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

ANONYMOUS AUTHOR(S)

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, and additionally the program's invariant annotations hold, 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 hardcoded but a parameter of the type system definition. We rely on a recent specification language for SMR algorithms. Our second contribution is to automate the type and the invariant check. For the type check, 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 and allows us to infer appropriate annotations automatically in a guess&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 an order of magnitude.

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

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.

Unfortunately, lock-free data structures are also particularly hard to get correct. The reason is the avoidance 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 9). Doherty et al. [2004], for instance, give a mechanized proof of a practical lock-free queue. Such proofs require a lot 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.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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 than it may seem at first glance. The root of the problem is that threads typically traverse the data structure without synchronization. Hence, threads may hold pointers to records that have already been removed from the structure. If those records are reclaimed immediately after the removal, 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 9).

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 [2019] 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 that 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 reallocations. Meyer and Wolff [2019] 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). Their state space exploration, unfortunately, scales poorly.

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 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 has four stages: (i) local, (ii) active, (iii) retired, and (iv) unallocated. 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. Once 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 always fear interferences by others. It is worth

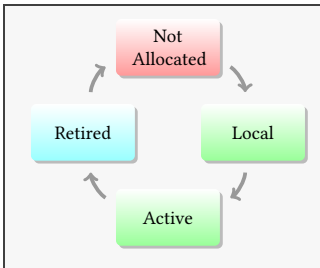

Fig. 1. Memory life cycle of records in lock-free data structures using SMR.

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 acquired a pointer to that record already. 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 need to integrate knowledge about the SMR algorithm. A main aspect of our development is that the actual SMR algorithm is an input to our type system—the type system is not tailored towards a specific SMR algorithm. Additionally, to maintain the guarantee in any execution of the program, the types need to be 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. However, SMR algorithms usually offer no means of checking 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*. In our setting, the problem is that 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 alleviate 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 check relying on invariant annotations to establish memory safety, (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 check. Our tool is able 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 SMR method [Michael 2002b] as well as epoch-base reclamation [Fraser 2004; McKenney and Slingwine 1998]. 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 introduce the programming model, §4 discuss preliminary results, §5 presents our type system for proving lock-free data structures memory safe for a user-specified SMR algorithm, §6 presents an efficient type checking

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

Fig. 2. Michael&Scott's lock-free queue [Michael and Scott 1996] with hazard pointers [Michael 2002b] to safely reclaim unused memory. The modifications to use hazard pointers are marked with H. Each thread has two hazard pointers.

algorithm, §7 presents an instrumentation of the data structure under scrutiny to discharge invariants fully automatically with the help of a GC verifier, §8 evaluates our approach on well-known lock-free data structures and demonstrates its practicality, and §9 discusses related work.

2 THE CONTRIBUTION ON AN EXAMPLE

We illustrate our approach on Micheal&Scott's lock-free queue [Michael and Scott 1996] 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 40 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 so-called safe memory reclamation (SMR) algorithms. A well-known example is the hazard pointer (HP) method by Michael [2002b]. It offers an operation `retire` that replaces the ordinary `delete`. Their difference is that `retire` does not immediately delete the nodes it receives. 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. To that end, HP offers another operation: `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 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. First, `delete` statements are replaced with calls to `retire`. Second, pointers that are accessed need protection to defer their deletion. However, simply calling `protect` is usually insufficient as the `protect` may be too late. To that end, 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 26 to 28. Line 26 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 27. 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 28 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 hazard pointers from Figure 2. At the heart of our method lies a type system which establishes all pointer operations in the data structure to be safe. In the case of hazard pointers, for instance, this requires to prove that all pointer accesses are 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, with the SMR operation invocations removed, under garbage collection. This allows off-the-shelf GC verifiers to be used.

2.1 A Type System to Simplify Verification

Our main contribution is a type system which establish that a given program does not perform 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 any 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 26 to 32. 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 26, 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 such that the actual type of `head` is $\emptyset$. 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 27 protects head. We set the type of head to $\mathbb{E}$. Remembering that head is protected is crucial for the subsequent conditional. Line 28 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. Subsequent code can access head without precautions. We incorporate this fact into the type of head by updating it to $\mathbb{S}$. (We skip the assignment to `tail` from Line 29, it does not affect the type check.) Next, Line 30 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 records, like `head->next`. Hence, next cannot obtain any guarantees from the assignment. Line 31 then protects next. Similarly to the above, we set its type to $\mathbb{E}$. The following conditional, Line 32, 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 before, this allows us to lift the type to $\mathbb{S}$. That is, using next subsequently becomes safe due to the conditional guaranteeing its activeness. The remainder of the type check is then straight forward since only protected and/or shared pointers are used.

```
{ shared:𝔸 }
  Node* head = Head;
{ shared:𝔸, head:∅ }
  protect(head, 0);
{ shared:𝔸, head:𝔼 }
  assume(head == Head);
{ shared:𝔸, head:𝕊 }
  Node* next = head->next;
{ shared:𝔸, head:𝕊, next:∅ }
  protect(next, 1);
{ shared:𝔸, head:𝕊, next:𝔼 }
  assume(head == Head);
{ shared:𝔸, head:𝕊, next:𝕊 }
```

Fig. 3. Idealized type check for the non-retrying branch of Lines 26 to 32.

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; O'Hearn 2004; 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 proving the correctness of such annotations manually, we automate the correctness check. To that end, we give an instrumentation of the program under scrutiny such that an ordinary GC verifier can discharge the invariants. Note 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 28 nevertheless, we annotate the `assume(head == Head)` statement with an invariant stating that head is active. Then, the type derivation for Line 28 remains the same as before. We argue that, provided the queue implementation is safe, there must be a code location between

the protection in Line 27 and the subsequent dereference in Line 30 where an active annotation can be placed. To see this, assume there is no such code location. This means head has been retired as it is not active. Following our discussion from before, it must have been retired before the protection since HP would guarantee a safe dereference otherwise. Hence, the dereference in Line 30 is unsafe, contradicting our assumption. For pointer next, we proceed similarly and add an active annotation to the second assumption (Line 32).

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 2019]: SMR automata. Then, our types capture the locations of the SMR automaton that can be reached at a given control point in the program. 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 records.

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 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. To simplify the type system in Section 5, we assume that commands are always nested in an atomic block.

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 $DExp = DVar \uplus DSel$ with $a.\texttt{data} \in DSel$ evaluate. A generalization of our development to further selectors is straightforward.

The memory is a partial function $m : PExp \uplus DExp \rightharpoonup Adr \uplus \{\texttt{seg}\} \uplus Dom$ that respects the typing. The initial memory is $\mathfrak{m}_\epsilon$. Pointer variables p are uninitialized, $\mathfrak{m}_\epsilon(p) = \texttt{seg}$. Data variables u have a default value, $\mathfrak{m}_\epsilon(u) = 0$. 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 $\mathfrak{m}_\tau$ for the memory resulting from the initial memory $\mathfrak{m}_\epsilon$ when applying the sequence of updates in τ.

Liberal Semantics. We define a semantics where a 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 $Y \subseteq X \subseteq Adr$ of addresses allowed to be reallocated and freed, respectively. A computation is a sequence τ of actions of the form $act = (t, com, up)$. 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, $\tau.act \in [\![P]\!]^Y_X$ 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 $\mathfrak{m}_\tau(p) = a$ and $\mathfrak{m}_\tau(q) = b$. There are similar conditions for the remaining assignments.

(Assume) If $act = (t, \texttt{assume}\ lhs = rhs, \emptyset)$ then $\mathfrak{m}_\tau(lhs) = \mathfrak{m}_\tau(rhs)$. 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 $a \in fresh(\tau)$ or $a \in freed(\tau) \cap Y$, and $d \in Dom$.

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

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 $fresh(\tau)$, and addresses which have been freed since their last allocation, $freed(\tau)$.

To fix the notation, we will use $ctrl(\tau)$ to refer to the control locations of threads reached after computation τ. We refrain from making this precise as we will only use statements of the form $ctrl(\tau) = ctrl(\sigma)$ to state that after τ and σ the same commands can be executed.

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 2019]: 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.

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

SMR Automata. An SMR algorithm offers a variety of functions $f(\bar{r})$ for the programmer to provide information about the intended access to the data structure, like `protect` and `retire` in the case of hazard pointers (cf. Section 2). An SMR automaton as depicted in Figure 4 is a finite control structure the transitions of which are labeled by these function symbols. Additionally, each transition comes with a guard. It allows to influence 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.

