

# Shape-value Abstraction for Verifying Linearizability

Viktor Vafeiadis

Microsoft Research Cambridge

**Abstract.** We present a novel abstraction for heap-allocated data structures that keeps track of both their shape and their contents. By combining our abstraction with thread-local analysis and rely-guarantee reasoning, we are able to verify a collection of fine-grained blocking and non-blocking concurrent algorithms for an arbitrary (unbounded) number of threads. We prove that these algorithms are linearizable, namely equivalent (modulo termination) to their sequential counterparts. Our prover is one to four orders of magnitude faster than the other known approaches (which currently handle only a fixed number of threads).

## 1 Introduction

Linearizability [1] is the standard correctness criterion for high-performance libraries of concurrent data structures, such as `java.util.concurrent` and Intel’s TBB (thread building blocks). Linearizability is a safety property. Informally, a library is *linearizable* if calling any of its exported operations appears to execute atomically at some instant between its invocation and its return. This instant when the entire observable effect of a method is deemed to occur is known as the *linearization point*. Equivalently, a concurrent library is linearizable if every concurrent execution consisting of calls to its exported operations is equivalent to a sequential execution that preserves the order of non-overlapping operations. As a result, a linearizable library can be fully specified by its sequential interface; any interesting concurrency is hidden inside the library.

One can easily achieve this atomicity with global lock, but concurrency experts use multiple fine-grained locks and non-blocking instructions, such as compare and swap (CAS), to get better performance and scalability. Unfortunately, even these experts make mistakes, and it is not unusual for published concurrent algorithms to have subtle errors. Our aim is to provide automated verification tools to these experts so that they can formally verify the correctness of their algorithms.

The literature contains several hand-crafted linearizability proofs [2–6], but until recently nobody had automated the derivation of such proofs. Amit et al. [7] used shape analysis to verify linearizability for a fixed (small) number of threads. Unfortunately, this analysis is very naïve regarding concurrency and does not scale to larger number of threads. In a recent technical report, Manevich et al. [8] improved its asymptotic complexity so that it works for a larger (fixed)

number of threads. Both of these works are based on reachability. They require a specialized abstract domain, do not handle memory deallocation, and do not prove that the linearization point occurred exactly once for each method call.

In contrast, we deal with an arbitrary (unbounded) number of threads; we check that the specified linearization points are sound, and we allow complex linearization points that occur in a different thread than the one being verified. In addition, our implementation can reason about memory deallocation, which affects linearizability in subtle ways (see §2), and (absence of) memory leaks.

*Implementation.* Our prototype implementation is based on RGSep [9], a program logic that combines rely/guarantee [10] and separation logic [11]. Our tool and examples are at <http://www.cl.cam.ac.uk/users/vv216/smallfootRG/>. Readers are encouraged to use the tool and to contribute their own examples. We aim to collect further examples that can serve as benchmark suite for concurrency verification tools.

*Main results.* The contributions of this paper are summarised below:

- We describe a generic proof technique for verifying linearizability and a suitable shape analysis. Our proof technique works even for cases where the linearization point is not statically determined. (see §3 and §4).
- We replace the complex RGSep atomic proof rule with two rules, thereby simplifying the presentation and enabling concise actions specifications for operations such as CAS (see §5).
- Our prover works for an unbounded number of threads, is publically available, and is 20–10,000 times faster than the other known tools (see §6).

*Limitations.* (1) We assume a sequentially consistent memory model; this means that parallel composition can be understood as trace interleaving. (2) The program must be accurately analysable by (sequential) shape analysis; this currently restricts our analysis to programs operating on linked lists. (3) The programmer must annotate the locations of the linearization points. (4) The programmer must describe the interference imposed by the module.

## 2 A Simple Example: Treiber’s Stack

Fig. 1 contains C-like pseudocode for Treiber’s stack. This is one of the simplest non-blocking concurrent algorithms. It represents the stack as a singly linked list rooted at  $S \rightarrow \text{Top}$  and uses CAS (compare and swap) to update the stack. CAS is a primitive operation that reads a word from a memory address and conditionally writes to the same address in one atomic step. In particular,  $\text{CAS}(\&S \rightarrow \text{Top}, t, x)$  atomically compares the value of  $S \rightarrow \text{Top}$  with the value of  $t$  and if the two match, the CAS succeeds: it stores the value of  $x$  in  $S \rightarrow \text{Top}$  and returns 1. Otherwise, the CAS fails: it returns 0 and does not change the value of  $S \rightarrow \text{Top}$ .

