusted oracle, e.g., the hand-written untrusted code can be programmed in an efficient programming language to compute a result, and verified (extracted), slower code then merely checks results before continuing the computation.

In this article, we apply these techniques to verify our computer-assisted proof of optimality of 25-comparator sorting networks on 9 channels. In order to complete this task successfully, we go one step further than previous authors, and further modularize the interaction between the untrusted computational tool and the trusted extracted code by logging the results of the former into a file that is further processed before being used by the latter to reconstitute a formal proof. Such an approach allows us to rearrange the data from the original computation in such a way that the checker's performance is optimized. While we use Coq as the theorem prover underlying our presentation, our ideas should be portable to other formal generic proof environments, whether based on dependent type theory (e.g. Agda [34]) or not (e.g. Isabelle/HOL [33] or HOL Light [23]).

The use of checkers that use oracles in formal proofs is briefly surveyed in Section 2, with emphasis on the works most relevant to our endeavour. Our problem is then stated in Section 3, which summarizes the underlying theory and the proof from [11] that we aim to formalize, as well as including the NP-hardness proofs that justify using an oracle. The formalization itself is the topic of Section 4. We conclude in Section 5 with an outlook of how our ideas can be applied to extend the power of present-day computer-assisted proofs.

This work extends material previously published in [14,15]. The complete development is available at `http://www.imada.sdu.dk/~petersk/sn/`.

2 Background and related work

The Curry–Howard correspondence states that every constructive proof of an existential statement embodies an algorithm to produce a witness of the required property. This correspondence has been made more precise by the development of program extraction mechanisms for the most popular theorem provers. In this work, we apply the mechanism described in [31] to extract a correct-by-construction program from a Coq formalization. Coq's program extraction feature allows targeting Ocaml, Haskell or Scheme. In this work, we target the programming language Haskell due to its native call-by-need/laziness: the heavy memory requirements that our extracted program has make it unfeasible to try to execute it with an eager evaluation strategy.

Early experiments of program extraction from a large-scale formalization that was built from a purely mathematical perspective demonstrated, however, that it is unreasonable to expect *efficient* program extraction as a side result of formalizing textbook proofs [13]. In spite of that, one can actually develop mathematically-minded formalizations that yield efficient extracted programs with only minor attention to definitions [29, 35]. This is in contrast to formalizations built with extraction as a primary goal, such as those in the CompCert project [30], or strategies that potentially compromise the validity of the extracted program (e.g. using imperative data structures as in [36]).

The program we extract uses information from an external source in the form of an *oracle*. For the purpose of this work, an oracle is a (partial) function from a set of problems to a set of answers – more concretely, a computer program that attempts to provide an answer (not necessarily correct) for any problem in its domain. If an oracle is total in the set of problems for which an answer exists, we call it an *omniscient* oracle. In formal verification, oracles come in two flavours: *trusted* and *untrusted*.

Trusted Oracles. A trusted oracle is one whose answers are assumed to be correct without verification. A typical example of using a trusted oracle is delegating a proof obligation to a SAT or SMT solver, and having the theorem prover blindly trust its output. There are different reasons for trusting an oracle's results; a particular case is that of using an external tool that has itself been formally verified, such as a formally certified SAT solver [17]. However, when combined with constructive theory-based theorem provers, such as Coq, this approach also requires adding explicit assumptions on the soundness of the oracle data. Similarly, Isabelle/HOL can be made to accept output of external tools on blind faith [19].

Untrusted Oracles. When using untrusted oracles, the answers provided are not taken at face value, and, consequently, they need to be enriched with sufficient information for the theorem prover to check them. The key observation, as already pointed out in [24], is that checking a proof is much more efficient than finding it. In the case of SAT solving, a simple way to do this is to provide a concrete valuation together with the answer that a given formula is satisfiable, allowing a simple verification of this fact by the theorem prover. Whether used as a trusted or untrusted oracle, SAT solvers are arguably omniscient oracles.

Another popular approach is for the untrusted oracle to provide an explicit proof term [16] or a proof in a formal language that can be translated to the languages of different theorem provers [5]. Recent years have seen untrusted oracles be used for a verified compiler [30] and for polyhedral analysis [21], for example. In both cases, the verified proof tool is relegated to a checker of the computations of the untrusted oracle, typically by using hand-written untrusted code to compute a result and verified (extracted) code to check it before continuing the computation.

The termination proof certification projects IsaFoR/CeTA [39], based on the theorem prover Isabelle/HOL, and A3PAT [12], based on Coq, go one step further. Here different termination analyzers provide proof witnesses, which are stored in a common format [5] and later checked. The termination analyzers are typically developed by one research group, while the verified checkers for the proof witnesses are developed by another group. Likewise, there is no coupling at runtime, with the search for a proof witness often being separate both in time and physical domain.

However, a typical termination proof has only 10–100 proof witnesses and totals a few KB to a few MB of data, and recent work [38] mentions that problems were encountered when dealing with proofs using "several hundred megabytes" of oracle data. In that particular case, they avoided having to deal with such amounts of data, however, by reducing the number of proof witnesses for this class of problems. In contrast, the proof we consider (presented in Section 3) requires dealing with 70 million proof witnesses, totaling more than 27 GB of oracle data *after* aggressively reducing the number of cases to be considered. Data sizes continue to increase; the recent proof of the Boolean Pythagorean Triples conjecture [25] generated 200 TB of proof witnesses.

3 The Problem

The goal of this work is to develop a formal verification of the recent computer-generated proofs of size optimality of sorting networks with up to 9 inputs, which were obtained by an *ad-hoc* computer program [10,11], co-authored by two of the authors of this article. In this section, we describe the problem domain and the particular solution, and afterwards we justify our choice of verifying it using a verified checker relying on an untrusted oracle.

3.1 The optimal-size sorting network problem

Sorting networks are hardware-oriented algorithms for sorting a fixed number n of inputs, given on n distinct channels, using a predetermined sequence of comparisons between them. They are built from a primitive operator, the *comparator*, which reads the values on two given channels, and interchanges them if necessary to guarantee that the smallest one is always on a predetermined channel. A *comparator network* C of size k is a sequence of k comparators. If C_1 and C_2 are comparator networks, then $C_1;C_2$ denotes the comparator network obtained by concatenating C_1 and C_2. Comparator networks are often represented as *Knuth diagrams* [27], where channels are represented as horizontal lines and comparators as vertical lines between them, ordered from left to right.

Figure 1 depicts a comparator network on 5 channels as it operates on the *input* $\langle 9,2,7,4,8\rangle$ provided on the left side. The *output* of this network on this input is $\langle 2,4,7,8,9\rangle$, available on the right side. In general, a comparator network C can be seen as a function, mapping input vectors into output vectors, and we say that C is a *sorting network* if this function maps each input to its ascendingly sorted version. This can be decided in finite time in light of the following result.

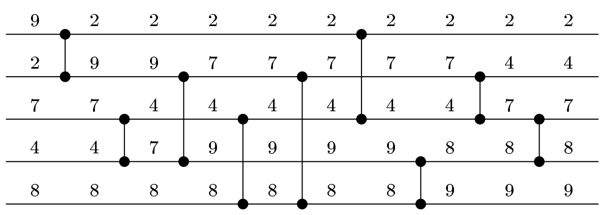

Fig. 1 A sorting network on 5 channels, operating on the input $\langle 9, 2, 7, 4, 8\rangle$.

Lemma 1 (Zero-one principle [27]) *A comparator network C is a sorting network on n channels if and only if C sorts all binary sequences of length n.*

We denote the output of the network for an input $\mathbf{x} \in \{0,1\}^n$ as $C(\mathbf{x})$.

Definition 1 The set of *outputs of* C is $\mathsf{outputs}(C) = \{C(\mathbf{x}) \mid \mathbf{x} \in \{0,1\}^n\}$.

We can rephrase the zero-one principle as stating that C is a sorting network if and only if all elements of $\mathsf{outputs}(C)$ are sorted.

All the comparators in the network in Figure 1 sort their inputs ascendingly. Such comparators are called *standard* comparators, and networks that only contain standard comparators are called *standard comparator networks*. However, there are situations when it pays off to use comparators that sort their inputs descendingly, even when the final output is sorted ascendingly. In practice, such *generalized comparator networks* may be more efficient due to the physical structure of the network (applying a compartor to two channels in close proximity is faster than when they are further apart). In theory, generalized comparator networks naturally arise in the proofs of some results, and we need to consider them in our formalization. Exercise 5.3.4.16 in [27] presents a *standardization* algorithm to construct a standard sorting network from a generalized sorting network that we discuss in more detail in Section 4.

The optimization problem we consider is the following: how many comparators do we need to sort n inputs? This question is known as the *optimal-size problem* in the literature. Most known answers to this problem rely on exhaustive analysis of state spaces. For 5 inputs, the state space is small enough to be exhausted by manual inspection and symmetry arguments [20]. For 7 inputs, a similar analysis was first performed by an *ad-hoc* computer program, also described in [20]. For 9 inputs, a similar approach eventually succeeded [11] – albeit nearly 50 years later. Optimality results for 6, 8, and 10 inputs follow by a theorem of Van Voorhis (Theorem 1 below), and therefore also depend on the soundness of the exhaustive case analysis. We denote by $S(n)$ the size of the smallest sorting network on n inputs; the known values of $S(n)$ are given in Table 1.

Theorem 1 (from [40]) *For all $n \geq 3$, $S(n+1) \geq S(n) + \lceil \log_2(n) \rceil$.*

n	1	2	3	4	5	6	7	8	9	10
$S(n)$	0	1	3	5	9	12	16	19	25	29

Table 1 Known sizes of optimal sorting networks on n inputs.

Conceptually, solving the optimal-size problem on n channels reduces to exhaustively generating all standard n-channel comparator networks of size $S(n) - 1$ and checking that none of them is a sorting network. However, even for small n such a naive approach is combinatorically infeasible. There are $n(n-1)/2$ standard comparators on n channels, and hence $(n(n-1)/2)^k$ networks with k comparators. For $n = 9$, this approach amounts to inspecting approximately 2.25×10^{37} comparator networks of size 24, which is clearly prohibitive. Instead, the strategy followed in [11] (which recovers ideas from [20]) is an iterative approach, *generate-and-prune*, which constructs a set of standard comparator networks one comparator at a time and uses symmetries to eliminate networks that need not be considered.

Definition 2 Let C_a and C_b be two comparator networks on n channels. We say that C_a subsumes C_b, and write $C_a \preceq C_b$, if there exists a permutation π of $1, \ldots, n$ such that $\pi(\mathsf{outputs}(C_a)) \subseteq \mathsf{outputs}(C_b)$, where permutations are lifted to sequences and sets of sequences in the natural way.

We write $C_a \preceq_\pi C_b$ when we need to make π explicit.

