Proving Lock-Freedom Easily and Automatically

Xiao Jia Wei Li
Shanghai Jiao Tong University *

Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Lock-freedom is a liveness property satisfied by most non-blocking concurrent algorithms. It ensures that at any point at least one thread is making progress towards termination; so the system as a whole makes progress.

As a global property, lock-freedom is typically shown by global proofs or complex iterated arguments. We show that this complexity is not needed in practice. By introducing simple loop depth counters into the programs, we can reduce proving lock-freedom to checking simple local properties on those counters. We have implemented the approach in CAVE and report on our findings.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Concurrency; Lock-freedom; Verification; RGSep

1. Introduction

Non-blocking synchronisation is a style of multithreaded programming that is extensively used in concurrent data structure libraries, such as `java.util.concurrent`, to achieve good average performance and progress even under bad schedules. Unlike blocking synchronisation primitives, such as mutual exclusion locks, which cannot guarantee progress whenever one thread fails or stalls indefinitely, non-blocking synchronisation ensures some form of system-wide progress even if some threads of the system are not scheduled for an indefinitely long time period. In detail, the literature contains three such liveness properties:

- *Wait-freedom* [6] is the strongest of the three properties. It requires every operation to terminate provided it is scheduled possibly intermittently for a sufficient number of steps, irrespective of any other concurrent operations. Such behaviour is indeed desirable, but this requirement is very strong and often results in complicated and inefficient implementations.
- *Lock-freedom* [11] requires that from each point in time onwards, some library operation will be completed provided that at least one thread executing a library operation is scheduled. This requirement ensures that the program as a whole makes progress and is never blocked. From the point of view of an individual thread, however, very little is guaranteed: a thread executing a certain operation might never terminate if other operations continuously get in its way and are completed instead.
- *Obstruction-freedom* [8] is the weakest requirement, guaranteeing progress only if a thread is run in isolation for a sufficiently long amount of time. In the presence of contention, the entire system can livelock with each thread undoing the work done by other threads.

In this paper, we concentrate on lock-freedom, for two reasons. First, it is the most practically relevant property of the three, because it provides both a reasonably strong progress guarantee and allows for efficient implementations in practice. As a point of reference, most non-blocking concurrent algorithms in the "Art of Multiprocessor Programming" book [7] are lock-free, but not wait-free. Second, as noted by Gotsman et al. [4], proving lock-freedom is more challenging than the other two properties, because whereas wait-freedom and obstruction-freedom concern the termination of one thread, lock-freedom is a global property concerning the progress of the system as a whole.

The literature contains a few approaches for verifying lock-freedom, but these are unnecessarily complicated and difficult to automate. We discuss them in detail in Section 6, but it suffices to say that they involve either a global termination argument (e.g., [1, 2]) or non-trivial extensions to previously known concurrent program logics (e.g., [4, 9, 10]).

We propose a simpler method for verifying lock-freedom, which does not require a new program logic or any sophisticated program analysis. We instrument the source code with assignments to certain auxiliary variables and add assertions to each back-edge of the program's control flow graph. We prove in Coq that if all these assertions are never violated, then the program is lock-free. Using an off-the-shelf program analysis or program logic, we can then verify these assertions, thereby establishing lock-freedom.

1.1 A Simple Example: the Read-Compute-Update Pattern

To illustrate our proof technique for proving lock-freedom, as a first example, let us consider the "read, compute and update" (henceforth, RCU) pattern that lies at the core of many non-blocking algorithms, such as the Treiber stack [13]. This pattern consists of a loop first reading a shared variable X, then doing some computation, and finally checking whether the value of X has changed: if it has, we ignore the results of the computation and restart the loop, otherwise, if the value of X has not changed, we update X and return. (Typically, the check that X has the same value and the update both happen in one atomic step using a CAS instruction. We elide these details, as the atomicity is irrelevant for the progress argument.)

$$op \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{while(true)}\ \{ \\ \quad t := X;\ \ldots; \\ \quad \mathsf{if}\ (X == t)\ \{X := \ldots; \mathsf{break};\} \\ \} \end{array}$$

Consider now a system with multiple threads that repeatedly execute this operation. We can immediately observe that the opera-

* Currently at Google; the work was done during an internship at MPI-SWS.

CPP '15, January 13–14, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676724.2693179

tion is not wait-free: there exist executions that always happen to change the value of X between the initial read and the equality test, thereby making the operation potentially loop forever. The operation, however, is lock-free. The only reason why the loop might fail to terminate is if another thread writes to X. But the only way a thread can write to X is by then exiting and finishing its operation. Thus, at all times, if a thread performing *op* is scheduled, then at least one thread is making progress towards finishing its ongoing operation.

To capture this global argument, our idea is to instrument the code with an auxiliary variable, P, counting the number of times progress towards an operation's completion is made. That is, we increment P at every program step that does not appear on a looping path. By this we mean any program step that either (1) is not inside a loop, or (2) that is inside only one loop, but on a path that will exit the loop, such as the assignment to X in *op*. (In the instrumented pseudocode, we use angle brackets to denote that the assignment to X and the auxiliary increment of P are executed together atomically.)

```
while(true) {
  r := P; t := X; ...;
  if (X == t) {⟨X := ...; ++P⟩; break;}
  assert(P > r);
}
```

For each loop, we check that for every time the operation goes round a loop (and hence might not terminate), P was incremented (and hence some other operation made progress towards termination). The introduced code does not change the behaviour of the program inasmuch as the assertion checks succeed. The proof obligation that the assertion checks always succeed can be discharged by standard thread-modular reasoning techniques.

1.2 Contribution and Paper Outline

In the remainder of this paper, we formalise the construction just described and generalise it to handle operations with nested loops and more subtle local termination arguments. The beauty of our technique is that by introducing auxiliary code, we reduce the global nature of lock-freedom proofs to checks that can be discharged by thread-modular techniques.

We have also automated our approach by building on an existing verification tool, CAVE [14], a sound but incomplete thread-modular verification tool suitable for verifying fine-grained concurrent linked list algorithms. We have mildly adapted the frontend of CAVE to instrument the given concurrent library with the appropriate auxiliary variables and assertions needed to prove lock-freedom, before calling the backend to prove that these assertions are never violated, which in turn means that the original concurrent library is lock-free.

In more detail, in Section 2, we introduce a simple concurrent programming language and formally define lock-freedom. Then, in Section 3, we present the essence of our technique, formalise and prove its soundness, and discuss its implementation in CAVE. Next, in Section 4, we present a more refined version that is necessary for dealing with more advanced lock-freedom arguments. In Section 5, we present our experimental results, and then, in Section 6, we discuss related work, and conclude in Section 7.