There are two technical restrictions that limit the expressiveness of SMR automata and make them interesting for automated verification. First, the variable values are chosen only once, in the beginning of the computation, and never change. This choice is non-deterministic. The idea is that the automaton picks some protection to track. Second, transition guards can only compare for equality. 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 observer 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 observer 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 induced by 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 will also use a restriction of the induced 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} from Figure 4. It states that the SMR algorithm supports a function `retire`(t, a) via which a thread can request the SMR algorithm to free an address. Moreover, it forbids the SMR implementation to free addresses that have not yet been retired or have not been retired since last being freed. The automaton is important. While every

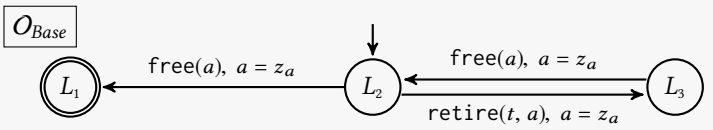

Fig. 4. Observer specifying that address z_a may be freed only if it has been retired and not been freed since. It uses one observer variable z_a. For legibility, we omit self loops on all locations for the remaining events.

SMR algorithm has its own SMR automaton, the practically relevant SMR automata are products[2] of O_{Base} with further SMR automata [Meyer and Wolff 2019]. 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.

We also require that the SMR algorithm does not remember addresses that have been freed in order to react to a reallocations—a very natural guarantee. Formally, we say that an SMR automaton *supports elision* if (i) the behavior on address a must not be affected by a free of another address b, (ii) the behavior on address a must not be affected by name swapping another two addresses b and c, (iii) the behavior on address a after history h must be included in the behavior on a after h', provided a is fresh after h', and (iv) the behavior on address a after history $h.\texttt{free}(a)$ must be included in the behavior on a after h. Intuitively, 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 2019], 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 gives a set of locations L' reachable by taking a com-labeled transition from L. The pointer p restricts the transitions to be considered. If p appears as a parameter in com, then the guard of the transition has to imply $p = z_a$. The parameterization in p makes the post image precise. Consider O_{Base} and the command $com = \texttt{enter retire}(p)$. 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:

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

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

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 actual ones. To be precise, we use as events the function names offered by the SMR automaton

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

plus `free`. The parameters to an event are the 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, $\mathcal{H}(\tau.(t, \texttt{enter}\ \mathit{func}(p), \varnothing)) = \mathcal{H}(\tau).\mathit{func}(t, \mathfrak{m}_\tau(p))$.

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 re-allocatable 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 free events, the SMR automaton does not play a role. We refer to the resulting semantics $[\![P]\!]_\varnothing^\varnothing$ as 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 $O[\![P]\!]_\varnothing^\varnothing$. By Assumption 1, only events $\texttt{free}(a)$ may lead to such accepting states.

Reconsider the SMR automaton O_{Base}. 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). 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 benefitial 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 epoch-based reclamation (EBR) [Fraser 2004; McKenney and Slingwine 1998]. It provides a function which guarantees the calling thread 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:

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

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

$$
\begin{array}{rcll}
inv(\epsilon) & := & true & \\
inv(act.\tau) & := & \exists r.\, inv(\tau) & \text{if } act = (t, \texttt{@inv angel } r, up) \\
inv(act.\tau) & := & \mathfrak{m}_\tau(cond) \wedge inv(\tau) & \text{if } act = (t, \texttt{@inv } cond, up) \\
inv(act.\tau) & := & \mathfrak{m}_\tau(p) \in r \ \wedge inv(\tau) & \text{if } act = (t, \texttt{@inv } p \texttt{ in } r, up) \\
inv(act.\tau) & := & \mathfrak{m}_\tau(p) \in active(\tau) \wedge inv(\tau) & \text{if } act = (t, \texttt{@inv active}(p), up) \\
inv(act.\tau) & := & r \subseteq active(\tau) \wedge inv(\tau) & \text{if } act = (t, \texttt{@inv active}(r), up) \\
inv(act.\tau) & := & inv(\tau) & \text{otherwise.}
\end{array}
$$

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, $\mathfrak{m}_\tau(cond)$ is $cond$ with the pointers p replaced by their values $\mathfrak{m}_\tau(p)$. Moreover, $active(\tau)$ is the set of addresses that are neither freed nor retired after τ. 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 7 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 [Reynolds 2002] do not apply. Meyer and Wolff [2019] 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 2019]). *If the semantics* $O[\![P]\!]^{\varnothing}_{Adr}$ *is pointer-race-free, then* $O[\![P]\!]^{Adr}_{Adr}$ *coincides with* $[\![P]\!]^{\varnothing}_{\varnothing}$.

We will not need the correspondence between the semantics in our development and prefer not to define it. It is precise enough for verification results to carry over from one semantics to the other. Pointer races generalize the concept of memory errors by taking into account the SMR algorithm [Haziza et al. 2016; Meyer and Wolff 2019]. 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 `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 10 and 27 of Michael&Scott's queue from Section 2. To support such calls, one deems a command `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.

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

With Theorem 4.1, the only property to be checked in the presence of memory reclamation is the premise of pointer race freedom. However, Meyer and Wolff [2019] 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.

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 notion of validity for pointers (cf. Section 4). We achieve the necessary precision by additionally tracking the SMR behavior. For example, we may derive during the type check that pointers are protected by the SMR algorithm, hence guaranteed not to be freed, and thus remain valid. The actual SMR algorithm is a parameter of the type system definition.

A key design decision of our type system is not to track information about the data structure shape. A programming-philosophical reason is that the same SMR algorithm may be used with different data structures. Hence, shape information should not help tracking its behavior. An algorithmic reason is that it saves us from having to maintain a memory representation, which easily degrades the performance of a program analysis. Instead, we rely on annotations in the program to deduce runtime specific information that is not available to the type system. In particular, we rely on annotations to mark active pointers and angels. Section 7 shows how to discharge those annotations with an off-the-shelf verifier for garbage collection.

5.1 Overview

The goal of our type system is to prove a program free from pointer races. Consequently, types need to carry validity information (cf. Section 4). As discussed in 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 a call to an SMR protection was successful. Additionally, types need to be stable under interferences from other threads. 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 overcome 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. In the case of hazard pointers, for example, such a guarantee could state that a pointer has been protected while it was still active. Thus, the pointer can be accessed safely even though it may have moved into the retired stage without the accessing thread noticing. 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 aliases of the pointer and the referenced node is not retired. This gives the thread holding the pointer exclusive access.
- $\mathbb{A}$ Pointers to records in the active stage. Active pointers are guaranteed to be valid, they can be accessed safely.
- $\mathbb{S}$ Pointers to records which are protected by the SMR 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.

An important aspect of our type system is the active guarantee and the reasoning power that comes with it. For shared nodes, a thread has to ensure that the access is safe. In the case of hazard pointers, this is done by acquiring a pointer p to the shared node. This pointer comes without access guarantees and needs to be protected. 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 makes sure p is active during the invokation. The type system detects this through an annotation that adds guarantee $\mathbb{A}$ to p. We then deduce from the SMR algorithm 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 $isValid(T)$ 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 locally from the code. Soundness then shows that these guarantees carry over to the computations of the overall program where threads interfere. To justify this local to global 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 interferences 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 guard g implying $t' \neq z_t$ we have $l' \in L$. We only introduce guarantees $\mathbb{E}_L$ for sets of locations L that are closed under interferences 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 execution 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 interferences from other threads and for which there is no transition $l \xrightarrow{\mathtt{free}(a),\, g} l'$ with $l \in SafeLoc(O)$ 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. Surprisingly, also for $\mathbb{L}$ we can use location L_2. That other pointers have no alias means the set of locations cannot be left by appending commands acting on other addresses. This happens to be the case for 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.

Guarantee $\mathbb{A}$ has a special role. It is the only guarantee for which the set of locations $Loc(\mathbb{A})$ is not closed under interferences (for $\mathbb{L}$ we have interference freedom for other addresses, see above). This means events of other threads may remove the guarantee from a type. We define an operation $rm(\Gamma)$ that takes an environment and removes all $\mathbb{A}$ guarantees for thread-local pointers and angels:

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

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 $\vdash \Gamma_{init}\ \{stmt\}\ \Gamma$ to indicate that $\Gamma_{init}\ \{stmt\}\ \Gamma$ is derivable with the given rules. We write $\vdash stmt$ if there is an environment Γ so that $\vdash \Gamma_{init}\ \{stmt\}\ \Gamma$. 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. The rules for primitive commands assume that the command appears inside an atomic block. With this assumption, they need not handle the fact that guarantee $\mathbb{A}$ is not closed under interferences. Interferences 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 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. The post type environment of Rules (ENTER) and (EXIT) simply infers the guarantees of the pre type environment wrt. 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 $safeEnter(\Gamma, func(\bar{p}, \bar{u}))$ evaluates to *true* iff the command `enter` $func(\bar{p}, \bar{u})$ 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 $isValid(\cdot)$. A special case of Rule (ENTER) is the invocation of `retire`(p), which require the argument p to be active. This will prevent double retires.

