Quantitative Reasoning for Proving Lock-Freedom

Jan Hoffmann
Yale University

Michael Marmar
Yale University

Zhong Shao
Yale University

***Abstract*—This article describes a novel quantitative proof technique for the modular and local verification of lock-freedom. In contrast to proofs based on temporal rely-guarantee requirements, this new quantitative reasoning method can be directly integrated in modern program logics that are designed for the verification of safety properties. Using a single formalism for verifying memory safety and lock-freedom allows a combined correctness proof that verifies both properties simultaneously.**

This article presents one possible formalization of this quantitative proof technique by developing a variant of concurrent separation logic (CSL) for total correctness. To enable quantitative reasoning, CSL is extended with a predicate for affine tokens to account for, and provide an upper bound on the number of loop iterations in a program. Lock-freedom is then reduced to total-correctness proofs. Quantitative reasoning is demonstrated in detail, both informally and formally, by verifying the lock-freedom of Treiber's non-blocking stack. Furthermore, it is shown how the technique is used to verify the lock-freedom of more advanced shared-memory data structures that use elimination-backoff schemes and hazard-pointers.

I. Introduction

The efficient use of multicore and multiprocessor systems requires high performance shared-memory data structures. Performance issues with traditional lock-based synchronization has generated increasing interest in *non-blocking shared-memory data structures*. In many scenarios, non-blocking data structures outperform their lock-based counterparts [1], [2]. However, their optimistic approach to concurrency complicates reasoning about their correctness.

A non-blocking data structure should guarantee that any sequence of concurrent operations that modify or access the data structure do so in a consistent way. Such a guarantee is a safety property which is implied by linearizability [3]. Additionally, a non-blocking data structure should guarantee certain *liveness properties*, which ensure that desired events eventually occur when the program is executed, independent of thread contention or the whims of the scheduler. These properties are ensured by *progress conditions* such as obstruction-freedom, lock-freedom, and wait-freedom [4], [5] (see §II). In general, it is easier to implement the data structure efficiently if the progress guarantees it makes are weaker. Lock-freedom has proven to be a sweet spot that provides a strong progress guarantee and allows for elegant and efficient implementations in practice [6], [7], [8], [9].

The formal verification of practical lock-free data structures is an interesting problem because of their relevance and the challenges they bear for current verification techniques: They employ fine-grained concurrency, shared-memory pointer-based data structures, pointer manipulation, and control flow that depends on shared state.

Classically, verification of lock-freedom is reduced to model-checking liveness properties on whole-program execution traces [10], [11], [12]. Recently, Gotsman et al. [13] have argued that lock-freedom can be reduced to modular, thread-local termination proofs of concurrent programs in which each thread only executes a single data-structure operation. Termination is then proven using a combination of concurrent separation logic (CSL) [14] and temporal trace-based rely-guarantee reasoning. In this way, proving lock-freedom is reduced to a finite number of termination proofs which can be automatically found. However, as we show in §II, this method is not intended to be applied to some lock-free data structures that are used in practice.

These temporal-logic based proofs of lock-freedom are quite different from informal lock-freedom proofs of shared data structures in the systems literature (e.g., [7], [9]). The informal argument is that the failure to make progress by a thread is always caused by *successful progress* in an operation executed by another thread. In this article, we show that this intuitive reasoning can be turned into a formal proof of lock-freedom. To this end, we introduce a *quantitative compensation scheme* in which a thread that successfully makes progress in an operation has to logically provide resources to other threads to compensate for possible interference it may have caused. Proving that all operations of a data structure adhere to such a compensation scheme is a safety property which can be formalized using minor extensions of modern program logics for fine-grained concurrent programs [14], [15], [16], [17].

We formalize one such extension in this article using CSL. We chose CSL because it has a relatively simple meta-theory and can elegantly deal with many challenges arising in the verification of concurrent, pointer-manipulating programs. Parkinson et al. [18] have shown that CSL can be used to derive modular and local safety proofs of non-blocking data structures. The key to these proofs is the identification of a *global resource invariant* on the shared-data structure that is maintained by each atomic command. However, this technique only applies to safety properties and the authors state that they are "investigating adding liveness rules to separation logic to capture properties such as obstruction/lock/wait-freedom".

We show that it is not necessary to add "liveness rules" to CSL to verify lock-freedom. As in Atkey's separation logic for quantitative reasoning [19] we extend CSL with a predicate for affine tokens to account for, and provide an upper bound on the number of loop iterations in a program. In this way,

we obtain the first separation logic for total correctness of concurrent programs.

Strengthening the result of Gotsman et al. [13], we first show that lock-freedom can be reduced to the total correctness of concurrent programs in which each thread executes a finite number of data-structure operations. We then prove the total correctness of these programs using our new quantitative reasoning technique and a *quantitative resource invariant* in the sense of CSL. Thus the proof of the liveness property of being lock-free is reduced to the proof of a stronger safety property. The resulting proofs are simple extensions of memory-safety proofs in CSL and only use standard techniques such as auxiliary variables [20] and read permissions [21].

We demonstrate the practicality of our compensation-based quantitative method by verifying the lock-freedom of Treiber's non-blocking stack (§VI). We further show that the technique can be used to verify shared-memory data structures such as Michael and Scott's non-blocking queue [7], Hendler et al.'s non-blocking stack with elimination backoff [9], and Michael's non-blocking hazard-pointer data structures [8] (§VII).

Our method is a clean and intuitive modular verification technique that works correctly for shared-memory data structures that have access to thread IDs or the total number of threads in the system (see §II for details). It can not only be applied to verify total correctness but also to directly prove liveness properties or to verify termination-sensitive contextual refinement. Automation of proofs in concurrent separation logic is an orthogonal issue which is out of the scope of this paper. It would require the automatic generation of loop invariants and resource invariants. Assuming that they are in place, the automation of the proofs can rely on backward reasoning and linear programming as described by Atkey [19].

In summary, we make the following *contributions*.

1) We introduce a new compensation-based quantitative reasoning technique for proving lock-freedom of non-blocking data structures. (§III and §V)
2) We formalize our technique using a novel extension of CSL for total correctness and prove the soundness of this logic. (§IV, §V, and §VI)
3) We demonstrate the effectiveness of our approach by verifying the lock-freedom of Treiber's non-blocking stack (§VI), Michael and Scott's lock-free queue, Hendler et al.'s lock-free stack with elimination backoff, and Michael's lock-free hazard-pointer stack (§VII).

In §VII, we also discuss how quantitative reasoning can verify the lock-freedom of data structures such as maps and sets, that contain loops that depend on the size of data structures. Finally, in §IX, we describe other possible applications of quantitative reasoning for proving liveness properties including wait-freedom and starvation-freedom. For all the rules of the logic, the semantics, and the full soundness proof, see the technical report version of this article [22].

II. Non-Blocking Synchronization

Recent years have seen increasing interest in *non-blocking data structures* [1], [2]: shared-memory data structures that provide operations that are synchronized without using locks and mutual exclusion in favor of performance. A non-blocking data structure is often considered to be correct if its operations are *linearizable* [3]. Alternatively, correctness can be ensured by an invariant that is maintained by each instruction of the operations. Such an invariant is a safety property that can be proved by modern separation logics for reasoning about concurrent programs [18].

