Proving That Non-Blocking Algorithms Don't Block

Alexey Gotsman
University of Cambridge

Byron Cook
Microsoft Research

Matthew Parkinson
University of Cambridge

Viktor Vafeiadis
Microsoft Research

Abstract

A concurrent data-structure implementation is considered *non-blocking* if it meets one of three following liveness criteria: *wait-freedom*, *lock-freedom*, or *obstruction-freedom*. Developers of non-blocking algorithms aim to meet these criteria. However, to date their proofs for non-trivial algorithms have been only manual pencil-and-paper semi-formal proofs. This paper proposes the first fully automatic tool that allows developers to ensure that their algorithms are indeed non-blocking. Our tool uses rely-guarantee reasoning while overcoming the technical challenge of sound reasoning in the presence of interdependent liveness properties.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Formal Verification, Concurrent Programming, Liveness, Termination

1. Introduction

Non-blocking synchronisation is a style of multithreaded programming that avoids the blocking inherent to lock-based mutual exclusion. Instead, alternative synchronisation techniques are used, which aim to provide certain progress guarantees even if some threads are delayed for arbitrarily long. These techniques are primarily employed by concurrent implementations of data structures, such as stacks, queues, linked lists, and hash tables (see, for example, the `java.util.concurrent` library). Non-blocking data structures are generally much more complex than their lock-based counterparts, but can provide better performance in the presence of high contention between threads [38].

An algorithm implementing operations on a concurrent data structure is considered *non-blocking* if it meets one of three commonly accepted liveness criteria that ensure termination of the operations under various conditions:

Wait-freedom [15]: Every running thread is guaranteed to complete its operation, regardless of the execution speeds of the other threads. Wait-freedom ensures the absence of livelock and starvation.

Lock-freedom [23]: From any point in a program's execution, some thread is guaranteed to complete its operation. Lock-freedom ensures the absence of livelock, but not starvation.

Obstruction-freedom [16]: Every thread is guaranteed to complete its operation provided it eventually executes in isolation. In other words, if at some point in a program's execution we suspend all threads except one, then this thread's operation will terminate.

The design of a non-blocking algorithm largely depends on which of the above three criteria it satisfies. Thus, algorithm developers aim to meet one of these criteria and correspondingly classify the algorithms as wait-free, lock-free, or obstruction-free (e.g., [14, 16, 25]). To date, proofs of the liveness properties for non-trivial cases have been only manual pencil-and-paper semi-formal proofs. This paper proposes the first fully automatic tool that allows developers to ensure that their algorithms are indeed non-blocking.

Reasoning about concurrent programs is difficult because of the need to consider all possible interactions between concurrently executing threads. This is especially true for non-blocking algorithms, in which threads interact in subtle ways through dynamically-allocated data structures. To combat this difficulty, we based our tool on rely-guarantee reasoning [18, 29], which considers every thread in isolation under some assumptions on its environment and thus avoids reasoning about thread interactions directly. Much of rely-guarantee's power comes from cyclic proof rules for safety; straightforward generalisations of such proof rules to liveness properties are unsound [1]. Unfortunately, in our application, we have to deal with interdependencies among liveness properties of threads in the program: validity of liveness properties of a thread can depend on liveness properties of another thread and vice versa. We resolve this apparent circularity by showing that (at least for all of the algorithms that we have examined) proofs can found that layer non-circular liveness reasoning on top of weak circular reasoning about safety. We propose a method for performing such proofs by repeatedly strengthening threads' guarantees using non-circular reasoning until they imply the required liveness property (Section 2). We develop a logic that allows us to easily express these layered proofs for heap-manipulating programs (Sections 3 and 4) and prove it sound with respect to an interleaving semantics (Section 6).

In addition, we have found that the rely and guarantee conditions needed for proving algorithms non-blocking can be of a restricted form: they need only require that certain events do not happen infinitely often. This allows us to automate proving the liveness properties by a procedure that systematically searches for proofs in our logic with relies and guarantees of this form (Section 5).

Using our tool, we have automatically proved a number of the published algorithms to be formally non-blocking, including challenging examples such as the HSY stack [14] and Michael's linked list algorithm [25]. Proofs for some of the verified algorithms require complex termination arguments and supporting safety properties that are best constructed by automatic tools.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'09, January 18–24, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-379-2/09/01...$5.00

2. Informal development

We start by informally describing our method for verifying liveness properties and surveying the main results of the paper.

Example. Figure 1 contains a simple non-blocking implementation of a concurrent stack due to Treiber [33], written in a C-like language. A client using the implementation can call several `push` or `pop` operations concurrently. To ensure the correctness of the algorithm, we assume that it is executed in the presence of a garbage collector (see [17, Section 10.6] for justification). We also assume that single word reads and writes are executed atomically. The stack is stored as a linked list, and is updated by compare-and-swap (CAS) instructions. CAS takes three arguments: a memory address, an expected value and a new value. It atomically reads the memory address and updates it with the new value when the address contains the expected value; otherwise, it does nothing. In C syntax this might be written as follows:

```
int CAS(WORD *addr, WORD v1, WORD v2) {
  atomic {
    if (*addr == v1) { *addr = v2; return 1; }
    else { return 0; }
  }
}
```

In most architectures an efficient CAS (or an equivalent operation) is provided natively by the processor.

The operations on the stack are implemented as follows. The function `init` initialises the data structure. The `push` operation (*i*) allocates a new node `x`; (*ii*) reads the current value of the top-of-the-stack pointer `S`; (*iii*) makes the `next` field of the newly created node point to the read value of `S`; and (*iv*) atomically updates the top-of-the-stack pointer with the new value `x`. If the pointer has changed between (*ii*) and (*iv*) and has not been restored to its initial value, the CAS fails and the operation is restarted. The `pop` operation is implemented in a similar way.

Liveness properties of non-blocking algorithms. Notice that a `push` or `pop` operation of Treiber's stack may not terminate if other threads are continually modifying `S`: in this case the CAS instruction may always fail, which will cause the operation to restart continually. Thus, the algorithm is not wait-free. However, it is lock-free: if `push` and `pop` operations execute concurrently, some operation will always terminate.

We note that an additional requirement in the definitions of the liveness properties given in Section 1 is that the properties have to be satisfied under any scheduler, including an unfair one that suspends some threads and never resumes them again: in this case the remaining threads still have to satisfy the liveness properties. The properties form a hierarchy [10]: if an algorithm is wait-free, it is also lock-free, and if it is lock-free, it is also obstruction-free. Note also that even the weakest property, obstruction-freedom, prevents the use of spinlocks, because if a thread has acquired a lock and is then suspended, another thread may loop forever trying to acquire that lock.

We first describe our approach for verifying lock-freedom.

Reducing lock-freedom to termination. We show that the checking of lock-freedom can be reduced to the checking of termination in the spirit of [36]. Consider a non-blocking data structure with operations $\mathtt{op}_1, \ldots, \mathtt{op}_n$. Let `op` be the command that non-deterministically executes one of the operations on the data structure with arbitrary parameters:

$$\mathtt{op} = \mathtt{if}\ (\mathtt{nondet}())\ \mathtt{op}_1;\ \mathtt{else\ if}\ (\mathtt{nondet}())\ \mathtt{op}_2;\ \ldots\ \mathtt{else}\ \mathtt{op}_n; \tag{2.1}$$

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

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));
}
```

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

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;
}
```

Figure 1. Treiber's non-blocking stack

We denote non-deterministic choice with `nondet()`. The definition of lock-freedom of the data structure requires that for all m in any (infinite) execution of the data structure's most general client $C(m)$ defined below, some operation returns infinitely often:

$$C(m) = \mathop{\Big\|}_{i=1}^{m} \mathtt{while}\ (\mathtt{true})\ \{\ \mathtt{op}\ \}$$

We now show that this is the case if and only if for all k the following program $C'(k)$ terminates:

$$C'(k) = \mathop{\Big\|}_{i=1}^{k} \mathtt{op} \tag{2.2}$$

The proof in the "only if" direction is by contrapositive: a non-terminating execution of $C'(k)$ can be straightforwardly mapped to an execution of $C(k)$ violating lock-freedom in which the `while` loops make at most one iteration executing the same operations with the same parameters as in $C'(k)$. For the "if" direction note that any infinite execution of $C(m)$ violating lock-freedom has only finitely many (say, k) operations started: those that complete successfully, those that are suspended by the scheduler and never resumed again, and those that do not terminate. Such an execution can then be mapped to a non-terminating execution of $C'(k)$, in which the operations are completed, suspended or non-terminating as above.

Thus, to check lock-freedom of an algorithm, we have to check the termination of an arbitrary number of its operations running in parallel.

Rely-guarantee reasoning and interference specifications. We prove termination of the program $C'(k)$ using rely-guarantee reasoning [18, 29]. Rely-guarantee avoids direct reasoning about all possible thread interactions in a concurrent program by specifying a relation (the *guarantee* condition) for every thread restricting how it can change the program state. For any given thread, the union of the guarantee conditions of all the other threads in the program (its *rely* condition) restricts how those threads can interfere with it, and hence, allows reasoning about this thread in isolation.

The logic we develop in this paper uses a variant of rely-guarantee reasoning proposed in RGSep [35]—a logic for reasoning about safety properties of concurrent heap-manipulating programs, which combines rely-guarantee reasoning with separation logic. RGSep partitions the program heap into several thread-local parts (each of which can only be accessed by a given thread) and the shared part (which can be accessed by all threads). The partitioning is defined by proofs in the logic: an assertion in the code of a thread restricts its local state and the shared state. Additionally, the

partitioning is dynamic, meaning that we can use ownership transfer to move some part of the local state into the shared state and vice versa. Rely and guarantee conditions are then specified with sets of actions, which are relations *on the shared state* determining how the threads change it. This is in contrast with the original rely-guarantee method, in which rely and guarantee conditions are relations *on the whole program state*. Thus, while reasoning about a thread, we do not have to consider local states of other threads.

For example, using RGSep we can prove memory safety (no invalid pointer dereferences) and data structure consistency (the linked list is well-formed) of Treiber's stack [34]. The proof considers the linked list with the head pointed to by the variable `S` to be in the shared state. When a `push` operation allocates a new node `x`, it is initially in its local state. The node is transferred to the shared state once it is linked into the list with a successful CAS instruction. The proof specifies interference between threads in the shared state with three actions, Push, Pop, and Id, with the following informal meaning: Push corresponds to pushing an element onto the stack (a successful CAS in `push`); Pop to removing an element from the stack (a successful CAS in `pop`); and Id represents the identity action that does not change the shared state (a failed CAS and all the other commands in the code of the threads).

Proving lock-freedom. Using the splitting of the heap into local and shared parts and the interference specification for Treiber's stack described above, we can establish its lock-freedom as follows. As we showed above, it is sufficient to prove termination of a fixed but arbitrary number of threads each executing a single `push` or `pop` operation with an arbitrary parameter. The informal proof of this (formalised in Section 4) is as follows:

I. *No thread executes* Push *or* Pop *actions infinitely often.*
This is because a Push or Pop action corresponds to a successful CAS, and once a CAS succeeds, the corresponding `while` loop terminates.

II. *The* `while` *loop in an operation terminates if no other thread executes* Push *or* Pop *actions infinitely often.*
This is because the operation does not terminate only when its CAS always fails, which requires the environment to execute Push or Pop actions infinitely often.

Hence, every thread terminates.

