Verifying Linearizability with Hindsight

Peter W. O'Hearn
Queen Mary
University of London
ohearn@dcs.qmul.ac.uk

Noam Rinetzky
Queen Mary
University of London
maon@dcs.qmul.ac.uk

Martin T. Vechev
IBM T.J. Watson
Research Center
mtvechev@us.ibm.com

Eran Yahav
IBM T.J. Watson
Research Center
eyahav@us.ibm.com

Greta Yorsh
IBM T.J. Watson
Research Center
gretay@us.ibm.com

ABSTRACT

We present a proof of safety and linearizability of a highly-concurrent optimistic set algorithm. The key step in our proof is the Hindsight Lemma, which allows a thread to infer the existence of a global state in which its operation can be linearized based on limited local atomic observations about the shared state. The Hindsight Lemma allows us to avoid one of the most complex and non-intuitive steps in reasoning about highly concurrent algorithms: considering the linearization point of an operation to be in a different thread than the one executing it.

The Hindsight Lemma assumes that the algorithm maintains certain simple invariants which are resilient to interference, and which can themselves be verified using purely thread-local proofs. As a consequence, the lemma allows us to unlock a perhaps-surprising intuition: a high degree of interference makes non-trivial highly-concurrent algorithms in some cases much easier to verify than less concurrent ones.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming; F.3.1[Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms: Algorithms, Design, Theory, Verification

Keywords: Optimistic concurrency, Linearizability, Wait-Freedom, Hindsight

1. INTRODUCTION

Highly concurrent algorithms are often considered to be amongst the most difficult to design and understand. The fantastic number of possibilities brought about by the high degree of interference translates into complexity for the programmer.

The purpose of this article is to argue that, seemingly paradoxically, such algorithms can be relatively easy to prove. We proceed by example. We give a proof of an optimistic list-based set algorithm based on Heller et. al. [8]. Our proof method in fact covers a number of algorithms, and can even justify potential improvements to them, but we concentrate on this particular algorithm in this paper, for concreteness. The concurrent set is near the leading edge of concurrent programming, and raises stern challenges for proof.

Our basic suggestion is that, contrary to popular belief, interference can be a blessing rather than a curse in formal proof. An optimistic algorithm can be easy to prove because its informal correctness argument does not rely on showing the effects of all subtle interactions between operations in different threads. By design, to be impervious to interference, every thread uses and maintains simple local invariants on the shared state, that hold continuously. The reason for such invariants is that, for efficiency reasons, threads do not access large parts of memory atomically.

Traditionally, reasoning about sequential data abstractions is done using abstraction functions [11]. For example, for a linked list representation of sets, the function takes a list to the set of its data values, forgetting link and order information. One then proves that the list-based implementation of each operation (e.g., `add`, `remove`, and `contains`) simulates the corresponding operation on sets. Our proof follows this standard approach, with a twist.

In the detailed proof steps about code, we do not carry around the abstraction function or a copy of the set represented in global memory. Rather, we verify a collection of thread-local invariants which are sufficient to prove that local observations in threads imply desired properties of the abstraction function: it is as if we decompose the abstraction function. The invariants themselves can be checked using purely thread-local proof methods, that do not use auxiliary state that mentions program points or other information about the states of other threads. Thus, our proposal is that the main source of proof blowup in concurrent programs, the state explosion resulting from simultaneous tracking of properties (e.g., control) in several threads, can be avoided for some highly concurrent algorithms.

Our proof strategy consists of several steps. First, we identify a collection of "integrity" properties concerning the algorithm's data structures and transitions and prove that these properties are invariants of the algorithm. An example data-structure "integrity" invariant is that there is a strictly-sorted linked list from the head to the tail. An example transition "integrity" invariant is that once a node is marked to be deleted its next pointer is never changed. Second, we show that a subset of these integrity properties implies the Hindsight Lemma, a lemma which allows us to conclude that a pointer link encountered in a list traversal was reachable from the head node sometime in the past. Hindsight then allows us to prove the Local Window Lemma, which shows information about a range of data values in the abstract set being represented in memory; this range or window "slides" as the data structure is traversed. Third, we show that each concrete operation simulates its abstract counterpart, but in one crucial case (the `contains` operation) this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODC'10, July 25–28, 2010, Zurich, Switzerland.
Copyright 2010 ACM 978-1-60558-888-9/10/07 ...$10.00.

involves reasoning about the abstraction function applied to a past state (using the Hindsight Lemma). The second step shows that `contains` is *wait-free*, i.e., that every invocation of `contains` is guaranteed to terminate regardless of interference by other threads. The third step shows *linearizability* [10] of the set algorithm.

The linearizability of the wait-free `contains` is the most remarkable, even surprising, part of the set algorithm. It searches a data structure without using *any* synchronization whatsoever, during which time many other threads might be interfering with the data structure. The Hindsight Lemma gives a direct explanation of the reason why this wait-free search works, in terms of a path built from links existing at different moments in time.

Previous proof efforts for concurrent data abstractions typically carry the global abstraction function or the "linearized value" throughout a proof. (See §9.) This works well up to a point, but becomes strained in some advanced algorithms. Particular difficulties arise when a linearization point, the point in time where an operation appears to "take effect", may happen in a different thread than the one which executes the operation itself. The `contains` operation of the optimistic list is just such an example. One way to deal with such cases has been to track a set of pending calls to an operation [4, 10, 21].

While correct in a theoretical sense, we felt that this proof method led to proofs that were removed from the reasons as to why the optimistic algorithms work, and much less direct than our use of the Hindsight Lemma. Is it really necessary to keep track of pending calls? Because optimistic algorithms behave in such a strongly local way, making tiny atomic observations on the store, it seems unfortunate to have to track global information concerning calls in other threads. Stated more technically, this proof technique leads to proofs that are not thread local.

The proofs of the integrity properties can be done in standard sequential Hoare logic (or sequential separation logic [12, 20], to deal simply with the pointer arguments). While the overall proofs for concurrency can be seen as special cases of Owicki-Gries [19] or rely-guarantee [13] logics, there is no need (in this case) to call in dedicated logics of concurrency. (See §8–§10.)

Our reaction has been to strive for a proof that matches the locality inherent in the algorithm as closely as possible. In hindsight, one way to view this work is as technology transfer, from algorithms to proofs. We have studied a specific proof technique, motivated by the special insights in the design of a class of algorithms, in the hope of finding simpler proofs. Of course, the reader will have to be the judge as to whether we have succeeded at all in this.

Proof outline. Our proof is a mixture of formal (machine-checkable) parts done in separation logic, and parts (in Sections 4-6) done in the ordinary informal (but rigorous) language of mathematics. We do not show the formal proofs, but describe the properties they establish, for use in our later mathematical work. (The conversion of the mathematical parts to machine-checkable or even generated proofs by tools is a topic for future work.)

The paper is organized according to the above outline. In the next two sections we describe the challenge example and the invariants that underpin our proof. The presentation of the invariants will be done using English rather than mathematical logic. In a separate technical report we have established that the invariant properties do indeed hold [18], using formal proofs in separation logic, but we avoid such details here. Our intention is to convey the intuitions about the algorithm's behaviour in relatively precise way, without descending into too many details. §4 then describes a mathematical model, which is used to give a statement of the Hindsight Lemma. At this point our reasoning detaches from the particular set algorithm. In §5 we show that the Hindsight Lemma holds, for any execution traces satisfying the invariants from §3, even ones that do not come from the set algorithm. §6 and §7 bring everything together by proving wait-freedom of `contains` and the linearizability of the set operations, respectively.