Progress Properties: In this article, we focus on complementary *liveness properties* that guarantee the *progress* of the operations of the data structure. There are three different progress properties for non-blocking data structures considered in literature. To define these, assume there is a fixed but arbitrary number of threads that are (repeatedly) accessing a shared-memory data structure exclusively via the operations it provides. Choose now a point in the execution in which one or more operations has started.

- A *wait-free* implementation guarantees that every thread can complete any started operation of the data structure in a finite number of steps [4].
- A *lock-free* implementation guarantees that some thread will complete an operation in a finite number of steps [4].
- An *obstruction-free* implementation guarantees progress for any thread that eventually executes in isolation [5] (i.e., without other active threads in the system).

Note that these definitions do not make any assumptions on the scheduler. We assume however that any code that is executed between the data-structure operations terminates. If a data structure is wait-free then it is also lock-free [4]. Similarly, lock-freedom implies obstruction-freedom [5]. Wait-free data structures are desirable because they guarantee the absence of live-locks and starvation. However, wait-free data structures are often complex and inefficient. Lock-free data structures, on the other hand, often perform more efficiently in practice. They also ensure the absence of live-locks but allow starvation. Since starvation is an unlikely event in many cases, lock-free data structures are predominant in practice and we focus on them in this paper. However, our techniques apply in principle also to wait-free data structures (see §IX).

Treiber's Stack: As a concrete example we consider Treiber's non-blocking stack [6], a classic lock-free data structure. The shared data structure is a pointer S to a linked list and the operations are *push* and *pop* as given in Figure 1.

The operation *push(v)* creates a pointer *x* to a new list node containing the data *v*. Then it stores the current stack pointer *S* in a local variable *t* and sets the next pointer of the new node *x* to *t*. Finally it attempts an atomic *compare and swap* operation *CAS(&S,t,x)* to swing *S* to point to the new node *x*. If the stack pointer *S* still contains *t* then *S* is updated and *CAS* returns *true*. In this case, the do-while loop terminates and the operation is complete. If however, the stack pointer *S* has been updated by another thread so that it no longer contains *t* then *CAS* returns *false* and leaves *S* unchanged. The do-while loop then does another iteration, updating the new list node to a new value of *S*. The operation *pop* works similarly to *push(v)*. If the stack is empty ($t == NULL$) then

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

Node *S;

void init()
  {S = NULL;}
```

```
void push(value_t v) {
  Node *t, *x;
  x = new Node();
  x->data = v;
  do { t = S;
    x->next = t;
  } while(!CAS(&S,t,x));
}
```

```
value_t pop() {
  Node *t, *x;
  do { t = S;
    if (t == NULL)
      {return EMPTY;}
    x = t->next;
  } while(!CAS(&S,t,x));
  return t->data;
}
```

Fig. 1. An implementation of Treiber's lock-free stack as given by Gotsman et al. [13].

