Pointer Life Cycle Types for Lock-Free Data Structures with Memory Reclamation

ROLAND MEYER, TU Braunschweig, Germany
SEBASTIAN WOLFF, TU Braunschweig, Germany

We consider the verification of lock-free data structures that manually manage their memory with the help of a safe memory reclamation (SMR) algorithm. Our first contribution is a type system that checks whether a program properly manages its memory. If the type check succeeds, it is safe to ignore the SMR algorithm and consider the program under garbage collection. Intuitively, our types track the protection of pointers as guaranteed by the SMR algorithm. There are two design decisions. The type system does not track any shape information, which makes it extremely lightweight. Instead, we rely on invariant annotations that postulate a protection by the SMR. To this end, we introduce angels, ghost variables with an angelic semantics. Moreover, the SMR algorithm is not hard-coded but a parameter of the type system definition. To achieve this, we rely on a recent specification language for SMR algorithms. Our second contribution is to automate the type inference and the invariant check. For the type inference, we show a quadratic-time algorithm. For the invariant check, we give a source-to-source translation that links our programs to off-the-shelf verification tools. It compiles away the angelic semantics. This allows us to infer appropriate annotations automatically in a guess-and-check manner. To demonstrate the effectiveness of our type-based verification approach, we check linearizability for various list and set implementations from the literature with both hazard pointers and epoch-based memory reclamation. For many of the examples, this is the first time they are verified automatically. For the ones where there is a competitor, we obtain a speed-up of up to two orders of magnitude.

CCS Concepts: • **Theory of computation → Program verification**; *Type theory*; *Shared memory algorithms*; *Concurrent algorithms*; *Program analysis*; *Invariants*; • **Software and its engineering → Memory management**; **Model checking**; *Automated static analysis.*

Additional Key Words and Phrases: lock-free data structures, safe memory reclamation, garbage collection, linearizability, verification, type systems, type inference

ACM Reference Format:
Roland Meyer and Sebastian Wolff. 2020. Pointer Life Cycle Types for Lock-Free Data Structures with Memory Reclamation. *Proc. ACM Program. Lang.* 4, POPL, Article 68 (January 2020), 36 pages. https://doi.org/10.1145/3371136

1 INTRODUCTION

In the last decade we have experienced an upsurge in massive parallelization being available even in commodity hardware. To keep up with this trend, popular programming languages include in their standard libraries features to make parallelization available to everyone. At the heart of this effort are concurrent (thread-safe) data structures. Consequently, efficient implementations are in high demand. In practice, lock-free data structures are particularly efficient.

Authors' addresses: Roland Meyer, TU Braunschweig, Germany, roland.meyer@tu-bs.de; Sebastian Wolff, TU Braunschweig, Germany, sebastian.wolff@tu-bs.de.

2475-1421/2020/1-ART68
https://doi.org/10.1145/3371136

Unfortunately, lock-free data structures are also particularly hard to get correct. The reason is the absence of traditional synchronization using locks and mutexes in favor of low-level synchronization using hardware instructions. This calls for formal verification of such implementations. In this context, the de-facto standard correctness property is linearizability [Herlihy and Wing 1990]. It requires, intuitively, that each operation of a data structure implementation appears to execute atomically somewhen between its invocation and return. For users of lock-free data structures, linearizability is appealing. It provides the illusion of atomicity—they can use the data structure as if they were using it in a sequential setting.

Proving lock-free data structures linearizable has received a lot of attention (cf. Section 10). Doherty et al. [2004], for instance, give a mechanized proof of a practical lock-free queue. Such proofs require plenty of manual work and take a considerable amount of time. Moreover, they require an understanding of the proof method and the data structure under consideration. To overcome this drawback, we are interested in automated verification. The CAVE tool by Vafeiadis [2010a,b], for example, is able to establish linearizability for singly-linked data structures fully automatically.

The problem with automated verification for lock-free data structures is its limited applicability. Most techniques are restricted to implementations that assume a garbage collector (GC). This assumption, however, does not apply to all programming languages. Take `C/C++` as an example. It does not provide an automatic garbage collector that is running in the background. Instead, it is the programmer's obligation to avoid memory leaks by reclaiming memory that is no longer in use (using `delete`). In lock-free data structures, this task is much harder than it may seem at first glance. The root of the problem is that threads typically traverse the data structure without synchronization. Hence, there may be threads holding pointers to records that have already been removed from the structure. If records are reclaimed immediately after the removal, those threads are in danger of accessing deleted memory. Such accesses are considered unsafe (undefined behavior in `C/C++` [ISO 2011]) and are a common cause for system crashes due to a `segfault`. The solution to this problem are so-called *safe memory reclamation (SMR)* algorithms. Their task is to provide lock-free means for deferring the reclamation/deletion until all unsynchronized threads have finished their accesses. Typically, this is done by replacing explicit deletions with calls to a function `retire` provided by the SMR algorithm which defers the deletion. Coming up with efficient and practical SMR implementations is difficult and an active field of research (cf. Section 10).

The use of SMR algorithms to manage manually the memory of lock-free data structures hinders verification, both manual and automated. This is due to the high complexity of such algorithms. As hinted before, an SMR implementation needs to be lock-free in order not to spoil the lock-free guarantee of the data structure using it. In fact, SMR algorithms are quite similar to lock-free data structures implementation-wise. This added complexity could not be handled by automatic verifiers up until recently. Meyer and Wolff [2019a] were the first to present a practical approach. Their key insight is that the data structure can be verified as if it was relying on a garbage collector rather than an SMR algorithm, provided the data structure does not perform unsafe memory operations. Since data structures from the literature are usually memory safe, the above insight is a powerful tool for verification. Nevertheless, it leaves us with a hard task: establishing that all memory operations are safe in the presence of memory reclamation. Meyer and Wolff [2019a] were not able to conduct this check under GC. Instead, they explore the entire state space of the data structure with SMR, restricting reallocations to a single address, to prove ABAs harmless (a criterion they require for soundness). Unfortunately, their state space exploration does not scale well.

In the present paper we tackle the challenge of proving a lock-free data structure memory safe. We present a type system to address this task. That is, we present a syntax-centric approach to establish the semantic property of memory safety. In particular, we no longer need expensive state space explorations that can handle SMR and memory reuse in order to prove memory safety. This

allows us to utilize the full potential of the above result: if our type check succeeds, we remove the SMR code from the data structure and verify the resulting implementation using an off-the-shelf GC verifier. The idea behind our type system is a life cycle common to lock-free data structures with manual memory management via SMR [Brown 2015]. The life cycle, depicted in Figure 1, has four stages: (i) local, (ii) active, (iii) retired, and (iv) not allocated. Newly allocated records are in the local stage. The record is known only to the allocating thread; it has exclusive read/write access. The goal of the local stage is to prepare records for being published, i.e., added to the shared state of the data structure. When a record is published, it enters the active stage. In this stage, accesses to the record are safe because it is guaranteed to be allocated. However, no thread has exclusive access and thus must fear interference by others. It is worth pointing out that a publication is irreversible. Once a record becomes active it cannot become local again. A thread, even if it removes the active record from the shared structures, must account for other threads that have already acquired a pointer to that record. To avoid memory leaks, removed records eventually become retired. In this stage, threads may still be able to access the record safely. Whether or not they can do so depends on the SMR algorithm used. Finally, the SMR algorithm detects that the retired record is no longer in use and reclaims it. Then, the memory can be reused and the life cycle begins anew.

The main challenge our type system has to address wrt. the above memory life cycle is the transition from the active to the retired stage. Due to the lack of synchronization, this can happen without a thread noticing. Programmers are aware of the problem. They protect records while they are active such that the SMR guarantees safe access even though the record is retired. To cope with this, our types integrate knowledge about the SMR algorithm. A core aspect of our development is that the actual SMR algorithm is an input to our type system—it is not tailored towards a specific SMR algorithm.

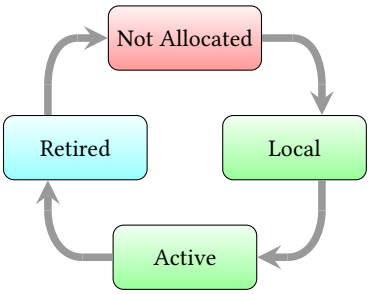


Fig. 1. Memory life cycle of records in lock-free data structures using SMR.

An additional challenge arises from the type system performing a thread-local analysis, it considers the program code as if it was sequential. This means the type system is not aware of the actual interference among threads, unlike state space explorations. To address this, we use types that are stable under the actions of interfering threads [Owicki and Gries 1976].

In practice, protecting a record while it is active is non-trivial. Between acquiring a pointer to the record and the subsequent SMR protection call, an interferer may retire the record, in which case the protection has no effect. SMR algorithms usually offer no means to check whether a protection was successful. Instead, programmers exploit intricate data structure invariants to perform this check. A common such invariant, for instance, is *all shared reachable records are active*. A type system typically cannot detect such data structure shape invariants. We turn this weakness into a strength. We deliberately do not track shape invariants nor alias information. Instead, we use simple annotations to mark pointers that point to active records. To relieve the programmer from arguing about their correctness, we show how to discharge annotations automatically. Interestingly, this can be done with off-the-shelf GC verifiers. It is worth pointing out that the ability to automatically discharge invariants allows for an automated guess-and-check approach for placing invariant annotations.

To increase the applicability of our type system, we use the theory of movers [Lipton 1975] as an enabling technique. Movers are a standard approach to transform a program into a *more atomic* version while retaining its behavior. That the resulting program is more atomic is beneficial for verification. The transformations are practical: Elmas et al. [2009], for example, automate them.

To demonstrate the usefulness of our approach, we implemented a linearizability checker which realizes the techniques presented in this paper. That is, our tool (i) performs a type inference to

establish memory safety relying on invariant annotations, (ii) discharges the annotations under GC using CAVE as a back-end, and (iii) verifies linearizability under GC using CAVE. Additionally, we implemented a prototype for automatically inserting annotations and applying movers. These program transformations are performed on demand, guided by a failed type inference. Our tool is able to establish linearizability for lock-free data structures from the literature, like Michael&Scott's lock-free queue [Michael and Scott 1996], the Vechev&Yahav CAS set [Vechev and Yahav 2008], the Vechev&Yahav DCAS set [Vechev and Yahav 2008], and the ORVYY set [O'Hearn et al. 2010], for the well-known hazard pointer method [Michael 2002b] as well as epoch-base reclamation [Fraser 2004]. We stress that our approach is not limited to CAVE as a back-end but can use any verifier for garbage collection. To the best of our knowledge, we are the first to automatically verify lock-free set implementations that use SMR.

The outline of our paper is as follows: §2 illustrates our contribution, §3 introduces the programming model, §4 discusses preliminary results, §5 presents our type system for proving lock-free data structures memory safe wrt. a user-specified SMR algorithm, §6 gives a comprehensive example of our approach, §7 presents an efficient type inference algorithm, §8 presents an instrumentation of the data structure under scrutiny to discharge invariant annotations fully automatically with the help of a GC verifier, §9 evaluates our approach on well-known lock-free data structures from the literature, and §10 discusses related work. This paper comes with a companion technical report [Meyer and Wolff 2019b] containing missing details.

2 THE CONTRIBUTION ON AN EXAMPLE

We illustrate our approach on Micheal&Scott's lock-free queue [Michael and Scott 1996], Figure 2, which is used, for example, as Java's `ConcurrentLinkedQueue` and as C++ Boost's `lockfree::queue`. The queue is organized as a `NULL`-terminated singly-linked list of nodes. The `enqueue` operation appends new nodes to the end of the list. To do so, an enqueuer first moves `Tail` to the last node as it may lack behind. Then, the new node is appended by pointing `Tail->next` to it. Last, the enqueuer tries to move `Tail` to the end of the list. This can fail as another thread may already have moved `Tail` to avoid waiting for the enqueuer. The `dequeue` operation removes the first node from the list. Since the first node is a dummy node, `dequeue` reads out the data value of the second node in the list and then moves the `Head` to that node. Additionally, `dequeue` maintains the property that `Head` does not overtake `Tail`. This is done by moving `Tail` towards the end of the list if necessary. (There is an optimized version due to Doherty et al. [2004] which avoids this step.) Note that updates to the shared list of nodes are performed exclusively with single-word atomic compare-and-swap (CAS).

So far, the discussed implementation assumes a garbage collector. The nodes allocated by `enqueue` are not reclaimed explicitly after being removed from the shared list by `dequeue`: the queue leaks memory. Unfortunately, there is no simple solution to this problem. Uncommenting the explicit deletion from Line 48 avoids the leak. However, it leads to use-after-free bugs. Due to the lack of synchronization, threads may still hold and dereference pointers to the now deleted node. A dereference of such a dangling pointer, however, is unsafe. In `C/C++`, for example, dereferencing a dangling pointer has *undefined behavior* [ISO 2011] and may make the system crash with a `segfault`.

To solve the problem, programmers employ safe memory reclamation (SMR) algorithms. Two well-known examples are epoch-based reclamation (EBR) [Fraser 2004] and hazard pointers (HP) Michael [2002b]. They offer a function `retire` that replaces the ordinary `delete`. The difference is that `retire` does not immediately delete nodes. Instead, it defers the deletion until it is safe. In order to discover whether a deletion is safe, threads need to declare which nodes they access. How this is done depends on the SMR algorithm.

Epoch-based reclamation offers two additional functions `leaveQ` and `enterQ`. Threads use the former to announce that they are going to access the data structure and use the latter to announce

```
1  struct Node { data_t data; Node* next; };
2  shared Node* Head, Tail;
3  atomic init() {
4    Head = Tail = new Node();
5    Head->next = NULL;
6  }
7  void enqueue(data_t input) {
8  E leaveQ();
9    Node* node = new Node();
10   node->data = input;
11   node->next = NULL;
12   while (true) {
13     Node* tail = Tail;
14 H   protect(tail, 0);
15 H   if (tail != Tail) continue;
16     Node* next = tail->next;
17     if (tail != Tail) continue;
18     if (next != NULL) {
19       CAS(&Tail, tail, next);
20       continue;
21     }
22     if (CAS(&tail->next, next, node)) {
23       CAS(&Tail, tail, node);
24       break;
25   } }
26 E enterQ();
27 }
28 data_t dequeue() {
29 E leaveQ();
30   while (true) {
31     Node* head = Head;
32 H   protect(head, 0);
33 H   if (head != Head) continue;
34     Node* tail = Tail;
35     Node* next = head->next;
36 H   protect(next, 1);
37     if (head != Head) continue;
38     if (next == NULL) {
39 E     enterQ();
40       return EMPTY;
41     }
42     if (head == tail) {
43       CAS(&Tail, tail, next);
44       continue;
45     } else {
46       data_t output = next->data;
47       if (CAS(&Head, head, next)) {
48         // delete head;
49 HE      retire(head);
50 E       enterQ();
51         return output;
52 } } } }
```

Fig. 2. Michael&Scott's lock-free queue [Michael and Scott 1996] with two different safe memory reclamation techniques: epoch-based reclamation (EBR) [Fraser 2004] and hazard pointers (HP) [Michael 2002b]. The modifications needed to use EBR (HP) are marked with E (H). For HP, we assume two hazard pointers per thread.

that they have finished the access. The function names, in particular the Q, refer to the fact that the threads are *quiescent* [McKenney and Slingwine 1998] between enterQ and leaveQ, meaning they do not modify the data structure. During the non-quiescent period, EBR guarantees that the shared reachable nodes are not reclaimed, even if they are removed from the data structure and retired. To use EBR, the programmer simply replaces delete statements with calls to retire and adds calls to leaveQ (enterQ) at the beginning (end) of data structure operations. Consider Figure 2 for an example; the lines marked by E are the modifications required to use EBR. While easy to use, EBR implementations usually stop reclaiming memory altogether upon thread failure. Hazard pointers do not suffer from this problem.

The hazard pointer method requires threads to declare which nodes they access in a per-node fashion. To that end, HP offers an additional function: protect. It signals that a deletion of the received node should be deferred. To be precise, HP guarantees that the deletion of a node is deferred if it has been continuously protected since before it was retired [Gotsman et al. 2013]. While this method is conceptually simple, it is non-trivial to apply.

To use hazard pointers with Michael&Scott's queue requires to add the code marked by H in Figure 2. As for EBR, delete statements are replaced with retire. Moreover, pointers that are accessed need protection to defer their deletion. Simply calling protect is usually insufficient as the protect may be too late. A common pattern for protecting pointers is to first protect them

and then check that they have not been retired since. In Michael&Scott's queue this is done by testing whether the protected nodes are still shared reachable—the queue maintains the invariant that nodes reachable from the shared pointers are never retired. To make this precise, consider Lines 31 to 33. Line 31 reads in `head` from the shared pointer `Head`. The dequeue operation will access (dereference) `head`. Hence, it has to make sure that the referenced node remains allocated. To do so, a protection of `head` is issued in Line 32. However, the node pointed to by `head` may have been dequeue and retired since `head` was read. To ensure that the protection is successful, that is, not too late, Line 33 restarts the dequeue operation in case `head` no longer coincides with `Head`. The remaining protections in the code follow the same principle.

Our contribution is a method for verifying lock-free data structures which use an SMR algorithm, like Michael&Scott's queue with EBR/HP from Figure 2. At the heart of our method lies a type system which proves safe all pointer operations in the data structure. In the case of hazard pointers, for instance, this requires to prove all pointer accesses appropriately protected. Once this property is established, we show that the actual verification does not need to consider the SMR algorithm: it suffices to verify the data structure under garbage collection; the SMR function invocations can be removed altogether. This allows the use of off-the-shelf GC verifiers.

2.1 A Type System to Simplify Verification

Our main contribution is a type system a successful type check of which proves a given program free from unsafe memory operations. The type assigned to a pointer specifies if it is safe to access that pointer. The types are influenced by both the memory life cycle from Section 1 and the SMR algorithm used. In the case of hazard pointers, a pointer may be protected and thus guaranteed not to be deleted. Hence, the protected pointer can be accessed without precautions. For an unprotected pointer, on the other hand, threads may need to take additional steps to guarantee that the pointer is not dangling, for instance, by establishing that it (to be precise, its address) is in the active stage.