This algorithm leaks memory: we cannot free popped nodes because of the following scenario. Assume the stack  $S$  initially consists of the nodes  $\alpha$  and  $\beta$ .

```

struct node {
    struct node *next;
    value_t data;
};

struct stack {
    struct node *Top;
};

struct stack *S;

void init() {
    S = alloc();
    S->Top = NULL;
}

[10] void push(value_t v) { struct node *t, *x;
[11]     x = alloc();
[12]     x->data = v;
[13]     do {
[14]         t = S->Top;
[15]         x->next = t;
[16]     } while (CAS(&S->Top,t,x));    // @1
[17] }

[20] value_t pop() { struct node *t, *x;
[21]     do {
[22]         t = S->Top;                // @2
[23]         if (t == NULL)
[24]             return EMPTY;
[25]         x = t->next;
[26]     } while (CAS(&S->Top,t,x));    // @3
[27]     return t->data;
[28] }

```

Fig. 1. Treiber’s non-blocking stack algorithm.

First, thread  $T$  calls `pop`, executes lines 21–25 and is then descheduled. At this point,  $T$ ’s local state is  $t = \alpha$  and  $x = \beta$ . Now, suppose some other thread comes along and pops  $\alpha$  off the stack and then pushes  $\gamma$  onto the stack. If `pop` were to dispose node  $\alpha$ , it is possible for a new node to be allocated at the same address  $\alpha$  and pushed on the stack. Hence, the stack can reach a configuration consisting of the nodes  $\alpha$ ,  $\gamma$ , and  $\beta$ . If  $T$  is rescheduled at this point, the CAS at line 26 will succeed, but will remove two nodes from the stack instead of one. This is known as the ‘ABA’ problem in the literature.

*Linearization points.* The linearization points are annotated with comments at the right-hand side. All of them are conditional: @1 and @3 are linearization points if and only if the respective CAS succeeds; @2 is a linearization point if and only if the value stored to  $t$  is NULL. (@2 is the linearisation point of a failed pop operation: at this point we know that the stack is empty.) To carry out the verification, we expect the programmer to annotate these points with auxiliary code asserting that they are linearization points.

*Actions.* In order to verify the given algorithm, we also require the user to specify a set of precondition-postcondition pairs (a.k.a. actions) that abstract the possible atomic effects of the algorithm. For Treiber’s stack, we need:

```

action APush()
    [S→Top: n                               * ABS→val: A]
    [S→Top: y * y→data: e, next: n * ABS→val: @list(@item(e), A)]
action APop()
    [S→Top: y * y→data: e, next: n * ABS→val: @list(@item(e), A)]
    [S→Top: n * y→data: e                   * ABS→val: A]

```

These actions use separation logic notation and (ignoring the `ABS` part) describe the effect of a successful `CAS` at lines 16 and 26 respectively. Variables starting with an underscore (e.g. `_n`) are logical variables and are implicitly quantified over both assertions of an action. In `APop`, we omit the value of the `next` field of `_y` in the postcondition, because the algorithm does not depend on the value of the `next` field of removed nodes. It does, however, depend on its `data` field, as it is read at line 27. We could have also written `_y→data:_e,next:_n`, but this would slow down the verification. The other lines, as well as failed `CASes` do not change any state visible to other threads.

`ABS` is an auxiliary variable representing the abstract stack that the algorithm supposedly implements. `APush` adds `_e` to the beginning of the abstract stack. Conversely, `APop` removes `_e` from the beginning of the abstract stack. If we initialise `ABS→val` to `@list()` in the constructor `init()`, our tool is able to infer the following invariant:

$$J \stackrel{\text{def}}{=} \exists nv. \text{S} \mapsto \text{Top}:n * \text{lseg}(n, \text{NULL}, v) * \text{ABS} \mapsto \text{val}:v,$$

which says that the concrete singly linked list represents the same value as is stored in the auxiliary variable `ABS`. (The predicate `lseg(n, NULL, v)` asserts that there is a singly list segment starting from `n` and ending with `NULL` that represents the value `v`.) This invariant, also known as the *abstraction map*, is crucial for the linearizability proof, and is used as the precondition of `push` and `pop`.