```
I := -1; //initialization

ping() ≜ if I == TID then { while (true) do {} }
         else { I := TID }
```

Fig. 2. A shared data structure that shows a limitation of the method of proving lock-freedom that has been introduced by Gotsman et al. [13]. For every n, the parallel execution of n *ping* operations terminates. However, the data structure is not lock-free. (It is based on an idea from James Aspnes.)

pop returns *EMPTY*. Otherwise it repeatedly tries to update the stack pointer with the successor of the top node using a do-while loop guarded by a CAS.

Treiber's stack is lock-free but not wait-free. If other threads execute infinitely many operations they could prevent the operation of a single thread from finishing. The starvation of one thread is nevertheless only possible if infinitely many operations from other threads succeed by performing a successful *CAS*. The use of do-while loops that are guarded by CAS operations is characteristic for lock-free data structures.

Lock-Freedom and Termination: Before we verify Treiber's stack, we consider lock-freedom in general. Following an approach proposed by Gotsman et al. [13], we reduce the problem of proving lock-freedom to proving termination of a certain class of programs. Let D be any shared-memory data structure with k operations $\pi_1, \ldots, \pi_k$. It has been argued [13] that D is lock-free if and only if the following program *terminates* for every $n \in \mathbb{N}$ and every $op_1, \ldots, op_n \in \{\pi_1, \ldots \pi_k\}$: $O_n = ||_{i=1,\ldots,n}\, op_i$. However, this reduction does not apply to all shared-memory data structures. Many non-blocking data structures have operations that can distinguish different callers, for instance by accessing their thread ID. A simple example is described in Figure 2. The shared data structure consists of an integer *I* and a single operation *ping*. If *ping* is executed twice by the same thread without interference from another thread then the second execution of *ping* will not terminate. Otherwise, each call of *ping* immediately returns. As a result, the program $||_{i=1,\ldots,n}$ *ping* terminates for every n but the data structure is not lock-free.

We are also aware of a similar example that uses the total number of threads in the system instead of thread IDs. It is in general very common to use these system properties in non-blocking data structures. Three of the five examples in our paper use thread IDs (the hazard pointer stack, the hazard pointer queue, and the elimination-backoff stack).

Consequently, we have to prove a stronger termination property to prove that a data structure is lock-free. Instead of assuming that each client only executes one operation, we assume that each client can execute finitely many operations. To this end, we define a set of programs $\mathcal{S}^n$ that sequentially execute n operations.

$$\mathcal{S}^n = \{op_1; \ldots; op_n \mid \forall i :\ op_i \in \{\pi_1, \ldots, \pi_k\}\}$$

Let $\mathcal{S} = \bigcup_{n\in\mathbb{N}} \mathcal{S}^n$. We now define the set of programs $\mathcal{P}^m$ that execute m programs in $\mathcal{S}$ in parallel.

$$\mathcal{P}^m = \{ \quad \Big|\Big| \quad s_i \mid \forall i : s_i \in \mathcal{S}\}$$
$$i=1,\ldots,m$$

Finally, we set $\mathcal{P} = \bigcup_{m\in\mathbb{N}} \mathcal{P}^m$. For proving lock-freedom, we rely on the following theorem. By allowing a fixed but arbitrary number of operations per thread we avoid the limitations of the previous approach.

Theorem 1. The data structure D with operations $\pi_1, \ldots, \pi_k$ is lock-free if and only if every program $P \in \mathcal{P}$ terminates.

Proof. Assume first that D is lock-free. Let $P \in \mathcal{P}$. We prove that P terminates by induction on the number of incomplete operations in P, that is, operations that have not yet been started or operations that have been started but have not yet completed. If no operation is incomplete then P immediately terminates. If n operations are incomplete then the scheduler has already or will start an operation by executing one of the threads. By the definition of lock-freedom, some operation will complete independently of the choices of the scheduler. So after a finite number of steps, we reach a point in which only $n-1$ incomplete operations are left. The termination argument follows by induction.

To prove the other direction, assume now that every program $P \in \mathcal{P}$ terminates. Furthermore, assume for the sake of contradiction that D is not lock-free. Then there exists some concurrent program P_∞ that only executes operations $op \in \{\pi_1, \ldots, \pi_k\}$ and an execution trace $\mathcal{T}$ of P_∞ in which some operations have started but no operation ever completes. It follows that P_∞ diverges and $\mathcal{T}$ is therefore infinite. Let n be the number of threads in P_∞ and let s_i be the sequential program that consists of all operations that have been started by thread i in the execution trace $\mathcal{T}$ in their temporal order. Then the program $||_{i=1,\ldots,n}\, s_i \in \mathcal{P}^n$ can be scheduled to produce the infinite execution trace $\mathcal{T}$. This contradicts the assumption that every program in $\mathcal{P}$ terminates. □

III. Quantitative Reasoning to Prove Lock-Freedom

A key insight of our work is that for many lock-free data structures, it is possible to give an upper bound on the total number of loop iterations in the programs in $\mathcal{P}$ (§II).

To see why, note that most non-blocking operations are based on the same optimistic approach to concurrency. They

repeatedly try to access or modify a shared-memory data structure until they can complete their operation without interference by another thread. However, lock-freedom ensures that such interference is only possible if another operation successfully makes progress:

> In an operation of a lock-free data structure, the failure of a thread to make progress is always caused by successful progress in an operation executed by another thread.

This property is the basis of a novel reasoning technique that we call a *quantitative compensation scheme*. It ensures that a thread is compensated for loop iterations that are caused by progress—often the successful completion of an operation—in another thread. In return, when a thread makes progress (e.g., completes an operation), it compensates the other threads. In this way, every thread is able to "pay" for its loop iterations without being aware of the other threads or the scheduler.

Consider for example Treiber's stack and a program P_n in which every thread only executes one operation, that is, $P_n = \|_{i=1,\ldots,n}\, s_i$ and $s_i \in \{\mathit{push}, \mathit{pop}\}$. An execution of P_n never performs more than n^2 loop iterations. Using a compensation scheme, this bound can be verified in a local and modular way. Assume that each of the threads has a number of tokens at its disposal and that each loop iteration in the program costs one token. After paying for the loop iteration, the token disappears from the system. Because it is not possible to create or duplicate tokens—tokens are an *affine resource*—the number of tokens that are initially present in the system is an upper bound on the total number of loop iterations executed.

Unfortunately, the maximum number of loop iterations performed by a thread depends on the choices of the scheduler as well as the number of operations that are performed by the other threads. To still make possible local and modular reasoning, we define a compensation scheme that enables the threads to exchange tokens. Since each loop iteration in P_n is guarded by a *CAS* operation this compensation scheme can be conveniently integrated into the specification of *CAS*. To this end, we require that (logically) $n-1$ tokens have to be available to execute a *CAS*.

(a) If the *CAS* is successful then it returns *true* and (logically) 0 tokens. Thus, the executing thread loses $n-1$ tokens.
(b) If the *CAS* is unsuccessful then it returns *false* and (logically) n tokens. Thus, the executing thread gains a token that it can use to pay for its next loop iteration.

