Lower Bounds of Comparison-Based Algorithms via Prophecy Variables

Paulo Emílio de Vilhena
p.devilhena@surrey.ac.uk
University of Surrey
Guildford, United Kingdom

Abstract

We present a novel technique to formally verify lower bounds of *comparison-based algorithms*. Unlike previous work, where a comparison-based algorithm is modelled as a *decision tree* that fixes the order in which comparison queries are performed, we model an algorithm simply and more generally as a piece of code written in an idealised functional programming language with convenient features such as mutable state, concurrency, and random sampling. The key ingredient to enable this generalisation are *prophecy variables*, which allow one to refer to the trace of queries performed by the algorithm. This enables a methodology whereby, to prove a lower-bound statement for a given algorithm, we show the existence of, and verify, an *adversary comparison function* that cautiously decides on the ordering of elements according to the ongoing history of queries sent by the algorithm so far to reveal the least information about this ordering and force the algorithm to make as many comparison queries as possible. As case studies, we apply this methodology to verify lower bounds of multiple comparison-based algorithms including sorting and maximum-finding algorithms. The proofs are mechanised in Rocq using Iris.

1 Introduction

A *comparison-based algorithm* is, roughly speaking, an algorithm that computes its result based *"only on the comparisons between the input elements"* [Cormen et al. 2009]. For example, a comparison-based *sorting* algorithm sorts an input list by rearranging its elements based only on the information it gathers from comparing elements but never on properties specific to the elements themselves and how they are represented. Because such a sorting algorithm does not depend on the representation of the input elements, it can be formulated as a generic function that can be used with any datatype. In OCaml, or in any language with polymorphism and arrow types, such a sorting algorithm can thus be represented as a polymorphic function `sort` that, in addition to a list of elements, receives a comparison function implementing their ordering:

```
sort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

That, for any integer n greater than 0, there is always an input list of size n that makes `sort` perform at least $n \log_2 n$ comparisons (modulo a fixed constant factor) is an important and well-known claim that can be found in most algorithms textbooks. This lower-bound claim is important because it shows the limits of comparison-based algorithms and motivates the investigation of algorithms that exploit specific properties of input elements to achieve sorting in linear time, such as *counting sort* and *radix sort*.

To promote this claim to a fact, rigorous textbooks such as Cormen et al. [2009]'s and Erickson [2019b]'s rephrase this claim as a formal statement and present its accompanying proof. When formulating this lower-bound claim as a formal statement, such textbooks make the simplifying assumption that, for any given input size, every execution of a sorting algorithm can be abstracted by a traversal of a fixed *decision tree*, a binary tree where each node represents a $\leq$-comparison between two elements: the root indicates the next comparison and the branches dictate how the algorithm evolves depending on the binary outcome of the comparison query. To make the discussion precise, we take inspiration from Eberl [2017] and represent a decision tree in OCaml as a value of the following type:

```
type 'a dtree =
  | Return of 'a list
  | Query of ('a * 'a) * 'a dtree * 'a dtree
```

Specifically, a decision tree is either (1) a node `Query ((`a`,` b`),` dl`,` dr`)` carrying `'a` elements a and b and subtrees dl and dr respectively describing the following queries depending on whether the comparison between a and b returns `true` or `false`; or (2) a leaf `Return` xs carrying the result of the algorithm, a sorted list xs.

We can now state the simplifying assumption more precisely: it posits that, for any integer n, there exists a decision tree d such that, for any input list xs of size n, the queries

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

performed by the sorting algorithm correspond to a traversal of d, where left or right branches are chosen according to whether the comparison returns `true` or `false` and where the leaf represents the returned sorted permutation of *xs*.

To illustrate, consider the following implementation of *insertion sort* in OCaml:

```
let rec isort leq xs =
  match xs with
  | ([] | [_]) -> xs
  | a :: bs -> insert leq a (isort bs)
and insert leq a bs =
  match bs with
  | [] -> [a]
  | b :: bs' ->
    if leq a b then a :: bs else
      b :: insert leq a bs'
```

It is easy to see that, because `isort` simply returns its input list *xs* when *xs* is empty or a singleton list, the comparisons performed by `isort` in such cases correspond to a traversal of the singleton tree `Return` *xs*; that is, `isort` performs no comparison. Moreover, it is also easy to see that, for any list `[`a`;` b`]` of size two, `isort` always starts by comparing a with b before terminating with a correct sorting of the list. Therefore, the comparisons performed by `isort` in the case of a list `[`a`;` b`]` correspond to a traversal of the tree

`Query ((`a`,` b`), Return [`a`;` b`], Return [`b`;` a`]).`

We can informally show (by an inductive argument and by observing `isort` is *pure*) that this pattern of `isort` carries on to any input size; in other words, `isort` satisfies the simplifying assumption.

Unfortunately, as soon as any form of *non-determinism* such as random sampling, concurrency, or mutable state is present, this simplifying assumption becomes too restrictive. Indeed, even if the algorithm uses non-determinism internally in its description but still describes a deterministic function with respect to the relation between its inputs and outputs, two runs of such an algorithm with the same inputs can trigger different comparison queries. Therefore, there is no fixed decision tree for which the comparison queries performed by any run of the algorithm correspond to a traversal of the tree. This limitation excludes, for example, algorithms for *quicksort* [Hoare 1961] that use random sampling.

Our goal is to develop a methodology for proving lower bounds of comparison-based algorithms that overcomes this limitation. To this end, we reject the simplifying assumption that comparison-based algorithms behave like traversals of a fixed decision tree. In fact, we make no assumption on the implementation of algorithms or their internal behaviour, we simply assume that a comparison-based algorithm can be modelled as a piece of code (written in a realistic programming-language model with concurrency, dynamically allocatable state, and higher-order functions) that comes with a formal specification.

For instance, in the case of sorting, a comparison-based sorting algorithm is modelled as an arbitrary function `sort` that is correct according to the following specification: for any list of values *xs* and total order $\leq$ on *xs*, `sort` *leq xs* returns a correct sorting of *xs* with respect to $\leq$ (provided that *leq* offers a correct implementation of $\leq$). To show the lower bound of `sort`, the idea is to implement an *adversary* comparison function *leq* that decides on the ordering of elements $\leq$ on the fly so as to force `sort` to make as many calls to *leq* as possible. We then reduce the proof of `sort`'s lower bound to a *program-verification* problem, where we verify that a counter incremented at every *leq* call holds an integer greater than or equal to $n \log_2 n$ (where n is the size of the input list) when `sort` terminates.

We will see that the key for this approach to work is to perform the verification in a logic that like *Iris* [Jung et al. 2018], which we use in this paper, has support for *prophecy variables* [Abadi and Lamport 1991; Jung et al. 2020], which will allow us to refer to the future trace of comparison calls before `sort` executes. Throughout the next sections, we give a detailed overview of our approach (§2 and §4) with an introduction to prophecy variables (§3) before concluding with the proofs of the lower bounds of sorting and maximum-finding algorithms (§5 and §6).

2 Pairwise Maximum

To illustrate our approach, let us consider perhaps one of the simplest (and yet useful) comparison-based algorithms: a function `max` of type

