Electronic Communications of the EASST
Volume 66 (2013)

Proceedings of the
Automated Verification of Critical Systems
(AVoCS 2013)

Compositional Verification of a Lock-Free Stack with RGITL

Bogdan Tofan, Gerhard Schellhorn, Gidon Ernst, Jörg Pfähler and Wolfgang Reif

15 pages

Guest Editors: Steve Schneider, Helen Treharne
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Compositional Verification of a Lock-Free Stack with RGITL

Bogdan Tofan, Gerhard Schellhorn, Gidon Ernst, Jörg Pfähler and Wolfgang Reif

(tofan,schellhorn,ernst,joerg.pfaehler,reif)@informatik.uni-augsburg.de
Institute for Software and Systems Engineering
University of Augsburg

Abstract: This paper describes a compositional verification approach for concurrent algorithms based on the logic Rely-Guarantee Interval Temporal Logic (RGITL), which is implemented in the interactive theorem prover KIV. The logic makes it possible to mechanically derive and apply decomposition theorems for safety and liveness properties. Decomposition theorems for rely-guarantee reasoning, linearizability and lock-freedom are described and applied on a non-trivial running example, a lock-free data stack implementation that uses an explicit allocator stack for memory reuse. To deal with the heap, a lightweight approach that combines ownership annotations and separation logic is taken.

Keywords: Compositional Verification, Rely-Guarantee Reasoning, Linearizability, Lock-Freedom, Ownership, Separation Logic

1 Introduction

Multi-core processors have become ubiquituous in our computers. One area, where lots of progress has been made over the last decade, is the efficient implementation of standard data types such as stacks, queues, sets, etc. Instead of simply locking the full data structure on every operation, these implementations either use sophisticated fine-grained locking schemes, or alternatively nonblocking techniques that avoid the use of locks. Such algorithms are used in operating system kernels, and are also included in libraries of common programming languages, e.g., Threading Building Blocks for C++, java.util.concurrent for Java, or System.Collections.Concurrent for C#.

A central safety property of these concurrent data types is linearizability [HW90]. Roughly speaking, a concurrent operation is linearizable if it corresponds to some atomic operation of an abstract data type. For nonblocking data structure implementations, the liveness property of lock-freedom [MP91] is common. It excludes both livelocks and deadlocks, even in the presence of indefinite delays of individual processes that access the data structure. However, formal correctness proofs of such algorithms are difficult.

We propose a fully mechanized, interactive verification approach. It uses the rely-guarantee (RG) method [Jon83], which is concerned with verifying general safety properties of concurrent systems with shared resources. RG reasoning provides a modular treatment of interference between system components, i.e., to analyze properties of the overall system, each component can be examined separately based on its specification of expected environment behavior. However, the original RG approach does not deal with specific aspects of heap manipulating algorithms, neither does it prove linearizability, nor does it address liveness verification (lock-freedom).

In our earlier work [STER11], we therefore proposed the logic RGITL, which integrates RG reasoning into a compositional temporal logic that can express a wide range of safety and liveness properties. RGITL has been implemented and mechanically verified correct in the interactive theorem prover KIV [KIV13]. We have derived decomposition theorems for linearizability [BSTR11] and lock-freedom [TBSR10] in the logic, and we describe a challenging application in [TSR11a].

Here we take an improved approach to deal with the heap, which also facilitates RG reasoning. Furthermore, a more generic approach for verifying lock-freedom is defined. These improved techniques are illustrated on a non-trivial running example, which to our knowledge, has had no mechanized verification before. More specifically, we previously [TSR11a] used disjointness and reachability predicates to explicitly model disjoint parts of the heap. Here we use ownership annotations [BDF^{+}04] of the program code to implicitly label portions of the heap with a distinct owner, plus separation logic's star operator [Rey02] to model acyclic heap structures. While in [TSR11a] we used a RG decomposition of the overall program state to two local process states (and the shared state), we can now use RG conditions on the shared state only, where interference might actually occur, and do not have to take the local state of any other process into account.

The improved approach is illustrated on a concurrent implementation based on the well-known lock-free stack from [Tre86]. Several other proofs for the stack do exist, but have mainly focused on linearizability for the simple version under garbage collection. Here we consider a version that is close to an implementation in environments without garbage collection. It uses generic lock-free push and pop operations in two contexts: first, to add / remove arbitrary data from a lock-free data stack and second, to add / remove heap locations from another stack that serves as a lock-free memory allocator. The verification of the full version poses additional challenges w.r.t. reasoning about the heap, the fundamental ABA problem of lock-free algorithms, compositional verification of sequential code (when verifying the client code, we want to reuse the proofs of the generic stack operations), and it also requires a generalization of our previous termination conditions for lock-freedom [TBSR10]. All proofs have been mechanized: KIV proofs for the running example and the decomposition theory can be accessed online [KIV13].

The rest of this paper is structured as follows: Section 2 introduces the running example. Section 3 gives a short introduction to RGITL and how RG reasoning is expressed in the logic. Section 4 describes the RG approach using ownership annotations and separation logic. Section 5 gives two decomposition theorems for linearizability and lock-freedom. Finally, Section 6 compares our approach with related work and Section 7 concludes with a short summary and possible future work.