Lemma 2 ([11], adapted from [7]) *Let C_a and C_b be comparator networks of the same size on n channels such that $C_a \preceq C_b$. If there exists a sorting network $C_b; C$ of size k, then there exists a sorting network $C_a; C'$ of size k.*

The optimality proof is then obtained by a program that works as follows. First, it initializes R_0^n to consist of a single element: the empty comparator network. Then, it repeatedly applies two types of steps, `Generate` and `Prune`, as follows.

1. `Generate`: Given R_k^n, construct N_{k+1}^n by appending one comparator to each element of R_k^n in all possible ways.
2. `Prune`: Given N_{k+1}^n, construct R_{k+1}^n such that every element of N_{k+1}^n is subsumed by an element of R_{k+1}^n.

The algorithm stops when a sorting network is found.

Throughout execution, the two sets R_k^n and N_k^n always contain comparator networks of size k on n channels. To implement `Prune`, we loop on N_k^n and check whether the current network is subsumed by any of the previous ones; if this is the case, we ignore it. Otherwise, we add it to R_k^n, and remove any networks already in this set that are subsumed by it. This yields a double loop over N_k^n where at each iteration we need to determine whether or not a subsumption exists – which, in the worst case, requires looping through all $n!$ permutations for each pair of networks. For $n = 9$, the largest set N_k^n is N_{15}^9, with over 18 million elements, thus there are potentially 300×10^{12} subsumptions to test. The precise sizes of N_k^9 and R_k^9 are given in Table 2; the number of outputs that needs to be considered in the subsumption tests can be found in Table 3.

Soundness of generate-and-prune follows from the observation that all N_k^n and R_k^n are *complete* for the optimal-size sorting network problem on n channels, in the following sense: if there exists an optimal-size sorting network on n channels, then there exists one of the form $C; C'$ for some $C \in N_k^n$ (or $C \in R_k^n$), for every k.

Successful execution of this algorithm for different values of n, as described in [11], confirmed all known values of $S(n)$ for $3 \leq n \leq 8$, and was able to obtain the new value $S(9) = 25$; since [27] includes a 29-comparator sorting network on 10 channels, Theorem 1 implies that $S(10) = 29$.

k	0	1	2	3	4	5	6	7	8	9	10
$\lvert N^9_k\rvert$	–	36	35	102	231	639	1,824	6,214	23,268	94,827	382,523
$\lvert R^9_k\rvert$	0	1	3	7	20	59	208	807	3,415	14,343	55,991

k	11	12	13	14	15	16
$\lvert N^9_k\rvert$	1,428,794	4,586,075	11,272,878	18,420,674	18,264,160	11,081,077
$\lvert R^9_k\rvert$	188,730	490,322	854,638	914,444	607,164	274,212

k	17	18	19	20	21	22	23	24	25
$\lvert N^9_k\rvert$	4,504,484	1,367,643	323,600	59,428	8,893	1,413	268	58	8
$\lvert R^9_k\rvert$	94,085	25,786	5,699	1,107	250	73	27	8	1

Table 2 Sizes of the sets R^9_k and N^9_k for $1 \leq k \leq 25$.

k	0	1	2	3	4	5	6	7	8	9	10	11
min	512	384	288	216	162	135	108	90	72	60	50	45
med	512	384	320	256	200	168	138	116	100	86	74	64
max	512	384	320	256	224	192	164	144	128	116	108	94

k	12	13	14	15	16	17	18	19	20	21	22	23	24	25
min	40	35	30	26	23	21	19	17	15	14	13	12	11	10
med	56	48	42	36	31	28	24	22	18	16	14	12	11	10
max	80	74	60	51	46	38	34	28	24	20	15	13	11	10

Table 3 Minimum, median, and maximum number of outputs of the networks in the sets N^9_k.

3.2 On the use of an oracle

In order to obtain formal verifications of the values of $S(n)$, we make use of an untrusted oracle. We justify this choice by discussing the extent to which the complexity of the original proof depends on search steps that can be bypassed. In order to make some of our analyses more precise, we focus on the concrete verification of $S(9) = 25$. The concrete problem we study is thus: how can we formally verify this value using a generate-and-prune approach?

The algorithm presented earlier includes two steps that consist of solving existential subproblems: deciding whether a network subsumes another; determining whether a comparator network is a sorting network. We discuss each of them in turn.

In `Prune`, we remove comparator networks from a set based on subsumption. The basic existential problem here is: given two comparator networks, does one of them subsume the other? This is a particular case of what we coin the *Binary String Sets Permutation Inclusion Problem.*

Instance: Two sets S and S' of binary strings of length k for some constant integer $k \geq 1$.
Question: Does there exist a permutation π such that $\forall x \in S\colon \pi(x) \in S'$?

We include a novel proof that this problem is NP-complete:

Theorem 2 *The Binary String Sets Permutation Inclusion problem is NP-complete.*

Proof Inclusion in NP is easy, given a permutation as a certificate. For the hardness part, we reduce from 3-SAT. Consider a 3-SAT formula with n clauses and m propositional variables. We assume without loss of generality that all clauses have exactly three literals and that no clause has more than one instance of any given propositional variable. In the following, we specify binary strings for both S and S', using positions in the strings starting from zero. Any position that we leave unspecified should be zero. All strings have length $k = 2m + 2 + 2(m + n)$.

For intuition, the first $2m$ position are used for encoding the truth values of the m propositional variables. The next two bits are used to get all strings up to the same number of ones before padding. This will be clear below, but these two bits will be either "00" or "11". This is because the representation of a propositional variable uses one one-bit and the representation of a clause uses three one-bits. Thus, these two bits can be used to always get up to three one-bits. The final $2(m+n)$ positions are used to pad strings with a different number of ones. Since a permutation preserves the number of ones, together with the arrangement above of always getting up to three ones, this can be used to control which strings in S' a given string in S can be mapped to.

We assume that the propositional variables are $p_1, \ldots, p_m$. For each propositional variable p_i, we create a string s_i^f in S with "01" in the positions $2(i-1)$ and $2(i-1)+1$, "11" in positions $2m$ and $2m + 1$, and ones in the last $2i$ positions.

As an example, if $m = 5$ and $n = 3$, then s_2^f is

$$00|01|00|00|00||11||0000000000001111\,,$$

where the vertical lines are included only to improve readability.

We also let s_i^f belong to S' together with an additional string, s_i^t, exactly like the one just described, except that we have "10" in positions $2(i-1)$ and $2(i-1)+1$.

Continuing the example, s_2^t is

$$00|10|00|00|00||11||0000000000001111\,.$$

Thus, s_i^f is in S, and both s_i^f and s_i^t are in S'. Furthermore, the strings s_i^f and s_i^t are the only ones in S or S' having exactly $3 + 2i$ ones. Since permutations preserve the number of ones, if there should exist a permutation π with the properties above, then it is necessary that either $\pi(s_i^f) = s_i^f$ or $\pi(s_i^f) = s_i^t$. In the reduction, the first choice corresponds to the truth assignment where p_i is false and the second to the one where p_i is true.

Assume that the clauses are C_1 through C_n. Now, for each clause, we make the following string for S. Assume that clause C_j contains propositional variables p_{h_q}, for $q \in \{1, 2, 3\}$. We make the string s_j^c by placing "01" in positions $2(h_q - 1)$ and $2(h_q - 1) + 1$, for $q \in \{1, 2, 3\}$, and ones in the last $2(j + m)$ positions.

Continuing the example, if clause C_2 is $(p_2 \vee \overline{p}_3 \vee p_5)$, we would get the string

$$00|01|01|00|01||00||0011111111111111\,.$$

Clause C_j gives rise to 7 strings in S'. Considering string s_j^c and the possibility of independently swapping the "01" occurrences to "10" for each of the propositional variables in the clause, this would give rise to 8 strings. We include all of these except one: The string where we have "01" for each literal occurring positively and "10" for each literal occurring negatively.

Continuing the example, for clause C_2 above, the string of this form that would *not* be included in S' would be

$$00|01|10|00|01||00||0011111111111111\,.$$

Finally, we include strings starting with $2i$ ones and the rest zero, for $i \in \{1,\ldots,m\}$, in both S and S', and refer to these as the control strings.

Observe that the transformation above is clearly polynomial time.

We claim that with this definition of S and S', a permutation with the desired property exists if and only if the formula is satisfiable.

Assume first that the formula is satisfiable, and consider a truth assignment A making it true. Since the "01" and "10" patterns corresponding to different propositional variables are in different positions, we can define a permutation π such that, for each s_i^f, $\pi(s_i^f) = s_i^f$ if $A(p_i)$ is false and $\pi(s_i^f) = s_i^t$ if $A(p_i)$ is true. This automatically implies $\pi(s) = s$ for each control string s. Since, for the truth assignment A, any clause C_j evaluates to true, $\pi(s_j^c)$ is one of the seven strings in S' created from C_j, and we have established a permutation as desired.

For the other direction, we assume that we have a permutation with the desired properties, and must now prove that the formula is satisfiable. First note that the number of ones is even in all the control strings and odd in all other strings. Since permutations preserve the number of ones, and each control string has a different number of ones, we must have that $\pi(s) = s$ for each control string s. From this it follows inductively that, for each $i \in \{1,\ldots,m\}$, π maps positions $\{2(i-1), 2(i-1)+1\}$ into $\{2(i-1), 2(i-1)+1\}$, and thus the string s_i^f must be mapped into s_i^f or s_i^t, since otherwise the number of ones is not preserved.

Thus, we design a truth assignment A from this by letting $A(p_i)$ be false if and only if $\pi(s_i^f) = s_i^f$. Since the only one of the 8 possible patterns created from C_j corresponding to all literals evaluating to false is the only combination not included in S', and since the other 7 options are the only ones where the number of ones in s_j^c is preserved, A must be a satisfying assignment. $\square$

This result does not directly imply that the subsumption problem (given two comparator networks C_a and C_b on n channels, deciding whether $C_a \preceq C_b$) is also NP-complete: although all instances of the subsumption problem are instances of the binary string sets permutation inclusion problem, the converse is not true, as not all sets of binary sequences of length n are sets of outputs of some comparator network. However, it is not known whether the additional structure in the subsumption problem can be used to solve it more efficiently; all known approaches use general methodologies for the more general problem [7,11].

On the other hand, the problem whose instances occur in `Prune` has an extra degree of complexity, as the comparator networks are also existentially quantified. It can thus be stated as follows: does a given set of comparator networks contain two networks such that one of them subsumes the other? In terms of being polynomial-time decidable, this problem is equivalent to the previous one, as it just adds a quadratic outer loop; of course, in practice this additional loop has an important impact on actual execution time.