### 3 Verifying Linearizability

Proving linearizability can be reduced to proving that one transition system simulates another transition system (e.g. [2, 3, 5]). The reduction is straightforward, but expensive: it converts a difficult problem into an even harder problem. Proofs done this way have involved significant human labour, especially in constructing the appropriate simulation relations between the two automata. In one case, Colvin et al. [5] even had to invent an intermediate automaton and construct two simulation relations.

Instead, we employ a simpler –but equally general– proof technique based on auxiliary code annotations. We assume that the programmer knows the linearization point of each method and he annotates this point in the source code. For simple algorithms, such as Treiber’s stack, this task is entirely straightforward and could perhaps be automated. More complicated algorithms generally require more annotations, but these are still manageable and, in any case, simpler than the corresponding simulation relations. For such examples, see [12, Chapter 5].

In order to prove that a method is linearizable, we need a specification describing the intended atomic effect of the method. In our examples, this specification is supplied by the user. If, however, the user does not provide such a specification explicitly, we can extract it from the code itself: we just symbolically execute the code in an isolated (sequential) environment. Usually this simplifies the source code quite dramatically. For example, the two `CASes` in Fig. 1 always succeed in an isolated environment.

Hence, we can assume that we have the concrete program annotated with its linearization points and its specification given as abstract code. To verify linearizability, we define an abstraction map  $J$ , we inline the specification at the annotated linearization points and check the following four properties:

1.  $J$  is an invariant of the system: the concrete and the abstract data structures are always related by  $J$ .
2. In every trace representing the execution (whether terminating or not) of a method, there is *at most one* linearization point of that method call.
3. Every terminating execution trace of a method has *at least one* linearization point.
4. Whenever a method terminates, it returns the same result as the specification embedded at the linearization point.

Putting (2) and (3) together means that terminating executions must have exactly one linearization point. In cases where the abstract code specifying a method has no side-effects (e.g. when `pop` returns `EMPTY`), we can drop condition (2). Dropping (2) simplifies the specification task of read-only methods because the freedom to execute the abstract effect of the method multiple times removes the need of inserting additional auxiliary code to ensure that the abstract effect was executed exactly once.

For a user-supplied abstraction map  $J$ , we verify property (1) by checking that  $J$  holds initially (when the constructor `init` returns) and that  $J$  is stable under interference from the given actions. Otherwise, we infer  $J$  by taking the inferred post-condition of `init` and doing a fix-point calculation to compute a weaker assertion that is stable under interference from the given actions. For details on this fix-point calculation, see [13].

Checking conditions (2) and (3) is trivial for Treiber’s stack, but can be quite difficult in general because the linearization point of a method may be in the code of a concurrent execution of some other method.

We verify these properties using a simple intentional encoding. For each method, we create a new boolean variable which we initialize to `false` at the beginning of the method call. At each linearization point inside the method, we check that that the variable contains `false` and update it to `true`. At the end of the method, we check that the variable is `true`.

Gao et al. [4] instead keep a counter initially set to 0, incremented at each linearization point, and prove that it contains 1 at the end of the method. Besides using more state than necessary, their approach does not imply property (2) for non-terminating executions.

### 3.1 Linearization Points in Other Threads

In more complex algorithms, the linearization point along some execution paths of a method is within code performed by another concurrently executing thread. This case arises frequently in methods that have no side-effects, and in algorithms that use ‘helping.’

Dealing with such linearization points is not difficult. For each method call, we simply associate an auxiliary descriptor record with one field containing the name of the method, one field for each argument of the method, and one additional field, `ABS_RESULT`, which is assigned at the linearization point. At the beginning of each method, we add auxiliary code that allocates a new such record in the heap and initializes its fields. To check that the linearization point happens at most once, we initialize `ABS_RESULT` with a certain reserved value `UNDEF` and at the linearization points we check that `ABS_RESULT` still contains this special value. At the method’s return point, we check that the value returned is the same as the one stored in `ABS_RESULT` (and different than `UNDEF`). This ensures that the linearization point occurred exactly once, but allows the auxiliary record to be shared and a different thread to have actually set the `ABS_RESULT` field.