2 The Lock-Free Data and Free Stacks

This section introduces the running example, which is used to illustrate our approach. Lock-free algorithms typically apply atomic synchronization primitives such as CAS (Compare-And-Swap) instead of locks.

```
atomic CAS(old, new, now) returns {bool}
    if (now = old) {now := new; return true;}
    else return false;
```

```
record Node = {.val:data, .nxt:ref};
record RC   = {.ref:ref, .cnt:nat};
var DTop,FTop:RC;

PUSH(d:data)
  var new := DoPop(FTop);
  if (new = null) {
    alloc(new);
  };
  new.val := d;
  DoPush(new, DTop);

DoPush(node:ref(Node), Top:ref(RC)
  var ltop:RC;
  repeat
    ltop := Top;
    node.nxt := ltop.ref;
  until
    CAS(ltop, (node, ltop.cnt + 1), Top);
```

```
POP() returns {empty | data}
  var rtop := DoPop(DTop);
  if (rtop = null) {
    return empty;
  };
  var lout := rtop.val;
  DoPush(rtop, FTop);
  return lout;

DoPop(Top:ref(RC)) returns {ref(Node)}
  var ltop:RC;
  repeat
    ltop := Top;
    if (ltop.ref = null) {break;}
    nxt := (ltop.ref).nxt;
  until
    CAS(ltop, (nxt, ltop.cnt), Top);
  return ltop.ref;
```

Figure 1: Running Example: Lock-Free Data and Free Stacks.

CAS compares the current value of a shared variable *now* with an older local copy of it *old*, called snapshot. If these values are equal, the *now* variable is updated to a new value *new* and true is returned; otherwise false is returned. The execution of CAS is atomic.

We consider a lock-free implementation of a data stack that is used by some finite number of concurrent processes to store arbitrary data. Since the application's environment does not offer garbage collection (GC), a second lock-free stack is used to allocate and free memory. This stack is called the free stack in the following. Explicit memory allocators are common in environments without (lock-free) GC, to avoid memory leaks. Both stacks are implemented as singly linked lists of nodes (pairs of values and locations having .val and .nxt selector functions). A shared variable *DTop* marks the top node of the data stack; it is a pair of a reference and a modification counter (of type $nat = \mathbb{N}_0$), with selector functions .ref and .cnt, respectively. Similarly, the free stack is accessible from a shared variable *FTop*. Each process that wants to push some data *d* on the data stack, first tries to allocate memory from the free stack and resorts to the machine's allocator (alloc) only if the free stack is currently empty. Whenever a process pops a node from the data stack, after reading its data, it pushes the node on the free stack, thus making its memory available for (concurrent) reuse.

Modification counters are widely used for CAS based data structures, since the concurrent reuse of locations can lead to corruption of the data structure, when a location is reinserted with modified contents and this reinsertion is not detected by CAS. This is a fundamental issue of CAS based implementations called the ABA problem [Tre86]. To detect the concurrent reuse, a modification counter is typically added to ABA prone shared resources, here to each top-of-stack pointer. This counter is incremented atomically with (either) the insertion or removal of a location from the data structure, thus making memory reuse visible to CAS. (The theoretical chance of bogus behavior due to wrap around of a modification counter is negligible [Tre86].)

Figure 1 shows the concrete algorithms in pseudo code. The client procedures PUSH and POP use two generic procedures DoPush and DoPop, which operate on either the data or the free stack. Operation DoPush repeatedly tries to switch a shared top-of-stack pointer *Top*.ref to a new node using CAS in a lock-free manner. It repeatedly takes an atomic snapshot of

Top (including both the current top-of-stack pointer and its modification counter). After setting the new node's next pointer to the snapshot's location, the new node becomes the target of a subsequent CAS: if it succeeds, the node is atomically added to the stack and the modification counter is incremented. The generic pop operation DoPop works similarly: it repeatedly reads the shared top. If its pointer is null, then it immediately returns with null. Otherwise, its next reference becomes the target of a subsequent CAS: if it succeeds, the top node is removed from the stack and the snapshot's location is subsequently returned. (In case of high contention on the top-of-stack pointer, further techniques such as elimination or diffraction can be helpful.)

Without a modification counter, an ABA problem could occur as follows: suppose that a pop process p takes a snapshot of the top pointer when the data stack consists of exactly one node at location A and the free stack is empty. Process p is preempted after setting its local reference *nxt* to null for another process, which removes A from the data stack without yet freeing it. Subsequently, a third process q executes a successful push, thereby allocating a new location B (by resorting to the machine's allocator). Then A is freed and q pushes A on the data stack, which now has two nodes at locations [A, B]. If now p is rescheduled, its CAS would erroneously succeed, removing both nodes A and B at once and possibly returning an unexpected value.

The additional verification challenges that arise from having an explicit memory allocator rather than assuming GC, are as follows. First, it becomes necessary to prove that the application does not leak memory. Here we use ownership annotations and separation logic's star operator to state that the application's heap is always separated into three distinct parts: one for the data stack, another one for the free stack and a third part that is owned by some of the (running) processes. Second, we have to show that an ABA problem does not occur. Here RG reasoning permits to express ABA prevention as an appropriate rely condition. Third, to avoid big redundant proofs, we want to verify the generic procedures DoPush and DoPop separately, and reuse these proofs as contracts in the verification of the sequential client code PUSH and POP, respectively. Here the sequential compositionality of RGITL is crucial, which allows us to replace the generic procedure calls with appropriate RG, linearizability and lock-freedom abstractions, respectively. Finally, since individual processes might now starve while accessing either the data or the free stack, the argument why individual starvation does not lead to global livelock becomes non-trivial. By slightly generalizing our previous decomposition of lock-freedom, we can yet prove lock-freedom for the example.

3 RGITL

This section gives a brief introduction to basic concepts of RGITL. We emphasize that it is not in the scope of this paper to detail every aspect of the logic. Instead, the interested reader is referred to [STER11]. A formal specification of the semantics of RGITL, including soundness proofs of the main rules are available online [KIV13].

3.1 Syntax and Semantics

The semantics of expressions of the logic is based on intervals, which are finite or infinite sequences of states. A state maps variables to values. Variables can be either dynamic (written uppercase) or static (written lowercase), where the latter do not change their value throughout an interval. The last state of a finite interval is characterized by the formula **last**. The interval semantics explicitly includes environment behavior, similar to reactive sequences in [RBH+01]: each interval $I = [I(0), I(0)', I(1), I(1)', \ldots]$ alternates system (or program) transitions with environment transitions, i.e., the 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 the environment and so forth.

