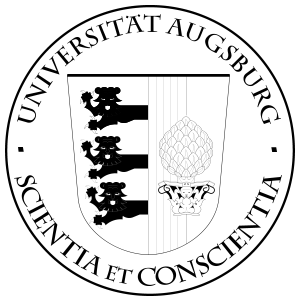
UNIVERSITÄT AUGSBURG



Verifying a Stack with Hazard Pointers in Temporal Logic

B. Tofan, G. Schellhorn, W. Reif

Report 2011-08 March 2011

INSTITUT FÜR INFORMATIK
D-86135 AUGSBURG

Verifying a Stack with Hazard Pointers in Temporal Logic

Bogdan Tofan Gerhard Schellhorn 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, but the verification of a simple lock-free stack with hazard pointers is challenging.**

This work contributes to the formal verification of lock-free algorithms. Using the temporal logic framework of the interactive prover KIV, we verify correctness and liveness of a lock-free stack with hazard pointers. The proof exploits the algorithm's symmetry and requires neither additional history variables nor temporal past operators to describe inter-process interference. Moreover, the verification shows a relation between hazard pointers and garbage collection, which makes it possible to reuse the verification conditions from the simpler proof under garbage collection.

I. Introduction

Lock-free (non-blocking) implementations of concurrent data structures avoid major problems associated with blocking, such as convoying, deadlocks or priority inversion. Their main liveness property *lock-freedom* [1] guarantees global progress even in the presence of arbitrary delays or failure of single processes. 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 lock-based implementations significantly in the presence of contention or multiprogramming [3]. These properties are even more important as multicore 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 automatic 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 can 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 [4]. It becomes manifest in subtle errors such as data structure corruption or wrong return values (see Section III).

Several reclamation schemes that compensate the absence of GC exist (see [5] for a performance analysis). Unfortunately, not much work has been done on their mechanized verification. It is nevertheless worthwhile to formally analyse their subtleties, as these schemes affect many critical applications.

Contributions Hazard pointers [6] enable safe memory reclamation by extending concurrent algorithms with local non-blocking garbage collection. Our work contributes an analysis of the central properties of hazard pointers and applies the results to verify an extended lock-free stack. Proving safe memory reclamation and ABA-avoidance for a stack with hazard pointers has been setup a challenge for program verification [7]. To the best of our knowledge, our proof is the first mechanized proof of such an algorithm.

The verification allows for a comparison of our approach (RGITL) –which applies interval temporal logic and symmetric rely-guarantee reasoning– with two other current approaches (CSL, HLRG) that have verified a similar case study (cf. Section VII). On the one hand, the proofs show that RGITL is able to mechanically analyze both safety and liveness requiring neither reasoning about the past nor additional history variables to capture temporal properties. On the other hand, both CSL and HLRG benefit from separation logic's more succinct modular reasoning about the heap.

Our proofs address all aspects of the reclamation scheme: memory-safety and ABA-prevention as well as preservation of linearizability and lock-freedom of the stack. The proof reveals that the central correctness arguments under GC [8], [9] can be reused with hazard pointers. Furthermore, all verification conditions are expressed in terms of at most two contending processes –a symmetry reduction which has not been exploited to this extent before. Thus the proofs can be kept moderately complex.

We have improved our previous decomposition theory (cf. [9], [10]) to allow for this symmetry reduction, but the focus in this work is rather on its application. Due to space limitations and to keep the presentation readable, we do not detail every formal aspect. However, the KIV system and a complete web presentation of all proofs from both the verification of the decomposition theory and its application to the stack (under GC and with hazard pointers) is available online [8].

The remainder of this paper is organized as follows: Section II gives an introduction to hazard pointers. Section III specifies the main case study of this paper, the extended

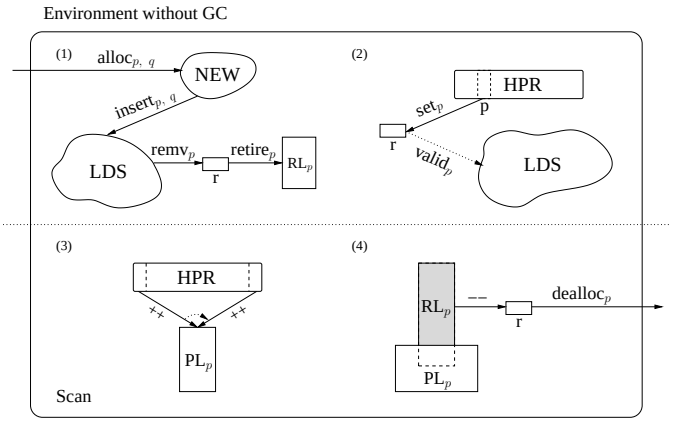


Fig. 1: Michael's Hazard Pointers Method.

stack algorithm. Section IV briefly introduces the verification framework which forms the base for the applied decomposition theory. The system model, the embedding of rely-guarantee reasoning and the decomposition of linearizability and lock-freedom are described in Section V. Section VI shows four central properties of the hazard pointers method and their specialization to formal verification conditions in the case study. Section VII presents related work and a comparison. Finally, Section VIII concludes with a summary of the main results and a short overview of our current and future work.

II. The Hazard Pointers Method

Figure 1 illustrates hazard pointers, which allow removed objects to be safely freed to the environment's memory management system without depending on special support from the underlying architecture.