2. Background

2.1 Programming Language

For the purposes of this paper, we will consider a minimal imperative programming language with an unspecified set of basic commands, BC, sequential and parallel composition, non-deterministic choice, a looping construct, and a break statement that terminates

$$\frac{(\sigma,\sigma') \in [\![BC]\!]}{\langle BC,\sigma\rangle \to \langle \mathsf{skip},\sigma'\rangle} \qquad \frac{\langle C_1,\sigma\rangle \to \langle C_1',\sigma'\rangle}{\langle C_1; C_2,\sigma\rangle \to \langle C_1'; C_2,\sigma'\rangle}$$

$$\frac{}{\langle \mathsf{skip}; C_2,\sigma\rangle \to \langle C_2,\sigma\rangle} \qquad \frac{}{\langle \mathsf{break}; C_2,\sigma\rangle \to \langle \mathsf{break},\sigma\rangle}$$

$$\frac{\langle C_1,\sigma\rangle \to \langle C_1',\sigma'\rangle}{\langle C_1 \parallel C_2,\sigma\rangle \to \langle C_1' \parallel C_2,\sigma'\rangle} \qquad \frac{\langle C_2,\sigma\rangle \to \langle C_2',\sigma'\rangle}{\langle C_1 \parallel C_2,\sigma\rangle \to \langle C_1 \parallel C_2',\sigma'\rangle}$$

$$\frac{}{\langle \mathsf{skip} \parallel \mathsf{skip},\sigma\rangle \to \langle \mathsf{skip},\sigma\rangle} \qquad \frac{}{\langle C_1 \oplus C_2,\sigma\rangle \to \langle C_1,\sigma\rangle}$$

$$\frac{}{\langle C_1 \oplus C_2,\sigma\rangle \to \langle C_2,\sigma\rangle} \qquad \frac{}{\langle \mathbf{L}_n(\mathsf{break}, C),\sigma\rangle \to \langle \mathsf{skip},\sigma\rangle}$$

$$\frac{}{\langle \mathbf{L}_n(\mathsf{skip}, C),\sigma\rangle \to \langle \mathbf{L}_n(C, C),\sigma\rangle}$$

$$\frac{\langle C_1,\sigma\rangle \to \langle C_1',\sigma'\rangle}{\langle \mathbf{L}_n(C_1, C_2),\sigma\rangle \to \langle \mathbf{L}_n(C_1', C_2),\sigma'\rangle}$$

Figure 1. Operational semantics of the language.

the innermost loop. Commands, C, in our language are given by the following grammar:

$$C ::= \mathsf{skip} \mid \mathsf{break} \mid BC \mid C_1; C_2 \mid C_1 \parallel C_2 \mid C_1 \oplus C_2 \mid \mathbf{L}_n(C_1, C_2)$$

Here, BC ranges over basic commands, such as assignments and "assume" statements. The combination of "assume" statements and non-deterministic choice allows us to encode conditionals in the standard way:

$$\mathsf{if}(B)\ C_1\ \mathsf{else}\ C_2 \stackrel{\mathsf{def}}{=} (\mathsf{assume}(B); C_1) \oplus (\mathsf{assume}(\neg B); C_2)$$

The final construct, $\mathbf{L}_n(C_1, C_2)$ is a somewhat non-standard looping construct that represents a partially executed loop, where C_2 is the loop body and C_1 is the remainder of the current loop iteration. (The n subscript will be used to record the value of a certain auxiliary program variable, but can be ignored for the time being.) The construct therefore first executes C_1 and then proceeds to execute C_2 repeatedly in a loop. The only way to exit a loop is by encountering a break statement. Normal while loops and standard non-deterministic loops, C^*, can be encoded as follows:

$$\mathsf{while}(B)\ C \stackrel{\mathsf{def}}{=} \mathbf{L}_0(\mathsf{skip}, \mathsf{if}(B)\ C\ \mathsf{else}\ \mathsf{break})$$
$$C^* \stackrel{\mathsf{def}}{=} \mathbf{L}_0(\mathsf{skip}, C \oplus \mathsf{break})$$

Operational Semantics Figure 1 presents the small-step operational semantics for our language as a reduction relation between configurations. Configurations are pairs, $\langle C,\sigma\rangle$, consisting of a command, $C \in \mathsf{Cmd}$ and a program state, $\sigma \in \mathsf{State}$.

The evaluation rules are standard. For basic commands, we assume their semantics is given by the function, $[\![\_]\!] : \mathsf{BasicCmd} \to \mathcal{P}(\mathsf{State} \times \mathsf{State})$, and we pick an arbitrary new state σ' such that $(\sigma,\sigma') \in [\![c]\!]$. This allows us to model both non-deterministic and blocking commands. The rules for sequential composition, parallel composition, and non-deterministic choice are straightforward: one point to note is that break commands propagate over sequential compositions. Finally, $\mathbf{L}_n(C_1, C_2)$ executes the current loop iteration, C_1, as long as possible. If $C_1 = \mathsf{break}$, the loop is exited, whereas if $C_1 = \mathsf{skip}$, it is restarted.

2.2 Specifying Lock-Freedom

A concurrent library consists of some initialisation code C_{init} and a collection of concurrent operations $C_1, \ldots, C_n$.

We define the most general client of the library to be a program that calls the library's initialisation code, and then creates k threads, each of which executes an unbounded number of the library's operations in any order inside a loop.

$$\mathsf{MGC}^k(C_{\mathsf{init}}, C_1, \ldots, C_n) \stackrel{\mathsf{def}}{=} C_{\mathsf{init}}; \underbrace{((C_1 \oplus \ldots \oplus C_n)^* \parallel \ldots \parallel (C_1 \oplus \ldots \oplus C_n)^*)}_{k \text{ threads}}$$

This client is most general in the sense that if we record the traces of calls to the library's operations from any concrete client of the library, the set of recorded traces would be a subset of those produced by the most general client.

Definition 1. *We say that a library is* lock-free *if every non-terminating execution of its most general client (for any k) has an infinite number of completed calls to the library's operations.*

Gotsman et al. [4] and later Hoffmann et al. [9] reduced proving lock-freedom to showing termination of a certain *bounded* most general client. Here, we use the theorem of Hoffmann et al. [9], as it is more general. (Gotsman et al.'s reduction is incorrect for implementations using thread identifiers or thread-local state.) The bounded most general client of a program is defined as follows:

$$\mathsf{BMGC}^{k,m}(C_{\mathsf{init}}, C_1, \ldots, C_n) \stackrel{\mathsf{def}}{=} C_{\mathsf{init}}; \underbrace{\left((C_1 \oplus \ldots \oplus C_n)^{\leq m} \parallel \ldots \parallel (C_1 \oplus \ldots \oplus C_n)^{\leq m}\right)}_{k \text{ threads}}$$

where

$$C^{\leq m} \stackrel{\mathsf{def}}{=} \mathsf{skip} \oplus C \oplus (C;C) \oplus \ldots \oplus \overbrace{(C;\ldots;C)}^{m \text{ times}} .$$

The definition is very similar to that of the most general client, except that the the number of calls to the library's operations per thread are bounded by m. Now we can state the reduction theorem as follows.

Theorem 1 (Hoffmann et al. [9])**.** *A concurrent library, Lib, is lock-free iff its bounded most general client, $\mathsf{BMGC}^{k,m}(Lib)$, terminates for all k and m.*

3. The Basic Scheme for Proving Lock-Freedom

Our proof technique for establishing termination and thereby lock-freedom consists of recording the auxiliary progress counters and proving that the program does not have any assertion violations. In this section, we first explain our technique by applying it on a more advanced example (§3.1). We then formalise it using an instrumented operational semantics (§3.2) and prove it suffices for verifying lock-freedom (§3.3). We finally discuss its implementation within the CAVE verifier (§3.4).

3.1 A Motivating Example: The Elimination Stack

We start with another motivating example, the elimination stack of Hendler et al. [5], that demonstrates the need for a slightly more advanced technique than what was needed for the simple "read-compute-update" (RCU) example of the introduction.