We illustrate our type system on the `dequeue` operation of Michael&Scott's queue. The interesting part is the typing of the local pointers `head` and `next` in Lines 31 to 37. The type derivation is depicted in Figure 3. Let us assume for the moment that the shared pointers and the nodes reachable through them are in the active stage. We denote this by $\mathit{shared}:\mathbb{A}$. It is the only type binding at the beginning of the operation. The first assignment, Line 31, adds a type binding for `head` to the type environment. The type for `head` is copied from the source pointer, `Head`. However, we remove $\mathbb{A}$ immediately after the assignment so that the actual type of `head` is $\varnothing$. The reason for this are interfering threads: as discussed above, an interferer can dequeue and retire the node pointed to by `head`. As a consequence, we cannot guarantee that `head` is active; we indeed need to remove $\mathbb{A}$. Next, Line 32 protects `head`. We set the type of `head` to $\mathbb{E}_{isu}$, encoding that a protection has been issued. Remembering that `head` is protected is crucial for the subsequent conditional. Line 33 tests whether `Head` has changed since it was read into `head`. If it has not, denoted by `assume(head == Head)` in Figure 3, we join the type of `head` with the type of `Head`. That is, `head` receives $\mathbb{A}$. Now, we know that the protection has been issued before the node pointed to by `head` has been retired. So the hazard pointer method guarantees that the node is not deleted. The subsequent code can access `head` without precautions. We incorporate this fact into the type of `head` by updating it to $\mathbb{S}$, indicating that accesses are safe. (We skip the assignment to `tail`

```
   { shared:𝔸 }
(31)  Node* head = Head;
   { shared:𝔸, head:∅ }
(32)  protect(head, 0);
   { shared:𝔸, head:𝔼isu }
(33)  assume(head == Head);
   { shared:𝔸, head:𝔼isu ∧ 𝔸 }
   { shared:𝔸, head:𝕊 }
(35)  Node* next = head->next;
   { shared:𝔸, head:𝕊, next:∅ }
(36)  protect(next, 1);
   { shared:𝔸, head:𝕊, next:𝔼isu }
(37)  assume(head == Head);
   { shared:𝔸, head:𝕊, next:𝕊 }
```

Fig. 3. Idealized typing for the non-retrying branch of Lines 31 to 37.

from Line 34, it does not affect the type check.) Next, Line 35 dereferences `head`. This dereference is safe since `head` has type $\mathbb{S}$, it is guaranteed to be allocated. The type assigned to `next` is $\varnothing$ because we do not assign types to pointers within nodes, like `head->next`. Hence, `next` cannot obtain any guarantees from the assignment. Line 36 then protects `next`. Similarly to the above, we set its type to $\mathbb{E}_{isu}$. The following conditional, Line 37, tests again if `Head` has changed since the beginning of the operation. Consider the case it has not. If we remember that `next` is the successor of `head`, we know that `next` references a shared reachable node. Hence, we can assign type $\mathbb{A}$ to `next`. As in the case for `head`, this allows us to lift the type to $\mathbb{S}$. That is, using `next` in the following code becomes safe due to the conditional guaranteeing its activeness. The remainder of the type check is then straightforward since only protected and/or shared pointers are used.

We stress that the actual SMR algorithm is a parameter to our type system—it is not limited to analyzing programs using hazard pointers.

2.2 Data Structure Invariants in the Type System

The type check as illustrated in Section 2.1 is idealized. We assumed that we maintain type $\mathbb{A}$ for shared pointers and the nodes reachable through them. Moreover, we assumed that `next` remains the successor of `head` during an execution of `dequeue`. Such invariants of the data structure are notoriously hard to derive. Typically, it requires a state-space exploration of all thread interleavings to find invariants of lock-free data structures. A major challenge in exploring the state space is the need for an effective (symbolic) way of tracking the data structure shape [Abdulla et al. 2013; Brookes 2004; O'Hearn 2004; O'Hearn et al. 2001; Reynolds 2002].

We tackle the above problem as follows: we do not track the data structure shape at all, not even pointer aliases. Instead, we require the programmer to annotate which pointers/nodes are active. This allows the type check to rely on data structure invariants which typically cannot be found by a type system. To free the programmer from manually proving the correctness of such annotations, we automate the correctness check. We give an instrumentation of the program under scrutiny such that an ordinary GC verifier can discharge the invariants. A thing to note is that the simple nature of active annotations and the ability to automatically discharge them makes it possible to find appropriate annotations fully automatically (guided by a failed type check).

Revisiting the previous example, the type environments never contain $\mathit{shared} : \mathbb{A}$. To arrive at type $\mathbb{S}$ for `head` in Line 33 nevertheless, we annotate the `assume(head == Head)` statement with an invariant stating that `head` is active. Then, the type derivation for Line 33 remains the same as before. We argue that, provided the queue implementation is memory safe, there must be a code location between the protection in Line 32 and the subsequent dereference in Line 35 where an active annotation can be placed. To see this, assume there is no such code location. This means `head` is not active in Lines 32 to 35. That is, it must have been retired before the protection in Line 32, rendering the protect unsuccessful. Hence, the dereference in Line 35 is unsafe, contradicting our assumption of memory safety. For pointer `next`, we proceed similarly and add an active annotation to the second assumption (Line 37).

With the above annotations our type system can rely on aspects of the dynamic behavior without requiring the programmer to manually take over parts of the verification. We believe that having annotations makes the type system more versatile (compared to having none) in the sense that data structures need not satisfy implicit invariants like *all shared pointers and nodes are active*. Moreover, relying on annotations rather than shape invariants allows for a much simpler type system.

2.3 Supporting Different SMR Algorithms

The above illustration focuses on hazard pointers. The actual type system we develop in Section 5 does not—it is not tailored towards a specific SMR algorithm. To achieve this degree of freedom, our

type system takes as a parameter a formal description of the SMR algorithm being used. We rely on a recent specification language for SMR algorithms [Meyer and Wolff 2019a]: SMR automata. Then, our types capture the locations of the SMR automaton that can be reached after having seen a sequence of SMR calls in the program (control-flow sensitive type system). This allows the types to track the relevant sequences of SMR calls. Moreover, it allows them to detect when the deletion of a node is guaranteed to be deferred in order to infer type $\mathbb{S}$.

3 PROGRAMMING MODEL

We introduce concurrent shared-memory programs that employ a library for safe memory reclamation (SMR) and are annotated by invariants. A programming construct that is new to our model are angels, ghost variables with an angelic semantics. Angels are second-order pointers holding sets of addresses. When typing (cf. Section 5), angels will help us track the protected nodes.

3.1 Programs

We define a core language for concurrent shared-memory programs. Invocations to a library for safe memory reclamation and invariant annotations will be added below. Programs P are comprised of statements defined by

$$\begin{aligned} \mathit{stmt} &::= \mathit{stmt}; \mathit{stmt} \mid \mathit{stmt} \oplus \mathit{stmt} \mid \mathit{stmt}^* \mid \mathit{com} \\ \mathit{com} &::= p := q \mid p := q.\texttt{next} \mid p.\texttt{next} := q \mid u := q.\texttt{data} \mid p.\texttt{data} := u \mid u := \texttt{op}(\bar{u}) \\ &\quad \mid p := \texttt{malloc} \mid \texttt{assume}\ \mathit{cond} \mid \texttt{beginAtomic} \mid \texttt{endAtomic} \\ \mathit{cond} &::= p = q \mid p \neq q \mid \texttt{pred}(\bar{u})\ . \end{aligned}$$

We assume a strict typing that distinguishes between data variables $u, u' \in \mathit{DVar}$ and pointer variables $p, q \in \mathit{PVar}$. Notation $\bar{u}$ is short for $u_1, \ldots, u_n$. The language includes sequential composition, non-deterministic choice, and Kleene iteration. The primitive commands include assignments, memory accesses, memory allocations, assumptions, and atomic blocks. They have the usual meaning. We make the semantics of commands precise in a moment.

Memory. Programs operate over addresses from Adr that are assigned to pointer expressions PExp. A pointer expression is either a pointer variable from PVar or a pointer selector $a.\texttt{next} \in \mathit{PSel}$. The set of shared pointer variables accessible by every thread is $\mathit{shared} \subseteq \mathit{PVar}$. Additionally, we allow pointer expressions to hold the special value $\texttt{seg} \notin \mathit{Adr}$ denoting undefined/uninitialized pointers. There is also an underlying data domain Dom to which data expressions $\mathit{DExp} = \mathit{DVar} \uplus \mathit{DSel}$ with $a.\texttt{data} \in \mathit{DSel}$ evaluate. A generalization of our development to further selectors is straightforward.

The memory is a partial function $m : \mathit{PExp} \uplus \mathit{DExp} \rightharpoonup \mathit{Adr} \uplus \{\texttt{seg}\} \uplus \mathit{Dom}$ that respects the typing. The initial memory is m_ϵ. Pointer variables p are uninitialized, $m_\epsilon(p) = \texttt{seg}$. Data variables u have a default value, $m_\epsilon(u) = 0$. We modify the memory with updates up of the form $e \mapsto v$. Applied to a memory m, the result is the memory $m' = m[e \mapsto v]$ defined by $m'(e) = v$ and $m'(e') = m(e')$ for all $e' \neq e$. Below, we define computations τ which give rise to sequences of updates. We write m_τ for the memory resulting from the initial memory m_ϵ when applying the sequence of updates in τ.

Liberal Semantics. We define a semantics where program P is executed by a possibly unbounded number of threads. In this semantics some addresses may be freed non-deterministically by the runtime environment. This behavior will be constrained by a memory reclamation algorithm in a moment. Formally, the liberal semantics of program P is the set of computations $[\![P]\!]^Y_X$. It is defined relative to two sets $Y \subseteq X \subseteq \mathit{Adr}$ of addresses allowed to be reallocated and freed, respectively. A computation is a sequence τ of actions of the form $\mathit{act} = (t, \mathit{com}, up)$. The action indicates that thread t executes command com that results in the memory update up. The definition of the liberal

semantics is by induction. The empty computation is always contained, $\epsilon \in [\![P]\!]^Y_X$. Then, action act can be appended to computation τ, denoted $\tau.act \in [\![P]\!]^Y_X$, if $\tau \in [\![P]\!]^Y_X$, act respects the control flow of P, and one of the following holds.

(Assign) If $act = (t, p.\texttt{next} := q, a.\texttt{next} \mapsto b)$ then $m_\tau(p) = a$ and $m_\tau(q) = b$. There are similar conditions for the remaining assignments.

(Assume) If $act = (t, \texttt{assume}\ lhs = rhs, \emptyset)$ then $m_\tau(lhs) = m_\tau(rhs)$. There are similar conditions for the remaining assumptions.

(Malloc) If $act = (t, p := \texttt{malloc},\ up)$, then up has the form $p \mapsto a, a.\texttt{next} \mapsto \texttt{seg}, a.\texttt{data} \mapsto d$ so that $a \in \mathit{fresh}(\tau)$ or $a \in \mathit{freed}(\tau) \cap Y$, and $d \in \mathit{Dom}$.

(Free) If $act = (\bot, \texttt{free}(a), \emptyset)$ then $a \in X$.

(Atomic) If $act = (t, \texttt{beginAtomic}, \emptyset)$ or $act = (t, \texttt{endAtomic}, \emptyset)$.

Note that Rule (Free) may spontaneously emit $\texttt{free}(a)$, although there is no `free` command in the programming language. Indeed, the `free` command will be issued by the memory reclamation algorithm defined in the next section (it is not part of P). The rule allows us to define the set of allocatable addresses for rule (Malloc) as addresses that have never been allocated in the computation, denoted by $\mathit{fresh}(\tau)$, and addresses which have been freed since their last allocation, $\mathit{freed}(\tau)$.

3.2 Safe Memory Reclamation

We consider programs that manage their memory with the help of a safe memory reclamation (SMR) algorithm. In this setting, threads do not free their memory themselves (no explicit `free` command), but request the SMR algorithm to do so. The SMR algorithm will have means of understanding whether an address is still accessed by other threads, and only execute the `free` when it is safe to do so. As a consequence, the semantics of the program depends on the SMR algorithm it invokes.

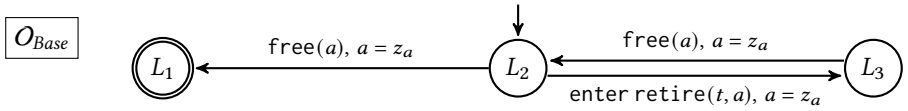
The means of detecting whether an address can be freed safely depend on the SMR algorithm. Despite the variety of techniques, it was recently observed that the behavior of major SMR algorithms can be captured by a common specification language [Meyer and Wolff 2019a]: SMR automata.[1] Intuitively, the SMR automaton models the protection protocol of its SMR algorithm, while abstracting from implementation details. We recall SMR automata and use them to restrict the liberal semantics to the frees performed by the SMR algorithm.

SMR Automata. An SMR algorithm offers a set of functions $f(\bar{r})$ for the programmer to provide information about the intended access to the data structure, like `leaveQ`, `enterQ`, and `retire` in the case of EBR (cf. Section 2). An SMR automaton, as depicted in Figure 4, is a finite control structure the transitions of which are labeled with these function symbols. Additionally, each transition comes with a guard. The guard influences the flow of control in the SMR automaton based on the actual parameters of function calls. To distinguish the parameters, the automaton maintains a finite set of local variables storing thread identifiers and addresses. Guards may then compare the actual parameters with the values of variables.

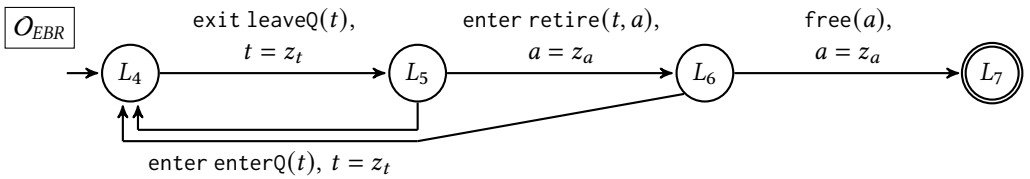
What makes SMR automata a useful modeling language is their compactness: complex SMR algorithms can be captured by fairly small SMR automata. This is achieved by an interesting definition of the semantics. SMR automata accept bad behavior, `free` commands that should not be executed after a sequence of SMR function calls protecting the address.

What makes SMR automata interesting for automated verification are two technical restrictions that limit their expressiveness. First, the variable values are chosen only once, in the beginning of the computation, and never changed. This choice is non-deterministic. The idea is that the automaton picks some protection to track. Second, transition guards can only compare for equality.

[1] Working on compositional verification, Meyer and Wolff [2019a] call them *observers*.



(a) SMR automaton specifying that address z_a may be freed only if it has been retired and not freed since. The automaton uses one variable z_a.



(b) SMR automaton characterizing when EBR defers frees, using two variables z_t and z_a. It states that address z_a must not be freed if it was retired while z_t is in-between leaveQ and enterQ calls.

Fig. 4. Epoch-based reclamation (EBR) is specified by the SMR automaton $O_{Base} \times O_{EBR}$. For legibility, we omit self loops on all locations for the events that are not given.

That this is sufficient to properly model the behavior of SMR algorithms can be explained by the fact that SMR algorithms are designed to work with very different data structures, from stacks to queues to trees. Hence, there is no point for the SMR algorithm to store information about the data structure more specific than the equality of pointers.

Syntactically, an SMR automaton O is a tuple consisting of a finite set of locations, a finite set of variables, and a finite set of transitions. There is a dedicated initial location and a number of accepting locations. Transitions are of the form $l \xrightarrow{f(\bar{r}),\, g} l'$ with locations l, l', event $f(\bar{r})$, and guard g. Events $f(\bar{r})$ consist of a type f and parameters $\bar{r} = r_1, \ldots, r_n$. The guard is a Boolean formula over equalities of variables and parameters $\bar{r}$.

Semantically, a (runtime) state s of the SMR automaton is a tuple (l, φ) where l is a location and φ maps variables to values. Such a state is initial if l is initial, and similarly accepting if l is accepting. Then, $(l, \varphi) \xrightarrow{f(\bar{v})} (l', \varphi)$ is an SMR step if $l \xrightarrow{f(\bar{r}),\, g} l'$ is a transition and $\varphi(g[\bar{r} \mapsto \bar{v}])$ evaluates to *true*. By $\varphi(g[\bar{r} \mapsto \bar{v}])$ we mean g with the variables replaced by their φ-mapped values and the formal parameters $\bar{r}$ replaced by the actual values $\bar{v}$. As mentioned before, the valuation φ is chosen non-deterministically in the beginning; it is not changed by steps. A *history* $h = f_1(\bar{v}_1) \ldots f_n(\bar{v}_n)$ is a sequence of events. If there are SMR steps $s \xrightarrow{f_1(\bar{v}_1)} \cdots \xrightarrow{f_n(\bar{v}_n)} s'$, we write $s \xrightarrow{h} s'$. If s' is accepting, we say that h is accepted by s.

Acceptance in SMR automata characterizes *bad behavior*, and a history h is said to violate O if there is an initial state s and an accepting state s' such that $s \xrightarrow{h} s'$. The specification of O is the set of histories that are not accepted:

$$\mathcal{S}(O) := \{h \mid \forall s, s'.\ s \xrightarrow{h} s' \wedge s \text{ initial } \implies s' \text{ not accepting}\} .$$

We also use a restriction of the specification. The set $\mathcal{F}_O(h, a)$ contains those continuations h' of h so that $h.h' \in \mathcal{S}(O)$ and moreover at most address a is freed in h'. As bad behavior means executing a forbidden free, we assume accepting states can only be reached by transitions labeled with free and cannot be left.