(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 that it removes from LDS in a local pool of retired locations RL_p. These locations are potential candidates for deallocation. However, the contending use of these retired locations must be considered first.

(2) each process is associated with a fixed (small) number of single-writer multi-reader shared pointers, so called hazard pointers. All hazard pointers of all processes are contained in a global 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.

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 when traversing HPR ($r \in RL_p - PL_p$), 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.

III. A Lock-Free Stack with Hazard Pointers

A. Lock-Free Stack

Instead of using locks, lock-free algorithms typically utilize an optimistic try-retry scheme and atomic synchronization primitives such as the widely supported single-word CAS (Compare-And-Swap) instruction. A lock-free operation speculatively applies several manipulations on a local copy (called snapshot) of the shared data and attempts to synchronize the global data with the updated copy using CAS.

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

A CAS instruction compares a global word-sized parameter G with an older local snapshot version *Old*. If these values are equal, then G is updated to a new value *New* and true is returned to indicate the successful update.

Throughout this work, we use formal KIV-specifications to describe programs. In the specification of CAS, the semicolon separates input from reference parameters; the additional parameter *Succ* is used to return a (boolean) value, a comma separates parallel assignments and **if*** denotes that the comparison requires no (extra) step.

Figure 2 illustrates the lock-free stack which provides concurrent push and pop operations (the shaded code in pop can be ignored for now). The algorithm is a prime example of a lock-free data structure, taken from Michael [6] and attributed to Treiber [4]. The shared stack is a singly linked list of cells in the application's memory heap H.

The heap is simply a partial function from references $r : ref$ (with *null* $\in ref$) to cells, which are pairs of values and locations having .val and .nxt selector functions. The heap is associated with several standard operations: $r \in H$ tests whether r is in H's domain (allocated), heap lookup is denoted as $H[r]$, deallocation as $H - r$, heap update as $H[r, c]$ and allocation as $H[r, ?]$, where ? denotes an arbitrary cell content.