## 4 Shape-value Abstraction

Most shape analyses abstract away the values stored in the data structures. This renders them practically useless for proving linearizability because the crucial invariant we need to prove is that the concrete data structure represents the same value as the abstract state.

Our abstraction follows a two step approach. First we abstract the shapes of the data structures, and then we abstract the values stored in those data structures. These two steps are independent to each other, and hence we can combine any suitable shape abstraction with any suitable value abstraction. Formally, our abstraction function is simply the composition of two abstractions:

$$\alpha_{\text{total}} = \alpha_{\text{value}} \circ \alpha_{\text{shape}}$$

The function  $\alpha_{\text{shape}}$  handles shape-related issues, whereas the function  $\alpha_{\text{value}}$  handles value-related issues. Correspondingly, the concretization function is the composition of the two corresponding concretization functions:

$$\gamma_{\text{total}} = \gamma_{\text{shape}} \circ \gamma_{\text{value}}$$

This setup simplifies proving correctness of the analysis: we can prove separately that the two abstraction functions are correct.

In practice, our abstract domain will simply be a subset of the concrete domain; hence, the  $\gamma$ -functions are just the corresponding inclusion (i.e. the identity) functions.

### 4.1 Shape Abstraction

Deriving the shape abstraction ( $\alpha_{\text{shape}}$ ) from a given shape analysis is usually straightforward. We just note that the abstractions performed by existing logic-based shape analyses can be decomposed in two more primitive abstractions: one that abstracts shape information, but treats values precisely, and a second one that abstract all value-related information.

We proceed by a concrete example. We derive a value-remembering shape abstraction from the shape analysis of Distefano et al. [14]. Based on separation logic, Distefano’s analysis handles singly linked list data structures. The most interesting assertion in their domain is  $\text{lseg}(x, y)$  which denotes a singly linked list segment starting at address  $x$  and ending at  $y$ . For technical reasons (see [13] for details), we prefer a slightly different version of the list segment predicate, whose inductive definition is given below:

$$\begin{aligned} \text{lseg}(x, y) = & (x = y \wedge \text{emp}) \\ & \vee \exists bz. z \mapsto \{\text{.next} = z, \text{.data} = b\} * \text{lseg}(z, y) \end{aligned}$$

We can straightforwardly extend the list segment predicate with an additional argument recording the sequence of values represented by the list. In the definition below,  $\epsilon$  denotes the empty sequence,  $\langle b \rangle$  is the singleton sequence containing  $b$ , and  $x \cdot y$  is the concatenation of sequences  $x$  and  $y$ :

$$\begin{aligned} \text{lseg}_{\text{new}}(x, y, a) = & (x = y \wedge a = \epsilon \wedge \text{emp}) \\ & \vee \exists bcz. a = \langle b \rangle \cdot c * z \mapsto \{\text{.next} = z, \text{.data} = b\} * \text{lseg}_{\text{new}}(z, y, c) \end{aligned}$$

Distefano’s abstraction function consists of applying a set of rewrite rules as much as possible. We do the same, but wherever Distefano would have used the rewrite rule  $\text{lseg}(x, y) * \text{lseg}(y, z) \Longrightarrow \text{lseg}(x, z)$ , we instead use the rewrite rule  $\text{lseg}_{\text{new}}(x, y, a) * \text{lseg}_{\text{new}}(y, z, b) \Longrightarrow \text{lseg}_{\text{new}}(x, z, a \cdot b)$ . Our rewrite rule treats the information about which value the list represents precisely.

In essence, we have decomposed Distefano’s abstraction function  $\alpha_{\text{Distefano}}$  into two steps,  $\alpha_{\text{Distefano}} = \alpha_{\text{forget\_values}} \circ \alpha_{\text{shape}}$ , where  $\alpha_{\text{forget\_values}}$  maps every  $\text{lseg}_{\text{new}}(x, y, v)$  into  $\text{lseg}(x, y)$ . Our abstraction will keep the  $\alpha_{\text{shape}}$  part, but replace the  $\alpha_{\text{forget\_values}}$  function with something more appropriate.

The proof of the soundness of  $\alpha_{\text{shape}}$  is a simple adaptation of Distefano’s proof of soundness.

