Shape-Value Abstraction for Verifying Linearizability

Viktor Vafeiadis

Microsoft Research, Cambridge, UK

Abstract. This paper presents a novel abstraction for heap-allocated data structures that keeps track of both their shape and their contents. By combining this abstraction with thread-local analysis and rely-guarantee reasoning, we can 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.

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. Therefore, 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. However, 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 [2–5], but until recently nobody had automated the derivation of such proofs. Amit et al. [6] used shape analysis to verify linearizability for a fixed (small) number of threads. More recently, Manevich et al. [7] and Berdine et al. [8] extended this analysis, so that it works for a larger (fixed) number of threads and for an unbounded number of threads respectively. These works 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 check that that the specified linearization points are sound, and we allow complex linearization points that occur in a different thread than the one being verified.

Our prototype implementation is based on RGSep [9], a program logic that combines rely-guarantee [10] and separation logic [11]. As a result, it deals with an unbounded number of threads, can reason about memory deallocation, which affects linearizability in subtle ways (see Sect. 2), and can prove the absence of memory leaks (where applicable).

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

– We present a simple proof method for verifying linearizability given a specified set of linearization points (see Sect. 3). Our method can handle linearization points occuring in a different thread than the one being verified.
– Our shape analysis can remember an adjustable amount of information about the values stored in a data structure (see Sect. 4). The amount of information can be adjusted by selecting a different backend value abstraction.
– 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 Sect. 5).
– Our tool compares favourably to the other known tools, and succeeded in proving that several concurrent algorithms are linearizable (see Sect. 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

Figure 1 contains C-like pseudocode for Treiber's stack [12], one of the simplest non-blocking concurrent algorithms. The stack is represented as a singly linked list rooted at `S->Top`, which is updated using CAS (compare and swap). CAS is a primitive operation that reads a word from a memory adress and conditionally writes to the same address in one atomic step. In particular, `CAS(&S->Top,t,x)` atomically compares the value of `S->Top` with the value of `t` and if the two match, the `CAS` succeeds: it stores the value of `x` in `S->Top` and returns `1`. Otherwise, the `CAS` fails: it returns `0` and does not change the value of `S->Top`.

This algorithm leaks memory: we cannot free popped nodes because of the following scenario. Assume the stack initially consists of the nodes α and β. First, thread T calls `pop`, executes lines `21–25` and is then descheduled. At this point, T's local state is $\mathtt{t} = \alpha$ and $\mathtt{x} = \beta$. Now, suppose some other thread comes along and pops α off the stack and then pushes γ onto the stack. If `pop` were to dispose node α, it is possible for a new node to be allocated at the same address

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

struct stack {
  struct node *Top;
};

struct stack *S;

void init() {
   S = alloc();
   S->Top = NULL;
/* ABS->val = ε; */
}
```

```
[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.