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

```
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, OHazard_pc, OTop, OSucc, RL, Top, H, HPR, Out) {
O2   let ONxt = ?, Lo = empty in {
O3     OSucc := false;
O4     while ¬ OSucc do {
O5       OTop := Top, OHazard_pc := false;
O6       if OTop = null then {
O7         OSucc := true
O8       } else {
O9         HPR(Id) := OTop;
O10        if* OTop = Top then {
O11          OHazard_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, OHazard_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: Lock-Free Data-Stack with Hazard Pointers.

global 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.

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 its local next reference $ONxt$ 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.

B. 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 global hazard pointer record $HPR : \mathbb{N} \to ref$ and the identifier $Id : \mathbb{N}$ of the current process. In line O10, before any hazardous 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 $OHazard_{pc}$ marks the hazardous code region in which the validated hazard pointer equals (covers) the snapshot $OTop$ (this flag 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 references 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 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 (initially 0) and adding it to a local pointer list PL ($BefInc_{pc}$ is a further 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 algebraically specified 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 $RL - PL$. Furthermore, we allow 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 simple 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.

IV. The Verification Framework

A. Interval Temporal Logic

Interval temporal logic (ITL) [11] in KIV [12] 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: in an interval $I = [I(0), I'(0), I(1), I'(1), \ldots]$ the first program transition leads from the initial state $I(0)$ to the primed state $I'(0)$ and the next transition (from state $I'(0)$ to $I(1)$) is a transition

of the program's environment. In this manner program and environment transitions alternate (similar to [13]).

We write V, V', V'' to denote variable V in states $I(0)$, $I'(0)$ and $I(1)$ respectively. Hence transitions of the form (V, V') refer to program transitions, whereas transitions (V', V'') denote environment transitions. 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 ($\Box$, $\Diamond$, $\bullet$, $\mathbf{until}, \dots$), but it does not include temporal past operators. In rely-guarantee proofs, formulas $R(V', V'') \xrightarrow{+} G(V, V')$ are of particular interest, where G resp. R is the guarantee resp. rely condition of a process and the "sustains" operator $\xrightarrow{+}$ ensures that a process sustains its guarantee in each step, as long as the environment has not previously violated its rely (cf. Section V).

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

where $\varphi \,\mathbf{unless}\, \psi$ abbreviates $(\Box\, \varphi) \vee (\varphi \,\mathbf{until}\, \psi)$.

The programming language provides the common sequential constructs ($:=$, $;$, **if**, $\dots$), a construct for weak-fair ($\|$) and one for non-fair ($\|_{\text{nf}}$) interleaving. Note that arbitrary programs α and formulas can be mixed, since they both evaluate to true or false over an algebra $\mathcal{A}$ and an interval I. In particular, α evaluates to true in I iff I is an execution of α (interleaved with arbitrary environment steps).

B. Symbolic Execution and Induction

The 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 claims 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:

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

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 [14], once environment behavior is suitably restricted.

Executing one step is done in two implicit phases which concern programs as well as formulas. In the first phase, information about the first program and environment transition is separated from information about the rest of an interval by applying unwinding rules. A program is unwound by calculating the effect of its first statement and discarding it; the sustains operator is unwound as follows:

$$\psi \xrightarrow{+} \varphi \ \equiv\ \varphi \wedge (\psi \rightarrow \bullet\, (\psi \xrightarrow{+} \varphi))$$

Applying this rule on the succedent of (2) yields

$$M' > M \wedge (M'' = M' \rightarrow \bullet\, (M'' = M' \xrightarrow{+} M' > M))$$

In other words, we have to show that the counter is incremented in the (first) program transition and $M' > M$ has to be sustained in the rest of the interval, only if the (first) environment transition leaves M unchanged.

The second phase of a symbolic execution step "moves" to the next state of an interval. In (2), the rest of the interval must satisfy:

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

Note that the counter has value 2 in the next state and the remaining program is β.

Induction Well-founded induction is used to deal with loops. A suitable induction term can be frequently derived from a known liveness property φ as the number of steps until φ holds. Thus the proof of a sustains formula on an infinite interval I can be carried out by inducing over the length of an arbitrary finite I-prefix. Further details on induction and the underlying temporal logic calculus can be found in [15], [16].

V. The System Model, Symmetric Rely-Guarantee Reasoning and the Decomposition Theorems

A. The Concurrent System Model

We define a generic interface procedure COP which models arbitrary operations (blocking or non-blocking) that a process can invoke on concurrent data structures.

$$\text{COP}(\mathit{In}; \mathit{LS}, \mathit{GS}, \mathit{Out})$$

Parameters In resp. Out are 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 $\mathit{GS} : \mathit{gstate}$ corresponds to the global state.

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

$$\begin{array}{l}\text{COP}(\mathit{In}; \mathit{LS}, \mathit{GS}, \mathit{Out})\ \{\\ \quad \textbf{if}\ \mathit{LS}.\text{id} \leq \text{MAXID}\ \textbf{then}\ \{\\ \qquad \mathit{Push}(\mathit{In}; \mathit{LS}, \mathit{GS}) \vee \mathit{Pop}(;\ \mathit{LS}, \mathit{GS}, \mathit{Out})\\ \qquad \vee\ \mathit{Scan}(;\ \mathit{LS}, \mathit{GS}) \vee \mathit{Reset}(; \mathit{LS}, \mathit{GS})\}\}\end{array}$$