The reduction from N_k^n to R_k^n involves a sequence of *dependent* subproblems, as each network that is eliminated changes the set given as input to the next subproblem. Often, N_k^n contains chains of subsumptions $C_a \preceq C_b \preceq C_c$; then we can remove C_c using the last subsumption and then remove C_b using the first subsumption, but

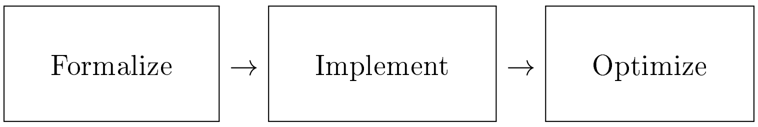

Fig. 2 The steps in the development of our formal proof.

attempting to perform these actions in reverse order fails, since C_b is no longer available – rather, we have to use the fact that $C_a \preceq C_c$ by transitivity (see Section 4.3). We can exploit this dependency in our oracle by choosing an optimal sequence of subproblems/subsumptions that optimizes the global complexity of the algorithm (and not just the oracle invocation), so that the complexity of the `Prune` step becomes linear in $|N_k^n|$ instead of quadratic. The details of this optimization are given in Section 4.3.

At the end of `Prune` we encounter a second type of existential problem when we need to check that we have not found a sorting network yet. Given a comparator network C on n channels, deciding if C is defective (not a sorting network) is NP-complete. This result is reported by Johnson [26], who attributes the result to Rabin.

4 The Formalization

In this section we describe how we can construct a correct-by-construction checker that validates all the results obtained by the untrusted computer program in [10, 11].

The formalization was done in the theorem prover Coq [4]. We assume the reader is familiar with functional programming, and explain the aspects of the syntax that are relevant for understanding our work in the presentation. The complete development is available at `http://www.imada.sdu.dk/~petersk/sn/`.

Our strategy for developing this proof consists of three steps, depicted in Figure 2.

1. We formalize the theory of sorting networks.
2. We implement a naive checker in Coq, using an untrusted oracle, and prove the checker correct.
3. We optimize the checker iteratively in lock-step with adapting the oracle, reproving the correctness of the checker after each change.

In the first step, our focus is on the mathematical theory of sorting networks. In order to separate concerns, we do *not* worry about how the results we prove are used in the actual implementation of the checker, and formalize the theory as close as possible to the mathematical definitions and other formal elements of the algorithm.

Afterwards, we implement a checker that follows the algorithm behind our *ad-hoc* computer program as closely as possible, where possible replacing computationally expensive subproblems by calls to an oracle. Since the use of an oracle is not directly compatible with Coq, the formalized checker depends on a parameter (the oracle). By applying program extraction, we obtain a concrete implementation in Haskell that is connected to a concrete implementation of the oracle. The soundness of the formalized checker is therefore universally quantified over all possible oracles.

Then we perform several optimizations of the checker, of two different types: optimizations of the algorithm and optimizations of the implementation. For the former, we use the fact that all answers needed from the oracle are available beforehand to improve the algorithm's performance. As we show later, this also requires changes to the implementation of the oracle. For the latter, the idea is to use standard computer science techniques to optimize performance – for example by using binary search trees instead of lists in relevant cases. After each change, we reprove the correctness of the optimized checker, benefiting from the modularity of our approach in two ways. First, by formalizing the theory of sorting networks for its own sake (rather than as a library tailored towards proving the correctness of the original algorithm), we ensure that all the results needed in this stage are available, and that they have been stated in general forms. Second, although changes to the algorithm require different inductive proofs, their correctness uses the same arguments as previously developed. Therefore, reproving the soundness of the checker is much simpler and faster than in the previous step. As a result, the total time spent on this third phase was approximately half of that spent in the first two phases.

Our formalization is constructive, but due to our use of rewriting with dependent types, we assume (through imported libraries) the axiom

```
JMeq_eq: ∀ (A:Type) (x y:A), JMeq x y → x=y+
```

(For a discussion of why this axiom is needed, see [32].)

The remainder of this section is divided into three parts, corresponding to the three steps above.

4.1 Formalizing sorting networks

We begin by giving an overview of the formalization of the theory of sorting networks, explaining the main challenges it poses.

Sorting networks. In order to represent comparator networks, we abstract from their physical structure and assume the channels are numbered from top to bottom, starting from 0.[1] A comparator is then simply a pair of numbers, and a comparator network is a list of such pairs.

```
Definition comparator : Set := nat * nat.
Definition CN : Set := list comparator.
```

Here, nat is the Coq type of natural numbers, and Set is the type of sets; so the first definition says that the type of comparators is simply $\mathbb{N} \times \mathbb{N}$, and that the collection of all comparators forms a set.

Not all pairs of natural numbers are valid, however. Their arguments should be distinct, and the number of channels in the network restricts the valid values for a comparator's arguments. Rather than incorporating these constraints into the definition, we characterize valid comparators by means of a predicate. This design decision allows the code implementing the checker to be syntactically closer to the one in [11].

```
Definition comp_channels (n:nat) (c:comparator) : Prop :=
  let (i,j) := c in (i<n) ∧ (j<n) ∧ (i<>j).
```

The term (comp_channels n c), which reads "c is a comparator on n channels", has type Prop, signaling that it is a proposition; in particular, it has no computational content, as we discuss later. A similar predicate channels states that all comparators in a comparator network are on n channels.

[1] Numbering from 0, rather than from 1, simplifies some aspects of the formalization.

As an example, the comparator network in Figure 1 is represented in Coq by the list ((0,1)::(2,3)::(1,3)::(2,4)::(1,4)::(0,2)::(3,4)::(1,2)::(2,3)::nil).

To characterize sorting networks, we take the zero-one principle (Lemma 1) as definition, and let comparator networks act only on binary sequences. Binary sequences are a dependent type (for each natural number n we have a type (bin_seq n)), similar to the standard library type Vector. These could also have been defined as (Vector bool); however, our choice makes proofs simpler, since induction on elements of type (bin_seq n) gives the three relevant cases directly without requiring an extra elimination over their element. This is also reflected in shorter extracted code.

```
Inductive bin_seq : nat → Set :=
  | empty : bin_seq 0
  | zero : ∀ n:nat, bin_seq n → bin_seq (S n)
  | one : ∀ n:nat, bin_seq n → bin_seq (S n).
```

Intuitively, empty is the empty binary sequence (of length 0), (zero n s) is the sequence of length $n+1$ obtained by prepending 0 to s, and likewise for (one n s). By using Coq's mechanisms for adding notation and making arguments implicit, we can write [ ], [0]s and [1]s for these terms, so that we can write e.g. [0][0][1][0][ ] for the sequence 0010.

We define operations get and set over binary sequences such that (get i s) returns the element (0 or 1) in position i of s, and (set i s k) sets position i of s to k, which is either 0 and 1. Setting an index larger than the length of a sequence leaves it unchanged, while attempting to get the value in an index out of range returns 2. These two cases – which do not occur in any context of our formalization – must be contemplated, as Coq requires that all functions be totally defined.

A binary sequence is sorted if its first element is 0 and the remaining sequence is sorted, or if it consists entirely of 1s.

```
Fixpoint all_ones {n:nat} (x:bin_seq n) : Prop := match x with
  | [ ] ⇒ True
  | [ 0 ]s ⇒ False
  | [ 1 ]s ⇒ all_ones s
  end.

Fixpoint sorted {n:nat} (x:bin_seq n) : Prop := match x with
  | [ ] ⇒ True
  | [ 0 ]y ⇒ sorted y
  | [ 1 ]y ⇒ all_ones y
  end.
```

The match constructor performs case analysis on the structure of x. Coq checks that these functions are total by verifying that the recursive calls are on structurally smaller terms.

Sequences propagate through comparator networks as expected.

```
Fixpoint apply (c:comparator) n (s:bin_seq n) :=
  let (i,j):=c in let x:=(get s i) in let y:=(get s j) in
    match (le_lt_dec x y) with
    | left _ ⇒ s
    | right _ ⇒ set (set s j x) i y
    end.

Fixpoint full_apply (S:CN) n (s:bin_seq n) := match S with
  | nil ⇒ s
  | cons c S' ⇒ full_apply S' _ (apply c s)
  end.
```

One comment on the first definition, as we use this code structure repeatedly. The term (`le_lt_dec x y`) has type {x ≤ y}+{y < x}, which is an example of a type of *decidable propositions.* Terms of the form {A}+{B}, where `A` and `B` have type `Prop`, are built by providing a proof that one of `A` or `B` holds, and we can compute which one – so such proofs include an algorithm that determines which of `A` or `B` is the case.[2] In our example, (`le_lt_dec x y`) states that we can decide whether `x` ≤ `y` or `y` < `x`; `left` and `right` are the constructs for showing that `A` (respectively, `B`) holds. So, `apply (i,j) n s` compares the values at positions `i` and `j` of `s` and swaps them if necessary, exactly as described informally in Section 3.

A sorting network is then a comparator network that sorts all binary inputs.

```
Definition sorting_network (n:nat) (S:CN) :=
  (channels n S) ∧ ∀ s:bin_seq n, sorted (full_apply S s).
```

We also define an alternative characterization of sorting networks in terms of their sets of outputs and prove its equivalence to this one.

```
Definition outputs (C:CN) (n:nat) : list (bin_seq n) :=
  (map (full_apply C n) (all_bin_seqs n)).

Theorem SNW_char : ∀ C n, channels n C → (∀ s, In s (outputs C n) → sorted s) →
                         sorting_network n C.
```

Here, the function `all_bin_seqs` generates all binary sequences of a given length.

This formalization was developed focusing on the theory at hand. This means that we included several relevant properties of the types we defined, which we omit here for the sake of brevity. These include relations between the operations on binary sequences (e.g. `get (set s i j) i = j` as long as `i` and `j` are in the correct ranges) or trivial properties of sorting networks (e.g. they do not change sorted inputs, or they preserve the number of 0s throughout execution).

We also define a Coq tactic to (try to) prove automatically that a comparator network `C` is a sorting network on n channels by applying `SNW_char` and checking that all binary sequences of length n are sorted by `C`. However, as n increases this tactic becomes too expensive, since it generates all 2^n subcases recursively and solves them afterwards. As a consequence, trying to prove that a 9-channel network is a sorting network causes the system to run out of memory. This also confirms our intuition that we would benefit from an approach based on a formalized checker, rather than a complete formal proof. Indeed, we can actually prove (in Coq) that the property of being a sorting network is decidable, which is instrumental in defining the checker. The notation `~A` denotes the negation of `A`.

```
Lemma SNW_dec : ∀ n C, channels n C → {sorting_network n C} + {~sorting_network n C}.
```

The subsumption lemma. The key result for proving the soundness of the algorithm in [11] is the subsumption lemma (Lemma 2, page 6), which we restate here.

Lemma 2. Let C_a and C_b be comparator networks of the same size on n channels such that $C_a \preceq C_b$. If there exists a sorting network $C_b; C$ of size k, then there exists a sorting network $C_a; C'$ of size k.

[2] This intuition is made precise by the program extraction mechanism of Coq [31], which generates precisely this algorithm.