The idea behind this compensation scheme is that every thread needs n tokens to perform a data structure operation. One token is used to pay for the first loop iteration and $n-1$ tokens are available during the loop as the loop invariant. If the *CAS* operation of a thread A is successful (case (a)) then this can cause at most $n-1$ *CAS* operations in the other threads to fail. These $n-1$ failed *CAS* operations need to return one token more than they had prior to their execution (case (b)). On the other hand, the successful thread A does not need its tokens anymore since it will exit the do-while loop. Therefore the $n-1$ tokens belonging to A are passed to the other $n-1$ threads to pay for the worst-case scenario in which this update causes $n-1$ more loop iterations.

If the *CAS* operation of a thread A fails (case (b)), then some other thread successfully updated the stack (case (a)) and thus provided a token for thread A. Since A had $n-1$ tokens before the execution of the *CAS*, it has n tokens after the execution. So thread A can pay a token for the next loop iteration and maintain its loop invariant of $n-1$ available tokens.

In our example program P_n, there are n^2 many tokens in the system at the beginning of the execution. So the number of loop iterations is bounded by n^2 and the program terminates.[1] More generally, we can use the same local and modular reasoning to prove that every program with n threads such that thread i executes m_i operations performs at most $\sum_{1\le i\le n} m_i \cdot n$ loop iterations. Thread i then starts with $m_i \cdot n$ tokens.

We will show in the following that this quantitative reasoning can be directly incorporated in total correctness proofs for these programs. We use the exact same techniques (for proving safety properties [18]) to prove liveness properties; namely *concurrent separation logic*, *auxiliary variables*, and *read permissions*. The only thing we add to separation logic is the notion of a *token* or a resource following Atkey [19].

IV. Preliminary Explanations

Before we formalize the proof outlined in §III, we give a short introduction to separation logic, quantitative reasoning, and concurrent separation logic. For the reader unfamiliar with the separation logic extensions of permissions and auxiliary variables, see the companion TR [22] and the relevant literature [20], [21]. Our full logic is defined in the TR [22].

Separation Logic: Separation logic [23], [24] is an extension of Hoare logic [25] that simplifies reasoning about shared mutable data structures and pointers. As in Hoare logic, programs are annotated with Hoare triples using predicates $P, Q, \ldots$ over program states (heap and stack). A *Hoare triple* $[P]\,C\,[Q]$ for a program C is a total-correctness specification of C that expresses the following. If C is executed in a program state that satisfies P then C safely terminates and the execution results in a state that satisfies Q.

In addition to the traditional logical connectives, predicates of separation logic are formed by logical connectives that enable local and modular reasoning about the heap. The *separating conjunction* $P * Q$ is satisfied by a program state if the heap of that state can be split in two disjoint parts such that one sub-heap satisfies P and one sub-heap satisfies Q. It enables the safe use of the frame rule

$$\frac{[P]\,C\,[Q]}{[P * R]\,C\,[Q * R]}\ (\textsc{Frame})$$

With the frame rule it is possible to specify only the part of the heap that is modified by the program C (using predicates P and Q). This specification can then be embedded in a larger proof to state that other parts of the heap are not changed (predicate R).

[1] In fact there are at most $\binom{n}{2}$ loop iterations in the worst case. However, the n^2 bound is sufficient to prove termination.

Quantitative Reasoning: Being based on the logic of bunched implications [26], separation logic treats heap cells as *linear resources* in the sense of linear logic. It is technically unproblematic to extend separation logic to reason about *affine consumable resources* too [19]. To this end, the logic is equipped with a special predicate $\lozenge$, which states the availability of one consumable resource, or *token*. The predicate is affine because it is satisfied by every state in which *one or more tokens* are available. This in contrast to a linear predicate like $E \mapsto F$ that is only satisfied by heaps H with $|\mathrm{dom}(H)| = 1$.

Using the separating conjunction $\lozenge * P$, it is straightforward to state that two or more tokens are available. We define $\lozenge^n$ to be an abbreviation for n tokens $\lozenge * \ldots * \lozenge$ that are connected by the separating conjunction $*$.

Since we use consumable resources to model the terminating behavior of programs, the semantics of the while command are extended such that a single token is consumed, if available, at the beginning of each iteration. Correspondingly, the derivation rule for while commands ensures that a single token is available for consumption on each loop iteration and thus that the loop will execute safely:

$$\frac{P \wedge B \implies P' * \lozenge \qquad I \vdash [P']\, C\, [P]}{I \vdash [P]\ \textit{while}\ B\ \textit{do}\ C\, [P \wedge \neg B]}\ (\textsc{While})$$

The loop body C must preserve the loop invariant P under the weakened precondition P'. C is then able to execute under the assumption that one token has been consumed and still restore the invariant P, thus making a token available for possible future loop iterations.

The tokens $\lozenge$ can be freely mixed with other predicates using the usual connectives of separation logic. For instance, the formula $x \mapsto 10 \vee (x \mapsto \_ * \lozenge)$ expresses that the heap-cell referred to by the variable x points to 10, or the heap-cell points to an arbitrary value and a token is available. Together with the frame rule, the tokens enable modular reasoning about quantitative resource usage.

Concurrent Separation Logic: Concurrent separation logic (CSL) is an extension of separation logic that is used to reason about concurrent programs [14]. The idea is that shared memory regions are associated with a *resource invariant*. Each atomic block that modifies the shared region can assume that the invariant holds at the beginning of its execution and must ensure that the invariant holds at the end of the atomic block.

The original presentation of CSL [14] uses *conditional critical regions (CCRs)* for shared variables. In this article, we follow Parkinson et al. [18] and assume a global shared region with one invariant so as to simplify the syntax and the logic. An extension to CCRs is possible. For predicates I,P, and Q, the judgment $I \vdash [P]\, C\, [Q]$ states that under the global resource invariant I, in a state where P holds, the execution of the concurrent program C is safe and terminates in a state that satisfies Q.

Concurrent execution of C_1 and C_2 is written $C_1 \parallel C_2$. We assume that shared variables are only accessed within atomic regions using the command *atomic(C)* and that atomic blocks are not nested. An interpretation of the resource invariant I is that it specifies a part of the heap owned by the shared region. The logical rule ATOM for the command *atomic* transfers the ownership of this heap region to the executing thread.

$$\frac{\mathit{emp} \vdash [P * I]\, C\, [Q * I]}{I \vdash [P]\ \textit{atomic}\{C\}\, [Q]}\ (\textsc{Atom})$$

Because the *atomic* construct ensures mutual exclusion, it is safe to share I between two programs that run in parallel. Pre- and post-conditions of concurrent programs are however combined by use of the separating conjunction[2]:

$$\frac{I \vdash [P_1]\, C_1\, [Q_1] \qquad I \vdash [P_2]\, C_2\, [Q_2]}{I \vdash [P_1 * P_2]\, C_1 \parallel C_2\, [Q_1 * Q_2]}\ (\textsc{Par})$$

Most of the other rules of sequential separation logic can be used in CSL by just adding the (unmodified) resource invariant I to the rules. The invariant is only used in the rule ATOM.