## 4.2 Value Abstraction

Now we turn to the abstraction of values appearing in a formula. Recall that the basic invariant in a linearizability proof is that two data structures represent the same value. Therefore we want an abstraction that remembers some correlations between equal values. To be concrete, consider we want to abstract the values in the following assertion:

$$\text{lseg}(x, 0, b \cdot c \cdot d) * \text{lseg}(y, 0, a \cdot b) * \text{lseg}(z, 0, a \cdot b).$$

There are two natural choices as to what the abstraction should achieve:

1. Keep track of the correlation between equal values (top-level expressions):

$$\exists uv. \text{lseg}(x, 0, u) * \text{lseg}(y, 0, v) * \text{lseg}(z, 0, v).$$

2. Also keep track of the correlation between equal subexpressions (such as  $b$ ):

$$\exists uvw. \text{lseg}(x, 0, u \cdot v) * \text{lseg}(y, 0, w \cdot u) * \text{lseg}(z, 0, w \cdot u).$$

For proving linearizability, it turns out that we need the second choice. The first choice can prove the linearizability of `push`, but not of `pop`. The additional correlations that the second choice remembers are needed in some proofs. In addition, the second choice is more robust against more aggressive shape abstraction, but it is more complicated and potentially slower. Furthermore, considering syntactic subexpressions is not sufficient, but one has to take account of the properties (e.g. associativity, commutativity) of the value constructors.

Our general approach for performing value abstraction works as follows. First, we collect the set  $T$  of all values appearing in the formula. From that set, we deduce a set of values,  $S$ , that we will ‘forget’ (i.e. existentially quantify over). For each value  $v_i$  in  $S$ , we introduce a fresh existentially quantified variable  $x_i$ , and we (back-)substitute  $x_i$  for  $v_i$  in the assertion. This abstraction is sound irrespective of  $S$ , because  $P(v_1, \dots, v_n) \implies \exists x_1, \dots, x_n. P(x_1, \dots, x_n)$ .

The way we select  $S$  is crucial for the precision of the analysis. To get the first choice, simply choose  $S = T$ . To get the second choice, more work is necessary. Below, we consider this additional work for two kinds of values: (i) sets and multisets, and (ii) strings/sequences.

*Sets & multisets.* Consider expressions denoting sets or multisets constructed using the operations: empty set/multiset, singleton set/multiset, and set/multiset union. (We shall ignore intersection and difference operators.) To take care of the associativity and commutativity of  $\cup$ , we represent such set expression canonically as a union of a set of expressions and we have a special constructor for singleton sets. For example, the set expression  $1, 2 \cup (x \cup y)$  would be represented as  $\{\text{singleton}(1), \text{singleton}(2), x, y\}$ . Then, we compute the set  $S$  (of set expressions) according to the following algorithm:

```

S := T \ {∅};
while ∃x, y ∈ S. x ≠ y ∧ x ∩ y ≠ ∅ do
  S := (S \ {x, y}) ∪ ({x \ y, x ∩ y, y \ x} \ {∅})

```

We start with  $T$ , the set of all (set) values appearing in the formula. In the loop, while there exist overlapping sets in  $S$ , we remove them from  $S$  and add the three partitions. At the end, all the elements  $S$  will be disjoint, and any element of  $T$  denoting a set will be expressible as a union of elements in  $S$ . Notice that these rewrites are confluent: the choice of  $x$  and  $y$  at each loop iteration does not affect the final result.

Here is our algorithm as applied to a small example:

```

Initial configuration:           {1, 2, 3}, {1, 4, 5}, {2, 3, 6}.
Choosing x = {1, 2, 3} and y = {1, 4, 5} yields {1}, {2, 3}, {4, 5}, {2, 3, 6}.
Choosing x = {2, 3} and y = {2, 3, 6} yields  {1}, {2, 3}, {4, 5}, {6}.
No further loop iterations are possible.