2. VERIFICATION CHALLENGE

We describe our approach using the optimistic set algorithm shown in Fig. 1. The algorithm is based on the optimistic set algorithm of Heller et al. [8], rewritten to use atomic sections instead of locks. Note that this is a highly-concurrent algorithm: every atomic section accesses a small bounded number of memory locations. To simplify presentation, our variant is *optimistic* [14], but not *lazy* [8]. Thus, it preserves the main difficulty of the proof which concerns us, i.e., proving that every invocation of `contains` is linearizable. We note that in [18], we also use our approach to prove linearizability of a set with a lazy remove procedure.

Set representation. The optimistic set algorithm uses an underlying sorted linked-list of dynamically-allocated objects of type `E`, shown in Fig. 1, which we refer to as *nodes*. Every node has three fields: an integer field `k` storing the key of the node, a field `n` pointing to a successor node, and a boolean field `m` indicating, as we shortly explain, that the node was deleted from the list. When the `m`-field of a node is set, we say that the node is *marked*, otherwise, we say that the node is *unmarked*. Given a memory state σ, we denote the set of allocated nodes in σ by N_σ, and use $u.k_\sigma$, $u.n_\sigma$, and $u.m_\sigma$ to denote respectively the values of the `k`-field, `n`-field, and `m`-field of a node $u \in N_\sigma$. When clear from context, we omit the subscript.

The list has designated sentinel *head* and *tail* nodes. The *head* node is always pointed to by the shared variable `H` and contains the value $-\infty$. The *tail* node is always pointed to by the shared variable `T`, and contains the value ∞. The value $-\infty$ resp. ∞ is smaller resp. greater than any possible value of a key.

For example, in the state depicted in Fig. 3, the set contains the keys 3, 7, and 15, stored in the unmarked nodes between the head node and the tail node. The state also contains removed marked nodes with the values 1, 5, 7, and 10. (The values 5 and 7 appear in several nodes.)

Set procedures. The set defines three procedures: `add`, `remove`, and `contains`. All three procedures use the internal `locate` procedure to traverse the list. The traversal is *optimistic* [14]: it is done without any form of synchronization. As a result, while a thread is traversing the list, other threads might concurrently change the list's structure. The `locate` procedure returns a pair of pointers to nodes. The node pointed to by the local variable `c` is the successor of the node pointed to by the local variable `p` at the time when the `n`-field of the node pointed to by `p` is read.[1] We refer to the nodes returned by `locate` as the *previous node* (pointed to by `p`) and the *current node* (pointed to by `c`).

`remove` and `add` operations may potentially modify the heap inside their *atomic sections*. (The atomic section of `remove` in shown in lines 65–78 and that of `add` in lines 85–98.) Both of the atomic sections begin with the evaluation of the *validation condition* `if (p.n==c && !p.m)`. The validation condition checks for *interference* at the fraction of the shared state containing the previous and current nodes. More specifically, the validation condition *atomically* tests the value of two memory locations in the shared memory: The value of the `n`-field of the previous node and

[1]The assignment `c = p.n` in line `26` usually gives rise to multiple primitive instructions. In our case, because both `p` and `c` are local variables, the primitive instruction of interest is the one in which the `n`-field of the node pointed to by `p` is read. For simplicity, we assume that the assignment `c = p.n` executes atomically.

```
   type E{
     int k;
     bool m;
     E n;
   }

   E H, T;

0  init(){
1    atomic{
2      T = alloc(E);
3      T.m = false;
4      T.k = ∞;
5      T.n = null;
6      H = alloc(E);
7      H.m = false;
8      H.k = −∞;
9      H.n = T
10   }
11 }
```

```
20 E×E locate(int k){
21   E p = H;
22   E c = p.n;

24   while(c.k < k){
25     p = c;
26     c = p.n
27   }

29   return p,c
30 }

40 bool contains(int k){
41   E×E p,c = locate(k);
42   bool b = (c.k==k);
43   return b
44 }
```

```
60 bool remove(int k){
61   bool restart=true, retval;
62   while (restart){
63     E×E p,c = locate(k);

65     atomic{
66       if (p.n==c && !p.m){
67         restart = false;
68         if (c.k==k){

70           c.m = true;


73           p.n = c.n;
74           retval = true
75         }
76         else retval = false
77       }
78     }
79   }
80   return retval
81 }
```

```
80 bool add(int k){
81   bool restart=true, retval;
82   while (restart){
83     E×E p,c = locate(k);

85     atomic{
86       if (p.n==c && !p.m){
87         restart = false;
88         if (c.k!=k){
89           E t = alloc(E);
90           t.m = false;
91           t.k = k;
92           t.n = c;
93           p.n = t;
94           retval = true
95         }
96         else retval = false
97       }
98     }
99   }
100  return retval
101 }
```

Figure 1: A concurrent set implemented using a sorted linked list.

$\{\mathcal{S} = \mathcal{A}\}$	`contains`(k)	$\{ret = (k \in \mathcal{A}) \land \mathcal{S} = \mathcal{A}\}$
$\{\mathcal{S} = \mathcal{A}\}$	`add`(k)	$\{ret = (k \notin \mathcal{A}) \land \mathcal{S} = \mathcal{A} \cup \{k\}\}$
$\{\mathcal{S} = \mathcal{A}\}$	`remove`(k)	$\{ret = (k \in \mathcal{A}) \land \mathcal{S} = \mathcal{A} \setminus \{k\}\}$

Figure 2: Sequential specification of a set. $\mathcal{S} \subset_{fin} \mathbb{Z}$ denotes the contents of the set. ret denotes the return value. $\mathcal{A}$ is a ghost variable implicitly universally quantified outside of each Hoare triple, and k is not modified. The initial set is empty.

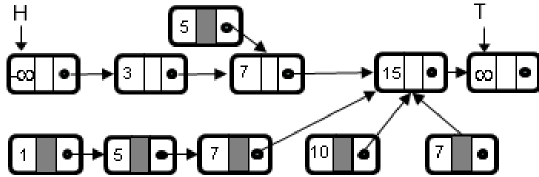

Figure 3: Example (thread-shared) state.

the value of the `m`-field of the current node. The condition `p.n==c` tests that the current node is the successor of the previous node, *as it was at the last time when the primitive instruction* `c = p.n` *was executed during the list traversal.* The condition `!p.m` tests that the node pointed to by `p` is unmarked. If the validation condition evaluates to *false*, indicating a possible interference, the operation *restarts*. Otherwise the remainder of the atomic section is executed:

- A `remove` operation checks if the key in the current node is different than the key that it tries to remove from the set. If it is, the operation returns *false* without modifying the shared state. Otherwise, it marks the current node; links the previous node to the current node's successor; and returns $true$.
- An `add` operation checks if the key of the current node is the same key that it tries to add to the set. If it is, the operation returns *false* without modifying the shared state. Otherwise, it allocates a new node; initializes its fields; places the new node between the previous node and the current node; and returns $true$.

In contrast, a `contains` operation never modifies the shared state, never restarts, and returns whether the key stored in the current node is equal to its input key *without* any form of synchronization.

Verification goal. Our goal is to prove memory safety and linearizability of the implementation of the concurrent set algorithm shown in Fig. 1 with respect to the standard specification of a sequential set given in Fig. 2. The main challenge here is to prove the linearizability of `contains`. Note that a `contains` operation never restarts. Thus, we add proving the wait-freedom of `contains` to our verification goal.

3. INTEGRITY INVARIANTS

In this section, we apply the first step of our approach to the running example: Identifying certain properties as invariants. Here, we describe these properties in an informal fashion. In [18], we formally define these properties and prove that they are invariants of this algorithm using (sequential) separation logic.

The invariants are listed in Fig. 4. The invariants come in two forms, *state* (or *representation*) *invariants* which describe properties of data structures, and *step* (or *transition*) *invariants* which describe properties of single steps in the program's execution. A *state property* φ determines a set of states $[\![\varphi]\!] \subseteq \Sigma$, where Σ is the set of program states, and a *step property* δ determines a set of pairs of states $[\![\delta]\!] \subseteq \Sigma \times \Sigma$.