A technical detail that is crucial for the soundness of the classic rule for conjunction [25] is that we require the resource invariant I to be *precise* [14] with respect to the heap [27]. This means that, for a given heap H and stack V, there is at most one sub-heap $H' \subseteq H$ such that the sate (H', V) satisfies I. All invariants we use in this article are precise. Note that precision with respect to the resource tokens $\lozenge$ is not required since they are affine and not linear entities.

V. Formalized Quantitative Reasoning

In the following, we show how quantitative concurrent separation logic can be used to formalize the quantitative compensation scheme that we exemplified with Treiber's non-blocking stack in §III. The most important rules of this logic are described in §IV. The logic is formally defined and proved sound in in the companion TR [22].

Before we verify realistic non-blocking data structures, we describe the formalized quantitative reasoning for a simple *producer and consumer* example.

Producer and Consumer Example: In the example in Figure 3, we have a heap location B that is shared between a number of producer and consumer threads. A producer checks whether B contains the integer 0 (i.e., B is empty). If so then it updates B with a newly produced value and terminates. Otherwise, it leaves B unchanged and terminates. A consumer checks whether B contains a non-zero integer (i.e, B is non-empty). If so then it consumes the integer, sets the contents of B to zero, and loops to check if B contains a new value to consume. If B contains 0 then the consumer terminates.

If we verify this program using our quantitative separation logic then we prove that the number of tokens specified by the precondition is an upper bound on the number of loop iterations of the program. Since the number of specified tokens is always finite, we have thus proved termination.

The challenge in the proof is that the loop iterations of the operation *Consumer* depend on the scheduler and on the number of *Producer* operations that are executed in parallel. However, it is the case that a program that uses n *Consumer*

[2]We omit the variable side-conditions here for clarity. They are included in the full set of derivation rules in the companion TR [22].

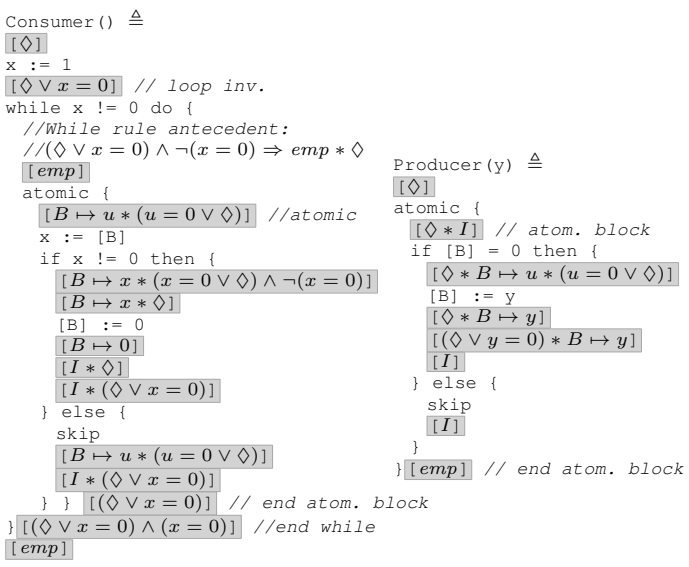

Fig. 3. A lock-free data structure B with the operations *Consumer* and *Producer*. The operation *Consumer* terminates if finitely many *Producer* operations are executed in parallel. The verification of lock-freedom and memory safety uses a compensation scheme and quantitative concurrent separation logic.

operations and m *Producer* operations performs at most $n+m$ loop iterations. We can prove this claim using our quantitative separation logic by deriving the following specifications.

$$I \vdash [\Diamond]\, \textit{Consumer()}\, [emp] \quad \text{and} \quad I \vdash [\Diamond]\, \textit{Producer(y)}\, [emp]$$

However, the modular and local specifications of these operations only hold in an environment in which all programs adhere to a certain policy. This policy can be expressed as a resource invariant I in the sense of concurrent separation logic. Intuitively, I states that the shared memory location B is read-writable, and either is empty ($B = 0$) or there is a token $\Diamond$ available. We define

$$I \triangleq \exists u.\, B \mapsto u * (u = 0 \vee \Diamond)\ .$$

Now we can read the specifications of *Consumer* and *Producer* as follows. The token $\Diamond$ in the precondition of *Consumer* is used to pay for the first loop iteration. More loop iterations are only possible if some producer updated the contents of heap location B to a non-zero integer v before the execution of the atomic block of *Consumer*. We then rely on the fact that the producer respected the resource invariant I. If $B \mapsto u$ and $u \neq 0$ then the only possibility of maintaining I is by providing a token $\Diamond$. The operation *Consumer* then updates B to zero and can thus establish the invariant I without using a token. So the token in the invariant becomes available to pay for the next loop iteration. Figure 3 contains an outline of the proof for *Producer* and *Consumer*. Note that our proof also verifies memory safety.

From Local Proofs to Lock-Freedom: Using the derived specifications of the operations and the frame rule, we inductively prove $I \vdash [\Diamond^k]\, op_1; \ldots; op_k\, [emp]$ where each op_i is a *Consumer* or *Producer* operation. In other words, we have then proved $[\Diamond^k]\, s\, [emp]$ for all $s \in \mathcal{S}^k$ (recall the definition from §II). Let now $s_i \in \mathcal{S}^{m_i}$ for $1 \leq i \leq n$. Using the rule PAR, we can then prove for $m = \sum_{i=1,\ldots,n} m_i$ that

$$I \vdash [\Diamond^m] \quad \Big|\Big|_{i=1,\ldots,n} s_i\, [emp]\ .$$

This shows that the program $||_{i=1,\ldots,n} s_i$ performs at most $m+1$ loop iterations (one token can be present in the resource invariant I) when it is executed. Following the discussion in §II, this proves that every program $p \in \mathcal{P}$ terminates and that $(B; \textit{Producer}, \textit{Consumer})$ is a lock-free data structure.

Similarly, we can in general derive a termination proof for every program in $\mathcal{P}$ from such specifications of the operations of a data structure. Assume that a shared-memory data structure $(S; \pi_1, \ldots, \pi_k)$ is given. Assume furthermore that we have verified for all $1 \leq i \leq k$ the specification $I(n) \vdash [\Diamond^{f(n)} * P]\, \pi_i\, [P]$. The notations $I(n)$ and $f(n)$ indicate that the proof can use a meta-variable n which ranges over $\mathbb{N}$. However, the proof is uniform for all n. Additionally, P might contain a variable tid for the thread ID. From this specification follows already the lock-freedom of S. To see why, we can argue as in the producer-consumer example. First, it follows for every n and $s \in \mathcal{S}^m$ that $I(n) \vdash [\Diamond^{m \cdot f(n)} * P]\, s\, [P]$. Second, a loop bound for $p = ||_{i=1,\ldots,n} s_i \in \mathcal{P}^n$ with $s_i \in \mathcal{S}^{m_i}$ is derived as follows. We use the rule PAR to prove for $m = \sum_{i=1,\ldots,n} m_i \cdot f(n)$ that

$$I(n) \vdash [\Diamond^m * \circledast_{0 \leq tid < n} P(tid)]\, p\, [\circledast_{0 \leq tid < n} P(tid)]\ .$$

Thus every $p \in \mathcal{P}$ terminates and according to the proof in §II, the data structure $(S; \pi_1, \ldots, \pi_k)$ is lock-free.

VI. LOCK-FREEDOM OF TREIBER'S STACK

We now formalize the informal proof of the lock-freedom of Treiber's stack that we described in §III. In in the TR [22], we outline how the proof can be easily extended to also verify memory safety. Figure 4 shows the implementation of Treiber's stack in the while language we use in this article.

Each thread that executes *push* or *pop* operations can be in one of two states. It either has some expectation on the contents of the shared data structure S (critical state) or it does not have any expectation (non-critical state). More concretely, a thread is in a critical state if and only if it is executing a *push* or *pop* operation and is in between the two atomic blocks in the while loop. The thread then expects that $t = [S]$. The resource invariant that we will formalize in quantitative CSL can be described as follows.

> For each thread T in the system one of the following holds. (1) The thread T is in a critical state and its expectation on the shared data structure is true. (2) The thread T is in a critical state and some other thread provided T with a token. (3) The thread T is in a non-critical state.

To formalize this invariant, we have to expose the local assumption of the threads ($t = [S]$) to the global state. This is why we use the auxiliary array A. If the thread with the thread ID tid is in a critical state then $A[tid]$ contains the value of its local variable t. Otherwise $A[tid]$ contains 0. Similarly, we have a second auxiliary array C such that $C[tid]$ contains a non-zero integer if and only if the thread with ID tid is in a