Priming of a dynamic variable V denotes over which state of an interval V has to be evaluated. While an unprimed variable V is evaluated over the first state $I(0)$, the primed variable V' is evaluated over $I(0)'$ and the double primed variable V'' over $I(1)$, respectively. Hence, formula $V' = 1$ states that V has value 1 after the first program transition of a given interval and formula $V'' = 2$ states that V has value 2 after the first environment transition of the interval. (For an empty interval [I(0)], both V' and V'' are evaluated over I(0).) Similarly, a predicate logic formula $G(V, V')$ specifies a guarantee for the first program transition and a formula $R(V', V'')$ defines a rely condition for the first environment transition.

RGITL provides standard temporal operators to describe interval properties: $\circ\ \varphi$ (strong next) holds in an interval I (written $I \models \circ\ \varphi$) iff I is not empty and formula φ holds in I's postfix interval $[I(1), \ldots]$. The formula φ_1 **until** φ_2 holds in I iff φ_2 holds over $[I(n), \ldots]$ for some interval state $I(n)$ and φ_1 holds in $[I(m), \ldots]$ for each $m < n$. Further standard operators are introduced as abbreviations, e.g., $\bullet\ \varphi \equiv \neg \circ \neg\ \varphi$ (weak next), $\Diamond\ \varphi \equiv$ true **until** φ, or $\Box\ \varphi \equiv \neg \Diamond \neg\ \varphi$. Hence, formula $\Box\ V' = V''$ states that throughout an interval, no environment transition ever changes the value of V.

Similar to ITL [Mos00], programs α are just a subset of the formulas of the logic. Hence, programs and formulas can be mixed. An interval I satisfies α (written $I \models \alpha$) iff I alternates α (program) transitions with arbitrary environment transitions. Finite intervals correspond to terminating program runs. The logic provides the common constructs for sequential programs, including nondeterministic choice, recursive procedures, plus an interleaving operator $||$. For brevity, we only give the interval semantics of the sequential composition operator ";" here. (This definition corresponds to the definition of the chop operator from ITL.)

$I \models \varphi_1; \varphi_2$ iff either I is infinite and satisfies φ_1, or there is a finite prefix of I where
φ_1 holds and φ_2 holds for the rest of I

3.2 Deduction

The assertion language is based on the sequent calculus. A sequent is an assertion of the form $\Gamma \vdash \Delta$ (where Γ and Δ are lists of formulas), which states that the conjunction of all formulas in antecedent Γ implies the disjunction of all formulas in succedent Δ. A sequent is implicitly universally closed. A typical temporal logic assertion for a program α has the form

$$Pre, \alpha, E \vdash \varphi$$

where Pre is a precondition for the initial state, formula E is an environment assumption, i.e., a temporal formula over primed and double primed variables, and φ is the property of α to be shown.

For deduction, the standard rules of the sequent calculus are used. Moreover, the following compositionality principle holds for the sequential composition operator (similar to ITL).

$$\frac{\alpha_1 \vdash \psi_1 \qquad \psi_1;\varphi_2 \vdash \psi}{\alpha_1;\varphi_2 \vdash \psi} \text{ compositionality of ;} \tag{1}$$

The rule states that formula ψ can be derived for the sequential composition $\alpha_1;\varphi_2$ if an abstraction ψ_1 can be derived for α_1 (premise 1), and ψ can be derived for $\psi_1;\varphi_2$ (premise 2). Rule (1) allows us to split the proof of the client code PUSH and POP into a part that separately verifies the generic operations DoPush and DoPop (premise 1) and a second part that verifies the client code using appropriate abstractions as contracts for the generic procedures (premise 2).

Temporal formulas and programs are verified using symbolic execution. Basically, a symbolic execution step moves forward to the next state of an interval in two phases. In the first phase, each formula of a sequent is transformed into an equivalent formula that consists of two parts: one part that refers to the first three states of an interval, and another part that refers to the rest of the interval from the third state on (using a leading next operator). The second phase then removes leading next operators and replaces variables V, V', V'' with new variables v_0, v_1, V, where the static variables v_0 and v_1 store the “old” values of V in $I(0)$ and $I(0)'$, respectively.

Example 1 *A symbolic execution step of the sequent*

$$V = 0,\ (V := V+2;\alpha),\ \Box\, V' = V'' \vdash \circ\, V = 2$$

generates the following new sequent.

$$v_0 = 0,\ v_1 = v_0+2,\ \alpha,\ v_1 = V,\ \Box\, V' = V'' \vdash V = 2$$

Executing the first assignment ($V := V+2$) results in $v_1 = v_0+2$ and the remaining program is α. The environment assumption has been unwound (using the equivalence $\Box\,\varphi \leftrightarrow \varphi \wedge \bullet\,\Box\,\varphi$), which gives the constraint $v_1 = V$ for the first environment transition and again $\Box\, V' = V''$ for the rest.

Finally, the logic provides induction rules that permit to reduce the verification of safety properties φ over an infinite interval I, to the verification of φ over an arbitrary finite prefix of I. Then well-founded induction over the length of the finite prefix is used to deal with loops during symbolic execution. This is necessary for the stack, since CAS loops can iterate indefinitely often due to concurrent changes of the shared top-of-stack variables.

