Formal Verification of a Lock-Free Stack with Hazard Pointers

Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg
{tofan,schellhorn,reif}@informatik.uni-augsburg.de

Abstract. A significant problem of lock-free concurrent data structures in an environment without garbage collection is to ensure safe memory reclamation of objects that are removed from the data structure. An elegant solution to this problem is Michael's hazard pointers method. The formal verification of concurrent algorithms with hazard pointers is yet challenging. This work presents a mechanized proof of the major correctness and progress aspects of a lock-free stack with hazard pointers.

1 Introduction

Non-blocking implementations of concurrent data structures avoid major problems associated with blocking, such as convoying, deadlocks or priority inversion. In particular, lock-free [1] algorithms guarantee termination of some operation in a finite number of steps, even when individual operations are arbitrarily delayed or fail. Their main correctness property linearizability [2], ensures that each operation appears to take effect instantly at one step (the linearization point) between its invocation and response. Thus, from an external point of view, a linearizable operation executes atomically and can be used in a modular way. In addition, performance results show that lock-free implementations can outperform their lock-based counterparts significantly in the presence of contention or multiprogramming. These properties are even more important as multi-core architectures have become mainstream.

The advantages of lock-free implementations come at the price of an increased complexity to develop and verify them. These data structures are often used in programming environments without support for garbage collection (GC). There, the problem of safe memory reclamation of objects that have been removed from the data structure imposes significant additional challenges on design and verification. Memory occupied by a removed object can not be simply deallocated (e.g., using a *free* library call in C / C++) as other processes typically still access this object in their operations. The possible concurrent reuse of locations introduces a further fundamental problem of lock-free algorithms, the ABA-problem [3]. It becomes manifest in subtle errors such as wrong return values or data structure corruption, as we explain in Section 3.1 for a lock-free stack.

Several memory reclamation schemes that compensate the absence of GC exist. Hazard pointers [4] enable safe memory reclamation by extending concurrent

algorithms with their own local, non-blocking garbage collection. The reclamation technique is applicable to a class of important concurrent algorithms. This work analyzes the central properties of the hazard pointers method and then applies the results to verify a well-known lock-free stack that uses hazard pointers. Proving safe memory reclamation and ABA-avoidance for such a stack has been declared a challenge for program verification [5].

Our main contribution is an intuitive verification that exploits the central properties of Michael's reclamation scheme. The proof is mechanized in the interactive theorem prover KIV [6] and addresses all major aspects: memory-safety, ABA-prevention as well as preservation of linearizability and lock-freedom of the stack with hazard pointers. We apply temporal logic and local rely-guarantee reasoning, but use neither complex history variables nor reasoning about the temporal past, as in other approaches (cf. Section 7). The proofs reveal that the correctness of the reclamation scheme can be expressed in terms of two contending processes. A further novel insight is that its relation to GC can be exploited to reuse central correctness arguments under GC.

To keep the presentation readable, we do not detail every formal aspect. In particular, the verification and an in-depth description of the applied decomposition theory is omitted. Further details can be found in [7]; a complete presentation that includes all KIV-proofs is available online [8].

The remainder of this paper is organized as follows: Section 2 gives an introduction to hazard pointers. Section 3 specifies the main case study of this paper, the extended stack algorithm. Section 4 briefly introduces the verification framework that forms the logical base for the applied decomposition theory, which is described in Section 5. Section 6 shows four central properties of the hazard pointers method and their specialization to formal verification conditions in the case study. Section 7 presents related work and a comparison. Finally, Section 8 concludes with a summary and discussion of the main results.

2 The Hazard Pointers Method

Figure 1 illustrates hazard pointers: (1) processes $p, q, \ldots$ can concurrently allocate and insert new objects NEW to a lock-free data structure LDS. Every process p collects the memory of objects r that it removes from LDS in a local pool of retired locations RL_p. These locations are candidates for deallocation. However, the contending use of these retired locations must be considered first.

(2) shows that each process is associated with a fixed (small) number of multi-reader single-writer shared pointers, so called hazard pointers. All hazard pointers of all processes are contained in a hazard pointer record HPR. By setting one of its hazard pointers to a location r, process p signals other contending processes not to deallocate this location. Crucially, to ensure that this signal is indeed considered, p subsequently checks whether r is still part of LDS. Only if this check – called hazard pointer validation – succeeds, p enters a hazardous code region where it accesses r.

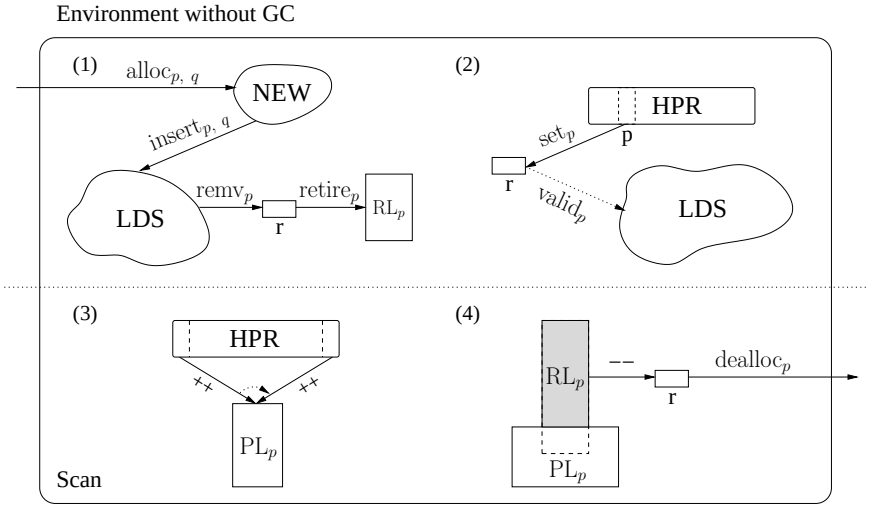

Fig. 1: Michael's hazard pointers method.