Besides the obvious classification of the invariants into state and step invariants, we further classify them along two other dimensions: We distinguish between *local* invariants which pertain only to properties of *small fractions* of the shared store (in our case, each fraction contains at most two nodes), and *global* invariants which describe properties of the *whole* shared state. It is the preservation of *all* of the local invariants which allows every thread to behave in a purely local manner and yet guarantee the preservation of the global invariants without the need for atomic access to large parts of the shared state. For example, the preservation of property $\varphi_{<}$, the local sortedness invariant, allows us to establish φ_{ac}, the global invariant guaranteeing that the heap contains no node cycles. On the other hand φ_{ac} together with the local invariants $\varphi_{T\not n}$ and φ_{n}, pertaining to the local link structure of every node, allows us to establish property φ_{rT}, i.e., that the tail node is reachable from every node, as an invariant. The latter property is the key reason for the memory safety of the algorithm.

The third dimension of our classification is the aspect of the implementation they pertain to. We show that different subsets of these properties allow to establish different lemmas in our proofs.

Type	Scope	Aspect	Name	Description
State	Local	Shape	φ_H	H has a non-*null* value (i.e., the head node exists)
			φ_T	T has a non-*null* value (i.e., the tail node exists)
			$\varphi_{T\not{n}}$	The tail node has no successor
			φ_n	Every node other than the tail node has a successor
		Data	φ_∞	The key of the tail node is ∞
			$\varphi_{-\infty}$	The key of the head node is $-\infty$
			$\varphi_<$	The key of every node is smaller than the key of its successor
	Global	Shape	φ_{rT}	The tail node is reachable from every node
			φ_{ac}	There are no cyclic heap paths in the heap
		Data	φ_s	Every list of nodes is sorted.
		Mark	φ_{UB}	A node is unmarked if and only if it is a backbone node
Step	Local	Shape	δ_H	The value of the global variable H never changes
			δ_T	The value of the global variable T never changes
		Data	δ_k	The key of a node does not change
		Mark	δ_m	A marked node does not become unmarked
	Global	Shape	δ_e	An exterior node does not become a backbone node
			δ_{en}	The successor of an exterior node does not change
			δ_{bn}	If the successor of a backbone node changes, the node must remain a backbone node in the following state

Figure 4: Invariants of the concurrent set of Fig. 1 classified according to type, scope and aspect.

This allows us to reuse parts of our proof for different implementations, as long as the subset of properties relevant to the lemma is proven to be an invariant of the new implementation.

The invariants are expected to hold outside atomic sections. Thus, an invariant might be violated in states which occur inside an atomic section. However, memory states which occur inside atomic section are "invisible" to all other threads. (See §4.)

Terminology. We say that a node is a *backbone node* in state σ if it is reachable from the head node in σ. A link $u.n \mapsto v$ in σ is a *backbone link* in σ if u is a backbone node in σ. We refer to a node in σ which is not a backbone node in σ as an *exterior* node in σ and to a link in σ whose source is an exterior node in σ as an *exterior* link in σ. (Note that a link in a state σ is either a backbone link in σ or an exterior link in σ.)

State invariants. The contents of the heap during any execution of the optimistic set algorithm of Fig. 1 can be viewed as a *list of reversed trees*.[2] The list, going from the head node to the tail node, consists of unmarked nodes. Every node in the list, with the exception of the head node, is the root of a reversed tree (potentially containing only the root node) of an unbounded degree in which all non-root nodes are marked. Furthermore, every path in the heap is sorted. This heap structure, guaranteed by the global state invariants φ_{rT}, φ_{ac}, φ_{UB}, and φ_s, is maintained regardless of interference between threads.

Invariants φ_H and φ_T guarantee, respectively, that the global variables H and T always point to allocated nodes. Algorithmically, thanks to invariant φ_H the algorithm does not need to check whether variable H has a *null* value before dereferencing it. Variable T is not used in the algorithm. It is only used to refer to the tail node in the invariants. Invariant φ_T ensures that these invariants are well defined.

The algorithm maintains several local data invariants relating key values to the shape of the heap: invariant $\varphi_{-\infty}$ ensures that the key of the head node is $-\infty$, invariant φ_∞ ensures that the key of the tail node is ∞, and invariant $\varphi_<$ guarantees that if the n-field of a node u points to node v, then the key of u is smaller than that of v.

[2] A reversed tree is a tree where pointers are reversed such that they point towards the root.

Algorithmically, invariant $\varphi_{-\infty}$ ensures that there is no need to look at the value of the key at the head node, invariant φ_∞ ensures that the n-field of the tail node is never dereferenced, and invariant $\varphi_<$ ensures that the head node has no predecessor. The three local data invariants, together with the local shape invariant $\varphi_{T\not{n}}$, ensure that in every state which occurs during the execution, the heap path between the head node and the tail node exists and induces a partitioning of the range $(-\infty, \infty)$. The global invariant φ_{UB} makes it possible to check locally whether a node is in the backbone by testing if it is marked or not. (Recall that the validation condition of `add` and `remove` checks that the previous node is not marked.)[3]

Step invariants. The invariants δ_H, δ_T, and δ_k ensure immutability: once initialized, certain parts of the shared state are never modified. Invariant δ_H ensures that the head node does not change during the execution. Similarly, invariant δ_T, guarantees that traversals reaching the tail of the list always end at the same tail object. Invariant δ_k ensures that the key field is immutable. This invariant ensures that when `locate` returns, the value of the input key of the operation which invoked it is greater than the key of the previous node and smaller or equal to the key of the current node.

Invariants δ_m, δ_e, and δ_{en} ensure *conditional* immutability: once a fraction of the shared state has a certain property, this fraction becomes immutable. Conditional immutability implies that the changes to the shared state are, in a sense, *monotonic*: The local step invariant δ_m says that an unmark node may become marked, but not vice-versa. Together with the global state invariant φ_{UB} it gives the global step invariants δ_e which ensures that an interior node may become an exterior node, but not vice-versa. The global step invariant δ_{en} ensures that the successor of an exterior node does not change. The global step invariant δ_{bn} ensures that only fields of backbone nodes can be mutated.

[3] We note that the algorithm manages to maintain invariant φ_{UB}, and thus avoids the need to check whether the current node is marked, because the `remove` operation is not lazy: The marking of a node and its removal from the list are done in the same atomic section. In the lazy list algorithm a weaker invariant is maintained: every unmarked node is a backbone node, but not necessarily the other way around, an thus a different validation condition is used. See [18].

We state that the properties shown in Fig. 4 are invariants of our challenge problem after presenting the mathematical model. We note that the step invariants together ensure that turning a node from a backbone node to an exterior node also fixes the value of its n-field. In a way, these invariants ensure that every exterior link provides a view of a 1-link fragment of the backbone list as it was at the time just before the source of the link turned from a backbone node to an exterior node. This "frozen view" of the past structure of the list is formally captured by the Hindsight Lemma (see §5) and plays a key role in our proof of linearizability (see §7) .

4. MATHEMATICAL MODEL

In this section we set some notations and formal details regarding our mathematical model. This concerns the structure of memory states $\sigma \in \Sigma$ and executions $\pi \in \Pi$. We concentrate on key features of the model, and formalize it (rather standardly) in [18].

Concurrent Objects. A *concurrent object* defines a set of *procedures* which may be invoked by client threads, potentially concurrently. Procedures may have local variables. Every invocation of a procedure p has its own *private* copy of p's local variables, i.e., the local variables of one invocation cannot be accessed by any other invocation. The invocations share access to the concurrent object's *shared variables* and to dynamically (heap) allocated objects.