```
max : ('a -> 'a -> bool) -> 'a -> 'a -> 'a
```

that, when applied to a comparison function *leq* and to two elements, returns the greatest of the two according to the (total) order implemented by *leq*.

Our goal is to prove there exists a set of inputs forcing `max` to make at least one call to *leq*. If we introduce the simplifying assumption that `max` performs comparisons according a traversal of a decision tree d (with leaves carrying single elements rather than lists), then the proof is easy: d has at least one node because, otherwise, `max` would be a constant function, as every traversal of d produces the same result.

For a simple function such as `max`, the simplifying assumption is reasonable, but, as previously explained, for more sophisticated algorithms like sorting, it is not. Therefore, for the sake of illustrating our approach, we proceed differently. We assume instead simply that `max` satisfies the following

specification (defined generally in terms of an arbitrary function *max* so this specification can be later reused):

$$\begin{array}{l} \mathit{ImplementsMax}\ \mathit{max} \triangleq \\ \quad \forall a\ b\ \mathit{leq} \le .\ \mathit{TotalOrder}\ (\{a,b\}, \le) \implies \\ \quad \{\ \mathit{ImplementsOrder}\ (\{a,b\}, \le)\ \mathit{leq}\ \} \\ \qquad \mathit{max}\ \mathit{leq}\ a\ b \\ \quad \{\ z.\ \mathit{isMax}\ (\{a,b\}, \le)\ z\ \} \end{array} \tag{1}$$

The specification says that for any pair of elements a and b, for any order $\le$ that is *total in the set* $\{a,b\}$[1], and for any comparison function *leq* that *implements* $\le$, the application `max` *leq a b* returns z such that the assertion $\mathit{isMax}\ (\{a,b\}, \le)\ z$ holds. This assertion is defined generally as follows:

$$\mathit{isMax}\ (X, \le)\ z \triangleq z \in X \land \forall x \in X.\ x \le z$$

So, in sum, `max` says it computes the greatest element in $\{a,b\}$ (according to $\le$), provided *leq* implements $\le$. The assumption that *leq* implements $\le$ is captured by the clause $\mathit{ImplementsOrder}\ (\{a,b\}, \le)\ \mathit{leq}$, which is defined, in a general fashion, as follows:

$$\begin{array}{l} \mathit{ImplementsOrder}\ (X, \le)\ \mathit{leq} \triangleq \\ \quad \forall x\ y.\ \{\ x \in X \land y \in X\ \}\ \mathit{leq}\ x\ y\ \{\ b.\ b \iff x \le y\ \} \end{array} \tag{2}$$

When `max` satisfies the specification *ImplementsMax*, we say `max` is a correct implementation of *pairwise maximum*. To make the discussion precise, we write both the specifications *ImplementsOrder* and *ImplementsMax* in the Iris program logic.[2] Because Iris is a strand of separation logic [O'Hearn 2019; Reynolds 2002], the specification of an imperative program does not necessarily mention how it manipulates local resources. This means that, even though *ImplementsOrder leq* and *ImplementsMax max* do not mention how *leq* and *max* modify the state, these functions are only *apparently pure*: they are allowed to internally use any imperative feature of the language including, for example, concurrency, random sampling, and memory operations.

This is a steep generalisation from the setting where we assumed `max` would always call its input function *leq* according to the traversal of a fixed decision tree. With no assumption about the internal behaviour of `max`, we must see `max` as a *black box* of which only its external behaviour is known (namely, that it computes the pairwise maximum). But then, how can we prove the property that `max` calls *leq* at least once, a property that concerns very much the internal behaviour of `max`?

[1] A tuple $(X, \le)$ with a set X and a relation $\le \subseteq X \times X$ is a *partial order* if $\le$ is reflexive, transitive, and anti-symmetric. It is a *total order* if, in addition to these three criteria, for any $x, y \in X$, either $x \le y$ or $y \le x$. We are often lax about the distinction between $(X, \le)$ and $\le$ and simply say $\le$ *is partial/total*. In the case of a set $\{a,b\}$, the order $\le$ being total implies either $a \le b$, $b \le a$, or $a = b$.

[2] The definitions can be found in our Rocq formalisation [Anonymous authors 2026]. The OCaml code is manually translated to Iris's default language HeapLang, a formal calculus with similar syntax and semantics to OCaml.

```
let lower_bound_witness max =
  let count = ref 0 in
  let cache = ref None in
  let trace = Proph.new () in
  let lock = Mutex.create () in

  let leq x y = Mutex.protect lock (fun () ->
    Proph.resolve trace y;
    incr count;
    match !cache with
    | Some w -> (x = y) || (y = w)
    | None -> cache := Some y; true
  ) in