Abstracting over all details that are irrelevant for lock-freedom, the code of the elimination stack is shown in Figure 2. Both stack operations, `push` and `pop`, have the same structure. Within a loop, they first perform the RCU pattern trying to update S, the shared pointer to the top of the stack. When the check that the shared pointer has not changed succeeds, the operation terminates. When, however, the check fails (because of contention), the algorithm does not immediately try the same pattern again, but rather takes part in the elimination scheme. This scheme consists of a (nested) RCU loop, where the operation tries to register itself at some random

```
push/pop ≝ while(true) {
             t := S; ...;
             if(S == t) { S := ...; break; }
             i := ...; ...;
             while(true) {
               c := C[i]; ...;
               if(C[i] == c) { C[i] := ...; break; }
             }
             ...;
           }
```

Figure 2. The essential part of the elimination stack.

```
while(true) {r1 := P1;
  ⟨t := S; ++P2⟩; ...;
  if(S == t) { ⟨S := ...; ++P1; ++P2⟩; break; }
  ⟨i := ...; ++P2⟩; ...;
  while(true) {r2 := P2;
    c := C[i]; ...;
    if(C[i] == c) { ⟨C[i] := ...; ++P2⟩; break; }
    assert(P2 > r2);
  }
  ...;
  assert(P1 > r1);
}
```

Figure 3. Instrumentation of the elimination stack.

index i in the collision array, C. After it successfully does so, it waits a while and determines whether it collided with another operation of the opposite kind, in which case both operations terminate; otherwise, the operation goes round the outer loop and attempts to update S again.

To verify lock-freedom of this program, it is clear that we cannot simply use one auxiliary variable for counting the progress outside looping paths, because in this way we will not be able to show that the nested loop terminates. Instead, we need one auxiliary variable for each loop nesting depth. For the outer loop, we use P_1, which we increment on every atomic statement that does not appear on a looping path, while for the inner loop, we use P_2 which gets incremented on statements that do not appear on looping paths through nested loops.

Figure 3 shows the instrumented code for the elimination stack example. In general, our instrumentation requires one auxiliary variable per loop nesting depth. These variables get incremented on every atomic step at smaller loop nesting depths. So, for example in Figure 3, the assignment to S increments both P_1 and P_2, because it is on an exit path of the outer loop, whereas the assignment to i increments only P_2 because it is inside a looping path of depth one.

3.2 Instrumented Operational Semantics

We now make this auxiliary variable instrumentation formal. Given a function $P : \mathbb{N} \to \mathbb{N}$ representing the values of the auxiliary progress counters and a natural number d representing the current loop depth, we define $\mathsf{Cinc}(P, d) : \mathbb{N} \to \mathbb{N}$ to increment all counters above d:

$$\mathsf{Cinc}(P, d) \stackrel{\mathsf{def}}{=} \lambda x. \begin{cases} P(x) & \text{if } x \leq d \\ P(x) + 1 & \text{if } x > d \end{cases}$$

In essence, $\mathsf{Cinc}(P, d)$ is the same as P except that all progress counters at depths greater than d have been incremented by one.

$$\frac{(\sigma,\sigma') \in [\![BC]\!]}{d \vdash \langle BC,\sigma,P\rangle \to \langle \mathsf{skip},\sigma',\mathsf{Cinc}(P,d)\rangle} \qquad \frac{d \vdash \langle C_1,\sigma,P\rangle \to \langle C_1',\sigma',P'\rangle}{d \vdash \langle C_1; C_2,\sigma,P\rangle \to \langle C_1'; C_2,\sigma',P'\rangle} \qquad \frac{}{d \vdash \langle \mathsf{skip}; C_2,\sigma,P\rangle \to \langle C_2,\sigma,P\rangle}$$

$$\frac{}{d \vdash \langle \mathsf{break}; C_2,\sigma,P\rangle \to \langle \mathsf{break},\sigma,P\rangle} \qquad \frac{d \vdash \langle C_1,\sigma,P\rangle \to \langle C_1',\sigma',P'\rangle}{d \vdash \langle C_1 \parallel C_2,\sigma,P\rangle \to \langle C_1' \parallel C_2,\sigma',P'\rangle} \qquad \frac{d \vdash \langle C_2,\sigma,P\rangle \to \langle C_2',\sigma',P'\rangle}{d \vdash \langle C_1 \parallel C_2,\sigma,P\rangle \to \langle C_1 \parallel C_2',\sigma',P'\rangle}$$

$$\frac{}{d \vdash \langle \mathsf{skip} \parallel \mathsf{skip},\sigma,P\rangle \to \langle \mathsf{skip},\sigma,P\rangle} \qquad \frac{}{d \vdash \langle C_1 \oplus C_2,\sigma,P\rangle \to \langle C_1,\sigma,P\rangle} \qquad \frac{}{d \vdash \langle C_1 \oplus C_2,\sigma,P\rangle \to \langle C_2,\sigma,P\rangle}$$

$$\frac{}{d \vdash \langle \mathbf{L}_n(\mathsf{break}, C),\sigma,P\rangle \to \langle \mathsf{skip},\sigma,P\rangle} \qquad \frac{}{d \vdash \langle \mathbf{L}_n(\mathsf{skip}, C),\sigma,P\rangle \to \langle \mathbf{L}_{P(d)}(C, C),\sigma,P\rangle}$$

$$\frac{d + \mathsf{Idinc}(C_1) \vdash \langle C_1,\sigma,P\rangle \to \langle C_1',\sigma',P'\rangle}{d \vdash \langle \mathbf{L}_n(C_1, C_2),\sigma,P\rangle \to \langle \mathbf{L}_n(C_1', C_2),\sigma',P'\rangle}$$

$$\frac{d \vdash \langle C_1,\sigma,P\rangle \to \mathbf{abort}}{d \vdash \langle C_1; C_2,\sigma,P\rangle \to \mathbf{abort}} \qquad \frac{d \vdash \langle C_1,\sigma,P\rangle \to \mathbf{abort}}{d \vdash \langle C_1 \parallel C_2,\sigma,P\rangle \to \mathbf{abort}} \qquad \frac{d \vdash \langle C_2,\sigma,P\rangle \to \mathbf{abort}}{d \vdash \langle C_1 \parallel C_2,\sigma,P\rangle \to \mathbf{abort}}$$

$$\frac{d + \mathsf{Idinc}(C_1) \vdash \langle C_1,\sigma,P\rangle \to \mathbf{abort}}{d \vdash \langle \mathbf{L}_n(C_1, C_2),\sigma,P\rangle \to \mathbf{abort}} \qquad \frac{P(d) \le n}{d \vdash \langle \mathbf{L}_n(\mathsf{skip}, C),\sigma,P\rangle \to \mathbf{abort}}$$

Figure 4. Instrumented operational semantics of the language.