The above proof uses rely-guarantee reasoning: it consists of proving several *thread-local* judgements, each of which establishes a property of a thread under an assumption about the interference from the environment. Properties of a parallel composition of threads are then derived from the thread-local judgements. This is done by first establishing the guarantee provided by Statement I and then using it to prove termination of the operations. This pattern—establishing initial guarantees and then deriving new guarantees from them—is typical for proofs of lock-freedom. We now consider a more complicated example in which the proof consists of more steps of this form.

Hendler, Shavit, and Yerushalmi [14] have presented an improved version of Treiber's stack that performs better in the case of higher contention between threads. Figure 2 shows an adapted and abridged version of their algorithm. The implementation combines two algorithms: Treiber's stack and a so-called elimination scheme (partially elided). A `push` or a `pop` operation first tries to modify the stack as in Treiber's algorithm, by doing a CAS to change the shared top-of-the-stack pointer. If the CAS is successful then the operation terminates. If the CAS fails (because of interference from another thread), the operation backs off to the elimination scheme. If this scheme fails, the whole operation is restarted.

The elimination scheme works on data structures that are separate from the list implementing the stack. The idea behind it is that two contending `push` and `pop` operations can eliminate each other without modifying the stack if `pop` returns the value that `push` is trying to insert. An operation determines the existence of another operation it could eliminate itself with by selecting a random slot `pos` in the `collision` array, and atomically reading that slot and overwriting it with its thread identifier `MYID`. The identifier of another thread read from the array can be subsequently used to perform elimination. The corresponding code does not affect the lock-freedom of the algorithm and is therefore elided in Figure 2. The algorithm implements the atomic read-and-write operation on the `collision` array in a lock-free fashion using CAS[1]. This illustrates a common pattern, when one lock-free data structure is used inside another.

An RGSep safety proof of the HSY stack would consider the data structures of the elimination scheme shared and describe interference on the shared state using the actions introduced for Treiber's stack and two additional actions: Xchg (which denotes the effect of the successful operation on the `collision` array described above) and Others (which includes all the operations on the other data structures of the elimination scheme). Given this interference specification, the informal proof of lock-freedom of the algorithm is as follows: in a parallel composition of several threads each executing one `push` or `pop` operation,

I. *No thread executes* Push *or* Pop *actions infinitely often.*

II. `push` *and* `pop` *do not execute the* Xchg *action infinitely often if no other thread executes* Push *or* Pop *actions infinitely often.*
This is because a thread can only execute Xchg infinitely often if its outer `while` loop does not terminate. This can only happen if some other thread executes Push or Pop infinitely often.

III. `push` *and* `pop` *terminate if no other thread executes* Push*,* Pop*, or* Xchg *actions infinitely often.*
This is because in this case both inner and outer `while` loops eventually terminate.

From Statements I and II, we get that no thread executes Push, Pop, or Xchg actions infinitely often. Hence, by Statement III every thread terminates.

The above proof is done in a layered style, i.e., starting from the weak guarantee provided by Statement I and strengthening it using already established guarantees until it implies termination. This is informally illustrated in Figure 3 for the case of two operations ($\mathtt{op}_1$ and $\mathtt{op}_2$) running in parallel. The validity of the property of Thread 1 in the middle layer depends on the validity of the counterpart property of Thread 2 and vice versa. However, it is unsound to remove the upper layer of Figure 3 and justify the guarantee in the middle layer by circular reasoning, i.e., by observing that a thread satisfies the guarantee if the other thread does.

We have found that the proof method described above was applicable in all of the examples of lock-free algorithms that we have considered. In the next two sections we develop a logic for formalising proofs following the method.

Automating lock-freedom proofs. The above informal proofs of lock-freedom use guarantee conditions of a restricted form that specifies two sets of actions: those that a thread can execute and those that it cannot execute infinitely often. We have found that guarantee conditions of this form were sufficient to prove lock-freedom for all the examples we considered. This observation allows us to automate proving lock-freedom of an algorithm by systematically searching for termination proofs for a program consist-

[1] Such an operation could be implemented with an atomic exchange instruction. The reason for implementing it with CAS is that in some architectures the atomic exchange instruction is either not available or slow.

```
struct Node {
  value_t data;
  Node *next;
};
Node *S;
int collision[SIZE];

void push(value_t v) {
  Node *t, *x;
  x = new Node();
  x->data = v;
  while (1) {
    t = S;
    x->next = t;
    if (CAS(&S,t,x)) { return; }
    // Elimination scheme
    // ...
    int pos = GetPosition();
    // 0 ≤ pos ≤ SIZE-1
    int hisId = collision[pos];
    while (!CAS(&collision[pos],hisId,MYID)) {
      hisId = collision[pos];
    }
    // ...
  }
}
```

```
value_t pop() {
  Node *t, *x;
  while (1) {
    t = S;
    if (t == NULL) {
      return EMPTY;
    }
    x = t->next;
    if (CAS(&S,t,x)) {
      return t->data;
    }
    // Elimination scheme
    // ...
    int pos = GetPosition();
    // 0 ≤ pos ≤ SIZE-1
    int hisId = collision[pos];
    while (!CAS(&collision[pos],hisId,MYID)) {
      hisId = collision[pos];
    }
    // ...
  }
}
```

Figure 2. The HSY non-blocking stack

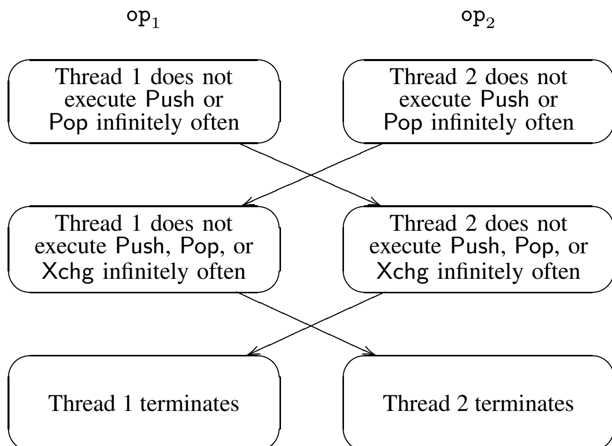

Figure 3. An informal proof argument where an arrow from statement A to statement B means that A is used as a rely condition while establishing the guarantee B.

ing of an arbitrary number of the algorithm's operations running in parallel: we search for proofs that follow the pattern described above and use rely and guarantee conditions of the restricted form. Our proof search procedure performs a forward search, constructing proof graphs like the one in Figure 3 top-down. It is able to construct proofs that the programs $C'(k)$ terminate for all k at once, because our guarantee conditions are such that if several threads satisfy a guarantee, then so does their parallel composition. We now informally describe the procedure using the HSY stack as the running example (the details are provided in Section 5).

Consider a program consisting of an arbitrary number of the algorithm's operations running in parallel. First, using existing tools for verifying safety properties of non-blocking algorithms [6], we can infer a splitting of the program state into local and shared parts and a set of actions describing how the operations change the shared state ($\{\mathsf{Push}, \mathsf{Pop}, \mathsf{Xchg}, \mathsf{Others}, \mathsf{Id}\}$ for the HSY stack). The set defines the initial guarantee provided by every operation in the program that ensures that the operation changes the shared state only according to one of the actions. Note that if several operations satisfy this guarantee, then so does their parallel composition. Hence, while checking a property of an operation in the program, we can rely on its environment satisfying the guarantee. The guarantee, however, is too weak to establish termination of the operations. We therefore try to strengthen it by considering every action in turn and attempting to prove that no operation executes the action infinitely often in an environment satisfying the guarantee. In our running example, we will be able to establish that the operations do not execute the actions Push and Pop infinitely often (but not Xchg and Others). Again, if several operations satisfy the guarantee strengthened in this way, then so does their parallel composition. Hence, we can check properties of the operations in the program assuming that their environment satisfies the guarantee. An attempt to prove their termination in this way fails again, and we have to strengthen the guarantee one more time. Namely, we try to prove that the operations do not execute the remaining actions Xchg and Others infinitely often. In this case, the available guarantee is strong enough to prove that the Xchg action is not executed infinitely often. Finally, the strengthened guarantee allows us to establish termination of the operations.

Proving obstruction-freedom and wait-freedom. Obstruction-freedom of an operation ensures its termination in an environment that eventually stops executing. Therefore, when proving obstruction-freedom, infinite behaviours of the environment are irrelevant and the necessary environment guarantee can always be

Values $= \{\dots,-1,0,1,\dots\}$	Locs $= \{1,2,\dots\}$
Vars $= \{x, y, \dots, \&\mathtt{x}, \&\mathtt{y}, \dots\}$	Stores $=$ Vars $\to$ Values
Heaps $=$ Locs $\rightharpoonup_{\text{fin}}$ Values	$\Sigma =$ Stores $\times$ Heaps

Figure 4. Program states Σ

represented by a safety property. For example, the operations of Treiber's stack guarantee that they modify the shared state according to one of the actions Push, Pop, and Id. To prove obstruction-freedom of a `push` or `pop` operation[2], it is thus sufficient to prove its termination in an environment that executes only finitely many such actions. This is true because, as in the proof of lock-freedom, in such an environment the CAS in the code of the operation will eventually succeed. In general, we can automatically check obstruction-freedom by checking termination of every operation in the environment satisfying the safety guarantee inferred by the safety verification tool and the additional assumption that it executes only finitely many actions. We describe this in more detail in Section 5.

To the best of our knowledge, loops in all practical wait-free non-blocking algorithms have constant upper bounds on the number of their iterations. For example, Simpson's algorithm [32] consists of straight-line code only, and the number of loop iterations in the wait-free `find` operation of a non-blocking linked list [37] is bounded by the number of distinct keys that can be stored in the list. For this reason, termination of operations in wait-free algorithms can be justified by considering only safety properties guaranteed by operations' environment. The automatic check for wait-freedom is similar to the one for obstruction-freedom (see Section 5).

3. Specifying liveness properties

Our logic for reasoning about liveness properties is a Hoare-style logic, which combines ideas from rely-guarantee reasoning and separation logic. It generalises a recent logic for reasoning about safety properties of non-blocking algorithms, RGSep [35]. As any Hoare logic, ours consists of two formal systems: an assertion language and a proof system for Hoare triples. In this section, we describe the assertion language, define the form of judgements in our logic, and show how to specify wait-freedom, lock-freedom, and obstruction-freedom in it. The next section presents the logic's proof system.

Programming language. We consider heap-manipulating programs written in the following simple programming language. Commands C are given by the grammar

$$C ::= C_1; C_2 \mid \mathtt{if}\ (e)\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{while}\ (e)\ C \mid \mathtt{x = new()} \mid \mathtt{x} = e \mid \mathtt{x} = {*}e_1 \mid {*}e_1 = e_2 \mid \mathtt{delete}\ e \mid \mathtt{atomic}\ C$$

where e ranges over arithmetic expressions, including non-deterministic choice `nondet()`. The command `atomic` C executes C in one indivisible step. Programs consist of initialisation code followed by a top-level parallel composition of threads: $C_0; (C_1 \| \dots \| C_n)$.

To avoid side conditions in our proof rules, we treat each program variable `x` as a memory cell at the constant address $\&\mathtt{x}$. Thus, any use of `x` in the code is just a shorthand for $*(\&\mathtt{x})$. Similarly, we interpret field expressions `x->next` as $*(\mathtt{x} + \mathtt{offset_of_next})$. In our examples, we also use other pieces of C syntax.