To give an example, consider the SMR automaton $O_{Base} \times O_{EBR}$ from Figure 4. It formalizes the informal specification of EBR from Section 2. Automaton O_{Base}, Figure 4a, forbids an EBR implementation to free addresses that have not yet been retired or have not been retired since their

last free. Put differently, it forbids spurious frees and double-frees. Automaton O_{EBR}, Figure 4b, requires the EBR implementation to defer the `free` of retired nodes which could still be accessed by some thread. A thread can still access the retired node if it has acquired a pointer to the node before it was retired (following the usage policy of EBR discussed in Section 2). This is the case if the thread started accessing the data structure before the `retire`, which it announces via a call to `leaveQ`.

While every SMR implementation has its own SMR automaton, the practically relevant SMR automata are products[2] of O_{Base} with further SMR automata [Meyer and Wolff 2019a], like for EBR in the above example. Our development relies on this.

We also assume that the SMR automaton has two distinguished variables z_t and z_a. Intuitively, variable z_t will store the thread for which the SMR automaton tracks the protection of the address stored in z_a. All SMR algorithms we know can be specified with only two variables. A possible explanation is that SMR algorithms do not seem to use helping [Herlihy and Shavit 2008] to protect pointers. We are not aware of an SMR algorithm where the protection of an address would be inferred from communication with another address or, more ambitiously, another thread.

Moreover, we inherit from [Meyer and Wolff 2019a] the natural requirement that SMR algorithms do not remember addresses that have been freed in order to detect (and react to) reallocations. Formally, an SMR automaton *supports elision* if for all histories h the behavior on address a after h (i) is not affected by a free of another address b, $\mathcal{F}_O(h.\texttt{free}(b), a) = \mathcal{F}_O(h, a)$, (ii) is not affected by renaming another two addresses b and c, $\mathcal{F}_O(h, a) = \mathcal{F}_O(h[b/c], a)$, (iii) is included in the behavior on a after another history h' provided a is fresh after h', $\mathcal{F}_O(h, a) \subseteq \mathcal{F}_O(h', a)$, and (iv) contains the behavior on a after $h.\texttt{free}(a)$, $\mathcal{F}_O(h.\texttt{free}(a), a) \subseteq \mathcal{F}_O(h, a)$. To understand (iv), note that the task of the SMR algorithm is to protect addresses from being freed. Hence it is safe to delay frees.

For convenience, we summarize our assumptions on SMR automata. All SMR automata we encountered, including the ones from [Meyer and Wolff 2019a], satisfy them.

Assumption 1. *SMR automata (i) reach accepting states only with* `free` *and do not leave them, (ii) are products with O_{Base}, (iii) have distinguished variables z_t and z_a, and (iv) support elision.*

It will be convenient to have a post-image $post_{p,com}(L)$ on the locations of SMR automata. The post-image yields a set of locations L' reachable by taking a *com*-labeled transition from L. The considered transition is restricted in two ways. First, its guard g must allow z_t to track thread t executing *com*. Second, if p appears as a parameter in *com*, then guard g must allow z_a to track p. Formally, these requirements translate to the satisfiability of $g \wedge t = z_t$ and $g \wedge p = z_a$, respectively. The parameterization in p makes the post-image precise. For an example, consider O_{Base} and the command com = `enter retire`(p). We expect the post-image of L_2 wrt. *com* and p to be $post_{p,com}(L_2) = \{L_3\}$. The address has definitely been retired. Without the parametrization in p, we would get $\{L_2, L_3\}$. The transition could choose not to track p.

SMR Semantics. To incorporate SMR automata into our programming model, we generalize the set of program commands *com* to include calls to and returns from SMR functions:

$$com \;::=\; com \mid \texttt{enter}\ func(\bar{p}, \bar{u}) \mid \texttt{exit}\ func\ .$$

We add corresponding actions to the liberal semantics $[\![P]\!]^Y_X$. They make visible the function call/return but do not lead to memory updates.

(Enter) $act = (t, \texttt{enter}\ func(\bar{p}, \bar{u}), \emptyset)$. **(Exit)** $act = (t, \texttt{exit}\ func, \emptyset)$.

To use SMR automata in the context of computations, we convert a computation τ into a history h by projecting τ to the `enter`, `exit`, and `free` commands and replacing the formal parameters with the

[2]The product operation on SMR automata is defined as expected and leads to an intersection of the specifications.

actual values. To be precise, we use as events the function names offered by the SMR algorithm plus `free`. The parameters to an event are the values of the actual parameters as well as the executing thread. In the case of `exit` events, we drop the actual parameters and in case of `free` events we drop the executing thread. For example, $\mathcal{H}(\tau.(t, \texttt{enter}\ \mathit{func}(p), \varnothing)) = \mathcal{H}(\tau).\mathit{func}(t, m_\tau(p))$.

The SMR semantics of a program is the restriction of the liberal semantics to the specification of the SMR automaton of interest. More precisely, given an SMR automaton O and sets $Y \subseteq X \subseteq \mathit{Adr}$ of reallocatable and freeable addresses, the SMR semantics induced by O, X, Y of program P is

$$O[\![P]\!]_X^Y \;:=\; \{\tau \mid \tau \in [\![P]\!]_X^Y \wedge \mathcal{H}(\tau) \in \mathcal{S}(O)\}\ .$$

SMR algorithms only restrict the execution of free commands, their functions can always be invoked by the program. SMR automata mimic this by including in their specification all histories that do not respect the control flow. In particular, we have the following property. In the absence of frees, the SMR automaton does not play a role. The resulting semantics, $[\![P]\!]_\varnothing^\varnothing$, is garbage collection (GC).

Lemma 3.1. *$O[\![P]\!]_\varnothing^\varnothing = [\![P]\!]_\varnothing^\varnothing$ for every SMR automaton O.*

To see the lemma, note that only accepting states in O may rule out computations from $[\![P]\!]_\varnothing^\varnothing$. By Assumption 1, only events $\texttt{free}(a)$ may lead to such accepting states.

Reconsider the SMR automaton O_{Base}. For this automaton to properly restrict the frees in a program, the program should not perform *double retires*, that is, not retire an address again before it is freed. The point is that SMR algorithms typically misbehave after a double retire (perform double frees), which is not reflected in O_{Base} (it does not allow for double frees after a double retire). Our type system will establish the absence of double retires for a given program.

3.3 Angels

Angels can be understood as ghost variables with an angelic semantics. Like for ghosts, their purpose is verification: angels store information about the computation that can be used in invariants but that cannot be used to influence the control flow. This information is a set of addresses, which means angels are second-order pointers. The set of addresses is determined by an angelic choice, a non-deterministic assignment that is beneficial for the future of the computation.

The idea behind angels is the following. When typing, some invariants of the runtime behavior may not be deducible by the type system. Angels allow the programmer to make them explicit in the program and thus available to the type check. Consider EBR's `leaveQ` function. It guarantees that all currently active addresses remain allocated, i.e., will not be reclaimed even if they are retired. An angelic choice is convenient for selecting the set. Subsequent dereferences can then use invariant annotations to ensure that the dereferenced pointer holds an address in the set captured by the angel. With this, our type system is able to detect that the access is safe.

To incorporate angels and invariant annotations into our programming model, we generalize the set of commands as follows

$$\mathit{com} \;::=\; \mathit{com} \mid \texttt{@inv angel}\ r \mid \texttt{@inv}\ p = q \mid \texttt{@inv}\ p\ \texttt{in}\ r \mid \texttt{@inv active}(p) \mid \texttt{@inv active}(r)\ .$$

Angels are local variables r from the set AVar. Invariant annotations include allocations of angels with the keyword `angel` r. Intuitively, this will map the angel to a set of addresses. Conditionals behave as expected. The membership assertion p `in` r checks that the address of p is included in the set of addresses held by the angel r. The predicate $\texttt{active}(p)$ expresses that the address pointed to by p currently is neither freed nor retired, and similar for $\texttt{active}(r)$. We use x to uniformly refer to pointers p and angels r.

In the liberal semantics $[\![P]\!]_X^Y$, the above commands do not lead to memory updates:

(Invariant) $\mathit{act} = (t, \texttt{@inv}\ \bullet\ , \varnothing)$.

$$
\begin{aligned}
inv(\tau) &:= inv_{\epsilon}(\tau) \\
inv_{\sigma}(\epsilon) &:= true \\
inv_{\sigma}(act.\tau) &:= \exists r.\, inv_{\sigma.act}(\tau) && \text{if } act = (t, \texttt{@inv angel } r, \emptyset) \\
inv_{\sigma}(act.\tau) &:= m_{\sigma}(cond) \wedge inv_{\sigma.act}(\tau) && \text{if } act = (t, \texttt{@inv } cond, \emptyset) \\
inv_{\sigma}(act.\tau) &:= m_{\sigma}(p) \in r \wedge inv_{\sigma.act}(\tau) && \text{if } act = (t, \texttt{@inv } p \texttt{ in } r, \emptyset) \\
inv_{\sigma}(act.\tau) &:= m_{\sigma}(p) \in active(\sigma) \wedge inv_{\sigma.act}(\tau) && \text{if } act = (t, \texttt{@inv active}(p), \emptyset) \\
inv_{\sigma}(act.\tau) &:= r \subseteq active(\sigma) \wedge inv_{\sigma.act}(\tau) && \text{if } act = (t, \texttt{@inv active}(r), \emptyset) \\
inv_{\sigma}(act.\tau) &:= inv_{\sigma.act}(\tau) && \text{otherwise.}
\end{aligned}
$$

Fig. 5. Formula capturing the correctness of invariant annotations in a computation τ.

Invariant annotations behave like assertions, they do not influence the semantics but it has to be verified that they hold for all computations. To make precise what it means for invariant annotations to hold for a computation τ, we construct a formula $inv(\tau)$. The invariant annotations are defined to hold for τ iff $inv(\tau)$ is valid. The construction of the formula is given in Figure 5. There, $active(\sigma)$ is the set of addresses that are neither freed nor retired after computation σ. We only consider programs leading to closed formulas, meaning every angel is allocated (and hence quantified) before it is used. The semantics of the formula is as expected: angels evaluate to sets of addresses, equality of addresses is the identity, and membership is as usual for sets. Section 8 shows how to automatically prove the correctness of invariant annotations for all computations.

4 GETTING RID OF MEMORY RECLAMATION

Despite the compact formulation of SMR algorithms as SMR automata, analyzing programs in the presence of memory reclamation remains difficult. Unlike for programs running under garbage collection, ownership guarantees [Bornat et al. 2005; Boyland 2003] and the resulting thread-local reasoning techniques [Brookes 2004; O'Hearn 2004; O'Hearn et al. 2001; Reynolds 2002] do not apply. Meyer and Wolff [2019a] bridge this gap. They show that it is sound and complete to conduct the verification under garbage collection provided the program properly manages its memory. So one can establish this requirement and then perform the actual verification under the simpler semantics. Their statement is as follows; we give the missing definitions in a moment.

Theorem 4.1 (Consequence of Theorem 5.20 in [Meyer and Wolff 2019a]). *If the semantics* $O[\![P]\!]^{\emptyset}_{Adr}$ *is pointer-race-free, then* $O[\![P]\!]^{Adr}_{Adr}$ *corresponds to* $[\![P]\!]^{\emptyset}_{\emptyset}$.

With the above theorem, the only property to be checked in the presence of memory reclamation is the premise of pointer race freedom. However, Meyer and Wolff [2019a] report on this task as being rather challenging, requiring an intricate state space exploration of a semantics much more complicated than garbage collection. The contribution of the present paper is a type system to tackle exactly this challenge (cf. Section 5).

We elaborate on pointer races and the correspondence between the semantics.

Pointer Race Freedom. Pointer races generalize the concept of memory errors by taking into account the SMR algorithm [Haziza et al. 2016; Meyer and Wolff 2019a]. A memory error is an access through a dangling pointer, a pointer to an address that has been freed. Such accesses are prone to system crashes, for example, due to `segfaults`. Indeed, the `C/C++11` standard considers programs with memory errors to have an undefined semantics (catch-fire semantics) [ISO 2011].

To make precise which pointers in a computation are dangling, Haziza et al. [2016] introduce the notion of *validity*. A pointer is then dangling if it is invalid. Initially, all pointers are invalid. A pointer

is rendered valid if it receives its value from an allocation or from a valid pointer. A pointer becomes invalid if its address is freed or it receives its value form an invalid pointer. It is worth pointing out that $\texttt{free}(a)$ invalidates all pointers to address a but a subsequent reallocation of a validates only the receiving pointer. We denote the set of valid pointers after a computation τ by $valid_\tau$.

We already argued that dereferences of invalid pointers may lead to system crashes. Consequently, passing invalid pointers to the SMR algorithm may also be unsafe. Consider a call to $\texttt{retire}(p)$ requesting the SMR algorithm to free the address of p. If p is invalid, then its address has already been freed, resulting in a system crash due to a double free. Yet, we cannot forbid invalid pointers from being passed to SMR functions altogether. For instance, `protect` may be invoked with invalid pointers in Lines 14 and 32 of Michael&Scott's queue from Section 2. To support such calls, one deems a command $\texttt{enter}\,func(\bar{p}, \bar{u})$ unsafe, if replacing the actual values of invalid pointer arguments with arbitrary values may exhibit new (and potentially undesired) SMR behavior. We inherit the the formal definition from Meyer and Wolff [2019a] as it is an integral part of their proof strategy.

Definition 4.2 (Definition 5.12 in Meyer and Wolff [2019a]). Consider a computation τ with history h. A subsequent action act is an *unsafe call* if its command is $\texttt{enter}\ func(\bar{p}, \bar{u})$ with $p_i \notin valid_\tau$ for some i, $m_\tau(\bar{p}) = \bar{a}$, $m_\tau(\bar{u}) = \bar{d}$, and:

$$\exists\, c\ \exists\, \bar{b}.\ \ \big(\forall i.\ (a_i = c \vee p_i \in valid_\tau) \implies a_i = b_i\big) \wedge \mathcal{F}_O(h.func(t, \bar{b}, \bar{d}), c) \not\subseteq \mathcal{F}_O(h.func(t, \bar{a}, \bar{d}), c)\ .$$

Definition 4.3 (Following Definition 5.13 in Meyer and Wolff [2019a]). A computation $\tau.act$ is a *pointer race* if act (i) dereferences an invalid pointer, (ii) is an assumption comparing an invalid pointer for equality, (iii) retires an invalid pointer, or (iv) is an unsafe call.

Correspondence. Theorem 4.1 establishes a correspondence between the behavior of full $O[\![P]\!]^{Adr}_{Adr}$ and the simpler, garbage collected semantics $[\![P]\!]^{\varnothing}_{\varnothing}$. It states that we find for every computation $\tau \in O[\![P]\!]^{Adr}_{Adr}$ another computation $\sigma \in [\![P]\!]^{\varnothing}_{\varnothing}$ such that σ mimics τ. We denote this by $\tau \prec \sigma$. Relation $\prec$ requires τ and σ to agree on the control locations of all threads and the valid memory of τ. Intuitively, this means that any pointer-race-free action after τ has the same effect after σ because the action cannot access the invalid part of the memory without raising a pointer race. Put differently, threads cannot distinguish whether they execute in τ or in σ. So they cannot distinguish whether or not memory is reclaimed.

Technically, τ and σ agree on the valid memory of τ if $m_\tau|_{valid_\tau} = m_\sigma|_{valid_\tau}$. Here, $m_\tau|_{valid_\tau}$ denotes the restriction of m_τ to its valid part $valid_\tau$. It restricts the domain of m_τ to $valid_\tau$ and to data variables and to data selectors of addresses referenced from $valid_\tau$. It is worth pointing out the asymmetry in the definition of $\tau \prec \sigma$: m_σ is restricted to $valid_\tau$. This is necessary because there are no `free` commands in σ and thus pointer expressions that are invalidated in τ are never invalidated in σ. The correspondence is precise enough for verification results of safety properties to carry over from one semantics to the other.

5 A TYPE SYSTEM TO PROVE POINTER RACE FREEDOM

We present a type system a successful type check of which entails pointer race freedom as required by Theorem 4.1. The guiding idea of our types is to under-approximate the validity of pointers. To achieve this, our types incorporate the SMR algorithm in use and the guarantees it provides. It does so in a modular way: a parameter of the type system definition is an SMR automaton specifying the SMR algorithm.

A key design decision of our type system is to track no information about the data structure shape. Instead, we deduce runtime specific information from automatically dischargeable annotations. We still achieve the necessary precision because the same SMR algorithm may be used with different data structures. Hence, shape information should not help tracking its behavior.

5.1 Overview

Towards a definition of our type system, recall the memory life cycle from Section 1. The transition from the active to the retired stage requires care. The type system has to detect that a thread is guaranteed safe access to a retired node. This means finding out that an SMR protection was successful. Additionally, types need to be stable under interference. Nodes can be retired without a thread noticing. Hence, types need to ensure that the guarantees they provide cannot be spoiled by actions of other threads.

To tackle those problems, we use intersection types capturing which *access guarantees* a thread has for each pointer. We point out that this means we track information about nodes in memory through pointers to them. We use the following guarantees.

$\mathbb{L}$: Thread-local pointers referencing nodes in the local stage. The guarantee comes with two more properties. There are no valid aliases of the pointer and the referenced node is not retired. This gives the thread holding the pointer exclusive access.

$\mathbb{A}$ Pointers to nodes in the active stage. Active pointers are guaranteed to be valid, they can be accessed safely.

$\mathbb{S}$ Pointers to nodes which are protected by the SMR algorithm from being reclaimed. Such pointers can be accessed safely although the referenced node might be in the retired stage.

$\mathbb{E}_L$ SMR-specific guarantee that depends on a set of locations in the given SMR automaton. The idea is to track the history of SMR calls performed so far. This history is guaranteed to reach a location in L. The information about L bridges the (SMR-specific) gap between $\mathbb{A}$ and $\mathbb{S}$. Accesses to the pointer are potentially unsafe.