```
S := alloc(1); [S] := 0;
A := alloc(max_tid); C := alloc(max_tid);

push(v) ≜
 pushed := false;
 x := alloc(2);
 [x] := v;
 [(pushed ∨ ◇^n) * γ_r(tid,_,_)] // loop invariant
 while ( !pushed ) do {
  //While rule antecedent:
  ((pushed ∨ ◇^n) * γ_r(tid,_,_)) ∧ !pushed ⇒ ◇^{n-1} * γ_r(tid,_,_) * ◇
  [◇^{n-1} * γ_r(tid,_,_)]
  atomic {
   [◇^{n-1} * γ_r(tid,_,_) * S ↦ u * α(tid,u) * I'(tid,u)] // atom block
   [◇^{n-1} * γ(tid,_,_) * S ↦ u * I'(tid,u)] // impl. & read perm.
   t := [S];
   [◇^{n-1} * γ(tid,_,_) * (S ↦ u ∧ t = u) * I'(tid,u)] // read & frame
   C[tid] := 1
   [◇^{n-1} * γ(tid,_,1) * (S ↦ u ∧ t = u) * I'(tid,u)] // assignment
   A[tid] := t
   [◇^{n-1} * (A[tid] ↦ t ∧ t = u) * C[tid] ↦ 1 * S ↦ u * I'(tid,u)]
   [◇^{n-1} * γ_r(tid,t,1) * S ↦ u * α(tid,u) * I'(tid,u)] // perm.
   [◇^{n-1} * γ_r(tid,t,1) * I] // exist. intro & (3)
  };
  [◇^{n-1} * γ_r(tid,t,1)] // atomic block & frame
  // [x+1] := t; this is not essential for lock-freedom
  atomic {
   [◇^{n-1} * γ_r(tid,t,1) * I] // atomic block
   [◇^{n-1} * γ_r(tid,t,1) * S ↦ u * ⊛_{1≤i≤n} α(i,u)] // exist. elim.
   s := [S]; if s == t then {
    [◇^{n-1} * γ(tid,_,_) * S ↦ t * ⊛_{{1,...,n}\{tid}} (γ(i,_,_))]
    [S] := x;
    [γ(tid,_,_) * S ↦ x * I'(tid,x)] // permissions & (4)
    pushed := true
    [(pushed ∨ ◇^n) * γ(tid,_,_) * ∃u. S ↦ u * I'(tid,u)]
   } else {
    [◇^{n-1} * t ≠ u ∧ γ_r(tid,t,1) * α(tid,u) * S ↦ u * I'(tid,u)]
    [◇^n * γ(tid,t,1) * S ↦ u * I'(tid,u)] // impl. using (5)
    skip
    [(pushed ∨ ◇^n) * γ(tid,_,_) * ∃u. S ↦ u * I'(tid,u)]
   };
   C[tid] := 0
   [(pushed ∨ ◇^n) * γ(tid,_,0) * S ↦ u * I'(tid,u)]
   // write & exist. elim (above) and permissions & impl.
   [(pushed ∨ ◇^n) * γ_r(tid,_,_) * α(tid,u) * S ↦ u * I'(tid,u)]
   [(pushed ∨ ◇^n) * γ_r(tid,_,_) * I] // exist. intro
  };
  [(pushed ∨ ◇^n) * γ_r(tid,_,_)] // atomic block end
 }
```

Fig. 4. An implementation of the *push* operation of Treiber's lock-free stack in our language and the verification of the while loop. The CAS operation is implemented using an atomic block that updates the local variable *pushed*. The auxiliary array *A* contains in *A[tid]* the value of the local variable *t* of the thread with ID *tid* or zero if the thread does not assume *t = [S]*. The loop invariant $pushed \vee \Diamond^n$ states that either the new element x has been pushed to the stack S or there are n tokens available. The predicates γ and γ_r are defined in (1).

critical state. As shown in Figure 4, the arrays A and C are never used on the right-hand side of an assignment and are only updated in the two atomic blocks of each operation.

Let n be the number of threads in the system. We define

$$I \triangleq \exists u.\ S \mapsto u * \circledast_{0 \le i < n} \alpha(i,u)$$

$$\alpha(i,u) \triangleq \exists a, c.\ C[i] \overset{r}{\mapsto} c * A[i] \overset{r}{\mapsto} a * (c = 0 \vee a = u \vee \Diamond)$$

The resource invariant I states that the shared region has a full permission for the heap location S that points to the value u. Additionally, the predicate $\alpha(i,u)$ states for each thread i that the shared region has read permissions for $C[i]$ and $A[i]$; and that thread i is in a non-critical section ($c = 0$), that the local variable t contains the value $[S]$ ($a = u$), or that there is a token $\Diamond$ available.

We use read permissions since threads need access to the local predicate $A[tid] \overset{r}{\mapsto} t$ at some point to infer that $A[tid]$ contains the value of the local variable t. This relation of the local variable t with the array A is the only technical difficulty in the proof. Just as in safety proofs, we can now use the rules of our quantitative concurrent separation logic to verify the following Hoare triples.

$$I \vdash [\gamma_r(tid,\_,\_) * \Diamond^n]\ \ \textit{push(v)}\ \ [\gamma_r(tid,\_,\_)]$$
$$I \vdash [\gamma_r(tid,\_,\_) * \Diamond^n]\quad \textit{pop()}\quad [\gamma_r(tid,\_,\_)]$$

Where γ and γ_r are defined as:

$$\gamma(t,a,c) \triangleq A[t] \mapsto a * C[t] \mapsto c \tag{1}$$

$$\gamma_r(t,a,c) \triangleq A[t] \overset{r}{\mapsto} a * C[t] \overset{r}{\mapsto} c \tag{2}$$

Thus, the execution of any operation requires n tokens and read permission to the heap locations $A[tid]$ and $C[tid]$. After execution, the tokens are consumed and we are left with the read permissions. We use the following abbreviation for parts of the invariant I that are not needed in the local proofs.

$$I'(j,u) \triangleq \circledast_{i \in \{0,\ldots,n-1\}\setminus\{j\}} \alpha(i,u)$$

We have for all values u and $j \in \{0,\ldots,n-1\}$ that

$$I = \exists u.\ S \mapsto u * \alpha(j,u) * I'(j,u) \tag{3}$$

$$\Diamond^{n-1} \circledast_{i \in \{0,\ldots,n-1\}\setminus\{j\}} (\gamma_r(i,\_,\_)) \implies I'(j,u) \tag{4}$$

$$t \neq u \wedge \gamma_r(j,t,1) * \alpha(j,u) \implies \Diamond * \gamma(j,t,1) \tag{5}$$

Using these assertions, the verification of *push* and *pop* is a straightforward application of the rules of our logic. Figure 4 describes the main part—the while loop—of the proof of *push*. The loop invariant $pushed \vee \Diamond^n$ states that either the new element x has been pushed onto the stack S or there are n tokens available. In the first atomic block we leave the assumptions $I'(tid,u)$ of the other thread untouched and just establish $A[tid] \overset{r}{\mapsto} t * C[tid] \overset{r}{\mapsto} 1$.

The key aspect of the proof is the second atomic block which corresponds to the CAS operation in the original code. In the *if* case, we possibly break the assumptions of the other threads (*[S] := x*). Then we have to use $n-1$ tokens and implication (4) to re-establish $I'(tid,u)$. Since the variable *pushed* is set to *true*, we can maintain the loop invariant without using another token. In the *else* case we use the inequality $t \neq u$ and implication (5) to derive the loop invariant. Finally, we re-establish $\alpha(tid,u)$ using $C[tid] \mapsto 0$.

The verification of the while loop of *pop* is similar. By applying the proof from the end of §V to the specifications of *push* and *pop*, we have then proved the lock-freedom of Treiber's stack.

An interesting aspect of the proof is that it is not essential for a thread to know the entire resource invariant I. The only part that is needed is the implication $S \mapsto \_ * \Diamond^n * \circledast_{0 \le i < n} A[i] \overset{r}{\mapsto} \_ \implies I$. This can be used to make the assumptions $A(i)$ of the threads on the global data structure abstract. The implication $S \mapsto \_ * \Diamond^n * \circledast_{0 \le i < n} A[i] \overset{r}{\mapsto} \_ \implies$

$\exists u.\ S \mapsto \_ * \circledast_{0 \le i < n}((A[i] \overset{r}{\mapsto} \_ * \lozenge) \vee A(i,u))$ holds for all predicates $A(i,u)$. A natural candidate for such an abstraction is (concurrent) abstract predicates [28], [29]. However, such an abstraction is not needed for our goal of verifying non-blocking data structures in this paper.

VII. Advanced Lock-Free Data Structures

In this section we investigate how our quantitative proof technique can be used to prove the lock-freedom of more complex shared-memory data structures.

In many cases, it is possible to derive a bound on the total number of loop iterations like we do for Treiber's stack. Table 5 gives an overview of our findings. It describes for several different data structures the number $t(n)$ of tokens that are needed per operation in a system with n threads. The derived loop bound on a system with n threads that executes m operations is then $t(n)*m$. In the hazard-pointer data structures, the natural number ℓ is a fixed global parameter of the data structure. The details are discussed in the following.

Michael and Scott's Non-Blocking Queue: Michael and Scott's non-blocking queue [7] implements a FIFO queue using a linked list with pointers to the head and the tail. New nodes are inserted at the tail and nodes are removed from the head.

To implement the queue in a lock-free way, the insert operation can leave the data structure in an apparently inconsistent state: The new node is inserted at the tail using a CAS-guarded loop, similar to Treiber's stack. The pointer to the tail is then updated by a second CAS operation, allowing other threads to access the data structure with an inaccurate tail pointer.

To deal with this problem, the operations of the queue maintain the invariant that the tail pointer points to the last or second-to-last node during the execution and to the last node after the execution of the operation. To maintain this invariant, each CAS-guarded loop checks if the tail pointer points to a node whose next pointer is *Null*. In this case, the tail pointer is up to date and the current iteration of the while loop can continue. Otherwise, the tail pointer is updated to point to the last node of the list and the while loop is restarted.