The rules for statements are given in Figure 6b. Rule (INFER) allows for type transformations at any point during the type check, 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 interferences. 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 post type environment, Rule (CHOICE). Loops require a type environment that is stable under the loop body, Rule (CHOICE).

$$\frac{p \notin \mathit{shared} \qquad T = \{\mathbb{L}\}}{\Gamma, p\ \{p := \texttt{malloc}\}\ \Gamma, p{:}T}\ (\textsc{malloc})$$

$$\frac{T' = T \setminus \{\mathbb{L}\}}{\Gamma, p, q{:}T\ \{p := q\}\ \Gamma, p{:}T', q{:}T'}\ (\textsc{assign1})$$

$$\frac{\Gamma(q) = T \qquad \mathit{isValid}(T)}{\Gamma, p\ \{p := q.\texttt{next}\}\ \Gamma, p{:}\varnothing}\ (\textsc{assign2})$$

$$\frac{\Gamma(p) = T \qquad \mathit{isValid}(T) \qquad T'' = T' \setminus \{\mathbb{L}\}}{\Gamma, q{:}T'\ \{p.\texttt{next} := q\}\ \Gamma, q{:}T''}\ (\textsc{assign3})$$

$$\frac{}{\Gamma\ \{u := \mathsf{op}(\bar{u})\}\ \Gamma}\ (\textsc{assign4})$$

$$\frac{\Gamma(q) = T \qquad \mathit{isValid}(T)}{\Gamma\ \{u := q.\texttt{data}\}\ \Gamma}\ (\textsc{assign5})$$

$$\frac{\Gamma(p) = T \qquad \mathit{isValid}(T)}{\Gamma\ \{p.\texttt{data} := u\}\ \Gamma}\ (\textsc{assign6})$$

$$\frac{\mathit{isValid}(T) \qquad \mathit{isValid}(T') \qquad T'' = (T \wedge T') \setminus \{\mathbb{L}\}}{\Gamma, p{:}T, q{:}T'\ \{\texttt{assume}\ p = q\}\ \Gamma, p{:}T'', q{:}T''}\ (\textsc{assume1})$$

$$\frac{\mathit{cond} \not\equiv p = q}{\Gamma\ \{\texttt{assume}\ \mathit{cond}\}\ \Gamma}\ (\textsc{assume2})$$

$$\frac{T'' = T \wedge T'}{\Gamma, p{:}T, q{:}T'\ \{\texttt{@inv}\ p = q\}\ \Gamma, p{:}T'', q{:}T''}\ (\textsc{equal})$$

$$\frac{T' = T \wedge \{\mathbb{A}\}}{\Gamma, x{:}T\ \{\texttt{@inv active}(x)\}\ \Gamma, x{:}T'}\ (\textsc{active})$$

$$\frac{r \notin \mathit{shared}}{\Gamma, r\ \{\texttt{@inv angel}\ r\}\ \Gamma, r{:}\varnothing}\ (\textsc{angel})$$

$$\frac{\Gamma(r) = T' \qquad T'' = T \wedge T'}{\Gamma, p{:}T\ \{\texttt{@inv}\ p\ \texttt{in}\ r\}\ \Gamma, p{:}T''}\ (\textsc{member})$$

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

$$\frac{\Gamma, \texttt{exit}\ \mathit{func} \rightsquigarrow \Gamma'}{\Gamma\ \{\texttt{exit}\ \mathit{func}\}\ \Gamma'}\ (\textsc{exit})$$

(a) Type rules for primitive commands.

$$\frac{\Gamma_1 \rightsquigarrow \Gamma_2 \qquad \Gamma_2\ \{\mathit{stmt}\}\ \Gamma_3 \qquad \Gamma_3 \rightsquigarrow \Gamma_4}{\Gamma_1\ \{\mathit{stmt}\}\ \Gamma_4}\ (\textsc{infer})$$

$$\frac{}{\Gamma\ \{\texttt{beginAtomic}\}\ \Gamma}\ (\textsc{begin})$$

$$\frac{}{\Gamma\ \{\texttt{endAtomic}\}\ \mathit{rm}(\Gamma)}\ (\textsc{end})$$

$$\frac{\Gamma\ \{\mathit{stmt}_1\}\ \Gamma' \qquad \Gamma'\ \{\mathit{stmt}_2\}\ \Gamma''}{\Gamma\ \{\mathit{stmt}_1; \mathit{stmt}_2\}\ \Gamma''}\ (\textsc{seq})$$

$$\frac{\Gamma\ \{\mathit{stmt}_1\}\ \Gamma' \qquad \Gamma\ \{\mathit{stmt}_2\}\ \Gamma'}{\Gamma\ \{\mathit{stmt}_1 \oplus \mathit{stmt}_2\}\ \Gamma'}\ (\textsc{choice})$$

$$\frac{\Gamma\ \{\mathit{stmt}\}\ \Gamma}{\Gamma\ \{\mathit{stmt}^*\}\ \Gamma}\ (\textsc{loop})$$

(b) Type rules for statements.

Fig. 6. Type rules.

5.4 Example

We revisit the type checking example from Section 2. The type derivation is given in Figure 7. The goal is to derive in our type system that head is successfully protected. For simplicity, we give in the type environments in Figure 7 only the pointers Head and head. The remaining pointers are also present but do not affect our example.

In the beginning, the type of Head and head is $\varnothing$. There are no guarantees. The first command assigns the value of Head to head. We apply Rule (ASSIGN1). Both pointers continue to have type $\varnothing$. Next, protect is invoked with head as a parameter. Using Rule (ENTER) we obtain type $\mathbb{E}_{inv}$ for head, stating that we have definitely invoked protect for head. The type of Head is not affected. After protect returns, head obtains type $\mathbb{E}_{isu}$ from an application of Rule (EXIT). It encodes the fact that a protection

has been issued. Accessing head, however, is not safe at this point since we do not know whether head has been retired in the meantime. Next, we enter an atomic section, it has no effect on the types as of Rule (BEGIN). The subsequent active annotation adds $\mathbb{A}$ to the type of Head, Rule (ACTIVE). Guarantee $\mathbb{A}$ is maintained because we are inside an atomic block. The following assumption conjoins the types of Head and head, Rule (ASSUME1). The result is $\mathbb{A} \wedge \mathbb{E}_{isu}$. An application of Rule (INFER) allows us to infer $\mathbb{S}$ for head. To this end, we perform an inclusion check among the corresponding locations, $Loc(\mathbb{A}) \cap Loc(\mathbb{E}_{isu}) \subseteq Loc(\mathbb{S})$, as required by the transformer relation $\rightsquigarrow$. We could also infer $\mathbb{S}$ for Head, but the guarantee will be removed when leaving the atomic section. Finally, Rule (END) updates the types to account for interference after leaving the atomic section. The type of Head is set to $\varnothing$ and and $\mathbb{A}$ is removed from head. The point to note here is that head maintains $\mathbb{S}$. That is, subsequent accesses are guaranteed to be safe.

```
{ Head:∅, head:∅ }
  Node* head = Head;
{ Head:∅, head:∅ }
  enter protect(head, 0);
{ Head:∅, head:E_inv }
  exit protect;
{ Head:∅, head:E_isu }
  beginAtomic;
{ Head:∅, head:E_isu }
  @inv active(Head)
{ Head:A, head:E_isu }
  assume(head == Head);
{ Head:A ∧ E_isu, head:A ∧ E_isu }
{ Head:A ∧ E_isu, head:A ∧ E_isu ∧ S }
  endAtomic;
{ Head:∅, head:E_isu ∧ S }
```

Fig. 7. Type check of Lines 26 to 28 for a successful protection of head.

5.5 Soundness

Our goal is to show that a successful type check $\vdash stmt$ implies pointer race freedom and the absence of double retires. There are two challenges. We already commented on the problematic local to global 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 we can use garbage collection for this purpose, meaning the invariant annotations only have to hold in $[\![P]\!]^{\varnothing}_{\varnothing}$, although the soundness result refers to the computations in $O[\![P]\!]^{\varnothing}_{Adr}$.

THEOREM 5.1 (SOUNDNESS). *If $\vdash P$ and $inv([\![P]\!]^{\varnothing}_{\varnothing})$ hold, then $O[\![P]\!]^{\varnothing}_{Adr}$ is pointer-race-free and $O[\![P]\!]^{Adr}_{Adr}$ does not perform double retires.*

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. The result gives an effective means for checking the premise of Theorem 4.1: determine a typing using the proposed type system and discharge the invariant annotations using an existing, off-the-shelf verification tool for garbage collection (cf. Section 7). The absence of double retires is the precondition for a meaningful application of O_{Base} (cf. Section 3).