To deallocate memory, a process p executes a scan operation in two phases (3) and (4). In (3), it consecutively collects all hazard pointers of all processes in a local pointer list PL_p by traversing *HPR*. In (4), all retired memory locations r that were not found during this traversal ($r \in RL_p - PL_p$), are freed to the environment's memory management system for arbitrary reuse.

A properly extended lock-free algorithm with hazard pointers has the following central correctness property:

A validated hazard pointer is not concurrently freed.	(1)

This is because at the time of its successful validation, a hazard pointer is in *LDS* and hence in no retired list. Consequently, no currently running scan will deallocate it. After its successful validation, a hazard pointer might be concurrently retired, while still being used. Yet it is not freed, since the retiring process collects the pointer during its traversal of *HPR*. (We intuitively formalize this central argument in Section 6.)

3 A Lock-Free Stack with Hazard Pointers

3.1 The Lock-Free Stack

Instead of using locks, lock-free algorithms typically utilize atomic synchronization primitives such as the widely supported single-word CAS (Compare-And-Swap) instruction. A CAS compares a shared value SV with an older local copy

of it *Old*, called snapshot. If these values are equal, then SV is updated to a new value *New* and true is returned; otherwise false is returned.

CAS($Old, New; SV, Succ$) {
if* $SV = Old$ **then** $\{SV := New,\ Succ := true\}$ **else** $Succ := false\}$

Throughout this work, we use formal KIV-specifications to describe programs and thereby explain the introduced syntax. In the specification of CAS, the semicolon separates input from in-output parameters; the comma indicates parallel assignments and in **if*** evaluating the if-condition requires no extra step.

Figure 2 illustrates the lock-free stack which provides concurrent push and pop operations. The shaded code in pop, the scan and reset operations can be ignored for now. The algorithm is a prime example of a lock-free data structure, taken from Michael [4] and attributed to Treiber [3]. The shared stack is a singly linked list of cells – pairs of values and locations with .val and .nxt selector functions – in the application's memory heap H. The heap is a partial function from locations $r : ref$ (with $null \in ref$) to cells with standard operations, e.g., $H[r, ?]$ is allocation with arbitrary content "?", $r \in H$ tests if r is allocated, $H[r]$ is lookup and $H - r$ deallocation. A shared variable *Top* points to the top cell of the stack.
Whenever a process executes a push, it first allocates a new cell *UNew* (lines U3 / U4 execute in one step) and initializes it with input value *In*. Then it repeatedly tries to CAS the shared top to point to this new cell (lines U6 – 9). A pop reads the shared top (if this snapshot is null, the special value *empty* is returned) and locally stores the snapshot's next reference which becomes the target of the subsequent CAS. If it succeeds, the top cell is removed from the stack and its value is returned. Variables *UNew*, *USucc*, *OTop* and *OSucc* are local variables of "pUsh" resp. "pOp". They are defined as in-output parameters instead of using **let**, to allow us to reason about them.

Simply deallocating a removed cell at the end of pop can cause contending pop-processes to dereference an illegal snapshot pointer. If the reference is concurrently reused, an ABA-problem can occur: suppose that a pop-process p takes a snapshot of the top pointer when the stack consists of exactly one cell at location A. Process p is delayed after setting *ONxt* to null in line O12 for another process q, which executes a successful pop, freeing A. Subsequently, q executes two successful push operations, thereby allocating reference B and then again A. Then p is rescheduled and its CAS operation in line O13 erroneously succeeds, violating the semantics of pop.

3.2 The Extended Stack

Applying the hazard pointers technique requires no modification of the push operation. The pop operation requires one hazard pointer to cover the hazardous usage of the snapshot pointer *OTop* in lines O12 and O13. This hazard pointer is atomically set in line O9, using the shared hazard pointer record $HPR : \mathbb{N} \to ref$ and the identifier $Id : \mathbb{N}$ of the current process. In line O10, before any hazardous

```
U1  Push(In; UNew, USucc, Top, H) {
U2    let UTop = ? in {
U3      choose r with (r ≠ null ∧ r ∉ H) in {
U4        UNew := r, H := H[r, ?], USucc := false;
U5        H[UNew].val := In;
U6        while ¬ USucc do {
U7          UTop := Top;
U8          H[UNew].nxt := UTop;
U9          CAS(UTop, UNew; Top, USucc)}}}}

O1  Pop(; Id, Hazard_pc, OTop, OSucc, RL, Top, H, HPR, Out) {
O2    let ONxt = ?, Lo = empty in {
O3      OSucc := false;
O4      while ¬ OSucc do {
O5        OTop := Top, Hazard_pc := false;
O6        if OTop = null then {
O7          OSucc := true
O8        } else {
O9          HPR(Id) := OTop;
O10         if* OTop = Top then {
O11           Hazard_pc := true;
O12           ONxt := H[OTop].nxt;
O13           CAS(OTop, ONxt; Top, OSucc)}}}
O14     if OTop ≠ null then {
O15       Lo := H[OTop].val′;
O16       RL := OTop + RL, Hazard_pc := false}
O17     Out := Lo}}

S1  Scan(; Scan, BefInc_pc, Lid, Lhp, PL, RL, H, HPR) {
S2    PL := [], Scan := true;
S3    while Lid ≤ MAXID do {
S4      Lhp := HPR(Lid), BefInc_pc := true;
S5      if Lhp ≠ null then {
S6        PL := Lhp + PL}
S7      Lid := Lid + 1, BefInc_pc := false};
S8    while Scan do {
S9      choose r with (r ∈ RL − PL) in {
S10       RL := RL − r, H := H − r}
S11     ifnone Scan := false, Lid := 0}}

R1  Reset(; Id, HPR) {HPR(Id) := null}
```