We refer to an invocation of a procedure of a concurrent object as an *operation*. We assume $\mathcal{T}$ is an unbounded set of *thread identifiers*. Our semantics is insensitive to the actual values of thread identifiers. Thus, without loss of generality, we assume that a thread may invoke a procedure on the object *at most once*. We identify every operation by the identifier $t \in \mathcal{T}$ of the thread which invoked it and use the terms *thread* and *operation* interchangeably.

Memory States. A *memory state* $\sigma \in \Sigma$ is comprised of a *thread-shared state* and a map from operations, identified using thread identifiers, to *thread-local states*. The thread-shared state records the values of the concurrent object's *shared variables* and contains a *heap* which maps fields from *locations* of allocated objects to their values. A value may be a boolean, an integer, a memory location, or the designated value $null$ which is not the location of any object. The thread-local state of thread t records the values of t's local variables and program counter.

Given a pair of nodes $e = \langle u, v \rangle$, we say that $u = src(e)$ is the *source* of e and $v = trg(e)$ is the *target* of e. A sequence of pairs of nodes $\xi = e_0, \ldots, e_k$ is a *node-pair path* from u to v if $u = src(e_0)$, $v = trg(e_k)$, and $trg(e_i) = src(e_{i+1})$ for every $0 \leq i < |\xi| - 1$.

A *link* in a state σ, denoted by $u.n \overset{\sigma}{\mapsto} v$, is a pair of nodes such that in state σ the n-field of node u points to v, i.e., $u.n_\sigma = v$. (We sometimes omit the subscript when clear from context). If $u.n \overset{\sigma}{\mapsto} v$ is a link in a state σ, we say that node v is the (necessarily unique) *successor* of node u in σ and that node u is a *predecessor* of node v in σ. A sequence ζ of links in state σ is a *heap path from* u *to* v *in* σ if ζ is also a node-pair path from u to v. A node v is *reachable* from node u in σ, if there is a heap path from u to v in σ.

Transitions, Executions and Traces. We assume that the semantics of a concurrent object is given by a transition relation $\overset{tr}{\leadsto}$ that *interleaves* the execution of different threads. A *transition* $\sigma \xrightarrow{\langle t,\kappa \rangle} \sigma' \in \overset{tr}{\leadsto}$ represents the fact that the *source* memory state σ can be transformed into the *target* memory state σ' by thread t executing *computation step* κ. A computation step is either an *invocation* of an operation of the concurrent object (possibly with input parameters), a *response* by an invoked operation (possibly with a return value), or an *atomic action*. An atomic action is either a *primitive instruction* or an *atomic section* containing several primitive instructions. The instructions inside an atomic section are guaranteed to execute without interference from other threads. We refer to the memory states which occur between computation steps as *visible memory states* and to the ones that occur "inside" atomic sections as *invisible memory states*. We use the term *state* as a shorthand for *visible memory state*.

An *execution* $\pi \in \Pi$ is a sequence of transitions with the target state of every transition being the same as the source state of the next transition. A *trace* τ is a sequence of memory states. The *trace of an execution* π, denoted by $trace(\pi)$, is the sequence of states that occur during π, i.e., $trace(\pi) = \sigma_0, \sigma_1, \ldots, \sigma_{|\pi|}, \sigma'_{|\pi|}$ where σ_i is the source memory state of transition $\pi(i)$ and $\sigma'_{|\pi|}$ is the target memory state of the last transition of π. An *execution step* $\xrightarrow{\langle t,\kappa \rangle}$ consists of a thread identifier and a computation step. A *schedule* Φ is a sequence of execution steps. The *schedule of an execution* π, denoted by $\Phi = \Phi(\pi)$, is the sequence of execution steps that occur during π.

An execution π is an *execution of the concurrent object* if (a) every transition belongs to the concurrent object's transition relation; (b) the source state of the first transition is a distinguished *initialized state* of the concurrent object. (The initialized state can be thought of as being produced by applying the *initialization procedure* of the concurrent object to a memory state in which no thread is running and the heap contains no objects); and (c) every operation starts by executing the invocation computation step and once it responds, it does not execute any more computation steps.

Properties and Invariants. We say that a state property φ *holds* in trace τ, denoted by $\tau \models \varphi$, if $\tau(i) \in [\![\varphi]\!] \subseteq \Sigma$ for every $0 \leq i < |\tau|$. We say that a step property δ *holds* in trace τ, denoted by $\tau \models \delta$, if $\langle \tau(i), \tau(i+1) \rangle \in [\![\delta]\!] \subseteq \Sigma \times \Sigma$ for every $0 \leq i < |\tau| - 1$. We use conjunction of properties in the evident way, and we also sometimes say that a property holds in an execution to mean that it holds in the trace of its component states. A property is an *invariant* of a concurrent object if it holds in all of its executions.

For simplicity, we assume that memory is not reclaimed. (See §8.) Thus, an execution is *memory safe* if it never dereferences a $null$-valued pointer. Memory safety can be captured by the state invariant $\Sigma \setminus \{\sigma_{error}\}$ which says that the error state σ_{error} resulting from a $null$-valued pointer dereference never occurs. The properties listed in Fig. 4 are state and step properties according to the above definitions. We can now state the following theorem.

THEOREM 4.1. *Every execution of the set algorithm shown in Fig. 1 is memory safe and satisfies all the state properties and all the step properties shown in Fig. 4.*

5. THE HINDSIGHT LEMMA

In this section, we present the second step of our proof strategy: We show that the Hindsight Lemma and the Local Window Lemma are implied from certain (not all) of the properties shown in Fig. 4.

5.1 Hindsight about Past Links

Informally, the Hindsight Lemma says that in every execution which satisfies the integrity shape properties shown in Fig. 4, if a link $u.n \overset{\sigma}{\mapsto} v$ is reachable from the head node (i.e., is a backbone link) at a state σ, and link $v.n \overset{\sigma'}{\mapsto} w$ with v in common is a link in a later state σ', then there is some state occurring between σ and σ' in which the later link $v.n \mapsto w$ is reachable from head. The key point is that $v.n \overset{\sigma'}{\mapsto} w$ need not be reachable (be "in" the data structure) at time σ': rather, we infer the existence of a prior time at which the link is in the structure.

The Hindsight Lemma is a property of executions more general than our challenge problem: it refers only to the shape properties shown in Fig. 4. In particular, it does not mention data related properties or mark bit related properties. Its insensitivity to these implementation-specific aspects of our running example makes it more widely applicable: If we have *any* execution satisfying the shape properties of Fig. 4, not just an execution coming from the program, then the Hindsight Lemma can be used to infer that certain links were reachable from the head node *in the past*.

DEFINITION 5.1 (SHAPE-LEGAL EXECUTION). *Let the* set's shape-state property *be* $\varphi^s = \varphi_H \wedge \varphi_T \wedge \varphi_n \wedge \varphi_{rT} \wedge \varphi_{ac}$ *and the* set's shape-step property *be* $\delta^s = \delta_H \wedge \delta_T \wedge \delta_e \wedge \delta_{en} \wedge \delta_{bn}$. *An execution* π *is a* shape-legal execution *if* $trace(\pi) \models \varphi^s \wedge \delta^s$.

LEMMA 5.2 (HINDSIGHT). *Let* $\tau = trace(\pi)$ *be the trace of a shape-legal execution* π. *For any states* $\sigma_i = \tau(i)$ *and* $\sigma_k = \tau(k)$ *such that* $0 \leq i \leq k < |\tau|$ *and for any nodes* u, v, w *such that* $u.n \overset{\sigma_i}{\mapsto} v$ *is a backbone link in* σ_i *and* $v.n \overset{\sigma_k}{\mapsto} w$ *is a link in* σ_k, *there exists a state* $\sigma_j = \tau(j)$ *such that* $i \leq j \leq k$ *and* $v.n \overset{\sigma_j}{\mapsto} w$ *is a backbone link in* σ_j.