```

*Sequences.* Sequences are strings over the alphabet of expressions. A sequence expression is built out of three operations: the empty sequence ( $\epsilon$ ), the singleton sequence (which we write  $\langle x \rangle$ ) and concatenation (denoted  $x \cdot y$ ). Analogously to sets, the analysis represents sequence expressions as a sequence of expressions that are concatenated together. We compute  $S$  as follows:

$$\begin{aligned}
S &:= T \setminus \{\epsilon\}; \\
\mathbf{while} &\left( \begin{array}{l} \exists x \in S, y \in S. \exists z, x_1, x_2, y_1, y_2. \\ x \neq y \wedge z \neq \epsilon \wedge x = x_1 \cdot z \cdot x_2 \wedge y = y_1 \cdot z \cdot y_2 \end{array} \right) \mathbf{do} \\
S &:= (S \setminus \{x, y\}) \cup (\{x_1, x_2, y_1, y_2, z\} \setminus \{\epsilon\})
\end{aligned}$$

We start with  $T$ , the set of all values in the formula. In the loop, while there exists a non-empty common subsequence ( $z$ ) in two elements of  $S$ , we remove those elements from  $S$ , and replace them with the partitions  $x_1$ ,  $x_2$ ,  $y_1$ ,  $y_2$ , and  $z$ . Unlike the set/multiset algorithm, different instantiations of the existential variables can lead to different final results. This is problematic because some results are better than others (we want to minimize the cardinality of the final  $S$  so that we do not accidentally miss any abstraction opportunities). Fortunately, a simple condition ensures that the best result is found: the  $z$  selected must be a (local) maximum. Formally, for all  $z'$ , if  $z \sqsubseteq z'$ , then  $z' \not\sqsubseteq x$  or  $z' \not\sqsubseteq y$ , where  $x \sqsubseteq y$  holds if and only if there exist  $w_1$  and  $w_2$  such that  $y = w_1 \cdot x \cdot w_2$ . Ensuring this condition is an easy programming task.

## 5 Extensions to RGSep

We have implemented our abstraction function in a static analyzer based on RGSep [13]. In this section, we will briefly go over the key concepts of RGSep, and show how we modified its atomic rule to deal with instructions such as CAS.

In RGSep, the state of the program is logically divided into a static number of (disjoint) partitions, which are called regions. Each thread of the system owns one region for its local data, and there are also regions containing data that is shared among threads.

The program logic permits each thread to access local state directly, and restricts shared state accesses to use some form of synchronisation (e.g. mutexes, atomic reads, CAS). At synchronisation points, the thread can re-adjust the boundaries between local and shared state. Whenever a thread modifies the shared state (or the partitioning of the shared state), the logic ensures that the correctness of the other threads is resistant to the modification. This is achieved with rely/guarantee reasoning.

In particular, the concurrent behaviour of each thread is abstracted by a set of precondition-postcondition pairs, known as *actions*. These actions summarise what modifications the atomic statements of a thread can perform on the shared state. For each atomic statement of a thread, Calcagno et al. [13] check that there is an action abstracting its entire effect. This is sufficient if all the atomic blocks consist of a single memory access, but is awkward for larger atomic statements such as CAS. CAS has a conditional effect: if it reads the expected value, then it

modifies the state; else it does nothing. We can write an action that captures this complex effect, but it will be quite complex itself. For instance, the `Apush` action of Section 2 would have to use a post-condition with a disjunction, encoding the two possibilities of the CAS:

```
_x==0 * S→Top:_y * _y→data:_s,next:_n * ABS→val:@list(@item(_e),_A)
|| _x!=0 * S→Top:_n * ABS→_A
```

Not only is the action unnecessarily long, but also the fix-point calculations that depend on it will be slower.

Instead we allow actions to specify parts of an atomic statement. For example, the actions of Section 2 describe only the effects of successful CASes. We change the input language of Calcagno et al. [13] by dropping action annotations from atomic statements and adding a new form of statement for action annotation (the `do...as` block) which can appear only inside atomic blocks.

In the proof rules below, the judgement  $\{P_0 \mid P_1\} C \{Q_0 \mid Q_1\}$  says that the program  $C$  has local precondition  $P_0$ , shared precondition  $P_1$  local postcondition  $Q_0$  and shared postcondition  $Q_1$ . (In reality, we have an indexed family of shared preconditions and shared postconditions, but we will describe our rules as if there was only one for simplicity.) Symbolic execution takes  $P_0$ ,  $P_1$ , and  $C$  as arguments and computes (strongest)  $Q_0$  and  $Q_1$ . Normally, commands can access only the local state  $P_0$ . As an exception, memory reads inside an atomic block can also access the shared state:

$$\{P \mid e \mapsto \text{field}:e' * Q\} x = e \rightarrow \text{field}; \{x = e' * P \mid e \mapsto \text{field}:e' * Q\}$$