We say that a command, C, is *loop-free* if it does not syntactically contain any loop. Formally, this is defined by structural recursion as follows:

$$\begin{array}{lcl}
\mathsf{lpFree}(\mathbf{L}_n(C_1, C_2)) & \stackrel{\mathsf{def}}{=} & \mathsf{false} \\
\mathsf{lpFree}(C_1; C_2) & \stackrel{\mathsf{def}}{=} & \mathsf{lpFree}(C_1) \wedge \mathsf{lpFree}(C_2) \\
\mathsf{lpFree}(C_1 \parallel C_2) & \stackrel{\mathsf{def}}{=} & \mathsf{lpFree}(C_1) \wedge \mathsf{lpFree}(C_2) \\
\mathsf{lpFree}(C_1 \oplus C_2) & \stackrel{\mathsf{def}}{=} & \mathsf{lpFree}(C_1) \wedge \mathsf{lpFree}(C_2) \\
\mathsf{lpFree}(C_{\mathrm{other}}) & \stackrel{\mathsf{def}}{=} & \mathsf{true}
\end{array}$$

We say that a command, C, is *loop exiting*, whenever it can be syntactically determined that all executions of C will terminate and end with break (and not skip). This is formally defined by structural recursion as follows:

$$\begin{array}{lcl}
\mathsf{lpExit}(\mathsf{break}) & \stackrel{\mathsf{def}}{=} & \mathsf{true} \\
\mathsf{lpExit}(C_1; C_2) & \stackrel{\mathsf{def}}{=} & \mathsf{lpExit}(C_1) \vee (\mathsf{lpFree}(C_1) \wedge \mathsf{lpExit}(C_2)) \\
\mathsf{lpExit}(C_1 \oplus C_2) & \stackrel{\mathsf{def}}{=} & \mathsf{lpExit}(C_1) \wedge \mathsf{lpExit}(C_2) \\
\mathsf{lpExit}(C_{\mathrm{other}}) & \stackrel{\mathsf{def}}{=} & \mathsf{false}
\end{array}$$

Finally, we define $\mathsf{Idinc}(C)$ to return the loop depth increment needed when considering a loop whose current iteration is C. The function returns 0 if C is loop exiting and 1 otherwise.

$$\mathsf{Idinc}(C) \stackrel{\mathsf{def}}{=} \begin{cases} 0 & \text{if } \mathsf{lpExit}(C) \\ 1 & \text{if } \neg\mathsf{lpExit}(C) \end{cases}$$

Figure 4 presents the small-step operational semantics for our language as a reduction relation between extended configurations and indexed by the current loop depth, d. Extended configurations are triples, $\langle C,\sigma,P\rangle$, consisting of a command, $C \in \mathsf{Cmd}$, a program state, $\sigma \in \mathsf{State}$, and a function, $P \in \mathbb{N} \to \mathbb{N}$, representing the auxiliary progress counters. (Recall that there is one progress counter per loop nesting depth.)

The rules in Figure 4 follow those of Figure 1 and except as noted below simply pass the P and d components around. The exceptions are:

- The rule for basic commands calls $\mathsf{Cinc}(P, d)$ to increment all the progress counters at depths greater than the current one.
- When going round through the loop, the value of the current progress counter, $P(d)$, is recorded in the annotation of the loop command.
- The rule for evaluating the body of the loop does so at depth $d + \mathsf{Idinc}(C_1)$; that is, at the same depth for loop exiting C_1 and at depth $d + 1$ otherwise.

In addition to normal configurations, we also have a special **abort** configuration for detecting assertion violations. The only assertion violation of interest here is when a loop fails to make progress. Progress is checked in the last rule of Figure 4, which fails in case the current progress counter, $P(d)$, is not greater than the recorded one, n. The other aborting rules ensure that violations are propagated to the top level.

3.3 Soundness of the Instrumentation

We now turn to proving soundness of the instrumented semantics, that programs with no assertion violations always terminate.

First, we relate the instrumented and the normal semantics. We define the erasure, $|C|$ of a command C to replace all the n annotations in all $\mathbf{L}_n(\_,\_)$ subterms of C with 0.

Definition 2 (Erasure)**.** *Given a command, C, we define its* erasure*, $|C|$, by structural recursion on the command as follows:*

$$\begin{array}{lcl}
|\mathbf{L}_n(C_1, C_2)| & \stackrel{\mathsf{def}}{=} & \mathbf{L}_0(|C_1|, |C_2|) \\
|C_1; C_2| & \stackrel{\mathsf{def}}{=} & |C_1|; |C_2| \\
|C_1 \parallel C_2| & \stackrel{\mathsf{def}}{=} & |C_1| \parallel |C_2| \\
|C_1 \oplus C_2| & \stackrel{\mathsf{def}}{=} & |C_1| \oplus |C_2| \\
|C_{\mathrm{other}}| & \stackrel{\mathsf{def}}{=} & C_{\mathrm{other}}
\end{array}$$

We can prove the following proposition:

Proposition 2 (Instrumentation Erasure)**.**

- *If $d \vdash \langle C,\sigma,P\rangle \to \langle C',\sigma',P'\rangle$, then $\langle |C|,\sigma\rangle \to \langle |C'|,\sigma'\rangle$.*
- *If $\langle |C|,\sigma\rangle \to \langle C',\sigma'\rangle$, then for all d, P, there exist C'', P' such that $d \vdash \langle C,\sigma,P\rangle \to \langle C'',\sigma',P'\rangle$ and $C' = |C''|$.*

We continue with some auxiliary definitions. We define the loop depth of a command to be its maximum loop nesting depth.

$$\frac{}{C \sqsubseteq C} \qquad \frac{}{\mathsf{skip} \sqsubseteq BC} \qquad \frac{C_1 \sqsubseteq C_1'}{C_1; C_2 \sqsubseteq C_1'; C_2}$$

$$\frac{C_2 \sqsubseteq C_2' \quad \neg\, \mathsf{lpExit}(C_1')}{C_2 \sqsubseteq C_1'; C_2'} \qquad \frac{\mathsf{break} \sqsubseteq C_1}{\mathsf{break} \sqsubseteq C_1; C_2} \qquad \frac{C_1 \sqsubseteq C_1' \quad C_2 \sqsubseteq C_2'}{C_1 \parallel C_2 \sqsubseteq C_1' \parallel C_2'}$$

$$\frac{}{\mathsf{skip} \sqsubseteq C_1 \parallel C_2} \qquad \frac{C \sqsubseteq C_1}{C \sqsubseteq C_1 \oplus C_2} \qquad \frac{C \sqsubseteq C_2}{C \sqsubseteq C_1 \oplus C_2}$$

$$\frac{\mathsf{break} \sqsubseteq C_2}{\mathsf{skip} \sqsubseteq \mathbf{L}_n(C_1, C_2)} \qquad \frac{C_1 \sqsubseteq C_2}{\mathbf{L}_n(C_1, C_2) \sqsubseteq \mathbf{L}_{n'}(C_1', C_2)}$$

Figure 5. Auxiliary reduction relation, $C \sqsubseteq C'$.

Definition 3 (Loop Depth)**.** *Given a command, C, we define its* loop depth*, $\langle\!|C|\!\rangle_{\mathsf{depth}}$, by structural recursion as follows:*

$$\begin{aligned}
\langle\!|\mathbf{L}_n(C_1, C_2)|\!\rangle_{\mathsf{depth}} &\stackrel{\mathsf{def}}{=} 1 + \mathbf{max}\,(\langle\!|C_1|\!\rangle_{\mathsf{depth}}, \langle\!|C_2|\!\rangle_{\mathsf{depth}}) \\
\langle\!|C_1; C_2|\!\rangle_{\mathsf{depth}} &\stackrel{\mathsf{def}}{=} \mathbf{max}\,(\langle\!|C_1|\!\rangle_{\mathsf{depth}}, \langle\!|C_2|\!\rangle_{\mathsf{depth}}) \\
\langle\!|C_1 \parallel C_2|\!\rangle_{\mathsf{depth}} &\stackrel{\mathsf{def}}{=} \mathbf{max}\,(\langle\!|C_1|\!\rangle_{\mathsf{depth}}, \langle\!|C_2|\!\rangle_{\mathsf{depth}}) \\
\langle\!|C_1 \oplus C_2|\!\rangle_{\mathsf{depth}} &\stackrel{\mathsf{def}}{=} \mathbf{max}\,(\langle\!|C_1|\!\rangle_{\mathsf{depth}}, \langle\!|C_2|\!\rangle_{\mathsf{depth}}) \\
\langle\!|C_{\mathrm{other}}|\!\rangle_{\mathsf{depth}} &\stackrel{\mathsf{def}}{=} 0
\end{aligned}$$