The proof makes explicit the information tracked by our type system. (i) The type environment annotating a program point approximates the history of every computation reaching this point. (ii) Moreover, $isValid(\cdot)$ approximates validity. The two statements follow from this. For pointer race freedom, consider a command which could raise a pointer race. Before executing the command, the corresponding rule performs a validity check. Take Rule (ASSIGN2) as an example. Before dereferencing q, it checks $isValid(T)$. By (ii), the pointer is valid. For the absence of double retires, note that Rule (ENTER) requires the pointer to be active. By (i), the computation has taken O_{Base} to state L_2. The state, however, can only be reached if all earlier retires of the address have been followed by a free. Hence, we do not have a double retire.

In the following two sections, we automate the checks in Theorem 5.1. We give an efficient algorithm for type checking $\vdash P$ and show how to discharge the invariants $inv([\![P]\!]^{\varnothing}_{\varnothing})$ with the help of off-the-shelf verification tools.

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

Fig. 8. Constraint system $\Phi(X, stmt, Y)$.

6 TYPE CHECKING

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

Theorem 6.1. *Given a program stmt, the type check* $\vdash stmt$ *is decidable in* $O(|\,stmt\,|^2)$.

As common in type systems [Pierce 2002], our algorithm for type checking 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 check. The correspondence with type checking 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 checking 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.

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 check. 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 greatest element.

The constraint system $\Phi(\Gamma_{init}, stmt, X)$ is defined in Figure 8. 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. 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 check 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 check 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 check. This is the key correspondence.

PROPOSITION 6.2 (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 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.

7 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 $\mathsf{havoc}(p)$ 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 7.1 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 7.1 (SOUNDNESS AND COMPLETENESS). *We have $inv([\![P]\!]^{\varnothing}_{\varnothing})$ if and only if $safe([\![inst(P)]\!]^{\varnothing}_{\varnothing})$. The source-to-source translation is linear in size.*

$$
\begin{aligned}
\mathit{inst}(\mathit{stmt}^*) &:= \mathit{inst}(\mathit{stmt})^* & \mathit{inst}(\texttt{enter}\ \mathit{func}(\bar{p}, \bar{u})) &:= \texttt{skip}\\
\mathit{inst}(\mathit{stmt}_1 \oplus \mathit{stmt}_2) &:= \mathit{inst}(\mathit{stmt}_1) \oplus \mathit{inst}(\mathit{stmt}_2) & \mathit{inst}(\texttt{exit}\ \mathit{func}) &:= \texttt{skip}\\
\mathit{inst}(\mathit{stmt}_1; \mathit{stmt}_2) &:= \mathit{inst}(\mathit{stmt}_1); \mathit{inst}(\mathit{stmt}_2) & \mathit{inst}(\texttt{@inv}\ p = q) &:= \texttt{assert}\ p = q\\
\mathit{inst}(\mathit{com}) &:= \mathit{com}
\end{aligned}
$$

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

Fig. 9. Source-to-source translation replacing SMR commands and invariant annotations.

The source-to-source translation is defined in Figure 9. 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 `@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 operation 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 `@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), has the address been retired. If so, 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.

8 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. To substantiate the 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; McKenney and Slingwine 1998]. We adapted the GC implementations of Vechev&Yahav DCAS set, the Vechev&Yahav CAS set, and the ORVYY set given in the literature to use the hazard pointer method. We briefly elaborate on EBR. In EBR, threads have access to the functions `leaveQ` and `enterQ` both of which do not take parameters. A call to `leaveQ` guarantees that all nodes active at the point of the call remain so until the thread calls `enterQ`. Typically, threads call `leaveQ` at the beginning of an operation and `enterQ` at the end. (A thread's state between `enterQ` and `leaveQ` is called quiescent, hence the function names.)

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 (i) the time taken for the type check with manually picked guarantees $\mathbb{E}_L$, (ii) the time taken for the type check with automatically inferred guarantees $\mathbb{E}_L$, (iii) the time taken for discharging the invariant annotations, and (iv) the time taken 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. The guarantees $\mathbb{E}_L$ were identical across all benchmarks. For HP, we picked 18 SMR-specific guarantees while our tool generated 315. For EBR, we picked 5 guarantees and 9 were generated.

Our approach is capable of verifying most of the lock-free data structures we considered. We observe that the type check using our hand-crafted instantiation is always faster than the type check with the synthesized instantiation. Comparing the total runtime with our competitors [Meyer and Wolff 2019], the only other approach capable of handling lock-free data structures with general SMR algorithms, we experience a speed-up of over one order 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. The DGLM queue is similar to Michael&Scott's queue, however, allows `Head` to overtake `Tail`. 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 this case. This is no limitation of our

[3] The non-anonymous supplementary materials of this submission contain the tool. We will make the tool publicly available online and submit it to the artifact evaluation.

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

SMR	Program	man. Types		gen. Types		Annotations		Lineariz.	
HP	Treiber's stack	1.2*s*	✓	26*s*	✓	12*s*	✓	1*s*	✓
	Opt. Treiber's stack	0.8*s*	✓	19*s*	✓	11*s*	✓	1*s*	✓
	Michael&Scott's queue	2.3*s*	✓	63*s*	✓	12*s*	✓	4*s*	✓
	DGLM queue	42.3*s*	✓	23*s*	✓	1*s*	✗[a]	5*s*	✓
	Vechev&Yahav DCAS set	11*s*	✓	230*s*	✓	13*s*	✓	98*s*	✓
	Vechev&Yahav CAS set	11*s*	✓	205*s*	✓	3.5*h*	✓	42*m*	✓
	ORVYY set	11*s*	✓	205*s*	✓	3.2*h*	✓	47*m*	✓
	Michael's set	23*s*	✓	479*s*	✓	90*s*	✗[a]	—	◔
EBR	Treiber's stack	0.12*s*	✓	0.16*s*	✓	10*s*	✓	1*s*	✓
	Opt. Treiber's stack	0.12*s*	✓	0.16*s*	✓	12*s*	✓	1*s*	✓
	Michael&Scott's queue	0.18*s*	✓	0.18*s*	✓	16*s*	✓	5*s*	✓
	DGLM queue	0.17*s*	✓	0.19*s*	✓	1*s*	✗[a]	6*s*	✓
	Vechev&Yahav DCAS set	0.23*s*	✓	0.25*s*	✓	38*s*	✓	200*s*	✓
	Vechev&Yahav CAS set	0.21*s*	✓	0.23*s*	✓	7*h*	✓	42*m*	✓
	ORVYY set	0.21*s*	✓	0.23*s*	✓	7*h*	✓	47*m*	✓
	Michael's set	0.48*s*	✓	0.51*s*	✓	22*s*	✗[a]	—	◔

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

approach, it is a shortcoming of the back-end verifier. Meyer and Wolff [2019] reported a similar issue that they solved by manually providing hints to improve the precision of their analysis. A similar problem occurs for Michael's set.

The annotation checks for set implementations reveal an interesting fact. 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 is that EBR implementations require angels. Interestingly, 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 movers [Lipton 1975], a well-known program transformation with applications ranging from [Lipton 1975] over [Elmas et al. 2009] to [Kragl and Qadeer 2018] (cf. Section 9). 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.) This allows us to check moverness for memory commands using existing techniques. For SMR commands, moverness can be read of the SMR automaton. Our tool is able to apply moverness arguments. Due to page limitations, we omit a thorough discussion of the matter.

9 RELATED WORK

Safe Memory Reclamation. Besides HP and EBR 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 aimed on improving the practical performance by tweaking or combining existing SMR techniques. For example, *DEBRA* [Brown 2015] is an optimized EBR implementation. Harris [2001] modifies EBR to store epochs inside records. 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 as well besides EBR and HP.

Memory Safety. We use our type system to show that a program is free from pointer races, meaning that dangling pointers are not used. There are a number of related tools that can (be used to) 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]. Those tools differ from ours as they can only handle sequential code. Moreover, our type system does not include a memory abstraction to track the data structure shape. The above tools do include such an abstraction to identify unsafe pointer operations. In our type system, this task is delegated 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.

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 simpler semantics without memory deletions.

Castegren and Wrigstad [2017] give a type system that guarantees the absence of data races. Their types encode a notion of ownership that prevents non-owning threads from accessing a node. There 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.

Linearizability. 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 less 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]. For a more detailed overview of manual techniques, we refer to the survey by Dongol and Derrick [2014].

Many manual verification approaches rely on a program logic [da Rocha Pinto et al. 2014; Gotsman et al. 2013; Turon et al. 2014; Vafeiadis and Parkinson 2007]. Program logics are conceptually related to our type system. However, such logics integrate several different ingredients to successfully verify intricate lock-free data structures [Turon et al. 2014]. Most importantly, they include memory abstractions, like separation logic [O'Hearn 2004; Reynolds 2002; Vafeiadis and Parkinson 2007], and thread interference abstractions, like rely-guarantee reasoning [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 and delegate the abstraction to a back-end verifier. Moreover, accounting for thread interferences in our type system boils down to defining guarantees as closed sets of locations and removing guarantee $\mathbb{A}$ upon exiting atomic blocks.