Proof: The pair of nodes $\langle v, w\rangle$ is a link in state σ_k. If $\langle v, w\rangle$ is a backbone link in σ_k then the lemma holds for $j = k$. Otherwise, $\langle v, w\rangle$ must be an exterior link in σ_k.

Assume that the pair $\langle v, w\rangle$ is an exterior link in σ_k. By definition node v is an exterior node in σ_k.

By the conditions of the lemma, link $u.n \mapsto v$ is a backbone link in state σ_i. Thus, v is a backbone node in σ_i. Recall that we assume that nodes are not reclaimed. Because node v exists in σ_i, it exists in any following memory state. In particular, v exists in any memory state $\tau(h)$ for $i \leq h \leq k$.

Let j be the maximal index in $i \leq j < k$ such that v is a backbone node. Such an index must exist because node v is a backbone node in σ_i and an exterior node in state σ_k. We have already shown that node v exists in any memory state $\tau(j), \ldots, \tau(k)$. Thus, by selection of j it holds that node v is an exterior node in $\tau(j+1)$. Furthermore, by property δ_e it holds that node v is an exterior node in any state $\tau(h)$ for every $j + 1 \leq h \leq k$.

Thus, by δ_{en}, the successor of node v is the same in every state $\tau(h)$ for every $j + 1 \leq h \leq k$. By assumption, the successor of node v in state $\sigma_k = \tau(k)$ is node w. Thus, the successor of node v in state $\tau(j + 1)$ is node w.

Node v is a backbone node in state $\tau(j)$ and an exterior node in state $\tau(j + 1)$. Node w has to be the successor of node v in state $\tau(j)$, for otherwise node v has different successors in $\tau(j)$ and in $\tau(j + 1)$ which by δ_{bn} would mean that v should be a backbone node in state $\tau(j + 1)$.

We have established that w is the successor of v in state σ_j. Node v is a backbone node in σ_j and thus, by definition, link $v.n \mapsto w$ is a backbone link in σ_j. □

Now we need to set up some information which will be used to apply the Hindsight Lemma in the proof of the optimistic traversal. We make these definitions here because the notions apply to all shape-legal executions, and (again) are independent of the algorithm to some extent.

We will show that a sequence of edges traversed by the `locate` procedure comprises a backbone, if we successively look back in time.

DEFINITION 5.3 (TEMPORAL BACKBONES). *A node-pair path* ζ *is a* temporal backbone *in trace* τ *going through a subsequence* τ_t *of* τ *if* $|\tau_t| = |\zeta|$, *the pair of nodes* $\zeta(i)$ *is a link in* $\tau_t(i)$ *for every* $0 < i < |\tau_t|$, *and* $\zeta(0)$ *is a backbone link in* $\tau_t(0)$.

Using the idea of a temporal backbone gives us a way to state an "inductive cousin" of the Hindsight Lemma.

LEMMA 5.4 (BACKBONE LEMMA). *Let* $\tau = trace(\pi)$ *be the trace of a shape-legal execution* π. *Let* ζ *be a temporal backbone going through the subsequence* $\tau(i_0), \ldots, \tau(i_k)$ *of* τ. *There exists a sequence* $i_0 = j_0 < j_1 < \ldots < j_k = i_k$ *such that for every* $0 \leq m \leq k$ *it holds that* $\zeta(m)$ *is a backbone link in* $\tau(j_m)$ *and* $i_0 \leq j_m \leq i_m$.

Proof: By induction on the length of ζ. The base case is immediate and the induction step follows directly from Lemma 5.2. □

5.2 Local Window to Past Data Ranges

When one satisfies the data as well as the shape invariants, it becomes possible to infer properties of the data values that we find to exist by hindsight.

Conceptually, in the context of the set algorithm, the Local Window Lemma allows us to infer that every edge traversed, even though there is interference, identifies a certain time in the past when data values in a particular range were or were not in the set represented by the data structure. In a way, a traversal over a temporal backbone opens a local window on the global abstraction function. This is the key to being able to reason about the linearizability of the set operations in a thread-local way.

Technically, the Local Window Lemma is an immediate consequence of sortedness and the backbone lemma.

DEFINITION 5.5 (DATA-SHAPE-LEGAL EXECUTION). *Let the* set's data-state property *be* $\varphi^d = \varphi_s$ *and the* set's data-step property *be* $\delta^d = \delta_k$. *A* data-shape-legal execution π *is a shape-legal execution such that* $trace(\pi) \models \varphi^d \wedge \delta^d$.

DEFINITION 5.6 (BACKBONE KEYS). *Let* σ *be a memory state such that* $\sigma \models \varphi^s$ *and* $\sigma \models \varphi^d$. *The* set of backbone keys *in state* σ, *denoted by* K_σ, *is comprised of the keys stored in the backbone nodes of* σ.

LEMMA 5.7 (LOCAL WINDOW LEMMA). *Let* $\tau = trace(\pi)$ *be the trace of a data-shape-legal execution* π. *Let* ζ *be a temporal backbone from the* i*th state of trace* τ *to the* j*th state of* τ. *Let* $\langle u, v\rangle$ *be a pair of nodes in* ζ. *There exists a state* $\sigma = \tau(h)$ *for some* $i \leq h \leq j$ *such that* $K_\sigma \cap \{u.k_\sigma, v.k_\sigma\} = \{u.k_\sigma, v.k_\sigma\}$.

Proof: π is a data-shape-legal execution. In particular, in every σ which occurs during π the integrity invariants φ_s, φ_H, φ_n, and φ_{rT} imply that there exists a strictly ascending sorted list between the head node and the tail node which satisfies property $\varphi_<$. This list contains, by definition, all backbone edges. From this and Lemma 5.4, the result follows. □

6. TEMPORAL TRAVERSAL

The hallmark of the optimistic set algorithm is the wait-free list traversal which is performed without any synchronization. In this section we provide a technical explanation for the reason `locate` works despite the fact that it does not use any form of synchronization during its traversal: we show that the optimistic list's spatial traversal goes over a temporal heap path which is "consecutive" in different times. Specifically, it traverses a temporal backbone. This leads to the wait-freedom of `contains`.

From this point on, our proofs become algorithm-dependent again. However, the results of this section only depends on the code of `locate` and the results established in §5 based on the set's shape and data properties. Thus, they can be reused to prove other implementations which satisfy the set's shape and data properties and use a similar `contains` procedure.

DEFINITION 6.1 (LINK CROSSING). *A transition* $\sigma \xrightarrow{\langle t,\kappa \rangle} \sigma'$ *of a shape-data-legal execution* π *is a* link crossing transition *if the executed computation step* κ *corresponds to instruction* c=p.n. *We refer to the source state* σ *of a link crossing transition as its* crossing state *and to the link comprised of the node pointed to by* p *and its successor in* σ *as the* crossed link *in* σ.

Note that crossing a link $u.n \mapsto v$ in one iteration of the traversal loop and then crossing the link $v.n \mapsto w$ in the following iteration, does not necessarily mean that there exists a state in which both $\langle u, v\rangle$ and $\langle u, v\rangle$ are backbone links or even mere links.

We first show that the last call to `locate` made by either an `add`, a `remove`, or a `contains` operation traverses over a temporal backbone.

DEFINITION 6.2 (TRAVERSAL TRAIL TAILS). *Let* π *be an execution of the set algorithm shown in* $Fig.$ 1. *The* traversal trail tail *of thread t in* π*, denote by* $\overleftrightarrow{\tau}_\pi(t)$*, is the suffix of the sequence of crossing states coming from transitions made by t in* π *which starts at the last crossing state in which t crossed a link emanating from the head node.*

LEMMA 6.3 (TEMPORAL TRAVERSAL). *Let* π *be an execution of the running example. The sequence of links crossed in the traversal trail tail of every operation t invoked in* π *is a temporal backbone in* $trace(\pi)$ *which goes through* $\overleftrightarrow{\tau}_\pi(t)$.