To prove the lock-freedom of Michael and Scott's queue, we extend the invariant I that we used in the verification of Treiber's stack with an additional condition: The next pointer of the node pointed to by the tail pointer is *Null* or there is a token that can be used to pay for an additional loop iteration.

$$\exists u, t, w.\mathit{heap} \mapsto u * \mathit{tail} \mapsto t * \mathit{tail}+1 \mapsto w * \circledast_{0 \le i < n} \beta(i,u,t) * (w = \mathit{nil} \vee \lozenge)$$

The formulas $\beta(i,u,t)$ are analogous to the formulas $\alpha(i,u)$ in the invariant that we used for the verification of Treiber's stack. With this invariant in a system with n threads, we can verify the operations of the queue using $n+1$ tokens in the respective preconditions.

Hazard Pointers: A limitation of Treiber's non-blocking stack is that it is only sound in the presence of garbage collection. This is due to the *ABA problem* (see for instance [8]) which appears in many algorithms that use compare-and-swap operations: Assume that a shared location which contains A is read by a thread t_1. Then thread t_2 gets activated by the scheduler, modifies the shared location to B, and then back to A. Now thread t_1 gets activated again, falsely assumes that the shared data has not been changed, and continues with its operation. The result can be a corrupted shared data structure, invalid memory access or an incorrect return value.

Data Structure	Tokens per Operation
Treiber's Stack [6]	n
Michael and Scott's Queue [7]	$n+1$
Hazard-Pointer Stack [8]	$n + (\ell \cdot n)$
Hazard-Pointer Queue [8]	$(n+1) + (\ell \cdot n)$
Elimination-Backoff Stack [9]	$n(n+1)$

Fig. 5. Quantitative reasoning for popular non-blocking data structures. The table shows the number $t(n)$ of tokens that are needed per operation in a system with n threads. The derived loop bound on a system with n threads that executes m operations is then $t(n) * m$. The natural number ℓ is a fixed global parameter of the data structure.

Michael [8] proposes *hazard pointers* to enable the safe reclamation of memory while maintaining the lock-freedom of non-blocking data structures. The idea is to introduce a global array that contains for each thread a number of pointers (in most cases just one) to data nodes that are currently in use by the thread. Additionally, each thread stores a local list of pointers that it wants to remove from the shared data structure (for instance by using *pop* in the case of a stack). After each successful removal of a node a thread checks if this local list has reached a fixed length threshold. If so, it checks the hazard pointers of each other thread to ensure that the pointers are not in use before reclaiming the space.

The use of hazard pointers does not affect the global resource invariants that we use in our quantitative verification technique. The reason is that hazard pointers only affect parts of the operations that are outside the loops guarded by CAS operations. Moreover, the worst-case number of loop iterations in this additional code can be easily determined: It is the maximal length ℓ of the local list multiplied with the maximal number of threads in the system.

For Treiber's stack with hazard pointers, the specifications of *push* and *pop* are:

$$I \vdash [\gamma_r(tid,\_,\_) * \lozenge^n] \quad \mathit{push(v)} \quad [\gamma_r(tid,\_,\_)]$$
$$I \vdash [\gamma_r(tid,\_,\_) * \lozenge^{n+(\ell*n)}] \quad \mathit{pop()} \quad [\gamma_r(tid,\_,\_)]$$

Where γ_r is defined as in (1). The invariant I is the same as in the specification of the version without hazard pointers.

Elimination Backoff: To improve the performance of Treiber's non-blocking stack in the presence of high contention, one can use an *elimination backoff scheme* [9]. The idea is based on the observation that a *push* operation followed by a *pop* results in a stack that is identical to the initial stack. So, if a stack operation fails because of the interference of another thread then the executing thread does not immediately retry the operation. Instead, it checks if there is another thread that is trying to perform a complementary operation. In this case, the two operations can be *eliminated* without accessing the

stack at all: The two threads use a different shared-memory cell to transfer the stack element.

Our method can also be used to prove that Hendler et al's elimination-backoff stack [9] is lock-free. The main challenge in the proof is that the *push* and *pop* operations consist of two *nested* loops guarded by CAS operations. Assume again a system with n threads. The inner loop can be treated as in Treiber's stack using n tokens in the precondition and 0 tokens in the postcondition. As a result, the number of tokens needed for an iteration of the outer loop is $n+1$. That means that a successful thread needs to transfer $(n-1)\cdot(n+1) = n^2-1$ tokens to the other threads to account for additional loop iterations in the other threads. Given this, we can verify the elimination-backoff stack using n^2 tokens in the precondition. Technically, we need an invariant of the form $I * J$, where I is an invariant like in Treiber's stack (for the inner loop) and J is like I but with every token $\Diamond$ replaced by $\Diamond^n$.

More details on the verification can be found in the TR [22].

Non-Blocking Maps and Sets: Quantitative compensation schemes can also be used to prove the lock-freedom of non-blocking maps and sets (e.g., [30], [31]).

As in other lock-free data structures, interference in map and set operations only occurs when another thread makes progress. For example, in Harris' non-blocking linked list [30], a thread will make an additional traversal (of the list) only if another thread causes interference by making a successful traversal. The number of these unsuccessful traversals can be bounded using a quantitative compensation scheme, as in our previous examples.

The number of loop iterations within each list traversal also depends on the length of the list. Nevertheless, it is possible to prove an upper bound on the number of loop iterations executed by programs in $\mathcal{P}$, since each of the n threads executes a fixed number m_i of operations. Thus the total number of operations is bounded by $m = \sum_{i=1,\ldots,n} m_i$. In many important shared-memory data structures, such as lists or maps, m (or a function of m) constitutes an upper bound on the size of the shared data structure. One can then use this bound to prove an upper bound on the number of loop iterations by introducing $\Diamond^m$ in the global resource invariant. Like Atkey [19], we can use ideas from amortized resource analysis [32] to deal with variable-size data structures. By assigning tokens to each element of a data structure we derive bounds that depend on the size of the data structure without explicitly referring to its size. For instance, an inductive list-predicate that states that $k\cdot|\ell|$ tokens are available, where ℓ is the list pointed to by u can be defined as follows.

$$\begin{aligned} LSeg'(x,y,k) \Leftrightarrow & (x = y \wedge emp) \vee \\ & (\exists v, z \;\; x \mapsto v * x+1 \mapsto z * LSeg'(z,y,k) * \Diamond^k) \end{aligned}$$

VIII. Related Work

There is a large body of research on verifying safety properties and partial correctness of non-blocking data structures. See for instance [18], [33], [34] and the references therein. This work deals with the verification of the complementary liveness property of being lock-free, which in comparison has received little attention.

Colvin and Dongol [10], [35] use manually-derived global well-founded orderings and temporal logic to prove the lock-freedom of Treiber's stack [6], Michael and Scott's queue [7], and a bounded array-based queue. Their technique is a non-modular whole program analysis of the most general client of the data structure. It is unclear if the approach applies to data-structure operations with nested loops. In contrast, our method is modular, can handle with nested loops, and does not require temporal reasoning.

Petrank et al. [11] endeavor to reduce lock-freedom to a safety property by introducing the more restrictive concept of bounded lock-freedom. It states that, in a concurrent program, there has to be progress after at most k steps, where k depends on the input size of the program but not on the number of threads in the system. They verify bounded lock-freedom with a whole program analysis using temporal logic and the model checker Chess. The technique is demonstrated by verifying a simple concurrent program that uses Treiber's stack. Our compensation-based quantitative reasoning does not provide an explicit bound on the steps between successful operations but rather a global bound on the number of loop iterations in the system. Additionally, our bound depends on the number of threads in the system and not on the size of the input. A conceptual difference of our work is that we prove the lock-freedom of a given data structure as opposed to the verification of a specific program. Moreover, our proofs are local and modular, and not a whole program analysis. We also show that compensation-based reasoning works for many advanced lock-free data structures.