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]. Memory abstractions are simpler and more efficient when reclamation is not considered, they can exploit ownership guarantees [Bornat et al. 2005; Boyland 2003] and the resulting thread-local reasoning techniques [Reynolds 2002]. Few works [Abdulla et al. 2013; Haziza et al. 2016; Holík et al. 2017; Meyer and Wolff 2019] address the challenge of verifying lock-free data structures under manual memory management. Besides Meyer and Wolff [2019], they use hand-crafted semantics that allow for accessing deleted memory. The work by Meyer and Wolff [2019] is the closest related. We build on their programming model and their reduction result as discussed in Sections 3 and 4, respectively.

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].

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

Ralph-Johan Back. 1989. A Method for Refining Atomicity in Parallel Algorithms. In *PARLE (2) (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.

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 (2) (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.

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

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

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

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.

Karl Crary, David Walker, and J. Gregory Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *POPL*. ACM, 262–275.

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

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

John Derrick, Gerhard Schellhorn, and Heike Wehrheim. 2011. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.* 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

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. 2014. Verifying linearizability: A comparative survey. *CoRR* abs/1410.6268 (2014). http://arxiv.org/abs/1410.6268

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

Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. 2008. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.* 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.

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.

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 Computer Society, 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.

Lindsay Groves. 2007. Reasoning about Nonblocking Concurrency using Reduction. In *ICECCS*. IEEE Computer Society, 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 (2) (LNCS)*, Vol. 9207. Springer, 449–465.

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. *ACM Trans. Program. Lang. Syst.* 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 *Haifa Verification Conference (LNCS)*, Vol. 10028. 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

Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. In *POPL*. ACM, 79–90.

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. Specification and Design of (Parallel) Programs. In *IFIP Congress*. 321–332.

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

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 (1) (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* (May 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/10.1145/2462156.2462189

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. *ACM Trans. Program. Lang. Syst.* 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.

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

Roland Meyer and Sebastian Wolff. 2018. Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis. *CoRR* abs/1810.10807 (2018). http://arxiv.org/abs/1810.10807

Roland Meyer and Sebastian Wolff. 2019. 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

Maged M. Michael. 2002a. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*. 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

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

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, 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 Computer Society, 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

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

R.Kent Treiber. 1986. *Systems programming: coping with parallelism*. Technical Report RJ 5118. IBM.

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.

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

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *CAV (2) (LNCS)*, Vol. 9207. Springer, 3–19. https://doi.org/10.1007/978-3-319-21668-3_1

A MISSING DETAILS

A.1 Definitions

Definition A.1. The liberal semantics is defined by the following rules, assuming $\tau \in [\![P]\!]_X^Y$ and that *act* respects the control flow of P.

(Assign1) If $act = (t, p.\texttt{next} := q, [a.\texttt{next} \mapsto b])$ then $\mathfrak{m}_\tau(p) = a$ and $\mathfrak{m}_\tau(q) = b$. There are similar conditions for the remaining assignments.
(Assign2) If $act = (t, p := q, [p \mapsto \mathfrak{m}_\tau(q)])$.
(Assign3) If $act = (t, p := q.\texttt{next}, [p \mapsto \mathfrak{m}_\tau(a.\texttt{next})])$ with $\mathfrak{m}_\tau(q) = a \in Adr$.
(Assign4) If $act = (t, u := \mathsf{op}(u_1', \ldots, u_n'), [u \mapsto d])$ with $d = \mathsf{op}(\mathfrak{m}_\tau(u_1'), \ldots, \mathfrak{m}_\tau(u_n'))$.
(Assign5) If $act = (t, p.\texttt{data} := u', [a.\texttt{data} \mapsto \mathfrak{m}_\tau(u')])$ with $\mathfrak{m}_\tau(p) = a \in Adr$.
(Assign6) If $act = (t, u := q.\texttt{data}, [u \mapsto \mathfrak{m}_\tau(a.\texttt{data})])$ with $\mathfrak{m}_\tau(q) = a \in Adr$.
(Assume) If $act = (t, \texttt{assume}\ lhs \triangleq rhs, \emptyset)$ then $\mathfrak{m}_\tau(lhs) \triangleq \mathfrak{m}_\tau(rhs)$.
(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 $a \in \mathit{freed}(\tau) \cap Y$.
(Free) If $act = (\bot, \texttt{free}(a), \emptyset)$ then $a \in X$.
(Enter) $act = (t, \texttt{enter}\ \mathit{func}(\bar{p}, \bar{u}), \emptyset)$, then $\mathfrak{m}_\tau(p) \neq \texttt{seg}$ for every p in $\bar{p}$.
(Exit) $act = (t, \texttt{exit}\ \mathit{func}, \emptyset)$.
(Invariant1) $act = (t, \texttt{@inv angel}\ r, \emptyset)$.
(Invariant2) $act = (t, \texttt{@inv}\ p\ \texttt{in}\ r, \emptyset)$.
(Invariant3) $act = (t, \texttt{@inv active}(p), \emptyset)$.
(Invariant4) $act = (t, \texttt{@inv}\ p = q, \emptyset)$.

Definition A.2. The history induced by a computation τ, denoted $\mathcal{H}(\tau)$, is defined by:

$$\begin{aligned}
\mathcal{H}(\epsilon) &= \epsilon \\
\mathcal{H}(\tau.(t, \texttt{free}(a), up, pc)) &= \mathcal{H}(\tau).\texttt{free}(a) \\
\mathcal{H}(\tau.(t, \texttt{enter}\ \mathit{func}(\bar{p}, \bar{u}), up, pc)) &= \mathcal{H}(\tau).\mathit{func}(t, \mathfrak{m}_\tau(\bar{p}), \mathfrak{m}_\tau(\bar{u})) \\
\mathcal{H}(\tau.(t, \texttt{exit}\ \mathit{func}, up, pc)) &= \mathcal{H}(\tau).\texttt{exit}\ \mathit{func}(t) \\
\mathcal{H}(\tau.act) &= \mathcal{H}(\tau) \qquad \text{otherwise.}
\end{aligned}$$

Definition A.3. The fresh addresses in a computation τ, denoted by $\mathit{fresh}(\tau)$, are defined by:

$$\begin{aligned}
\mathit{fresh}(\epsilon) &= Adr \\
\mathit{fresh}(\tau.act) &= \mathit{fresh}(\tau) \setminus \{a\} && \text{if } \mathit{com}(act) \equiv \texttt{free}(a) \\
\mathit{fresh}(\tau.act) &= \mathit{fresh}(\tau) \setminus \{a\} && \text{if } \mathit{com}(act) \equiv p := \texttt{malloc} \wedge \mathfrak{m}_{\tau.act}(p) = a \\
\mathit{fresh}(\tau.act) &= \mathit{fresh}(\tau) && \text{otherwise.}
\end{aligned}$$

The definition carries over naturally to histories.

Definition A.4. The freed addresses in a computation τ, denoted by $\mathit{freed}(\tau)$, are defined by:

$$\begin{aligned}
\mathit{freed}(\epsilon) &= \emptyset \\
\mathit{freed}(\tau.act) &= \mathit{freed}(\tau) \cup \{a\} && \text{if } \mathit{com}(act) \equiv \texttt{free}(a) \\
\mathit{freed}(\tau.act) &= \mathit{freed}(\tau) \setminus \{a\} && \text{if } \mathit{com}(act) \equiv p := \texttt{malloc} \wedge \mathfrak{m}_{\tau.act}(p) = a \\
\mathit{freed}(\tau.act) &= \mathit{freed}(\tau) && \text{otherwise.}
\end{aligned}$$

Definition A.5. The retired addresses in a computation τ, denoted by $retired(\tau)$, are defined by:

$$\begin{aligned}
retired(\epsilon) &= \emptyset \\
retired(\tau.act) &= retired(\tau) \cup \{a\} && \text{if } com(act) \equiv \texttt{enter retire}(p) \wedge a = \mathfrak{m}_\tau(p) \\
retired(\tau.act) &= retired(\tau) \setminus \{a\} && \text{if } com(act) \equiv \texttt{free}(a) \\
retired(\tau.act) &= retired(\tau) && \text{otherwise.}
\end{aligned}$$

Definition A.6. The active addresses in a computations τ are:

$$active(\tau) := Adr \setminus (freed(\tau) \cup retired(\tau)) \ .$$

Definition A.7 (Angel Denotation). Consider some $\tau \in [\![P]\!]^{Adr}_{Adr}$. Let $inv(\tau)$ have the prenex normal form $\exists r_1 \ldots \exists r_n.\phi$, where ϕ is quantifier-free. Let r_n be the instance of angel r resulting from the last allocation in τ. The set of addresses possibly represented by angel r after computation τ is

$$repr_\tau(r) := \{a \in Adr \mid \exists A_1, \ldots, A_n \subseteq Adr.a \in A_n \wedge (A_1, \ldots, A_n) \models \phi\} \ .$$