Fig. 2: A lock-free data-stack with hazard pointers.

usage, the hazard pointer is validated. Crucially, only after this test succeeds, it can be guaranteed that the snapshot cell is not concurrently freed and possibly reused. An additional boolean flag $Hazard_{pc}$ marks the hazardous code region in which the validated hazard pointer equals (covers) the snapshot $OTop$. This

simple auxiliary variable is required in the verification only, since our logic does not use program counters. In line O16, a location that has been removed from the stack is added to a local list of retired locations RL.

Operation *Scan*, characterized by boolean flag *Scan*, frees retired locations that are not concurrently used. In its first loop, a scan sequentially traverses the hazard pointer record, reading each hazard pointer and collecting it in a further local pointer list PL, where constant MAXID denotes the greatest occuring process identifier. This includes atomically taking a snapshot *Lhp* of the *HPR* entry at process index *Lid* ($BefInc_{pc}$ is a further simple program counter substitute used in the proofs). In the second loop, retired memory locations that are not in PL are consecutively deallocated.

To simplify verification while maintaining the core ideas of Michael's algorithm, our version of the extended stack uses several algebraic data structures. In particular, we use a function to model the hazard pointer record, while Michael proposes a singly linked heap list. In the second loop of the scan operation, the **choose** summarizes some merely local steps that are required to determine the deallocable references $RL - PL$. This avoids some standard sequential reasoning. Furthermore, we slightly generalize Michael's version, by allowing a scan to be performed arbitrarily between stack operations, while Michael calls a scan at the end of pop, depending on the current number of retired locations. As a further minor extension, we consider the possible reset of a hazard pointer *Reset* between executions of push, pop or scan, while the original code does not explicitly reset.

4 The Verification Framework

4.1 Interval Temporal Logic

Interval temporal logic (ITL) [9] in KIV is based on algebras and intervals. Algebras define a semantic for the signature and intervals (executions) are finite or infinite sequences of states which evolve from program execution. A state maps variables to values in the algebra. In contrast to standard ITL, the logic explicitly includes the behavior of the program's environment in each step. Similar to "reactive sequences" [10], in an interval $I = [I(0), I'(0), I(1), I'(1), \ldots]$ the first transition from state $I(0)$ to the primed state $I'(0)$ is a program transition, whereas the next transition from state $I'(0)$ to $I(1)$ is a transition of a program's environment. In this manner program and environment transitions alternate. A variable V is evaluated over $I(0)$, whereas its primed resp. double primed version V' resp. V'' is evaluated over $I'(0)$ and $I(1)$ respectively. E.g., formula $V \neq V'$ denotes that variable V is changed in the first program transition, whereas $V' = V''$ states that V is not changed in the first environment transition. The last state of an interval is characterized by the atomic formula **last**.

The logic uses standard temporal operators to express future properties of an interval ($\Box, \Diamond, \bullet$, **until**). In rely-guarantee proofs, formulas $R(V', V'') \xrightarrow{+} G(V, V')$ are of particular interest, where G resp. R are guarantee resp. rely conditions and the "sustains" operator $\xrightarrow{+}$ ensures that the guarantee is sustained

by a program, as long as its environment has not previously violated the rely (cf. Section 5).

$$R(V', V'') \xrightarrow{+} G(V, V') :\leftrightarrow \neg\, (R(V', V'')\, \mathbf{until} \neg\, G(V, V'))$$

The programming language provides the common sequential constructs, a construct for weak-fair and one for non-fair interleaving. Note that arbitrary programs α and formulas can be mixed, since they both evaluate to true or false over an algebra and an interval I. In particular, α evaluates to true in I iff I is an execution of α interleaved with arbitrary environment steps.

4.2 Symbolic Execution and Induction

The verification framework is based on the sequent calculus. A sequent is an assertion of the form $\Gamma \vdash \Delta$, where Γ, Δ are lists of formulas. It states that the conjunction of all formulas in antecedent Γ implies the disjunction of all formulas in succedent Δ. Sequents are implicitly universally closed. A typical sequent (proof obligation) about concurrent programs has the form $\alpha, E, F \vdash \varphi$ where a program α executes the program steps in an environment constrained by temporal formula E. Predicate logic formula F describes the current state of an α-execution and φ denotes the temporal property of interest. A sequent of the aforementioned form is:

$$(M := M + 1;\ \beta),\ M = 1 \ \vdash\ M' = M'' \xrightarrow{+} M' > M \tag{2}$$

The executed program is the sequential composition $M := M+1;\ \beta$, environment behavior is unrestricted ($E = \mathit{true}$ omitted), the current state maps M to 1 and the succedent claims that the program increments M as long as its environment leaves M unchanged ($M' = M'' \xrightarrow{+} M' > M$).