The global state GS consists of the shared variables Top, H, HPR for the top-of-stack pointer, the application's heap and the hazard pointer record, whereas the local state LS is the tuple of all local variables Id, UNew, USucc, $\mathit{OHazard}_{pc}$, OTop, OSucc, Scan, BefInc_{pc}, Lid, Lhp, PL, RL.

The generic procedure SEQ defines the possible sequential behaviors of each process. A process can either instantly terminate or execute, finitely or infinitely often, the following steps (denoted by the star operator *****): it executes steps that are unrelated to COP (abstracted by *skip*), or it runs one of the data structure operations.

$$\begin{array}{l}\textsc{Seq}(p; In, LS, GS, Out)\ \{\\ \ \{skip \vee \textsc{Cop}(In; LS, GS, Out)\}^*\}\end{array}$$

The generic system model SPAWN recursively spawns $n+1$ processes ($n : \mathbb{N}$) to execute in parallel.

$$\begin{array}{l}\textsc{Spawn}(n; Inf, LSf, GS, Outf)\ \{\\ \ \textbf{if*}\ n = 0\ \textbf{then}\\ \quad \textsc{Seq}(0; Inf(0), LSf(0), GS, Outf(0))\\ \ \textbf{else}\\ \quad \textsc{Seq}(n; Inf(n), LSf(n), GS, Outf(n))\\ \quad \|\ \textsc{Spawn}(n-1; Inf, LSf, GS, Outf)\}\end{array}$$

Parameters $Inf, LSf, Outf$ are global functions from process identifiers to the corresponding process-local component, e.g., $LSf : \mathbb{N} \to lstate$ maps each process to its local state.

The interleaving of process steps in SPAWN is weakly fair. However, lock-freedom must also ensure global progress under an unfair scheduler which can discard single processes for good. To relax the assumption of weak-fair interleaving for the decomposition of lock-freedom, we have defined a further standard concurrent system (similar to [17]), in which each process executes non-blocking COP calls in an infinite loop and the interleaving is not fair (cf. [8]). In both systems, the decomposition yields the same local proof obligations for rely-guarantee reasoning and lock-freedom (cf. Sections V-B, V-C), i.e., these proof obligations also ensure lock-freedom under an unfair scheduler in the standard system.

B. Symmetric Rely-Guarantee Reasoning

To reduce the proof effort that arises from interleaving process steps in the overall concurrent system SPAWN, we have embedded a symmetric version of rely-guarantee reasoning [18], [19] within the temporal logic framework. The symmetry reduction avoids reasoning over the whole program state $LSf \times GS$, allowing specifications to consider at most two processes p resp. q with local states LS resp. LSQ.

The rely-guarantee embedding avoids reasoning about interleaved executions, by abstracting 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}. Our central rely-guarantee proof obligation claims that in each execution of COP, each program transition sustains G_{ext} if the preceding environment transitions have preserved R_{ext}.

$$\textsc{Cop}(In; LS, GS, Out)\ \vdash R_{ext} \xrightarrow{+} G_{ext} \tag{3}$$

We introduce further subpredicates to structure G_{ext} and R_{ext} into three categories: step invariant guarantee and rely conditions G and R, state invariant conditions Inv and $Disj$ (to encode disjointness between the two local states), plus, local pre- postconditions $Idle$ which hold between COP-executions only. A more detailed description of the structural predicates is given in the following; their use in the stack case study is shown in Section VI in detail.

According to G_{ext}, program steps in COP executions maintain the guarantee conditions and the state invariants, plus, they establish the pre- postconditions.

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

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

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

The first parameter of the transitive rely relation $R \subseteq lstate \times gstate \times gstate$ corresponds to a process' local state before an environment transition. The second resp. third parameter of a rely is the global state before resp. after an environment transition. In the case study, R ensures for instance that the content of a new cell in push is not concurrently changed, as long as it is not part of the stack.

$$\neg\ USucc' \rightarrow H''[UNew'] = H'[UNew']$$

The first three parameters of the reflexive guarantee condition $G \subseteq lstate \times lstate \times gstate \times lstate \times gstate$ denote the local states of the two processes and the global state before a program step; the last two parameters stand for the current process' local state and the global state after this step. The local state of the other process is used to express a central correlation between guarantee and rely conditions: guarantee steps are rely steps from the point of view of the other process.

$$\begin{array}{l}\quad G(LS_0, LSQ_0, GS_0, LS_1, GS_1)\\ \rightarrow R(LSQ_0, GS_0, GS_1)\end{array} \tag{4}$$

The state predicates are entirely decoupled from R and G to avoid unnecessarily strong rely resp. guarantee conditions. Every state in a COP-execution (including intermediate states) must satisfy the invariant predicate $Inv \subseteq lstate \times gstate$ and the symmetric disjointness predicate between the two local states $Disj \subseteq lstate \times lstate$. The idle predicate $Idle \subseteq lstate$ encodes local state pre- postconditions of finite executions of COP. In the case study, idle states satisfy the following local restrictions:

$$\begin{array}{rl}Idle(LS) :\leftrightarrow & USucc \wedge OSucc \wedge \neg\ OHazard_{pc}\\ & \wedge \neg\ Scan \wedge \neg\ BefInc_{pc} \wedge Lid = 0\end{array}$$

Together, the full version of (3), which takes into account the structural predicates, is:

$$\begin{array}{l}\quad \textsc{Cop}(In; LS, GS, Out),\ Idle(LS), Inv(LS, GS),\\ \quad LS.\text{id} \neq LSQ.\text{id},\ Inv(LSQ, GS),\ Disj(LS, LSQ)\\ \vdash R_{ext} \xrightarrow{+} G_{ext}\end{array} \tag{5}$$

Local proof obligations (4) and (5) imply several global properties of interleaved executions of SPAWN (cf. [8] for details).

Theorem 1 (Symmetric Rely-Guarantee Decomposition)**.** *If formulas (4) and (5) hold for two arbitrary disjoint local*