Assertion language. Let p, q and r be separation logic assertions:

$$p, q, r ::= \mathsf{emp} \mid e_1 \mapsto e_2 \mid p * q \mid \mathbf{false} \mid p \Rightarrow q \mid \exists x.p \mid \dots$$

[2] This example is used here for illustrative purposes only: obstruction-freedom of Treiber's stack follows from its lock-freedom.

Separation logic assertions denote sets of program states Σ represented by store-heap pairs (Figure 4). A store is a function from variables to values; a heap is a finite partial function from locations to values. We omit the standard formal semantics for most of the assertions [30]. Informally, emp describes the states where the heap is empty; $e_1 \mapsto e_2$ describes the states where the heap contains a single allocated location at the address e_1 with contents e_2; $p*q$ describes the states where the heap is the union of two disjoint heaps, one satisfying p and the other satisfying q. The formal semantics of the assertion $p * q$ is defined using a partial operation $\cdot$ on Σ such that for all $(t_1, h_1), (t_2, h_2) \in \Sigma$

$$(t_1, h_1) \cdot (t_2, h_2) = (t_1, h_1 \uplus h_2)$$

if $t_1 = t_2$ and $h_1 \uplus h_2$ is defined, and $(t_1, h_1) \cdot (t_2, h_2)$ is undefined otherwise. Then

$$u \in [\![p * q]\!] \Leftrightarrow \exists u_1, u_2.\ u = u_1 \cdot u_2 \wedge u_1 \in [\![p]\!] \wedge u_2 \in [\![q]\!]$$

As we argued in Section 2, while reasoning about concurrent heap-manipulating programs it is useful to partition the program state into thread-local and shared parts. Therefore, assertions in our logic denote sets of pairs of states from Σ. The two components represent the state *local* to the thread in whose code the assertion is located and the *shared* state. We use the assertion language of RGSep [35], which describes the local and shared components with separation logic assertions and is defined by following grammar:

$$P, Q ::= p \mid \boxed{p} \mid P * Q \mid \mathbf{true} \mid \mathbf{false} \mid P \vee Q \mid P \wedge Q \mid \exists x.P$$

An assertion p denotes the local-shared state pairs with the local state satisfying p; $\boxed{p}$ the pairs with the shared state satisfying p and the local state satisfying emp; $P * Q$ the pairs in which the local state can be divided into two substates such that one of them together with the shared state satisfies P and the other together with the shared state satisfies Q. The formal semantics of $P * Q$ is defined using a partial operation $\star$ on Σ^2 such that for all $(l_1, s_1), (l_2, s_2) \in \Sigma^2$

$$(l_1, s_1) \star (l_2, s_2) = (l_1 \cdot l_2, s_1)$$

if $s_1 = s_2$ and $l_1 \cdot l_2$ is defined, and $(l_1, s_1) \star (l_2, s_2)$ is undefined otherwise. Thus,

$$\sigma \in [\![P * Q]\!] \Leftrightarrow \exists \sigma_1, \sigma_2.\ \sigma = \sigma_1 \star \sigma_2 \wedge \sigma_1 \in [\![P]\!] \wedge \sigma_2 \in [\![Q]\!]$$

Note that by abuse of notation we denote the connectives interpreted by $\cdot$ and $\star$ with the same symbol $*$. It should be always clear from the structure of the assertion which of the two connectives is being used. We denote global states from Σ with small Latin letters (u, l, s), and local-shared state pairs from Σ^2 with small Greek letters (σ).

Judgements. The judgements in our logic include rely and guarantee conditions determining how a command or its environment change the shared state. These represent languages of finite and infinite words over the alphabet Σ^2 of relations on shared states and are denoted with capital calligraphic letters ($\mathcal{R}, \mathcal{G}, \dots$). A word in any of the languages describes the sequences of changes to the shared state. Thus, relies and guarantees can define liveness properties. This generalises the RGSep logic, in which rely and guarantee conditions define safety properties and are therefore represented with relations on the shared state. Our proof system has two kinds of judgements:

- $\mathcal{R}, \mathcal{G} \vdash \{P\}\ C\ \{Q\}$: if the initial state satisfies P and the environment changes the shared state according to $\mathcal{R}$, then the program is safe (i.e., it does not dereference any invalid pointers), changes the shared state according to $\mathcal{G}$, and the final state (if the program terminates) satisfies Q.

- $\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{P\}\ C_1 \| C_2\ \{Q\}$: if the initial state satisfies P and the environment changes the shared state according to $\mathcal{R}$, then the program $C_1 \| C_2$ is safe, C_1 changes the shared state according to $\mathcal{G}_1$, C_2 changes the shared state according to $\mathcal{G}_2$, and the final state (if both threads terminate) satisfies Q. Distinguishing the guarantee of each thread is crucial for liveness proofs done according to the method described in Section 2.

The informal definition of judgements' validity given above assumes a semantics of programs that distinguishes between the local and the shared state. We sketch how such a semantics can be defined later in this section. In Section 6 we give formal definitions of the semantics and the notion of validity, and relate them to the standard ones, which consider the program state as a whole.

Specifying rely and guarantee conditions. As noted above, a rely or a guarantee condition defines sequences of atomic changes to the shared state. We describe each such change with the aid of *actions* [35] of the form $p \leadsto q$, where p and q are separation logic assertions. Informally, this action changes the part of the shared state that satisfies p into one that satisfies q, while leaving the rest of the shared state unchanged. Formally, its meaning is a binary relation on shared states:

$$[\![p \leadsto q]\!] = \{(s_1 \cdot s_0, s_2 \cdot s_0) \mid s_1 \in [\![p]\!] \wedge s_2 \in [\![q]\!]\}$$

It relates some initial state s_1 satisfying the precondition p to a final state s_2 satisfying the postcondition q. In addition, there may be some disjoint state s_0 that is not affected by the action.

For example, we can define the three actions used in the proof of lock-freedom of Treiber's stack mentioned in Section 2 as follows:

$$\begin{aligned} \&\mathsf{S}{\mapsto}y &\leadsto \&\mathsf{S}{\mapsto}x * x{\mapsto}\mathsf{Node}(v,y) \quad &(\mathsf{Push})\\ \&\mathsf{S}{\mapsto}x * x{\mapsto}\mathsf{Node}(v,y) &\leadsto \&\mathsf{S}{\mapsto}y * x{\mapsto}\mathsf{Node}(v,y) \quad &(\mathsf{Pop})\\ \mathsf{emp} &\leadsto \mathsf{emp} \quad &(\mathsf{Id}) \end{aligned}$$

Here $x{\mapsto}\mathsf{Node}(v,y)$ is a shortcut for $x{\mapsto}v * (x+1){\mapsto}y$. Recall that the algorithm is assumed to be executed in the presence of a garbage collector. Hence, the node removed from the list by Pop stays in the shared state as garbage.

In our examples, we syntactically define rely and guarantee conditions using linear temporal logic (LTL) with actions as atomic propositions. Let False and True be the actions denoting the relations $\emptyset$ and Σ^2 respectively. We denote temporal operators "always", "eventually", and "next" with $\Box$, $\Diamond$, and $\bigcirc$, respectively. Their semantics on infinite words is standard [22]. The semantics on finite words is defined as follows [20]: for $m \geq 0$

$$\begin{aligned} \delta_1 \dots \delta_m \models \Box\Psi &\Leftrightarrow \forall i.\ 1 \leq i \leq m \Rightarrow \delta_i \dots \delta_m \models \Psi\\ \delta_1 \dots \delta_m \models \Diamond\Psi &\Leftrightarrow \exists i.\ 1 \leq i \leq m\ \wedge\ \delta_i \dots \delta_m \models \Psi\\ \delta_1 \dots \delta_m \models \bigcirc\Psi &\Leftrightarrow m \geq 2 \Rightarrow \delta_2 \dots \delta_m \models \Psi \end{aligned}$$

Note that here $\bigcirc$ is the *weak* next operator: it is true if there is no next state to interpret its argument over. For example, $\Box R$, where $R \subseteq \Sigma^2$, denotes the language of words in which every letter is from R (including the empty word), $\neg\Box\Diamond R$ denotes words which contain only finitely many letters from R (including all finite words), and $\Diamond\bigcirc\mathsf{False}$ denotes exactly all finite words. We specify termination of a command by requiring that it satisfy the guarantee $\Diamond\bigcirc\mathsf{False}$, i.e., we interpret termination as the absence of infinite computations. This is adequate for programs in the language introduced above, since they cannot deadlock.

The semantics of triples in our logic makes no assumptions about the scheduler. In particular, it can be unfair with respect to the command in the triple: the guarantee condition includes words corresponding to the command being suspended and never executed again. For this reason, all rely and guarantee conditions in this paper are prefix-closed, i.e., for any word belonging to a rely or a guarantee all its prefixes also belong to it. Additionally, we require that relies and guarantees represent nonempty languages.

Splitting states into local and shared parts. We now informally describe the *split-state* semantics of programs that splits the program state into local and shared parts (formalised in Section 6).

We partition all the atomic commands in the program into those that access only local state of the thread they are executed by and those that can additionally access the shared state. By convention, the only commands of the latter kind are `atomic` blocks. For example, in the operations of Treiber's stack, all the commands except for CASes access only the local state. Further, we annotate each `atomic` block with an action $p \leadsto q$ determining how it treats the shared state, written $\mathtt{atomic}_{p \leadsto q}\ C$. These annotations are a part of proofs in our logic. For the logic to be sound, all the judgements used in a proof of a program have to agree on the treatment of the shared state. We therefore require that the same annotation be used for any fixed `atomic` block throughout the proof.

In the split-state semantics the command $\mathtt{atomic}_{p \leadsto q}\ C$ executes as follows: it combines the local state of the thread it is executed by and the part of the shared state satisfying p, and runs C on this combination. It then splits the single resulting state into local and shared parts, determining the shared part as the one that satisfies q. The new shared state is thus this part together with the part of the shared state untouched by C. For the splittings to be defined uniquely (and for our logic to be sound), we require that p and q in all annotations be precise assertions [28]. An assertion r is precise if for any state u there exists at most one substate $\mathsf{sat}_r(u)$ satisfying r: $u = \mathsf{sat}_r(u) \cdot \mathsf{rest}_r(u)$ for some $\mathsf{rest}_r(u)$. The assertions in all the actions used in this paper are precise.

Let $\mathtt{CAS}_{\mathsf{A}}(\mathtt{addr}, \mathtt{v1}, \mathtt{v2})$ be defined as follows:

```
if (nondet())
  atomic_A {
    assume(*addr == v1); *addr = v2; return 1;
  }
else
  atomic_Id { assume (*addr != v1); return 0; }
```

where the `assume` command acts as a filter on the state space of programs—e is assumed to evaluate to 1 after $\mathtt{assume}(e)$ is executed. The above definition of CAS is semantically equivalent to the definition in Section 2, but allows different action annotations for the successful and the failure cases. We annotate the CAS commands in the `push` and `pop` operations of Treiber's stack as $\mathtt{CAS}_{\mathsf{Push}}(\&\mathtt{S}, \mathtt{t}, \mathtt{x})$ and $\mathtt{CAS}_{\mathsf{Pop}}(\&\mathtt{S}, \mathtt{t}, \mathtt{x})$, respectively. Similarly, we annotate the CAS in the inner loop of the HSY stack as $\mathtt{CAS}_{\mathsf{Xchg}}(\&\mathtt{collision}[\mathtt{pos}], \mathtt{hisId}, \mathtt{MYID})$.