Symbolic Execution Proving sequents that contain temporal assertions is done by symbolically stepping forward to the next states of an interval, calculating strongest post conditions for each program step, possibly weakened according to environment assumptions. Thus the calculus is rather similar to classic symbolic execution of sequential programs [11], once environment behavior is suitably restricted.

A step computes by applying unwinding rules to both programs and formulas. A program is unwound by calculating the effect of its first statement and discarding it; the sustains operator is unwound using the rule $R \xrightarrow{+} G \ \equiv\ G \wedge (R \rightarrow \bullet\, (R \xrightarrow{+} G))$. Applying it on the succedent of (2) yields $M' > M \wedge (M' = M'' \rightarrow \bullet\, (M' = M'' \xrightarrow{+} M' > M))$. That is, we must prove that the counter is incremented by the (first) program transition as a first subgoal ($M' > M$). If the following environment transition leaves M unchanged ($M' = M''$), then the sustains formula must further hold in the rest of the interval ($\bullet$). Thus, we get a second subgoal when proving (2):

$$\beta,\ M = 2 \ \vdash\ M' = M'' \xrightarrow{+} M' > M$$

Induction Well-founded induction is used to deal with loops. For infinite intervals a term for well-founded induction can often be derived from a known liveness property $\Diamond\ \varphi$ as the number of steps N until φ holds.

$$\Diamond\ \varphi\ \leftrightarrow\ \exists\ N.\ (N = N'' + 1)\ \textbf{until}\ \varphi.$$

This equivalence states that φ is eventually true iff there is a natural number N which can be decremented until φ becomes true. Note that N is a fresh variable and $N = N'' + 1$ is equivalent to $N'' = N - 1 \wedge N > 0$.

An induction term can be also extracted from a sustains formula.

$$R \xrightarrow{+} G\ \leftrightarrow\ \forall\ B.\ (\Diamond\ B) \rightarrow ((R \wedge \neg\ B) \xrightarrow{+} G)$$

Thus, the proof of a sustains formula on an infinite interval I can be carried out by induction over the length of an arbitrary finite I-prefix, which ends when the fresh boolean variable B is true for the first time. Further details on the underlying calculus can be found, e.g., in [12, 13].

5 The System Model and the Decomposition Theory

This section briefly describes the decomposition theory which we have applied to verify the case study. It contains several improvements over the theory used in [14, 15], which are independent from verifying the stack. Their description is not in the scope of this paper (cf. [7] for more details).

The Concurrent System Model The system model SPAWN$(n; \dots)$ recursively spawns $n + 1$ processes ($n : \mathbb{N}$) to execute in parallel. Each process executes finitely or infinitely often operations COP$(\mathit{In}; \mathit{LS}, S, \mathit{Out})$ on shared data structures. Variables In resp. Out are thereby used to insert resp. return values. Parameter $\mathit{LS} : \mathit{lstate}$ is the exclusive local state of the invoking process (with process identifier LS.id), whereas $S : \mathit{sstate}$ is the shared state.

In the stack case study, COP is instantiated with the non-deterministic choice between one of the operations that each legal process, having an identifier $\leq$ MAXID, can concurrently execute. Illegal processes just skip.

COP$(\mathit{In}; \mathit{LS}, S, \mathit{Out})$ {
  if LS.id $\leq$ MAXID **then** {
    $\mathit{Push}(\mathit{In}; \mathit{LS}, S) \vee \mathit{Pop}(;\ \mathit{LS}, S, \mathit{Out}) \vee \mathit{Scan}(;\ \mathit{LS}, S) \vee \mathit{Reset}(; \mathit{LS}, S)$}}

The shared state S consists of the shared variables Top, H, HPR, whereas the local state LS is the tuple of all local variables UNew, USucc, OTop, OSucc, Id, Hazard_{pc}, Scan, BefInc_{pc}, Lid, Lhp, PL, and RL.

Local Rely-Guarantee Reasoning To avoid reasoning about interleaved executions of SPAWN, we use a local version of rely-guarantee reasoning [16] that is embedded in the temporal logic framework. Different from the original approach [16], it does not enforce reasoning over the whole system state with $n + 1$ local states. Specifications instead consider two processes p resp. q with

local states LS resp. LSQ. Such a reduction to a few representative processes is often useful for the verification of concurrent data types.

The rely-guarantee embedding abstracts from interference from other processes using rely conditions R_{ext}. In return, each process guarantees a certain behavior towards its environment according to guarantee conditions G_{ext}. Both G_{ext} and R_{ext} are structured into three categories: step invariant guarantee and rely conditions G and R, state invariant conditions Inv and $Disj$ (to symmetrically encode disjointness between the two local states), plus, local idle state conditions $Idle$ which hold between COP-executions only. (The use of these structural predicates in the case study is shown in Section 6.) Thus, the central proof obligation for rely-guarantee reasoning is:

$$\begin{array}{l}\mathrm{COP}(In; LS, S, Out), Idle(LS), Inv(LS, S),\\ LS.\mathrm{id} \neq LSQ.\mathrm{id}, Inv(LSQ, S), Disj(LS, LSQ) \vdash R_{ext} \xrightarrow{+} G_{ext}\end{array} \tag{3}$$

According to G_{ext}, COP-steps maintain the guarantee conditions and the state invariants, plus, establish the idle state conditions.

$$\begin{array}{l}G_{ext}(LS, LSQ, S, LS', LSQ', S') :\leftrightarrow\\ \quad G(LS, LSQ, S, LS', S')\\ \wedge\ (\quad Inv(LS, S) \wedge Inv(LSQ, S) \wedge Disj(LS, LSQ)\\ \quad\ \rightarrow Inv(LS', S') \wedge Inv(LSQ', S') \wedge Disj(LS', LSQ')) \wedge (\mathbf{last} \rightarrow Idle(LS))\end{array}$$