Proof: A traversal trail tail starts with the crossing state of a transition in which a link emendating from the node pointed to by p is crossed. Local variable p was set to point to the head node at the previous instruction performed by t. Because the head node never changes (δ_H) and because no thread can modify the values of the local variables of another thread, the local variable p of thread t also points to the head node at the crossing state in which the instruction `c = p.n` in line 22 is executed by t. Thus, the first link traversed is a backbone link and in the resulting state the local variable c points to the target of the crossed link. By induction, every time line 26 is executed and a link is crossed, its source must be the target of the previous link in the traversal trail tail because only local variables of the operation executed by thread t are used. □

The following theorem says that every invocation of `contains` finishes in a finite number of steps. This establishes that `contains` is wait-free (see, e.g., [9]).

THEOREM 6.4 (WAIT-FREEDOM OF CONTAINS). *The* `contains` *procedure shown in Fig. 1 is wait free.*

Proof: From Theorem 4.1 we get that every execution of the concurrent set algorithm shown in Fig. 1 is a memory-safe shape-data-legal execution. The only loop in the `contains` procedure is in the `locate` procedure. Thus, from invariant φ_{ac}, a `contains` operation never traverses over a link emanating from the head node more than once. From Lemma 6.3, we get that the sequence of traversed links in the traversal trail tail of `contains` induces a temporal backbone. In particular, every time `locate` crosses a link $u.n \mapsto v$, the key of v is strictly greater than that of u. Thus at every iteration the key of the node pointed to by c is greater than the key of the node which was pointed by c at the previous iteration. The traversal ends when a node with a key greater or equal to the value of the input key parameter k is reached. This input parameter k does no change its value throughout the execution of the `contains` procedure. The standard order on integers is a locally finite total order. Thus the number of iterations that `locate` can perform before c points to a node with a key equal or greater to the value of k is finite. Wait freedom follows. □

7. LINEARIZABILITY BY HINDSIGHT

In this section, we describe the notion of linearizability and prove that the concurrent set algorithm shown in Fig. 1 is linearizable with respect to the sequential specification in Fig. 2.

7.1 Linearizability

Linearizability is a property of the externally-observable behavior of concurrent objects [9, 10]. Intuitively, an execution of a concurrent object is linearizable with respect to a given sequential specification if each invoked operation seems to take effect instantaneously at some unique point in time between its invocation and response, and the resulting sequence of (seemingly instantaneous) operations respects the given specification. More technically, linearizability is defined using the notion of *histories*, which we describe below using the terminology of §4.

DEFINITION 7.1 (HISTORIES). *A* history H *is a sequence of invocation and response execution steps.* H *is* sequential *if the execution step immediately preceding each response execution step is its matching invocation.* H *is* well formed *if for all threads t, the sequence of operations in* H *performed by t in* H *is sequential.* H *is* complete *if it is well-formed and every invocation has a matching response.* H *is a* completion *of a well-formed history* H' *if* H *is complete and can be obtained from* H' *by (possibly) extending* H' *with some response execution steps and (possibly) removing some invocation execution steps. The* history of an execution π*, denoted by* $H(\pi)$*, is the maximal subsequence of invocation and response execution steps in* π*. A* sequential specification $\mathfrak{H}$ *is a prefix-closed set of well-formed sequential histories.*

Intuitively, in a sequential history all operations seem to run atomically. The well-formedness captures two properties of histories. Firstly, it ensures that all the responses should have corresponding invocations. Secondly, it formalizes the intuition that each thread, if it is considered in isolation, is a sequential program. A completion of H capture the idea that some operation in H which have not responded have "taken their effect" while others have not.

DEFINITION 7.2 (LINEARIZABILITY). *A history* H *is* linearizable *with respect to a well-formed sequential history* H_S *if there exists a completion* H *of* $H(\pi)$*, a history* $H_S \in \mathfrak{H}$*, and a bijection* $b : \{0, \ldots, |H| - 1\} \to \{0, \ldots, |H_S| - 1\}$ *such that (1) for every* $0 \le i < |H|$ *it holds that* $H(i) = H_S(b(i))$ *and (2) for every* $0 \le i < j < |\pi|$ *if* $H(i)$ *is an operation response and* $H(j)$ *is an operation invocation then* $b(i) < b(j)$*. A concurrent object is* linearizable *with respect to a given sequential specification* $\mathfrak{H}$ *if for every execution* π *of the concurrent object there exists a history* $H_S \in \mathfrak{H}$ *such that* $H(\pi)$ *is linearizable with respect to* H_S.

Intuitively, a concurrent object is linearizable with respect to a sequential specification $\mathfrak{H}$ according to Def. 7.2 if for every execution π of the concurrent object, the interaction between the threads and the concurrent object in π can be "explained" by a sequential history H_S in $\mathfrak{H}$: In H_S, every thread performs the same sequence of invocations and responses as in (a completion of) $H(\pi)$ and the real time precedence order between non overlapping operations in π is preserved in H_S.

7.2 Proving Linearizability with Hindsight

Theorem 7.5 proves the linearizability of the set algorithm shown in Fig. 1 with respect to the set's sequential specification shown in Fig. 2. The proof uses the *abstraction function* [11] *Abs* which maps every state σ to its *abstract value*: the set of integer keys stored in σ's backbone nodes, i.e.,

$$Abs(\sigma) = K_\sigma \setminus \{-\infty, \infty\} .$$

DEFINITION 7.3 (MARK-DATA-SHAPE-LEGAL EXECUTIONS). *A* mark-data-shape-legal *execution π is a* data-shape-legal execution *such that $trace(\pi) \models \varphi_{UB} \wedge \delta_m$.*

LEMMA 7.4. *Every execution π of the set algorithm shown in Fig. 1 is a mark-data-shape-legal execution. In every state σ which occurs during π the abstraction function Abs is well defined and $Abs(\sigma) = \{u.k_\sigma \mid u \in N_\sigma \wedge \neg u.m_\sigma\} \setminus \{-\infty, \infty\}$.*

Proof: Follows from property φ_{UB}, Theorem 4.1 and Def. 7.3. □

THEOREM 7.5. *Every execution π of the set algorithm shown in Fig. 1 is linearizable with respect to the set's sequential specification shown in Fig. 2.*

Proof: The proof is done in two stages. Firstly, we construct a sequential history H_S and show that $H(\pi)$ is linearizable with respect to H_S. Secondly, we show that H_S respects the sequential specification of the set.

We begin by constructing a completion of $H(\pi)$. Let H' be a history obtained from $H(\pi)$ by first adding a matching response execution step to every `add` or `remove` operation t which executed its atomic section in π but has not responded. The return value of the response added to the invocation of an operation t is the value of t's local variable `retval` at the state which arises right after the execution of t's atomic section. Let H be the history obtained by removing from H' all remaining invocation execution steps which do not have a matching response. By construction, H is a completion of $H(\pi)$.

We now construct a complete sequential history H_S which contains the same execution steps as H using the auxiliary function ι, defined below. We first define ι and then describe its use in the construction of H_S.

Let $\tau = trace(\pi)$ be the trace of π. The auxiliary function ι associates every operation invocation t launched in π with (the index of) a state of τ. (Recall that in §4, we defined operation t to be the single procedure invocation made by thread t. Thus, ι has type $\mathcal{T} \hookrightarrow \{0, \ldots, |\tau| - 1\}$ in the notation of §4). ι is defined as follows: An `add` resp. `remove` operation t is associated with the source state of the transition in which t executes its atomic section. A `contains` operation t is associated with the last state occurring at or before the last crossing state in the traversal trail tail of t in which the (last) crossed link is a backbone link. Such a state must exist by Lemma 5.4 and Lemma 6.3.