Specifying wait-freedom, lock-freedom, and obstruction-freedom. A non-blocking data structure is given by an initialisation routine `init` and operations $\mathtt{op}_1, \dots, \mathtt{op}_n$ on the data structure. We require that the initialisation routine satisfy the triple

$$\Box\mathsf{Id}, \Box\mathsf{True} \vdash \left\{\boxed{\mathsf{emp}}\right\}\ \mathtt{init}\ \left\{\mathit{Inv}\right\}$$

for some data structure invariant *Inv* restricting only the shared state: the routine creates an instance of the data structure in the shared state when run in isolation (i.e., in the environment that does not change the shared state). For Treiber's stack an invariant maintained by all the operations on the data structure is that `S` points to the head of a linked list. We can express this in our assertion language using an inductive predicate assertion $\mathsf{lseg}(x, y)$ of separation logic that represents the least predicate satisfying

$$\begin{aligned} \mathsf{lseg}(x,y) \Leftrightarrow\ & (x = y \wedge \mathsf{emp})\\ & \vee (\exists z.\ x \neq y \wedge x{\mapsto}\mathsf{Node}(_, z) * \mathsf{lseg}(z, y)) \end{aligned}$$

Thus, $\mathsf{lseg}(x, \texttt{NULL})$ represents all of the states in which the heap has the shape of a (possibly empty) linked list starting from location x and ending with NULL. The invariant can then be expressed as

$$Inv = \boxed{\exists x.\ \&\mathtt{S} \mapsto x * \mathsf{lseg}(x, \texttt{NULL}) * \mathbf{true}} \tag{3.1}$$

In our logic, we can express the liveness properties of non-blocking algorithms we introduced before as follows. Wait-freedom of an operation op_i is captured by the triple

$$\mathcal{R}, \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \mathsf{op}_i\ \{\mathbf{true}\} \tag{3.2}$$

which ensures termination of the operation under the interference from the environment allowed by the rely condition $\mathcal{R}$. Obstruction-freedom of an operation op_i can be expressed as

$$\mathcal{R} \wedge \Diamond\bigcirc\mathsf{False}, \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \mathsf{op}_i\ \{\mathbf{true}\} \tag{3.3}$$

Here $\mathcal{R}$ describes the allowed interference from the operation's environment, and the conjunct $\Diamond\bigcirc\mathsf{False}$ ensures that eventually all the threads in the environment will be suspended. As we showed in Section 2, lock-freedom can be reduced to proving termination of several operations run in isolation, which is ensured by the validity of the triples

$$\Box\mathsf{Id}, \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ C'(k)\ \{\mathbf{true}\} \tag{3.4}$$

for all k, where the program $C'(k)$ is defined by (2.2).

Note that obstruction-freedom and wait-freedom are directly compositional properties and can thus be specified for every operation separately. The specification of lock-freedom considers all operations at once, however, as we show in the next section, we can still reason about lock-freedom in a compositional way.

4. Compositional proof system for liveness and heaps

To reason about judgements of the form introduced in the previous section we need *(i)* a method for proving thread-local triples (i.e., those giving a specification to a single thread) and *(ii)* a proof system for combining thread-local triples into triples about parallel compositions. We describe an automatic method for proving thread-local triples in Section 5 (the THREADLOCAL procedure and Figure 7). In this section, we present the second component—a compositional proof system for reasoning about liveness properties of heap-manipulating programs, shown in Figure 5. We explain the proof rules by example of formalising the informal proofs of lock-freedom from Section 2. In Section 5 we show how to construct such proofs automatically, and in Section 6 we prove the proof rules sound with respect to an interleaving semantics.

We first introduce two operations on languages used by the rules. Let $\mathcal{L}(A)$ denote the language of all finite and infinite words over an alphabet A. We denote the concatenation of a finite word $\alpha \in \mathcal{L}(A)$ and a word (either finite or infinite) $\beta \in \mathcal{L}(A)$ with $\alpha\beta$. The safety closure $\mathsf{CL}(\mathcal{G})$ of a language $\mathcal{G} \subseteq \mathcal{L}(A)$ is the smallest language defining a safety property that contains $\mathcal{G}$ [2]. In the setting of this paper, where all the languages considered are prefix-closed, $\mathsf{CL}(\mathcal{G})$ can be defined as the set of words α such that every prefix of α is in $\mathcal{G}$:

$$\mathsf{CL}(\mathcal{G}) = \{\alpha \in \mathcal{L}(A) \mid \forall \beta, \gamma.\ \alpha = \beta\gamma \Rightarrow \beta \in \mathcal{G}\}$$

For two words $\alpha, \beta \in \mathcal{L}(A)$, we denote the set of their fair interleavings with $\alpha \| \beta$ (we omit the standard definition [4]). We lift this to languages $\mathcal{G}_1, \mathcal{G}_2 \subseteq \mathcal{L}(A)$ as follows:

$$\mathcal{G}_1 \| \mathcal{G}_2 = \bigcup \{\alpha \| \beta \mid \alpha \in \mathcal{G}_1 \wedge \beta \in \mathcal{G}_2\}$$

4.1 Proving lock-freedom of Treiber's non-blocking stack

We start by proving termination of any two operations with arbitrary parameters (which we denote with op_{i1} and op_{i2}) running in parallel and consider the general case later. To prove this, we have to derive the triple

$$\Box\mathsf{Id}, \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \tag{4.1}$$

for the data structure invariant *Inv* defined by (3.1).

$$\frac{\mathcal{R}\|\mathsf{CL}(\mathcal{G}_2), \mathcal{G}_1 \vdash \{P_1\}\ C_1\ \{Q_1\} \qquad \mathcal{R}\|\mathsf{CL}(\mathcal{G}_1), \mathcal{G}_2 \vdash \{P_2\}\ C_2\ \{Q_2\}}{\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{P_1 * P_2\}\ C_1\|C_2\ \{Q_1 * Q_2\}}\ \text{PAR-C}$$

$$\frac{\mathcal{R}, \vec{\mathcal{G}} \vdash \{P\}\ C\ \{Q\} \qquad P' \Rightarrow P \quad \mathcal{R}' \subseteq \mathcal{R} \quad \vec{\mathcal{G}} \subseteq \vec{\mathcal{G}}' \quad Q \Rightarrow Q'}{\mathcal{R}', \vec{\mathcal{G}}' \vdash \{P'\}\ C\ \{Q'\}}\ \text{CONSEQ}$$

$$\frac{\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{P_1 * P_2\}\ C_1\|C_2\ \{\mathbf{true}\} \qquad \mathcal{R}\|\mathcal{G}_2, \mathcal{G}'_1 \vdash \{P_1\}\ C_1\ \{Q_1\} \qquad \mathcal{R}\|\mathcal{G}_1, \mathcal{G}'_2 \vdash \{P_2\}\ C_2\ \{Q_2\}}{\mathcal{R}, (\mathcal{G}'_1, \mathcal{G}'_2) \vdash \{P_1 * P_2\}\ C_1\|C_2\ \{Q_1 * Q_2\}}\ \text{PAR-NC}$$

$$\frac{\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{P\}\ C_1\|C_2\ \{Q\}}{\mathcal{R}, \mathcal{G}_1\|\mathcal{G}_2 \vdash \{P\}\ C_1\|C_2\ \{Q\}}\ \text{PAR-MERGE}$$

$$\frac{\mathcal{R}', \vec{\mathcal{G}}' \vdash \{P'\}\ C\ \{Q'\} \qquad \mathcal{R}'', \vec{\mathcal{G}}'' \vdash \{P''\}\ C\ \{Q''\}}{\mathcal{R}' \cap \mathcal{R}'', \vec{\mathcal{G}}' \cap \vec{\mathcal{G}}'' \vdash \{P' \wedge P''\}\ C\ \{Q' \wedge Q''\}}\ \text{CONJ}$$

Figure 5. Proof rules for reasoning about liveness properties of heap-manipulating programs. $\vec{\mathcal{G}}$ denotes either $\mathcal{G}$ or $(\mathcal{G}_1, \mathcal{G}_2)$ depending on whether the triple distinguishes between the guarantees provided by the different threads. In the latter case operations on $(\mathcal{G}_1, \mathcal{G}_2)$ are done componentwise.

Statement I. Formally, the statement says that every thread has to satisfy the guarantee

$$\mathcal{G} = \Box(\mathsf{Push} \vee \mathsf{Pop} \vee \mathsf{Id}) \wedge \neg\Box\Diamond(\mathsf{Push} \vee \mathsf{Pop})$$

where the actions Push, Pop, and Id are defined in Section 3. The first conjunct specifies the actions that the thread can execute, and the second ensures that it cannot execute the actions Push and Pop infinitely often. In order to establish this guarantee, we do not have to make any liveness assumptions on the behaviour of other threads; just knowing the actions they can execute (Push, Pop, and Id) is enough. We therefore use the rule PAR-C to establish $\mathcal{G}$. It is a circular rely guarantee rule [1] adapted for reasoning about heaps. It allows two threads to establish their guarantees simultaneously, while relying on the safety closure of the other thread's guarantee that is being established. Note that without the safety closure the circular rules like this are unsound for liveness properties [1]. Note also that pre- and postconditions of threads in the premises of the rule are $*$-conjoined in the conclusion: according to the semantics of the assertion language, this takes the disjoint composition of the local states of the threads and enforces that the threads have the same view of the shared state. It is this feature of our proof rules that allows us to reason modularly in the presence of heap.

Applying PAR-C with $\mathcal{G}_1 = \mathcal{G}_2 = \mathcal{G}$ and $\mathcal{R} = \mathsf{CL}(\mathcal{G})$, we get:

$$\frac{\mathsf{CL}(\mathcal{G})\|\mathsf{CL}(\mathcal{G}), \mathcal{G} \vdash \{Inv\}\ \mathsf{op}_{i1}\ \{\mathbf{true}\} \qquad \mathsf{CL}(\mathcal{G})\|\mathsf{CL}(\mathcal{G}), \mathcal{G} \vdash \{Inv\}\ \mathsf{op}_{i2}\ \{\mathbf{true}\}}{\mathsf{CL}(\mathcal{G}), (\mathcal{G}, \mathcal{G}) \vdash \{Inv * Inv\}\ \mathsf{op}_{i1}\|\mathsf{op}_{i2}\ \{\mathbf{true} * \mathbf{true}\}} \tag{4.2}$$

Taking the safety closure of $\mathcal{G}$ removes the second conjunct representing the liveness part of $\mathcal{G}$:

$$\mathsf{CL}(\mathcal{G}) = \Box(\mathsf{Push} \vee \mathsf{Pop} \vee \mathsf{Id})$$

Additionally, $\mathsf{CL}(\mathcal{G}) \| \mathsf{CL}(\mathcal{G}) = \mathsf{CL}(\mathcal{G})$, so that the premises simplify to triples

$$\mathsf{CL}(\mathcal{G}), \mathcal{G} \vdash \{\mathit{Inv}\}\ \mathsf{op}_j\ \{\mathbf{true}\},\ \ j \in \{i1, i2\} \qquad (4.3)$$

which ensure that the thread does not execute Push and Pop actions infinitely often, provided the environment executes only actions Push, Pop, and Id. We show how to discharge such triples in Section 5. Their proof would formalise the informal justification of Statement I given in Section 2 and would use the annotations at `atomic` blocks introduced in Section 3 to determine the splitting of states into local and shared parts.