According to R_{ext}, transitions of COP's environment do not modify LS and they maintain R and the state invariants.

$$\begin{array}{l}R_{ext}(LS', LSQ', S', LS'', LSQ'', S'') :\leftrightarrow\\ \quad LS' = LS'' \wedge R(LS', S', S'')\\ \wedge\ (\quad Inv(LS', S') \wedge Inv(LSQ', S') \wedge Disj(LS', LSQ')\\ \quad\ \rightarrow Inv(LS'', S'') \wedge Inv(LSQ'', S'') \wedge Disj(LS'', LSQ''))\end{array}$$

Theorem 1 (Local Rely-Guarantee Reasoning). *If (3) can be proved for some transitive rely predicate R, reflexive predicate G with $G(LS, LSQ, S, LS', S') \rightarrow R(LSQ, S, S')$, symmetric predicate $Disj$ and predicates $Idle$ and Inv, then each system step of* Spawn$(n; \dots)$ *is a guarantee step G which does not modify the local state of other processes, the invariant conditions Inv and $Disj$ hold for all processes at all times, and each process is $Idle$, just before it invokes* COP.

The Decomposition of Linearizability and Lock-Freedom Linearizability [2] and lock-freedom [1] are major, global correctness resp. progress properties of concurrent systems. We define local proof obligations for COP which imply linearizability and lock-freedom of Spawn. They are based on a local invariant property ISR that each process may always assume during its execution of $\mathrm{COP}(In; LS, S, Out)$, according to Theorem 1.

$$ISR :\leftrightarrow Inv(LS, S) \wedge Inv(LS', S') \wedge LS' = LS'' \wedge R(LS', S', S'')$$

Linearizability We prove linearizability by locating the linearization point (i.e., the step where a call appears to take effect) of each operation during its

execution. Conceptually, the linearization point of an execution of COP is determined in a refinement proof using an abstraction function $Abs \subseteq sstate \times astate$ (a partial function defined on shared states that satisfy Inv, which returns a corresponding abstract state). In the stack example, Abs maps a linked list representation of the stack to a finite algebraic list St of its data values.

$$\begin{aligned} &Abs(Top, H, [\,]) :\leftrightarrow Top = null \\ &Abs(Top, H, v + St) :\leftrightarrow \quad Top \neq null \wedge Top \in H \wedge H[Top].\mathrm{val} = v \\ &\qquad\qquad\qquad\qquad\qquad \wedge Abs(H[Top].\mathrm{nxt}, H, St) \end{aligned}$$

To prove linearizability, one has to show that each concrete operation from COP, non-atomically refines a corresponding abstract operation, which is defined in a further generic procedure AOP on an abstract state AS. In the case study, AOP is the non-deterministic choice between an abstract push or pop on St, or a sequence of mere skip steps for the scan and reset operations, which leave the stack unchanged. Hence, a sufficient process-local proof obligation for linearizability is:

$$\begin{aligned} &\mathrm{COP}(In; LS, S, Out), \Box\ (ISR \wedge Abs(S, AS) \wedge Abs(S', AS')), Idle(LS) \\ &\vdash \mathrm{AOP}(In; AS, Out) \end{aligned} \tag{4}$$

Theorem 2 (Decomposition of Linearizability). *In a setting in which the preconditions of Theorem 1 and proof obligation (4) hold for a suitable abstraction function Abs, the concurrent system* Spawn *is linearizable.*

Lock-Freedom Lock-free data structures ensure that even when single processes crash, neither deadlocks nor livelocks occur. In the stack example, single push and pop operations can be forced to always retry their loop if another process modifies the shared top pointer. If such an interference occurs, it is the interfering process which terminates its current execution and without interference, the current process terminates.

We capture this intuitive argument using an additional reflexive and transitive relation $U \subseteq sstate \times sstate$ to describe interference freedom ("unchanged"). To prove lock-freedom, one has to do two process-local termination proofs for each data structure operation. First, termination without U-interference and second, termination after violating U in a step. Thus, a sufficient process-local proof obligation for lock-freedom is (cf. [8, 15] for more details):

$$\begin{aligned} &\mathrm{COP}(In; LS, S, Out), \Box\ ISR, Idle(LS) \\ &\vdash \Box\ ((\Box\ U(S', S'')) \vee \neg\ U(S, S') \rightarrow \Diamond\ \mathbf{last}) \end{aligned} \tag{5}$$

Theorem 3 (Decomposition of Lock-Freedom). *In a setting in which the preconditions of Theorem 1 and proof obligation (5) hold for a reflexive and transitive relation U, the concurrent system* Spawn *is lock-free.*

6 Verifying the Stack with Hazard Pointers

This section shows central properties of hazard pointers and their specialization to formal verification conditions for the stack from Figure 2. To keep the

presentation readable, we only give some major conditions explicitly (all formal conditions are described in [7]). All conditions are expressed in terms of at most two processes. This is possible, since a retired location r can only be freed by the process, which has removed r from the stack and then retired it. Thus, when a process is in its hazardous code region, there is at most one other process which could free its critical pointer.

6.1 Central Properties of Hazard Pointers

The following central invariant property of hazard pointers ensures that heap access errors do not occur in hazardous code regions.

$$\boxed{HPR_{valid} \subseteq H} \tag{6}$$

According to (6), each validated hazard pointer is in the application's heap at all times, i.e., it is never freed (cf. (1)). This property correlates with GC where one may assume that a heap location r is not concurrently freed if it is just referenced in some operation. With hazard pointers, one can make the same assumption if r is covered by a *validated* hazard pointer.