3.3 Rely-Guarantee Assertions

A compositionality rule, which is similar to rule (1), also holds for the interleaving operator in RGITL. This makes it possible to derive decomposition rules as theorems for interleaved programs in the logic. An important case of such decomposition rules are RG rules, which break down the verification of an RG assertion for a concurrent system, to the verification of a corresponding RG assertion for each system component. An RG assertion for a program $\alpha(V)$, which uses variables V, has the following form.

$$Pre(V) \vdash [R(V',V''), G(V,V'), \mathrm{Inv}(V), \alpha(V)]\ Post(V)$$

Informally, an RG assertion states that runs of α that start in an initial state *Pre*, preserve the guarantee G and the invariant *Inv* as long as previous environment transitions satisfy the rely R and also maintain *Inv*. In finite executions of α, the last state satisfies the postcondition *Post*.

Formally, an RG assertion expands to

$$\begin{aligned} Pre(V), \alpha(V) \vdash \quad & (R(V', V'') \wedge (Inv(V') \rightarrow Inv(V''))) \\ & \xrightarrow{+} (\textbf{if last then } Post(V) \textbf{ else } G(V, V') \wedge (Inv(V) \rightarrow Inv(V'))) \end{aligned}$$

where operator $\varphi_1 \xrightarrow{+} \varphi_2$ abbreviates the formula $\neg\, (\varphi_1 \textbf{ until } \neg\, \varphi_2)$. Thus, RG assertions are safety formulas, which can be verified by well-founded induction over the length of a finite interval prefix.

The definitions and the decomposition theory in the rest of this paper are not hard-wired into the semantics of the logic, but built on top of it as higher-order and temporal logic specifications. In particular, the RG theorem (cf. Section 4) and the decomposition theorems for linearizability and lock-freedom (cf. Section 5) are derived in the logic.

4 Rely-Guarantee Reasoning with Ownership and Separation

This section describes the RG decomposition rule that lies at the core of the decomposition theorems for linearizability and lock-freedom and explains how ownership annotations, known from the verification of object-oriented sequential programs [BDF+04], facilitate its applicability. One consequence of these annotations is that separation logic's star operator $*$ [Rey02] can be used to model the shape of the heap. Our experience with sequential program verification shows that using $*$ can be advantageous over the use of inductive reachability predicates when reasoning about acyclic heap structures. However, introducing $*$ in a concurrent setting is more delicate than in the sequential case, which is mainly due to possible concurrent changes of the heap.

4.1 Rely-Guarantee Decomposition

Our RG rule is similar to the original rule of Jones [Jon83], since rely and guarantee conditions are simply binary predicates over an arbitrary shared variable S : state that represents the state of the concurrent system. The system recursively interleaves $n+1$ processes $p : \mathbb{N}_0$, where each process executes indefinitely often a generic procedure COP.

$$\mathrm{COP}(0, In; S, Out)^* \;||\; \ldots \;||\; \mathrm{COP}(n, In; S, Out)^*$$

The KIV syntax is as follows: the star operator * denotes finite or infinite iteration; the parameters of procedure COP are separated by a semicolon into input resp. in-output parameters, where parameter *In* is an input for the procedure, which writes its return value to the in-output parameter *Out* (a return statement is not used). [1]

[1] An additional operation index and a history variable to which an invoke and a return event are added before resp. after each COP call, are required to prove linearizability; an auxiliary variable is necessary in the lock-freedom decomposition proof. They are omitted here.

The following RG decomposition rule is derivable in the logic [STER11].

$$\frac{\begin{array}{l} p \neq q, G_p \vdash R_q \quad \vdash \mathrm{trans}(R_p) \quad \vdash \mathrm{refl}(G_p) \quad \vdash \exists\, S.\ \mathit{Init}(S) \quad \mathit{Init} \vdash \mathit{Idle} \wedge \mathit{Inv} \\ \vdash \mathrm{stable}(R_p, \mathit{Idle}_p) \quad \mathit{Inv}, \mathit{Idle}_p \vdash [R_p, G_p, \mathit{Inv}, \mathrm{COP}(p, \ldots)]\ \mathit{Idle}_p \end{array}}{\mathit{Init} \vdash [R, G, \mathit{Inv}, \mathrm{COP}(0, \ldots)^* \| \ldots \| \mathrm{COP}(n, \ldots)^*]\ \mathit{Idle}} \tag{2}$$

$$\text{where } R \equiv \bigwedge_p R_p,\ G \equiv \bigvee_p G_p,\ \mathit{Idle} \equiv \bigwedge_p \mathit{Idle}_p$$

Essentially, the rule decomposes a global RG assertion about the interleaved system, to the following local RG assertion for one process p, executing COP once.

$$\mathit{Idle}_p(S), \mathit{Inv}(S) \vdash [R_p(S', S''), G_p(S, S'), \mathit{Inv}(S), \mathrm{COP}(p, \mathit{In}; S, \mathit{Out})]\ \mathit{Idle}_p(S)$$

The rule also has several predicate logic side conditions. An important one is that the local guarantee G_p of a process p must imply the rely R_q for each other process $q \neq p$. The local guarantee must be reflexive and the local rely transitive. A global initial state must exist, where the invariant Inv must hold and each process is idle, according to its idle state predicate Idle_p, and each idle predicate is assumed to be stable over its rely, i.e., $\mathrm{stable}(R_p, \mathit{Idle}_p) \leftrightarrow R_p(S', S'') \wedge \mathit{Idle}_p(S') \rightarrow \mathit{Idle}_p(S'')$.

Applying rule (2) on the running example directly is inconvenient: it requires taking the entire program state into account by lifting relevant local state information to the global state using process functions for local variables. As an example, consider the following disjointness property: concurrent local pointers to nodes that are pushed on one of the stacks are disjoint. With an extra boolean function $\mathrm{BefCAS} : \mathbb{N} \rightarrow \mathit{bool}$ to characterize the code-range of the CAS-loop in DoPush before it succeeds, it can be formalized as

$$\forall\ p \neq q.\ \mathrm{BefCAS}(p) \wedge \mathrm{BefCAS}(q) \rightarrow \mathrm{Node}(p) \neq \mathrm{Node}(q)$$

In [TSR11a], we therefore proposed a local RG rule that considers two explicit local states (and the shared state), and thus avoids process functions and quantification over process identifiers. Here we take a different approach based on ownership annotations that leads to further significant simplifications.