Since *Inv* restricts only the shared state, $\mathit{Inv} * \mathit{Inv} \Leftrightarrow \mathit{Inv}$, hence, the conclusion of (4.2) is equivalent to

$$\mathsf{CL}(\mathcal{G}), (\mathcal{G}, \mathcal{G}) \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \qquad (4.4)$$

Since $\mathcal{G} \subseteq \mathsf{CL}(\mathcal{G})$, we can then apply a variation on the rule of consequence of Hoare logic, CONSEQ, which allows us to strengthen the rely condition to $\mathcal{G}$:

$$\mathcal{G}, (\mathcal{G}, \mathcal{G}) \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \qquad (4.5)$$

Statement II. Termination is captured by the guarantee $\Diamond\bigcirc\mathsf{False}$, which says that eventually the program does not execute any transitions. To prove this guarantee, we use the non-circular rely-guarantee rule PAR-NC, which allows the first thread to replace its guarantee with a new one based on the already established guarantee of the other thread, and vice versa. Note that the first premise need only establish the postcondition $\mathbf{true}$, since the postcondition $Q_1 * Q_2$ of the conclusion is implied by the other two premises. Applying PAR-NC with $\mathcal{R} = \mathcal{G}_1 = \mathcal{G}_2 = \mathcal{G}$ and $\mathcal{G}_1' = \mathcal{G}_2' = \Diamond\bigcirc\mathsf{False}$, we get:

$$\frac{\begin{array}{c}\mathcal{G}, (\mathcal{G}, \mathcal{G}) \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \\ \mathcal{G} \| \mathcal{G}, \Diamond\bigcirc\mathsf{False} \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1}\ \{\mathbf{true}\} \\ \mathcal{G} \| \mathcal{G}, \Diamond\bigcirc\mathsf{False} \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i2}\ \{\mathbf{true}\}\end{array}}{\mathcal{G}, (\Diamond\bigcirc\mathsf{False}, \Diamond\bigcirc\mathsf{False}) \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\}} \qquad (4.6)$$

We have already derived the first premise. Since $\mathcal{G} \| \mathcal{G} = \mathcal{G}$, we need to discharge the following thread-local triples (again postponed to Section 5):

$$\mathcal{G}, \Diamond\bigcirc\mathsf{False} \vdash \{\mathit{Inv}\}\ \mathsf{op}_j\ \{\mathbf{true}\},\ \ j \in \{i1, i2\} \qquad (4.7)$$

We no longer need to distinguish between the guarantees of the two threads in the conclusion of (4.6). Hence, we use the rule PAR-MERGE, which merges the guarantees provided by the threads into a single guarantee provided by their parallel composition:

$$\frac{\mathcal{G}, (\Diamond\bigcirc\mathsf{False}, \Diamond\bigcirc\mathsf{False}) \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\}}{\mathcal{G}, (\Diamond\bigcirc\mathsf{False}) \| (\Diamond\bigcirc\mathsf{False}) \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\}}$$

The conclusion is equivalent to

$$\mathcal{G}, \Diamond\bigcirc\mathsf{False} \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \qquad (4.8)$$

from which (4.1) follows by CONSEQ. This proves termination of the two operations.

Arbitrary number of operations. We can generalise our proof to an arbitrary number of operations as follows. First, note that applying PAR-MERGE on (4.4), we get:

$$\mathsf{CL}(\mathcal{G}), \mathcal{G} \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \qquad (4.9)$$

Hence, the proof for two operations establishes (4.9) and (4.8) given (4.3) and (4.7), i.e., it shows that the parallel composition $\mathsf{op}_{i1} \| \mathsf{op}_{i2}$ preserves the properties (4.3) and (4.7) of its constituent operations. Note that this derivation is independent of the particular definitions of op_{i1} and op_{i2} satisfying (4.3) and (4.7).

This allows us to prove by induction on k that

$$\begin{array}{c}\mathsf{CL}(\mathcal{G}), \mathcal{G} \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \dots \| \mathsf{op}_{ik}\ \{\mathbf{true}\} \\ \mathcal{G}, \Diamond\bigcirc\mathsf{False} \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \dots \| \mathsf{op}_{ik}\ \{\mathbf{true}\}\end{array} \qquad (4.10)$$

is derivable in our proof system for any $k \geq 1$. For $k = 1$ the triples are established by (4.3) and (4.7). For the induction step, we just repeat the previous derivation with op_{i1} replaced by $\mathsf{op}_{i1} \| \dots \| \mathsf{op}_{ik}$ and op_{i2} replaced by $\mathsf{op}_{i(k+1)}$.

Applying CONSEQ to (4.10), we get (3.4), which entails lock-freedom of Treiber's stack.

Note that instead of doing induction on the number of threads, we could have formulated our proof rules for k threads. To simplify the presentation, we chose the minimalistic proof system.

4.2 Proving lock-freedom of the HSY non-blocking stack

The action Xchg used in the informal proof of lock-freedom of the HSY stack (Section 2) can be formally defined as follows:

$$0 \leq i \leq \mathtt{SIZE} - 1 \wedge \mathtt{collision}[i] \mapsto _ \rightsquigarrow \mathtt{collision}[i] \mapsto _ \qquad (\mathsf{Xchg})$$

The abridged data structure invariant is:

$$\mathit{Inv} = \boxed{\begin{array}{l}\exists x.\ \&\mathtt{S} \mapsto x * \mathsf{lseg}(x, \mathtt{NULL}) * \mathbf{true} \\ * \circledast_{i=0}^{\mathtt{SIZE}-1} \mathtt{collision}[i] \mapsto _ * \dots\end{array}}$$

(We elided some of the data structures of the elimination scheme.)

We now formalise the informal proof from Section 2 for two operations op_{i1} and op_{i2} running in parallel.

Statement I. The statement requires us to establish the guarantee

$$\mathcal{G} = \Box(\mathsf{Push} \vee \mathsf{Pop} \vee \mathsf{Xchg} \vee \mathsf{Others} \vee \mathsf{Id}) \wedge \neg\Box\Diamond(\mathsf{Push} \vee \mathsf{Pop})$$

where Others describes the interference caused by the elimination scheme (elided). As before, we can do this using PAR-C. Given the thread-local triples (4.3) with the newly defined op_{i1}, op_{i2}, $\mathcal{G}$, and *Inv*, we can again derive (4.5) and (4.9), where

$$\mathsf{CL}(\mathcal{G}) = \Box(\mathsf{Push} \vee \mathsf{Pop} \vee \mathsf{Xchg} \vee \mathsf{Others} \vee \mathsf{Id})$$

Statement II. Now, provided that a thread satisfies the guarantee $\mathcal{G}$, we have to prove that the other thread satisfies the guarantee

$$\mathcal{G}' = \Box(\mathsf{Push} \vee \mathsf{Pop} \vee \mathsf{Xchg} \vee \mathsf{Others} \vee \mathsf{Id}) \wedge \neg\Box\Diamond\mathsf{Xchg}$$

To this end, we use the non-circular rely-guarantee rule PAR-NC:

$$\frac{\begin{array}{c}\mathcal{G}, (\mathcal{G}, \mathcal{G}) \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \\ \mathcal{G} \| \mathcal{G}, \mathcal{G}' \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1}\ \{\mathbf{true}\} \\ \mathcal{G} \| \mathcal{G}, \mathcal{G}' \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i2}\ \{\mathbf{true}\}\end{array}}{\mathcal{G}, (\mathcal{G}', \mathcal{G}') \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\}}$$

We thus have to establish the following thread-local triples:

$$\mathcal{G}, \mathcal{G}' \vdash \{\mathit{Inv}\}\ \mathsf{op}_j\ \{\mathbf{true}\},\ \ j \in \{i1, i2\} \qquad (4.11)$$

We can now use the conjunction rule, CONJ, to combine the guarantees $\mathcal{G}$ and $\mathcal{G}'$ into a single guarantee $\mathcal{G}''$:

$$\frac{\begin{array}{c}\mathcal{G}, (\mathcal{G}, \mathcal{G}) \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \\ \mathcal{G}, (\mathcal{G}', \mathcal{G}') \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\}\end{array}}{\mathcal{G}, (\mathcal{G}'', \mathcal{G}'') \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\}} \qquad (4.12)$$

where

$$\begin{array}{r}\mathcal{G}'' = \mathcal{G} \wedge \mathcal{G}' = \Box(\mathsf{Push} \vee \mathsf{Pop} \vee \mathsf{Xchg} \vee \mathsf{Others} \vee \mathsf{Id}) \\ \wedge \neg\Box\Diamond(\mathsf{Push} \vee \mathsf{Pop} \vee \mathsf{Xchg})\end{array}$$

This combines Statements I and II. Applying CONSEQ, we get:

$$\mathcal{G}'', (\mathcal{G}'', \mathcal{G}'') \vdash \{\mathit{Inv}\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\}$$

```
procedure LockFree(init, op)
    (Inv, G_1) := SafetyGuarantee(init, op)
    G_2 := ∅
    do
        𝒢 := □G_1 ∧ ¬□◇G_2
        if ThreadLocal(𝒢, ◇○False ⊢ {Inv} op {true})
            return "Lock-free"
        G_2^0 := G_2
        for each A ∈ (G_1 \ G_2^0) do
            if ThreadLocal(𝒢, ¬□◇A ⊢ {Inv} op {true})
                G_2 := G_2 ∪ {A}
    while G_2^0 ≠ G_2
    return "Don't know"
```

Figure 6. Proof search procedure for lock-freedom

Thus, we have strengthened the guarantee $\mathcal{G}$ of Statement I to the guarantee $\mathcal{G}''$.

Statement III. Finally, we can formalise Statement III by applying the rule PAR-NC to establish termination:

$$\frac{\begin{array}{c}\mathcal{G}'', (\mathcal{G}'', \mathcal{G}'') \vdash \{Inv\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \\ \mathcal{G}'' \| \mathcal{G}'', \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \mathsf{op}_{i1}\ \{\mathbf{true}\} \\ \mathcal{G}'' \| \mathcal{G}'', \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \mathsf{op}_{i2}\ \{\mathbf{true}\}\end{array}}{\mathcal{G}'', (\Diamond\bigcirc\mathsf{False}, \Diamond\bigcirc\mathsf{False}) \vdash \{Inv\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\}}$$

provided we can establish the thread-local triples

$$\mathcal{G}'', \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \mathsf{op}_j\ \{\mathbf{true}\}, \quad j \in \{i1, i2\} \tag{4.13}$$

By PAR-MERGE we then get:

$$\mathcal{G}'', \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \tag{4.14}$$

which proves the termination of the two operations.

Arbitrary number of operations. From the conclusion of (4.12), by PAR-MERGE we get:

$$\mathcal{G}, \mathcal{G}'' \vdash \{Inv\}\ \mathsf{op}_{i1} \| \mathsf{op}_{i2}\ \{\mathbf{true}\} \tag{4.15}$$

Thus, the above derivation establishes (4.9), (4.15), and (4.14) given (4.3), (4.11), and (4.13). As before, this allows us to prove by induction on k that the following triples are derivable in our logic for any $k \geq 1$:

$$\begin{array}{c}\mathsf{CL}(\mathcal{G}), \mathcal{G} \vdash \{Inv\}\ \mathsf{op}_{i1} \| \ldots \| \mathsf{op}_{ik}\ \{\mathbf{true}\} \\ \mathcal{G}, \mathcal{G}'' \vdash \{Inv\}\ \mathsf{op}_{i1} \| \ldots \| \mathsf{op}_{ik}\ \{\mathbf{true}\} \\ \mathcal{G}'', \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \mathsf{op}_{i1} \| \ldots \| \mathsf{op}_{ik}\ \{\mathbf{true}\}\end{array}$$

The last one implies lock-freedom of the HSY stack.