The interplay among these guarantees tackles the aforementioned challenges as follows. Consider a thread that just acquired a pointer p to a shared node. In the case of hazard pointers, this pointer comes without access guarantees. Hence, the thread issues a protection of p. We denote this with an SMR-specific type $\mathbb{E}$. For the protection to be successful, the programmer has to make sure that p is active during the invocation. The type system detects this through an annotation that adds guarantee $\mathbb{A}$ to p. We then deduce from the SMR automaton that p can be accessed safely because the protection was successful. This adds guarantee $\mathbb{S}$. (We have seen this on an example in Section 2.)

5.2 Types

Throughout the remainder of the section we fix an SMR automaton O relative to which we describe the type system. The SMR automaton induces a set of intersection types [Coppo and Dezani-Ciancaglini 1978; Pierce 2002] defined by the following grammar:

$$T \;::=\; \varnothing \mid \mathbb{L} \mid \mathbb{A} \mid \mathbb{S} \mid \mathbb{E}_L \mid T \wedge T\,.$$

The meaning of the guarantees $\mathbb{L}$ to $\mathbb{E}_L$ is as explained above. We also write a type T as the set of its guarantees where convenient. We define the predicate $\mathit{isValid}(T)$ to hold if $T \cap \{\mathbb{S}, \mathbb{L}, \mathbb{A}\} \neq \varnothing$. The three guarantees serve as syntactic under-approximations of the semantic notion of validity from the definition of pointer races (cf. Section 4).

There is a restriction on the sets of locations L for which we provide guarantees $\mathbb{E}_L$. To understand it, note that our type system infers guarantees about the protection of pointers thread-locally from the code, that is, as if the code was sequential. Soundness then shows that these guarantees carry over to the computations of the overall program where threads interfere. To justify this sequential to concurrent lifting, we rely on the concept of interference freedom due to Owicki and Gries [1976]. A set of locations L in the SMR automaton O is *closed under interference from other threads*, if no SMR command issued by a thread different from z_t (whose protection we track) can leave the locations. Formally, we require that for every transition $l \xrightarrow{f(t',*),\,g} l'$ with $l \in L$ and $l' \notin L$ we have

guard g implying $t' = z_t$. We only introduce guarantees $\mathbb{E}_L$ for sets of locations L that are closed under interference from other threads.

Type environments Γ are total functions that assign a type to every pointer and every angel in the code being typed. To fix the notation, $\Gamma(x) = T$ or $x : T \in \Gamma$ means x is assigned T in environment Γ. We write $\Gamma, x : T$ for $\Gamma \uplus \{x : T\}$. If the type of x does not matter, we just write Γ, x. The initial type environment Γ_{init} assigns $\varnothing$ to every pointer and angel.

Our type system will be control-flow sensitive [Crary et al. 1999; Foster et al. 2002; Hunt and Sands 2006], which means type judgements take the form

$$\{\,\Gamma_{pre}\,\}\ stmt\ \{\,\Gamma_{post}\,\}\ .$$

The thing to note is that the type assigned to a pointer/angel is not constant throughout the program but depends on the commands that have been executed. So we may have the type assignment $x : T$ in Γ_{pre} but $x : T'$ in the type environment Γ_{post} with $T \neq T'$.

Control-flow sensitivity requires us to formulate how types change under the execuction of SMR commands. Towards a definition, we associate with every type a set of locations in $O = O_{Base} \times O'$. Guarantee $\mathbb{E}_L$ already comes with a set of locations. Guarantee $\mathbb{S}$ grants safe access to the tracked address. In terms of locations, it should not be possible to free the address stored in z_a. We define $SafeLoc(O)$ to be the largest set of locations in the SMR automaton that is closed under interference from other threads and for which there is no transition $l \xrightarrow{\texttt{free}(a),\,g} l'$ with $l \in SafeLoc(O)$, l' not accepting, and g implying $a = z_a$. Guarantee $\mathbb{A}$ is characterized by location L_2 in O_{Base}. Indeed, a pointer is active iff O_{Base} is in its initial location. For $\mathbb{L}$ we also use location L_2. The discussion yields:

$$\begin{aligned} Loc(\varnothing) &:= Loc(O) & Loc(\mathbb{E}_L) &:= L \\ Loc(\mathbb{A}) &:= \{L_2\} \times Loc(O') & Loc(\mathbb{S}) &:= SafeLoc(O) \\ Loc(\mathbb{L}) &:= \{L_2\} \times Loc(O') & Loc(T_1 \wedge T_2) &:= Loc(T_1) \cap Loc(T_2)\ . \end{aligned}$$

The set of locations associated with a type is defined to over-approximate the locations reachable in the SMR automaton by the (history of the) current computation. With this understanding, it should be possible for command com to transform $x : T$ into $x : T'$ if the locations associated with T' over-approximate the post-image under x and com of the locations associated with T. We define the *type transformer relation* $T, x, com \rightsquigarrow T'$ by the following conditions:

$$\begin{aligned} post_{x,com}(Loc(T)) &\subseteq Loc(T') \\ isValid(T') &\Rightarrow isValid(T) \\ \{\mathbb{L}, \mathbb{A}\} \cap T' &\subseteq \{\mathbb{L}, \mathbb{A}\} \cap T\ . \end{aligned}$$

The over-approximation of the post-image is the first inclusion. The implication states that SMR commands cannot validate pointers. We can, however, deduce from the fact that the address has not been retired ($\mathbb{A}$ or $\mathbb{L}$) and the SMR command has been executed, that it is safe to access the address ($\mathbb{S}$). The last inclusion states that SMR commands cannot establish the guarantees $\mathbb{L}$ and $\mathbb{A}$. It is worth pointing out that the relation $T, x, com \rightsquigarrow T'$ only depends on the SMR automaton, up to a choice of variable names. This means we can tabulate it to guarantee quick access when typing a program. We also write $\Gamma, com \rightsquigarrow \Gamma'$ if we have $\Gamma(x), x, com \rightsquigarrow \Gamma'(x)$ for all pointers/angels x. We write $\Gamma \rightsquigarrow \Gamma'$ if we take the post-image to be the identity. For an example, refer to Section 6.1.

Guarantees $\mathbb{L}$ and $\mathbb{A}$ are special in that their sets of locations, $Loc(\mathbb{L})$ and $Loc(\mathbb{A})$, are not closed under interference. For $\mathbb{L}$, the type rules ensure interference freedom. They do so by enforcing that `retire` is not invoked with invalid pointers. Hence, the fact that $\mathbb{L}$-pointers have no valid aliases implies that other threads cannot retire them. So O_{Base} remains in L_2 no matter the interference. For $\mathbb{A}$,

the type rules account for interference. We define an operation $rm(\Gamma)$ that takes an environment and removes all $\mathbb{A}$ guarantees for thread-local pointers and angels:

$$rm(\Gamma) \;:=\; \{x{:}T \setminus \{\mathbb{A}\} \mid x{:}T \in \Gamma \wedge x \notin \mathit{shared}\} \;\cup\; \{x{:}\varnothing \mid x \in \mathit{shared}\}\,.$$

The operation also has an effect on shared pointers and angels where it removes all guarantees. The reasoning is as follows. An interference on a shared pointer or angel may change the address being pointed to. Guarantees express properties about addresses, indirectly via their pointers. As we do not have any information about the new address, the pointer receives the empty set of guarantees.

5.3 Type System

Our type system is given in Figure 6. We write $\vdash \{\,\Gamma_{\mathit{init}}\,\}\ \mathit{stmt}\ \{\,\Gamma\,\}$ to indicate that $\{\,\Gamma_{\mathit{init}}\,\}\ \mathit{stmt}\ \{\,\Gamma\,\}$ is derivable with the given rules. We write $\vdash \mathit{stmt}$ if there is an environment Γ so that $\vdash \{\,\Gamma_{\mathit{init}}\,\}\,\mathit{stmt}\{\,\Gamma\,\}$. In this case, we say the program *type checks*. Soundness will show that a type check entails pointer race freedom and the absence of double retires.

We distinguish between rules for statements and rules for primitive commands. We assume that primitive commands *com* are wrapped inside an atomic block, like `beginAtomic`; *com*; `endAtomic`. With this assumption, the rules for primitive commands need not handle the fact that guarantee $\mathbb{A}$ is not closed under interference. Interference will be taken into account by the rules for statements. The assumption of atomic blocks can be established by a simple preprocessing of the program. We do not make it explicit but assume it has been applied.

The rules for primitive commands, Figure 6a, that are not related to SMR are straightforward. Rule (ASSIGN1) copies the type of the right-hand side pointer to the left-hand side pointer of the assignment. Additionally, both pointers lose their $\mathbb{L}$ qualifier since the command creates an alias. Rule (ASSIGN2) ensures that the dereferenced pointer is valid and then sets the type of the assigned pointer to the empty type. The assigned pointer does not receive any guarantees since we do not track guarantees for selectors. Rule (ASSIGN3) checks the dereferenced pointer for validity and removes $\mathbb{L}$ from the pointer that is aliased. Data assignments, Rules (ASSIGN4), (ASSIGN5), and (ASSIGN6), simply check dereferenced pointers for validity. Allocations grant the target pointer the $\mathbb{L}$ guarantee, Rule (MALLOC). Assumptions of the form $p = q$ check that both pointers are valid and join the type information, Rule (ASSUME1). Guarantee $\mathbb{L}$ is removed due to the alias. All other assumptions have no effect on the type environment, Rule (ASSUME2). Similarly, Rule (EQUAL) joins type information in the case of assertions. However, no validity check is performed and $\mathbb{L}$ is not removed. Rule (ACTIVE) adds the $\mathbb{A}$ guarantee. Note that x is a pointer or an angel. Angels are always local variables. Their allocation does not justify any guarantees, in particular not $\mathbb{L}$, as they hold full sets of addresses, Rule (ANGEL). We can also assert membership of an address held by a pointer in a set of addresses held by an angel, Rule (MEMBER).

SMR-related commands may change the entire type environment, rather than manipulating only the pointers that occur syntactically in the command. This is because of pointer aliasing on the one hand, and because of the SMR automaton on the other hand (for instance, `enterQ` has an effect on all pointers). The post type environment of Rules (ENTER) and (EXIT) simply infers guarantees wrt. the pre type environment and the emitted event. Note that this is the only way to infer SMR-specific guarantees $\mathbb{E}_L$, i.e., these guarantees solely depend on the SMR commands. Moreover, Rule (ENTER) performs a pointer race check as defined in Section 4. Predicate $\mathit{safeEnter}(\Gamma, \mathit{func}(\bar{p}, \bar{u}))$ evaluates to *true* iff the command `enter` $\mathit{func}(\bar{p}, \bar{u})$ is guaranteed to be free from pointer races given the types in Γ. The formalization coincides with Definition 4.2 except that it replaces *valid* by the under-approximation $\mathit{isValid}(\cdot)$. A special case of Rule (ENTER) is the invocation of `retire`(p), which requires the argument p to be active. This will prevent double retires.

$$\text{(MALLOC)}\ \frac{p \notin \mathit{shared} \qquad T = \{\mathbb{L}\}}{\{\Gamma, p\}\ p := \texttt{malloc}\ \{\Gamma, p:T\}} \qquad \text{(ASSIGN1)}\ \frac{T' = T \setminus \{\mathbb{L}\}}{\{\Gamma, p, q:T\}\ p := q\ \{\Gamma, p:T', q:T'\}} \qquad \text{(ASSIGN2)}\ \frac{\Gamma(q) = T \qquad \mathit{isValid}(T)}{\{\Gamma, p\}\ p := q.\texttt{next}\ \{\Gamma, p:\varnothing\}}$$

$$\text{(ASSIGN3)}\ \frac{\Gamma(p) = T \qquad \mathit{isValid}(T) \qquad T'' = T' \setminus \{\mathbb{L}\}}{\{\Gamma, q:T'\}\ p.\texttt{next} := q\ \{\Gamma, q:T''\}} \qquad \text{(ASSIGN4)}\ \frac{}{\{\Gamma\}\ u := \mathsf{op}(\bar{u})\ \{\Gamma\}} \qquad \text{(ASSIGN5)}\ \frac{\Gamma(q) = T \qquad \mathit{isValid}(T)}{\{\Gamma\}\ u := q.\texttt{data}\ \{\Gamma\}}$$

$$\text{(ASSIGN6)}\ \frac{\Gamma(p) = T \qquad \mathit{isValid}(T)}{\{\Gamma\}\ p.\texttt{data} := u\ \{\Gamma\}} \qquad \text{(ASSUME1)}\ \frac{\mathit{isValid}(T) \qquad \mathit{isValid}(T') \qquad T'' = (T \wedge T') \setminus \{\mathbb{L}\}}{\{\Gamma, p:T, q:T'\}\ \texttt{assume}\ p = q\ \{\Gamma, p:T'', q:T''\}}$$

$$\text{(ASSUME2)}\ \frac{\mathit{cond} \not\equiv p = q}{\{\Gamma\}\ \texttt{assume}\ \mathit{cond}\ \{\Gamma\}} \qquad \text{(EQUAL)}\ \frac{T'' = T \wedge T'}{\{\Gamma, p:T, q:T'\}\ \texttt{@inv}\ p = q\ \{\Gamma, p:T'', q:T''\}}$$

$$\text{(ACTIVE)}\ \frac{T' = T \wedge \{\mathbb{A}\}}{\{\Gamma, x:T\}\ \texttt{@inv active}(x)\ \{\Gamma, x:T'\}} \qquad \text{(ANGEL)}\ \frac{r \notin \mathit{shared}}{\{\Gamma, r\}\ \texttt{@inv angel}\ r\ \{\Gamma, r:\varnothing\}} \qquad \text{(MEMBER)}\ \frac{\Gamma(r) = T' \qquad T'' = T \wedge T'}{\{\Gamma, p:T\}\ \texttt{@inv}\ p\ \texttt{in}\ r\ \{\Gamma, p:T''\}}$$

$$\text{(ENTER)}\ \frac{\mathit{safeEnter}(\Gamma, \mathit{func}(\bar{p}, \bar{u})) \qquad \Gamma, \texttt{enter}\ \mathit{func}(\bar{p}, \bar{u}) \rightsquigarrow \Gamma' \qquad \mathit{func}(\bar{p}, \bar{u}) \equiv \texttt{retire}(p) \wedge \Gamma(p) = T \implies \mathbb{A} \in T}{\{\Gamma\}\ \texttt{enter}\ \mathit{func}(\bar{p}, \bar{u})\ \{\Gamma'\}} \qquad \text{(EXIT)}\ \frac{\Gamma, \texttt{exit}\ \mathit{func} \rightsquigarrow \Gamma'}{\{\Gamma\}\ \texttt{exit}\ \mathit{func}\ \{\Gamma'\}}$$

(a) Type rules for primitive commands.

$$\text{(INFER)}\ \frac{\Gamma_1 \rightsquigarrow \Gamma_2 \qquad \{\Gamma_2\}\ \mathit{stmt}\ \{\Gamma_3\} \qquad \Gamma_3 \rightsquigarrow \Gamma_4}{\{\Gamma_1\}\ \mathit{stmt}\ \{\Gamma_4\}} \qquad \text{(BEGIN)}\ \frac{}{\{\Gamma\}\ \texttt{beginAtomic}\ \{\Gamma\}} \qquad \text{(END)}\ \frac{}{\{\Gamma\}\ \texttt{endAtomic}\ \{\mathit{rm}(\Gamma)\}}$$

$$\text{(SEQ)}\ \frac{\{\Gamma\}\ \mathit{stmt}_1\ \{\Gamma'\} \qquad \{\Gamma'\}\ \mathit{stmt}_2\ \{\Gamma''\}}{\{\Gamma\}\ \mathit{stmt}_1; \mathit{stmt}_2\ \{\Gamma''\}} \qquad \text{(CHOICE)}\ \frac{\{\Gamma\}\ \mathit{stmt}_1\ \{\Gamma'\} \qquad \{\Gamma\}\ \mathit{stmt}_2\ \{\Gamma'\}}{\{\Gamma\}\ \mathit{stmt}_1 \oplus \mathit{stmt}_2\ \{\Gamma'\}} \qquad \text{(LOOP)}\ \frac{\{\Gamma\}\ \mathit{stmt}\ \{\Gamma\}}{\{\Gamma\}\ \mathit{stmt}^*\ \{\Gamma\}}$$

(b) Type rules for statements.

Fig. 6. Type rules.

The rules for statements are given in Figure 6b. Rule (INFER) allows for type transformations at any point, in particular to establish the proper pre/post environments for the Rules (CHOICE) and (LOOP). Entering an atomic block, Rule (BEGIN), has no effect on the type environment. Exiting an atomic block allows for interference. Hence, Rule (EXIT) removes any type information from the type environment that can be tampered with by other threads. Sequences of statements are straightforward, Rule (SEQ). Choices require a common pre and post type environment, Rule (CHOICE). Loops require a type environment that is stable under the loop body, Rule (LOOP).

5.4 Soundness

Our goal is to show that a successful type check $\vdash \mathit{stmt}$ implies pointer race freedom and the absence of double retires. There are two challenges. We already commented on the problematic sequential

to concurrent lifting that motivated the definition of interference freedom. The second difficulty is that the type system relies on the program's invariant annotations. The set of computations ignores these annotations. To reconcile the assumptions about the program, we have to prove the invariant annotations correct. Interestingly, we can use garbage collection for this purpose, meaning the invariant annotations only have to hold in $[\![P]\!]^{\varnothing}_{\varnothing}$, although the following results refer to the computations in $O[\![P]\!]^{\varnothing}_{Adr}$. Intuitively, garbage collection is sufficient because we have elision support (cf. Section 3): it allows us to remove frees from a computation and then apply Lemma 3.1.