  let z = max leq false true in
  Mutex.lock lock;
  if !count = 0 then begin
    Proph.resolve trace (not z);
    assert false
  end;
  !count
```

Figure 1. Witness of `max`'s lower bound.

The intuitive idea is to consider an application of `max` with a comparison function that gathers information about `max`'s inner workings; that is, a function that *spies* [de Vilhena et al. 2020; Longley 1999] on `max`. Indeed, consider an application of `max` with the arguments `false` and `true`[3] of the set $B \triangleq \{\texttt{true}, \texttt{false}\}$ and with a function leq_B that maintains a cache storing the second argument y it received the first time it was called (and the value `None` otherwise). This comparison function implements a carefully defined total order $\le_B$ on the set B: if leq_B's cache stores y, then $\le_B$ is defined by the ordering $(\texttt{not}\ y) \le_B y$, otherwise, if `max` does not call $\le_B$ (and therefore its cache is empty), then $\le_B$ is defined by the ordering $z \le_B (\texttt{not}\ z)$, where z is the output of `max`.[4] Consequently, if leq_B's cache stores y, then the maximum of B according to $\le_B$ is y, otherwise it is `not` z, the negation of `max`'s output. Now leq_B's mischievous plan becomes clear: it uses its cache to check whether it has been called and, if not, to implement an order that contradicts `max`'s claim that z is the maximum of B. Then, because we assumed the correctness of `max`, it follows by contradiction that `max` must have called leq_B!

[3] The specific values `false` and `true` are not important. Any other pair of distinguishable values would work.

[4] Because B has only two elements, a single ordering $x \le_B y$ with $x \neq y$ suffices to determine a total order in B.

Figure 1 summarises the construction of this *adversary* comparison function. (For now, ignore the lines in light purple containing the `Proph` operations.) The comparison function `leq` maintains a `cache`, which is initialised with `None` and then updated once with `Some` y as soon as `leq` receives a query with second argument y. When answering to a query with arguments x and y for the first time, `leq` simply sets y to the be maximum and replies `true`. On further queries, it first checks whether x and y are equal returning `true` in the positive case as justified by reflexivity. If x and y are not equal, then one of them must be the maximum of B stored in `cache`. Therefore, it suffices to check whether this is y.

`leq` also maintains a counter `count` to store the number of times it has been called. Of course, we are only concerned with whether or not `leq` is called. For this purpose, the information stored in `cache` is enough. However, having a separate counter makes the code more idiomatic.

Because `max` is allowed to use concurrency, a lock is needed to protect `cache` and `count` against data races. The function `Mutex.create` creates a lock, the function `Mutex.protect` handles the acquiring and releasing of a given lock around the execution of a thunk also given as an argument, and `Mutex.acquire` is used to acquire the lock after the execution of `max` (to regain ownership of the resources protected by the lock).

The lower-bound claim for `max` (namely, that for any correct implementation, there exists a set of inputs that forces the comparison function to be called at least once) can now be formalised as a specification of the program `lower_bound_witness` saying that the integer stored in `count` after `max` has returned must be greater than or equal to one:

Theorem 2.1. *Every correct pairwise-maximum algorithm calls the comparison function defined in* `lower_bound_witness` *(Figure 1) at least once:*

$$\begin{array}{l}\forall max.\\ \{\, \mathit{ImplementsMax}\ max \,\}\\ \quad \texttt{lower\_bound\_witness}\ max\\ \{\, k.\ k \geq 1 \,\}\end{array}$$

The proof of Theorem 2.1 corresponds to the formal verification that `lower_bound_witness` satisfies its specification. It roughly follows the intuitive explanation of why the comparison function leq_B must be called. However, the attentive reader will have noticed an apparent flaw with that explanation: the definition of $\leq_B$, which leq_B purports to implement, depends on the runtime information of leq_B and `max`, namely on whether leq_B has been called (and with which arguments) and on the output of `max`. How is this possible? This is where *prophecy variables* come in. We briefly introduce prophecy variables in the next section before coming back to a formal account of the proof of Theorem 2.1 in §4.

NewProph
$\{\, \mathit{True} \,\}$ `Proph.new ()` $\{\, p.\ \exists zs.\ p \rightsquigarrow zs \,\}$

ResolveProph
$\{\, p \rightsquigarrow zs \,\}$
`Proph.resolve` $p\ z$
$\{\, ().\ \exists zs'.\ p \rightsquigarrow zs' \wedge zs = z :: zs' \,\}$

Figure 2. Rules for prophecy variables in Iris.

3 Prophecy Variables

A prophecy variable stores a trace of *events*. What are these events will become clear soon, but, for now, it suffices to understand the interesting property that, when a prophecy variable is created, the prophecy is already initialised with this trace of events. Therefore, a prophecy variable stores information about the *future* (which property explains the term *prophecy*).

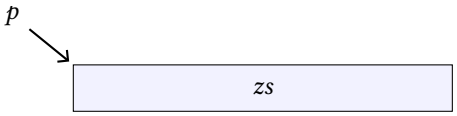
Of course, it would be unrealistic for a program to depend on the future. For this reason, a prophecy variable cannot be read. In fact, a prophecy variable has no effect on the operational behaviour of a program. From this perspective, a prophecy variable is only meaningful as a verification technique allowing one to refer to a future set of events and use this information for the purposes of the proof but never to interfere with the program's behaviour.

Prophecy variables were introduced by Abadi and Lamport [1991] and traditionally used in the verification of concurrent programs [Sezgin et al. 2010; Vafeiadis 2008]. They were first integrated to separation logic by Jung et al. [2020], who introduced them in Iris through a relatively simple interface.

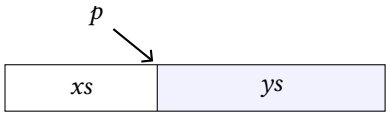
Figure 2 shows the interface for prophecy variables in Iris. It consists of two instructions: `Proph.new` and `Proph.resolve`. The instruction `Proph.new ()` can be used to create a prophecy variable p. Rule NewProph tells that p initially stores a list of values zs. This is expressed via the assertion $p \rightsquigarrow zs$. The list zs is the list of all the values z with which `Proph.resolve` $p\ z$ operations are performed and in the order they are performed. It is in this sense that zs can be viewed as a trace of *events*: an event is simply the execution of an operation `Proph.resolve`. This operation is called a *prophecy resolution*, because, by emitting an event z with `Proph.resolve` $p\ z$, we learn that z is the first among the events in zs, that is, zs is of the form $z :: zs'$, thus partially "resolving" the prophecy. This prophecy-resolution mechanism is captured by Rule ResolveProph, which requires $p \rightsquigarrow zs$ as a precondition and ensures $zs = z :: zs'$ and $p \rightsquigarrow zs'$. The inclusion of the assertion $p \rightsquigarrow zs'$ in `Proph.resolve`'s postcondition means that, in addition to the information that z is one of the events of the trace zs, the prophecy-resolution also updates p to store the remaining future events zs'. In this

way, a prophecy variable p will always store the list of upcoming prophecy-resolution events.

Pictorially, we can imagine that a prophecy variable p hovers over the trace of events zs indicating the suffix of the trace that remains to happen. Initially, a prophecy variable points to the totality of the trace zs:



Then, after a series of prophecy-resolution events corresponding to a prefix xs of zs, the prophecy p points to the suffix ys of zs such that $zs = xs +\!\!+ ys$:



$$zs = xs +\!\!+ ys$$

Soundness of prophecy variables. The soundness of prophecy variables in Iris is established by an *erasure* theorem [Jung et al. 2020, Theorem 3.2] stating that, if a program e is proved *safe* with respect to a pure predicate ϕ[5], then so is the version of e where all prophecy operations are erased. The operational semantics of the prophecy operations is modelled in such a way that any attempt to read a prophecy p, to compare two prophecy variables, or to use prophecies to influence program execution in any other way is considered a programming mistake. Therefore, if a program e is safe, then a prophecy variable does not affect e's operational behaviour. This implies that prophecy variables can be erased from a program e that has been verified safe in Iris (possibly using prophecy variables) while preserving e's semantics and safety guarantee.

4 Lower Bound of Pairwise Maximum

With prophecy variables, we can resume the discussion from §2 and give a more formal account of the proof of Theorem 2.1.

The prophecy variable `trace` in `lower_bound_witness` (Figure 1) is created via `Proph.new` and initialised with a list of values ys corresponding to the prophecy-resolution events remaining to happen. The meaning of ys thus depends on how `Proph.resolve` is called. There are two cases: either `max` calls `leq` or not. Whenever `leq` is queried at arguments x and y, `Proph.resolve` is called with y. Therefore, if `max` calls `leq`, then ys corresponds to the list of second arguments y at which `leq` is queried. If, on the other hand, `max` does not call `leq`, then `Proph.resolve` is called with `not` z,

[5]A program e is safe with respect to ϕ if no execution of e *crashes*; that is, any execution of e either diverges or terminates with a value v such that $\phi(v)$.

CreateLock

$$\frac{}{\{\, R \,\}\ \texttt{Mutex.create}\ ()\ \{\ lock.\ \mathit{IsLock}\ lock\ R\,\}}$$

Acquire

$$\frac{\mathit{IsLock}\ lock\ R}{\{\, \mathit{True}\,\}\ \texttt{Mutex.lock}\ lock\ \{\ ().\ R\,\}}$$

Release

$$\frac{\mathit{IsLock}\ lock\ R}{\{\, R\,\}\ \texttt{Mutex.unlock}\ lock\ \{\ ().\ \mathit{True}\,\}}$$

Protect

$$\frac{\mathit{IsLock}\ lock\ R \qquad \{\, P * R\,\}\ f\ ()\ \{\ y.\ Q\ y * R\,\}}{\{\, P\,\}\ \texttt{Mutex.protect}\ lock\ f\ \{\ Q\,\}}$$

Figure 3. Rules for handling locks.

the negation of `max`'s output. In this case, the list ys corresponds to `not` z `::` `_`. To summarise, the first element of ys is either the second argument y with which `max` calls `leq` the first time or the negation of `max`'s output z.

Recall that our informal definition of the total order $\leq_B$ essentially amounted to setting the maximum of B to be either y, if `max` calls `leq` with y, or the negation of `max`'s output, otherwise. The maximum of B therefore corresponds precisely to the first element of ys! We can exploit this relation to formally define $\leq_B$ as follows:

$$\top_B \triangleq \begin{cases} \mathit{head}\ ys & \mathit{if}\ ys \neq [] \\ \texttt{true} & \mathit{otherwise} \end{cases}$$

$$x \leq_B y \triangleq x = y \lor y = \top_B$$

When the prophecy `trace` is created, Rule NewProph guarantees simply the existence of ys with which `trace` is initialised, but, in particular, gives no guarantee on the size of ys. Therefore, when setting the maximum of B to be the head of ys we have to account for the case in which ys is empty. Here we choose the value `true`, but, because a prophecy resolution is always reachable in `lower_bound_witness`, we can always show ys is nonempty, so the choice of `true` is immaterial.

Given $\top_B$, the order $\leq_B$ is defined to contain pairs (x, y) whenever $x = y$, to ensure reflexivity, or whenever $y = \top_B$, to ensure $\top_B$ is indeed the maximum of B.

After the definition of $\leq_B$, the crux of Theorem 2.1's proof becomes to show that the comparison function `leq` (defined in `lower_bound_witness`) implements the order $\leq_B$; that is, to show that the following assertion holds:

$$\mathit{ImplementsOrder}\ (B, \leq_B)\ \texttt{leq} \tag{3}$$

This assertion is necessary to fulfil the precondition of `max`, whose specification, the assertion *ImplementsMax* `max`, requires a correct implementation of a total order on B. We

thus explain its proof (after a brief explanation of the necessary rules for locks) and then conclude the section with the contradiction argument that finishes the proof of Theorem 2.1.

Locks. Since the implementation of `leq` relies on a lock to protect its resources from concurrent racy accesses, to prove Specification 3 we need the rules for manipulating locks shown in Figure 3. These rules are standard in separation logic. They are expressed in terms of the assertion *IsLock lock R*, which asserts that *lock* protects the resources *R*. Rule CreateLock specifies the creation of a lock *lock* to protect *R*, Rule Acquire gives access to *R*, Rule Release transfers ownership of *R* back to the lock, and Rule Protect lets the thunk *f* access the resources in *R* provided *f* can return them back at the end of its execution.

The resources protected by the lock `lock` in the program `lower_bound_witness` are described by the assertion *Inv*:

$$Inv \triangleq \begin{pmatrix} \texttt{trace} \rightsquigarrow ys * \\ \texttt{cache} \mapsto \texttt{None} * \\ \texttt{count} \mapsto 0 \end{pmatrix} \vee \begin{pmatrix} \exists ys'.\ \texttt{trace} \rightsquigarrow ys' * \\ \texttt{cache} \mapsto \texttt{Some}\ \top_B * \\ \exists k.\ \texttt{count} \mapsto k * k \geq 1 \end{pmatrix}$$

Inv is defined as a disjunction of two clauses. The left clause describes `trace`, `cache`, and `count` at their initial state, whereas the right clause describes them after `leq` has been called. It is clear that *Inv* holds when the lock is created, so the assertion *IsLock* `lock` *Inv* is justified by Rule CreateLock.

Proof of Specification 3. By unfolding the definition of *ImplementsOrder*, the proof of Specification 3 simplifies to the following goal (where x and y are universally quantified):