states LS, LSQ, the global state GS and some transitive rely predicate R, reflexive predicate G, symmetric predicate $Disj$ and predicates $Idle$ and Inv, then:

$$\textsc{Spawn}(n;\dots),\ \Box\, R_{\textsc{Spawn}},\ Init \vdash \Box\, \varphi$$

where $\varphi :\leftrightarrow \varphi_G \wedge \varphi_{Inv} \wedge \varphi_{Disj} \wedge \varphi_{Idle}$. According to φ_G, each system step is a guarantee step G which does not modify the local state of other processes.

$$\begin{array}{l}\varphi_G :\leftrightarrow \\ \exists\, p \le n.\ \forall\, q \neq p. \quad G(LSf(p), LSf(q), GS, LSf'(p), GS') \\ \qquad\qquad \wedge\ LSf'(q) = LSf(q)\end{array}$$

The invariant conditions hold for all processes at all times

$$\begin{array}{l}\varphi_{Inv} :\leftrightarrow \forall\, p.\ Inv(LSf(p), GS) \wedge Inv(LSf'(p), GS') \\ \varphi_{Disj} :\leftrightarrow \\ \forall\, p \neq q.\ Disj(LSf(p), LSf(q)) \wedge Disj(LSf'(p), LSf'(q))\end{array}$$

and all processes are in their idle states just before they invoke COP*, according to φ_{Idle}.*

The overall system starts in an initial state satisfying $Init$, which must imply Inv, $Disj$ and $Idle$ for all processes. The *system*'s environment behavior is restricted by a further rely condition $R_{\textsc{Spawn}}$ which is the identity relation over all reference parameters in SPAWN. A weaker system rely could be defined to account for an external garbage collector which removes unused locations from the global heap, but this is not required here.

C. The Decomposition of Linearizability and Lock-Freedom

Linearizability Basically, we prove linearizability –the major correctness property of lock-free algorithms– by locating the linearization point of each operation during its execution. Our current approach suffices to verify linearizability of algorithms that have an internal linearization point (within the code of the executing process), even when its location depends on system behavior. This is possible, since future states of an interval can be easily analyzed in ITL (refer to [9] for details).

Conceptually, the linearization point of an execution of COP is determined in a refinement proof using an abstraction function $Abs \subseteq gstate \times astate$ (a partial function defined on global 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.

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

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. In the case study, AOP is the non-deterministic choice between an abstract push or pop, or a sequence of mere skip steps for the scan and reset operations, which leave the stack unchanged.

$$\begin{array}{l}\mathrm{AOP}(In; St, Out)\ \{ \\ \quad \mathrm{APush}(In; St) \vee \mathrm{APop}(;\ St, Out) \vee skip^*\}\end{array}$$

$$\begin{array}{l}\mathrm{APush}(In; St)\ \{ \\ \ skip^*;\ St := push(In, St);\ skip^*\}\end{array}$$

$$\begin{array}{l}\mathrm{APop}(; St, Out)\ \{ \\ \ \textbf{let}\ Lo = \mathit{empty}\ \textbf{in}\ \{ \\ \quad skip^*; \\ \quad \textbf{if*}\ St \neq [\,]\ \textbf{then}\ \{Lo := top(St), St := pop(St)\}; \\ \quad skip^*; Out := Lo\}\}\end{array}$$

Fig. 3: Formal definition of the abstract stack operations.

Figure 3 shows the abstract stack operations APush and APop. They use atomic operations $push$ resp. pop to add resp. remove an element from St at concrete linearization points and additional skip steps at non-linearization points.

Refinement (i.e., trace inclusion) between COP and AOP is simply expressed as COP $\vdash$ AOP in the framework. Hence the process-local proof obligation for linearizability is:

$$\begin{array}{l}\mathrm{COP}(In; LS, GS, Out), \\ \Box\ (\quad LS'' = LS' \wedge R(LS', GS', GS'') \\ \qquad \wedge\ Inv(LS, GS) \wedge Inv(LS', GS') \\ \qquad \wedge\ Abs(GS, AS) \wedge Abs(GS', AS')),\ Idle(LS) \\ \vdash \mathrm{AOP}(In; AS, Out)\end{array} \tag{6}$$

(6) is based on the established conditions φ_G, φ_{Inv} and φ_{Idle} from Theorem 1. In particular, since φ_G implies that system steps never change other local states, the executing process may assume that its local state is not concurrently changed. Moreover, since each system step satisfies its guarantee, it is a rely step for other processes according to (4) and we may assume R for environment steps in (6).

Theorem 2 (Decomposition of Linearizability)**.** *In a setting in which the preconditions of Theorem 1 and proof obligation (6) hold for a suitable abstraction function Abs, the concurrent system* SPAWN *is linearizable [2].*

Lock-Freedom Lock-free data structures ensure that even when single processes crash, no deadlocks occur. In the stack example, single push and pop operations can be forced to always retry their loop if another process modifies the global 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 $U \subseteq gstate \times gstate$ 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:

$$\begin{array}{l}\Box\ (\Box\ U(GS', GS'') \rightarrow \Diamond\ \mathbf{last}) \\ \Box\ (\neg\ U(GS, GS') \rightarrow \Diamond\ \mathbf{last})\end{array}$$

Together, the process-local proof obligation for lock-freedom, (again) based on properties φ_G, φ_{Inv} and φ_{Idle} from Theorem 1, is as follows (cf. [8], [10] for details).

$$\begin{array}{l}\mathrm{COP}(In; LS, GS, Out), \\ \Box\ (\quad LS'' = LS' \wedge R(LS', GS', GS'') \\ \qquad \wedge\ Inv(LS, GS) \wedge Inv(LS', GS')),\ Idle(LS) \\ \vdash \Box\ ((\Box\ U(GS', GS'')) \vee \neg\ U(GS, GS') \rightarrow \Diamond\ \mathbf{last})\end{array} \tag{7}$$

Theorem 3 (Decomposition of Lock-Freedom)**.** *In a setting in which the preconditions of Theorem 1 and proof obligation (7) hold for a reflexive and transitive relation U, the concurrent system* SPAWN *is lock-free.*

VI. VERIFYING THE STACK WITH HAZARD POINTERS

The introduced reduction theory can be applied to verify the stack, considering two representative processes only. 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.

A. Central Properties of Hazard Pointers

Two central invariant properties of the hazard pointers method ensure that heap access errors do not occur in hazardous code regions and in deallocation steps respectively.

$$\boxed{HPR_{valid} \subseteq H} \tag{8}$$

$$RL \subseteq (H - LDS) \tag{9}$$

According to (8), each validated hazard pointer is in the application's heap at all times, i.e., it is never freed (cf. (1)). This central property correlates with GC where one may assume that a heap location r is not concurrently freed if it is just referenced by a pointer 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.