5. Automation

In this section we describe our automatic prover for liveness properties of non-blocking concurrent algorithms. Our tool's input is a liveness property to be proved and a program in a C-like language consisting of the code of operations $\mathsf{op}_1, \ldots, \mathsf{op}_n$ of a non-blocking algorithm, together with a piece of initialisation code `init`. We remind the reader that we denote with `op` the command, defined by (2.1), that non-deterministically executes one of the operations op_i on the data structure. We first describe how our tool handles lock-freedom.

Proving lock-freedom via proof search. Recall that to prove lock-freedom, we have to prove termination of the program $C'(k)$ defined by (2.2) for an arbitrary k. All rely and guarantee conditions used in the examples of such proofs in Section 4 had a restricted form $\Box A_1 \wedge \neg\Box\Diamond A_2$, where A_1 and A_2 are sets (disjunctions) of actions and $A_2 \subseteq A_1$. Here A_1 is the set of all actions that a thread can perform, whereas A_2 is the set of actions that the thread performs only finitely often. In fact, for all the non-blocking algorithms we have studied, it was sufficient to consider rely and guarantee conditions of this form to prove lock-freedom.

We prove termination of $C'(k)$ by searching for proofs of triple (3.4) in our proof system in the style of those presented in Section 4 with relies and guarantees of the form $\Box A_1 \wedge \neg\Box\Diamond A_2$. There are several ways in which one can organise such a proof search. The strategy we use here is to perform forward search as explained informally in Section 2.

Figure 6 contains our procedure LOCKFREE for proving lock-freedom. It is parameterised with two auxiliary procedures, whose implementation is described later:

- SAFETYGUARANTEE(`init`, `op`) computes supporting safety properties for our liveness proofs, namely, a data structure invariant *Inv* such that

$$\Box\mathsf{Id}, \Box\mathsf{True} \vdash \left\{\boxed{\mathsf{emp}}\right\}\ \mathtt{init}\ \left\{Inv\right\} \tag{5.1}$$

  and an initial safety guarantee provided by every operation, which is defined by a set of actions $G_1 = \{\mathsf{A}_1, \ldots, \mathsf{A}_n\}$ such that

$$\Box G_1, \Box G_1 \vdash \{Inv\}\ \mathsf{op}\ \{Inv\} \tag{5.2}$$

- THREADLOCAL($\mathcal{R}, \mathcal{G} \vdash \{Inv\}\ \mathsf{op}\ \{\mathbf{true}\}$) attempts to prove the thread-local triple $\mathcal{R}, \mathcal{G} \vdash \{Inv\}\ \mathsf{op}\ \{\mathbf{true}\}$ valid. The notion of validity of thread-local triples used by THREADLOCAL corresponds to the informal explanation given in Section 3 and is formalised in Section 6.

LOCKFREE first calls SAFETYGUARANTEE to compute the data structure invariant *Inv* and the safety guarantee $\Box G_1$. In our proofs of liveness properties, rely and guarantee conditions are then represented using LTL formulae with actions in G_1 as atomic propositions. A side-effect of SAFETYGUARANTEE is that it annotates `atomic` blocks in `op` with actions from G_1 as explained in Section 3. These annotations are used by the subsequent calls to THREADLOCAL, which ensures that all thread-local reasoning in the proof of lock-freedom uses the same splitting of the program state into local and shared parts.

Having computed the safety guarantee, we enter into a loop, where on every iteration we first attempt to prove termination of `op` using the available guarantee. If this succeeds, we have proved lock-freedom. Otherwise, we try to strengthen the guarantee $\Box G_1 \wedge \neg\Box\Diamond G_2$ by considering each action in $G_1 \setminus G_2$ and trying to prove that it is executed only finitely often using the current guarantee as a rely condition. If we succeed, we update the guarantee by adding the action to the set of finitely executed actions G_2. If we cannot prove that any action from $G_1 \setminus G_2$ is executed only finitely often, we give up the search and exit the loop.

This procedure scales because in practice the set of actions G_1 computed by SAFETYGUARANTEE is small. This is due to the fact that actions $p \rightsquigarrow q$ are local in the sense that p and q describe only the parts of the shared state modified by atomic blocks.

It is possible to show that a successful run of LOCKFREE constructs proofs of triples (3.4) for all k. We can construct the proofs for any number of threads uniformly because the guarantees $\mathcal{G}$ used in them are such that $\mathcal{G} \| \mathcal{G} = \mathcal{G}$. The construction follows the method of Section 4. The only difference is that the proofs constructed by LOCKFREE first apply the rule PAR-C to triples (5.2) with $\mathcal{G}_1 = \mathcal{G}_2 = \Box G_1$. Since $\mathsf{CL}(\Box G_1) = \Box G_1$, this establishes the initial safety guarantee $\Box G_1$, which is then strengthened using

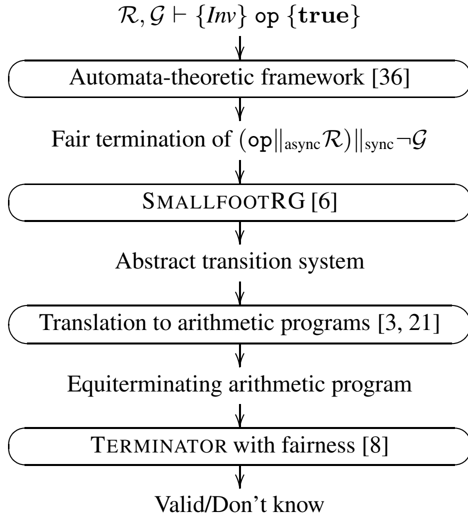

Figure 7. The high-level structure of the THREADLOCAL procedure for discharging thread-local triples

the rule PAR-NC. In the proofs of Section 4, these two steps were performed with one application of the rule PAR-C.

We now describe the two auxiliary procedures used by LOCK-FREE—SAFETYGUARANTEE and THREADLOCAL.

The* SAFETYGUARANTEE *procedure. We implement the procedure using the SMALLFOOTRG tool for verifying safety properties of non-blocking algorithms [6]. SMALLFOOTRG computes a data structure invariant and an interference specification by performing abstract interpretation of the code of `init` and `op` over an abstract domain constructed from RGSep formulae. This abstract interpretation is thread-modular, i.e., it repeatedly analyses separate threads without enumerating interleavings using an algorithm similar to the one described in [12]. For the invariant and interference specifications computed by SMALLFOOTRG to be strong enough for use in liveness proofs, its abstract domain has to be modified to keep track of the lengths of linked lists as described in [21].

RGSep judgements can be expressed in our logic by triples with rely and guarantee conditions of the form $\Box A$, where A is a set of actions. SMALLFOOTRG proves the validity of RGSep judgements that, when translated to our logic in this way, yield (5.1) and (5.2).

The* THREADLOCAL *procedure. We prove a thread-local triple $\mathcal{R}, \mathcal{G} \vdash \{Inv\}$ op $\{\mathbf{true}\}$ using a combination of several existing methods and tools, as shown in Figure 7. For technical reasons, in this procedure we assume that $\mathcal{R}$ and $\mathcal{G}$ consist of only infinite words and `op` has only infinite computations. This can always be ensured by padding the finite words in $\mathcal{R}$ and $\mathcal{G}$ with a special dummy action and inserting an infinite loop at the end of `op` executing the action. To prove the triple $\mathcal{R}, \mathcal{G} \vdash \{Inv\}$ op $\{\mathbf{true}\}$:

- We first represent $\mathcal{R}$ and $\neg\mathcal{G}$ as Büchi automata, whose transitions are labelled with actions from the set G_1 computed by SAFETYGUARANTEE and apply the automata-theoretic framework for program verification [36]. This reduces proving the triple to proving that the program $(\texttt{op}\|_{\text{async}}\mathcal{R})\|_{\text{sync}}\neg\mathcal{G}$ terminates when run from states satisfying the precondition *Inv* under the fairness assumptions extracted from the accepting conditions of the automata for $\mathcal{R}$ and $\neg\mathcal{G}$. Here $\texttt{op}\|_{\text{async}}\mathcal{R}$ is the asynchronous parallel composition interleaving the executions of `op` and the automaton $\mathcal{R}$ in all possible ways. The program $(\texttt{op}\|_{\text{async}}\mathcal{R})\|_{\text{sync}}\neg\mathcal{G}$ is the synchronous parallel composition of $\texttt{op}\|_{\text{async}}\mathcal{R}$ and the automaton $\neg\mathcal{G}$ synchronising on actions of `op`. Intuitively, fair infinite executions of the program $(\texttt{op}\|_{\text{async}}\mathcal{R})\|_{\text{sync}}\neg\mathcal{G}$ correspond to the executions of `op` in an environment satisfying the rely $\mathcal{R}$ that violate the guarantee $\mathcal{G}$. Its fair termination implies that there are no such executions.
- To check fair termination of $(\texttt{op}\|_{\text{async}}\mathcal{R})\|_{\text{sync}}\neg\mathcal{G}$, we analyse it with the abstract interpreter of SMALLFOOTRG [6], which produces an abstract transition system over-approximating the program's behaviour. The interpreter uses the annotations at atomic blocks computed by SAFETYGUARANTEE to choose the splitting of the heap into local and shared parts.
- Using the techniques of [3, 21], from this transition system we then extract an arithmetic program (i.e., a program without the heap with only integer variables), whose fair termination implies fair termination of $(\texttt{op}\|_{\text{async}}\mathcal{R})\|_{\text{sync}}\neg\mathcal{G}$. The arithmetic program makes explicit the implicit arithmetic information present in the heap-manipulating program that can be used by termination provers to construct ranking functions. For example, it contains integer variables tracking the lengths of linked lists in the original program.
- Finally, we run a termination prover (TERMINATOR with fairness [8]) to prove fair termination of the arithmetic program.

We note that proofs of thread-local statements may be more complicated then the ones in the examples of Section 2, which were based on control-flow arguments. For example, for Michael's non-blocking linked list algorithm [25] they involve reasoning about lengths of parts of the shared data structure. Furthermore, the proofs may rely on complex supporting safety properties that ensures that the data structure is well-formed. Automatic tool support is indispensable in constructing such proofs.