Next, in Figure 5, we define when a command, C, is a reduction of another command, C', which we denote as $C \sqsubseteq C'$. The $\sqsubseteq$ relation is reflexive and includes a case $C' \sqsubseteq C$ whenever $\langle C, \sigma\rangle \to \langle C', \sigma'\rangle$. A few rules are noteworthy:

- The first interesting case is the rule deducing $C_2 \sqsubseteq C_1'; C_2'$. This checks that C_1 is *not* loop exiting, because otherwise C_2' is never executed.
- The next interesting case is the second to last rule. According to this rule, $\mathsf{skip} \sqsubseteq \mathbf{L}_n(C_1, C_2)$ holds only if the loop may be exited, i.e. if break is a reduction of the entire loop body, C_2.
- The final rule deduces $\mathbf{L}_n(C_1, C_2) \sqsubseteq \mathbf{L}_{n'}(C_1', C_2)$ by ignoring the current loop iteration C_1' and just checking that $C_1 \sqsubseteq C_2$. This is because the current iteration might terminate and another iteration may start from C_2. For the same reason, the recorded progress counter may differ.

We show that $\sqsubseteq$ is transitive, relates commands as reduced by the operational semantics, and preserves loop depths.

Proposition 3 (Properties of $\sqsubseteq$)**.**

- *If $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_3$, then $C_1 \sqsubseteq C_3$.*
- *If $\langle C, \sigma\rangle \to \langle C', \sigma'\rangle$, then $C' \sqsubseteq C$.*
- *If $C_1 \sqsubseteq C_2$ then $\langle\!|C_1|\!\rangle_{\mathsf{depth}} \leq \langle\!|C_2|\!\rangle_{\mathsf{depth}}$.*

We move on to some well-formedness properties of commands. We define *user* commands to have no partially executed loops, and *well-formed* commands to only have partially executed loops that could arise from a full loop.

Definition 4 (User Command)**.** *A command, C, is a* user command*, denoted $\mathsf{U}(C)$, iff for every loop $\mathbf{L}_n(C_1, C_2)$ it contains, $C_1 = \mathsf{skip}$ and $n = 0$. Formally,*

$$\begin{aligned}
\mathsf{U}(\mathbf{L}_0(\mathsf{skip}, C_2)) &\stackrel{\mathsf{def}}{=} \mathsf{U}(C_2) \\
\mathsf{U}(\mathbf{L}_p(C_1, C_2)) &\stackrel{\mathsf{def}}{=} \mathsf{false} \\
\mathsf{U}(C_1; C_2) &\stackrel{\mathsf{def}}{=} \mathsf{U}(C_1) \wedge \mathsf{U}(C_2) \\
\mathsf{U}(C_1 \parallel C_2) &\stackrel{\mathsf{def}}{=} \mathsf{U}(C_1) \wedge \mathsf{U}(C_2) \\
\mathsf{U}(C_1 \oplus C_2) &\stackrel{\mathsf{def}}{=} \mathsf{U}(C_1) \wedge \mathsf{U}(C_2) \\
\mathsf{U}(C_{\mathrm{other}}) &\stackrel{\mathsf{def}}{=} \mathsf{true}
\end{aligned}$$

Definition 5 (Well-Formed Command)**.** *A command, C, is* well-formed *iff it contains partially executed loops only at evaluated positions and for every such loop $\mathbf{L}_n(C_1, C_2)$, we have $C_1 \sqsubseteq C_2$ or $C_1 = \mathsf{skip}$. Formally,*

$$\begin{aligned}
\mathsf{WF}(\mathbf{L}_p(C_1, C_2)) &\stackrel{\mathsf{def}}{=} \mathsf{WF}(C_1) \wedge \mathsf{U}(C_2) \wedge (C_1 \sqsubseteq C_2 \vee C_1 = \mathsf{skip}) \\
\mathsf{WF}(C_1; C_2) &\stackrel{\mathsf{def}}{=} \mathsf{WF}(C_1) \wedge \mathsf{WF}(C_2) \\
\mathsf{WF}(C_1 \parallel C_2) &\stackrel{\mathsf{def}}{=} \mathsf{WF}(C_1) \wedge \mathsf{WF}(C_2) \\
\mathsf{WF}(C_1 \oplus C_2) &\stackrel{\mathsf{def}}{=} \mathsf{WF}(C_1) \wedge \mathsf{WF}(C_2) \\
\mathsf{WF}(C_{\mathrm{other}}) &\stackrel{\mathsf{def}}{=} \mathsf{true}
\end{aligned}$$

We say that a configuration is safe at a given loop nesting depth if it contains a well-formed command and the configuration can never reach an aborting state.

Definition 6 (Safety)**.** *A configuration, $\langle C, \sigma, P\rangle$, is* safe *at depth d iff C is well-formed and $\neg(d \vdash \langle C, \sigma, P\rangle \to^* \mathbf{abort})$.*

From Propositions 2 and 3, we deduce that safety is preserved by steps of the operational semantics.

Lemma 4 (Preservation)**.** *If $\langle C, \sigma, P\rangle$ is safe at loop depth d and $d \vdash \langle C, \sigma, P\rangle \to \langle C', \sigma', P'\rangle$, then $\langle C', \sigma', P'\rangle$ is also safe at d.*

We move on to the metric we use to prove termination. We take the domain of our termination metric to be infinite sequences over natural numbers, which we represent as functions $\mathbb{N} \to \mathbb{N}$.

Definition 7 (Lexicographic Ordering)**.** *Given two sequences $P, P' : \mathbb{N} \to \mathbb{N}$, we define $\prec_k$ to be the strict lexicographic order on the first k elements of P and P'.*

$$P \prec_k P' \iff \exists i \leq k.\, P(i) < P'(i) \wedge \forall j < i.\, P(j) = P'(j)\,.$$

It is easy to show (by induction on k and well-foundedness of $<$ on $\mathbb{N}$) that our lexicographic order, $\prec_k$, is well-founded.

Given a command, C, counters, $P : \mathbb{N} \to \mathbb{N}$, and the current depth $d \in \mathbb{N}$, we define $\langle\!|C, P|\!\rangle^d : \mathbb{N} \to \mathbb{N}$ to be the 'size' of a configuration $\langle C, \sigma, P\rangle$ at loop depth d.

Our goal is to show that the size of safe configurations decreases by each execution step according to the lexicographic order we have just defined. Formally, we want to establish the following.

Lemma 5. *If $d \vdash \langle C, \sigma, P\rangle \to \langle C', \sigma', P'\rangle$ and $\langle C, \sigma, P\rangle$ is safe at depth d, then $\langle\!|C', P'|\!\rangle^d \prec_{d + \langle\!|C|\!\rangle_{\mathsf{depth}}} \langle\!|C, P|\!\rangle^d$.*

We now proceed to the definition $\langle\!|C, P|\!\rangle^d$. Intuitively, the size of a command records the number of nodes in the command's abstract syntax tree at each loop nesting level.

The formal definition, found in Figure 6, is quite subtle in order to deal with partially executed loops and break statements, and ensure that Lemma 5 holds.