Unlike Calcagno et al., when we enter an atomic block, we do nothing. When we exit, we compute a weaker shared postcondition that is resistant to interference from other threads.

$$\frac{\{P_0 \mid P_1\} C \{Q_0 \mid Q_1\}}{\{P_0 \mid P_1\} \text{atomic } C \{Q_0 \mid \text{stabilize}(Q_1)\}}$$

When we encounter an action annotation, we have more work to do. At the beginning of the block, we remove the precondition  $P$  of the action from the shared state, and add it to the local state. Correspondingly, at the end of the block we remove the postcondition  $Q$  of the action from the local state and add it to the shared state.

$$\frac{\{P_0 * P \mid P_2\} C \{Q_0 * Q \mid Q_2\}}{\{P_0 \mid P * P_2\} \text{do } C \text{ as}_{P \rightsquigarrow Q} \{Q_0 \mid Q * Q_2\}}$$

This ensures that the annotated action accounts for any change that  $C$  makes to the shared state. Therefore, as the shared state can be changed only within `do...as` blocks, the set of annotated actions covers every possible shared state change that the program can make.

Experience suggests that writing these action annotations is straightforward and that the process of figuring out the correct actions has a very local, syntactic nature. It is possible that in many simple cases these annotations can be inferred automatically, but we have not investigated this possibility yet.

Data structure	MS	NU	Full	NU [7]	NU [8]	NU [8]
DCAS stack	0.2s	0.2s	0.3s	N/A	N/A	N/A
Two-lock queue [15]	0.8s	2.6s	3.4s	1296s (4)	90s (5)	2122s (15)
Lock-coupling list [6]	0.5s	3.2s	11.0s	14153s (2)	N/A	N/A
Treiber stack [16]	0.4s	0.6s	1.2s	555s (3)	22s (4)	3421s (20)
with too precise action	4.9s	6.8s	9.6s			
Non-blocking queue [15]	12s	23s	99s	1874s (2)	N/A	N/A
field deletion optim.	12s	21s	82s			
user-supplied invariant	10s	15s	73s			
both optimisations	10s	14s	63s			
Non-blocking queue [2]	17s	41s	107s	340s (2)	694s (3)	1 day (15)

**Table 1.** Experiments with standard concurrent algorithms.

## 6 Evaluation

Table 1 presents our experimental results. We verify a number of concurrent algorithms from the literature.

The first three algorithms do not leak memory. The DCAS stack is similar to Treiber’s stack (presented in Section 2), but `pop` uses a double compare-and-swap instruction instead of a single CAS. The two-lock queue is due to Michael and Scott [15] and uses two locks: one for protecting the head of the list, and one for the tail of the list. The lock-coupling list [6] represents a set of integers as a sorted linked list with one lock per node. When traversing the list, locks are acquired in a hand-over-hand fashion. The available operations are single element addition, removal, and test of membership. As we have not implemented an abstraction for sorted lists, we currently verify that when these operations succeed, they are the correct multiset operations.

The next three algorithms have memory leaks and depend on a garbage collector for correctness. Treiber’s stack was presented in Section 2. We verify Treiber’s stack with the actions shown in Section 2, and also with the alternative action for `pop` which keeps track of a lot of unnecessary state. The first non-blocking queue algorithm is Michael and Scott’s well-known algorithm [15]. We verify the original algorithm, and we measure the effect of two manual optimisations: (i) write an invalid address to a field that is never used again; (ii) provide a semantically equivalent invariant to the one inferred automatically that avoids an unnecessary case split. The second non-blocking queue algorithm is a slight variation which was verified by Doherty et al. [2].

Each column records verification time in seconds. Our tests were conducted on a 2.6GHz E6600 Core 2 Duo processor running Linux. In all cases, our memory consumption is under 3 megabytes. **MS** checks that there are no memory errors (e.g. null pointer dereferences), and for the first three algorithms that there are also no memory leaks. **NU** also checks that the concrete and the abstract data structure represent the same value and that each concrete method call returns the same value as the abstract operation at the linearization point. Finally,

**Full** also checks that the abstract operation happens *at most once* on every execution and *exactly once* on every terminating execution. This constitutes a proper linearizability proof.