Pointer race freedom and the absence of double retires will be consequences of a more general soundness result that makes explicit the information tracked by our type system. We give some auxiliary definitions that ease the formulation. We write $\tau \models_{\varphi} T$ if there is a location $l \in Loc(T)$ associated with the type T so that $(l_{init}, \varphi) \xrightarrow{\mathcal{H}(\tau)} (l, \varphi)$. The definition is parameterized in the valuation φ determining the thread and the address to be tracked. We write $\tau, t \models x : T$ if for every address $a \in m_\tau(x)$ we have $\tau \models_{\varphi} T$, with $\varphi = \{z_t \mapsto t, z_a \mapsto a\}$. The thread is given. The address is the one held by the pointer or among the ones held by the angel, as determined by the computation. We write $\tau, t \models \Gamma$ if we have $\tau, t \models x : T$ for all type assignments $x : T \in \Gamma$.

Soundness states that a type environment annotating a program point approximates the history of every computation reaching this point. Moreover, $isValid(\cdot)$ approximates validity. To make this precise, we define the relation $\models \{\,\Gamma_{init}\,\}\ stmt\ \{\,\Gamma\,\}$. It requires for every $\tau \in O[\![P]\!]^{\varnothing}_{Adr}$ where thread t executes $stmt$ to completion that (i) $\tau, t \models \Gamma$ holds, and (ii) for every $p : T \in \Gamma$ with $isValid(T)$ we have $p \in valid_\tau$. The soundness result now lifts the syntactic derivation relation $\vdash$ to the semantic soundness relation $\models$.

Theorem 5.1 (Soundness). *If* $inv([\![P]\!]^{\varnothing}_{\varnothing})$, *then* $\vdash \{\,\Gamma_{init}\,\}\ stmt\ \{\,\Gamma\,\}$ *implies* $\models \{\,\Gamma_{init}\,\}\ stmt\ \{\,\Gamma\,\}$.

Proof Sketch. We proceed by induction on the length of computations τ from $O[\![P]\!]^{\varnothing}_{Adr}$. During the proof, we need to access the types encountered by thread t along the execution of $stmt$. To make them explicit, we define the straight-line version $stmt(\tau, t)$ of $stmt$ induced by τ and t. It is obtained by projecting τ to the commands of thread t. One can show that if we can derive a typing for the program then we can derive it for the induced straight-line program:

$$\vdash \{\,\Gamma_{init}\,\}\ stmt\ \{\,\Gamma\,\} \qquad \text{implies} \qquad \vdash \{\,\Gamma_{init}\,\}\ stmt(\tau, t)\ \{\,\Gamma\,\}\ .$$

The implication should be intuitive. The typing of the overall program can be seen as an intersection over the typings of the induced straight-line programs.

The induction hypothesis links the current type environment Γ derived for the straight-line program to the semantic information carried by the computation. The hypothesis strengthens the requirements (i) and (ii) in the definition of soundness by the following two conditions, where we assume $\Gamma(x) = T$. (iii) If $\mathbb{L} \in T$, then x is a pointer that does not have valid aliases. That is, $m_\tau(x) = m_\tau(q)$ implies $q \notin valid_\tau$. Note that angels cannot obtain $\mathbb{L}$ according to the type rules. (iv) If $\mathbb{A} \in T$, then thread t is in an atomic block. The interesting argumentation in the induction step is in the case when another thread appends an action, $\tau.act$. It can be summarized as follows. Property (i) continues to hold for $\tau.act$ because the type T of x is closed under interference; for $\mathbb{L}$ and $\mathbb{A}$ we argue separately in the following. If $\mathbb{L} \in T$, then act cannot use a valid alias of x. In particular, it cannot retire x according to the premise of Rule (enter). If $\mathbb{A} \in T$, then thread t is in an atomic block and there is no chance to append action act of another thread. The case does not occur. Consider property (ii). Assume $isValid(T)$ holds. That is, T contains one of $\mathbb{A}, \mathbb{L}, \mathbb{S}$. If $\mathbb{L} \in T$ or $\mathbb{A} \in T$, then the above reasoning for (i) already implies (ii). Otherwise, we have $\mathbb{S} \in T$. It implies (ii) because $\mathbb{S}$ is closed under interference. Property (iii) follows from the fact that act cannot contain, and thus cannot create, a valid alias of x. Lastly, to conclude property (iv), note that another thread cannot append an action while t is inside an atomic block. □

The first consequence of soundness is that a successful type check implies pointer race freedom. Phrased differently, the rules from Figure 6 allow for a successful typing only if there are no pointer races. That is, our type system performs a pointer race freedom check indeed.

Proposition 5.2. *If* $inv(\llbracket P \rrbracket^{\varnothing}_{\varnothing})$, *then* $\vdash P$ *implies that* $O\llbracket P \rrbracket^{\varnothing}_{Adr}$ *is pointer-race-free.*

The proposition gives an effective means of checking the premise of Theorem 4.1: determine a typing using the proposed type system (cf. Section 7) and discharge the invariant annotations using an off-the-shelf verification tool (cf. Section 8).

Proof Sketch. To see the proposition, consider $\tau.act \in O\llbracket P \rrbracket^{\varnothing}_{Adr}$. We focus on the case where the last action is a dereference, say due to command *com* being $p := q.\texttt{next}$. The remaining cases in the definition of pointer races are similar. We show that the dereference is safe, $q \in valid_\tau$. Let thread t perform the dereference. Let $stmt(\tau.act, t) = stmt; com$ be the induced straight-line program. As observed above, since the program type checks also *stmt*; *com* will type check. Say we can derive $\{\, \Gamma_{init} \,\}\ stmt; com\ \{\, \Gamma \,\}$. The only way to type a composition *stmt*; *com* is Rule (SEQ). It requires an environment Γ' so that $\{\, \Gamma_{init} \,\}\ stmt\ \{\, \Gamma' \,\}$ and $\{\, \Gamma' \,\}\ com\ \{\, \Gamma \,\}$ are derivable. The only way to type an assignment $p := q.\texttt{next}$ is Rule (ASSIGN2). By its premise, $\Gamma'(q) = T$ with $isValid(T)$. Theorem 5.1 yields $q \in valid_\tau$. The dereference of q is safe. □

The second consequence of soundness is that a successful type check means the program does not perform double retires. This is the precondition for a meaningful application of O_{Base} (cf. Section 3).

Proposition 5.3. *If* $inv(\llbracket P \rrbracket^{\varnothing}_{\varnothing})$, *then* $\vdash P$ *implies that* $O\llbracket P \rrbracket^{Adr}_{Adr}$ *does not perform double retires.*

The argumentation is along the lines of Proposition 5.2. To perform a retire, Rule (ENTER) requires the pointer to be active. This, in turn, means O_{Base} is in state L_2. The state, however, can only be reached if there were no earlier retires of the address or the earlier retires have been followed by a free. In both cases, we do not have a double retire.

The next section gives an in-depth example on how to apply our type system. The two sections thereafter automate the checks in Theorem 5.1: we give an efficient algorithm for type inference $\vdash P$ and show how to discharge the invariants $inv(\llbracket P \rrbracket^{\varnothing}_{\varnothing})$ with the help of off-the-shelf verification tools.

6 EXAMPLE

We apply our type system to Michael&Scott's queue with EBR from Section 2. Here, a single custom guarantee $\mathbb{E}_{acc}$ is sufficient. We define $Loc(\mathbb{E}_{acc})$ to be those locations where thread z_t is guarantee to have returned from a call to `leaveQ` but has not yet invoked `enterQ`. That is, $\mathbb{E}_{acc}$ captures when z_t is accessing the data structure. The sets of locations represented by $\mathbb{A}$, $\mathbb{S}$, and $\mathbb{E}_{acc}$ can be read of the cross-product SMR automaton $O_{Base} \times O_{EBR}$ in Figure 7. It is worth pointing out that $Loc(\mathbb{S})$ does not contain location (L_2, L_4). For a set containing (L_2, L_4) to be closed under interference we would need to have (L_3, L_4) in that set. However, (L_3, L_4) allows for a `free` of z_a and thus must not belong to $Loc(\mathbb{S})$ by definition.

In the following, we illustrate the type transformer relation, the use of angels, the typing of programs, and explain how to find suitable annotations for the type inference to go through.

6.1 Type Transformer Relation

We illustrate the computation of the type transformer relation for `exit leaveQ` and the inference of $\mathbb{S}$.

First, we establish the type transformer relation $\varnothing, x, \texttt{exit leaveQ} \rightsquigarrow \mathbb{E}_{acc}$. This boils down to checking $post_{x,\texttt{exit leaveQ}}(Loc(\varnothing)) \subseteq Loc(\mathbb{E}_{acc})$ because the remaining properties of the type transformer relation are trivially satisfied (we do not add any of $\{\mathbb{A}, \mathbb{L}, \mathbb{S}\}$). The empty type corresponds

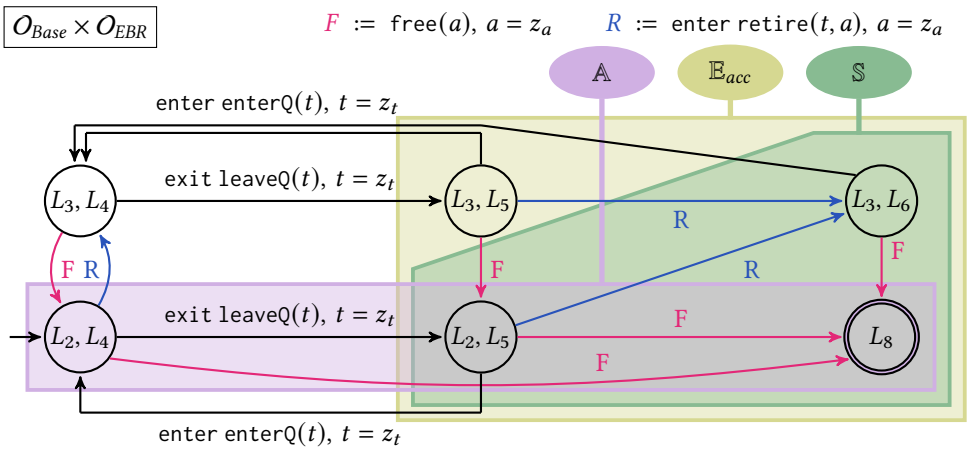


Fig. 7. Cross-product SMR automaton for $O_{Base} \times O_{EBR}$ and EBR-specific types.

to no knowledge about previously executed SMR commands, which means $Loc(\varnothing) = L$ with L the set of all locations of $O_{Base} \times O_{EBR}$. We compute the post-image of L wrt. x and `exit leaveQ` in the SMR automaton from Figure 7. To this end, we consider all transitions labeled with `exit leaveQ`(t). The pointer or angel x does not play a role. We derive the desired inclusion as follows:

$$post_{x,\texttt{exit leaveQ}}(Loc(\varnothing)) \;=\; post_{x,\texttt{exit leaveQ}}(L) \;=\; L \setminus \{(L_2, L_4), (L_3, L_4)\} \;=\; Loc(\mathbb{E}_{acc})\;.$$

Second, we show how to infer $\mathbb{S}$. From Figure 7 we know that $\mathbb{E}_{acc}$ alone does not yield $\mathbb{S}$ because of location (L_3, L_5); we also need $\mathbb{A}$. We establish $\mathbb{E}_{acc} \wedge \mathbb{A} \rightsquigarrow \mathbb{E}_{acc} \wedge \mathbb{A} \wedge \mathbb{S}$. Since $\mathbb{E}_{acc} \wedge \mathbb{A}$ is valid and we do not add $\mathbb{L}$, the key task is to establish $Loc(\mathbb{E}_{acc} \wedge \mathbb{A}) \;\subseteq\; Loc(\mathbb{E}_{acc} \wedge \mathbb{A} \wedge \mathbb{S})$. As $Loc(\mathbb{E}_{acc} \wedge \mathbb{A}) \subseteq Loc(\mathbb{E}_{acc} \wedge \mathbb{A})$ trivially holds, it suffices to show $Loc(\mathbb{E}_{acc} \wedge \mathbb{A}) \subseteq Loc(\mathbb{S})$:

$$Loc(\mathbb{E}_{acc} \wedge \mathbb{A}) \;=\; Loc(\mathbb{E}_{acc}) \cap Loc(\mathbb{A}) \;=\; \{(L_2, L_5), L_8\} \;\subseteq\; \{(L_2, L_5), (L_3, L_6), L_8\} \;=\; Loc(\mathbb{S})\;.$$

6.2 Angels

To illustrate the use of angels, consider the excerpt of the dequeue operation depicted in Figure 8. Note that calls to SMR functions lead to two consecutive commands. The atomic block ensures the commands are executed without interruption by other threads. To infer it, we rely on standard moverness arguments [Lipton 1975]: command `enter leaveQ()` is a right-mover because it does not affect the memory nor the observer $O_{Base} \times O_{EBR}$. The call to `leaveQ` guarantees that no currently active address is reclaimed until `enterQ` is called. It thus protects an unbounded number of addresses before a thread acquires a pointer to them. Later, when a thread acquired a pointer to such an address in order to access it, the address may no longer be active and thus the type system may not be able to infer $\mathbb{S}$ (cf. Section 6.1). To overcome this problem, we use an angel r. Given its angelic semantics, r will capture all addresses that are protected by the `leaveQ` call, Lines 67

```
67  @inv angel r;
68  beginAtomic
69    enter leaveQ();
70    exit leaveQ;
71    @inv active(r);
72  endAtomic
73  // ...
74  Node* head = Head;
75  Node* tail = Tail;
76  @inv head in r;
77  Node* next = head->next;
78  // ...
79  @inv next in r;
80  data_t output = next->data;
81  // ...
82  enter exitQ(); exit exitQ;
```

Fig. 8. Excerpt of dequeue using angel r.

```
83  { Head,head,next,r:∅ }
84    @inv angel r;
85  { Head,head,next,r:∅ }
86    beginAtomic
87  { Head,head,next,r:∅ }
88    enter leaveQ();
89  { Head,head,next,r:∅ }
90    exit leaveQ;
91  { Head,head,next:∅, r:𝔼acc }
92    @inv active(r);
93  { Head,head,next:∅, r:𝔼inv ∧ 𝔸 }
94  { Head,head,next:∅, r:𝔼inv ∧ 𝔸 ∧ 𝕊 }
95    endAtomic
96  { Head,head,next:∅, r:𝔼inv ∧ 𝕊 }
97    // ...
98    // ...
99  { Head,head,next:∅, r:𝔼inv ∧ 𝕊 }
100   Node* head = Head;
101 { Head,head,next:∅, r:𝔼inv ∧ 𝕊 }
102   // ...
103 { Head,head,next:∅, r:𝔼inv ∧ 𝕊 }
104   @inv head in r;
105 { Head,next:∅, head,r:𝔼inv ∧ 𝕊 }
106   Node* next = head->next;
107   // ...
108 { Head,next:∅, head,r:𝔼inv ∧ 𝕊 }
109   @inv next in r;
110 { Head:∅, next,head,r:𝔼inv ∧ 𝕊 }
111   data_t output = next->data;
112   // ...
```

Fig. 9. Typing for EBR using angels.

to 71. Later, upon accessing/dereferencing a pointer p, we make sure that r captures the address pointed to by p, Lines 76 and 79.

6.3 Typing

We give a typing for the code from Figure 8 in Figure 9. We start in Line 83 with type $\emptyset$ for all pointers and the angel r. The allocation of r in Line 84 has no effect on the type assignment. The same holds when entering an atomic block, Line 86. Line 88 invokes `leaveQ`. Again, the types are not affected because the SMR automaton has no transitions labeled with `enter leaveQ`. Next, the invocation returns, Line 90. Following the discussion from Section 6.1, we obtain $\mathbb{E}_{acc}$ for r, Line 91. It is worth pointing out that r is treated like an ordinary pointer when it comes to the type transformer relation.

To capture in the type system the set of addresses that can be safely accessed in the subsequent code, we want to lift $\mathbb{E}_{acc}$ of r to $\mathbb{S}$. We annotate r to hold a set of active addresses, Line 92. This yields type $\mathbb{E}_{acc} \wedge \mathbb{A}$ for r, Line 93. As explained above, we can now lift this type to $\mathbb{E}_{acc} \wedge \mathbb{A} \wedge \mathbb{S}$, Line 94. Recall that the allocation of r in Line 84 is angelic. That is, the addresses held by r will indeed be chosen to be active.

In the subsequent code, we already added annotations (cf. Section 6.2) ensuring that accessed/dereferenced pointers are captured by the angel r. For instance, Line 104 requires the address of `head` to be captured by r. That this is the case indeed is established when the annotations are discharged. For the typing, we can copy $\mathbb{E}_{acc} \wedge \mathbb{S}$ from r over to `head`. As a consequence, the dereference of `head` in Line 106 is safe. Similarly, we require `next` to be captured by r in Line 109 such that the dereference in Line 111 is safe.

6.4 Annotations

We explain our algorithm to automatically add to the program in Figure 2 the annotations in Figure 8 in order to arrive at the typing in Figure 9. We focus on the dereference of `head` in Line 77. Without annotations, the type inference will fail because it cannot conclude that `head` is guaranteed to be valid. To fix this, we implemented a sequence of tactics that we invoke one after the other. If none of them fixes the issue, we give up the type inference and report the failure to the user.

The first tactic simply adds an `@inv active(head)` annotation to Line 77. This makes `head` valid and the type inference go through for Line 77. However, we should only add the annotation if it

actually holds. To check this, we employ the technique from Section 8. In this particular case, we will find that the annotation does not hold; so we try to fix the problem with another tactic.

The second tactic adds an angel r to the (syntactically) most recent `leaveQ` call. We use a template to transform the sequence `enter leaveQ(); exit leaveQ;` to the code from Lines 67-72. (A subsequent use of this tactic will skip this step and reuse the existing angel.) Then, we fix Line 77 by adding the annotation `@inv head in r` before it, as shown in Line 76. This makes `head` valid. Whether the annotation holds is again checked with the technique from Section 8.