Several variants of this result occur in the literature, and the proofs are all similar: if $C_a \preceq_\pi C_b$ and $C_b; C$ is a sorting network, then $C_a; \mathsf{std}(\pi(C))$ is also a sorting network, where $\pi(C)$ is obtained by renaming the channels according to π (assuming they have been numbered as in our presentation) and std is the *standardization procedure* described by Knuth (Exercise 5.3.4.16 in [27]). Each proof then proceeds to show that the network thus constructed satisfies the additional properties needed for the problem at hand.

From a mathematical perspective, instead of directly formalizing Lemma 2, it makes more sense to study the operations involved in its proof – applying a permutation to a comparator network and standardizing it. We first show how we formalize permutations, and then discuss the formalization of std and the proof of the subsumption lemma.

There are several common alternatives to representing permutations in Coq. The standard library includes an inductive type stating that two lists are permutations of each other; but manipulating it is cumbersome. Furthermore, we only use permutations to rename channels in comparator networks, so we want a definition that makes it easy and efficient to apply permutations to objects.

For this reason, we chose to represent permutations as finite functions. A permutation P is a list of pairs of natural numbers, with the intended meaning that $(i, j) \in P$ corresponds to P mapping i into j. We assume that P does not change i if there is no pair $(i, j) \in P$ – this makes it much simpler to represent transpositions, which are the only permutations we need to represent explicitly in the formalization. We focus on permutations of the numbers $0, \ldots, n-1$, which we refer to as "permutations of $[n]$". In order for P to be a valid permutation of $[n]$, several conditions have to hold:

1. all pairs $(i, j) \in P$ must satisfy $i < n$ and $j < n$;
2. no number may occur twice either as the first or as the second element of distinct pairs in P; and
3. the sets of numbers occuring as first or second elements of the pairs in P must coincide.

As before, we separate the syntactic datatype of permutations from the semantic property of being a permutation. Here, `NoDup` is the Coq standard library predicate stating that a list does not have duplicate elements, and `all_lt(n,l)` is an inductive predicate stating that all elements of `l` are smaller than `n`.

```
Definition permut := list (nat*nat).

Definition dom (P:permut) := map (fst (A:=nat) (B:=nat)) P.
Definition cod (P:permut) := map (snd (A:=nat) (B:=nat)) P.

Definition permutation n (P:permut) :=
  NoDup (dom P) ∧ all_lt n (dom P) ∧ (∀ i, In i (dom P) ↔ In i (cod P)).
```

As a sanity-check, we prove the relationship with the permutations in the Coq standard library.

```
Lemma permutation_Permutation : ∀ (n:nat) (P:permut),
  permutation n P → Permutation (dom P) (cod P).
```

All properties of permutations are added to the `core` hint database, so that Coq can automatically prove most properties of permutations required during the formalization.

We provide mechanisms to define permutations in three different ways, corresponding to the usage of permutations in proofs.

1. The identity permutation is simply the empty list, which is a permutation of $[n]$ for any n.
2. Given a permutation P, we construct its inverse (inverse_perm P) by reversing all pairs in P. If P is a permutation of $[n]$, then so is (inverse_perm P).
3. The transposition i $\longleftrightarrow$ j is the permutation that switches i and j, leaving all other values unchanged. This transposition is defined as the list $\{(i,j),(j,i)\}$, and it is a permutation if $i \neq j$ (otherwise the list $\{i,j\}$ contains duplicate elements). This side condition appears in some results about transpositions, but it is never a problem since we only us transpositions with indices i and j originating from a comparator (i,j).

Recall that a comparator (i,j) is said to be *standard* if $i < j$. We now enrich our theory of comparator networks with a notion of standard comparator.

```
Definition comp_std (n:nat) (c:comparator) := let (i,j) := c in (i<n) ∧ (j<n) ∧ (i<j).
```

This definition includes some redundancy, as $i < j$ and $j < n$ imply that $i < n$, but it is technically useful to keep a similar structure to the definition of comp_channels. As before, we extend this predicate to lists, so that (standard n C) holds if C is a standard comparator network on n channels.

The definition of standardization in [27] is as follows: given a comparator network C, pick the first comparator (i,j) for which $i > j$, replace it with (j,i) and exchange i with j in all subsequent comparators, then iterate the process until a fixpoint is reached. To formalize this operation in Coq, we need to use well-founded recursion as follows.

```
Function std (S:CN) {measure length S} : CN := match S with
  | nil ⇒ nil
  | cons c S'  ⇒ let (x,y) := c in match (le_lt_dec x y) with
         | left _ ⇒ ((x,y) ::  std S')
         | right _ ⇒ ((y,x) ::  std (permute x y S'))
end end.
```

This function is not structurally decreasing, as its recursive call does not take S' as an argument. Therefore, we annotate the definition with {measure length S}, which tells the proof assistant that all recursive calls are made over arguments of smaller length; this leaves a proof obligation (that (permute x y S') has length smaller than cons c S'), which is shown using results over permutations (namely, that they preserve the length of comparator networks they are applied to). This guarantees that std S always terminates, and therefore std is a total function.

Then we prove that standardizing a comparator network on n channels yields a standard comparator network on n channels. Other simple properties include, e.g., that standardization preserves the size of a network, that it is an idempotent operation, and that it does not change standard comparator networks. The key result is that it also preserves sorting networks.

```
Theorem std_sort : sorting_network n C → sorting_network n (std C).
```

The proof of this result is non-trivial. Informally, the argument given in [27] seems simple: the function computed by $\mathsf{std}(C)$ is the composition of some permutation with the function computed by C, and since $\mathsf{std}(C)$ is standard, it preserves sorted sequences, which implies that the given permutation must be the identity. Formalizing this result, however, requires computing the concrete permutation explicitly and identifying several implicit facts about permutations that were never mentioned explicitly. The formal proof proceeds in three steps. First, we characterize the relation between the function computed by a comparator network before and after applying a permutation.

```
Lemma full_apply_permute : ∀ n x y, x <> y → x < n → y < n → ∀ C, channels n C →
                                ∀ s:bin_seq n, let T := x⟷ y in
                               (permute x y C)[s] = apply_perm T (C[apply_perm T s]).
```

Here, `C[s]` stands for (`full_apply C s`), the result of applying a comparator network `C` to a binary sequence `s`. This is a particular case of a more general result, which we state in mathematical notation.

Lemma 3 *If C is a comparator network on n channels, π is a permutation of $0 \ldots n-1$ and s is a binary sequence of length n, then $\pi(C)(s) = \pi^{-1}(C(\pi(s)))$.*

To the best of our knowledge, this identity was never stated previously; instead, some of the published proofs of variants of Lemma 2 (namely [7]) apparently use the incorrect equality $\pi(C)(s) = C(\pi(s))$ – which fortunately does not compromise the validity of the result. Using this lemma, we can now prove that the sets of inputs that are mapped to the same output are the same for C as for $\mathsf{std}(C)$.

```
Lemma standardization_char : ∀ n C, channels n C → ∀ s s':bin_seq n,
                        C[s] = C[s'] → (standardize C)[s] = (standardize C)[s'].
```

The final step requires considering the case where `s'` is the sorted version of `s` and using the fact that standard networks do not change sorted sequences.

```
Lemma standardization_sort_lemma : ∀ C n, channels n C →
      (∀  s:bin_seq n, sorted C[s]) → ∀ s:bin_seq n, sorted (standardize C)[s].
```

It is then simple to prove `std_sort` from this last result. A consequence of this lemma is that we can work only with standard comparator networks, which drastically reduces the size of the search space.

It turned out to be easier to formalize a completely different proof of Lemma 2 than the one given in [7], which we now summarize. For legibility, we present it in mathematical notation.

Proof (Lemma 2) Let C_a and C_b be comparator networks of the same size on n channels such that $C_a \preceq C_b$, and assume that $C_b; C$ is a sorting network of size k. We show that $\mathsf{std}(C_a; \pi(C'))$ is a sorting network of the same size as $C_b; C'$.

Given a binary sequence s, we write $\mathsf{sort}(s)$ for the sequence obtained by sorting s, and we begin by showing that $C'(\pi(C_a(s))) = C'(\pi(C_a(\mathsf{sort}(s))))$. By hypothesis, $C'(\pi(C_a(s))) = C'(C_b(x)) = (C_b; C')(x) = \mathsf{sort}(x)$ for some x; likewise, $C'(\pi(C_a(\mathsf{sort}(s)))) = \mathsf{sort}(y)$ for some y, and $\mathsf{sort}(x) = \mathsf{sort}(y)$ since x and y have the same number of 0s (because they have the same number of 0s as s and $\mathsf{sort}(s)$, respectively).

Since permutations are injective, $\pi^{-1}(C'(\pi(C_a(s)))) = \pi^{-1}(C'(\pi(C_a(\mathsf{sort}(s)))))$. Rewriting using Lemma 3 yields $\pi(C')(C_a(s)) = \pi(C')(C_a(\mathsf{sort}(s)))$, so we conclude that $(C_a; \pi(C'))(s) = (C_a; \pi(C'))(\mathsf{sort}(s))$. However, if $(C_a; \pi(C'))$ maps two inputs to the same output, then so does $\mathsf{std}(C_a; \pi(C'))$, so we also have that $\mathsf{std}(C_a; \pi(C'))(s) = \mathsf{std}(C_a; \pi(C'))(\mathsf{sort}(s))$. Since $\mathsf{std}(C_a; \pi(C'))$ is standard, it does not affect sorted sequences, and therefore $\mathsf{std}(C_a; \pi(C'))(\mathsf{sort}(s)) = \mathsf{sort}(s)$, so $\mathsf{std}(C_a; \pi(C'))(s) = \mathsf{sort}(s)$.

Thus, $\mathsf{std}(C_a; \pi(C'))$ is a sorting network. □

We summarize the definition of subsumption and the formal statement of Lemma 2. The third result we include shows that, given two comparator networks C_a, C_b and a permutation π, we can decide whether $C_a \preceq_\pi C_b$. The type `permut` is the type of permutations.

```
Definition subsumption (n:nat) (C C':CN) (P:permut) (HP:permutation n P) :=
  ∀ s:bin_seq n, In s (outputs C n) → In (apply_perm P s) (outputs C' n).

Theorem SNW_subsumption : ∀ n P HP C C' N,
  standard n C → subsumption n C C' P HP → sorting_network n (C'++ N) →
  sorting_network n (standardize (C ++ apply_perm_to_net P N)).

Lemma subsumption_dec : ∀ n C C' P HP,
                        {subsumption n C C' P HP} + {~subsumption n C C' P HP}.
```

The details of these last proofs illustrate an important point: the development of this formalization is an intrinsically mathematical task, requiring the same mathematical capabilities as developing any other formalization from both the developer and the theorem prover. By constrast, the subsequent development of a checker on top of this formalization is much more of a computer science task.