```
{ x ∈ B ∧ y ∈ B }
  Mutex.protect lock (fun () ->
    Proph.resolve trace y;
    incr count;
    match !cache with
    | Some w -> (x = y) || (y = w)
    | None -> cache := Some y; true
  )
{ b. b ⟺ x ≤_B y }
```

We start by applying Rule Protect, which lets us assume *Inv* but which also requires us to prove that *Inv* is left invariant by `leq`.

The proof proceeds by case disjunction on *Inv*.

The case where the right disjunct of *Inv* holds corresponds to when `leq` is being called a second or further time. The reference `cache` stores `Some` $\top_B$, so the `Some` branch of the pattern match is taken with `w` bound to $\top_B$. Recall that $x \leq_B y$ is defined as $(x = y) \vee (y = \top_B)$. Hence, it is easy to see that the expression `(x = y) || (y = ⊤_B)` implements $x \leq_B y$ correctly. It is also easy to see that the invariant *Inv* is maintained because (1) `cache` is not modified, (2) the increment to `count` preserves the property that its content is greater than or equal to 1, and (3) the prophecy resolution keeps ownership of `trace`.

The case where the left disjunct of *Inv* holds corresponds to when `leq` is being called for the first time. In this case, the prophecy `trace` still stores the complete trace *ys*, so, after the prophecy resolution with *y*, we learn that $ys = y :: ys'$, for some ys', and that `trace` $\rightsquigarrow ys'$ holds. Recall that $\top_B$ is defined as the head of the list *ys*. Therefore, after the prophecy resolution, we learn that $\top_B = y$. This justifies the update to `cache`, since it proves `cache` contains `Some` $\top_B$ at the end. Moreover, with the increment to `count`, we have that *Inv* is also maintained in this case, because *Inv*'s right disjunct can be easily proved. The equality $\top_B = y$ also justifies that `true` is the correct output in this case, because $x \leq_B \top_B$.

Conclusion of the proof of Theorem 2.1. We use the proof that `leq` implements $\leq_B$ in conjunction with the specialisation of `max`'s specification (*ImplementsMax* `max`) to the assertion

$$\begin{array}{l} \{\ \mathit{ImplementsOrder}\ (B, \leq_B)\ \texttt{leq}\ \} \\ \quad \texttt{max leq false true} \\ \{\ z.\ \mathit{isMax}\ (B, \leq_B)\ z\ \} \end{array}$$

to conclude that the output z of `max` is the maximum of B with respect to $\leq_B$, that is, $z = \top_B$.

We then apply Rule Acquire to show that `Mutex.acquire` transfers *Inv* back to `lower_bound_witness` and proceed again by case disjunction on *Inv*.

The case where the right disjunct of *Inv* holds is straightforward. Since `count` stores an integer $k \geq 1$, the `if` branch is skipped. It is then easy to see that the result k satisfies `lower_bound_witness`'s postcondition.

Now, assume the left disjunct of *Inv* holds. In this case, the prophecy `trace` still stores the complete trace *ys* and `count` holds 0, so the `if` branch is taken. After the prophecy resolution with `not` z, we learn that $ys = \texttt{not}\ z :: ys'$. We can then derive the chain of equations

$$\begin{array}{rll} z & = \top_B & \textit{(by correctness of } \texttt{max}\textit{)} \\ & = \mathit{head}\ ys & \textit{(by definition)} \\ & = \texttt{not}\ z & \end{array}$$

showing that $z = \texttt{not}\ z$, a contradiction. Therefore, it cannot be the case that `count` holds 0.

5 Lower Bound of List Maximum

The detailed explanation of pairwise maximum's lower bound helps to clarify the general structure of our methodology. In the interest of conciseness, we now give a less detailed account focusing only on the main aspects of the proofs.

To provide a smooth transition to the presentation of the lower bound of sorting (§6), in this section we present the lower bound of maximum-finding algorithms on lists. This problem is a good illustration of how we model an asymptotic

```
let find adj x =
  match Hashtbl.find_opt adj x with
  | None -> []
  | Some ys -> ys

let insert adj x y =
  Hashtbl.replace adj y (x :: find adj y)

let rec downset adj x =
  let ys = find adj x in
  let yss = List.map (downset adj) ys in
  x :: List.concat yss

let reach adj x y = List.mem x (downset adj y)

let is_maximal adj w zs =
  List.for_all (fun z ->
    w = z || not (reach adj w z)) zs

let lower_bound_witness list_max n =
  let count = ref 0 in
  let adj = Hashtbl.create n in
  let trace = Proph.new () in
  let lock = Mutex.create () in

  let leq x y = Mutex.protect lock (fun () ->
    Proph.resolve trace (x, y);
    incr count;
    (reach adj x y ||
      (not (reach adj y x) &&
        (insert adj x y; true)))
  ) in

  let xs = List.init n (fun i -> i) in
  let z = list_max leq xs in
  Mutex.lock lock;
  if !count < n - 1 then begin
    List.iter (fun w ->
      if z <> w && is_maximal adj w zs then
        Proph.resolve trace (z, w)
    ) zs;
    assert false
  end;
  !count
```

Figure 4. Witness of `list_max`'s lower bound.

lower bound and, moreover, it shares most of the mathematical definitions with the sorting case study.

Consider a function `list_max` of type