The last three columns display the results of Amit et al. [7] and Manevich et al. [8]. Both works verify linearizability for fixed number of threads, here displayed in brackets, and do not check that the linearisation point occurred at most once in every execution trace. Comparison with these works is purely indicative; direct comparison is unfair because the tools are quite different. We require slightly more annotations than them, but use a simpler abstraction and verify linearizability for an unbounded number of threads. In all the examples we used the same shape abstraction, whereas [7, 8] use slightly specialized abstractions for some algorithms.

We have also performed tests where we inserted errors in the algorithms. In all these cases, our tool failed to prove linearizability.

## 7 Related Work

*Automatic Verification.* Wang and Stoller [17] present a static analysis that verifies linearizability for an unbounded number of threads. Their analysis essentially detects certain coding patterns, which are known to be atomic irrespective of the environment. Algorithms such as Michael and Scott’s non-blocking queue that do not follow these coding patterns have to be rewritten.

Amit et al. [7] presented a shape difference abstraction that tracks the difference between two heaps. This approach works well if the concrete heap and the abstract heap have almost identical shapes during the entire algorithm. If, however, we are verifying a concurrent tree algorithm that rebalances the tree every so often, then the concrete heap and the abstract heap may differ dramatically regarding their shape, but not the values stored. In such cases, any abstraction requiring that the two heaps are isomorphic will fail completely.

Finally, Yahav and Sagiv [18] and Calcagno et al. [13] use shape analysis to check simple safety properties of list-based concurrent algorithms, but cannot verify linearisability.

*Semi-automatic Verification.* In [2–5], the PVS theorem prover was used to check hand-crafted linearizability proofs. These papers prove linearizability using different techniques than the one used here. See §3 for details.

## 8 Conclusion

We have demonstrated that RGSep and shape-value abstraction enable effective automatic linearizability proofs. The examples verified are typical of the research literature 5–10 years ago. The techniques can also cope with more complex algorithms, but the shape analyses must be powerful enough to describe the data structures used in the algorithms. As shape analyses based on separation logic are relatively new, they are still restricted to linked-list data structures.

In the future, we plan to improve the underlying shape analyses to handle other kinds of data structures such as arrays, and to attempt to infer the necessary action annotations automatically.

## References

1. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Program. Languages and Systems* **12**(3) (1990) 463–492
2. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: *Int. Conf. on Formal Techniques for Networked and Dist. Sys. (FORTE 2004)*. Volume 3235 of LNCS., Madrid, Spain, IFIP WG 6.1, Springer (September 2004) 97–114
3. Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. *Electronic Notes in Theoretical Computer Science* **137**(2) (2005) 93–110
4. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free dynamic hash tables with open addressing. *Distributed Computing* **18**(1) (July 2005) 21–42
5. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set. In: *18th CAV*. Volume 4144 of LNCS., Springer (2006) 475–488
6. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: *11th PPOPP*, ACM (2006) 129–136
7. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearisability. In: *19th CAV*. (2007)
8. Manevich, R., Lev-Ami, T., Ramalingam, G., Sagiv, M., Berdine, J.: Heap decomposition for concurrent shape analysis. Technical Report TR-2007-11-85453, Tel Aviv University (November 2007)
9. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: *18th International Conference on Concurrency Theory (CONCUR)*. Volume 4703 of LNCS., Springer (September 2007)
10. Jones, C.B.: Specification and design of (parallel) programs. In: *IFIP Congress*. (1983) 321–332
11. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, IEEE Computer Society (2002) 55–74
12. Vafeiadis, V.: Fine-grained concurrency verification. Ph.D. dissertation, University of Cambridge Computer Laboratory (2007)
13. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: *14th International Static Analysis Symposium (SAS 2007)*. Volume 4634 of LNCS., Springer (August 2007)
14. Distefano, D., O’Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: *TACAS*. Volume 3920 of LNCS. (2006) 287–302
15. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: *PODC*. (1996)
16. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr. (1986)
17. Wang, L., Stoller, S.D.: Static analysis of atomicity for programs with non-blocking synchronization. In: *10th PPOPP*, New York, NY, USA, ACM Press (2005) 61–71
18. Yahav, E., Sagiv, M.: Automatically verifying concurrent queue algorithms. *Electronic Notes in Theoretical Computer Science* **89**(3) (2003)