Definition A.8 (Valid Expressions). The *valid pointer expressions* in a computation $\tau \in O[\![P]\!]^{Adr}_{Adr}$, denoted by $valid_\tau \subseteq PExp$, are defined by:

$$\begin{aligned}
valid_\epsilon &:= PVar \\
valid_{\tau.(t,p:=q,up)} &:= valid_\tau \cup \{p\} && \text{if } q \in valid_\tau \\
valid_{\tau.(t,p:=q,up)} &:= valid_\tau \setminus \{p\} && \text{if } q \notin valid_\tau \\
valid_{\tau.(t,p.\texttt{next}:=q,up)} &:= valid_\tau \cup \{a.\texttt{next}\} && \text{if } \mathfrak{m}_\tau(p) = a \in Adr \wedge q \in valid_\tau \\
valid_{\tau.(t,p.\texttt{next}:=q,up)} &:= valid_\tau \setminus \{a.\texttt{next}\} && \text{if } \mathfrak{m}_\tau(p) = a \in Adr \wedge q \notin valid_\tau \\
valid_{\tau.(t,p:=q.\texttt{next},up)} &:= valid_\tau \cup \{p\} && \text{if } \mathfrak{m}_\tau(q) = a \in Adr \wedge a.\texttt{next} \in valid_\tau \\
valid_{\tau.(t,p:=q.\texttt{next},up)} &:= valid_\tau \setminus \{p\} && \text{if } \mathfrak{m}_\tau(q) = a \in Adr \wedge a.\texttt{next} \notin valid_\tau \\
valid_{\tau.(t,\texttt{free}(a),up)} &:= valid_\tau \setminus invalid_a \\
valid_{\tau.(t,p:=\texttt{malloc},up)} &:= valid_\tau \cup \{p, a.\texttt{next}\} && \text{if } [p \mapsto a] \in up \\
valid_{\tau.(t,\texttt{assume } p=q,up)} &:= valid_\tau \cup \{p, q\} && \text{if } \{p, q\} \cap valid_\tau \neq \emptyset \\
valid_{\tau.(t,act,up)} &:= valid_\tau && \text{otherwise.}
\end{aligned}$$

We have $invalid_a := \{p \mid \mathfrak{m}_\tau(p) = a\} \cup \{b.\texttt{next} \mid \mathfrak{m}_\tau(b.\texttt{next}) = a\} \cup \{a.\texttt{next}\}$.

Definition A.9 (Observer Behavior). The behavior allowed by O on address a after history h, denoted by $\mathcal{F}_O(h, a)$, is the set $\mathcal{F}_O(h, a) := \{h' \mid h.h' \in \mathcal{S}(O) \wedge frees(h') \subseteq a\}$. If clear from the context, we just write $\mathcal{F}(h, a)$.

Definition A.10 (Unsafe Access). A computation $\tau.act$ raises an *unsafe access* if $com(act)$ contains $p.\texttt{data}$ or $p.\texttt{next}$ with $p \notin valid_\tau$.

Definition A.11 (Unsafe Assumption). A computation $\tau.act$ raises an *unsafe assumption* if $com(act)$ is of the form $\texttt{assume } p = q$ with $\{p, q\} \not\subseteq valid_\tau$.

Definition A.12 (Unsafe Retire). A computation $\tau.act$ raises an *unsafe retire* if $com(act)$ is of the form $\texttt{enter retire}(p)$ with $p \notin valid_\tau$.

Definition A.13 (Pointer Race). A computation $\tau.act$ raises a *pointer race (PR)* if it raises (i) an unsafe access, (ii) an unsafe assumption, (iii) an unsafe call, or (iv) an unsafe retire. It is *pointer race free (PRF)* if none of its prefixes raises a PR.

Definition A.14 (Renaming). A *renaming of address a and b in a history h*, denoted by $h[a/b]$, replaces in h every occurrence of a with b, and vice versa, as follows:

$$\begin{aligned}\epsilon[a/b] &= \epsilon\\ \big(h.func(\bar{c},\bar{d})\big)[a/b] &= \big(h[a/b]\big).\big(func(\bar{c}[a/b],\bar{d})\big)\\ \big(h.\texttt{free}(c)\big)[a/b] &= \big(h[a/b]\big).\big(\texttt{free}(c[a/b])\big)\\ h.evt[a/b] &= h[a/b].evt \qquad \text{otherwise.}\end{aligned}$$

where $a[a/b] = b$, $b[a/b] = a$, and $c[a/b] = c$ for all $a \neq c \neq b$.

Definition A.15 (Elision Support). Observer O *supports elision of memory reuse* if
(i) $\mathcal{F}_O(h.\texttt{free}(a), b) = \mathcal{F}_O(h, b)$ for all h, a, b with $a \neq b$ and $h.\texttt{free}(a) \in \mathcal{S}(O)$,
(ii) $\mathcal{F}_O(h, c) = \mathcal{F}_O(h[a/b], c)$ for all h, a, b, c with $a \neq c \neq b$, and
(iii) $\mathcal{F}_O(h, a) \subseteq \mathcal{F}_O(h[a/b], a)$ for all h, a, b with $a \notin \mathit{retired}(h_1)$ and $b \in \mathit{fresh}(h_1)$.
(iv) $\mathcal{F}_O(h.\texttt{free}(a), a) \subseteq \mathcal{F}_O(h, a)$ for all h, a.

Definition A.16. The *domain* of a type environment Γ is defined by $dom(\Gamma) = \{x \mid \exists T.\ x : T \in \Gamma\}$.

Definition A.17. Consider some Γ and $\bar{p} = p_1, \ldots, p_n$ with $\Gamma(p_i) = T_i$. Then we define:

$$\begin{aligned}&\mathit{safeEnter}(\Gamma, \mathit{func}(\bar{p}, \bar{u})) = \mathit{true}\\ \text{iff}\quad &\forall h\ \forall \bar{a}, \bar{b}, c, \bar{d}.\ \big(\forall i.\ (a_i = c \vee \mathit{isValid}(T_i)) \implies a_i = b_i\big)\\ &\qquad \wedge\ \mathcal{F}_O(h.\mathit{func}(t, \bar{b}, \bar{d}), c) \not\subseteq \mathcal{F}_O(h.\mathit{func}(t, \bar{a}, \bar{d}), c)\ .\end{aligned}$$

Definition A.18 (Post Image). The post image for pointers p and angles r is defined by:

$$\mathit{post}_{p,\mathit{com}}(L) := \{l' \mid \exists\, l\, \exists\, \varphi\, \exists\, m.\ (l,\varphi) \xrightarrow{m(\mathit{com}_t)} (l',\varphi)\ \wedge\ l \in L\ \wedge\ \varphi(z_t) = t\ \wedge\ \varphi(z_a) = m(p)\}$$
$$\mathit{post}_{r,\mathit{com}}(L) := \{l' \mid \exists\, l\, \exists\, \varphi\, \exists\, m.\ (l,\varphi) \xrightarrow{m(\mathit{com}_t)} (l',\varphi)\ \wedge\ l \in L\ \wedge\ \varphi(z_t) = t\}$$

where $m(\mathit{com}_t)$ means the event that results from thread t executing com under memory m.

A.2 Reduction

Definition A.19. We write $m(e) = \bot$ if $e \notin dom(m)$.

Definition A.20 (In-Use Addresses). An address a is *in-use* in memory m if m contains a pointer to a. Formally, the addresses in-use are $adr(m) := (range(m) \cup dom(m)) \cap Adr$ where we use $\{a.\texttt{next}\} \cap Adr = a$ and likewise for data selectors.

Definition A.21 (Restrictions). A restriction of m to a set $P \subseteq PExp$, denoted by $m|_P$, is a new m' with $dom(m') := P \cup DVar \cup \{a.\texttt{data} \in DExp \mid a \in m(P)\}$ and $m(e) = m'(e)$ for all $e \in dom(m')$.

Definition A.22 (Computation similarity). Two computations τ and σ are similar, denoted by $\tau \sim \sigma$, if $ctrl(\tau) = ctrl(\sigma)$ and $\mathfrak{m}_\tau|_{valid_\tau} = \mathfrak{m}_\sigma|_{valid_\sigma}$.

Definition A.23 (Observer Behavior Inclusion). Consider $\tau, \sigma \in O[\![P]\!]^{Adr}_{Adr}$. Then, σ *includes the (observer) behavior* of τ, denoted by $\tau \lessdot \sigma$, if $\mathcal{F}_O(\tau, a) \subseteq \mathcal{F}_O(\sigma, a)$ holds for all $a \in adr(\mathfrak{m}_\tau|_{valid_\tau})$.