Property (9) ensures that retired locations are in the application's heap, but not in the lock-free data structure. 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.

$$\textbf{if } r \in HPR_{valid} \textbf{ then } r \notin NEW \tag{10}$$

$$\begin{array}{l}\textbf{if } \text{under GC: } H''(r) = H'(r) \textbf{ then} \\ \quad \textbf{if } r \in HPR_{valid} : \ H''(r) = H'(r)\end{array} \tag{11}$$

(10) 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 (11) holds: if the contents of a heap location r are not concurrently changed in an environment with GC, then they also remain unchanged when r is covered by a validated hazard pointer.

Pred	Name	Property
Inv	φ_t	$OHazard_{pc} \wedge \boldsymbol{OTop \neq null} \rightarrow \boldsymbol{OTop \in H} \wedge HPR(Id) = OTop$
	φ_{rl}	$\forall\, r \in RL.\ r \neq \textit{null} \wedge r \in H \wedge \neg\, reach(Top, r, H)$
	φ_{nod}	$\neg\, dups(RL)$
	φ_{rm}	$OHazard_{pc} \wedge OSucc \rightarrow \neg\, reach(Top, OTop, H)$
	φ_n	$\boldsymbol{\neg\, USucc \rightarrow UNew \neq null \wedge UNew \in H \wedge \neg\, reach(Top, UNew, H)}$
	φ_{st}	$\boldsymbol{\exists\, St.\ Abs(Top, H, St)}$
	φ_{ill}	$Id > \text{MAXID} \rightarrow RL = [] \wedge Idle(LS)$
Disj	δ_{ish}	$ishazard(LS, LSQ)$
	δ_{rl}	$disj(RL, RLq)$
	δ_{trl}	$OHazard_{pc} \wedge OSucc \rightarrow OTop \notin RLq$
	δ_{rm}	$OHazard_{pc} \wedge OSucc \wedge OHazardq_{pc} \wedge OSuccq \rightarrow OTop \neq OTopq$
	δ_{nrl}	$\neg\, USucc \rightarrow UNew \notin RLq$
	δ_{tn}	$OHazard_{pc} \wedge \boldsymbol{\neg\, USuccq \rightarrow OTop \neq UNewq}$
	δ_{nn}	$\boldsymbol{\neg\, USucc \wedge \neg\, USuccq \rightarrow UNew \neq UNewq}$
R	ρ_t	${OHazard_{pc}}' \wedge \boldsymbol{OTop' \neq null} \rightarrow \boldsymbol{H''[OTop'] = H'[OTop']}$
	ρ_n	$\boldsymbol{\neg\, USucc' \rightarrow H''[UNew'] = H'[UNew']}$
	ρ_{hp}	$HPR''(Id') = HPR'(Id')$
G	γ_{nol}	$pvnoleak(LS, GS, LS', GS')$
	γ_R	$\boldsymbol{R}$

TABLE I: Formal verification conditions for the stack with hazard pointers and under GC (bold script).

B. Memory-Safety and ABA-Prevention of the Extended Stack

Properties (8) - (11) are specialized to verification conditions which ensure memory-safety and ABA-avoidance for the extended stack from Figure 2. To improve readability, we have summarized all conditions in Table I, where column **Pred** contains the (structural) predicate which subsumes the properties from column **Name**; the properties in bold script are the verification conditions of the stack under GC, which we have simply reused, due to the aforementioned relation between hazard pointers and GC.

Absence of Access Errors The stack-specific counterpart of generic property (8) ensures that the snapshot pointer is allocated (and covered by a validated hazard pointer) in the hazardous code region of pop (φ_t). The stack-specific version of (9) implies that retired locations are allocated and disjoint from the stack (φ_{rl}), where a standard reachability predicate checks whether a location r is in the stack $reach(Top, r, H)$. Together, verification conditions φ_t and φ_{rl} ensure that no heap access errors occur in pop and scan.

To sustain φ_t at all times in every possible execution, the validated hazard pointer $OTop = HPR(Id)$ used in a pop operation of process p ($OHazard_{pc}$ 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:

$ishazard(LS, LSQ) :\leftrightarrow$
if $OHazard_{pc} \wedge OTop \in (RLq - PLq) \wedge Scanq$ **then**
  if $BefIncq_{pc}$ **then** $Lidq < Id \vee (Lidq = Id \wedge Lhpq = OTop)$
  else $Lidq \leq Id$

To sustain invariant φ_{rl} at all times, we must establish that retired lists are always duplicate free and pairwise disjoint ($\varphi_{nod},\ \delta_{rl}$). 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 ($\varphi_{rm}, \delta_{trl}, \delta_{rm}$).

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 (φ_n) and never (concurrently) retired (δ_{nrl}), hence never freed.

Absence of Leaks The hazard pointers method avoids memory leaks, i.e., all heap locations are either in the lock-free data structure or owned by a process. In terms of the stack, this heap-global property is defined as

$noleak(LSf, GS) :\leftrightarrow$
$\forall\ r \in H.\ reach(Top, r, H) \vee \exists\ p.\ owns(r, LSf(p))$

where a process owns its new, removed and retired locations.

$owns(r, LS)$: $\leftrightarrow$
  $(\neg\ USucc \wedge UNew = r)$
$\vee\ (OHazard_{pc} \wedge OSucc \wedge OTop = r) \vee r \in RL$

We decompose the absence of leaks to a process-local guarantee condition $pvnoleak$, which guarantees that each process step preserves ownership of a reference r.

$pvnoleak(LS, GS, LS', GS') :\leftrightarrow$
$\forall\ r.\ r \notin H \vee reach(Top, r, H) \vee owns(r, LS)$
  $\rightarrow r \notin H' \vee reach(Top', r, H') \vee owns(r, LS')$

ABA-Prevention The stack-specific version of property (10) ensures that the validated snapshot pointer in pop is not reused, thus it is disjoint from (other) new cells (δ_{tn}). The specialization of (11) yields a rely condition which ensures that the snapshot's contents are immutable in the hazardous code region of pop (ρ_t). Hence, an ABA problem does not occur between the execution of lines O12 and O13. An ABA problem is avoided in push as well, since the contents of a new cell remain unchanged according to rely condition ρ_n. To maintain this rely for the other process, when the current push process updates the new cell's next reference (line U8), new cells must be disjoint (δ_{nn}).

Finally, two simple verification conditions ensure that a process' hazard pointer entry is not concurrently modified (ρ_{hp}) and that illegal processes are irrelevant (φ_{ill}). We note that all structural predicates from Table I, except for $Disj$, are defined as the conjunction of their subproperties. To ensure symmetry of the disjointness predicate, we use the following stronger definition.

$Disj(LS, LSQ) :\leftrightarrow disj(LS, LSQ) \wedge disj(LSQ, LS)$
$disj(LS, LSQ) :\leftrightarrow \delta_{ish} \wedge \ \ldots\ \wedge \delta_{nn}$

The full guarantee G of each process includes all rely conditions R, to ensure that guarantee steps are rely steps for other processes (cf. (4)).