- In the sequential composition case, $\langle\!|C_1; C_2, P|\!\rangle^d$, if C_1 is a loop exiting command, then C_2 is dead code and hence we do not count its size.
- In the case for loops, $\langle\!|\mathbf{L}_n(C_1, C_2), P|\!\rangle^d(m)$, there are four subcases. If the current iteration, C_1, is bound to exit, then we return its size at the current depth d. Otherwise, we return some

$$
\begin{aligned}
\langle\!|\mathsf{skip}, P|\!\rangle^d(m) &\stackrel{\text{def}}{=} 0\\
\langle\!|\mathsf{break}, P|\!\rangle^d(m) &\stackrel{\text{def}}{=} 1\\
\langle\!|BC, P|\!\rangle^d(m) &\stackrel{\text{def}}{=} 1\\
\langle\!|C_1; C_2, P|\!\rangle^d(m) &\stackrel{\text{def}}{=} \begin{cases} \langle\!|C_1, P|\!\rangle^d(m) + 1 & \text{if } \mathsf{lpExit}(C_1)\\ \langle\!|C_1, P|\!\rangle^d(m) + \langle\!|C_2, P|\!\rangle^d(m) + 1 & \text{if } \neg\mathsf{lpExit}(C_1)\end{cases}\\
\langle\!|\mathbf{L}_n(C_1, C_2), P|\!\rangle^d(m) &\stackrel{\text{def}}{=} \begin{cases} \langle\!|C_1, P|\!\rangle^d(m) & \text{if } \mathsf{lpExit}(C_1)\\ \langle\!|C_2, P|\!\rangle^{d+1}(m) & \text{else if } m < d\\ \langle\!|C_1, P|\!\rangle^{d+1}(m) + \langle\!|C_2, P|\!\rangle^{d+1}(m) + \langle\!|C_2, P|\!\rangle^{d+1}(m) + 1 & \text{else if } n < P(d)\\ \langle\!|C_1, P|\!\rangle^{d+1}(m) + \langle\!|C_2, P|\!\rangle^{d+1}(m) & \text{otherwise}\end{cases}\\
\langle\!|C_1 \parallel C_2, P|\!\rangle^d(m) &\stackrel{\text{def}}{=} \langle\!|C_1, P|\!\rangle^d(m) + \langle\!|C_2, P|\!\rangle^d(m) + 1\\
\langle\!|C_1 \oplus C_2, P|\!\rangle^d(m) &\stackrel{\text{def}}{=} \langle\!|C_1, P|\!\rangle^d(m) + \langle\!|C_2, P|\!\rangle^d(m) + 1
\end{aligned}
$$

Figure 6. Definition of $\langle\!|C, P|\!\rangle^d : \mathbb{N} \to \mathbb{N}$ representing the size of the configuration $\langle C, \sigma, P\rangle$ at loop depth d.

combination of the sizes of C_1 and C_2 at depth $d+1$. More specifically, if $m < d$ (i.e., we are calculating the size of the loop at a depth smaller than the current depth), then we return a constant response, the size of C_2. (We chose the size of C_2 and not some other constant because the size must decrease whenever a transition makes C_1 loop exiting. So, we chose C_2 whose size is greater or equal to that of C_1.)

Now, if $m \geq d$, then we have to take the current loop iteration into account. To do so, we look at the current progress counter, $P(d)$, to determine whether the current loop iteration is the last one or not. If $P(d)$ is larger than n (the recorded of the counter when the iteration started), the loop is allowed to go round once again; otherwise, the current iteration is the last one.

- In the latter case (i.e., $P(d) \leq n$), we just take the total size of C_1 and C_2. We take into account the size of C_2 because we also want the size of a configuration, $\langle\!|C, P|\!\rangle^d$, to be monotonic (with respect to m) assuming that P is also monotonic.
- In the former case (i.e., if $n < P(d)$), we must define the size to be even bigger so that the size decreases when going round the loop (i.e., $\mathbf{L}_n(\mathsf{skip}, C) \to \mathbf{L}_{P(d)}(C, C)$). That is how we end up with the size of C_1 plus twice the size of C_2 plus 1.

To prove Lemma 5, we perform an induction over the reduction relation of the operational semantics. However, for the induction to work out, we first prove a stronger version of the lemma with the same premises and a stronger conclusion. The conclusion of the stronger version is that there exists $i \leq d + \langle\!|C|\!\rangle_{\mathsf{depth}}$ such that:

- for all $j \leq i$, $P(j) = P'(j)$; and
- for all $j \leq i$, $\langle\!|C', P'|\!\rangle^d(j) \leq \langle\!|C, P|\!\rangle^d(j)$; and
- for all $k \geq i$, if $P(j) = P'(j)$ holds for every $j \leq k$, then $\langle\!|C', P'|\!\rangle^d(k) \leq \langle\!|C, P|\!\rangle^d(k)$.

Lemma 5 then follows as a corollary of this stronger version.

As a consequence of Lemma 5 and the well-foundedness of $\prec$, we can prove that any program without assertion violations always terminates.

Theorem 6 (Termination)**.** *If $\langle C, \sigma, \lambda x.\ 1\rangle$ is safe at depth 0, then $\langle C, \sigma\rangle$ always terminates.*

As a corollary of this theorem and Theorem 1, we conclude that a library is lock-free if its bounded most general client has no assertion violations.

Corollary 7 (Lock-freedom)**.** *Given a concurrent library, Lib, if $\langle \mathsf{BMGC}^{k,m}(Lib), \sigma_0, \lambda x.\ 1\rangle$ is safe for all k and m, and valid initial states σ_0, then the library is lock-free.*

3.4 Implementation within CAVE

To automate proving lock-freedom, we have mildly adapted the implementation of CAVE [14]. CAVE takes as its input a program consisting of some initialisation code and a number of concurrent methods, which are all executed in parallel an unbounded number of times each. When successful, it produces a proof in RGSep [16] that the program has no memory errors and none of its assertions are violated at run time. Internally, it performs a thread-modular program analysis, RGSep action inference [15], that figures out a set of actions abstracting the updates to shared memory cells performed by the various operations.

We first run CAVE's action inference algorithm, which gives us a set of actions abstracting over the shared state changes performed by the program threads. CAVE also returns a mapping from the individual atomic program commands to the actions that they are responsible for.

We then calculate the maximum loop nesting depth of the program, M, and create a fresh auxiliary variable $level_{t,d}$ for each thread t and each loop nesting level $1 \leq d \leq M$. We use this variable to record whether the progress variable $P(d)$ was incremented since the start of the loop at depth d by thread t. We then perform the following instrumentation.

- At each loop header, we add the auxiliary assignment
$$level_{t,k} := \mathsf{false}$$
where t is the identifier of the current thread and k is the loop's nesting depth.
- At each loop back edge, we add the assertion check
$$\mathsf{assert}(level_{t,k})$$
where t and k are as before.
- For each action inferred by CAVE, we calculate the maximum loop nesting depth under which it is performed. Let that level be k. Then, we modify the action to set $level_{t',i} := \mathsf{true}$ for every t' and for every i such that $k < i \leq M$. For example, an action performed only outside loops would have $k = 0$, and can therefore set all the $level$ variables to true. On the other hand, an action that may be performed inside a single loop will have $k = 1$, and therefore cannot set the $level_1$ to true because that would enable a loop at the same level to avoid terminating.

```
typedef struct Node_s {
  int val;
  Node tl;
} *Node;

typedef struct Queue_s {
  Node head, tail;
} *Queue;

Queue new_queue(void) {
  Queue Q = malloc(...);
  Node nd = malloc(...);
  nd->tl = NULL;
  Q->head = nd;
  Q->tail = nd;
  return Q;
}
```

```
void enqueue(Queue Q, int value) {
  Node node, next, tail;
  node = new_node();
  node->val = value;
  node->tl = NULL;
  while(true) {
    tail = Q->tail;
    next = tail->tl;
    if (Q->tail == tail) {
      if (next == NULL) {
        if (CAS(&tail->tl,next,node))
          break;
      } else {
        CAS(&Q->tail,tail,next);
      }
    }
  }
  CAS(&Q->tail,tail,node);
}
```

```
int tryDequeue(Queue Q) {
  Node next, head, tail;
  int pval;
  while(true) {
    head = Q->head;
    tail = Q->tail;
    next = head->tl;
    if (Q->head == head) {
      if (head == tail) {
        if (next == NULL) return EMPTY;
        CAS(&Q->tail,tail,next);
      } else {
        pval = next->val;
        if (CAS(&Q->head,head,next))
          return pval;
      }
    }
  }
}
```

Figure 7. The Michael and Scott non-blocking queue implementation.