4.2 Ownership Annotations for the Stack

The core idea of using ownership annotations is simple: shared resources are augmented with auxiliary state that represents their distinct owner. The idea is applicable on shared resources in general, but we restrict our focus on concurrent heaps in the following. A concurrent heap with ownership is a partial function $H : \mathit{ref} \rightharpoonup (\mathit{node}, \mathit{owner})$ from locations to nodes with owner o.

In the running example, it is sufficient to discern three possible owners per heap location:

$$\mathit{owner} \ ::= \ p \mid \mathit{dstack} \mid \mathit{fstack}$$

That is, each heap location r is either owned by some process p (i.e., $H(r).\mathit{owr} = p$), or it belongs to either the data or the free stack. Additionally, we augment the program code to adhere to this ownership concept as Figure 2 shows. [2]

[2] The KIV syntax is as follows: $\vee$ denotes nondeterministic choice, **let** declares local variables, a comma separates

```
COP$(p, d; DTop, FTop, H, Out)$ {
      PUSH$(p, d; DTop, FTop, H)$
   $\vee$ POP$(p; DTop, FTop, H, Out)$}

PUSH$(p, d; DTop, FTop, H)$ {
  let $New$ in {
    DoPop$(p; FTop, H, New)$;
    if $New = \text{null}$ {
      choose $(New_0 \neq \text{null} \wedge New_0 \notin H)$ in {
       $H := H + New_0, New := New_0$,
       $H(New_0).owr := p$}};
    $H(New).\text{val} := d$;
    DoPush$(dstack, New; DTop, H)$}}

POP$(p; DTop, FTop, H, Out)$ {
  let $LOut = \text{empty}, ROut$ in {
    DoPop$(p; DTop, H, ROut)$;
    if $ROut \neq \text{null}$ {
      $LOut := H(ROut).\text{val}$;
      DoPush$(fstack, ROut; FTop, H)$};
    $Out := LOut$}}
```

```
DoPush$(o, Node; Top, H)$ {
  let $Succ = \text{false}, LTop$ in {
    while $\neg\, Succ$ {
      $LTop := Top$;
      $H(Node).\text{nxt} := LTop.\text{ref}$;
      if* $LTop = Top$ {          /* CAS */
        $Top := (Node, LTop.\text{cnt} + 1), Succ := \text{true}$,
        $H(Node) := o$}}}

DoPop$(p; Top, H, ROut)$ {
  let $Succ = \text{false}, LTop, Nxt$ in {
    while $\neg\, Succ$ {
      $LTop := Top$;
      if $LTop.\text{ref} = \text{null}$ { $Succ := \text{true}$}
      else {
        $Nxt := H(LTop.\text{ref}).\text{nxt}$;
        if* $LTop = Top$ {          /* CAS */
          $Top.\text{ref} := Nxt, Succ := \text{true}$,
          $H(LTop.\text{ref}) := p$}}};
    $ROut := LTop.\text{ref}$}}
```

Figure 2: KIV Specification of the Stack Algorithms with Ownership Annotations (shaded)

The effects of these simple auxiliary state annotations are worth noting. First of all, no further heap disjointness properties must be defined, since they are already implied by the ownership annotations. (Our technical report [TSR11b] shows that several disjointness properties between local states would be necessary without ownership annotations.) Second, we can completely avoid talking about local variables, in particular program labels. Hence, when applying rule (2), the state variable S can be simply instantiated with the tuple $DTop, FTop, H$, which is the shared state of the algorithm, where interference can actually occur. (The local state of *one* currently running process could be added to the rule, but this is not required here.) Third, we can uniformly handle typical heap modifications and use separation logic on (owned) heap predicates to avoid inductive reachability arguments. This is further explained in the next subsection.

4.3 Concurrent Heaps with Ownership and Separation

Instead of integrating heaps into the semantics of RGITL, we use a lightweight embedding of separation logic into higher-order logic (available as a KIV library), where heap assertions are encoded as heap predicates P, Q of type $heap \rightarrow bool$. The lifting of this theory to heaps with ownership is done in the standard way: an owned heap predicate $o[P]$ with owner o holds over heap H, iff P holds and every location in H has owner o. Similarly, the common operators

parallel assignments and **if*** executes its test and program atomically. In the prover, the heap H is actually represented as a tuple (D, Nf, Of) with $dom(H)$ = D, node function Nf and an auxiliary ownership function Of.

from separation logic are overloaded. For instance, the star operator between two owned heap predicates $o_0[P]$ and $o_1[Q]$ has the following semantics.

$$(o_0[P] \ast o_1[Q])(H) \leftrightarrow \exists\, H_0, H_1.\ dom(H_0) \cap dom(H_1) = \emptyset \wedge (H_0 \cup H_1 = H) \wedge P(H_0) \wedge Q(H_1) \\ \wedge \forall\, r.\ (r \in H_0 \to H_0(r).owr = o_0) \wedge (r \in H_1 \to H_1(r).owr = o_1)$$

In a concurrent setting, assertions about the permissions of processes to access shared resources are typically required. Again, we do not enrich the semantics of RGITL with permissions, but simply define them based on ownership. It is common to assume that a heap location, which is owned by some *process* can only be read by others, but neither deallocated, nor modified. The following rely predicate encodes this restriction.

$$PR_p(H', H'') \leftrightarrow \forall\, r.\quad ((H'(r).owr = p \wedge r \in H') \leftrightarrow (H''(r).owr = p \wedge r \in H'')) \\ \wedge (H'(r).owr = p \wedge r \in H' \to H'(r) = H''(r))$$

The rely $PR_p(H', H'')$ implies the following stability property

$$(p[P] \ast \mathrm{true})(H') \wedge PR_p(H', H'') \to (p[P] \ast \mathrm{true})(H'')$$

and it is easy to prove that under PR_p, the annotated program $\mathrm{COP}(p, \ldots)$ does not change any portion of the heap, which is owned by another process.