Completeness. Soundness of the generate-and-prune algorithm relies on the notion of a *complete set of filters*, introduced in [11]. Intuitively, a set of comparator networks is complete if one of its elements can be extended to an optimal sorting network (the term *filter* is often used to denote a comparator network that is a prefix of another network). A close inspection of the soundness proof reveals that the definition given in [11] was incomplete, as it implicitly used additional properties of these sets. The complete (formalized) definition of this notion reads as follows.

```
Definition size_complete (R:list CN) (n:nat) := ∀ k:nat,
   (∃ C:CN, sorting_network n C ∧ length C = k) →
    ∃ C' C'':CN, In C' R ∧ standard n (C'++ C'')
         ∧ (∀ C1 c C2, (C'++ C'') = (C1++ c::C2) → ~redundant n C1 c)
         ∧ sorting_network n (C'++ C'') ∧ length (C'++ C'') ≤ k.
```

In mathematical notation, in order for a set of comparator networks R to be a complete set of filters (as specified by `size_complete`), it must satisfy the following property: if there exists a sorting network of size k, then R must contain an element C' such that C' can be extended to a standard sorting network of size at most k. (In particular, if k is the size of an optimal sorting network, then this size must be exactly k.) The property that this sorting network is standard is crucial for the soundness of generate-and-prune, as its implementation only constructs standard networks (as we show shortly).

The other constraint states that the sorting network cannot have *redundant* comparators. A comparator (i,j) in comparator network $C_a;(i,j);C_b$ is redundant if $\mathbf{x}_i < \mathbf{x}_j$ for all $\mathbf{x} \in \mathsf{outputs}(C_a)$ – in other words, (i,j) never changes its inputs. This notion is a simplification of that proposed in Exercise 5.3.4.51 of [27] (credited to R.L. Graham). The condition is again necessary because generate-and-prune does not build networks with redundant comparators, using the fact that these cannot occur in optimal sorting networks.

We proceed to formalize redundancy. We establish a number of properties of redundant comparators, including that they can always be removed from a sorting network, and show that it can be decided whether a comparator is redundant. We also introduce the more general notion of redundancy with respect to a set of inputs, useful for defining the operation of removing all redundant comparators from a network.

```
Definition redundant (n:nat) (C:CN) (c:comparator) :=
       ∀ s:bin_seq n, (apply c (C[s])) = C[s].

Definition red_wrt (n:nat) (I:list (bin_seq n)) (C:CN) (c:comparator) :=
       ∀ s, In s I → (apply c (C[s])) = C[s].

Lemma red_wrt_dec : ∀ n I C c, {red_wrt n I C c} + {~red_wrt n I C c}.
Lemma redundant_dec : ∀ n C c, {redundant n C c} + {~redundant n C c}.

Fixpoint rem_red (n:nat) (C:CN) (I:list (bin_seq n)) := match C with
  | nil ⇒ nil
  | c :: C' ⇒ match (red_wrt_dec _ I nil c) with
             | left _ ⇒ rem_red n C' I
             | right _ ⇒ c :: rem_red n C' (map (apply c (n:=n)) I)
  end end.

Definition remove_red (n:nat) (C:CN) := rem_red n C (all_bin_seqs n).

Lemma rem_red_SNW : ∀ n C, sorting_network n C → sorting_network n (remove_red n C).
```

Regarding completeness, we prove that the set $\{[]\}$ is complete, and that if there is a complete set of filters R whose elements all have size k, then all sorting networks on n channels have size at least k. This key property does not hold for the informal definition of size completeness in [11].

```
Lemma empty_complete : ∀ n, size_complete (nil::nil) n.

Lemma complete_size : ∀ R n k, size_complete R n → (∀ C, In C R → length C = k) →
                        ∀ S, sorting_network n S → length S ≥ k.
```

The above formalization of the theory of sorting networks closely follows the mathematical definitions, lemmas and theorems described in e.g. [11,27,37]. There is, however, an interesting difference: instead of stating and proving results of the form $\forall N.\varphi(N) \to \exists N'.\psi(N')$, we define particular operations T (e.g. std), and prove that $\forall N.\varphi(N) \to \psi(T(N))$, from which we can straightforwardly prove the original statement. The existence of these operations follows directly from the fact that the theory of sorting networks is constructive. However, the fact that the proofs of these statements given in [7,11,27,37] all proceed by explicitly constructing the witness allows us to formalize this theory as a mathematical theory of operators over sorting networks and their properties, rather than a theory of existential statements about sorting networks.

4.2 Implementing a naive checker

We now proceed to the second stage of the formalization: implementing the generate-and-prune algorithm in Coq, and proving its soundness. By applying the program extraction mechanism, we obtain a Haskell program that executes that algorithm and satisfies the soundness properties specified in the formalization by virtue of the soundness of program extraction [31].

Preliminaries. Before presenting the checker, we explain the idea behind program extraction and illustrate with some of the results presented in the previous section. In the Coq framework, the Curry–Howard correspondence states that we can view terms and types in the Coq type system from two different perspectives. From a functional programming point of view, it is a programming language where types correspond to the typical function types and terms to function definitions; from a logical point of view, types correspond to propositions and terms to proofs of those propositions. The Coq framework favors a mixture of both visions, which we have implicitly adhered to in the previous section, by distinguishing (higher-level) types `Set` for "computationally meaningful types" and `Prop` for "logical types". (There is a more general family of computationally meaningful types, but we do not use those in this presentation.) Thus, in our formalization, an object of type `CN` is a computational object (an instance of a datatype), whereas an object of type (`channels n C`) is not (it is a proof that a particular object satisfies some property).

The mechanism of program extraction [31] takes this correspondence one step further, by defining formal mappings from Coq to functional programming languages that map computational types into functional programs, "forgetting" all logical statements. In particular, we are interested in the translation to Haskell. The structure of the mapping is such that every Haskell program is guaranteed to satisfy any additional properties of the original Coq term it originated from.

As an example, the Coq type of binary sequences presented above is extracted to the following Haskell datatype. Observe that this is no longer a dependent type.

```
data Bin_seq =
   Empty
 | Zero Int Bin_seq
 | One Int Bin_seq
```

Likewise, the types of comparator networks and all functions defined on those are extracted to Haskell programs with the same behaviour. In particular, we show the code for deciding whether a comparator network is a sorting network.

```
sNW_dec :: Int → Cn → Bool
sNW_dec n c =
  sumbool_rec (\_ → True) (\_ → False) (all_sorted_dec n (outputs c n))

sumbool_rec :: (() → a1) → (() → a1) → Bool → a1
sumbool_rec f f0 s =
  case s of {
   True → f __;
   False → f0 __}
```

We explain this code. The operator `sumbool_rec` performs case analysis on the type of decidable statements, which is isomorphic to the type of Boolean values. If the third argument evaluates to `True`, `sumbool_rec` returns its first argument, otherwise it returns its second argument.[3] The result of (`sNW_dec n c`) is therefore essentially that of (`all_sorted_dec n (outputs c n)`). The function `all_sorted_dec` is obtained from another decidability predicate (given a list of binary sequences, we can decide whether all of its elements are sorted), and `outputs` computes the set of outputs of the network.

The important part is that this code does *not* include the actual proof that `c` is (not) a sorting network when the result is `True` (`False`), since that part of the original term is non-computational. However, soundness of the extraction mechanism guarantees that (`sNW_dec n c`) evaluates to `True` precisely when `c` is a sorting network – since the extracted code for (`sNW_dec n c`) performs precisely this test.

In the current stage, we are interested in writing a checker in Coq, and that means we essentially write functional code that is extracted directly to very similar-looking

[3] The indirect way in which this is done is unfortunately typical of program extraction, and adds some layers of complexity to what a direct program would achieve.

Haskell functions, while the soundness results are not extracted at all. However, we use case analysis on the decidability results we proved earlier, so all the algorithms implicit in those proofs impact the behaviour of our program. It is interesting to note that the direct proofs implement efficient algorithms in many cases (as `sNW_dec` stated above: it has been shown [8] that any algorithm to decide whether a comparator network is a sorting network by testing binary inputs must check all non-sorted inputs). The notable exception is the decidability of subsumption, which we discuss in the section on optimizations.

The generation step. The formalization of the generation step proceeds in two phases. First, we define the simple function appending a comparator to the end of a comparator network in all possible ways, and `Generate` simply maps it into a set. The function `all_st_comps` produces a list of all standard comparators on n channels. We prove that `Generate` maps complete sets into complete sets, as long as the input does not contain a sorting network. (If the latter is the case, then all elements in the generated set contains redundant comparators, which prevents the set from being complete.)

```
Definition add_to_all (cc:list comparator) (C:CN) :=
  map (fun c ⇒ (C ++ (c :: nil))) cc.

Fixpoint Generate (R:list CN) (n:nat) := match R with
  | nil ⇒ nil
  | cons C R' ⇒ (add_to_all (all_st_comps n) C) ++ Generate R' n
  end.

Theorem Generate_complete : ∀ R n, size_complete R n →
  (∀ C, In C R → ~sorting_network n C) → size_complete (Generate R n) n.
```

In our formalization, we do this as an additional step, *removing* all networks whose last comparator is redundant from the set. This is done using a specialized version of the notion of redundancy.

```
Definition last_red (n:nat) (C:CN) := ∃ C' c, redundant n C' c ∧ C = (C' ++ c :: nil).

Lemma last_red_dec : ∀ n C, {last_red n C} + {~last_red n C}.

Fixpoint filter_nred (n:nat) (R:list CN) := match R with
  | nil ⇒ nil
  | (C :: R') ⇒ match last_red_dec n C with
                | left _ ⇒ filter_nred n R'
                | right _ ⇒ C :: filter_nred n R'
end end.

Definition OGenerate (R:list CN) (n:nat) := filter_nred n (Generate R n).

Theorem OGenerate_complete : ∀ R n, size_complete R n →
  (∀ C, In C R → ~sorting_network n C) → size_complete (OGenerate R n) n.
```

The extracted code for these functions essentially coincides with their Coq definition, since they use no proof terms, and matches the pseudo-code in [11].

With regards to the runtime complexity, `Generate` is $O(|\mathtt{R}|)$, i.e., linear in the size of the set of networks it is applied to. However, the constant hidden in the asymptotic notation is fairly large, as each network in `R` gives rise to $\binom{n}{2}$ new networks, and for each of these networks, a redundancy test is made for the latest added comparator. This is as hard as deciding the NP-complete problem of whether a network is defective without the latest added comparator. Since we use an exhaustive approach of up to 2^n tests of all inputs, even for our small $n = 9$, this becomes a large constant.