Definition A.24. $O[\![P]\!]^{\varnothing}_{one} = \bigcup_{a \in Adr} O[\![P]\!]^{\varnothing}_{\{a\}}$

Lemma A.25. $[\![P]\!]^Y_X$ *is prefix closed by Assumption 1.*

Lemma A.26. *Consider $\tau \in [\![P]\!]^{Adr}_{Adr}$. Then, $adr(\mathfrak{m}_\tau|_{valid_\tau}) = (valid_\tau \cap Adr) \cup \mathfrak{m}_\tau(valid_\tau)$.*

Lemma A.27. *Consider $\tau, \sigma \in O[\![P]\!]^{Adr}_{Adr}$. If $\tau \sim \sigma$, then $valid_\tau = valid_\sigma$.*

Lemma A.28. *Let* $evt = func(t, \bar{a}, \bar{d})$. *Then,* $h_1 \in \mathcal{F}_O(h_2.evt, a)$ *iff* $evt.h_1 \in \mathcal{F}_O(h_2, a)$.

Lemma A.29. *Consider* $\tau \in [\![P]\!]^{\varnothing}_{Adr}$ *PRF. Then,* $adr(\mathfrak{m}_\tau|_{valid_\tau}) \cap \mathfrak{m}_\tau(PExp \setminus valid_\tau) = \varnothing$.

Lemma A.30. *If* $\tau \in [\![P]\!]^{Adr}_{Adr}$ *and* $\mathfrak{m}_\tau(pexp) \in freed(\tau)$, *then* $pexp \notin valid_\tau$.

Lemma A.31. *Let* $\tau \in O[\![P]\!]^{Adr}_{Adr}$, $a \in (fresh(\tau) \cup freed(\tau)) \setminus retired(\tau)$, *and* $b \in fresh(\tau)$. *If* O *supports elision, then* $\mathcal{F}_O(\tau, a) \subseteq \mathcal{F}_O(\tau, b)[b/a]$.

Lemma A.32. *Assume* O *supports elision. Let* $\tau \in O[\![P]\!]^{\varnothing}_{Adr}$, $a \notin adr(\mathfrak{m}_\tau|_{valid_\tau})$, *and* $A \subseteq Adr$ *with* $|A| < \infty$. *Then there is* $\sigma \in O[\![P]\!]^{\varnothing}_{Adr}$ *and* $b \in fresh(\tau) \setminus A$ *with:*

- $\tau \sim \sigma$ *and* $\tau \lessdot \sigma$ *and* $retired(\tau) \subseteq retired(\sigma) \cup \{a\}$ *and* $a \in fresh(\sigma)$,
- $b \in fresh(\sigma) \iff a \in fresh(\tau)$ *and* $fresh(\sigma) \setminus \{a, b\} = fresh(\tau) \setminus \{a, b\}$,
- $\mathcal{F}_O(\tau, a) = \mathcal{F}_O(\sigma, b)[b/a]$ *and* $\mathcal{F}_O(\tau, b)[b/a] = \mathcal{F}_O(\sigma, a)$,
- $\forall c.\ a \neq c \neq b \implies \mathcal{F}_O(\tau, c) = \mathcal{F}_O(\sigma, c)$, *and*
- $\forall e, e' \in PVar \cup \{c.\texttt{next} \mid c \in \mathfrak{m}_\tau(valid_\tau)\}.\ \mathfrak{m}_\tau(e) \neq \mathfrak{m}_\tau(e') \implies \mathfrak{m}_\sigma(e) \neq \mathfrak{m}_\sigma(e')$.

Lemma A.33. *Assume that* O *supports elision and that* $O[\![P]\!]^{\varnothing}_{Adr}$ *is PRF. Then, for every* $\tau \in O[\![P]\!]^{Adr}_{Adr}$ *there is some* $\sigma \in O[\![P]\!]^{\varnothing}_{Adr}$ *with* $\tau \sim \sigma$, $\tau \lessdot \sigma$, *and* $retired(\tau) \subseteq retired(\sigma)$. *Moreover,* $\mathfrak{m}_\tau(e) \neq \mathfrak{m}_\tau(e')$ *implies* $\mathfrak{m}_\sigma(e) \neq \mathfrak{m}_\sigma(e')$ *for all* $e, e' \in PVar \cup \{b.\texttt{next} \mid b \in \mathfrak{m}_\tau(valid_\tau)\}$.

Lemma A.34. *If* $\tau \in [\![P]\!]^{Adr}_{Adr}$ *and* $a \in fresh(\tau)$, *then* $a \notin range(\mathfrak{m}_\tau)$.

Lemma A.35. *If* $\tau \in [\![P]\!]^{Adr}_{Adr}$, *then* $fresh(\tau) \cap retired(\tau) = \varnothing$.

Lemma A.36. *If* $\tau \in [\![P]\!]^{\varnothing}_{Adr}$ *PRF and* $pexp \in PExp$ *with* $pexp \notin valid_\tau$, *then* $\mathfrak{m}_\tau(pexp) \in freed(\tau)$ *or* $pexp \equiv a.\texttt{next} \wedge a \in fresh(\tau) \cup freed(\tau)$.

Lemma A.37. *If* $\tau \in [\![P]\!]^{\varnothing}_{Adr}$ *PRF and* $p \in PVar$ *with* $p \notin valid_\tau$, *then* $\mathfrak{m}_\tau(p) \in freed(\tau)$.

Lemma A.38. *Assume* O *supports elision. Consider* $\tau.act \in O[\![P]\!]^{Adr}_{Adr}$ *with* $com(act) = \texttt{free}(a)$. *Let* $\mathcal{H}(\tau) = h$. *Then,* $\mathcal{S}(h.\texttt{free}(a)) \subseteq \mathcal{S}(h)$.

Lemma A.39. *Consider some* $\tau \in O[\![P]\!]^{\varnothing}_{Adr}$ *and some* $a \in Adr$. *Let* $\varphi = \{z_a \mapsto a\}$. *Then:*

$$(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (L_3, \varphi) \iff a \in retired(\tau)$$
$$(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (L_2, \varphi) \iff a \notin retired(\tau)$$
$$a \in active(\tau) \implies (L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (L_2, \varphi)$$

Lemma A.40. *Let* $\tau.(t, \texttt{free}(a), up) \in O[\![P]\!]^{\varnothing}_{Adr}$. *If* $O = O_{Base} \times O_{Impl}$, *then* $a \in retired(\tau)$.

Lemma A.41. *Assume* O *supports elision. Then, for every* $\tau \in O[\![P]\!]^{\varnothing}_{Adr}$ *there is* $\sigma \in [\![P]\!]^{\varnothing}_{\varnothing}$ *with: (i)* $ctrl(\tau) = ctrl(\sigma)$, *(ii)* $\mathfrak{m}_\tau = \mathfrak{m}_\sigma$, *and (iii)* $fresh(\tau) \subseteq fresh(\sigma)$. *Moreover, if* $O = O_{Base} \times O_{Impl}$, *then (iv)* $retired(\tau) \subseteq retired(\sigma)$, *(v)* $freed(\tau) \subseteq retired(\sigma)$, *and (vi)* $inv(\sigma) \implies inv(\tau)$.

Theorem A.42 (Formalization of Theorem 4.1). *If* O *supports elision and the semantics* $O[\![P]\!]^{\varnothing}_{Adr}$ *is pointer-race-free, then* $O[\![P]\!]^{Adr}_{Adr} \sim [\![P]\!]^{\varnothing}_{\varnothing}$.

A.3 Type System

In this section we assume, if not stated otherwise, a fixed program P and a fixed SMR automaton O to avoid notational clutter. A generalization to arbitrary programs is straight forward. Recall from Section 3 that we assume that O is of the form $O = O_{Base} \times O_{Impl}$ for some SMR automaton O_{Impl}.