To express absence of memory leaks, three simple heap predicates are defined: $owned(H)$ states that each r in H is owned by some process; $owns\text{-}none_p(H)$ denotes that p owns no location in H, and $owns\text{-}one_{p,r}(H)$ denotes that p owns exactly location r in H. Obviously, predicates $owns\text{-}none_p$ and $owns\text{-}one_{p,r}$ are stable over the permission rely PR_p, and it is easy to show that predicate $owns\text{-}none_p$ is an idle state condition of the annotated program $\mathrm{COP}(p, \ldots)$.

Finally, to verify linearizability we want to express that some abstract data list x is represented by a heap location r. The heap predicate $lst(r)$ defines this property, and the $\ast$ operator enforces acyclicity of the heap structure under r.

$$lst(r) = \textbf{if } r = \mathrm{null} \textbf{ then } ls(r, [\,]) \textbf{ else } \exists\, d, x.\ ls(r, d + x)$$
$$ls(r, []) = \mathrm{emp} \wedge (r = null)$$
$$ls(r, d + x) = \exists\, r_0.\ ((r \mapsto (d, r_0)) \ast ls(r_0, x))$$

where emp holds for the empty heap only, and $(r \mapsto (d, r_0))$ defines a heap consisting of one node at location r, which stores data d and a next reference r_0.

4.4 Instantiating the RG Predicates for the Running Example

This section defines the concrete instances of the predicates from rule (2) based on the previous notions of concurrent heaps. The state variable S is simply $DTop, FTop, H$. The global initial state condition $Init(DTop, FTop, H)$ requires H to be empty, and both $DTop$ and $FTop$ to be $(\mathrm{null}, 0)$. The invariant claims that the heap always consists of two distinct linked lists (with owner *dstack* resp. *fstack*) and a separate portion where each location is owned by some process.

$$Inv(DTop, FTop, H) \leftrightarrow (dstack[lst(DTop.\mathrm{ref})] \ast fstack[lst(FTop.\mathrm{ref})] \ast owned)(H)$$

The idle state predicate $Idle_p(DTop, FTop, H)$ is simply $owns\text{-}none_p(H)$. Since each process owns no portion of H in idle states, the application does not leak memory.

For stack nodes with owner *dstack* or *fstack*, the possible concurrent access is determined by the specific use of modification counters. In contrast to locations that are owned by some process, both the content and the ownership information of a stack location can change when the location is concurrently removed from the data structure. An appropriate stack rely condition that captures the correctness of the memory reclamation protocol and ensures that an ABA problem does not occur on neither the data nor the free stack is the following.

$$\begin{aligned}
&SR(o, Top', H', Top'', H'') \leftrightarrow Top'.\mathrm{cnt} \le Top''.\mathrm{cnt}\\
&\quad \wedge (Top'.\mathrm{ref} \neq \mathrm{null} \rightarrow \quad Top' = Top'' \wedge H'(Top'.\mathrm{ref}) = H''(Top'.\mathrm{ref}) \wedge H''(Top'.\mathrm{ref}).owr = o\\
&\qquad\qquad\qquad\qquad\qquad \vee H''(Top'.\mathrm{ref}).owr \neq o \vee Top'.\mathrm{cnt} < Top''.\mathrm{cnt})\\
&\quad \wedge (\forall\, r.\ r \neq \mathrm{null} \wedge H'(r).owr \neq o \rightarrow H''(r).owr \neq o \vee Top'.\mathrm{cnt} < Top''.\mathrm{cnt})
\end{aligned}$$

The specific stack relies $SR(dstack, \dots)$ and $SR(fstack, \dots)$ ensure that during DoPop on one of the two stacks, the ABA prone snapshot location either stays in the stack and its contents (including ownership annotation) are unchanged, or if it is concurrently removed, then it is not reinserted unless the modification counter is increased.

Finally, it remains to define the full rely R_p as the conjuction

$$\begin{aligned}
&R_p(DTop', FTop', H', DTop'', FTop'', H'') \leftrightarrow\\
&\qquad PR_p(H', H'') \wedge SR(dstack, DTop', H', DTop'', H'') \wedge SR(fstack, FTop', H', FTop'', H'')
\end{aligned}$$

and the guarantee as

$$G_p(DTop, FTop, H, DTop', FTop', H') \leftrightarrow \forall\, q \neq p.\ R_q(DTop, FTop, H, DTop', FTop', H')$$

The actual proof of the local RG assertion from rule (2) for the sequential code $\mathrm{COP}(p, \dots)$, uses compositionality rule (1). This splits the proof in two parts: one which verifies the RG assertion for DoPush and DoPop and a second part that uses the RG abstractions as contracts for the generic procedures. In the proofs, the current shape of the heap is typically given by the invariant *Inv* above and a formula $(p[(Ref \mapsto node)] * \mathrm{true})(H)$, which defines the local state of the current process (*Ref* corresponds to either the local variable *New* or *LTop*.ref). To transfer local state to and from one of the stacks, two simple generic merge and split lemmas are used. Verifying DoPop is most challenging, since transitive arguments over several symbolic execution steps are required to derive that if the snapshot location is concurrently removed, the following CAS operation does fail.