The Main Proof The main effort of the case study is to prove the rely-guarantee proof obligation (5) – sustainment of the verification conditions during steps of each operation. We proceed by case analysis over $\text{OP} \in \{Scan, Pop, Push, Reset\}$. 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 IV. Only major arguments are outlined.

Lemma 1 (OP $\xrightarrow{+}$)**.**

$Op, LS.id \leq MAXID, Idle(LS), Inv(LS, GS),$
$LS.id \neq LSQ.id,\ Inv(LSQ, GS),\ Disj(LS, LSQ)$
$\vdash R_{ext} \xrightarrow{+} G_{ext}$

Proof: OP $\equiv Scan$: It is rather subtle to establish the symmetric version of $ishazard$ ($ishazard(LSQ, LS)$) when the current process switches to the next hazard pointer entry (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, the symmetric version of $ishazard$ before this step implies that $Lhp = OTopq$, i.e., the validated hazard pointer has just been collected in the current iteration ($OTopq \in PL$).

In the deallocation step (line S10), the symmetric version of $ishazard$ ensures that the validated snapshot location of the other process is not freed (φ_t). The proof is by contradiction: if the other process is in its hazardous code region and its snapshot pointer is in $RL-PL$, then (the symmetric version of) $ishazard$ 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 $\text{MAXID} + 1 = Lid \leq Idq \leq \text{MAXID}$).

OP $\equiv Pop$: In the succeeding hazard pointer validation step (lines O10 / O11), φ_t and $ishazard$ 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 (line O13), we know from φ_{rl} that it can not be in the current process' retired list RL. Hence, we can establish φ_{rl} again in the retiring step (line O16), since both $OTop$ and RL are local.

OP $\equiv Push$: The allocation step (lines U3 / U4) resets the contents of a new cell. However, it does not affect allocated locations and thus neither rely condition ρ_n nor ρ_t 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 (φ_{rl}), when the new cell is added to the stack (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.

■

C. Linearizability and Lock-Freedom of the Stack

Preservation of Linearizability The actual proof of linearizability (proof obligation (6)) is not described in more detail, since it resembles our former proof under garbage collection [8], [9]. It 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 (φ_{rl}).

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 ρ_t ensures that the next reference of the snapshot cell and its value are immutable. Thus the successful CAS corresponds to an abstract pop, returning the correct value. In case of a push operation, the linearization point is the successful CAS. Rely ρ_n 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 (7), 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 over the top-of-stack pointer.

$$U(GS_0, GS_1) :\leftrightarrow Top_1 = Top_0$$

It is then relatively simple (compared to [10], where we prove lock-freedom of a more complicated lock-free queue) to show that the push and pop operations terminate. Since the scan operation is wait-free, we can prove its termination without U.

$$Scan(; LS, GS),\ \Box\ LS'' = LS' \vdash \Diamond\ \mathbf{last}$$

Termination of the first scan loop uses well-founded induction over the term MAXID$-Lid$ which decreases in every iteration. Similarly, termination of the second loop follows with an inductive argument over the number of retired locations.

VII. Related Work and Comparison

Related approaches that have been used to verify lock-free algorithms, can be classified according to their degree of automation as follows.

Automatic Approaches No current automatic technique proves correctness or lock-freedom, without assuming GC.

Model Checking is good at quickly finding bugs in concurrent algorithms [20]. Yet it only checks short executions of a few processes and thus possibly provides counterexamples, but not full proofs. Shape Analysis in contrast proves linearizability in systems with an unbounded number of processes [21].

An automatic approach for verifying linearizability is taken by Vafeiadis et al. [22] based on rely-guarantee reasoning and separation logic (RGSep). To verify lock-freedom, RGSep is substantially extended in [23], while our approach integrates correctness and liveness analysis in one logical framework.

Interactive Verifications As far as we know, there is no interactive (mechanized) verification of a lock-free algorithm with hazard pointers. [24] describes an automata-based proof of linearizability of a lock-free queue, considering the less advanced reclamation technique of modification counters [4]. To verify the dequeue operation which has a conditional linearization point that depends on system behavior, backward simulation is used, while we have verified linearizability (and lock-freedom) without additional techniques (cf. [9], [10]).

Manual Proofs Michael himself [6] gives a semantic verification condition which ensures safe memory reclamation for an arbitrary 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. Our concrete verification of the stack formally resembles Michael's arguments, while avoiding both global reasoning and reasoning about the past.

There are two formal pen and pencil proofs of a Treiber-like stack with hazard pointers. Parkinson et al. [7] apply concurrent separation logic (CSL) to verify the stack, focusing on heap-modular reasoning and fractional permissions. 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). They use history variables to capture this property, while we ensure ABA avoidance by directly claiming immutability of cell contents in a simple rely condition (ρ_t).

Fu et al. [25] verify the stack in a new program logic for history (HLRG), which extends previous work by Feng et al. [26]. It provides temporal operators of the past only and evaluates state assertions in the last state of an execution. Thus their logic is inherently limited to finite executions. Their operation "retireNode(t)" is not lock-free, since it does not complete when location t is covered by a hazard pointer and the associated process fails. However, the combination of temporal logic, rely-guarantee reasoning and separation logic is a promising approach that is relevant for our future work.