```
list_max : ('a -> 'a -> bool) -> 'a list -> 'a
```

that, when applied to a comparison function *leq* and to a nonempty list *xs*, returns the maximum element of *xs*. We call this problem of finding the maximum of a nonempty the problem of *list maximum*.

Our goal is to prove that, for every n, any comparison-based algorithm must make $n-1$ comparisons to find the maximum of a list of size n.

Following the methodology explained in §2, we proceed in four steps:

1. Restrict the functions `list_max` to those that satisfy a correctness specification.
2. Write a function `lower_bound_witness` that applies `list_max` to an adversary comparison function `leq` triggering the wished lower-bound behaviour.
3. Formulate the lower-bound claim as a specification of `lower_bound_witness`.
4. Prove the specification of `lower_bound_witness`.

Step 1 (Specify* `list_max`*). The first step is direct:

$$\begin{array}{l} \mathit{ImplementsListMax}\ \mathit{list\_max} \triangleq \\ \quad \forall \mathit{xs}\ \mathit{leq} \le .\ \mathit{xs} \neq [] \land \mathit{TotalOrder}\ (\mathit{xs}, \le) \implies \\ \quad \{\ \mathit{ImplementsOrder}\ (\mathit{xs}, \le)\ \mathit{leq}\ \} \\ \qquad \mathit{list\_max}\ \mathit{leq}\ \mathit{xs} \\ \quad \{\ z.\ \mathit{isMax}\ (\mathit{xs}, \le)\ z\ \} \end{array}$$

The specification generalises *ImplementsMax* from the case of two elements to the case of a list *xs*. The term $(xs, \le)$ is a deliberate abuse of notation conflicting a list with a set.

Step 2 (*`lower_bound_witness`*). An implementation of `lower_bound_witness` that achieves the second step appears in Figure 4. In a nutshell, the idea of the implementation is to apply `list_max` to a function `leq` that incrementally builds an ordering on the elements of the **input list** $xs \triangleq [0; \ldots; n-1]$. We then show that, if `list_max` does not query `leq` a minimum amount of times, there will be enough variability in the order implemented by `leq` so that the maximality of `list_max`'s output can be contradicted (by adding a new ordering via a prophecy resolution).

The comparison function `leq` in `lower_bound_witness` implements this incremental order construction by means of a hash map `adj`. If we think of `adj` as the representation of a graph G where nodes are elements of *xs*, then the order implemented by `leq` (up to a certain point of the history of comparison queries) is the *reachability relation* in G, whereby x is lesser than or equal to y if x is reachable from y in G.

Initially, before `leq` has received any queries, the map `adj` is empty, so the order in *xs* is unspecified (apart from the reflexive orderings). Then, for each query at x and y received by `leq`, if neither is reachable from the other, the map `adj` is updated so that x becomes reachable from y.

The function `reach` implements the reachability check in `adj`. Specifically, to check whether x is reachable from y, it checks membership of x in the *downset* of y, the set of all nodes reachable from y. The function `downset` computes this set and represents it as a list. Duplicates can occur in this list, because two nodes can have a common neighbour. Moreover, a realistic implementation of `reach` could completely bypass this construction by using one of the traditional graph traversal algorithms (depth-first search, breadth-first search, etc.). These efficiency concerns however are not important. The purpose of `lower_bound_witness` is only to provide information about `list_max`'s complexity behaviour (in terms of number of comparison queries). We therefore choose simplicity over efficiency.

We discuss the remaining aspects of the implementation (namely the prophecy variables and the `if`-branch at the end of `lower_bound_witness`) in the fourth step, when we explain the proof of `lower_bound_witness`'s specification.

Step 3 (Formal lower-bound claim). The lower-bound claim is formalised as the following theorem:

Theorem 5.1. *Every correct list-maximum algorithm calls the comparison function defined in* `lower_bound_witness` *(Figure 4) at least $n-1$ times:*

$$\begin{array}{l}\forall \mathit{list\_max}\, n.\ n > 0 \implies \\ \{\ \mathit{ImplementsListMax}\ \mathit{list\_max}\ \} \\ \quad \texttt{lower\_bound\_witness}\ \mathit{list\_max}\ n \\ \{\ k.\ k \geq n-1\ \}\end{array}$$

The theorem states that the final deference of the counter `count` returns an integer k that is greater than or equal to $n-1$. Because this bound holds for any n that is greater than a constant (in this case 0), the specification shows a lower bound on the the asymptotic worst-case complexity of any list-maximum algorithm.

Step 4 (Proof of the lower-bound claim). The proof of Theorem 6.1 follows the overall structure of Theorem 2.1's proof (explained in §4): first, we explain how to formally define the order $\leq_{xs}$ implemented by `leq`; second, we describe the lock-protected resource *Inv*, which works as an invariant of the proof; third, we show that `leq` is a correct implementation of $\leq_{xs}$ (showing, in passing, that it keeps *Inv* invariant); finally, we finish the proof with a contradiction argument.

Formal definition of $\leq_{xs}$. Recall that we informally defined the order $\leq_{xs}$ as the reachability relation induced by the graph incrementally constructed by `leq` (and stored in `adj`). Like in §2, this informal definition has the problem that it depends on the runtime information accumulated by `leq`. Moreover, the reachability relation induced by the edges of a graph is not guaranteed to be a total order because the graph is not necessarily connected. In fact, this relation might not even be a partial order, because the graph could be cyclic thus invalidating the anti-symmetry requirement.

To solve the dependency problem, we use the prophecy variable `trace`. Because the prophecy-resolution operation `Proph.resolve` is called with (x, y) every time `leq` is queried at these elements, the trace stored in `trace` is the **list of queries** qs sent to `leq`. We fix qs from now on.

To then define the order using qs, we need to first formalise the incremental construction of the adjacency map. We do so through the function *fromEdges*:

$$\begin{array}{l}\mathit{fromEdges}'\ \mathit{Adj}\ [\,] \triangleq \mathit{Adj} \\ \mathit{fromEdges}'\ \mathit{Adj}\ ((x, y) :: es) \triangleq \\ \quad \mathit{fromEdges}'\ (\mathit{insert}\ (x, y)\ \mathit{Adj})\ es\end{array}$$

$$\mathit{fromEdges}\ es \triangleq \mathit{fromEdges}'\ \emptyset\ es$$

The function *insert* is the mathematical counterpart of `insert`. Unlike `insert`, however, *insert* only adds an edge (x, y) to the graph if neither x nor y is reachable from the other. (`insert` is called only when this condition is met, so the check is not necessary.) Like in `reach`, reachability is defined using the notion of the downset of an element x, noted here $\downarrow_{Adj} x$:

$$\downarrow_{Adj} x \triangleq \{x\} \cup \left(\bigcup_{y \in Adj(x)} \downarrow_{Adj} y\right)$$

$$\begin{array}{l}\mathit{insert}\ (x, y)\ \mathit{Adj} \triangleq \\ \quad \mathit{if}\ x \in \downarrow_{Adj} y \ \vee\ y \in \downarrow_{Adj} x\ \mathit{then} \\ \quad\quad \mathit{Adj} \\ \quad \mathit{else} \\ \quad\quad [y \mapsto x :: Adj(y)]\mathit{Adj}\end{array}$$

It is easy to see that, for any es, the graph *fromEdges es* is acyclic by an inductive argument on es. We can then conclude that the reachability relation is a partial order:

Lemma 5.2. *Let Adj be the adjacency map fromEdges es. The order $x \leq_{Adj} y$ defined as $x \in \downarrow_{Adj} y$ is a partial order.*

To derive a total order out of the partial order $\leq_{Adj}$, we simply take any topological ordering of xs consistent with the order $\leq_{Adj}$. We note this extension of $\leq_{Adj}$ to a total order as $\widehat{\leq}_{Adj}$. The choice of the particular topological ordering is immaterial because, regardless of the choice, the property $\leq_{Adj} \subseteq \widehat{\leq}_{Adj}$ holds and because (thanks to the prophecy resolutions) we can always infer sufficient information about the order $\leq_{Adj}$ to carry on the proof.[6]

We can now finally introduce the formal definition of the total order $\leq_{xs}$ implemented by `leq`:

$$x \leq_{xs} y \triangleq x \mathrel{\widehat{\leq}_{\mathit{fromEdges}\ qs}} y$$

Lock-protected invariant. Recall that the prophecy variable `trace` initially stores qs, the entire list of prophecy-resolution events. The lock-protected invariant maintains

[6] This is similar to how the specific choice of `true` for $\top_B$ when $ys = [\,]$ is immaterial, since there is always a path in the implementation of `lower_bound_witness` of Figure 1 that proves $ys \neq [\,]$ (via a prophecy resolution).

the property that `trace` always points to the list of remaining events and that `count` stores the number of past events:

$$Inv \triangleq \exists l\, r.\ \texttt{count} \mapsto |l| * \texttt{trace} \rightsquigarrow r * qs = l +\!\!+ r$$

Correctness of `leq`. With these definitions in place, it is straightforward to show that `leq` implements $\leq_{xs}$ (and that it preserves *Inv*):

$$\mathit{ImplementsOrder}\ (xs, \leq_{xs})\ \texttt{leq}$$

Concluding contradiction argument. With the proof that `leq` implements $\leq_{xs}$, we can use `list_max`'s specification to conclude its output z is the maximum element of xs with respect to $\leq_{xs}$:

$$\{\ \mathit{ImplementsOrder}\ (xs, \leq_{xs})\ \texttt{leq}\ \}$$
$$\quad \texttt{list\_max leq } xs$$
$$\{\ z.\ \mathit{isMax}\ (xs, \leq_{xs})\ z\ \}$$

After *Inv* is restored with Rule Acquire, we obtain l and r such that $qs = l +\!\!+ r$, `count` stores the length of l, and `trace` stores r. We then proceed by a case disjunction on whether or not $|l|$ is lesser than $n-1$.

In the negative case, there is nothing to do.

In the positive case, the `if` branch is taken and we must show a contradiction. Our final punch comes with the observation that the number of roots in the graph represented by the adjacency map `adj` decreases by at most one with each query:

Lemma 5.3. *For any list l of pairs in $[0, \ldots, n-1]^2$, the number of roots in fromEdges l is greater than or equal to $n - |l|$:*

$$|\mathit{roots}\ (\mathit{fromEdges}\ l)| \geq n - |l|$$

A root in *fromEdges* l is a maximal element[7] of xs with respect to the order $\leq_{\mathit{fromEdges}\ l}$. Therefore, if $|l| < n-1$, then $|\mathit{roots}\ (\mathit{fromEdges}\ l)| \geq 2$, so there must be a maximal element w that is different from z.

So far this is not yet a contradiction to the maximality of z, because z is the maximum (and thus unique maximal) of the order $\leq_{xs}$. However, because w and z are distinct maximal elements of *fromEdges* l, none is reachable from the other; that is, they are incomparable with respect to $\leq_{\mathit{fromEdges}\ l}$. Therefore, when we learn that $r = (z, w) :: r'$ (for some r') thanks to the prophecy resolution, we can show that this ordering is taken into account by the *insert* operation in *fromEdges*. Formally, we can show the following chain of inclusions:

$$\begin{aligned}(z, w) \ &\in\ \leq_{\mathit{fromEdges}\ (l +\!\!+ [(z,w)])} \\ &\subseteq\ \leq_{\mathit{fromEdges}\ (l +\!\!+ [(z,w)] +\!\!+ r')} \\ &=\ \leq_{\mathit{fromEdges}\ qs} \\ &\subseteq\ \widehat{\leq}_{\mathit{fromEdges}\ qs} \\ &=\ \leq_{xs}\end{aligned}$$

[7] Recall that a *maximal element* of a partial order $(X, \leq)$ is an element $x \in X$ such that, for every $y \in X$, if $x \leq y$, then $x = y$.

The ordering (z, w) is thus included in $\leq_{xs}$, a contradiction to the maximality of z.

6 Lower Bound of Sorting

We reached the final case study: the proof that, for any comparison-based sorting function `sort` of type