Before a process p validates a location r, however, it can be concurrently freed by another process q and arbitrarily reused even if p has already set its hazard pointer to r. This happens when $HPR_p := r$ is executed after the location has been retired by q, and q has passed p's hazard pointer entry in its current traversal of HPR. Therefore, we omit any assertions about hazard pointers which are not validated yet. This differs from Parkinson et al. [5], who include such locations in their main correctness argument (cf. Section 7).

A difference between hazard pointers and GC is that while locations that are reachable from a root location can be concurrently freed if they are no longer covered by a validated hazard pointer, they would typically not be freed under GC, as long as their root is used.

The next central property of hazard pointers ensures that retired locations are in the application's heap, but not in the lock-free data structure.

$$RL \subseteq (H - LDS) \tag{7}$$

This has two major consequences. First, deallocation steps are safe, as they do not affect locations which are not in the application's heap. Second, succeeding validations (a location is in LDS at that time) imply that the validated location is currently not retired, hence not a deallocation candidate of any current scan.

Two further central properties of hazard pointers ensure that no ABA-problem occurs.

$$\textbf{if } r \in HPR_{valid} \textbf{ then } r \notin NEW \tag{8}$$

$$\textbf{if } \text{under GC: } H'(r) = H''(r) \textbf{ then if } r \in HPR_{valid} : \ H'(r) = H''(r) \tag{9}$$

(8) states that if a location r is covered by a validated hazard pointer, then it is not reused, i.e., it is not reinserted in the data structure which averts the ABA-problem. This property is also related to GC, where a heap location is not reused

as long as it is referenced in some operation. Hence, the environment assumption (9) holds: if the content of a heap location r is not concurrently changed in an environment with GC, then it is also unchanged when r is covered by a validated hazard pointer.

6.2 Verification Conditions for the Stack

Properties (6) - (9) are specialized to formal verification conditions which ensure memory-safety and ABA-avoidance for the stack. Properties in bold script are the corresponding verification conditions under GC, which we have simply reused.

Absence of Access Errors The stack-specific counterpart of generic property (6) ensures that the snapshot pointer is allocated and covered by a validated hazard pointer in the hazardous code region of pop.

$$Hazard_{pc} \wedge \boldsymbol{OTop \neq null \rightarrow OTop \in H} \wedge HPR(Id) = OTop \tag{10}$$

The stack-specific version of (7) implies that retired locations are allocated and disjoint from the stack, where a standard reachability predicate checks whether a location r is in the stack.

$$\forall\, r \in RL.\ r \neq null \wedge r \in H \wedge \neg\, reach(Top, r, H) \tag{11}$$

(10) and (11) ensure that heap access errors do not occur in pop and scan.

To sustain (10) at all times in every possible execution, the validated hazard pointer $OTop = HPR(Id)$ used in a pop operation of process p ($Hazard_{pc}$ holds, Id is the process identifier of p) must not be freed by any process q. The worst case is that q has retired $OTop$, just traverses HPR, but has not yet collected it ($OTop \in RLq - PLq$). Then q must not have passed the entry of p yet ($Lidq \leq Id$) and if it has reached p's entry, it must store $OTop$ in the local variable $Lhpq$ to ensure that it is collected. Invariant *ishazard* encodes this criterion precisely:

$$\begin{array}{l} ishazard(LS, LSQ) :\leftrightarrow \\ Hazard_{pc} \wedge OTop \in (RLq - PLq) \wedge Scanq \ \rightarrow \\ \quad \textbf{if } BefIncq_{pc} \textbf{ then } Lidq < Id \vee (Lidq = Id \wedge Lhpq = OTop) \textbf{ else } Lidq \leq Id \end{array}$$

Note that *ishazard* is independent from the underlying data structure, except for mentioning the concrete hazardous reference $OTop$. It can be easily adapted to ensure memory-safety for other lock-free data structures as well.

To sustain invariant (11) at all times, we must establish that retired lists are always duplicate free and pairwise disjoint. Otherwise, a retired list might contain a freed location after a deallocation step. Furthermore, three basic heap-disjointness properties are necessary: removed locations, which are subsequently retired, must be disjoint from the stack and they must not be concurrently retired, plus, concurrently removed locations must be disjoint.

To ensure that heap access faults do not occur in push either, we claim that new cells that have not been inserted yet, are always allocated and never concurrently retired, hence never freed.

ABA-prevention The stack-specific version of (8) ensures that the validated snapshot in pop is not reused, thus it is disjoint from other new cells.

$$Hazard_{pc} \wedge \neg\ \boldsymbol{USuccq} \rightarrow \boldsymbol{OTop} \neq \boldsymbol{UNewq} \tag{12}$$

The specialization of (9) yields the following rely condition which ensures that the snapshot's content is immutable in the hazardous code region of pop, to avoid an ABA-problem between the execution of lines O12 and O13.

$$Hazard_{pc}{}' \wedge \boldsymbol{OTop'} \neq \boldsymbol{null} \rightarrow \boldsymbol{H'}[\boldsymbol{OTop'}] = \boldsymbol{H''}[\boldsymbol{OTop'}] \tag{13}$$

An ABA-problem does not happen in push as well, since the content of a new cell remains unchanged.