The pruning step. For the pruning step, we need to work with the untrusted oracle. We define an oracle to be a list of subsumption triples `(C,C',pi)`, with the intended meaning that `C` subsumes `C'` via the permutation `pi`.[4] For efficiency reasons, the log files record permutations as their output on the set $[n]$, so for example the transposition over $[4]$ exchanging 0 and 2 would be represented as $2, 1, 0, 3$. Since we do not trust the oracle, we need to test that these lists represent valid permutations. We show that the property of a list of natural numbers corresponding to a permutation on $[n]$ is decidable, and define a function `make_perm` to translate lists of natural numbers into (syntactic) permutations, and show that the resulting object satisfies `permutation` if the original list corresponds to a permutation.

```
Variable n:nat.
Variable l:list nat.

Definition pre_perm := NoDup l ∧ all_lt n l ∧ length l = n.

Lemma pre_perm_dec : {pre_perm} + {~pre_perm}.

Lemma pre_perm_lemma : pre_perm → permutation n (make_perm l).
```

Thus, our checker is able to get a list of natural numbers from the oracle, test whether it corresponds to a permutation, and in the affirmative case use this information.

One might wonder why we did not represent permutations uniformly throughout the whole formalization. The reason for not doing so is that we have two distinct objectives in mind. While formalizing results, we are working with an unknown number n of channels, and it is much simpler to represent permutations by only explicitly mentioning the values that are changed, as this allows for uniform representations of transpositions and the identity permutation. Also, computing the inverse of a permutation is very simple with the finite function representation, but not from the compact list representation given by the oracle. When running the extracted checker, however, we are concerned with efficiency. The oracle provides information on millions of subsumptions, so it is of the utmost importance to minimize its size.

Using the oracle, we define the pruning step as follows.

```
Function Prune (O:Oracle) (R:list CN) (n:nat) {measure length R} : list CN :=
  match O with
  | nil ⇒ R
  | cons (C,C',pi) O'  ⇒ match (CN_eq_dec C C') with
      | left _ ⇒ R
      | right _ ⇒ match (In_dec CN_eq_dec C R) with
          | right _ ⇒ R
          | left _ ⇒ match (pre_perm_dec n pi) with
              | right _ ⇒ R
              | left A ⇒ match (subsumption_dec n C C' pi' Hpi) with
                  | right _ ⇒ R
                  | left _ ⇒ Prune O' (remove CN_eq_dec C' R) n
end end end end end.
```

[4] Technically, we are formalizing just the *oracle data*: the result of the oracle (a computer program), rather than the program itself (which we do not want to formalize). We deliberately blur the distinction between these concepts in this section.

We comment on the terms used in the successive `match` statements. In (`CN_eq_dec C C'`) we check that `C` and `C'` are not the same network. In (`In_dec CN_eq_dec C R`) we check that `C` occurs in the list `R` (where `CN_eq_dec` is used for checking equality between `C` and the members of `R`). In (`pre_perm_dec n pi`) we verify that `pi` represents a valid permutation. Finally, in (`subsumption_dec n C C'pi' Hpi`) we verify that the required subsumption actually holds. For legibility, we write `pi'` for the actual permutation generated by `pi` and `Hpi` for the proof term stating that this is indeed a permutation.

All these tests have to be passed in order for the subsumption to be processed. The soundness of the pruning step uses all these facts, together with the subsumption lemma, and further requires (i) that the input set `R` contain only standard networks; (ii) that `R` contain no networks with redundant comparators; and (iii) that all networks in `R` have the same size.

```
Theorem Prune_complete : ∀ O R n, size_complete R n →
  (∀ C, In C R → standard n C) →
  (∀ C C' c C'', In C R → C = C'++ c::C'' → ~redundant n C' c) →
  (∀ C C', In C R → In C' R → length C = length C') →
  size_complete (Prune O R n) n.
```

Note the universal quantification on `O`: this result holds *regardless* of whether the oracle provides correct information or not – hence the "untrusted" qualification of the oracle in our approach. This implementation is simpler than the pseudo-code in [11], as the oracle allows us to bypass all search steps – both for permutations and for possible subsumptions.

With regards to the runtime complexity, `Prune` is $O(|\texttt{O}| \cdot |\texttt{R}|)$, since it implements a tail-recursive traversal of the oracle `O`, where each element is potentially located in the set of networks `R` via a linear scan. Potentially, at the cost of a more complicated correctness proof, the domain of all networks could be equipped with a (natural) total ordering, `R` could be organized as an efficient search structure, such as a balanced binary search tree, for instance, and the time complexity could be reduced to $O(|\texttt{O}| \cdot \log|\texttt{R}|)$. However, as explained earlier, at this stage we are interested in formalizing a simple checker, which we later optimize.

Linking everything together. We now define the iterative generate-and-prune algorithm and prove its correctness. Here we deviate somewhat from the algorithm presented in [11], as we have to provide the oracle; we also specify the number of iterations. Our implementation receives two natural numbers as input (the number of channels `n` and the number of iterations `m`) and returns one of three possible answers: (`yes n k`), meaning that a sorting network of size `k` was found and that no sorting network of smaller size exists; (`no n m R H1 H2 H3`), meaning that `R` is a set of standard (`H3`) comparator networks of size `m` (`H2`), with no duplicates (`H1`); or `maybe`, meaning that an error occurred. The proof terms in `no` are necessary for the correctness proof, but they are all removed in the extracted checker. Since they make the code complex to read, we abbreviate them to □.

```
Inductive Answer : Set :=
  | yes : nat → nat → Answer
  | no : ∀ n k:nat, ∀ R:list CN, NoDup R → (∀ C, In C R → length C = k) →
                                           (∀ C, In C R → standard n C) → Answer
  | maybe : Answer.

Fixpoint Generate_and_Prune (n m:nat) (O:list Oracle) := match m with
  | 0 ⇒ match n with
```

```
            | 0 ⇒ yes 0 0
            | 1 ⇒ yes 1 0
            | _ ⇒ no n 0 (nil :: nil) □□□
            end
  | S k ⇒ match O with
       | nil ⇒ maybe
       | X:: O' ⇒ let GP := (Generate_and_Prune n k O') in
          match GP with
          | maybe ⇒ maybe
          | yes p q ⇒ yes p q
          | no p q R □□□⇒ let GP' := Prune X (OGenerate R p) p in
                            match (Exists_SNW_dec p GP' _) with
                                   | left _ ⇒ yes p (S q)
                                   | right _ ⇒ no p (S q) GP' □□□
end end end end.
```

In the case of a positive answer, the network constructed in the original proof is guaranteed to be a sorting network; therefore we do not need to return it. Note the elimination over `Exists_SNW_dec`, which states that we can decide whether a set contains a sorting network.

```
Lemma Exists_SNW_dec : ∀ n l, (∀ C, In C l → channels n C) →
  {∃ C, In C l ∧ sorting_network n C} + {∀ C, In C l → ~sorting_network n C}.
```

The correctness of the answer is shown in the two main theorems, covering the cases where the answer is (`yes n k`) and (`no n m R □□□`). Again, these results quantify universally over `O`, thus holding regardless of whether the oracle gives right or wrong information.

```
Theorem GP_yes : ∀ n m O k, Generate_and_Prune n m O = yes n k →
                (∀ C, sorting_network n C → length C ≥ k) ∧
                 ∃ C, sorting_network n C ∧ length C = k.

Theorem GP_no : ∀ n m O R HR0 HR1 HR2,
                Generate_and_Prune n m O = no n m R HR0 HR1 HR2 →
                ∀ C, sorting_network n C → length C > m.
```

The next step is extracting `Generate_and_Prune` (and everything it depends on) to Haskell, using the extraction mechanism. The result is a file `Checker.hs` containing among others a Haskell function

```
generate_and_Prune :: Nat → Nat → (List Oracle) → Answer
```

In order to run this extracted certified checker, we have written an interface that calls function `generate_and_Prune` with the number of channels, the maximum size of the networks, and the list of the oracle information, and then prints the answer. For convenience, the oracle information is stored in a compact human-readable file format, with the networks and permutations essentially being stored as sequences of whitespace-separated integers. The interface not only parses this format, but also includes conversion functions from Haskell integers to the extracted naturals and a function implementing the oracle, as well as a definition of `Checker.Answer` as an instance of the type class `Show` for printing the result. The main function of the Haskell front-end reads as follows.

```
main = do
  (n:k:_) ← getArgs
  os ← mapM file_to_oracle (file_names (read n) (read k))
  let nn = int_to_nat (read n)
```

```
      kk = int_to_nat (read k) in
        print Checker.generate_and_Prune nn kk (list_to_List os)
```

It is important to recall that we do *not* need to worry about soundness of almost any function defined in the interface, as the oracle is untrusted. For example, a wrong conversion from natural numbers to their Peano representation does not impact the correctness of the execution (although it definitely impacts the execution *time*, as all subsumptions become invalid). We only need to worry about the function printing the result, but this is straightforward to verify.

This extracted checker was able to validate the proofs of optimal size up to and including $n = 8$ in around one day – roughly the same time it took to produce the original proof, albeit without search. This already required processing more than 300 MB of proof witnesses for the roughly 1.6 million subsumptions. To the best of our knowledge, it constitutes the first formal proof of the results in [20]. These experiments also suggested that this extracted checker would take around 20 years to verify the case of $n = 9$.

4.3 Optimizing the extracted checker

In order to be able to address the case of 9 inputs in a realistic time frame, the checker described above needs to be optimized in various ways. Rather than presenting this optimization process in detail, we focus on three aspects that illustrate the gains we obtain.

The three optimizations we consider are: changing data structures to reduce memory footprint; improving algorithms underlying decidability proofs; and reimplementing `Prune` in a more efficient way. In all three cases, we show that we can motivate, perform, and evaluate the changes described from a purely local perspective, i.e., without regard for the context in which they are used. This illustrates the modularity of our development.

Changing data structures. One of the main limitations of running the extracted checker is its memory consumption. In particular, profiling reveals that half of the memory is being used just to represent natural numbers, due to the unary representation of these in Coq. There are several workarounds to overcome this limitation; since our formalization essentially uses natural numbers as labels, we opted for extracting them to native Haskell integers (following a suggestion by Pierre Letouzey) rather than optimizing their Coq representation. In principle, this could break soundness of extraction, but the fact that the only operations we use are successor and predecessor on the closed set $\{1, \ldots, 9\}$ makes this a non-issue. Given that in general Haskell objects are stored on the heap, the fact that Haskell's memory management avoids duplication of small integers by storing them only once on the heap, which has dramatic benefits for memory consumption. In particular, this means that explicitly defining the set $\{1, \ldots, 9\}$ as a set with nine constants and extracting it would increase the memory requirements.

Additionally, we can halve the memory consumption by representing comparators in a more efficient way: the set of all comparators is countable and, thus, a bijection to $\mathbb{N}$ can be used to represent them as a single natural number. This has the added effect of also removing the memory overhead of the constructor for pairs. As luck would have it, the bijection $\varphi(i, j) = \frac{1}{2}j \times (j - 1) + i$ happens to map very nicely to the function `all_st_comps` described earlier, since the comparator (i, j) is exactly the

$\varphi(i,j)$-th element of `all_st_comps n` (as long as $i,j < n$). We can then compute φ^{-1} by means of a look-up in this list, and Haskell's caching effects ensure that there is no measurable performance loss.