α and pushed on the stack. Hence, the stack can reach a configuration consisting of the nodes α, γ, and β. 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 linearization 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 summarize 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:⟨e⟩·A]
action APop()  [S↦Top:y ∗ y↦data:e,next:n ∗ ABS↦val:⟨e⟩·A]
               [S↦Top:n ∗ y↦data:e,next:n ∗ ABS↦val:A]
```

These actions use separation logic notation[1] and (ignoring the ABS part) describe the effect of a successful CAS at lines 16 and 26 respectively. The italicized

[1] S↦Top:n denotes that S is a pointer to a structure whose Top field contains n. The $*$ operator is similar to conjunction, but $P * Q$ also asserts that P and Q describe disjoint parts of the memory.

variables (e.g. n) are logical variables and are implicitly quantified over both assertions of an action. The other lines, as well as failed `CAS`es do not change any state visible to other threads.

`ABS` is an auxiliary variable representing the abstract stack that the algorithm supposedly implements. This is formalized as a mathematical sequence. We write ϵ for the empty sequence, $\langle e\rangle$ for the singleton sequence consisting of e, and $\cdot$ for sequence concatenation. Action `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 ϵ in the constructor `init()`, our tool is able to infer the following invariant:

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

which says that the concrete singly linked list represents the same value as is stored in the auxiliary variable `ABS`. (The predicate $\mathsf{lseg}(n, \mathtt{NULL}, v)$ asserts that there is a singly list segment starting from n and ending with `NULL` that represents the sequence 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, 4]). 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. [4] 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 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 [13, 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 `CAS`es in Fig. 1 always succeed in an isolated environment.

Hence, we can assume that the concrete program is annotated with its linearization points and its specification given as abstract code. To verify linearizability: we infer 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.
   We satisfy this property by construction. When inferring J, we start with the inferred postcondition of the constructor `init()` and do a fixpoint calculation to compute a weaker assertion that is stable under interference from the given actions. This fixpoint calculation is also known as *stabilization*. For more details about how stabilization is done, see [14]. As the actions soundly overapproximate the system, this implies that the inferred assertion is an invariant of the system.
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) typically reduces the annotation overhead for read-only methods because we do not need to ensure that the abstract effect of the method was executed exactly once.

Checking conditions (2) and (3) may seem trivial for Treiber's stack, but can be quite difficult in general because the linearization point along some execution paths of a method may be 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.'

We verify conditions (2), (3), and (4) with a simple intentional encoding. For each method call, we create 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`. At the linearization points we check that `ABS_RESULT` still contains this special value and update it with the result of the abstract operation. 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.

As the auxiliary record is stored in the heap, it can be shared, and hence, a different thread can execute the auxiliary code that updates the `ABS_RESULT` field. Thus, we are able to handle methods whose linearization points along some executions are in a different thread.

Our use of the reserved value `UNDEF` encodes whether the linearization point has occurred or not. Alternatively, the same information can be recorded by a boolean variable. Gao et al. [3] 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.

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 needed in order to prove linearizability is that the concrete data structure represents the same value as the abstract state.

A possible solution to this problem is to develop a specialized abstract domain that can express this invariant. This approach was followed by Amit et al. [6], who presented an abstract domain tracking graph isomorphism. Here, we will consider a different, possibly more general, solution.

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 the composition of two abstractions:

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

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

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

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

In the following, the abstract domains are just subsets of the concrete domain; hence, the γ-functions are the corresponding inclusion (i.e. the identity) functions.

4.1 Shape Abstraction

Given a shape analysis based on separation logic, deriving the shape abstraction (α_{shape}) is straightforward. The shape analysis's abstraction function can be decomposed in two more primitive functions: one that abstracts shape information, but treats values precisely, and a second one that abstracts all value-related information.

We proceed with a concrete example. We derive a value-remembering shape abstraction from the shape analysis of Distefano et al. [15]. Distefano's analysis is based on separation logic, and handles singly linked data structures.