It is worth pointing out that the second tactic is EBR-specific. From our experience, every SMR algorithm/automaton comes with a small set of tactics that significantly help finding the right annotations—EBR requires the above tactic and HP requires two specific tactics. We do not believe that there is a silver bullet of tactics since SMR algorithms may vary greatly, as seen in the cases of EBR and HP. Theoretically speaking, one could find the annotations by an exhaustive search (finitely many angels will suffice), but this will not scale.

6.5 Hazard Pointers

Our approach applies to lock-free data structures with hazard pointers just as well as in the case of EBR. The main difference is that HP typically does not require angels because pointers are protected after they are acquired. However, the size of the SMR automaton for HP grows in the number of hazard pointers. For two hazard pointers it consists of 17 locations [Meyer and Wolff 2019b]. We cannot cover a comprehensive example here.

7 TYPE INFERENCE

We show that type inference is surprisingly efficient, namely quadratic time.

Theorem 7.1. *Given a program stmt, the type inference* $\vdash stmt$ *is computable in time* $O(|\,stmt\,|^2)$.

As common in type systems [Pierce 2002], our algorithm for type inference is constraint-based. We associate with the program *stmt* a constraint system $\Phi(\Gamma_{init}, stmt, X)$. The variables X are interpreted over the set of type environments enriched with a value $\top$ for a failed type inference. The correspondence between solving the constraint system and type inference will be the following. An environment Γ can be assigned to X in order to solve the constraint system if and only if $\{\,\Gamma_{init}\,\}\ stmt\ \{\,\Gamma\,\}$. As a consequence, a non-trivial solution to X will show $\vdash stmt$.

Our type inference algorithm will be a fixed-point computation. The canonical choice for a domain over which to compute would be the set of types ordered by $\rightsquigarrow$. The problem is that types of the form $\mathbb{E}_L$ and $\mathbb{E}_L \wedge \mathbb{E}_{L'}$ with $L \subseteq L'$ are comparable, $\mathbb{E}_L \rightsquigarrow \mathbb{E}_L \wedge \mathbb{E}_{L'}$ and $\mathbb{E}_L \wedge \mathbb{E}_{L'} \rightsquigarrow \mathbb{E}_L$. This is not merely a theoretical issue of the domain being a quasi instead of a partial order. It means we compute over too large a domain, namely a powerset lattice where we should have used a lattice of antichains [Wulf et al. 2006]. We factorize the set of all types along such equivalences $\rightsquigarrow \cap \rightsquigarrow^{-1}$. The resulting $AntiChainTypes := (Types/_{\rightsquigarrow \cap \rightsquigarrow^{-1}}, \rightsquigarrow)$ is a complete lattice [Birkhoff 1948].

Type environments can be understood as total functions into this antichain lattice. We enrich the set of functions by a value $\top$ to indicate a failed type inference. The result is the complete lattice of enriched type environments

$$Envs_\top \ := \ (AntiChainTypes^{Vars} \cup \{\top\}, \sqsubseteq)\ .$$

Between environments, we define $\Gamma \sqsubseteq \Gamma'$ to hold if for all $x \in Vars$ we have $\Gamma(x) \rightsquigarrow \Gamma'(x)$. This lifts $\rightsquigarrow$ to the function domain. Value $\top$ is defined to be the largest element.

The constraint system $\Phi(\Gamma_{init}, stmt, X)$ is defined in Figure 10. We proceed by induction over the structure of statements and maintain triples $(X, stmt, Y)$. The idea is that statement *stmt* will turn the enriched type environment stored in variable X into an environment upper bounded by Y.

$$\begin{aligned}
\Phi(X, com, Y): &\quad sp(X, com) \sqsubseteq Y \\
\Phi(X, stmt_1; stmt_2, Y): &\quad \Phi(X, stmt_1, Z) \wedge \Phi(Z, stmt_2, Y), \quad Z \text{ fresh} \\
\Phi(X, stmt_1 \oplus stmt_2, Y): &\quad \Phi(X, stmt_1, Y) \wedge \Phi(X, stmt_2, Y) \\
\Phi(X, stmt^*, Y): &\quad \Phi(Y, stmt, Y) \wedge X \sqsubseteq Y
\end{aligned}$$

Fig. 10. Constraint system $\Phi(X, stmt, Y)$.

Consider the case of basic commands. We will define $sp(X, com)$ to be the strongest enriched type environment resulting from the environment in X when applying command com. The constraint $sp(X, com) \sqsubseteq Y$ requires Y to be an upper bound. Note that Y still contains safe type information. For a sequential composition, we introduce a fresh variable Z for the enriched type environment determined by $stmt_1$ from X. We then require $stmt_2$ to transform this environment into Y. For a choice, Y should upper bound the effects of both $stmt_1$ and $stmt_2$ on X. This guarantees that the type information is valid independent of which branch is chosen. For iterations, we have to make sure Y is an upper bound for the effect of arbitrarily many applications of $stmt$ to X. This means the environment in Y is at least X because the iteration may be skipped. Moreover, if we apply $stmt$ to Y then we should again obtain at most the environment in Y.

It remains to define $sp(X, com)$, the strongest enriched type environment resulting from X under command com. We refer to the typing rules in Figure 6 and extract pre_{com} and up_{com}. The former is a predicate on environments capturing the premise of the rule associate with command com. To give an example, for Rule (ASSIGN2) the predicate $pre_{com}(\Gamma)$ is $isValid(T)$ with $T = \Gamma(q)$. The latter is a function on environments. It captures the update of the given environment as defined in the consequence of the corresponding rule. For (ASSIGN2), the update $up_{com}(\Gamma)$ is $\Gamma[p \mapsto \varnothing]$. The strongest enriched environment preserves the information that a type inference has failed, $sp(\top, com) := \top$, for all commands. For a given environment, we set

$$sp(\Gamma, com) \;:=\; pre_{com}(\Gamma) \;?\; up_{com}(\Gamma) \;:\; \top\,.$$

We evaluate the premise of the rule. If it does not hold, the type inference will fail and return $\top$. Otherwise, we determine the update of the current type environment, $up_{com}(\Gamma)$. We rely on the fact that $sp(\cdot, com)$ is monotonic and hence (as the domains are finite) continuous.

We apply a Kleene iteration to obtain the least solution to the constraint system $\Phi(\Gamma_{init}, stmt, X)$. The least solution is a function $lsol$ that assigns to each variable in the system an enriched type environment. We focus on variable X that captures the effect of the overall program on the initial type environment. Then $lsol(X)$ is the strongest type environment that can be obtained by a successful type inference. This is the key correspondence.

PROPOSITION 7.2 (PRINCIPLE TYPES). *Consider $\Phi(\Gamma_{init}, stmt, X)$. Then $lsol(X) = \bigsqcap_{\vdash \{\,\Gamma_{init}\,\}\ stmt\ \{\,\Gamma\,\}} \Gamma$. Hence, $lsol(X) \neq \top$ if and only if $\vdash stmt$.*

It remains to check the complexity of the Kleene iteration. In the lattice of enriched type environments, chains have length at most $|Var| \cdot |\{\mathbb{A}, \mathbb{L}, \mathbb{S}, \mathbb{E}_1, \ldots, \mathbb{E}_n\}| + 1$. This is linear in the size of the program as the guarantees only depend on the SMR algorithm, which is not part of the input. With one variable for each program point, also the number of variables in the constraint system is linear in the size of the program. It remains to compute $sp(\cdot, com)$ for the Kleene approximants. This can be done in constant time. The premise and the update of a rule only modify a constant number of variables. Moreover, we can look-up the effect of commands on a type in constant time. Combined, we obtain the overall quadratic complexity.

$$
\begin{aligned}
inst(stmt^*) &:= inst(stmt)^* & inst(\texttt{enter}\ func(\bar{p}, \bar{u})) &:= \texttt{skip}\\
inst(stmt_1 \oplus stmt_2) &:= inst(stmt_1) \oplus inst(stmt_2) & inst(\texttt{exit}\ func) &:= \texttt{skip}\\
inst(stmt_1;\ stmt_2) &:= inst(stmt_1);\ inst(stmt_2) & inst(\texttt{@inv}\ p = q) &:= \texttt{assert}\ p = q\\
inst(com) &:= com
\end{aligned}
$$

$$
\begin{aligned}
inst(\texttt{enter retire}(q)) &:= \texttt{skip} \oplus \big(\texttt{retire\_ptr} := q;\ \texttt{retire\_flag} := \texttt{true}\big)\\
inst(\texttt{@inv active}(p)) &:= \texttt{assert !retire\_flag} \vee \texttt{retire\_ptr} \neq p\\
inst(\texttt{@inv angel}\ r) &:= \texttt{havoc}(r);\ \texttt{included\_}r := \texttt{false};\ \texttt{failed\_}r := \texttt{false}\\
inst(\texttt{@inv}\ q\ \texttt{in}\ r) &:= \texttt{skip} \oplus \big(\texttt{assume}\ q = r;\ \texttt{assert !failed\_}r;\ \texttt{included\_}r := \texttt{true}\big)\\
inst(\texttt{@inv active}(r)) &:= \texttt{skip} \oplus \big(\texttt{assume retire\_flag} \wedge \texttt{retire\_ptr} = r;\\
&\qquad\qquad \texttt{assert !included\_}r;\ \texttt{failed\_}r := \texttt{true}\big)
\end{aligned}
$$

Fig. 11. Source-to-source translation replacing SMR commands and invariant annotations.

8 INVARIANT CHECKING

The type system from Section 5 relies on invariant annotations in the program under scrutiny in order to incorporate runtime behavior that is typically not available to a type system. For the soundness of our approach, we require those annotations to be correct. Recall from Section 5 that the annotations need only hold in the garbage collected (GC) semantics. We now show how to use an off-the-shelf GC verifier to discharge the invariant annotations fully automatically. In our experiments, we rely on CAVE [Vafeiadis 2009, 2010a,b].

Making the link to tools is non-trivial. Our programs feature programming constructs that are typically not available in off-the-shelf verifiers. We present a source-to-source translation that replaces those constructs by standard ones. The constructs to be replaced are SMR commands, invariants guaranteeing pointers to be active (not retired), and invariants centered around angels. For the translation, we only rely on ordinary assertions `assert` *cond* and non-deterministic assignments $\texttt{havoc}(p)$ to pointers. Both are usually available in verification tools.

The correspondence between the original program P and its translation $inst(P)$ is documented in Theorem 8.1 and as required. Predicate $safe(\cdot)$ evaluates to true iff the assertions hold, i.e., verification is successful. Recall that $[\![P]\!]^{\varnothing}_{\varnothing}$ is the GC semantics where addresses are neither freed nor reclaimed. Note that this semantics is the weakest a tool can assume. Our instrumentation also works if the GC tool collects and subsequently reuses garbage nodes.

THEOREM 8.1 (SOUNDNESS AND COMPLETENESS). *We have* $inv([\![P]\!]^{\varnothing}_{\varnothing})$ *iff* $safe([\![inst(P)]\!]^{\varnothing}_{\varnothing})$. *The source-to-source translation is linear in size.*

The source-to-source translation is defined in Figure 11. It preserves the structure of the program and does not modify ordinary commands. SMR calls and returns will be taken care of by the type system. They are ignored, except for retire. Invariants guaranteeing pointer equality yield assertions.

The purpose of invariants $\texttt{@inv active}(p)$ is to guarantee that the address held by the pointer has not been retired since its last allocation. The idea of our translation is to guess the moment of failure, the retire function after which such an invariant will be checked. We instrument the program by an additional pointer `retire_ptr` and a Boolean variable `retire_flag`. Both are shared. A retire translates into a non-deterministic choice between skipping the command or being the retire after which an invariant will fail. In the latter case, the address is stored in `retire_ptr` and `retire_flag` is raised. Note that the instrumentation is tailored towards garbage collection. As long as `retire_ptr` points to the address, it will not be reallocated. Therefore, we do not run the risk of the address becoming active ever again. The invariant $\texttt{@inv active}(p)$ now translates into an

assertion that checks the address of p for being the retired one and the flag for being raised. A thing to note is that the instrumentation of the retire function is not atomic. Hence, there may be an interleaving where a pointer has been stored in `retire_ptr` but the flag has not yet been raised. The assertion would consider this interleaving safe. However, if there is such an interleaving, there is also one where the assertion fails. Hence, atomicity is not needed.

For invariants involving angels, the idea of the instrumentation is the same as for pointers, guessing the moment of failure. What makes the task more difficult is the angelic semantics. We cannot just guess a value for the angel and show that it makes an invariant fail. Instead, we have to show that, no matter how the value is chosen, it inevitably leads to an invariant failure. This resembles the idea of having a strategy to win against an opponent in a turn-based game, a common phenomenon when quantifier alternation is involved [Grädel et al. 2002]. Another source of difficulty is the fact that angels are second-order variables storing sets. We tackle the problem by guessing an element in the set for which verification fails.

The instrumentation proceeds as follows. We consider angels r to be ordinary pointers. For each angel, we add two Boolean variables `included_`r and `failed_`r that are local to the thread. When we allocate an angel using `@inv angel` r, we guess the address that (i) will inevitably belong to the set of addresses held by the angel and (ii) for which an active invariant will fail. To document that we are sure of (i), we raise flag `included_`r. For (ii), we use `failed_`r. If we are sure of both facts, we let verification fail. Note that we can derive the facts in arbitrary order.

An invariant `@inv` q `in` r forces the angel to contain the address of q. This may establish (i). The reason it does not establish (i) for sure is that the angel denotes a set of addresses, and the address of q could be different from the one for which an active invariant fails. Hence, we non-deterministically choose between skipping the invariant or comparing q to r. If the comparison succeeds, we raise `included_`r. Moreover, we check (ii). If the address has been retired, we report a bug.

Invariant `@inv active(`r`)` forces all addresses held by the angel to be active. In the instrumented program, r is a pointer that we compare to `retire_ptr` introduced above. If the address has been retired, we are sure about (ii) and document this by raising `failed_`r. If we already know (i), the address inevitably belongs to the set held by the angel, verification fails.

9 EVALUATION

We implemented our approach in a C++ tool called SEAL.[3]. As stated before, we use the state-of-the-art tool CAVE [Vafeiadis 2009, 2010a,b] as a back-end verifier for discharging annotations and checking linearizability. For the type inference, our tool computes the most precise guarantees $\mathbb{E}_L$ on-the-fly; there is no need for the user to manually specify them. To substantiate the claim of usefulness of our approach, we evaluated SEAL on examples from the literature. We considered the following data structures: Treiber's stack [Michael 2002b; Treiber 1986], Michael&Scott's lock-free queue [Michael 2002b; Michael and Scott 1996], the DGLM queue [Doherty et al. 2004], the Vechev&Yahav CAS set [Vechev and Yahav 2008], the Vechev&Yahav DCAS set [Vechev and Yahav 2008], the ORVYY set [O'Hearn et al. 2010], and Michael's set [Michael 2002a]. Our benchmarks include a version of each data structure for hazard pointers (HP) [Michael 2002b] and epoch-based reclamation (EBR) [Fraser 2004]. We adapted the GC implementations of the Vechev&Yahav DCAS set, the Vechev&Yahav CAS set, and the ORVYY set given in the literature to use HP/EBR.

Our findings are listed in Table 1. The experiments were conducted on an Intel i5-8600K@3.6GHz with 16GB of RAM. The table includes the time taken (i) for the type inference, (ii) for discharging the invariant annotations, and (iii) to check linearizability. We mark tasks with ✓ if they were successful, with ✗ if they failed, and with ◔ if they timed out after 12h wall time.

[3] Available at: https://wolff09.github.io/seal/

Table 1. Experimental results for verifying singly-linked data structures using safe memory reclamation. The experiments were conducted on an Intel i5-8600K@3.6GHz with 16GB of RAM.

SMR	Program	Type Inference	Annotations	Linearizability
HP	Treiber's stack	0.7*s* ✓	12*s* ✓	1*s* ✓
	Opt. Treiber's stack	0.5*s* ✓	11*s* ✓	1*s* ✓
	Michael&Scott's queue	0.6*s* ✓	12*s* ✓	4*s* ✓
	DGLM queue	0.6*s* ✓	1*s* ✗[a]	5*s* ✓
	Vechev&Yahav DCAS set	1.2*s* ✓	13*s* ✓	98*s* ✓
	Vechev&Yahav CAS set	1.2*s* ✓	3.5*h* ✓	42*m* ✓
	ORVYY set	1.2*s* ✓	3.2*h* ✓	47*m* ✓
	Michael's set	1.2*s* ✓	90*s* ✗[a]	— ◔
EBR	Treiber's stack	0.6*s* ✓	10*s* ✓	1*s* ✓
	Michael&Scott's queue	0.7*s* ✓	16*s* ✓	5*s* ✓
	DGLM queue	0.7*s* ✓	1*s* ✗[a]	6*s* ✓
	Vechev&Yahav DCAS set	0.8*s* ✓	38*s* ✓	200*s* ✓
	Vechev&Yahav CAS set	0.8*s* ✓	7*h* ✓	42*m* ✓
	ORVYY set	0.9*s* ✓	7*h* ✓	47*m* ✓
	Michael's set	0.2*s* ✓	22*s* ✗[a]	— ◔

[a]False-positive due to imprecision in the back-end verifier.

Our approach is capable of verifying most of the lock-free data structures we considered. Comparing the total runtime with our competitors [Meyer and Wolff 2019a], the only other approach capable of handling lock-free data structures with general SMR algorithms, we experience a speed-up of over two orders of magnitude on examples like Michael&Scott's queue. Besides the speed-up, we are the first to automatically verify lock-free set algorithms that use SMR.

We were not able to discharge the annotations of the DGLM queue and Michael's set. Imprecision in the thread-modular abstraction of our back-end verifier resulted in false-positives being reported. Hence, we cannot guarantee the soundness of our analysis in these cases. This is no limitation of our approach, it is a shortcoming of the back-end verifier. Meyer and Wolff [2019a] reported a similar issue that they solved by manually providing hints to improve the precision of their analysis.