Proving obstruction-freedom and wait-freedom. As we showed in Section 2, proving obstruction-freedom or wait-freedom of an operation in a non-blocking algorithm usually requires only safety guarantees provided by the operation's environment. In our tool, we use the guarantee $\Box G_1$ inferred by SAFETYGUARANTEE. Namely, we prove obstruction-freedom of an operation $\texttt{op}_i$ by establishing triple (3.3) with $\mathcal{R} = \Box G_1$ via a call to

$$\text{THREADLOCAL}(\Box G_1 \wedge \Diamond\bigcirc\mathsf{False}, \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \texttt{op}_i\ \{\mathbf{true}\})$$

We can prove wait-freedom of an operation $\texttt{op}_i$ by establishing triple (3.2) with $\mathcal{R} = \Box G_1$ via a call to

$$\text{THREADLOCAL}(\Box G_1, \Diamond\bigcirc\mathsf{False} \vdash \{Inv\}\ \texttt{op}_i\ \{\mathbf{true}\})$$

Experiments. Using our tool, we have proved a number of non-blocking algorithms lock-free and have found counterexamples demonstrating that they are not wait-free. The examples we analysed include a DCAS-based stack, Treiber's stack [33], the HSY stack [14], a non-blocking queue due to Michael and Scott [26] and its optimised version due to Doherty et al. [9], a restricted double-compare single-swap operation (RDCSS) [13], and Michael's non-blocking linked list [25]. In all cases except Michael's algorithm the tool found a proof of lock-freedom in less then 10 minutes. Verification of Michael's algorithm takes approximately 8 hours, which is due to the unoptimised arithmetic program generator and the inefficient version of the termination prover that we currently use.

We have also tested our tool by proving the obstruction-freedom of the above lock-free algorithms. (Obstruction-free algorithms that are not lock-free typically traverse arrays, handling which is beyond the scope of the shape analysis that we use.) Additionally, we have checked that the deletion operation of a linked list algorithm by Vechev and Yahav [37, Figure 2] is not obstruction-free (as observed by the authors), even though it does not use locks.

We do not report any results for wait-free algorithms in this paper. Operations consisting of straight-line code only are trivially wait-free. Proving termination of wait-free `find` operations in non-blocking linked lists mentioned in Section 2 requires tracking the keys stored in the list, which is not handled by our shape analysis.

6. Semantics and soundness

We give semantics to programs with respect to labelled transition systems with states representing the whole program heap. The proof of soundness of our logic with respect to this global semantics is done in two steps. We first show that, given a transition system denoting a program, we can construct another transition system operating on states that distinguish between local and shared heap, according to the informal description given in Section 3. Interpretation of judgements in this split-state semantics is straightforward. We then relate the validity of judgements in the split-state semantics to the standard global notion of validity. The results in this section do not follow straightforwardly from existing ones, however, the techniques used to formulate the split-state semantics and prove the soundness theorems are the same as for the RGSep logic [35].

6.1 Global semantics

We represent denotations of programs as a variant of labelled transition systems (LTS).

DEFINITION 1 (LTS). *A labelled transition system (LTS) is a quadruple $S = (\Sigma, \top, \Phi, T)$, where*

- *Σ is the set of non-erroneous states of the transition system,*
- *$\top \notin \Sigma$ is a distinguished error state (arising, for example, when a program dereferences an invalid pointer),*
- *$\Phi \subseteq \Sigma$ is the set of final states, and*
- *T is the set of transitions such that every $\tau \in T$ is associated with a transition function $f_\tau : \Sigma \to \mathcal{P}(\Sigma) \cup \{\top\}$, where $\mathcal{P}(\Sigma)$ is the powerset of Σ.*

DEFINITION 2 (Computation of an LTS). *A computation of an LTS $(\Sigma, \top, \Phi, T)$ starting from an initial state $u_0 \in \Sigma$ is a maximal sequence $u_0, u_1, \ldots$ of states $u_i \in \Sigma \cup \{\top\}$ such that for all i there exists a transition $\tau \in T$ such that $u_{i+1} = \top$ if $f_\tau(u_i) = \top$ and $u_{i+1} \in f_\tau(u_i)$ otherwise.*

Given a thread C in the programming language of Section 3, we can construct the corresponding LTS $[\![C]\!]$ in the following way. Let us assume for the purposes of this construction that the program counter of the thread is a memory cell at a distinguished address &`pc`, implicitly modified by every primitive command. As the set of states Σ of the LTS we take the one defined in Figure 4. The final states are those in which the program counter has a distinguished final value. Every atomic command in the thread, including `atomic` blocks, corresponds to a transition in the LTS. Conditions in `if` and `while` commands are translated in the standard way using `assume` commands. The transition functions are then just the standard postcondition transformers [30, 5].

The denotation of a parallel composition of threads is the parallel composition of their denotations, defined as follows.

DEFINITION 3 (Parallel composition of LTSes). *The parallel composition of two LTSes $S_1 = (\Sigma, \top, \Phi_1, T_1)$ and $S_2 = (\Sigma, \top, \Phi_2, T_2)$, where $T_1 \cap T_2 = \emptyset$, is defined as the LTS $S_1 \| S_2 = (\Sigma, \top, \Phi_1 \cap \Phi_2, T_1 \uplus T_2)$.*

As follows from Definitions 2 and 3, the parallel composition interleaves transitions from two LTSes on the same memory Σ without any fairness constraints. Note that we can always satisfy $T_1 \cap T_2 = \emptyset$ by renaming transitions appropriately.

6.2 Split-state semantics

We now show that given an LTS we can construct a *split LTS* that distinguishes between the local and the shared state. To this end, we assume a labelling function π that maps each transition in an LTS to either Local for operations that only access the local state, or $\mathsf{Shared}(p \rightsquigarrow q)$ for operations that access both the local and the shared state. Note that for a program C we can construct such a labelling π_C from the annotations we introduced in Section 3: commands outside `atomic` blocks are mapped to Local and annotations at `atomic` blocks give the parameters of Shared.

Given a labelling π for an LTS $(\Sigma, \top, \Phi, T)$, we can define the corresponding split LTS as $(\Sigma^2, \top, \Phi \times \Sigma, T')$, where T' consists of fresh copies of transitions τ' for every transition $\tau \in T$. The program counter of a thread is always in its local state, hence, the set of states of the split LTS in which it has the final value is $\Phi \times \Sigma$. The transition functions for the split LTS are defined as follows. If $\pi(\tau) = \mathsf{Local}$, then τ' executes τ on the local state and preserves the shared state: $f_{\tau'}(l, s) = f_\tau(l) \times \{s\}$ if $f_\tau(l) \neq \top$, and $f_{\tau'}(l, s) = \top$ otherwise. If $\pi(\tau) = \mathsf{Shared}(p \rightsquigarrow q)$, then the execution of τ' follows the informal description of the execution of `atomic` blocks in Section 3:

$$f'_{\tau'}(l, s) = \bigcup \{(\mathsf{rest}_q(u), \mathsf{sat}_q(u) \cdot \mathsf{rest}_p(s)) \mid u \in f_\tau(l \cdot \mathsf{sat}_p(s))\}$$

if $\mathsf{sat}_p(s)$ is defined, $f_\tau(l \cdot \mathsf{sat}_p(s)) \neq \top$, and $\mathsf{sat}_q(u)$ is defined for all $u \in f_\tau(l \cdot \mathsf{sat}_p(s))$; otherwise, τ' faults: $f_{\tau'}(l, s) = \top$.

6.3 Validity in the split-state semantics

To define validity of triples in the split-state semantics, we have to define the meaning of interleaving computations of a split LTS $(\Sigma^2, \top, \Phi \times \Sigma, T)$ with actions of an environment changing the shared state according to a rely condition $\mathcal{R} \subseteq \mathcal{L}(\Sigma^2)$. We represent these computations with *traces* $\alpha \in \mathcal{L}(\Sigma^2 \times (\Sigma^2 \cup \{\top\}) \times (\{\mathbf{e}\} \cup T))$. The first two components of every letter in a trace define how the state of the LTS changes. The third component defines if the change was made by a transition of the LTS ($\tau \in T$) or the environment ($\mathbf{e}$). We require that the environment does not change the local state and does not fault, i.e., all $\mathbf{e}$-letters in a trace are of the form $((l, s), (l, s'), \mathbf{e})$.

We often need to project a trace α without error states to a word that records how the shared state is changed by a particular set of transitions $U \subseteq \{\mathbf{e}\} \cup T$. We define such a projection $\alpha|_U \in \mathcal{L}(\Sigma^2)$ as the image of α under the following homomorphism

$$h : \Sigma^2 \times \Sigma^2 \times (\{\mathbf{e}\} \cup T) \to \mathcal{L}(\Sigma^2)$$

$$h((l, s), (l', s'), \tau) = \begin{cases} (s, s'), & \tau \in U; \\ \varepsilon, & \text{otherwise} \end{cases}$$

where ε is the empty word. We write $\alpha{\downarrow}\sigma$ if α is a nonempty trace and its last letter is of the form (σ', σ, τ) for some σ' and τ.

DEFINITION 4 (Traces). *For a rely condition $\mathcal{R} \subseteq \mathcal{L}(\Sigma^2)$ and a split LTS $S = (\Sigma^2, \top, \Phi \times \Sigma, T)$, the set $\mathsf{tr}(S, \mathcal{R}, \sigma_0)$ of traces of S executed in an environment satisfying $\mathcal{R}$ starting from an initial state $\sigma_0 \in \Sigma^2$ is defined as the set of traces $\alpha \in \mathcal{L}(\Sigma^2 \times (\Sigma^2 \cup \{\top\}) \times (\{\mathbf{e}\} \cup T))$ of the following two forms:*

- *finite or infinite traces $\alpha = (\sigma_0, \sigma_1, \tau_0)(\sigma_1, \sigma_2, \tau_1)\ldots$, where $\sigma_i \neq \top$, $\alpha|_{\{\mathbf{e}\}} \in \mathcal{R}$, and if $\tau_i \neq \mathbf{e}$, then $\sigma_{i+1} \in f_{\tau_i}(\sigma_i)$; and*
- *finite traces $\alpha = \beta(\sigma_n, \top, \tau_n)$ for some $\beta = (\sigma_0, \sigma_1, \tau_0)$ $(\sigma_1, \sigma_2, \tau_1)\ldots(\sigma_{n-1}, \sigma_n, \tau_{n-1})$ such that $\beta|_{\{\mathbf{e}\}} \in \mathcal{R}$, $f_{\tau_n}(\sigma_n) = \top$, and if $\tau_i \neq \mathbf{e}$ for $i < n$, then $\sigma_{i+1} \in f_{\tau_i}(\sigma_i)$.*

The first case in this definition corresponds to safe traces, and the second to unsafe traces, i.e., those in which both the program

and its environment stop executing after the program commits a memory fault (the treatment of the later case relies on $\mathcal{R}$ being prefix-closed). Note that, since we assume that the scheduler is possibly unfair, the set of traces in this definition includes those in which S is preempted and is never executed again. Hence, the set of projections $\alpha\rfloor_T$ of traces $\alpha \in \mathsf{tr}(S, \mathcal{R}, \sigma_0)$ on the transitions of the LTS S, representing the guarantee condition provided by S, is prefix-closed.

Let $F_0(C)$, respectively $F_\mathtt{f}(C)$, be the $*$-conjunction over all the threads in a program C of formulae $\&\mathtt{pc} \mapsto \mathtt{pc}_0$, respectively $\&\mathtt{pc} \mapsto \mathtt{pc}_\mathtt{f}$, where $\mathtt{pc}$ is the program counter of the thread, $\mathtt{pc}_0$ is its initial value, and $\mathtt{pc}_\mathtt{f}$ is the final one. Note that $F_0(C)$ and $F_\mathtt{f}(C)$ do not restrict the shared state.

DEFINITION 5 (Validity).