5 A Decomposition of Linearizability and Lock-Freedom

This section briefly describes two decomposition theorems for the global properties of linearizability [HW90] and lock-freedom [MP91]. Both theorems can be derived in RGITL, but their proofs are rather complex and we emphasize that neither their formalization nor their derivation is in the scope of this paper. Instead, the proofs are available online [KIV13].

5.1 A Decomposition of Linearizability based on RG Reasoning and Refinement

Having verified the premises of RG rule (2), from the local view of one process p executing $\mathrm{COP}(p,\dots)$, all environment transitions preserve its rely at all times and the invariant can be assumed to hold in each state, i.e., $\square\,(R_p(S',S'') \wedge Inv(S) \wedge Inv(S'))$ holds locally. This property is now used in a local refinement proof, which implies linearizability of the interleaved system.

Linearizability requires that an operation appears to take effect instantly in one step during its execution. This step is called a linearization point. We prove linearizability using a special case of non-atomic refinement from COP to an abstract program AOP, which identifies the linearization point for the concrete program as follows. The abstract program AOP is defined to execute some stutter steps first (indefinitely many). Then it executes the linearization point *LIN* atomically on the abstract state *AS*. (For the stack, this is simply an atomic push or pop operation on an algebraic data list x for *AS*.) Finally, some further stutter steps are executed until the operation finishes with final output value *Out*. Both concrete and abstract operation work on the same input and must yield the same output. Moreover, concrete and abstract states are always related by an abstraction function *Abs*. Hence, the main local proof obligation is

$$Idle_p(S), \mathrm{COP}(p,In;S,Out), \square\,(R_p(S',S'') \wedge Inv(S) \wedge Inv(S')), \\ \square\,(Abs(S) = AS \wedge Abs(S') = AS') \vdash \mathrm{AOP}(In;AS,Out) \tag{3}$$

where the abstract program is defined as

$$\mathrm{AOP}(In;AS,Out)\ \{\textbf{let}\ \ LOut\ \textbf{in}\ \{ \\ \quad \textbf{skip}^*;\{LIN(In,AS,AS',LOut') \wedge \circ\,\textbf{last}\};\textbf{skip}^*;Out := LOut\}\}$$

Theorem 1 *The concurrent system* $\mathrm{COP}(0,\dots)^*||\dots||\mathrm{COP}(n,\dots)^*$ *is linearizable if the premises of RG rule (2) and the refinement proof obligation (3) holds.*

The abstraction function for the data stack simply corresponds to $dstack[ls(DTop.\mathrm{ref},x)]$. The refinement proofs (3) for the stack also use rule (1). This gives compositional proofs that replace the generic push and pop operations with basically $\textbf{skip}^*;\{LIN \wedge \circ\,\textbf{last}\};\textbf{skip}^*$ or just $\textbf{skip}^*$ for the data and free stack, respectively.

5.2 A Decomposition of Lock-Freedom based on RG Reasoning

Lock-freedom is a progress property that is relevant in various application domains, such as high-availability or real-time systems. A concurrent system is lock-free if infinitely often one of its running operations progresses, i.e., both deadlocks and livelocks are excluded. In the following, two simple local termination conditions for an individual $\mathrm{COP}(p,\dots)$ are defined, which ensure lock-freedom of the interleaved system.

The first termination condition requires that COP must terminate whenever it does not suffer from critical interference from its environment. This interference is specified using an additional, reflexive and transitive predicate U (“unchanged”). [3]

$$Idle_p(S), \mathrm{COP}(p,In;\mathrm{S},Out), \square\,(R_p(S',S'') \wedge Inv(S) \wedge Inv(S')) \vdash \Diamond\,\square\,U(S',S'') \to \Diamond\,\textbf{last} \tag{4}$$

[3] Predicate U specifies under which conditions the termination of COP can be guaranteed. Different from rely conditions R_p, which are safety properties that always hold, predicate U can be repeatedly violated.

The second termination condition, enforces that COP violates U only a finite number of times. Formally, executions in which U is violated infinitely often in COP transitions are ruled out as follows.

$$Idle_p(S), \mathrm{COP}(p, In; \mathrm{S}, Out), \Box\ (R_p(S', S'') \wedge Inv(S) \wedge Inv(S')) \vdash \Box \Diamond \neg\ U(S, S') \rightarrow \Diamond\ \mathbf{last} \quad (5)$$

Theorem 2 *The concurrent system* $\mathrm{COP}(0, \ldots)^{*} || \ldots || \mathrm{COP}(n, \ldots)^{*}$ *is lock-free if the premises of RG rule (2) and the termination conditions (4) and (5) hold.*

In the running example, the unchanged relation is simply defined as the identity relation over *DTop* and *FTop*. The actual proofs of lock-freedom are also compositional, i.e., they verify (4) and (5) for the generic operations (largely automatically) and apply rule (1) to complete the proof for the client code.

Our previously published termination condition for lock-freedom [TBSR10] requires that when COP violates U once, then it subsequently terminates, i.e., the right hand side of (5) was $\Box\ (\neg\ U(S, S') \rightarrow \Diamond\ \mathbf{last})$. This can not be shown in the running example, since a step of POP can violate U by removing a node from the data stack and then the operation can starve while executing the subsequent CAS loop of the free stack. However, predicate U is violated at most once in infinite runs, which corresponds to our more generic proof obligation (5) that tolerates an arbitrary finite number of such violations.

6 Related Work

This section compares our work with related approaches. Full coverage is impossible here for two reasons: the plentitude of existing approaches and the lack of space.

To our knowledge, the only proof of the stack with modification counters is the pen-and-paper proof in Groves et al. [GC09]. Their verification approach is rather different from ours, since it is based on trace reduction and incremental refinement. They consider linearizability of a data stack with modification counters that reuses memory from an abstract set of free locations, while we consider an actual implementation of a free stack and give fully mechanized proofs of memory-safety, ABA prevention, linearizability and lock-freedom.