The annotation checks for set implementations are interesting. While the HP version of an implementation is typically more involved than the corresponding version using EBR, the annotation checks for the HP version are more efficient. The reason for this could be that EBR implementations require angels. The conjecture suggests that discharging angels is harder for CAVE than discharging active annotations although our instrumentation uses the same idea for both annotation types.

For the benchmarks from Table 1 we preprocessed the implementations by applying mover types [Lipton 1975], a well-known program transformation (cf. Section 10). Intuitively, a command is a mover if it can be reordered with commands of other threads. This allows for the command to be moved to the next command of the same thread, effectively constructing an atomic block containing the mover and the next command. What is remarkable in our setting is that SMR commands (`enter`, `exit`, `free`) always move over ordinary memory commands, and vice versa. (Technically, this requires `enter` commands to contain only thread-local variables, a property than be checked/established easily.) As a result, we can find movers for memory commands using existing techniques. For SMR commands, movers can be read of the SMR automaton. Our tool is able to find and apply movers. Due to space constraints, we omit a thorough discussion of the matter.

10 RELATED WORK

Safe Memory Reclamation. Besides EBR and HP there is another basic SMR technique: reference counting (RC). RC extends records with an integer field counting the number of pointers to the record. Safely modifying counters in a lock-free manner, however, requires hazard pointers [Herlihy et al. 2005] or a mostly unavailable CAS for two arbitrary memory locations [Detlefs et al. 2001].

Recent efforts in developing SMR algorithms have mostly combined existing SMR techniques. For example, *DEBRA* [Brown 2015] is an optimized EBR implementation. Harris [2001] modifies EBR to store epochs inside records. *Hyaline* [Nikolaev and Ravindran 2019] is used like EBR. Optimized HP implementations include the work by Aghazadeh et al. [2014], the work by Dice et al. [2016], and *Cadence* [Balmau et al. 2016]. *Dynamic Collect* [Dragojevic et al. 2011], *StackTrack* [Alistarh et al. 2014], and *ThreadScan* [Alistarh et al. 2015] are HP-esque implementations exploring the use of operating system and hardware support. *Drop the Anchor* [Braginsky et al. 2013], *Optimistic Access* [Cohen and Petrank 2015b], *Automatic Optimistic Access* [Cohen and Petrank 2015a], *QSense* [Balmau et al. 2016], *Hazard Eras* [Ramalhete and Correia 2017], and *Interval-based Reclamation* [Wen et al. 2018] combine EBR and HP. *Free Access* [Cohen 2018] automates the application of Automatic Optimistic Access. While the method promises to be correct by construction, we believe that performance-critical applications choose the SMR technique based on performance rather than ease of use. The demand for automated verification remains. *Beware&Cleanup* [Gidenstam et al. 2005] combines HP and RC. *Isolde* [Yang and Wrigstad 2017] combines EBR and RC. We believe our approach can handle other SMR algorithms besides EBR and HP as well.

Memory Safety. We use our type system to show that a program is free from pointer races, meaning that it is memory safe. There are a number of related tools that can check pointer programs for memory safety. For example: a combination of CCURED [Necula et al. 2002] and BLAST [Henzinger et al. 2003] due to Beyer et al. [2005], INVADER [Yang et al. 2008], XISA [Laviron et al. 2010], SLAYER [Berdine et al. 2011], INFER [Calcagno and Distefano 2011], FORESTER [Holík et al. 2013], PREDATOR [Dudka et al. 2013; Holík et al. 2016], and APROVE [Ströder et al. 2017]. These tools can only handle sequential code. Moreover, unlike our type system, they include memory/shape abstractions to identify unsafe pointer operations. We delegate this task to a back-end verifier with the help of annotations. That is, if the related tools were to support concurrent programs, they were candidates for the back-end. We used CAVE [Vafeiadis 2010a,b] as it can also prove linearizability.

Despite the differences, we point out that the combination of BLAST and CCURED [Beyer et al. 2005] is close to our approach in spirit. CCURED performs a type check of the program under scrutiny which checks for unsafe memory operations. While doing so, it annotates pointer operations in the program with run-time checks in case the type check could not establish the operation to be safe. The run-time checks are then discharged using BLAST. The approach is limited to sequential programs. Moreover, we incorporate the behavior of the SMR. Finally, our type system is more lightweight and we discharge the invariants in a simpler semantics without memory deletions.

Castegren and Wrigstad [2017] give a type system that guarantees the absence of data races. Types encode a notion of ownership that prevents non-owning threads from accessing a node. Their method is tailored towards GC and requires to rewrite programs with appropriate type specifiers. Recently, Kuru and Gordon [2019] presented a type system for checking the correct use of RCU. Unlike our approach, they integrate a fixed shape analysis and a fixed RCU specification. This makes the type system considerably more complicated and the type check potentially more expensive. Unfortunately, Kuru and Gordon [2019] did not implement their approach.

Besides memory safety, tools like INVADER, SLAYER, INFER, FORESTER, PREDATOR, and the type system by Kuru and Gordon [2019] discover memory leaks. A successful type check with our type system does not imply the absence of memory leaks. We believe that the outcome of our analysis

could help a leak detection tool. For example, by performing a linearizability check to find the abstract data type the data structure under consideration implements. We consider this as future work.

Typestate. Typestate [Strom and Yemini 1986] extents an object's type to carry a notion of state. The methods of an object can be annotated to modify this state and to be available only in a certain state. Existing analyses checking for methods being called only in the appropriate state include [Bierhoff and Aldrich 2007; DeLine and Fähndrich 2004; Fähndrich and DeLine 2002; Fink et al. 2006; Foster et al. 2002]. Our types can be understood as typestates for pointers (and the objects they reference) geared towards SMR. However, whereas an object's typestate has a global character, our types reflect a thread's local perception. Das et al. [2002] give a typestate analysis based on symbolic execution to increase precision. Similarly, we increase the applicability of our approach by using annotations that are discharged by a back-end verifier. For a more detailed overview on typestate, refer to [Ancona et al. 2016].

Program Logics. There are several program logics for verifying concurrent programs with heap. Examples are: SAGL [Feng et al. 2007], RGSEP [Vafeiadis and Parkinson 2007] (used by CAVE [Vafeiadis 2010b]), LRG [Feng 2009], Deny-Guarantee [Dodds et al. 2009], CAP [Dinsdale-Young et al. 2010], HLRG [Fu et al. 2010], and the work by Gotsman et al. [2013]. Program logics are conceptually related to our type system. However, such logics integrate further ingredients to successfully verify intricate lock-free data structures [Turon et al. 2014]. Most importantly, they include memory abstractions, like (concurrent) separation logic [Brookes 2004; O'Hearn 2004; O'Hearn et al. 2001; Reynolds 2002], and mechanisms to reason about thread interference, like rely-guarantee [Jones 1983]. This makes them much more complex than our type system. We deliberately avoid incorporating a memory abstraction into our type system to keep it as flexible as possible. Instead, we use annotations to delegate the shape analysis to a back-end verifier, achieving modularity in verifying the data structure and its memory management separately. Moreover, accounting for thread interference in our type system boils down to defining guarantees as closed sets of locations and removing guarantee $\mathbb{A}$ upon exiting atomic blocks.

Oftentimes, invariant-based reasoning about interference turns out too restrictive for verification. To overcome this, program logics like CARESL [Turon et al. 2013], FCSL [Nanevski et al. 2014], ICAP [Svendsen and Birkedal 2014], TADA [da Rocha Pinto et al. 2014], GPS [Turon et al. 2014], and IRIS [Jung et al. 2015] make use of protocols. A protocol captures possible thread interference, for example, using state transition systems. (Rely-guarantee is a particular instantiation of a protocol [Jung et al. 2015; Turon et al. 2013].) In our approach, SMR automata are protocols that govern memory deletions and protections, that is, describe the influence of SMR-related actions among threads. Our types describe a thread's local, per-pointer perception of that global protocol.

Besides protocols, recent logics like CARESL, TADA, and IRIS integrate reasoning in the spirit of atomicity abstraction/refinement [Dijkstra 1982; Lipton 1975]. Intuitively, they allow the client of a fine-grained module to be verified against a coarse-grained specification of the module. For example, a client of a data structure can be verified against its abstract data type, provided the data structure refines the abstract data type. Following [Meyer and Wolff 2019a], we use the same idea wrt. SMR algorithms: we consider SMR automata instead of the actual SMR implementations.

Some program logics can also unveil memory leaks [Bizjak et al. 2019; Gotsman et al. 2013].

Linearizability. Linearizability testing [Burckhardt et al. 2010; Cerný et al. 2010; Emmi and Enea 2018; Emmi et al. 2015; Horn and Kroening 2015; Liu et al. 2009, 2013; Lowe 2017; Travkin et al. 2013; Vechev and Yahav 2008; Yang et al. 2017; Zhang 2011] is a bug hunting technique to find non-linearizable executions in large code bases. Since we focus on verification, we do not go into the details of linearizability testing. However, it could be worthwhile to use a linearizability

tester instead of a verification back-end in our approach to provide faster feedback during the development process and only use a verifier once the development is considered finished.

Verification techniques for linearizability fall into two categories: manual techniques (including tool-supported but not fully automated techniques) and automatic techniques. Manual approaches require the human checker to have a deep understanding of the proof techniques as well as the program under scrutiny—in our case, this includes a deep understanding of the lock-free data structure as well as the SMR implementation. This may be the reason why many manual proofs do not consider reclamation [Bäumler et al. 2011; Bouajjani et al. 2017; Colvin et al. 2005, 2006; Delbianco et al. 2017; Derrick et al. 2011; Doherty and Moir 2009; Elmas et al. 2010; Groves 2007, 2008; Hemed et al. 2015; Henzinger et al. 2013; Jonsson 2012; Khyzha et al. 2017; Liang and Feng 2013; Liang et al. 2012, 2014; O'Hearn et al. 2010; Schellhorn et al. 2012; Sergey et al. 2015a,b]. There are fewer works that consider reclamation [Dodds et al. 2015; Doherty et al. 2004; Fu et al. 2010; Gotsman et al. 2013; Krishna et al. 2018; Parkinson et al. 2007; Tofan et al. 2011]. (The work by Gotsman et al. [2013] checks memory safety and discovers memory leaks as well.) For a more detailed overview of manual techniques, we refer to the survey by Dongol and Derrick [2015].

The landscape of related work for automated linearizability proofs is similar to its manual counterpart. Most automated approaches ignore memory reclamation, that is, assume a garbage collector [Abdulla et al. 2016; Amit et al. 2007; Berdine et al. 2008; Segalov et al. 2009; Sethi et al. 2013; Vafeiadis 2010a,b; Vechev et al. 2009; Zhu et al. 2015]. When reclamation is not considered, memory abstractions are simpler and more efficient, they can exploit ownership guarantees [Bornat et al. 2005; Boyland 2003] and the resulting thread-local reasoning techniques [O'Hearn et al. 2001; Reynolds 2002]. Few works [Abdulla et al. 2013; Haziza et al. 2016; Holík et al. 2017; Meyer and Wolff 2019a] address the challenge of verifying lock-free data structures under manual memory management. Besides Meyer and Wolff [2019a], they use hand-crafted semantics that allow for accessing deleted memory. The work by Meyer and Wolff [2019a] is the closest related. We build on their programming model and their reduction result as discussed in Sections 3 and 4, respectively. Moreover, we rely to their results for proving an SMR implementation against an SMR automaton.

Moverness. Movers where first introduced by Lipton [1975]. They were later generalized to arbitrary safety properties [Back 1989; Doeppner 1977; Lamport and Schneider 1989]. Movers are a widely applied enabling technique for verification. To ease the verification task, the program is made *more atomic* without cutting away behavior. Because we use standard moverness arguments, we do not give an extensive overview. Flanagan et al. [2008]; Flanagan and Qadeer [2003] use a type system to find movers in Java programs. The CALVIN tool [Flanagan et al. 2005, 2002; Freund and Qadeer 2004] applies movers to establish pre/post conditions of functions in concurrent programs using sequential verifiers. Similarly, QED [Elmas et al. 2009] rewrites concurrent code into sequential code based on movers. These approaches are similar to ours in spirit: they take the verification task to a much simpler semantics. However, movers are not a key aspect of our approach. We employ them only to increase the applicability of our tool in case of benign pointer races. Elmas et al. [2010] extend QED to establish linearizability for simple lock-free data structures. QED is superseded by CIVL [Hawblitzel et al. 2015; Kragl and Qadeer 2018]. CIVL proves programs correct by repeatedly applying movers to a program until its specification is obtained. The approach is semi-automatic, it takes as input a so-called layered program that contains intermediary steps guiding the transformation [Kragl and Qadeer 2018]. Movers were also applied in the context of relaxed memory [Bouajjani et al. 2018].

ACKNOWLEDGMENTS

We thank the POPL'20 reviewers for their valuable feedback and suggestions for improvements.

REFERENCES

Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2013. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. In *TACAS (LNCS)*, Vol. 7795. Springer, 324–338. https://doi.org/10.1007/978-3-642-36742-7_23

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2016. Automated Verification of Linearization Policies. In *SAS (LNCS)*, Vol. 9837. Springer, 61–83. https://doi.org/10.1007/978-3-662-53413-7_4

Zahra Aghazadeh, Wojciech M. Golab, and Philipp Woelfel. 2014. Making Objects Writable. In *PODC*. ACM, 385–395. https://doi.org/10.1145/2611462.2611483

Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. 2014. StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation. In *EuroSys*. ACM, 25:1–25:14. https://doi.org/10.1145/2592798.2592808

Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. 2015. ThreadScan: Automatic and Scalable Memory Reclamation. In *SPAA*. ACM, 123–132. https://doi.org/10.1145/2755573.2755600

Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. 2007. Comparison Under Abstraction for Verifying Linearizability. In *CAV (LNCS)*, Vol. 4590. Springer, 477–490. https://doi.org/10.1007/978-3-540-73368-3_49

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

Ralph-Johan Back. 1989. A Method for Refining Atomicity in Parallel Algorithms. In *PARLE (LNCS)*, Vol. 366. Springer, 199–216. https://doi.org/10.1007/3-540-51285-3_42

Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *SPAA*. ACM, 349–359. https://doi.org/10.1145/2935764.2935790

Simon Bäumler, Gerhard Schellhorn, Bogdan Tofan, and Wolfgang Reif. 2011. Proving linearizability with temporal logic. *Formal Asp. Comput.* 23, 1 (2011), 91–112. https://doi.org/10.1007/s00165-009-0130-y

Josh Berdine, Byron Cook, and Samin Ishtiaq. 2011. SLAyer: Memory Safety for Systems-Level Code. In *CAV (LNCS)*, Vol. 6806. Springer, 178–183. https://doi.org/10.1007/978-3-642-22110-1_15

Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. 2008. Thread Quantification for Concurrent Shape Analysis. In *CAV (LNCS)*, Vol. 5123. Springer, 399–413. https://doi.org/10.1007/978-3-540-70545-1_37

Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2005. Checking Memory Safety with Blast. In *FASE (LNCS)*, Vol. 3442. Springer, 2–18. https://doi.org/10.1007/978-3-540-31984-9_2

Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *OOPSLA*. ACM, 301–320. https://doi.org/10.1145/1297027.1297050

Garrett Birkhoff. 1948. *Lattice Theory (revised edition)*. American Mathematical Society.

Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing Obligations in Higher-order Concurrent Separation Logic. *PACMPL* 3, POPL (2019), 65:1–65:30. https://doi.org/10.1145/3290378

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission Accounting in Separation Logic. In *POPL*. ACM, 259–270. https://doi.org/10.1145/1040305.1040327

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving Linearizability Using Forward Simulations. In *CAV (LNCS)*, Vol. 10427. Springer, 542–563. https://doi.org/10.1007/978-3-319-63390-9_28

Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. 2018. Reasoning About TSO Programs Using Reduction and Abstraction. In *CAV (LNCS)*, Vol. 10982. Springer, 336–353. https://doi.org/10.1007/978-3-319-96142-2_21

John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (LNCS)*, Vol. 2694. Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the Anchor: Lightweight Memory Management for Non-blocking Data Structures. In *SPAA*. ACM, 33–42. https://doi.org/10.1145/2486159.2486184

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR (LNCS)*, Vol. 3170. Springer, 16–34. https://doi.org/10.1007/978-3-540-28644-8_2

Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *PODC*. ACM, 261–270. https://doi.org/10.1145/2767386.2767436

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: A Complete and Automatic Linearizability Checker. In *PLDI*. ACM, 330–340. https://doi.org/10.1145/1806596.1806634

Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (LNCS)*, Vol. 6617. Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

Elias Castegren and Tobias Wrigstad. 2017. Relaxed Linear References for Lock-free Data Structures. In *ECOOP (LIPIcs)*, Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 6:1–6:32. https://doi.org/10.4230/LIPIcs.ECOOP.2017.6

Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. 2010. Model Checking of Linearizability of Concurrent List Implementations. In *CAV (LNCS)*, Vol. 6174. Springer, 465–479. https://doi.org/10.1007/978-3-642-14295-6_41

Nachshon Cohen. 2018. Every Data Structure Deserves Lock-free Memory Reclamation. *PACMPL* 2, OOPSLA (2018), 143:1–143:24. https://doi.org/10.1145/3276513

Nachshon Cohen and Erez Petrank. 2015a. Automatic Memory Reclamation for Lock-free Data Structures. In *OOPSLA*. ACM, 260–279. https://doi.org/10.1145/2814270.2814298

Nachshon Cohen and Erez Petrank. 2015b. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *SPAA*. ACM, 254–263. https://doi.org/10.1145/2755573.2755579

Robert Colvin, Simon Doherty, and Lindsay Groves. 2005. Verifying Concurrent Data Structures by Simulation. *Electr. Notes Theor. Comput. Sci.* 137, 2 (2005), 93–110. https://doi.org/10.1016/j.entcs.2005.04.026

Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. 2006. Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In *CAV (LNCS)*, Vol. 4144. Springer, 475–488. https://doi.org/10.1007/11817963_44