Finally, we rerun CAVE to verify the modified program thread-modularly under the updated set of actions. This, in effect, means that CAVE will check that the appropriate value $level_{t,k}$ variable was set in every loop iteration, meaning that there was interference by a statement from a concurrent thread that was at a lower loop nesting level, which as we have shown suffices to prove lock-freedom.

We note that since CAVE is both thread- and procedure-modular, it suffices to check each operation on its own, under the assumption they could be arbitrarily many other concurrent operations. We never need to explicitly quantify over the parameters k and m of the bounded most general client.

4. A Refined Scheme for Proving Lock-Freedom

The basic proof technique presented in the previous section works well for programs following the RCU pattern, such as the examples from Sections 1.1 and 3.1, but has a clear limitation that prevents its applicability to more complex examples.

The main problem is that it requires loops to terminate in a single iteration when there is no contention. While this assumption holds for the RCU patterns, it is clearly not true in general. Consider, for instance, the program:

$$i := 0;\\ \mathsf{while}(i < 2)\ \{i{+}{+};\}$$

While this program clearly always terminates, instrumenting the program as discussed in Section 3 leads to the following program

$$\langle i := 0; {+}{+}P_1\rangle;\\ \mathbf{L}_0(\mathsf{skip},\ p := P_1;\\ \qquad \mathsf{if}(i < 2)\ i{+}{+};\ \mathsf{else\ break};\\ \qquad \mathsf{assert}(P_1 > p);\ )$$

that has an obvious assertion violation.

In the remainder of this section, we discuss a refined proof technique that overcomes this evident limitation of the basic scheme.

4.1 Another Example: The Michael and Scott Queue

Before delving into our refined proof technique, let us first consider another motivating example. Figure 7 presents a concurrent non-blocking queue implementation due to Michael and Scott [12].

The queue is represented by two pointers into a null-terminated singly-linked list. The first pointer (`Q->head`) points to the beginning of the list and is updated by dequeueing operations. The second pointer (`Q->tail`) is used to find the end of the list so that `enqueue` can locate the last node of the list. It does not necessarily point to the last node of the list, but it can lag behind. This is because `enqueue` first appends a node onto the list with its first `CAS` instruction, and then modifies `Q->tail` with its final `CAS` instruction.

Whenever an enqueue or a dequeue operation notices that the tail pointer lags behind, it firsts perform a `CAS` to advance the tail pointer it forward (cf. the underlined instructions in Figure 7) before proceeding with doing its own work. What complicates the lock-freedom proof is that this advancement of the tail pointer happens inside a loop and can prevent a concurrent dequeue from terminating.

However, one can easily argue that if no further elements are added to the queue, the tail pointer cannot lag behind more than the number of elements that have been ever added to the queue, which for any given point in time is some finite number. The tail pointer can therefore be advanced at most a finite number of times. (With a more careful argument, one can show that the tail pointer can only be at most one node behind the real tail of the list, but we do not need such a precise bound for proving lock-freedom.)

4.2 The Refined Proof Method

To reason about the aforementioned examples, we therefore also need to take into account loops that in the absence of contention do not terminate immediately, but for which we can still argue that they will eventually terminate.

To do so, we adopt one of the standard techniques for proving termination of sequential programs, namely the use of ranking functions. A ranking function, f, is a function from program states to a well founded domain (typically, $\mathbb{N}$ with $<$) such that whenever executing the body of a loop changes the state from σ to σ', we have that $f(\sigma') < f(\sigma)$.

We assume that a ranking function is attached to each loop. When executing a loop (i.e., in the $\mathbf{L}(C_1, C_2)$ construct), we now record not only the value of the progress counter at the beginning of the loop iteration, but also the state at that time, as well as the ranking function. When having to check whether the loop can be restarted or not, we relax the check to allow reentering the loop if the ranking of the state has decreased. Formally, we replace the last

rule of Figure 4 with the following:

$$\frac{P(d) \leq n \qquad f(\sigma') \not\leq f(\sigma)}{d \vdash \langle \mathbf{L}_{f,n,\sigma}(\mathsf{skip}, C), \sigma', P\rangle \rightarrow \mathbf{abort}}$$

We have added another premise to the rule, thereby making programs abort less frequently. One can show that even with this stronger abort rule, if a certain program never aborts, then it always terminates. To prove this statement, however, we need a more complex termination metric, a lexicographic product that combines the metric on the progress variables and on the recorded states.

Returning to Michael and Scott queue, which function should we choose as a ranking function? Naturally, we could take the length of the path through the heap following the `next` fields from `Q->tail` to `NULL`. The length of this path is decremented whenever the tail pointer is advanced, and is unaffected by local accesses; the only access that may increase this path length is enqueueing a new node, but this only happens on a loop-free code path.

A bit more generically, we can take as a ranking function the sum of the lengths of all distinct acyclic paths between any two pairs of nodes in the heap. We can further instrument the program to track how that sum is affected by individual heap updates. Specifically, whenever we have the field assignment $x \rightarrow f := y$, and we know (1) that x is not part of a cyclic data structure and (2) that there was a path from x to y of at least length two, then we know that the field assignment decrements the length of heap paths. Otherwise, we conservatively say that the field assignment might increment the length of paths in the heap.

4.3 Implementation within CAVE

For each action that Cave generates, we calculate whether the action decreases the total length of the heap paths. If so, we call the action *decreasing*; we call it *possibly increasing* otherwise.

- We introduce two additional auxiliary variables, $inc_{t,i}$ and $dec_{t,i}$, for each thread t and each loop nesting depth of the program, i.e. $1 \leq i \leq M$.
- We augment every decreasing action to set $dec_{t,k} := \mathsf{true}$ for every t and every $1 \leq k \leq M$, and every possibly increasing one to set $inc_{t,k} := \mathsf{true}$ for every t and every $1 \leq k \leq M$.
- At the loop header, we initialise the increment and decrement variables of the current thread's current loop nesting depth to false.
- At the loop back edges, we check that either the progress flag appropriate to the loop level has been set, or the heap paths have been decremented and not possibly incremented. Formally, we add the check:

  $$\mathsf{assert}(level_{t,k} \vee (dec_{t,k} \wedge \neg inc_{t,k}))$$

  where t and k are the current thread identifier and loop nesting depth respectively.