H_S is obtained by reordering the operations according to the partial order induced by ι while ensuring that if multiple operations are mapped to $\iota(t)$ and $\tau(\iota(t))$ is the source state of a transition executed by t, then t is placed after all the other operations t' such that $\iota(t) = \iota(t')$; note that there is exactly one operation which executes the $\iota(t)$-th transition, so if t' is a different operation then t it doesn't execute the $\iota(t)$-th transition.

Every operation starts with an invocation and does not execute any step after it responds. Thus, by construction of ι, if the response of t_1 precedes the invocation of t_2 in H then $\iota(t_1) < \iota(t_2)$, and thus t_1 also precedes t_2 in H_S. By construction of H_S, it contains the same execution steps as H' which is a completion of $H(\pi)$. It follows that $H(\pi)$ is linearizable with respect to H_S.

Having given the construction of H_S, all that remains to be shown is that H_S respects the specification of the set. Intuitively, we show that, based on the idea behind the Hindsight Lemma, we have defined H_S in a way that puts a `contains` before any other kinds of operations invoked on the same abstract state.

The proof continues by induction on the number of operations in H_S. The induction actually proves more than what is formally required: It shows that for every operation t in H_S it holds that (a) the set $Abs(\tau(\iota(t)))$ is the set produced according to the specification in Fig. 2 by applying the sequence of operations in H_S preceding t starting from the empty set, and (b) the return value of operation t and its effect on $Abs(\tau(\iota(t)))$, the abstract value of the $\tau(\iota(t))$-state of τ, are in concert with the set's sequential specification.

Technically, (a) and (b) are proven by a simultaneous induction, as we spell out in the long version. Here, because (a) is comparatively easy, we only sketch why it holds.

For the base case, we assume that H_S has no operations. By definition, the abstract value of the initial state is the empty set.

For the induction case, we assume that (a) and (b) hold for the first k operations in H_S and show that they also hold for the first $k+1$ operations. We denote by t' and t the identifiers of the k-th and $k+1$-th operation in H_S, respectively.

To establish (a) we observe that by construction of H_S either $\iota(t') < \iota(t)$ or $\iota(t') = \iota(t)$.

If $\iota(t') < \iota(t)$ then we observe that (1) the abstraction function is defined in every state of τ (Lemma 7.4) and (2) the abstract value of the source state and of the target of a transition may differ only if the computation step of the transition is the atomic section of `add` or of `remove`. (No other action changes the shared state, and the value of Abs depends only on the contents of the shared state.) However, by construction of ι, there can be no such transition in π between the $\iota(t')$-th state and the $\iota(t)$-th state. Thus, $Abs(\tau(\iota(t') + 1)) = \ldots = Abs(\tau(\iota(t)))$, and hence (a) follows from the induction assumption.

If $\iota(t') = \iota(t)$ then $Abs(\tau(\iota(t'))) = Abs(\tau(\iota(t)))$. Furthermore, by definition of ι, operation t' is necessarily a `contains` operation. According to the sequential specification, a `contains` operation does not change the abstract state of the set. Thus, (a) follows from the induction assumption.

At this point, we have established that (a) holds for the first $k+1$ operations of H_S. We use this fact to establish that (b) holds for the $k+1$-operation. This, together with the induction assumption, establishes that (b) also holds for the first $k+1$ operations of H_S.

First, we note that Theorem 4.1 and property δ_k ensure that in any operation t and regardless of any possible interference the following holds after `locate` returns: The local variable `p` of operation t points to a node u with a key smaller then k and the local variable `c` of operation t points to a node v with a key greater or equal to k, where k is the value of the input parameter of t.

The proof continues by case analysis on the kind of operation t.

If t is a `contains` operation then using Lemma 7.4 and following the same reasoning as in Lemma 5.7 (the Local Window Lemma), we get that in state $\sigma = \tau(\iota(t))$ the set of keys stored in the backbone nodes contained u's key and v's key, but nothing in between, i.e., $K_\sigma \cap \{u.k_\sigma, \ldots, v.k_\sigma\} = \{u.k_\sigma, v.k_\sigma\}$. Recall that $\tau(\iota(t))$ was selected to be a state in which the pair of nodes $\langle u, v\rangle$ is a backbone link. Thus, by the definition of Abs, we get that $Abs(\sigma) \cap \{u.k_\sigma, \ldots, v.k_\sigma\} = \{u.k_\sigma, v.k_\sigma\} \setminus \{-\infty, \infty\}$. We have established that $u.k_\sigma < k \leq v.k_\sigma$ and by assumption, $-\infty < k < \infty$. Thus, $k \in \{u.k_\sigma, v.k_\sigma\}$ if and only if $k \in \{u.k_\sigma, v.k_\sigma\} \setminus \{-\infty, \infty\}$ if and only if $k = v.k_\sigma$. Therefore, `contains` returns the required value. Also, `contains` does not change the shared state, and thus, as required, the abstract value does not change either. (Here, we use the fact that the ι-based construction of H_S has, in a sense, pushed t back before any `add` or `remove` operation which is associated with the same state.)

If t is an `add` or a `remove` operation, then the validation condition `if (p.n==c && !p.m)` ensures that the pair of nodes $\langle u, v\rangle$ is a link in state $\sigma = \tau(\iota(t))$ and that u is unmarked. (See Fig. 1, lines `66` and `86`, respectively.) From property φ_{UB} we get that u is a backbone node and thus, by definition, that $\langle u, v\rangle$ is a

backbone link in σ.[3] Following the same reasoning as in the case of `contains`, we get that $k \in Abs(\sigma)$ if and only if $k = v.k_{\sigma}$. From this, the set's state properties, and the definition of the representation function, it is easy to check that the return value is according to the specification and that if there is a local mutation to the link structure then the resulting state has the required abstract value. □

8. LIMITATIONS

The first step of our proof strategy is the identification of integrity properties. These properties are used to formally describe the set of traces to which the Hindsight Lemma applies. We currently have no algorithmic approach to identify these properties, and rely on the human to provide them.

On the bright side, the Hindsight lemma is not specific to our running example algorithm and is more widely applicable: it is applicable to several linked-list-based set implementations. While this is a restricted family of algorithms, it is still an interesting one. Furthermore, the Hindsight Lemma and the Local Window Lemma can be used in the proofs of other algorithms as long as these algorithms satisfy the shape and the shape and data properties, shown in Fig. 4, respectively. The question of the formulation and existence of Hindsight-like lemmas for other concurrent algorithms, or even all linked-list-based implementations of concurrent data structures, is an intriguing question, which we leave open. This could be an important question because (we believe) the Hindsight Lemma provides the main insight for the correctness of the lazy set algorithm.

A technical limitation is that our atomic sections cover accesses to more than one memory location. This allows us to use invariants that are simpler than would otherwise be so, because we can ignore intermediate states which would break some invariants. One consequence in this paper is that the algorithm is optimistic but not lazy. But at the expense of a more complex proof we can verify a lazy algorithm, as done in the technical report [18]. A more fundamental consequence is that our current formal treatment of memory allocation requires that the allocation is done inside atomic sections. The reason is that, to maintain one shared invariant (or a conjunction of assertions) describing all relevant heap storage, we need to be able to allocate a node and connect it to the data structure all in one go.

The proofs of the integrity properties we have done in [18] can be seen as taking place in Owicki-Gries logic [19], where the need for interference checking is essentially eliminated due to the thread-local nature of assertions. We used separation logic merely to ease the sequential proof steps for reasoning about the heap in Owicki-Gries logic. This approach works well when essentially all the heap-allocated state of the algorithm is shared, but leads to the limitation regarding the need to "allocate and connect in one go". We believe that this limitation could be removed by replacing the Owicki-Gries logic with one of the versions of separation logic that distinguish shared and local state [6, 17, 24], though we need rather less than the full power that these logics provide (we need transfer from local to shared state, but not conversely). We make these remarks because we are interested in knowing the *minimum* technical machinery that is needed to do simple proofs (with, among other things, an eye towards automation).