$$\neg\ \boldsymbol{USucc'} \rightarrow \boldsymbol{H'}[\boldsymbol{UNew'}] = \boldsymbol{H''}[\boldsymbol{UNew'}] \tag{14}$$

To maintain rely (14) for the other process, when the current push process updates the new cell's next reference in line U8, new cells must be disjoint.

$$\neg\ \boldsymbol{USucc} \wedge \neg\ \boldsymbol{USuccq} \rightarrow \boldsymbol{UNew} \neq \boldsymbol{UNewq} \tag{15}$$

Verification conditions (10) and (11) are a main part of the structural predicate *Inv* from Section 5. Conditions *ishazard*, (12) and (15) are part of the symmetric predicate *Disj*, which is defined as:

$$Disj(LS, LSQ) :\leftrightarrow ishazard(LS, LSQ) \wedge ishazard(LSQ, LS) \wedge (12) \wedge \ \ldots$$

Rely conditions (13) and (14) are the major part of R; guarantee G is defined to maintain R for the other process and a simple step-invariant which ensures that COP-steps do not create memory leaks. Finally, the *Idle* predicate encodes the following local restrictions:

$$USucc \wedge OSucc \wedge \neg\ Hazard_{pc} \wedge \neg\ Scan \wedge \neg\ BefInc_{pc} \wedge Lid = 0$$

6.3 The Main Proofs

Sustainment of the Verification Conditions The main effort of the case study is to prove the rely-guarantee proof obligation (3) – sustainment of the verification conditions during steps of each operation. We proceed by case analysis over $\textsc{Op} \in \{Scan, Pop, Push, Reset\}$. The proof resembles a Hoare-style proof of a sequential program. We use $\xrightarrow{+}$ induction for loops and consecutively, symbolically execute each program statement in Op according to Section 4. Only some major arguments are outlined.

$\textsc{Op} \equiv Scan$: It is rather subtle to establish $ishazard(LSQ, LS)$ when the current process switches to the next hazard pointer entry in line S7. This step must not miss a validated hazard pointer $OTopq$ of the other process q if the current process p has retired, but not yet collected it ($OTopq \in RL - PL$). If the snapshot *Lhp* of the current *HPR* entry is not null, we know from previous symbolic

execution that it is in PL. If the current iteration examines q, *ishazard* before this step implies $Lhp = OTopq$, i.e., the validated hazard pointer has just been collected in the current iteration ($OTopq \in PL$), implying $ishazard(LSQ, LS)$.

In the deallocation step (line S10), *ishazard* ensures that the validated snapshot location of the other process is not freed (10). The proof is by contradiction: if the other process is in its hazardous code region and its snapshot pointer is in $RL - PL$, then *ishazard* before this step implies that the current process must not have finished its traversal. However, the current process is in its second scan loop already (technically, the contradiction is $\text{MAXID} + 1 = Lid \leq Idq \leq \text{MAXID}$).

$\text{OP} \equiv Pop$: In the succeeding hazard pointer validation step (lines O10 / O11), *ishazard* and (10) can be established, since the hazard pointer is in the data structure, hence allocated and not concurrently retired. Immediately after removal of the snapshot $OTop$ from the stack in line O13, we know from (11) that it can not be in the current process' retired list RL. Hence, we can establish (11) again in the retiring step (line O16), since both $OTop$ and RL are local.

$\text{OP} \equiv Push$: The allocation step (lines U3 / U4) resets the content of a new cell. However, it does not affect allocated locations and thus neither rely condition (13) nor (14) of the other process are violated. We additionally establish $UNew \notin RL$ in this step which allows to prove disjointness of retired locations from the data structure (11), when the new cell is added to the stack in line U9.

$Op \equiv Reset$: The reset of a hazard pointer entry is safe, since it happens outside of the hazardous code region in pop.

Preservation of Linearizability The proof of linearizability (4) distinguishes between the four possible concrete operations. In case of the hazard pointer operations scan and reset, each concrete step refines an abstract skip step. In particular, the deallocation step (lines S9 / S10) does not affect the stack, as retired locations are disjoint from the stack, according to (11).

The extended pop operation still has one linearization point in line O5 if the stack is empty, or else in line O13 if the CAS succeeds. Rely (13) ensures that the next reference of the snapshot cell and its value are immutable. Thus, the successful CAS corresponds to an abstract pop and returns the correct value. In case of a push operation, the linearization point is the successful CAS. Rely (14) ensures that the initial value of the new cell and its next reference are immutable. Hence, the successful CAS corresponds to an abstract push of the invoked value.

Preservation of Lock-Freedom According to (5), the proof of lock-freedom requires termination proofs for each data structure operation if environment behavior is restricted according to U and if a step violates U. We determine the unchanged relation as identity of the top-of-stack pointer. It is then relatively simple to show that push and pop terminate. Since the scan operation is wait-free, we can prove its termination without U. Termination of the first scan loop uses well-founded induction over the term $\text{MAXID} - Lid$ which decreases in every iteration. Similarly, termination of the second loop follows by induction over the number of retired locations.

7 Related Work and Comparison

Current automatic techniques do not prove linearizability or lock-freedom without implicitly assuming GC, which significantly simplifies the proofs. Thus they are not directly related to this work. We do not know of any other mechanized verification of a lock-free algorithm with hazard pointers. [17] describes a mechanized proof of a lock-free queue with modification counters [3], which focuses on linearizability. Neither an ABA-problem nor lock-freedom are discussed.