$\mathcal{R}, \mathcal{G} \models \{P\}\ C\ \{Q\} \Leftrightarrow$
$\quad \forall \sigma_0 \in [\![P * F_0(C)]\!].\ \forall \alpha \in \mathsf{tr}(S, \mathcal{R}, \sigma_0).\ \forall \sigma.$
$\quad\quad (\alpha\downarrow\sigma \Rightarrow \sigma \neq \top) \wedge$ *(safety)*
$\quad\quad (\alpha\downarrow\sigma \wedge \sigma \in (\Phi \times \Sigma) \Rightarrow \sigma \in [\![Q * F_\mathtt{f}(C)]\!]) \wedge$ *(correctness)*
$\quad\quad (\alpha\rfloor_T \in \mathcal{G})$ *(guarantee)*

where $S = (\Sigma^2, \top, \Phi \times \Sigma, T)$ *is the split LTS constructed out of the LTS* $[\![C]\!]$ *using the labelling* π_C.

$\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \models \{P\}\ C_1 \| C_2\ \{Q\} \Leftrightarrow$
$\quad \forall \sigma_0 \in [\![P * F_0(C_1 \| C_2)]\!].\ \forall \alpha \in \mathsf{tr}(S_1 \| S_2, \mathcal{R}, \sigma_0).\ \forall \sigma.$
$\quad\quad (\alpha\downarrow\sigma \Rightarrow \sigma \neq \top) \wedge$ *(safety)*
$\quad\quad (\alpha\downarrow\sigma \wedge \sigma \in ((\Phi_1 \cap \Phi_2) \times \Sigma) \Rightarrow \sigma \in [\![Q * F_\mathtt{f}(C_1 \| C_2)]\!]) \wedge$ *(correctness)*
$\quad\quad (\alpha\rfloor_{T_1} \in \mathcal{G}_1) \wedge (\alpha\rfloor_{T_2} \in \mathcal{G}_2)$ *(guarantee)*

where $S_1 = (\Sigma^2, \top, \Phi_1 \times \Sigma, T_1)$ *and* $S_2 = (\Sigma^2, \top, \Phi_2 \times \Sigma, T_2)$ *are the split LTSes constructed out of the LTSes* $[\![C_1]\!]$ *and* $[\![C_2]\!]$ *using the labellings* π_{C_1} *and* π_{C_2}*, respectively.*

THEOREM 1. *The proof rules in Figure 5 preserve validity.*

COROLLARY 1. *If* $\mathcal{R}, \mathcal{G} \vdash \{P\}\ C\ \{Q\}$ *is derived from valid thread-local triples using the rules in Figure 5, then* $\mathcal{R}, \mathcal{G} \models \{P\}\ C\ \{Q\}$.

6.4 Soundness

We now relate the notion of validity with respect to a split LTS to validity with respect to the global LTS used to construct the split one. For a closed program (i.e., a program executing in isolation), we can formulate a global notion of validity of triples without rely and guarantee conditions as follows.

DEFINITION 6 (Validity with respect to a global LTS). *For* $p, q \subseteq \Sigma$ *and a command* C *such that* $[\![C]\!] = (\Sigma, \top, \Phi, T)$ *we define* $\models \{p\}\ C\ \{q\}$ *if for all* $u_0 \in p$ *and for any computation* $u_0, u_1, \ldots$ *of* $[\![C]\!]$ *we have* $u_i \neq \top$*, and if the computation is finite and ending with a state* $u \in \Phi$*, then* $u \in q$*. We define* $\models [p]\ C\ [q]$ *if* $\models \{p\}\ C\ \{q\}$ *and every computation of* $[\![C]\!]$ *starting from a state in* p *is finite.*

THEOREM 2. *Let* $[\![C]\!] = (\Sigma, \top, \Phi, T)$ *and* $S' = (\Sigma^2, \top, \Phi \times \Sigma, T')$ *be a corresponding split LTS with respect to any labelling* π_C*. Then*

- $\mathcal{R}, \mathcal{G} \models \{P\}\ C\ \{Q\}$ *implies* $\models \{\gamma([\![P * F_0(C)]\!])\}\ C\ \{\gamma([\![Q * F_\mathtt{f}(C)]\!])\}$,
- *If* $\mathcal{R}, \Diamond\bigcirc\mathsf{False} \models \{P\}\ C\ \{Q\}$ *implies* $\models [\gamma([\![P * F_0(C)]\!])]\ C\ [\gamma([\![Q * F_\mathtt{f}(C)]\!])]$,

where $\gamma(P) = \{l \cdot s \mid (l, s) \in P\}$ *for any assertion* P.

Theorem 2 and Corollary 1 show that the provability of triple (3.4) from valid thread-local triples in the proof system of Section 4 implies that the program $C'(k)$ terminates, and hence, the corresponding algorithm is lock-free. Similar soundness results can be formulated for obstruction-freedom and wait-freedom.

7. Related work

Our proof system draws on the classical circular and non-circular rely-guarantee rules for shared-variable concurrency [18, 29, 1] to achieve compositionality, and on separation logic (specifically, RGSep—a combination of rely-guarantee and separation logic [35, 11, 34]) to achieve modular reasoning in the presence of heap. Its technical novelty over previous rely-guarantee proof systems lies in our method of combining applications of circular and non-circular rules using judgements that distinguish between guarantees provided by different threads in a parallel composition.

Colvin and Dongol [7] have recently proved the most basic non-blocking algorithm, Treiber's stack [33], to be lock-free. They did this by manually constructing a global well-founded ordering over program counters and local variables of all the threads in the algorithm's most general client. Unfortunately, their method requires each operation to have at most one lock-free loop, which rules out more modern non-blocking algorithms, such as the HSY stack and Michael's list algorithm. Moreover, because their well-founded ordering is over the whole program, their method is non-modular and does not scale to the more realistic examples of the kind we consider in Section 5. In contrast, our method is modular, both in the treatment of threads and heaps. We can reason about every thread separately under simple assumptions about its environment that do not consider parts of the heap local to other threads. Furthermore, our method is fully automatic.

Kobayashi and Sangiorgi [19] have recently proposed a type-based method for checking lock-freedom in π-calculus. Their programming model and the notion of lock-freedom are different from the ones used for non-blocking data structures, which makes their results incomparable to ours. Moore [27] presents a proof of a progress property for a non-blocking counter algorithm in the ACL2 proof assistant. His proof is thread-modular, but the algorithm considered is extremely simple. McMillan [24] addresses the issue of circular dependencies among a class of liveness properties in the context of finite-state hardware model checking. He takes a different approach from ours to resolving the circularities by doing induction over time.

8. Conclusion

Wait-freedom, lock-freedom, and obstruction-freedom are the essential properties that make "non-blocking algorithms" *actually* non-blocking. We have proposed the first fully automatic tool that allows their developers to verify these properties. Our success was due to choosing a logical formalism in which it was easy to express proofs about non-blocking algorithms and then observing that proofs of the liveness properties in it follow a particular pattern.

We conclude by noting some limitations of our tool; lifting these presents interesting avenues for future work. First, we prove the soundness of our logic with respect to an interleaving semantics, which is inadequate for modern multiprocessors with weak memory models. It happens that even proving safety properties of programs with respect to a weak memory model is currently an open problem. Moreover, the published versions of concurrent algorithms *assume* a sequentially consistent memory model. In fact, most of non-blocking algorithms are incorrect when run on multiprocessors with weak memory models as published: one has to insert additional fences or (on x86) locked instructions for them to run correctly. In the future, we hope to address this problem,

building on a recent formalisation of weak memory model semantics [31]. Second, our tool can currently handle only list-based algorithms, because we use an off-the-shelf shape analysis. We believe that the methods described in this paper should be applicable to more complicated data structures as well, provided the necessary shape analysis infrastructure is available.

The above-mentioned limitations notwithstanding, this paper presents the first successful attempt to give modular proofs of liveness properties to complex heap-manipulating concurrent programs.

Acknowledgements. We would like to thank Josh Berdine, Mike Dodds, Tim Harris, Michael Hicks, Andreas Podelski, Moshe Vardi, Eran Yahav, and the anonymous reviewers for comments and discussions that helped to improve the paper.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–534, 1995.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[3] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV'06: Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 386–400. Springer, 2006.

[4] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[5] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS'07: Symposium on Logic in Computer Science*, pages 366–378. IEEE, 2007.

[6] C. Calcagno, M. J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS'07: Static Analysis Symposium*, volume 4634 of *LNCS*, pages 233–248. Springer, 2007.

[7] R. Colvin and B. Dongol. Verifying lock-freedom using well-founded orders. In *ICTAC'07: International Colloquium on Theoretical Aspects of Computing*, volume 4711 of *LNCS*, pages 124–138. Springer, 2007.

[8] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL'07: Symposium on Principles of Programming Languages*, pages 265–276. ACM, 2007.

[9] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE'04: Conference on Formal Techniques for Networked and Distributed Systems*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.

[10] B. Dongol. Formalising progress properties of non-blocking programs. In *ICFEM'06: Conference on Formal Engineering Methods*, volume 4260 of *LNCS*, pages 284–303. Springer, 2006.

[11] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP'07: European Symposium on Programming*, volume 4421 of *LNCS*, pages 173–188. Springer, 2007.

[12] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI'07: Conference on Programming Language Design and Implementation*, pages 266–277. ACM, 2007.

[13] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC'02: Symposium on Distributed Computing*, volume 2508 of *LNCS*, pages 265–279. Springer, 2002.

[14] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA'04: Symposium on Parallelism in Algorithms and Architectures*, pages 206–215. ACM, 2004.

[15] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[16] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS'03: International Conference on Distributed Computing Systems*, pages 522–529. IEEE, 2003.

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

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

[19] N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. In *CAV'08: Conference on Computer Aided Verification*, volume 5123 of *LNCS*, pages 80–93. Springer, 2008.

[20] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Conference on Logics of Programs*, volume 193 of *LNCS*, pages 196–218. Springer, 1985.

[21] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS'07: Static Analysis Symposium*, volume 4634 of *LNCS*, pages 419–436. Springer, 2007.

[22] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, 1992.

[23] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel (Abstract). *Operating Systems Review*, 26(2):108, 1992.

[24] K. L. McMillan. Circular compositional reasoning about liveness. In *CHARME'99: Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *LNCS*, pages 342–345. Springer, 1999.

[25] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA'02: Symposium on Parallelism in Algorithms and Architectures*, pages 73–82. ACM, 2002.

[26] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC'96: Symposium on Principles of Distributed Computing*, pages 267–275. ACM, 1996.

[27] J. S. Moore. A mechanically checked proof of a multiprocessor result via a uniprocessor view. *Formal Methods in System Design*, 14(2):213–228, 1999.

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

[29] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, pages 123–144. Springer, 1985.

[30] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[31] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86 multiprocessor machine code. This volume.

[32] H. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings*, 137(1):17–30, 1990.

[33] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[34] V. Vafeiadis. Modular fine-grained concurrency verification. PhD Thesis, University of Cambridge Computer Laboratory, 2008.

[35] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR'07: Conference on Concurrency Theory*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[36] M. Vardi. Verification of concurrent programs—the automata-theoretic framework. *Ann. Pure Appl. Logic*, 51:79–98, 1991.

[37] M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI'08: Conference on Programming Languages Design and Implementation*, pages 125–135. ACM, 2008.

[38] I. William N. Scherer, D. Lea, and M. L. Scott. Scalable synchronous queues. In *PPoPP'06: Symposium on Principles and Practice of Parallel Programming*, pages 147–156. ACM, 2006.