$$\begin{array}{rcl}
\mathsf{Node}(y,z,b) & \Longrightarrow & \mathsf{junk} \\
\mathsf{Node}(x,y,a) * \mathsf{Node}(y,z,b) & \Longrightarrow & \mathsf{lseg}_{\mathsf{new}}(x,z,\langle a\rangle\cdot\langle b\rangle) \\
\mathsf{lseg}_{\mathsf{new}}(x,y,a) * \mathsf{Node}(y,z,b) & \Longrightarrow & \mathsf{lseg}_{\mathsf{new}}(x,z,a\cdot\langle b\rangle) \\
\mathsf{lseg}_{\mathsf{new}}(y,z,b) & \Longrightarrow & \mathsf{junk} \\
\mathsf{Node}(x,y,a) * \mathsf{lseg}_{\mathsf{new}}(y,z,b) & \Longrightarrow & \mathsf{lseg}_{\mathsf{new}}(x,z,\langle a\rangle\cdot b) \\
\mathsf{lseg}_{\mathsf{new}}(x,y,a) * \mathsf{lseg}_{\mathsf{new}}(y,z,b) & \Longrightarrow & \mathsf{lseg}_{\mathsf{new}}(x,z,a\cdot b)
\end{array}$$

Fig. 2. Shape abstraction rules.

Their abstract domain is a subset of separation logic assertions that includes $*$-conjunction, disjunction, $\mapsto$, emp, junk, lseg, equalities and disequalities. The assertion emp denotes the empty heap (in which nothing is allocated); junk is true for any heap, whether empty or consisting of some allocated nodes. Finally, the predicate $\mathsf{lseg}(x,y)$ denotes a singly linked list segment starting at address x and ending at y. For technical reasons (see [14] for details), we prefer a slightly different version of the list segment predicate, whose inductive definition is given below:

$$\mathsf{lseg}(x,y) \stackrel{\text{def}}{=} (x = y \wedge \mathsf{emp}) \vee (\exists bz.\ \mathsf{Node}(x,z,b) * \mathsf{lseg}(z,y))$$

where $\mathsf{Node}(x,y,v) \stackrel{\text{def}}{=} x \mapsto \{\texttt{.next} = y, \texttt{.data} = v\}$.

We can extend the list segment predicate with an additional argument recording the sequence of values represented by the list.

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

Distefano's abstraction function consists of applying a set of rewrite rules as much as possible. Each rewrite rule is a valid separation logic implication, and eliminates one existentially quantified variable from the input assertion. This ensures that the abstraction function is sound and always terminates. Distefano also proves that his abstract domain is finite; hence, fixpoints in the abstract domain converge.

Our abstraction has the same structure, but we have modified the rewrite rules to record value-related information accurately (see Fig. 2). For example, our last rule is a direct adaptation of Distefano's rule for merging two list segments:

$$\mathsf{lseg}(x,y) * \mathsf{lseg}(y,z) \Longrightarrow \mathsf{lseg}(x,z).$$

Abstraction applies these rules aggressively whenever y is an existentially quantified variable that does not appear in the rest of the formula. Abstraction is sound, because each rewrite rule is a valid separation logic implication.

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

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:

$$\mathsf{lseg}(k, 0, b{\cdot}c{\cdot}d{\cdot}e) * \mathsf{lseg}(l, 0, a{\cdot}b) * \mathsf{lseg}(m, 0, a{\cdot}b) * \mathsf{lseg}(n, 0, e).$$

There are three natural choices as to what abstraction should do:

A. Keep track of the equalities between top-level expressions (such as $a{\cdot}b$):

$$\exists uvw.\ \mathsf{lseg}(k, 0, u) * \mathsf{lseg}(l, 0, v) * \mathsf{lseg}(m, 0, v) * \mathsf{lseg}(n, 0, w)$$

B. Also keep track of the correlations between a top-level expression and subexpressions of another expression (such as e).

$$\exists uvw.\ \mathsf{lseg}(k, 0, u{\cdot}w) * \mathsf{lseg}(l, 0, v) * \mathsf{lseg}(m, 0, v) * \mathsf{lseg}(n, 0, w)$$

C. Also keep track between any two subexpressions (such as b).

$$\exists tuvw.\ \mathsf{lseg}(k, 0, u{\cdot}v{\cdot}w) * \mathsf{lseg}(l, 0, t{\cdot}u) * \mathsf{lseg}(m, 0, t{\cdot}u) * \mathsf{lseg}(n, 0, w)$$

It turns out that choice A is too weak for linearizability proofs, and that we need one of the other two choices. In particular, choice A can prove the linearizability of `push`, but not of `pop`. In the proof outline of `pop`, one of the disjuncts of the assertion between lines `15` and `16` of `pop` is

$$\exists \alpha\beta.\ \mathtt{S{\mapsto}Top{:}t} * \mathtt{t{\mapsto}data{:}}\alpha\mathtt{,next{:}x} * \mathsf{lseg}(\mathtt{x}, \mathtt{0}, \beta) * \mathtt{ABS{\mapsto}val{:}}\langle\alpha\rangle{\cdot}\beta$$

In this case, the first choice would forget the correlation between the value between the concrete data structure and `ABS->val`, which would make it impossible to prove that the concrete `pop` returns the same result as the abstract `pop`.

In our example programs, choice B was sufficient for proving linearizability. Choice C also works, but as it distinguishes more abstract states, it is potentially slower. A benefit of choice C is that it is more robust against more aggressive shape abstractions. Considering syntactic subexpressions is not sufficient, but one has to take the properties (such as associativity and commutativity) of the value constructors into account.

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 $P(v_1, \ldots, v_n) \implies \exists x_1, \ldots, x_n.\ P(x_1, \ldots, x_n)$.

The way we select S is crucial for the precision of the analysis. To get choice A, simply choose $S = T$. To get the other two choices, 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 set expressions 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 $\{\mathit{singleton}(1), \mathit{singleton}(2), x, y\}$. Then, in order to get choice C, we compute the set S of set expressions according to the following algorithm:

```
$S := T \setminus \{\emptyset\}$;
**while** $\exists x, y \in S.\ x \neq y \ \wedge\ x \cap y \neq \emptyset$ **do**
    $S := (S \setminus \{x, y\}) \cup (\{x \setminus y, x \cap y, y \setminus x\} \setminus \{\emptyset\})$
```

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.	

To get choice B, we also require that either $x \subseteq y$ or $y \subseteq x$.

Sequences. Sequences are strings over the alphabet of expressions. They are built out of three operations: the empty sequence (ϵ), 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. To get choice C, we compute S as follows:

```
$S := T \setminus \{\epsilon\}$;
**while** $\begin{pmatrix} \exists x \in S, y \in S.\ \exists z, x_1, x_2, y_1, y_2. \\ x \neq y \wedge z \neq \epsilon \wedge x = x_1 \cdot z \cdot x_2 \wedge y = y_1 \cdot z \cdot y_2 \end{pmatrix}$ **do**
    $S := (S \setminus \{x, y\}) \cup (\{x_1, x_2, y_1, y_2, z\} \setminus \{\epsilon\})$
```

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. To get choice B, we also require that either $x \sqsubseteq y$ or $y \sqsubseteq x$, where $x \sqsubseteq y$ holds if and only if there exist w_1 and w_2 such that $y = w_1 \cdot x \cdot w_2$. Equivalently, to get choice B, we require that either $x_1 = x_2 = \epsilon$ or $y_1 = y_2 = \epsilon$.

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 opportunites). 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 $z' \not\sqsubseteq y$. 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, 14]. 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 perfom on the shared state.

For each atomic statement of a thread, Calcagno et al. [14] 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 from Sect. 2 would have to use a postcondition with a disjunction, encoding the two possibilities of the CAS:

$$\begin{array}{ll} & \texttt{\_x==0} * \texttt{S}\mapsto\texttt{Top:}y * y\mapsto\texttt{data:}e\texttt{,next:}n * \texttt{ABS}\mapsto\texttt{val:}\langle e\rangle\cdot A \\ \texttt{||} & \texttt{\_x!=0} * \texttt{S}\mapsto\texttt{Top:}n * \texttt{ABS}\mapsto\texttt{val:}A \end{array}$$

Not only is the action unnecessarily long (and therefore difficult to specify or to infer), but it also slows down stabilization. Stabilization is an expensive computation that is executed after the symbolic execution of every atomic command. Given an assertion, it does a fixpoint calculation to compute a weaker assertion that it stable under the set of given actions. Its execution time is roughly proportional to the size of the action definitions.

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. [14] by dropping action annotations from atomic statements and adding a new form of statement for action annotation (the '`do...as`' block). We impose a syntactic restriction that these '`do...as`' blocks 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:

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

Unlike Calcagno et al., our symbolic execution does nothing at entries to atomic blocks. At exits, it computes a weaker shared postcondition that is resistant to interference from other threads.

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

When symbolic execution encounters an action annotation, it has more work to do. At the beginning of the block, it removes the precondition P of the action from the shared state, and adds it to the local state. Correspondingly, at the end of the block it removes the postcondition Q of the action from the local state and adds it to the shared state.

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

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.

6 Evaluation

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

The first four 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-slot buffer is an obstruction-free implementation of an atomic register with a single reader and a single writer.

Data structure	**Shape analysis**	**Linearizability**	**Berdine et al.** [8]
DCAS stack	0.2s	0.2s	–
Two-slot buffer	0.4s	1.2s	–
Two-lock queue [16]	0.5s	0.6s	17s
Lock-coupling list [5]	0.3s	0.5s	–
Treiber stack [12]	0.2s	0.3s	7s
Non-blocking queue [16]	2.6s	5.1s	–
Non-blocking queue [2]	2.6s	4.8s	252s
RDCSS [17]	1.6s	87.7s	–

Table 1. Verification times for a collection of concurrent algorithms.

The two-lock queue is due to Michael and Scott [16] and uses two locks: one for protecting the head of the list, and one for the tail of the list. The lock-coupling list [5] 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 four algorithms have memory leaks and depend on a garbage collector for correctness. Treiber's stack was presented in Sect. 2. The first non-blocking queue algorithm is the well-known Michael and Scott's queue [16]. The second non-blocking queue algorithm is a slight variation which was verified by Doherty et al. [2]. Finally, RDCSS [17] is a lock-free implementation of restricted double-compare single-swap primitive. Proving linearizability of RDCSS is challenging because some of its linearization points are executed by different threads, and specifying them requires an auxiliary prophecy variable.