RGSep [VP07] is a program logic that combines RG reasoning and separation logic for heap-modular, Hoare-style reasoning about the safety of concurrent programs. A tool for automatically verifying linearizability based on RGSep has also been developed. In contrast to RGSep, heaps are not part of the semantics of RGITL, which makes no restrictions on the possible modifications of a program to a heap variable. This makes it more difficult for us to express which part of the heap a program leaves unmodified, without changing the semantics, and our current approach with ownership annotations is a step towards this end. In RGSep, local and shared assertions refer to either the local heap of one process or the shared heap, which corresponds to our annotations of portions of the heap with a distinct owner. Different from RGITL, deriving decomposition theorems, refinement and liveness proofs are not in the scope of RGSep.

Most approaches to RG reasoning justify their rules on a semantic level (e.g., [RBH^{+}01]). A mechanized soundness proof for global RG rules for interleaved programs with shared variables has been given in [Pre03]. The verification is based on Isabelle's higher-order logic and therefore in essence had to explicitly formalize intervals. Our soundness proofs of RG decomposition rules are simpler, since they are based on a compositional temporal logic, where intervals are already part of the semantics.

In general, reduction techniques for symmetric system components (as they can often be found in concurrent data type implementations) have also been developed for model checking. However, proving linearizability using (symmetric) model checking fails in general [VYY09]. Model checking approaches are fully automated and useful to quickly find bugs by checking short runs of usually two interleaved operations, but do not give full derivations.

Several concepts that we use in our approach are also implemented in tools for specific programming languages, e.g., [CMST10, JP08] for annotated (concurrent) C code.

7 Conclusion

This paper describes an approach for the verification of concurrent algorithms, which is based on a combination of different techniques. These were illustrated on a non-trivial running example. The approach incorporates RG reasoning into a compositional temporal logic, which also makes liveness proofs possible. For the verification of concurrent heap algorithms, ownership annotations and separation logic are used. Finally, we have briefly sketched two decomposition theorems for the important properties of linearizability and lock-freedom.

Some possible areas of future work are as follows. Concrete proofs of premise 2 of rule (1) require several interactions (mainly for the symbolic execution of the abstraction ψ_1 with induction). We leave it for our own future work to implement derived rules for specific classes of formulas, to better automate these proofs. Another option for future work is to integrate our current ownership and separation approach with the RG decomposition rule (2) and the verification of further challenging case studies.

Bibliography

[BDF$^+$04] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, W. Schulte. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology* 3, 2004.

[BSTR11] S. Bäumler, G. Schellhorn, B. Tofan, W. Reif. Proving Linearizability with Temporal Logic. *Formal Aspects of Computing (FAC)* 23(1):91–112, 2011.

[CMST10] E. Cohen, M. Moskal, W. Schulte, S. Tobies. Local verification of global invariants in concurrent programs. In *Proc. of CAV*. Pp. 480–494. Springer, 2010.

[GC09] L. Groves, R. Colvin. Trace-based Derivation of a Scalable Lock-Free Stack Algorithm. *Formal Aspects of Computing (FAC)* 21(1–2):187–223, 2009.

[HW90] M. Herlihy, J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Prog. Languages and Systems* 12(3):463–492, 1990.

[Jon83] C. B. Jones. Specification and Design of (Parallel) Programs. In *Proceedings of IFIP'83*. Pp. 321–332. North-Holland, 1983.

[JP08] B. Jacobs, F. Piessens. The VeriFast Program Verifier. Technical Report CW-520, KU Leuven, 2008.

[KIV13] KIV. Presentation of KIV proofs for AVOCS'13. 2013. URL: https://swt.informatik.uni-augsburg.de/swt/projects/avocs13.html.

[Mos00] B. C. Moszkowski. A Complete Axiomatization of Interval Temporal Logic with Infinite Time. In *LICS 2000: Proc. of the 15th IEEE Symposium on Logic in Computer Science*. Pp. 241–252. IEEE Computer Society Press, 2000.

[MP91] H. Massalin, C. Pu. A Lock-Free Multiprocessor OS Kernel. Technical report CUCS-005-91, Columbia University, 1991.

[Pre03] L. Prensa Nieto. The Rely-Guarantee method in Isabelle /HOL. In Degano (ed.), *ESOP'03*. LNCS 2618, pp. 348–362. Springer, 2003.

[RBH$^+$01] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in TCS 54. Cambridge University Press, 2001.

[Rey02] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS*. Pp. 55–74. IEEE Computer Society, 2002.

[STER11] G. Schellhorn, B. Tofan, G. Ernst, W. Reif. Interleaved Programs and Rely-Guarantee Reasoning with ITL. In *Proc. of TIME*. IEEE CS, pp. 99–106. 2011.

[TBSR10] B. Tofan, S. Bäumler, G. Schellhorn, W. Reif. Temporal Logic Verification of Lock-Freedom. In *In Proc. of MPC 2010*. Springer LNCS 6120, pp. 377–396. 2010.

[Tre86] R. K. Treiber. System programming: Coping with parallelism. Technical report RJ 5118, IBM Almaden Research Center, 1986.

[TSR11a] B. Tofan, G. Schellhorn, W. Reif. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *Proc. ICTAC*. Pp. 239–255. Springer LNCS 6916, 2011.

[TSR11b] B. Tofan, G. Schellhorn, W. Reif. Local Rely-Guarantee Conditions for Linearizability and Lock-Freedom. Reports in Informatics 26, KIT, 2011.

[VP07] V. Vafeiadis, M. J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*. Springer LNCS 4703, pp. 256–271. 2007.

[VYY09] M. Vechev, E. Yahav, G. Yorsh. Experience with Model Checking Linearizability. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Pp. 261–278. Springer-Verlag, 2009.