Gotsman et al. [13] reduce lock-freedom proofs to termination proofs of programs that execute n data structure operations in parallel. They then prove termination using separation logic and temporal rely-guarantee reasoning by layering liveness reasoning on top of a circular safety proof. Using several tools and manually formulating appropriate proof obligations, they are able to automatically verify the lock-freedom of involved algorithms such as Hendler et al.'s non-blocking stack with elimination backoff [9]. While these automation results are very impressive, the used reduction to termination is not intended to be applied to shared data structures that use thread IDs or other system information (see §II for details). In comparison, our compensation reasoning does not restrict the use of thread IDs or other system information. According to the authors (personal communication), the termination proofs of [13] would also work for a modification of the reduction that we used here.

Tofan et al. [12] describe a fully-mechanized technique based on temporal logic and rely-guarantee reasoning that is similar to the work of Gotsman et al. However, they assume weak fairness of the scheduler while we do not pose any restriction on the scheduler. Kobayashi and Sangiorgi [36] propose a type system that proves lock-freedom for programs written in the π-calculus. The target language and examples seem however to be quite different from the programs we prove lock-free in this article.

IX. Conclusion

We have shown that lock-freedom proofs of shared-memory data structures can be reduced to safety proofs in concurrent separation logic (CSL). To this end, we proposed a novel quantitative compensation scheme which can be formalized in CSL using a predicate $\Diamond$ for affine tokens. While similar logics have been used to verify the resource consumption of sequential programs [19], this is the first time that a quantitative reasoning method has been used to verify liveness properties of concurrent programs.

In the future, we plan to investigate the extent to which quantitative reasoning can be applied to other liveness properties of concurrent programs. The quantitative verification of *wait-freedom* seems to be similar to the verification of lock-freedom if we require that tokens cannot be transferred among the threads. *Obstruction-freedom* might require the creation of tokens in case of a conflict. We also plan to adapt our method to reason about progress properties of locking data structures, such as *fairness and starvation-freedom*. These properties are more challenging to verify with our quantitative method since they rely on a fair scheduler, whereas non-blocking algorithms do not. To enable such proofs, we plan to extend our compensation scheme to include the behavior of the scheduler.

Ultimately, we envision integrating our compensation-based proofs into a logic for termination-sensitive contextual refinement.

Acknowledgments

We thank James Aspnes and Alexey Gotsman for helpful discussions. This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0910670 and 0915888 and 1065451. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] W. N. Scherer III, D. Lea, and M. L. Scott, "Scalable Synchronous Queues," *Commun. ACM*, vol. 52, no. 5, pp. 100–111, 2009.

[2] N. Shavit, "Data Structures in the Multicore Age," *Commun. ACM*, vol. 54, no. 3, pp. 76–84, 2011.

[3] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[4] M. Herlihy, "Wait-Free Synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[5] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-Free Synchronization: Double-Ended Queues as an Example," in *23rd Int. Conf. on Distributed Comp. Systems (ICDCS'03)*, 2003.

[6] R. K. Treiber, "Systems Programming: Coping with Parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, 1986.

[7] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in *15th Symp. on Principles of Distributed Computing (PODC'96)*, 1996, pp. 267–275.

[8] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004.

[9] D. Hendler, N. Shavit, and L. Yerushalmi, "A Scalable Lock-Free Stack Algorithm," in *16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*, 2004, pp. 206–215.

[10] R. Colvin and B. Dongol, "Verifying Lock-Freedom Using Well-Founded Orders," in *Theoretical Aspects of Computing - 4th International Colloquium (ICTAC'07)*, 2007, pp. 124–138.

[11] E. Petrank, M. Musuvathi, and B. Steesngaard, "Progress Guarantee for Parallel Programs via Bounded Lock-Freedom," in *Conf. on Prog. Lang. Design and Impl. (PLDI'09)*, 2009, pp. 144–154.

[12] B. Tofan, S. Bäumler, G. Schellhorn, and W. Reif, "Temporal Logic Verification of Lock-Freedom," in *Mathematics of Prog. Construction, 10th Int. Conf., (MPC'10)*, 2010, pp. 377–396.

[13] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis, "Proving that Non-Blocking Algorithms Don't Block," in *36th Symp. on Principles of Prog. Lang. (POPL'09)*, 2009, pp. 16–28.

[14] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theor. Comput. Sci.*, vol. 375, no. 1-3, pp. 271–307, 2007.

[15] V. Vafeiadis and M. J. Parkinson, "A Marriage of Rely/Guarantee and Separation Logic," in *Concurrency Theory, 18th Int. Conference (CONCUR'07)*, 2007, pp. 256–271.

[16] X. Feng, "Local Rely-Guarantee Reasoning," in *36th Symp. on Principles of Prog. Langs. (POPL'09)*, 2009, pp. 315–327.

[17] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent Abstract Predicates," in *Object-Oriented Programming, 24th European Conf. (ECOOP'10)*, 2010, pp. 504–528.

[18] M. Parkinson, R. Bornat, and P. O'Hearn, "Modular Verification of a Non-Blocking Stack," in *34th Symp. on Principles of Prog. Lang. (POPL'07)*, 2007, pp. 297–302.

[19] R. Atkey, "Amortised Resource Analysis with Separation Logic," in *19th European Symposium on Programming (ESOP'10)*, 2010, pp. 85–103.

[20] S. S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Commun. ACM*, vol. 19, no. 5, pp. 279–285, 1976.

[21] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission Accounting in Separation Logic," in *32nd Symp. on Principles of Prog. Lang. (POPL'05)*, 2005, pp. 259–270.

[22] J. Hoffmann, M. Marmar, and Z. Shao, "Quantitative Reasoning for Proving Lock-Freedom," Yale University, Tech. Rep. YALEU/DCS/TR-1476, 2013, http://cs-www.cs.yale.edu/homes/hoffmann/papers/lockfree2013-tr.pdf.

[23] S. S. Ishtiaq and P. W. O'Hearn, "BI as an Assertion Language for Mutable Data Structures," in *28th Symp. on Principles of Prog. Lang. (POPL'01)*, 2001, pp. 14–26.

[24] J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *17th IEEE Symp on Logic in Computer Science (LICS'02)*, 2002, pp. 55–74.

[25] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[26] P. W. O'Hearn and D. J. Pym, "The Logic of Bunched Implications," *Bulletin of Symbolic Logic*, vol. 5, no. 2, pp. 215–244, 1999.

[27] V. Vafeiadis, "Concurrent Separation Logic and Operational Semantics," *Electr. Notes Theor. Comput. Sci.*, vol. 276, pp. 335–351, 2011.

[28] M. J. Parkinson and G. M. Bierman, "Separation Logic and Abstraction," in *32nd Symp. on Principles of Prog. Lang. (POPL'05)*, 2005, pp. 247–258.

[29] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent Abstract Predicates," in *Object-Oriented Programming, 24th European Conf. (ECOOP'10)*, 2010, pp. 504–528.

[30] T. L. Harris, "A Pragmatic Implementation of Non-blocking Linked-Lists," in *15th International Conf. on Distributed Computing (DISC'01)*, 2001, pp. 300–314.

[31] M. Greenwald, "Non-blocking Synchronization and System Design," Ph.D. dissertation, Stanford University, 1999, tR STAN-CS-TR-99-1624.

[32] M. Hofmann and S. Jost, "Static Prediction of Heap Space Usage for First-Order Functional Programs," in *30th Symp. on Principles of Prog. Lang. (POPL'03)*, 2003, pp. 185–197.

[33] V. Vafeiadis, "Modular Fine-Grained Concurrency Verification," Ph.D. dissertation, University of Cambridge, 2007, tR UCAM-CL-TR-726.

[34] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang, "Reasoning about Optimistic Concurrency Using a Program Logic for History," in *Concurrency Theory, 21th Int. Conference (CONCUR'10)*, 2010, pp. 388–402.

[35] R. Colvin and B. Dongol, "A General Technique for Proving Lock-Freedom," *Sci. Comput. Program.*, vol. 74, no. 3, pp. 143–165, 2009.

[36] N. Kobayashi and D. Sangiorgi, "A Hybrid Type System for Lock-Freedom of Mobile Processes," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 5, 2010.