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A New Type Assignment for λ-Terms. *Arch. Math. Log.* 19, 1 (1978), 139–156. https://doi.org/10.1007/BF02011875

Karl Crary, David Walker, and J. Gregory Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *POPL*. ACM, 262–275. https://doi.org/10.1145/292540.292564

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP (LNCS)*, Vol. 8586. Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time. In *PLDI*. ACM, 57–68. https://doi.org/10.1145/512529.512538

Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent Data Structures Linked in Time. In *ECOOP (LIPIcs)*, Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.8

Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *ECOOP (LNCS)*, Vol. 3086. Springer, 465–490. https://doi.org/10.1007/978-3-540-24851-4_21

John Derrick, Gerhard Schellhorn, and Heike Wehrheim. 2011. Mechanically Verified Proof Obligations for Linearizability. *ToPLaS* 33, 1 (2011), 4:1–4:43. https://doi.org/10.1145/1889997.1890001

David Detlefs, Paul Alan Martin, Mark Moir, and Guy L. Steele Jr. 2001. Lock-free Reference Counting. In *PODC*. ACM, 190–199. https://doi.org/10.1145/383962.384016

Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast Non-intrusive Memory Reclamation for Highly-concurrent Data Structures. In *ISMM*. ACM, 36–45. https://doi.org/10.1145/2926697.2926699

Edsger W. Dijkstra. 1982. *On Making Solutions More and More Fine-Grained.* Springer New York, New York, NY, 292–307. https://doi.org/10.1007/978-1-4612-5695-3_53

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP (LNCS)*, Vol. 6183. Springer, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *ESOP (LNCS)*, Vol. 5502. Springer, 363–377. https://doi.org/10.1007/978-3-642-00590-9_26

Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. In *POPL*. ACM, 233–246. https://doi.org/10.1145/2676726.2676963

Thomas W. Doeppner, Jr. 1977. Parallel Program Correctness Through Refinement. In *POPL*. ACM, 155–169. https://doi.org/10.1145/512950.512965

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2004. Formal Verification of a Practical Lock-Free Queue Algorithm. In *FORTE (LNCS)*, Vol. 3235. Springer, 97–114. https://doi.org/10.1007/978-3-540-30232-2_7

Simon Doherty and Mark Moir. 2009. Nonblocking Algorithms and Backward Simulation. In *DISC (LNCS)*, Vol. 5805. Springer, 274–288. https://doi.org/10.1007/978-3-642-04355-0_28

Brijesh Dongol and John Derrick. 2015. Verifying Linearisability: A Comparative Survey. *ACM Comput. Surv.* 48 (2015). https://doi.org/10.1145/2796550

Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. On the Power of Hardware Transactional Memory to Simplify Memory Management. In *PODC*. ACM, 99–108. https://doi.org/10.1145/1993806.1993821

Kamil Dudka, Petr Peringer, and Tomás Vojnar. 2013. Byte-Precise Verification of Low-Level List Manipulation. In *SAS (LNCS)*, Vol. 7935. Springer, 215–237. https://doi.org/10.1007/978-3-642-38856-9_13

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying Linearizability Proofs with Reduction and Abstraction. In *TACAS (LNCS)*, Vol. 6015. Springer, 296–311. https://doi.org/10.1007/978-3-642-12002-2_25

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A Calculus of Atomic Actions. In *POPL*. ACM, 2–15. https://doi.org/10.1145/1480881.1480885

Michael Emmi and Constantin Enea. 2018. Sound, Complete, and Tractable Linearizability Monitoring for Concurrent Collections. *PACMPL* 2, POPL (2018), 25:1–25:27. https://doi.org/10.1145/3158113

Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Monitoring Refinement via Symbolic Reasoning. In *PLDI*. ACM, 260–269. https://doi.org/10.1145/2737924.2737983

Manuel Fähndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*. ACM, 13–24. https://doi.org/10.1145/512529.512532

Xinyu Feng. 2009. Local Rely-guarantee Reasoning. In *POPL*. ACM, 315–327. https://doi.org/10.1145/1480881.1480922

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *ESOP (LNCS)*, Vol. 4421. Springer, 173–188. https://doi.org/10.1007/978-3-540-71316-6_13

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective Typestate Verification in the Presence of Aliasing. In *ISSTA*. ACM, 133–144. https://doi.org/10.1145/1146238.1146254

Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. 2008. Types for Atomicity: Static Checking and Inference for Java. *ToPLaS* 30, 4 (2008), 20:1–20:53. https://doi.org/10.1145/1377492.1377495

Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. 2005. Modular verification of multithreaded programs. *Theor. Comput. Sci.* 338, 1-3 (2005), 153–183. https://doi.org/10.1016/j.tcs.2004.12.006

Cormac Flanagan and Shaz Qadeer. 2003. A Type and Effect System for Atomicity. In *PLDI*. ACM, 338–349. https://doi.org/10.1145/781131.781169

Cormac Flanagan, Shaz Qadeer, and Sanjit A. Seshia. 2002. A Modular Checker for Multithreaded Programs. In *CAV (LNCS)*, Vol. 2404. Springer, 180–194. https://doi.org/10.1007/3-540-45657-0_14

Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. 2002. Flow-Sensitive Type Qualifiers. In *PLDI*. ACM, 1–12. https://doi.org/10.1145/512529.512531

Keir Fraser. 2004. *Practical Lock-freedom.* Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193

Stephen N. Freund and Shaz Qadeer. 2004. Checking Concise Specifications for Multithreaded Software. *Journal of Object Technology* 3, 6 (2004), 81–101. https://doi.org/10.5381/jot.2004.3.6.a4

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR (LNCS)*, Vol. 6269. Springer, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. 2005. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. In *ISPAN*. IEEE, 202–207. https://doi.org/10.1109/ISPAN.2005.42

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *ESOP (LNCS)*, Vol. 7792. Springer, 249–269. https://doi.org/10.1007/978-3-642-37036-6_15

Erich Grädel, Wolfgang Thomas, and Thomas Wilke (Eds.). 2002. *Automata, Logics, and Infinite Games*. LNCS, Vol. 2500. Springer. https://doi.org/10.1007/3-540-36387-4

Lindsay Groves. 2007. Reasoning about Nonblocking Concurrency using Reduction. In *ICECCS*. IEEE, 107–116. https://doi.org/10.1109/ICECCS.2007.39

Lindsay Groves. 2008. Verifying Michael and Scott's Lock-Free Queue Algorithm using Trace Reduction. In *CATS (CRPIT)*, Vol. 77. Australian Computer Society, 133–142. http://crpit.com/abstracts/CRPITV77Groves.html

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC (LNCS)*, Vol. 2180. Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *CAV (LNCS)*, Vol. 9207. Springer, 449–465. https://doi.org/10.1007/978-3-319-21668-3_26

Frédéric Haziza, Lukás Holík, Roland Meyer, and Sebastian Wolff. 2016. Pointer Race Freedom. In *VMCAI (LNCS)*, Vol. 9583. Springer, 393–412. https://doi.org/10.1007/978-3-662-49122-5_19

Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular Verification of Concurrency-Aware Linearizability. In *DISC (LNCS)*, Vol. 9363. Springer, 371–387. https://doi.org/10.1007/978-3-662-48653-5_25

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. Software Verification with BLAST. In *SPIN (LNCS)*, Vol. 2648. Springer, 235–239. https://doi.org/10.1007/3-540-44829-2_17

Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-Oriented Linearizability Proofs. In *CONCUR (LNCS)*, Vol. 8052. Springer, 242–256. https://doi.org/10.1007/978-3-642-40184-8_18

Maurice Herlihy, Victor Luchangco, Paul A. Martin, and Mark Moir. 2005. Nonblocking Memory Management Support for Dynamic-sized Data Structures. *ACM Trans. Comput. Syst.* 23, 2 (2005), 146–196. https://doi.org/10.1145/1062247.1062249

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ToPLaS* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Lukás Holík, Michal Kotoun, Petr Peringer, Veronika Soková, Marek Trtík, and Tomás Vojnar. 2016. Predator Shape Analysis Tool Suite. In *HVC (LNCS)*, Vol. 10028. Springer, 202–209. https://doi.org/10.1007/978-3-319-49052-6_13

Lukás Holík, Ondrej Lengál, Adam Rogalewicz, Jirí Simácek, and Tomás Vojnar. 2013. Fully Automated Shape Analysis Based on Forest Automata. In *CAV (LNCS)*, Vol. 8044. Springer, 740–755. https://doi.org/10.1007/978-3-642-39799-8_52

Lukás Holík, Roland Meyer, Tomás Vojnar, and Sebastian Wolff. 2017. Effect Summaries for Thread-Modular Analysis - Sound Analysis Despite an Unsound Heuristic. In *SAS (LNCS)*, Vol. 10422. Springer, 169–191. https://doi.org/10.1007/978-3-319-66706-5_9

Alex Horn and Daniel Kroening. 2015. Faster Linearizability Checking via P-Compositionality. In *FORTE (LNCS)*, Vol. 9039. Springer, 50–65. https://doi.org/10.1007/978-3-319-19195-9_4

Sebastian Hunt and David Sands. 2006. On Flow-sensitive Security Types. In *POPL*. ACM, 79–90. https://doi.org/10.1145/1111037.1111045

ISO. 2011. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Standard ISO/IEC 14882:2011. International Organization for Standardization, Geneva, CH. https://www.iso.org/standard/50372.html

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ToPLaS* 5, 4 (1983), 596–619. https://doi.org/10.1145/69575.69577

Bengt Jonsson. 2012. Using refinement calculus techniques to prove linearizability. *Formal Asp. Comput.* 24, 4-6 (2012), 537–554. https://doi.org/10.1007/s00165-012-0250-7

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. 2017. Proving Linearizability Using Partial Orders. In *ESOP (LNCS)*, Vol. 10201. Springer, 639–667. https://doi.org/10.1007/978-3-662-54434-1_24

Bernhard Kragl and Shaz Qadeer. 2018. Layered Concurrent Programs. In *CAV (LNCS)*, Vol. 10981. Springer, 79–102. https://doi.org/10.1007/978-3-319-96145-3_5

Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *PACMPL* 2, POPL (2018), 37:1–37:31. https://doi.org/10.1145/3158125

Ismail Kuru and Colin S. Gordon. 2019. Safe Deferred Memory Reclamation with Types. In *ESOP (LNCS)*, Vol. 11423. Springer, 88–116. https://doi.org/10.1007/978-3-030-17184-1_4

Leslie Lamport and Fred B. Schneider. 1989. Pretending Atomicity. *SRC Research Report 44* (1989). https://www.microsoft.com/en-us/research/publication/pretending-atomicity/

Vincent Laviron, Bor-Yuh Evan Chang, and Xavier Rival. 2010. Separating Shape Graphs. In *ESOP (LNCS)*, Vol. 6012. Springer, 387–406. https://doi.org/10.1007/978-3-642-11957-6_21

Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-fixed Linearization Points. In *PLDI*. ACM, 459–470. https://doi.org/10.1145/2491956.2462189

Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A Rely-guarantee-based Simulation for Verifying Concurrent Program Transformations. In *POPL*. ACM, 455–468. https://doi.org/10.1145/2103656.2103711

Hongjin Liang, Xinyu Feng, and Ming Fu. 2014. Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations. *ToPLaS* 36, 1 (2014), 3:1–3:55. https://doi.org/10.1145/2576235

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *CACM* 18, 12 (1975), 717–721.

Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. 2009. Model Checking Linearizability via Refinement. In *FM (LNCS)*, Vol. 5850. Springer, 321–337. https://doi.org/10.1007/978-3-642-05089-3_21

Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong. 2013. Verifying Linearizability via Optimized Refinement Checking. *IEEE Trans. Software Eng.* 39, 7 (2013), 1018–1039. https://doi.org/10.1109/TSE.2012.82

Gavin Lowe. 2017. Testing for linearizability. *Concurrency and Computation: Practice and Experience* 29, 4 (2017). https://doi.org/10.1002/cpe.3928

Paul E. McKenney and John D. Slingwine. 1998. Read-copy Update: Using Execution History to Solve Concurrency Problems.

Roland Meyer and Sebastian Wolff. 2019a. Decoupling Lock-free Data Structures from Memory Reclamation for Static Analysis. *PACMPL* 3, POPL (2019), 58:1–58:31. https://doi.org/10.1145/3290371

Roland Meyer and Sebastian Wolff. 2019b. Pointer Life Cycle Types for Lock-Free Data Structures with Memory Reclamation. *CoRR* abs/1910.11714 (2019). http://arxiv.org/abs/1910.11714

Maged M. Michael. 2002a. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *SPAA*. ACM, 73–82. https://doi.org/10.1145/564870.564881

Maged M. Michael. 2002b. Safe Memory Reclamation for Dynamic Lock-free Objects Using Atomic Reads and Writes. In *PODC*. ACM, 21–30. https://doi.org/10.1145/571825.571829

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*. ACM, 267–275. https://doi.org/10.1145/248052.248106

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP (LNCS)*, Vol. 8410. Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *POPL*. ACM, 128–139. https://doi.org/10.1145/503272.503286

Ruslan Nikolaev and Binoy Ravindran. 2019. Hyaline: Fast and Transparent Lock-Free Memory Reclamation. In *PODC*. ACM, 419–421. https://doi.org/10.1145/3293611.3331575

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR (LNCS)*, Vol. 3170. Springer, 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS)*, Vol. 2142. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying Linearizability with Hindsight. In *PODC*. ACM, 85–94. https://doi.org/10.1145/1835698.1835722

Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.* 6 (1976), 319–340. https://doi.org/10.1007/BF00268134

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular Verification of a Non-blocking Stack. In *POPL*. ACM, 297–302. https://doi.org/10.1145/1190216.1190261

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *SPAA*. ACM, 367–369. https://doi.org/10.1145/3087556.3087588

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Gerhard Schellhorn, Heike Wehrheim, and John Derrick. 2012. How to Prove Algorithms Linearisable. In *CAV (LNCS)*, Vol. 7358. Springer, 243–259. https://doi.org/10.1007/978-3-642-31424-7_21

Michal Segalov, Tal Lev-Ami, Roman Manevich, Ganesan Ramalingam, and Mooly Sagiv. 2009. Abstract Transformers for Thread Correlation Analysis. In *APLAS (LNCS)*, Vol. 5904. Springer, 30–46. https://doi.org/10.1007/978-3-642-10672-9_5

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015a. Mechanized Verification of Fine-grained Concurrent Programs. In *PLDI*. ACM, 77–87. https://doi.org/10.1145/2737924.2737964

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015b. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *ESOP (LNCS)*, Vol. 9032. Springer, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Divjyot Sethi, Muralidhar Talupur, and Sharad Malik. 2013. Model Checking Unbounded Concurrent Lists. In *SPIN (LNCS)*, Vol. 7976. Springer, 320–340. https://doi.org/10.1007/978-3-642-39176-7_20

Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann. 2017. Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic. *J. Autom. Reasoning* 58, 1 (2017), 33–65. https://doi.org/10.1007/s10817-016-9389-x

Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS)*, Vol. 8410. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. 2011. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *ICTAC (LNCS)*, Vol. 6916. Springer, 239–255. https://doi.org/10.1007/978-3-642-23283-1_16

Oleg Travkin, Annika Mütze, and Heike Wehrheim. 2013. SPIN as a Linearizability Checker under Weak Memory Models. In *HVC (LNCS)*, Vol. 8244. Springer, 311–326. https://doi.org/10.1007/978-3-319-03077-7_21

R. Kent Treiber. 1986. *Systems Programming: Coping with Parallelism*. Technical Report RJ 5118. IBM.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-order Concurrency. In *ICFP*. ACM, 377–390. https://doi.org/10.1145/2544174.2500600

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *OOPSLA*. ACM, 691–707. https://doi.org/10.1145/2660193.2660243

Viktor Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *VMCAI (LNCS)*, Vol. 5403. Springer, 335–348. https://doi.org/10.1007/978-3-540-93900-9_27

Viktor Vafeiadis. 2010a. Automatically Proving Linearizability. In *CAV (LNCS)*, Vol. 6174. Springer, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

Viktor Vafeiadis. 2010b. RGSep Action Inference. In *VMCAI (LNCS)*, Vol. 5944. Springer, 345–361. https://doi.org/10.1007/978-3-642-11319-2_25

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR (LNCS)*, Vol. 4703. Springer, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

Martin T. Vechev and Eran Yahav. 2008. Deriving Linearizable Fine-grained Concurrent Objects. In *PLDI*. ACM, 125–135. https://doi.org/10.1145/1375581.1375598

Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2009. Experience with Model Checking Linearizability. In *SPIN (LNCS)*, Vol. 5578. Springer, 261–278. https://doi.org/10.1007/978-3-642-02652-2_21

Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based Memory Reclamation. In *PPOPP*. ACM, 1–13. https://doi.org/10.1145/3178487.3178488

Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. 2006. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *CAV (LNCS)*, Vol. 4144. Springer, 17–30. https://doi.org/10.1007/11817963_5

Albert Mingkun Yang and Tobias Wrigstad. 2017. Type-assisted Automatic Garbage Collection for Lock-free Data Structures. In *ISMM*. ACM, 14–24. https://doi.org/10.1145/3092255.3092274

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. 2008. Scalable Shape Analysis for Systems Code. In *CAV (LNCS)*, Vol. 5123. Springer, 385–398. https://doi.org/10.1007/978-3-540-70545-1_36

Xiaoxiao Yang, Joost-Pieter Katoen, Huimin Lin, and Hao Wu. 2017. Verifying Concurrent Stacks by Divergence-Sensitive Bisimulation. *CoRR* abs/1701.06104 (2017). http://arxiv.org/abs/1701.06104

Shao Jie Zhang. 2011. Scalable Automatic Linearizability Checking. In *ICSE*. ACM, 1185–1187. https://doi.org/10.1145/1985793.1986037

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *CAV (LNCS)*, Vol. 9207. Springer, 3–19. https://doi.org/10.1007/978-3-319-21668-3_1