In practice, this requires no changes to the formalization of the theory, and only minor ones to `Generate` and `Prune`. We define a type `OCN` (simply as `list nat`) and a mapping `OCN_to_CN : nat → OCN → CN`, guaranteed to generate a standard comparator network when all its arguments are in the right ranges, and we prove that this mapping is a bijection under that hypothesis. We then define a complete set of optimized comparator networks by replacing `R` with `(map (OCN_to_CN n) R)` in the only place where it occurs in the original definition.

`Generate` now works by adding all natural numbers between the 0 and $\frac{n(n-1)}{2}$, where n is the number of channels, to all the optimized comparator networks.

```
Fixpoint Generate (R:list OCN) (n:nat) : list OCN :=
  match R with
  | nil ⇒ nil
  | cons C R' ⇒ (add_to_all (till_n' (length (all_st_comps n))) C) ++ Generate R' n
  end.
```

Reproving the key lemma `Generate_complete` is then simply a matter of unfolding the definition of `OCN_to_CN` and adding a few rewriting steps in the previous proofs, and this can be done almost mechanically. Adapting `OGenerate` is even simpler: in the definition of `filter_nred`, we replace the subterm `(last_red_dec n C)` with the term `(last_red_dec n (OCN_to_CN n C))`, and all previous proofs still go through.

Finally, the definition of `Prune` needs to use decidability results on equality over `OCN` (rather than over `CN`), but again the changes are completely straightforward.

We point out that there were no changes to the *original* formalization of the theory of sorting networks, but only to the proofs of soundness of `Generate_and_Prune` – and these were very localized, not affecting the top-level structure of the proofs.

Reproving decidability results. As we mentioned at the beginning of the previous section, our checker often relies on decidability results in the original formalization, which get extracted to Haskell functions evaluating the computational part of those results. We focus on one such result, which yields a function that is straightforward to optimize using simple computer science techniques: the subsumption check. Recall that `subsumption_dec` states that subsumption is decidable. This Coq term gets extracted to Haskell as

```
subsumption_dec :: Int → Cn → Cn → Permut → Bool
subsumption_dec n c c' p =
  all_in_dec (bin_seq_eq_dec n) (apply_perm p n) (outputs c n) (outputs c' n)
```

where `(outputs c n)` computes the list of all outputs of `c`, `(apply_perm p n)` applies the permutation `p` to this list, and `all_in_dec` checks that all of its elements are in `(outputs c' n)`.

The lists of outputs contain 2^n elements, as they are built simply by computing the output of the network for each input and storing it, without sorting or removing elements. The naive approach of checking for each element in one set containment in the other is quadratic in 2^n, so the overall complexity is $O\left(2^{2n}\right)$. If, however, we sort and merge the two sets or store the values of one set in a data structure suited for searching – such as balanced binary search trees – we immediately reduce this bound to $O\left(n \times 2^n\right)$; this also makes it immediately natural to eliminate duplicates, lowering the number of stored values, as the first comparators in comparator networks tend to halve the number of outputs. (Empirical observations show that, for 9 inputs, 10-comparator networks already have as few as 80 outputs on average, rather than $2^9 = 512$.)

Again, the interesting aspect is that we can incorporate this optimization into the checker with very few changes to the formalization. The biggest work is defining a type of binary search trees (since, unfortunately, no formalization of them seems to exist with an arbitrary type T as parameter). Using these, we define an alternative notion of subsumption, prove that it coincides with the old one, and reprove decidability of the old notion by reducing to the new one. This guarantees that none of the remainder of the development needs to be changed, but the algorithm extracted is now the optimized one.

```
Definition opt_outputs C n := list_to_BTree _ (bin_seq_compare _) (outputs C n).

Definition subs_opt (n:nat) (C C':comparator_network) (P:permut) (HP:permutation n P) :=
  ∀ s:bin_seq n, BT_In s (opt_outputs C n) → BT_In (apply_perm P s) (opt_outputs C' n).

Lemma subs_opt_to : ∀ n C C' P HP, subs_opt n C C' P HP → subsumption n C C' P HP.

Lemma subsumption_to_opt : ∀ n C C' P HP, subsumption n C C' P HP → subs_opt n C C' P HP.

Lemma subs_opt_dec : ∀ n C C' P HP, {subs_opt n C C' P HP} + {~subs_opt n C C' P HP}.

Lemma subsumption_dec : ∀ n C C' P HP,
                               {subsumption n C C' P HP} + {~subsumption n C C' P HP}.
```

Each of these proofs is less than ten lines long; the impact, however, is a substantial reduction of the time spent validating the information provided by the oracle.

Exploiting the oracle. The asymptotically dominating step in our algorithm is Prune, whose execution time depends linearly both on the size of the oracle and on the size of the set to be pruned. We observed earlier that we could reduce the dependency on the latter to a logarithmic one; however, we can do even better.

Essentially, each Prune step needs to do three things.

1. Check that all subsumptions are valid.
2. Remove all subsumed networks.
3. Check that all networks used in subsumptions are kept.

Each subsumption in step 1 is checked individually, so this step scales linearly in the number of networks. The other two steps can be significantly improved.

Step 2 can be optimized substantially by delaying the removals until all subsumptions have been read. Thus, we reprogram our oracle to provide the networks to be removed in exactly the same order as they are generated in our checker. An inspection of the definition of Generate shows that this order is the lexicographic order on the sequence of integers representing the comparators; by pre-processing the oracle information accordingly, we can remove all subsumed networks in one single pass over the whole set. Then we can use the following function `remove_all` to complete step 2 in linear time, where we use the fact that `l` is a subsequence of `w` (so, when `x` and `y` are different, `y` is smallest).

```
Variable A:Type.
Variable A_dec : ∀ x y : A, {x = y} + {x <> y}.

Fixpoint remove_all (l w:list A) := match l,w with
```

```
  | nil, _ ⇒ w
  | _, nil ⇒ w
  | x:: l', y:: w'  ⇒ match (A_dec x y) with
                      | left _ ⇒ remove_all l' w'
                      | right _ ⇒ y :: remove_all l w'
end end.
```

Applying similar ideas to step 3 cannot be done directly, however, since sorting the oracle information on the *subsumed* networks yields an unsorted sequence of *subsuming* networks. Instead, we begin by observing that, rather than checking that the subsuming networks are kept at each step, we can check that they are present in the final (reduced) set, `R'`, the size of which, in the most time-consuming steps, is only around 5% of the size of the original one. This requires some care, as we explain below. We begin by evaluating the practical impact of such a change.

The number of subsumed networks is bounded by $|\mathtt{O}|$, so by using a balanced binary search tree for storing the subsuming networks, we may eliminate duplicates as we go along, and complete the process in time $O(|\mathtt{O}|\log|\mathtt{R'}|)$. Additionally, we can afterwards retrieve an ordered sequence of the networks in `R'` in $O(|\mathtt{R'}|)$. Theoretically, we could do even better. What is demanded of the storage structure for `R'` is fast insertion, and ordered retrieval in linear time after all insertions. Since we have an upper bound on the universe size, U (all possible bitstrings representing a network of at most 25 comparators), we could use a van Emde Boas tree [18] instead, and the time for one insertion improves to $O(\log\log U)$. Since there are only 37 different comparators over 9 inputs, one can be represented using 6 bits, and, thus, a network can be represented using 150 bits. This means that $\log\log U < 8$, so the time is really $O(|\mathtt{O}|)$.

The benefit of such optimizations has to be weighed against the added complexity and ensuing challenges in reproving the correctness of the algorithms. Thus with the goal of a formal verification in mind, a good enough solution allowing to reap the main part of the potential benefits is often preferable. Consequently, in the Coq code below, we simply use a binary search tree without a balancing scheme. This rather simple optimization has proved sufficient in significantly reducing the runtime of the extracted proof checker.

The formalized definitions for the improved pruning step now look as follows. The function `run_oracle` receives information from the oracle, checking that all subsumptions are valid (step 1), and stores the networks to be removed in a list (keeping their order) and those justifying the subsumptions in a binary tree (which also removes duplicates). Then `oracle_test` performs the test in step 3. The definition of `Prune` combines the three steps: it first applies `run_oracle` (step 1), then removes all subsumed networks using `remove_all` (step 2), and finally checks that all the necessary networks are present (step 3).

```
Definition Oracle := list (OCN * OCN * (list nat)).
Definition BTree := BinaryTree OCN.

Fixpoint run_oracle (n:nat) (O:Oracle) : bool * (BTree * (list OCN)) := match O with
  | nil ⇒ (true,(nought,nil))
  | (C,C',P) :: O' ⇒ match (pre_permutation_dec n P) with
               | right _ ⇒ (false,(nought,nil))
               | left A ⇒ match (subsumption_dec n (OCN_to_CN n C) (OCN_to_CN n C')
                                        (make_permutation P)
                                        (pre_permutation_lemma _ _ A)) with
                      | right _ ⇒ (false,(nought,nil))
                      | left _ ⇒ let (b,Tl) := (run_oracle n O') in let (T,l) := Tl in
```

```
                                (b,( BT_add OCN_compare C T,C'::l))
  end end end.

Fixpoint oracle_test (F:BTree) (R:list OCN) := match F with
  | nought ⇒ true
  | _      ⇒ match R with
         | nil ⇒ false
         | C' :: R' ⇒ let (C,F') := (BT_split F nil) in match (OCN_eq_dec C C') with
                    | left _ ⇒ oracle_test F' R'
                    | right _ ⇒ oracle_test F R'
  end end end.

Definition Prune (O:Oracle) (R:list OCN) (n:nat) : list OCN :=
  match (run_oracle n O) with
  | (false,_) ⇒ R
  | (true,XY) ⇒ let (X,Y) := XY in
           let R' := remove_all OCN_eq_dec Y R in
           match (oracle_test X R') with
           | false ⇒ R
           | true ⇒ R'
  end end.
```

Yet again, we find that these changes are completely modular: reproving the correctness of `Prune` requires proving correctness of its auxiliary functions, which can be done mostly by adapting the old proofs. The correctness results for the checker then remain unchanged, and their proofs require only minor tweaking, with trivial changes that require no deep insights.

The optimization of step 3 requires further changes to the oracle. The implementation of `Prune` now fully prunes the set and checks that the networks used for subsumption are present in the final set; this is however not true of the data obtained from the original proof in [11]. There are often chains of subsumptions $C_1 \preceq C_2 \preceq \ldots \preceq C_n$, which pose no problem for the original algorithm, but would now result in a false negative result; consider C_2, for example, which is used to remove C_3, but which is itself removed by C_1. Using the transitivity of subsumption, we can replace such chains by "reduced" subsumptions $C_1 \preceq C_2$, $C_1 \preceq C_3$, ..., $C_1 \preceq C_n$; once again, this requires pre-processing the oracle information, identifying such chains and computing adequate permutations for the new resulting subsumptions, which is only possible due to the offline nature of the oracle.