Each column records verification time in seconds. Our tests were conducted on a 3.4GHz Pentium 4 processor running Windows Vista. In all cases, memory consumption was under 5 megabytes. The first column measures the time required by the underlying shape analysis. This infers the shape of the data structures used in the heap and checks that there are no memory errors (e.g. null pointer dereferences). For the first four algorithms, it also checks that there are also no memory leaks. The second column measures the total time required to prove linearizability using the techniques described in this paper. The difference between these two columns represents the additional amount of work that is needed in order to prove linearizability.

Finally, the last column displays the results of Berdine et al. [8]. Comparison with this work is purely indicative; direct comparison is unfair because the tools are quite different. We used the same shape abstraction for all the examples, but require actions to be annotated. In contrast, Berdine et al. do not require any action annotations, but use slightly different abstractions for each algorithm and require an user-supplied heap decomposition.

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 [18] 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. [6] 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. More recently, Manevich et al. [7] and Berdine et al. [8] have presented some improvements to this analysis, which are orthogonal to the task of verifying linearizability.

Finally, Yahav and Sagiv [19] and Calcagno et al. [14] use shape analysis to check simple safety properties of list-based concurrent algorithms, but cannot verify linearizability.

Semi-automatic Verification. In [2–4], 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 Sect. 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.

We believe that both further instances of shape-value abstraction as well as the presented value abstractions apply equally to other verification problems, but we have not investigated this possibility yet.

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.

Acknowledgments. We would like to thank Alan Mycroft, Hongseok Yang and the anonymous referees for providing helpful feedback on earlier drafts of this work.

References

1. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. on Program. Languages and Systems **12**(3): 463-492 (1990)
2. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97-114. Springer, Heidelberg (2004)
3. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free dynamic hash tables with open addressing. Distributed Computing **18**(1): 21-42 (2005)
4. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set. In Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 473-488. Springer, Heidelberg (2006)
5. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In PPoPP 2006, pp. 129-136. ACM Press (2006)
6. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearisability. In Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590. Springer, Heidelberg (2007)
7. Manevich, R., Lev-Ami, T., Ramalingam, G., Sagiv, M., Berdine, J.: Heap decomposition for concurrent shape analysis. In Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS vol. 5079, pp. 363-377. Springer, Heidelberg (2008)
8. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In Gupta, A., Malik, S. (eds.) CAV 2008. LNCS vol. 5123, pp. 399-413. Springer, Heidelberg (2008)
9. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256-271. Springer, Heidelberg (2007)
10. Jones, C.B.: Specification and design of (parallel) programs. In IFIP Congress, pp. 321-332. (1983)
11. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In LICS 2002, pp. 55-74. IEEE Computer Society (2002)
12. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr. (1986)
13. Vafeiadis, V.: Fine-grained concurrency verification. Ph.D. dissertation, University of Cambridge (2007) Also available as Technical Report UCAM-CL-TR-726.
14. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 233–248. Springer, Heidelberg (2007)
15. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287-302. Springer, Heidelberg (2006)
16. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In PODC, pp. 267-275. ACM Press (1996)
17. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 265-279. Springer, Heidelberg (2002)
18. Wang, L., Stoller, S.D.: Static analysis of atomicity for programs with non-blocking synchronization. In PPoPP 2005, pp. 61-71. ACM Press (2005)
19. Yahav, E., Sagiv, M.: Automatically verifying concurrent queue algorithms. Electronic Notes in Theoretical Computer Science **89**(3) (2003)