Our assumption that memory is not reclaimed simplifies the formalization of the shape invariants. However, in the price of some complication, we can allow automatic memory reclamation.

Finally, our proof that the integrity properties are invariants of the running example is formal: the work in the technical report can be mechanically checked. However, our proofs of the Hindsight Lemma and the Local Window Lemma are rigorous mathematical proofs, but we have not mechanically checked them.

9. DISCUSSION AND RELATED WORK

The proof strategy we use in this paper builds on prior work on data refinement. We have already mentioned the early work of Hoare [11]. Lamport added to this the idea of a continuously-defined abstraction function, for use in concurrent contexts [15]. Our abstraction function is continuously defined, and this seems to fit well with the intuitions surrounding optimistic algorithms, but (curiously) we did not need this fact in our proof: essentially, we only need that the abstraction function is defined at the "linearization points" of effectful operations and that its value does not change between these points. When proving refinement (linearizability), we do not always check the value of the abstraction function at a particular post-state of an operation, but sometimes look for (certain properties of) its value in the near past.

Our work also builds on the fundamental paper of Herlihy and Wing [10] which, besides defining the notion of linearizability, introduces a proof method based on abstraction functions. There it is suggested to use an abstraction function which computes the set of all possible abstract (linearized) values, and to track pending method calls using auxiliary (global) state. In contrast, we maintain only a single abstract value, which is consistent with the externally observable behavior of all effectful operations, and rely on the Hindsight Lemma to provide a (non-constructive) evidence for the linearizability of the effectless operations.

The PhD thesis of Vafeiadis [21] gives a quite detailed proof sketch for a variant of the algorithm considered here. He takes from Herlihy and Wing the idea of tracking pending method calls of the `contains` operation in all threads, but rejects their use of a set of abstract values rather than a single value. Apart from this treatment of `contains`, Vafeiadis's proof is completely thread modular. His invariants are similar to ours, though he carries out the proofs in a special rely-guarantee logic. We speculate that the Hindsight Lemma could be used in concert with Vafeiadis's proof techniques to remove his dependence on pending calls, and to obtain a thread-modular and simpler proof.

Colvin et al. [4] provide a machine-checked proof of linearizability for the lazy set algorithm. Their proof is based on a backward simulation instead of linearization points. It also uses the pending method calls idea in a proof, and our relation to them is in this respect similar to our relation to Vafeiadis.

Most of the existing methods for automatic verification of linearizability are based on the linearization points proof method where the linearization points are either user-specified [1, 2, 22] or automatically inferred [23]. Linearization points induce an order between overlapping operations of a concurrent execution which can be exploited in the verification tool to effectively verify a simulation relation between the implementation and its specification. When the linearization point of an operation is not in the same thread which executes the operation, existing automatic tools fail to establish this simulation. Model-checking based tools for checking linearizability [3, 5, 7, 16, 25, 26] do not require linearization points. However, they can only show that an implementation is *not* linearizable and cannot, in general, prove that it is. As far as we know, only [4] has been able to *semi*automatically verify the linearizability of the latter.

10. CONCLUSIONS

Proving correctness of shared memory concurrent programs is a problem whose importance increases with recent advances in processors architecture. One of the most challenging aspects of this problem is the need to reason about interference between concurrently executing threads. In this paper, we make a rather sur-

prising observation that for a certain class of algorithms —highly-concurrent optimistic algorithms—the high degree of concurrency leads to the use of certain idiomatic design principles of fine-grained synchronization which, once formally captured, can lead to rather simple and elegant proofs that are thread-modular.

The classic work of Owicki and Gries [19] identified the importance of auxiliary state in reasoning about programs, and gave simple programs that cannot be proven in a thread-modular fashion. One such example is the parallel composition of an atomic increment instruction with itself: without auxiliary variables, one cannot prove that this program increments by two, but with auxiliary variables that track program points in the two threads the proof can be done. This limitation of purely thread-modular proofs (which don't use this form of auxiliary state) carries over to subsequent works including rely-guarantee and concurrent separation logic [13, 17].

Given this fundamental limitation of thread-modular reasoning for even trivial programs, we were surprised to be able to find thread-modular proofs for some comparatively complicated concurrent programs. An important point, though, is that proving the correctness of the internals of a data abstraction is different from proving a particular client: it is not difficult to construct a particular concurrent client of the set and a particular property which requires auxiliary state to prove, just as in the Owicki-Gries examples.

The Hindsight Lemma allows us to avoid one of the most complex and puzzling steps in reasoning about highly concurrent algorithms: finding linearization points in other threads than the one containing an operation to be linearized. The lemma expresses the reason why wait-free search procedures, which use no synchronization while traversing a list, can work: a kind of backbone through the data structure can be created by selecting links at different points in time. Using this property, together with the sortedness invariant, the Local Window Lemma then explains why there is no mathematical need to take a snapshot of the entire memory when reasoning about the correctness of a search operation, and this fits well with the intuitions underlying the algorithm.

We illustrated our techniques on an optimistic list-based set algorithm based on Heller et. al. [8], but we have been able to prove linearizability and wait freedom of procedures for additional algorithms using the Hindsight Lemma. In [18], we prove the concurrent set algorithm from [25], whose effectless invocations of the `add` and `remove` procedures use less synchronization. (The optimization is based on the intuition that when these procedures return *false* they essentially act like `contains`.) In addition, in [18] we prove also a lazy set algorithm, based on [8], in which the marking (logical delete) and the pointer surgery removing the node from the data structure are done in different atomic steps.

As to future work, we are particularly interested in exploring more general forms of the Hindsight Lemma. We also hope that the results in this paper can be exploited in the development of automated tools for verifying linearizability which will only need to establish invariants of the kind described in this paper.

Acknowledgements. We are grateful to Richard Bornat, Viktor Vafeiadis, and Hongseok Yang for discussions on this and related work. The London authors acknowledge the support of the EPSRC. O'Hearn acknowledges the support of a Royal Society Wolfson Research Merit Award.

11. REFERENCES

[1] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, pages 477–490, 2007.

[2] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, pages 399–413, 2008.

[3] S. Burckhardt, C. Dern, M. Musuvathi, , and R. Tan. Line-up: A complete and automatic linearizability checker. In *PLDI*, 2010. to appear.

[4] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, pages 475–488, 2006.

[5] T. Elmas, S. Tasiran, and S. Qadeer. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI*, pages 27–37, 2005.

[6] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007.

[7] C. Flanagan. Verifying commit-atomicity using model-checking. In *SPIN*, pages 252–266, 2004.

[8] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.

[9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[10] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[11] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271 – 281, 1972.

[12] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[13] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[14] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *TODS*, 6(2):213–226, 1981.

[15] L. Lamport. Specifying concurrent program modules. *TOPLAS*, 5(2):190–222, 1983.

[16] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *FM*, pages 321–337, 2009.

[17] P. W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 375(1-3):271–307, 2007.

[18] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. Tech. rep., Queen Mary University of London, 2010. Available at "*http://www.dcs.qmul.ac.uk/∼maon/pubs/hindsigt-tr.pdf*".

[19] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

[20] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[21] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, Computer Laboratory, 2008. Also available as technical report UCAM-CL-TR-726.

[22] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, pages 335–348, 2009.

[23] V. Vafeiadis. Automatically proving linearizability. In *CAV*, 2010. to appear.

[24] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[25] M. T. Vechev and E. Yahav. Deriving fine-grained concurrent linearizable objects. In *PLDI*, pages 125–135, 2008.

[26] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.*, 17(1-2):164–182, 1993.