```
sort : ('a -> 'a -> bool) -> 'a list -> 'a list,
```

and for any n, there exists an adversary comparison function that forces `sort` to make $n \log_2 n$ comparison queries (modulo a fixed constant that is independent of n).

In fact, we show that the number of comparisons k is such that $k \geq \log_2 n!$, or, equivalently, that $2^k \geq n!$. The expression $\log_2 n!$ can then be written as $n \log_2 n$ (modulo a constant) using Stirling's formula.

We now follow the usual four steps of our approach.

Step 1 (Specify* `sort`*). This is the specification of `sort`:

$$\begin{aligned}&\mathit{ImplementsSort}\ sort\ \triangleq \\ &\quad \forall xs\ leq\ \leq .\ \mathit{TotalOrder}\ (xs, \leq) \implies \\ &\quad \{\ \mathit{ImplementsOrder}\ (xs, \leq)\ leq\ \} \\ &\qquad sort\ leq\ xs \\ &\quad \{\ zs.\ \mathit{isSorting}\ (xs, \leq)\ zs\ \}\end{aligned}$$

The assertion *isSorting* in the postcondition claims that the output zs of `sort` is a correct sorting zs of an input list xs:

$$\begin{aligned}&\mathit{isSorting}\ (xs, \leq)\ zs\ \triangleq \\ &\quad zs \equiv_{\mathsf{p}} xs \wedge \forall (z_1, z_2) \in \mathit{pairs}\ zs.\ z_1 \leq z_2\end{aligned}$$

The assertion $zs \equiv_{\mathsf{p}} xs$ means zs is a permutation of xs and the term *pairs* zs denotes the list of consecutive pairs of elements of zs.

Step 2 (*`lower_bound_witness`*). We now fix n and the **input list** $xs \triangleq$ `[0; ... ; n-1]`. The idea for an adversary comparison function `leq` that triggers `sort`'s desired lower bound is similar to the one in the previous section (§5): `leq` keeps an adjacency map `adj` representing a partial order on xs that is incrementally updated when `leq` is queried at incomparable elements x and y. The main difference is that, whereas the adversary comparison function of `list_max` would always insert the pair (x, y) in this order, now the adversary comparison function `leq` of `sort` is more careful: instead `leq` decides on the ordering of the pair (x, y) that will reveal *"the least information"* about the order $\leq_{xs}$ that it claims to implement. Intuitively, any total ordering that extends the partial order represented by `adj` is a viable choice for $\leq_{xs}$. Therefore, when deciding between adding either the edge (x, y) or the edge (y, x), `leq` picks the one that allows the greatest number of extensions.

The implementation of such a function `leq` appears in Figure 5. The functions `reach` and `add` are defined exactly like in Figure 4, therefore we omit their definition. The function `pairs` implements *pairs*. The functions `pow` and `fact`

```
let lower_bound_witness sort n =
  let count = ref 0 in
  let adj = Hashtbl.create n in
  let trace = Proph.new () in
  let lock = Mutex.create () in

  let xs = List.init n (fun i -> i) in

  let leq x y = Mutex.protect lock (fun () ->
    incr count;
    if reach adj x y then begin
      Proph.resolve trace ((x, y), true); true
    end else if reach adj y x then begin
      Proph.resolve trace ((x, y), false); false
    end else begin
      let xss = total_extensions adj xs in
      let m = count_compatible (x, y) xss in
      let l = count_compatible (y, x) xss in
      if m >= l then begin
        Proph.resolve trace ((x, y), true);
        add adj x y; true
      end else begin
        Proph.resolve trace ((x, y), false);
        add adj y x; false
      end
    end
  ) in