For completeness, we summarize this process, which is reminescent of topological sorting. In the pre-processing, we define a labeled graph whose nodes are comparator networks, and where there is an edge between C' and C labeled by π if $C \preceq_\pi C'$. Since subsumed networks occur at most once in a triple in the oracle, this graph can be viewed as a forest, where, in each tree, C would be a parent of C' if $C \preceq_\pi C'$. For each tree, we can now start with the root and move towards the leaves, composing permutations (starting with the identity permutation) that are found on the path towards any node – internal as well as leaves. This enables us to create a mapping from networks C_i to pairs (C_1, π), where C_1 is the root of C_i's tree and π is the composition of all the permutations on the path from C to C_i. Finally, in a sweep of the oracle information, the mapping is used to create the "reduced" subsumptions.

After completing these optimizations, our checker was able to verify the original proof of optimality of 25 comparators for sorting 9 inputs in 163.8 hours, or just under one week, compared to the 18 CPU years required to obtain the result in the first place [11]. The original proof was generated by a Prolog program consisting of approximately 100 clauses (1000 lines, or 35 KB), and produced 70 million proof witnesses corresponding to 27 GB of oracle data. This data was pre-processed in a few hours, as described above, by a Java program (288 lines, or 11 KB). The full Coq formalization consists of 6838 lines of code, including 102 definitions and 405 lemmas, with a total file size of 206 KB. The extracted Haskell program is only around 650 lines of Haskell code, or 16 KB, and interacts with the oracle by means of an interface program (53 lines, or 1.6 KB), also written in Haskell.

5 Concluding Remarks

We have presented a formal verification of the computer-generated proof of optimality of 25 comparators for sorting 9 inputs. We have shown that a straightforward application of standard verification strategies is not able to succeed at this task, and we proposed a successful alternative approach that capitalizes on optimizing the data provided by an oracle to guide the proof. Furthermore, we have exemplified how previous knowledge of all the information that will be provided by the oracle is instrumental in significantly reducing both execution time and asymptotic complexity of the formal proof.

Concretely, we reduced the original time complexity of $O(|\texttt{O}| \cdot |\texttt{R}|)$, where `O` is the oracle and `R` is a parameter of the problem of size comparable to the size of `O`, to $O(|\texttt{O}| \log|\texttt{R'}|)$, where `R'` is about 5% of the size of `R`, or even to time $O(|\texttt{O}|)$. Practically, our checker was able to verify the original proof of optimality of 25 comparators for sorting 9 inputs in 163.8 hours, or just under one week, compared to an estimate of 20 years using the initial formalization.

The approach taken in this work is described in detail in the context of the Coq theorem prover. The choice of Coq naturally dictates some design decisions, in particular the use of program extraction. Other theorem provers may allow for other decisions regarding the integration of untrusted oracles (e.g. in ACL2, one might use meta-rules to integrate oracle data). However, we believe that the general idea of starting with a simple formalization, which is successively optimized in lock step with the oracle data, has great potential benefits regardless of the actual theorem prover used.

Although we have focused on one case study, the characteristics that make this approach successful can be found in many other problems regarding formal verification of large-scale computer-generated proofs. Furthermore, the types of optimization we performed are based on generic (and rather basic) computer science techniques, rather than on intrinsic properties of our problem. Case in point, using a similar approach we have very recently completed a full formal verification of the proof of the Boolean Pythagorean Triples conjecture [25], which required verifying billions of cases using 200 TB of proof witnesses.

Acknowledgements The authors would like to thank Mike Codish for all the fruitful discussions on sorting networks and Pierre Letouzey for helping with the extraction mechanism and giving suggestions on how to improve the performance of the extracted code.

References

1. Appel, K., Haken, W.: Every planar map is four colorable. Part I: discharging. Illinois J. of Math. **21**, 429–490 (1977)

2. Appel, K., Haken, W.: The four color proof suffices. The Mathematical Intelligencer **8**(1), 10–20 (1986)
3. Appel, K., Haken, W., Koch, J.: Every planar map is four colorable. Part II: reducibility. Illinois J. of Math. **21**, 491–567 (1977)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Texts in Theoretical Computer Science. Springer (2004)
5. Blanqui, F., Koprowski, A.: Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. Mathematical Structures in Computer Science **21**, 827–859 (2011)
6. Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.): Interactive Theorem Proving, ITP 2013, Proceedings, *LNCS*, vol. 7998. Springer (2013)
7. Bundala, D., Závodný, J.: Optimal sorting networks. In: A.H. Dediu, C. Martín-Vide, J.L. Sierra-Rodríguez, B. Truthe (eds.) LATA, *LNCS*, vol. 8370, pp. 236–247. Springer (2014)
8. Chung, M.J., Ravikumar, B.: Bounds on the size of test sets for sorting and related networks. Discrete Mathematics **81**(1), 1–9 (1990)
9. Claret, G., González-Huesca, L., Régis-Gianas, Y., Ziliani, B.: Lightweight proof by reflection using a posteriori simulation of effectful computation. In: Blazy et al. [6], pp. 67–83
10. Codish, M., Cruz-Filipe, L., Frank, M., Schneider-Kamp, P.: Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In: ICTAI 2014, pp. 186–193. IEEE (2014)
11. Codish, M., Cruz-Filipe, L., Frank, M., Schneider-Kamp, P.: Sorting nine inputs requires twenty-five comparisons. Journal of Computer and System Sciences **82**(3), 551–563 (2016)
12. Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: Automated certified proofs with CiME3. In: M. Schmidt-Schauß (ed.) RTA 2011, *LIPIcs*, vol. 10, pp. 21–30. Schloss Dagstuhl (2011). URL `http://www.dagstuhl.de/dagpub/978-3-939897-30-9`
13. Cruz-Filipe, L., Letouzey, P.: A large-scale experiment in executing extracted programs. Electronic Notes in Computer Science **151**(1), 75–91 (2006)
14. Cruz-Filipe, L., Schneider-Kamp, P.: Formalizing size-optimal sorting networks: Extracting a certified proof checker. In: C. Urban, X. Zhang (eds.) ITP 2015, *LNCS*, vol. 9236, pp. 154–169. Springer (2015)
15. Cruz-Filipe, L., Schneider-Kamp, P.: Optimizing a certified proof checker for a large-scale computer-generated proof. In: M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, V. Sorge (eds.) CICM 2015, *LNAI*, vol. 9150, pp. 55–70. Springer (2015)
16. Cruz-Filipe, L., Wiedijk, F.: Hierarchical reflection. In: K. Slind, A. Bunker, G. Gopalakrishnan (eds.) TPHOLs, *LNCS*, vol. 3223, pp. 66–81. Springer (2004)
17. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: ICTAC, *Lecture Notes in Computer Science*, vol. 6255, pp. 260–274. Springer (2010)
18. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Information Processing Letters **6**(3), 80–82 (1977)
19. Erkök, L., Matthews, J.: Using Yices as an automated solver in Isabelle/HOL. In: Automated Formal Methods'08, Princeton, New Jersey, USA, pp. 3–13. ACM Press (2008)
20. Floyd, R., Knuth, D.: The Bose–Nelson sorting problem. In: A survey of combinatorial theory, pp. 163–172. North-Holland (1973)
21. Fouilhé, A., Monniaux, D., Périn, M.: Efficient generation of correctness certificates for the abstract domain of polyhedra. In: F. Logozzo, M. Fähndrich (eds.) SAS 2013, *LNCS*, vol. 7935, pp. 345–365. Springer (2013)
22. Gonthier, G.: Formal proof – the four-color theorem. Notices of the AMS **55**(11), 1382–1393 (2008)
23. Harrison, J.: HOL Light: A tutorial introduction. In: M.K. Srivas, A.J. Camilleri (eds.) FMCAD, *Lecture Notes in Computer Science*, vol. 1166, pp. 265–269. Springer (1996)
24. Harrison, J., Théry, L.: A skeptic's approach to combining HOL and maple. J. Autom. Reasoning **21**(3), 279–294 (1998)
25. Heule, M., Kullmann, O., Marek, V.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: N. Creignou, D. Le Berre (eds.) SAT 2016, *Lecture Notes in Computer Science*, vol. 9710, pp. 228–245. Springer (2016)
26. Johnson, D.S.: The np-completeness column: An ongoing guide. Journal of Algorithms **3**, 288–300 (1982)
27. Knuth, D.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley (1973)

28. Konev, B., Lisitsa, A.: A SAT attack on the Erdős discrepancy conjecture. In: C. Sinz, U. Egly (eds.) SAT 2014, *LNCS*, vol. 8561, pp. 219–226. Springer (2014)
29. Krebbers, R., Spitters, B.: Computer certified efficient exact reals in Coq. In: J. Davenport, W. Farmer, J. Urban, F. Rabe (eds.) Calculemus 2011, *LNCS*, vol. 6824, pp. 90–106. Springer (2011)
30. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
31. Letouzey, P.: Extraction in Coq: An overview. In: A. Beckmann, C. Dimitracopoulos, B. Löwe (eds.) CiE 2008, *LNCS*, vol. 5028, pp. 359–369. Springer (2008)
32. McBride, C.: Elimination with a motive. In: P. Callaghan, Z. Luo, J. McKinna, R. Pollack (eds.) TYPES, *LNCS*, vol. 2277, pp. 197–216. Springer (2002)
33. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Springer (2002)
34. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology (2007)
35. O'Connor, R.: Certified exact transcendental real number computation in Coq. In: O. Mohamed, C. Muñoz, S. Tahar (eds.) TPHOLs 2008, *LNCS*, vol. 5170, pp. 246–261. Springer (2008)
36. Oury, N.: Observational equivalence and program extraction in the Coq proof assistant. In: M. Hofmann (ed.) TLCA 2003, *LNCS*, vol. 2701, pp. 271–285. Springer (2003)
37. Parberry, I.: A computer-assisted optimal depth lower bound for nine-input sorting networks. Mathematical Systems Theory **24**(2), 101–116 (1991)
38. Sternagel, C., Thiemann, R.: The certification problem format. In: C. Benzmüller, B. Paleo (eds.) UITP 2014, *EPTCS*, vol. 167, pp. 61–72 (2014)
39. Thiemann, R.: Formalizing bounded increase. In: Blazy et al. [6], pp. 245–260
40. Van Voorhis, D.: Toward a lower bound for sorting networks. In: R. Miller, J. Thatcher (eds.) Complexity of Computer Computations, The IBM Research Symposia Series, pp. 119–129. Plenum Press, New York (1972)