5. Evaluation

As discussed in Sections 3.4 and 4.3, we have implemented a tool for proving lock-freedom as an extension of CAVE. We have applied our tool to verify several concurrent algorithms from the literature. These include two integer counters: a simple CAS-based one and one with a secondary counter as a back-off scheme for high contention cases; three non-blocking stack algorithms: the Treiber stack [13], a variant of it using a DCAS operation, and the Hendler et al. [5] elimination stack; a non-blocking counter; and three concurrent queues: the Michael and Scott queue [12] from Section 4.1, the DGLM queue [3], and a variant of the Michael and Scott queue with a simpler `tryDequeue` implementation.

Algorithm	LOC	Techniques	Time
CAS counter	41	Level	0.02s
Double counter	64	Level	0.11s
Treiber stack	52	Level	0.11s
DCAS stack	52	Level	0.11s
Elimination stack	76	Level	1.22s
MS queue	83	Level+HeapDec	3.01s
DGLM queue	83	Level+HeapDec	3.92s
MSV queue	74	Level+HeapDec	2.85s

Table 1. Experimental results for the automated lock-freedom prover. For each example, we report the number of lines of code excluding blank lines and comments, the techniques necessary for the verification, and the total verification time.

Table 1 reports on our experimental results. The tool has successfully proved lock-freedom of all of these algorithms. The basic loop level technique described in Section 3 was sufficient for verifying the counter and stack implementations, but in order to verify the concurrent queue algorithms, we needed the refined proof technique introduced in Section 4. In both cases, taking the lengths of heap paths as a termination metric was sufficient.

We have also tested our tool on variants of these algorithms that support blocking methods, such as a `dequeue` operation that busily waits whenever the queue is empty. As expected, in all these cases, the tool failed to prove lock-freedom.

We would also like to stress that our automated proof technique for proving lock-freedom is heuristic, and while it may work very well for standard examples, it is easy to defeat it. Specifically, we calculate loop nesting depths as a heuristic way of detecting when interference by another thread makes enough progress to block the termination of a certain loop. This heuristic can easily be confused by adding a clone of one of the looping methods of a concurrent library, but wrapping its code inside a terminating loop. Then, our heuristic will fail to detect this, and will not be able to prove the modified library lock-free. Needless to say, however, we have not encountered such cases in practice.

6. Related Work

The literature contains a few approaches for proving lock-freedom.

The most closely related approach to ours is by Colvin and Dongol [2]. They verified the MSqueue algorithm shown in Figure 7 by determining which program statements make progress towards termination of an operation and constructing a global well-founded order that is decremented whenever a program makes a step different from those statements. This approach is very nice and corresponds closely to ours in the case where the operations do not have nested loops. For programs with nested loops, such as the HSY stack, however, coming up with a global well-founded order that ensures progress can be rather difficult. In such cases, our technique of having a counter per loop nesting level can substantially simplify the proof. Further, even for algorithms without nested loops, such as the MSqueue, we do not really need to come up with a custom global well-founded order because the default heap path decrementing heuristic explained in Section 4.2 suffices. In an earlier paper [1], the same authors verified the Treiber stack using just a global well-founded order, but as the authors themselves admit in their next paper [2] the global approach it too difficult to apply for more advanced algorithms.

In a very different style, Gotsman et al. [4] presented an automatic approach for verifying lock-freedom by reducing the lock-freedom problem to termination. They then presented an advanced combination of separation logic, rely-guarantee, and linear temporal logic, and used it to prove lock-freedom of the Treiber stack,

the HSY stack and a few more algorithms. They also reported on a prototype implementation, which is sadly not available.

An instructive example to compare their approach with ours is the elimination stack [5]. Gotsman et al. [4] prove lock-freedom in an iterated fashion. First, they show that the bounded most general client can perform only a finite number of changes to the central stack object. Next, they show that under the assumption that the threads eventually stop updating the central stack object, the number of updates to the collision array is also finite. Finally, assuming that updates to both the central stack and the collision array will eventually end, they show that the bounded most general client program terminates. In our method, however, this iterative argument is unnecessary, as it is implicitly captured in the different auxiliary progress variables we introduce per loop nesting depth.

Later, Hoffmann et al. [9] showed that Gotsman et al.'s reduction is incorrect for implementations using thread identifiers or thread-local state, and propose an alternative reduction to termination that is correct for such implementations. They then proposed an extension of concurrent separation logic with permission tokens for reasoning about termination. These tokens act like reservations from a shared pool of fuel. By construction, each loop iteration consumes one unit of fuel, and so each thread has to have enough tokens (units of fuel) in its precondition to guarantee that it will terminate successfully. Threads are allowed to transfer any unused units of fuel to another thread, thereby allowing that second thread to run for longer. While the underlying idea of this logic is nice and clear, unfortunately proofs in this logic are not entirely straightforward and currently only manual. The main difficulty is coming up with the right transfer of fuel reservations between threads and encoding this using CSL's resource invariants.

More recently, Liang et al. [10] have developed a program logic for proving termination-preserving refinement of concurrent programs, which can be used to prove lock-freedom of concurrent libraries. As far as proving termination and lock-freedom is concerned, they use a progress token assertion, $\mathsf{wf}(n)$, in a similar way to Hoffmann et al. [9].

While using any of the aforementioned approaches, one can in principle prove lock-freedom for all our examples, the proofs one gets are more complicated than necessary and are consequently hard to derive automatically. In contrast, our approach does not aim at generality, but is geared towards automation. After the instrumentation with auxiliary variables, any existing concurrent program verifier can be used to check the absence of assertion violations, thereby showing lock-freedom.

7. Conclusion

In this paper, we have discussed how to verify lock-freedom by instrumenting the program with suitable auxiliary variables. Using our approach we have successfully verified lock-freedom of a number of stack and queue algorithms. In principle, one can also apply our technique to other lock-free algorithms, such as list- or tree-based implementations of sets. Our tool, however, cannot yet verify such implementations because it does not have any support for inferring suitable ranking functions.

In the future, it would also be useful to see whether introducing similar auxiliary variables could assist in proving liveness properties for locked-based concurrent libraries, such as deadlock-freedom and starvation-freedom under some fairness assumptions about the scheduler.

Acknowledgements

We would like to thank Marko Doko and the anonymous CPP'15 reviewers for their helpful feedback. We acknowledge support from the EC FET project ADVENT.

References

[1] Robert Colvin and Brijesh Dongol. Verifying lock-freedom using well-founded orders. In *ICTAC 2007: 4th International Colloquium on Theoretical Aspects of Computing*, pages 124–138, 2007.

[2] Robert Colvin and Brijesh Dongol. A general technique for proving lock-freedom. *Sci. Comput. Program.*, 74(3):143–165, 2009.

[3] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE 2014: Formal Techniques for Networked and Distributed Systems*, volume 3235 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2004. .

[4] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 16–28. ACM, 2009. .

[5] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.*, 70(1):1–12, 2010. .

[6] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[7] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008. ISBN 978-0-12-370591-4.

[8] Maurice P. Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 522–529. IEEE, 2003.

[9] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Proceedings of the 28th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS)*, pages 124–133. IEEE, 2013.

[10] Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014)*. ACM, 2014.

[11] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *ACM SIGOPS Operating Systems Review*, 26(2):108, 1992.

[12] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC 1996: 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275. ACM, 1996. .

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

[14] Viktor Vafeiadis. Automatically proving linearizability. In *CAV 2010: 22nd Int. Conference on Computer Aided Verification*, pages 450–464. Springer, 2010.

[15] Viktor Vafeiadis. RGSep action inference. In *VMCAI 2010: 11th Int. Conference on Verification, Model Checking, and Abstract Interpretation*, pages 345–361. Springer, 2010.

[16] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007: 18th International Conference on Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007. .