  let zs = sort leq xs in
  Mutex.lock lock;
  if pow 2 (!count) < fact n then begin
    let xss = total_extensions adj xs in
    List.iter (fun ws ->
      if zs <> ws then
        List.iter (fun (u, v) ->
          Proph.resolve trace ((u, v), true)
        ) (pairs ws)
    ) xss;
    assert false
  end;
  !count
```

Figure 5. Witness of `sort`'s lower bound.

respectively implement exponentiation and factorial operations on integers. The function `total_extensions` computes all permutations *ys* of *xs* that *extend* `adj`, in the sense that, the total ordering induced by the order in which elements occur in *ys* does not contradict the partial order represented by `adj`. The function `count_compatible` receives a pair (x, y) and a list of lists *xss* and counts the number of lists *ys* in *xss* where x occurs before y in *ys*. All these functions are omitted from Figure 5 for brevity, but their definition can be found in the Appendix (§A).

It is through `count_compatible` that `leq` counts both the number of total extensions of the order represented by `adj` that contain the ordering (x, y) and the number of those that contain (y, x), and decides on which edge to add to `adj`.

Like in the previous section, we explain the remaining aspects of the implementation (prophecy variables and the `if`-branch at the end of `lower_bound_witness`) in the following steps.

Step 3 (Formal lower-bound claim). The lower-bound claim is formalised as the following theorem:

Theorem 6.1. *Every correct sorting algorithm calls the comparison function defined in* `lower_bound_witness` *(Figure 5) at least a number of times k such that $2^k \geq n!$:*

$$\begin{array}{l} \forall \mathit{sort}\ n. \\ \{\ \mathit{ImplementsSort}\ \mathit{sort}\ \} \\ \quad \texttt{lower\_bound\_witness}\ \mathit{sort}\ n \\ \{\ k.\ 2^k \geq n!\ \} \end{array}$$

Step 4 (Proof of the lower-bound claim). By studying the code of `lower_bound_witness`, we can see that the prophecy variable `trace` stores a **list of queries and answers** *qs*; that is, *qs* is a list of tuples of the form $((x, y), b)$, with not only the query (x, y) sent to `leq`, but also the Boolean answer b returned by `leq`. This extra information allows us to reuse the mathematical machinery developed in the previous section. Indeed, we can now reconstruct, at the mathematical level, the adjacency map corresponding to the processing of a list of queries and answers *es*. It suffices to flip a query (x, y) in *es* according to the Boolean answer and then feeding them to *fromEdges*:

$$\begin{array}{l} \mathit{flipEdges}\ [] \triangleq [] \\ \mathit{flipEdges}\ (((x, y), b) :: es) \triangleq \\ \quad (\mathit{if}\ b\ \mathit{then}\ (x, y)\ \mathit{else}\ (y, x)) :: \mathit{flipEdges}\ es \\ \mathit{fromFlipEdges}\ es \triangleq \mathit{fromEdges}\ (\mathit{flipEdges}\ es) \end{array}$$

Formal definition of $\leq_{xs}$. We can now define the order $\leq_{xs}$ implemented by `leq` in a similar fashion as in the previous section by taking a total extension of the order $\leq_{\mathit{fromFlipEdges}\ qs}$ (which, on its turn, is defined as the reachability relation in *fromFlipEdges qs*):

$$x \leq_{xs} y \triangleq x \mathrel{\widehat{\leq}}_{\mathit{fromFlipEdges}\ qs} y$$

Let l denote the prefix of queries and answers so far processed by `leq` The heart of the proof is the observation that the best `sort` can achieve with each query sent to `leq` is to halve the number of possible total extensions of (the

reachability relation in) *fromFlipEdges l*. Indeed, with each query (x, y), the set A of total extensions of *fromFlipEdges l* can be partitioned into those that contain the ordering (x, y) and those that do not. One of these partition must have a size that is greater than or equal to half of the size of A, and the implementation of `leq` inserts the ordering of x and y in `adj` that corresponds to the greater partition.

Lock-protected invariant. We can formally express this property as the lock-protected invariant *Inv*:

$$\mathit{Inv} \triangleq \exists l\, r.\ \texttt{count} \mapsto |l| * \texttt{trace} \rightsquigarrow r * qs = l +\!\!+ r * \\ |\mathit{totalExtensions}\ (\mathit{fromFlipEdges}\ l)\ xs| \geq \lceil n!/2^{|l|} \rceil$$

The invariant claims that the entire list of queries and answers *qs* can be split into a prefix l of past query-answer pairs and a suffix r of future query-answer pairs. Moreover, it asserts that `count` keeps the size of l and that the prophecy variable `trace` points to r. Finally, it claims that the number of total extensions[8] of *fromFlipEdges l* is greater than or equal to $\lceil n!/2^{|l|} \rceil$. This claim corresponds to the property that each query can at most halve this set of extensions. Indeed, before any query, there are $n!$ possible total extensions of *fromFlipEdges* [], because *xs* has n distinct elements, so any permutation is a possible total extension of the *discrete* order on *xs*. Therefore, after $|l|$ queries, there will be, at least $\lceil n!/2^{|l|} \rceil$ total extensions left.

Correctness of `leq`. The proof that a query indeed at most halves the set of total extensions follows the *"averaging-principle"* argument (that either a or b is greater than or equal to $(a + b)/2$) we explained. This proof is performed as part of the verification that `leq` is correct and (consequently) that it maintains this invariant:

$$\mathit{ImplementsOrder}\ (xs, \leq_{xs})\ \texttt{leq}$$

Concluding contradiction argument. We can then conclude that the output *zs* of `sort` is a correct sorting of *xs* (w.r.t. $\leq_{xs}$):

$$\{\ \mathit{ImplementsOrder}\ (xs, \leq_{xs})\ \texttt{leq}\ \} \\ \quad \texttt{sort leq}\ xs \\ \{\ zs.\ \mathit{isSorting}\ (xs, \leq_{xs})\ zs\ \}$$

We restore *Inv* via Rule Acquire and obtain l and r such that $qs = l +\!\!+ r$, `count` stores the length of l, `trace` stores r, and

$$|\mathit{totalExtensions}\ (\mathit{fromFlipEdges}\ l)\ xs| \geq \lceil n!/2^{|l|} \rceil.$$

We then proceed by a case disjunction on whether or not $2^{|l|} < n!$.

In the negative case, there is nothing to be done: the output of `lower_bound_witness` satisfies the postcondition. In the positive case, we learn that

$$|\mathit{totalExtensions}\ (\mathit{fromFlipEdges}\ l)\ xs| \geq 2.$$

Hence, there must exist a permutation *ws* that is different from *zs* but which also extends *fromFlipEdges l*. The code inside the `if`-branching will find such a permutation, so, without loss of generality, let *ws* be this permutation.

Because *ws* and *zs* are different, there must exist a pair (u, v) of distinct elements u and v of *xs* such that (u, v) belongs to (the order induced by) *ws* and (v, u) belongs to (the order induced by) *zs* and to $\leq_{xs}$ (because *zs* is sorted w.r.t. $\leq_{xs}$).

With the prophecy resolutions we learn that $r = \mathit{pairs}'\ \ ws +\!\!+ r'$ (for some r') where $\mathit{pairs}'\ \ ws$ contains the pairs of consecutive elements of *ws* together with the Boolean `true`. Because *ws* extends *fromFlipEdges l*, the reachability order in *fromFlipEdges* $(l +\!\!+ (\mathit{pairs}'\ \ ws))$ coincides with (the order induced by) *ws*. We can thus show the following chain of inclusions:

$$\begin{aligned} (u, v) &\in \leq_{\mathit{fromFlipEdges}\ (l +\!\!+ (\mathit{pairs}'\ ws))} \\ &\subseteq \leq_{\mathit{fromFlipEdges}\ (l +\!\!+ (\mathit{pairs}'\ ws) +\!\!+ r')} \\ &= \leq_{\mathit{fromFlipEdges}\ qs} \\ &\subseteq \widehat{\leq}_{\mathit{fromEdges}\ qs} \\ &= \leq_{xs} \end{aligned}$$

We now have $u \leq_{xs} v$ and $v \leq_{xs} u$, therefore we must have $u = v$ by anti-symmetry of $\leq_{xs}$. This is, however, a contradiction to the assumption that u and v are distinct.

7 Rocq Formalisation

We provide a Rocq formalisation of our results as supplemental material [Anonymous authors 2026].

Code translation. We manually translate to HeapLang all `lower_bound_witness` implementations (Figures 1, 4 and 5). HeapLang is Iris's default language. It is a core functional language with imperative features, with syntax and semantics similar to OCaml. Since the code is relatively concise, it is then easy to check that the semantics is preserved.

In the translation of recursive auxiliary library functions (such as the functions on adjacency maps from Figure 4), we introduce an extra integer argument that is decremented with each recursive call. This trick allows us to verify these definitions with respect to Iris's total Hoare triples, which guarantee termination. In this way, we ensure that, if an application of `lower_bound_witness` diverges, then the client code (`max`, `list_max`, or `sort`) is to blame.

Prophecy interface. In the formalisation, we use the interface from de Vilhena et al. [2020] for *typed* prophecy variables, which allow us to avoid the explicit invariant that values of a prophecy variable have a specific shape (such as a pair or a tuple). We gloss over this aspect in the paper.

Proofs. We leveraged automation (in the list-maximum and sorting case studies) through AI (Claude Sonnet 5) as follows. First, we write the main definitions (order-theory

[8] The set of total extensions is computed by the function *totalExtensions*. This is function is the mathematical counterpart of `total_extensions` and has an analogous definition. See the Appendix (§A) for the definition of `total_extensions`.

concepts, the various *Implements* predicates, and the mathematical counterparts of the auxiliary library functions) so that the statements of the lower-bound claims can be formalised and type checked. Second, we write a detailed proof sketch (similar to the explanations in this paper), and let the AI translate it into a formal proof. Finally, we check that the produced proof follows our initial sketch.

8 Related Work

8.1 Prophecy Variables

Abadi and Lamport [1991] introduce prophecy variables to show completeness of their *refinement* definition, whereby a *state machine* S_1 *refines a state machine* S_2 if there exists a map f, called a *refinement map*, from the states of S_1 to the states of S_2 that preserves S_1 transitions and maps initial states to initial states (among other criteria). Their completeness result says that, if two machines S_1 and S_2 behave similarly (in the sense that the *external* behaviours of S_1 are included in the *external* behaviours of S_2), then there exists a refinement map from S_1 to S_2 provided S_1 can be annotated with *auxiliary variables* including prophecy variables. This is similar in spirit to our use of prophecy variables to prove a theorem whose premise is a correctness specification telling only information about the external behaviour of a function.

A typical application of prophecy variables in program logics is to verify concurrent programs, particularly concurrent programs with *helping*, whereby the *linearisation point* of a concurrent operation is executed by a different thread than the one that started this operation. Prophecy variables are useful in such cases because they allow one to obtain at once the trace of all linearisation events with convenient information necessary to carry on the proof (such as, for example, the *thread identifiers* of the threads that executed the linearisation points). An early example of this use of prophecy variables is from Vafeiadis [2008], who verifies the RDCSS data structure [Harris et al. 2002]. It is with the same goal of verifying this structure in Iris that Jung et al. [2020] propose the interface for prophecy variables in separation logic shown in Figure 2. Our suggested view of a prophecy variable as a variable that stores a trace of events is inspired by this interface.

Our use of prophecy variables is inspired by de Vilhena et al. [2020], who use prophecy variables to verify interesting properties of higher-order functions that *spy* on their clients. Indeed, de Vilhena et al. [2020] verify a separation-logic specification of Longley [1999]'s *modulus* function `Mod`. In short, this function allows one to dynamically discover the dependencies between a second-order function `ff` and its first-order function argument `f`; that is, it discovers the points at which `ff` queries `f`. de Vilhena et al. [2020] use prophecy variables to refer to this set of dependencies and to prove a form of the conjunction rule in separation logic.

Another interesting application of prophecy variables, or, more precisely, of prophecy-inspired techniques, is in the verification of Rust programs [Denis et al. 2022; Matsushita et al. 2021; Ogawa et al. 2025]. Matsushita et al. [2021], for instance, introduce a notion of prophecies to refer to the value of a borrow at the end of its lifetime, an idea they apply to translate a Rust program to a *Constrained Hoare Clause*. Denis et al. [2022] build on this idea and propose Creusot, a verification framework for Rust.

8.2 Proofs of Lower Bounds

Proofs of lower bounds of comparison-based algorithms can be found in many textbooks [Cormen et al. 2009; Erickson 2019b]. To the extent of our knowledge, all such textbooks make the assumption that a comparison-based algorithm can be modelled by a *decision tree*, a structure that decides once and for all the order in which comparison queries are made by the algorithm. However, algorithms are often described in terms of stateful operations or random-sampling primitives in which case this assumption falls apart. Instead, we argue in favour of a more general, and also more abstract and simpler, assumption that a sorting algorithm can be modelled as a piece of code (written in a language with no restriction on the programming features it can use) together with a proof of correctness.

The idea of proving a lower-bound claim using an *adversary* comparison function can also be found in algorithms textbooks [Erickson 2019a], but, also in this case, previous works rely on decision trees.

To our knowledge, the first to formalise proofs of lower bounds of comparison-based algorithms is Eberl [2017], who proves, among others facts, the lower bound of sorting. His proof follows the textbook proof from Cormen et al. [2009], and therefore also relies on the simplifying assumption that comparison-based algorithms can be modelled by decision trees.

9 Conclusion and Future Work

We have identified a limitation in the standard approach to prove lower bounds of comparison-based algorithms: that the algorithm must be expressed in a way that fixes the order in which comparison queries are performed.

We have proposed a new approach that overcomes this limitation: instead, a comparison-based algorithm can be expressed in terms of any standard implementation written in a language with common and convenient programming features to formulate algorithms (features such as random sampling, mutable store, and preemptive concurrency).

Our key idea is to translate a lower-bound claim into a program-verification problem, where we verify that an *adversary* comparison function triggers the desired number of comparison claims. The key ingredient in this approach are prophecy variables, which allow us to refer to a *future*

trace of comparison queries. This is, in part, how we bypass decision trees.

We have demonstrated our approach to prove lower bounds of three classes of algorithms: pairwise maximum, list maximum, and sorting. In the future, we would like to explore how this approach scales to proofs of *average-case* lower bounds of *randomised algorithms*.

A downside from our approach is that, by bringing the problem to program-verification territory, a number of mathematical concepts has to be duplicated in code. In the future, we would like to explore how this limitation could be addressed.

References

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. doi:10.1016/0304-3975(91)90224-P

Anonymous authors. 2026. Rocq formalisation. Submitted as *Supplementary Material*.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press. https://mitpress.mit.edu/books/introduction-algorithms

Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy game: verifying a local generic solver in Iris. *Proc. ACM Program. Lang.* 4, POPL (2020), 33:1–33:28. doi:10.1145/3371101

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13478)*, Adrián Riesco and Min Zhang (Eds.). Springer, 90–105. doi:10.1007/978-3-031-17244-1_6

Manuel Eberl. 2017. Lower bound on comparison-based sorting algorithms. *Archive of Formal Proofs* (March 2017). https://isa-afp.org/entries/Comparison_Sort_Lower_Bound.html, Formal proof development.

Jeff Erickson. 2019a. Adversary Arguments. More Algorithms Lecture Notes, *Algorithms*, University of Illinois Urbana-Champaign. Note 13. https://jeffe.cs.illinois.edu/teaching/algorithms/notes/13-adversary.pdf.

Jeff Erickson. 2019b. Algorithms. University of Illinois Urbana-Champaign. https://jeffe.cs.illinois.edu/teaching/algorithms/.

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings (Lecture Notes in Computer Science, Vol. 2508)*, Dahlia Malkhi (Ed.). Springer, 265–279. doi:10.1007/3-540-36108-1_18

C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (1961), 321. doi:10.1145/366622.366644

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. doi:10.1145/3371113

John Longley. 1999. When is a Functional Program Not a Functional Program?. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*, Didier Rémy and Peter Lee (Eds.). ACM, 1–7. doi:10.1145/317636.317775

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 15:1–15:54. doi:10.1145/3462205

Hiromi Ogawa, Taro Sekiyama, and Hiroshi Unno. 2025. Thrust: A Prophecy-Based Refinement Type System for Rust. *Proc. ACM Program. Lang.* 9, PLDI (2025), 2056–2080. doi:10.1145/3729333

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. doi:10.1145/3211968

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. doi:10.1109/LICS.2002.1029817

Ali Sezgin, Serdar Tasiran, and Shaz Qadeer. 2010. Tressa: Claiming the Future. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6217)*, Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani (Eds.). Springer, 25–39. doi:10.1007/978-3-642-15057-9_2

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Technical Report UCAM-CL-TR-726. University of Cambridge, Computer Laboratory. doi:10.48456/tr-726

A Auxiliary functions