Both related formal verifications [7], [25] view the stack as an implementation of a lock-free memory allocator which integrates hazard pointers in its access operations, abstracting from cell data. In contrast, we abstract from the underlying memory allocator and focus on a lock-free data-stack implementation in an environment without garbage collection. HLRG and CSL 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 explicitely. Their verification considers memory-safety and structural invariance of the free-stack only. They prove neither linearizability nor lock-freedom of the stack. Their proofs use global conditions which quantify over all processes, without exploiting the symmetry of the reclamation technique.

VIII. Summary

This work describes the first mechanized verification of a challenging lock-free stack with hazard pointers. The proof shows central properties of the hazard pointers method and takes advantage of the relation between Michael's method and GC, carrying over the verification conditions from the simpler proof under GC with just minor adaptations. As we believe, these ideas can be reused for the verification of other extended lock-free data structures, possibly in other logics and formalisms as well. As a step towards higher automation, our proof avoids global reasoning by exploiting the symmetry of the hazard pointers method, which allows to express the central correctness conditions in terms of two processes only. Moreover, the proof is, as we believe, intuitive since it avoids any reasoning about additional history variables and the temporal past.

As an extension of our verification, 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 global value. The proofs only changed minimally and are available online too [8].

In future work, we will verify further algorithms that use hazard pointers. Moreover, we strive to generalize and improve the decomposition of linearizability to treat complex linearization points (such as in the elimination stack [3]), applying these improved techniques to further challenging case studies.

Acknowledgments

We thank Jörg Pfähler who has done all proofs of the stack under GC. Further, we owe thanks to Alexander Knapp for helpful comments on this work and Maged Michael for sharing his background knowledge about lock-free data structures with us.

References

[1] H. Massalin and C. Pu, "A lock-free multiprocessor os kernel," Columbia University, Tech. Rep. CUCS-005-91, 1991.

[2] M. Herlihy and J. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[3] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *SPAA '04: ACM symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM Press, 2004, pp. 206–215.

[4] R. K. Treiber, "System programming: Coping with parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, 1986.

[5] T. E. Hart, P. E. Mckenney, and A. D. Brown, "Making lockless synchronization fast: Performance implications of memory reclamation," in *IPDPS 2006*, 2006.

[6] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004.

[7] M. Parkinson, R. Bornat, and P. O'Hearn, "Modular verification of a non-blocking stack," *SIGPLAN Not.*, vol. 42, no. 1, pp. 297–302, 2007.

[8] "Web presentation of KIV proofs for lock-free algorithms in ITL," 2010, URL: http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html.

[9] S. Bäumler, G. Schellhorn, B. Tofan, and W. Reif, "Proving linearizability with temporal logic," *FAC*, 2009, appeared online first, http://www.springerlink.com/content/7507m59834066h04/.

[10] B. Tofan, S. Bäumler, G. Schellhorn, and W. Reif, "Temporal logic verification of lock-freedom," in *Proc. MPC 2010*, ser. Springer LNCS 6120, 2010, pp. 377–396.

[11] B. Moszkowski, *Executing Temporal Logic Programs*. Cambridge University Press, 1986.

[12] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser, "Structured specifications and interactive proofs with KIV," in *Automated Deduction—A Basis for Applications*, W. Bibel and P. Schmitt, Eds. Dordrecht: Kluwer Academic Publishers, 1998, vol. II: Systems and Implementation Techniques, ch. 1: Interactive Theorem Proving, pp. 13 – 39.

[13] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers, *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001, no. 54.

[14] R. M. Burstall, "Program proving as hand simulation with a little induction," *Information processing 74*, pp. 309–312, 1974.

[15] S. Bäumler, M. Balser, F. Nafz, W. Reif, and G. Schellhorn, "Interactive verification of concurrent systems using symbolic execution," *AI Communications*, vol. 23, no. (2,3), pp. 285–307, 2010.

[16] M. Balser, "Verifying concurrent system with symbolic execution – temporal reasoning is symbolic execution with a little induction," Ph.D. dissertation, University of Augsburg, Augsburg, Germany, 2005.

[17] R. Colvin and B. Dongol, "A general technique for proving lock-freedom," *Sci. Comput. Program.*, vol. 74, no. 3, pp. 143–165, 2009.

[18] C. B. Jones, "Specification and design of (parallel) programs," in *Proceedings of IFIP'83*. North-Holland, 1983, pp. 321–332.

[19] J. Misra, "A reduction theorem for concurrent object-oriented programs," in *Programming methodology*, A. McIver and C. Morgan, Eds. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 69–92.

[20] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-up: a complete and automatic linearizability checker," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 330–340.

[21] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv, "Thread quantification for concurrent shape analysis," in *CAV'08*. Springer, 2008.

[22] V. Vafeiadis, "Modular fine-grained concurrency verification," Ph.D. dissertation, University of Cambridge, 2007.

[23] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis, "Proving that nonblocking algorithms don't block," in *Principles of Prog. Lang.* ACM, 2009, pp. 16–28.

[24] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Formal verification of a practical lock-free queue algorithm," in *FORTE 2004*, ser. LNCS, vol. 3235, 2004, pp. 97–114.

[25] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang, "Reasoning about optimistic concurrency using a program logic for history," in *CONCUR*, 2010, pp. 388–402.

[26] X. Feng, "Local rely-guarantee reasoning," in *Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, 2009, pp. 315–327.