Manual Proofs Michael [4] gives a semantic verification condition which ensures safe memory reclamation for a lock-free algorithm with hazard pointers. This global condition requires the existence of a time in the past from which a hazardous location is safely covered by a hazard pointer. Michael verifies neither linearizability nor lock-freedom of the extended stack, but informally ensures safety by construction. Our verification of the stack formally resembles Michael's arguments, while avoiding both global reasoning and reasoning about the past. A key idea was to map Michael's temporal interval in which memory-safety and ABA-prevention are guaranteed, to a corresponding code interval ($Hazard_{pc}$).

There are two formal pen and paper proofs of a Treiber-like stack with hazard pointers. Parkinson et al. [5] apply concurrent separation logic (CSL) to verify a variant of the original stack, focusing on heap-modular reasoning and fractional permissions, which are used for simple properties such as (12) or (15). Their central correctness argument states that after a hazard pointer covers a location t, it can not be removed from the stack and then reinserted, which avoids the ABA-problem. Restricting this property to the case that t is covered by a *validated* hazard pointer better captures the essence of the reclamation scheme. While we use mainly simple formulas to ensure ABA-avoidance for validated hazard pointers, their proof requires rather complex auxiliary data structures.

Fu et al. [18] verify the stack in a program logic for history (HLRG). It provides temporal operators of the past only and evaluates state assertions in the last state of an execution. Their proof is based on rather complex global arguments about the temporal past of finite executions, while our verification conditions are just state/step invariants. Their implementation is not lock-free, since their *HPR*-traversal does not complete when a location is covered by a hazard pointer and the associated process fails. Michael's traversal, however, completes independent from environment behavior.

CSL and HLRG are based on separation logic and use abstract code annotations in their verification, while we use refinement, separating concrete from abstract code. They benefit from the implicit treatment of different heap locations by the separating conjunction operator, while we have to encode some disjointness properties explicitly. Their verification considers memory-safety and structural invariance of the stack only, but proves neither linearizability nor lock-freedom. They use process-global conditions and do not exploit symmetry.

8 Summary and Discussion

This work describes the first mechanized verification of a challenging lock-free stack. The proof intuitively applies central properties of the hazard pointers method and takes advantage of the relation between Michael's method and GC. It addresses the main safety and liveness aspects, avoiding process-global reasoning, complex history variables and reasoning about the past. Hence, it contributes an improved formal verification of the stack with hazard pointers.

Furthermore, we have applied our verification technique to the Michael-Scott queue with hazard pointers [4], where each process requires two hazard pointers. The central verification condition *ishazard* has been used analogously to ensure that the hazardous snapshot locations of the queue are not concurrently freed. The verification conditions from our previous proof under GC have been simply reused (cf. [8]). This indicates that the results of this work can be carried over to verify other lock-free algorithms in a similar way. A mechanized, schematic proof of correctness for an arbitrary underlying data structure, however, is left for future work.

As a further extension of our work, Maged Michael proposed that reading and writing hazard pointers non-atomically should be safe too, even though the scan algorithm may then read corrupted values. We confirmed this conjecture by replacing the atomic assignments with generic read and write procedures. These were specified to work correctly only if the environment does not concurrently modify the shared value. Just a few minor modifications of the proofs were necessary (cf. [8]).

Our current approach to verify linearizability suffices for algorithms that have an internal linearization point within the code of the executing process, even when its location depends on subsequent system behavior. This is possible, since future states of an interval can be easily analyzed in ITL (refer to [14] for more details). A generalization of the technique, using the results of [19], is part of current work.

Acknowledgments

We thank Jörg Pfähler for verifying the Michael-Scott queue with hazard pointers, resp. Alexander Knapp and Maged Michael for fruitful discussions.

References

1. Massalin, H., Pu, C.: A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Columbia University (1991)
2. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Trans. on Prog. Languages and Systems **12**(3) (1990) 463–492
3. Treiber, R.K.: System programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
4. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6) (2004) 491–504

5. Parkinson, M., Bornat, R., O'Hearn, P.: Modular verification of a non-blocking stack. SIGPLAN Not. **42**(1) (2007) 297–302
6. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In Bibel, W., Schmitt, P., eds.: Automated Deduction—A Basis for Applications. Volume II: Systems and Implementation Techniques. Kluwer Academic Publishers, Dordrecht (1998) 13 – 39
7. Tofan, B., Schellhorn, G., Reif, W.: Verifying a stack with hazard pointers in temporal logic. Technical Report 2011-08, Universität Augsburg (2011) `http://opus.bibliothek.uni-augsburg.de/volltexte/2011/1717/`.
8. KIV: Presentation of proofs for concurrent algorithms (2011) URL: `http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html`.
9. Moszkowski, B.: Executing Temporal Logic Programs. Cambr. Univ. Press (1986)
10. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2001)
11. Burstall, R.M.: Program proving as hand simulation with a little induction. Information processing 74 (1974) 309–312
12. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Communications **23**((2,3)) (2010) 285–307
13. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Proc. of TIME, to appear. IEEE, CPS (2011)
14. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. Formal Aspects of Computing (FAC) (2009) appeared online first, http://www.springerlink.com/content/7507m59834066h04/.
15. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal logic verification of lock-freedom. In: In Proc. of MPC 2010. Springer LNCS 6120 (2010) 377–396
16. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP'83, North-Holland (1983) 321–332
17. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE 2004. Volume 3235 of LNCS. (2004) 97–114
18. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about optimistic concurrency using a program logic for history. In: CONCUR. (2010) 388–402
19. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisabilty with potential linearisation points. In: Proc. Formal Methods (2011), to appear