```
let find adj x =
  match Hashtbl.find_opt adj x with
  | None -> []
  | Some ys -> ys

let insert adj x y =
  Hashtbl.replace adj y (x :: find adj y)

let rec downset adj x =
  let ys = find adj x in
  let yss = List.map (downset adj) ys in
  x :: List.concat yss

let reach adj x y = List.mem x (downset adj y)
```

Figure 6. Adjacency maps (same as in Figure 4).

```
let pairs xs =
  List.(combine
    (take (length xs - 1) xs) (tl xs))

let rec interleave x ys =
  match ys with
  | [] -> [[x]]
  | y :: ys' ->
    let yss' = interleave x ys' in
    (x :: ys) :: List.(map (cons y) yss'))

let rec permutations xs =
  match xs with
  | [] -> [[]]
  | x :: xs' ->
    let xss' = permutations xs' in
    List.(concat (map (interleave x) (xss')))

let find_ind tgt xs =
  List.find_mapi (fun i x ->
    if x = tgt then Some i else None) xs

let find_ind' tgt xs =
  match find_ind tgt xs with Some i -> i
  | None -> assert false
```

Figure 7. Auxiliary functions on lists.

```
(* Checks that no pair [(x, y)] of [ys] *)
(* _contradicts_ the partial order [leq]: *)
(* [(y, x)] must not be in [leq] if [x <> y]. *)
(* (But it's ok if [(x, y)] is not in [leq].) *)
let agrees leq ys =
  List.for_all (fun (x, y) ->
    x = y || not leq y x) (pairs ys)

let total_extensions adj xs =
  let xss = permutations xs in
  List.filter (agrees (reach adj)) xss

let compatible (x, y) xs =
  find_ind' x xs < find_ind' y xs

let count_compatible (x, y) xss =
  List.(length (filter (compatible (x, y)) xss))
```

Figure 8. Counting and computing total extensions.

```
let rec fact n =
  if n <= 1 then 1 else n * fact (n - 1)

let rec pow base exp =
  if exp = 0 then 1 else
  if exp = 1 then base else
    let k = if mod exp 2 = 0 then 1 else base in
    k * pow (base * base) (exp / 2)
```

Figure 9. Simple arithmetic functions on integers.