*Definition A.43 (*skip*).* We use the skip as syntactic sugar for a command that has no effect, for example, assume $u = u$ where u is some data variable. We assume that $\vdash \Gamma\ \{\texttt{skip}\}\ \Gamma$ holds for all Γ.

$$\text{(sos1)}\quad \frac{act = (t, com, up)}{(com, \tau) \dashrightarrow_t (\texttt{skip}, \tau.act)} \qquad \text{(sos2)}\quad \frac{}{(\texttt{skip}; stmt, \tau) \dashrightarrow_t (stmt, \tau)} \qquad \text{(sos3)}\quad \frac{i \in \{1, 2\}}{(stmt_1 \oplus stmt_2, \tau) \dashrightarrow_t (stmt_i, \tau)}$$

$$\text{(sos4)}\quad \frac{stmt_1 \not\equiv \texttt{skip} \qquad (stmt_1, \tau) \dashrightarrow_t (stmt_1', \tau')}{(stmt_1; stmt_2, \tau) \dashrightarrow_t (stmt_1'; stmt_2, \tau')} \qquad \text{(sos5)}\quad \frac{stmt' \in \{\texttt{skip}, stmt, stmt; stmt^*\}}{(stmt^*, \tau) \dashrightarrow_t (stmt', \tau)}$$

$$\text{(sos6)}\quad \frac{pc(t) = stmt \qquad (stmt, \tau) \dashrightarrow_t (stmt', \tau') \qquad lock(\tau) = \{t\}}{(pc, \tau) \dashrightarrow_t (pc[t \mapsto stmt'], \tau')}$$

$$\text{(sos7)}\quad \frac{pc(t) = stmt \qquad (stmt, \tau) \dashrightarrow_t (stmt', \tau) \qquad lock(\tau) = \varnothing}{(pc, \tau) \dashrightarrow_t (pc[t \mapsto stmt'], \tau)}$$

$$\text{(sos8)}\quad \frac{pc(t) = stmt \qquad (stmt, \tau) \dashrightarrow_t (stmt', \tau.\tau') \qquad lock(\tau) = \varnothing \qquad lock(\tau.\tau') = \{t\}}{(pc, \tau) \dashrightarrow_t (pc[t \mapsto stmt'], \tau.\tau')}$$

$$\text{(sos9)}\quad \frac{act = (\bot, \texttt{free}(a), \varnothing) \qquad lock(\tau) = \varnothing}{(pc, \tau) \dashrightarrow_\bot (pc, \tau.act)}$$

$$\begin{aligned}
lock(\epsilon) &:= \varnothing \\
lock(\tau.act) &:= lock(\tau) \cup \{t\} && \text{if } act = (t, \texttt{beginAtomic}, up) \\
lock(\tau.act) &:= lock(\tau) \setminus \{t\} && \text{if } act = (t, \texttt{endAtomic}, up) \\
lock(\tau.act) &:= lock(\tau) && \text{otherwise}
\end{aligned}$$

Fig. 10. The SOS rules for the transition relation $\dashrightarrow$ among configurations.

Definition A.44. Indexing a (pointer/angel/data) variable var by a thread t yields a new variable var_t. Indexing all non-shared variables $var \notin shared$ by t in P gives a new program $P^{[t]}$.

Definition A.45. The thread-local variables of t is the set $local_t = \{p_t \mid p \notin shared\}$ of non-shared variables indexed by t.

Definition A.46. The initial program counter is pc_{init} with $pc_{init}(t) = P^{[t]}$ for all threads t.

Assumption 2. *We assume that ghost variables r are local, that is, $r \notin shared$.*

Definition A.47. The initial type environment for P is Γ_{init}. For $P^{[t]}$ it is $\Gamma_{init}^{[t]}$. Formally:

$$\Gamma_{init} := \{x : \varnothing \mid x \in PVar \cup AVar\}$$

$$\Gamma_{init}^{[t]} := \{p : \varnothing \mid p \in PVar \cap shared\} \cup \{p_t : \varnothing \mid p \in PVar \setminus shared\} \cup \{r_t : \varnothing \mid r \in AVar\}$$

Definition A.48. We define $ctrl(\tau) = \{pc \mid (pc_{init}, \epsilon) \dashrightarrow^* (pc, \tau)\}$ where $\dashrightarrow$ is the transition relation among configurations from Figure 10. Then, $ctrl_t(\tau) = \{pc(t) \mid pc \in ctrl(\tau)\}$.

Assumption 3. *We assume that computations adhere to the control flow. Formally, this means* $ctrl(\tau) \neq \emptyset$ *for all* $\tau \in [\![P]\!]^{Adr}_{Adr}$.

Remark 1. *Assumption 3 requires that all primitive commands are wrapped inside atomics, that is, occur somewhere between* `beginAtomic` *and* `endAtomic`.

Definition A.49. A computation $\tau \in [\![P]\!]^{Adr}_{Adr}$ induces a flat line program for thread t, denoted by $flat_t(\tau)$, as follows:

$$\begin{aligned} flat_t(\epsilon) &:= \texttt{skip} \\ flat_t(\tau.act) &:= flat_t(\tau); com && \text{if } act = (t, com, up) \\ flat_t(\tau.act) &:= flat_t(\tau) && \text{if } act = (t', com, up) \wedge t \neq t' \end{aligned}$$

Definition A.50. A pointer p has no valid alias in a computation τ, denoted by $noalias_\tau(p)$, if $\texttt{seg} \neq \mathfrak{m}_\tau(p) \notin \mathfrak{m}_\tau(valid_\tau \setminus \{p\})$.

Definition A.51. The locations reached in O by a history h wrt. to some thread t and some address a is defined by $reach_{O,t,a}(h) := \{l \mid \exists \varphi.\ (l_{init}, \varphi) \xrightarrow{h} (l, \varphi) \wedge \varphi(z_t) = t \wedge \varphi(z_a) = a\}$ where l_{init} is the initial location in O. For seg we define $reach_{O,t,a}(h) = \top$ to contain all locations of O. The definition of *reach* extends naturally to sets of histories.

Lemma A.52. *If* $\Gamma_1 \rightsquigarrow \Gamma_2$ *and* $\Gamma_2 \rightsquigarrow \Gamma_3$, *then* $\Gamma_1 \rightsquigarrow \Gamma_3$.

Lemma A.53. *Consider* $\vdash \Gamma_1\ \{stmt\}\ \Gamma_2$ *and* $(stmt, \tau) \dashrightarrow_t (stmt', \tau.\tau')$. *Then there is* Γ *such that* $\vdash \Gamma_1\ \{flat_t(\tau')\}\ \Gamma$ *and* $\vdash \Gamma\ \{stmt'\}\ \Gamma_2$.

Lemma A.54. *Let* $\vdash \Gamma_{init}\ \{P\}\ \Gamma$. *Consider* $(pc_{init}, \epsilon) \dashrightarrow^* (pc, \tau)$ *and some thread* t. *Then there is* Γ_1, Γ_2 *with* $\vdash \Gamma^{[t]}_{init}\ \{flat_t(\tau)\}\ \Gamma_1$ *and* $\vdash \Gamma_1\ \{pc(t)\}\ \Gamma_2$.

Lemma A.55. *Let* $\tau.act \in [\![P]\!]^{\emptyset}_{Adr}$ *and* $t \neq thrd(act)$. *Let* $\vdash \Gamma^{[t]}_{init}\ \{flat_t(\tau)\}\ \Gamma$ *and* $x \in PVar \cup AVar$. *Then* $\mathbb{A} \notin \Gamma(x)$ *and* $x \notin local_t \implies \Gamma(x) \cap \{\mathbb{L}, \mathbb{S}\} = \emptyset$.

Lemma A.56. *Let* $\tau.act \in [\![P]\!]^{\emptyset}_{Adr}$ *and* $t \neq thrd(act) \neq \bot$. *Let* $p \in PVar \cap local_t$. *Then,* $p \in valid_\tau$ *implies* $p \in valid_{\tau.act}$. *Moreover,* $noalias_\tau(p)$ *implies* $noalias_{\tau.act}(p)$.

Lemma A.57. *Consider* $\tau.act \in [\![P]\!]^{\emptyset}_{Adr}$ *PRF with* $act = (t, \texttt{@inv}\ p = q, up)$ *and* $inv(\tau.act)$. *Then,* $\{p, q\} \cap valid_\tau \neq \emptyset$ *implies* $\{p, q\} \subseteq valid_\tau$.

Lemma A.58. *Consider* $\tau.act \in O[\![P]\!]^{\emptyset}_{Adr}$ *PRF with* $act = (t, \texttt{@inv active}(p), up)$ *and* $inv(\tau.act)$. *Then,* $p \in valid_{\tau.act}$ *and* $reach_{O,t,a}(\mathcal{H}(\tau.act)) \subseteq Loc(\mathbb{A})$ *for* $a = \mathfrak{m}_{\tau.act}(p)$.

Lemma A.59. *Consider* $\tau.act \in O[\![P]\!]^{\emptyset}_{Adr}$ *PRF with* $act = (t, \texttt{@inv active}(r), up)$ *and* $inv(\tau.act)$. *Then,* $repr_{\tau.act}(r) \cap freed(\tau.act) = \emptyset$ *and* $reach_{O,t,a}(\mathcal{H}(\tau.act)) \subseteq Loc(\mathbb{A})$ *for all* $a \in repr_{\tau.act}(r)$.

Lemma A.60. *Let* $\tau \in O[\![P]\!]^{\emptyset}_{Adr}$. *Let* Γ, Γ' *such that* $\Gamma \rightsquigarrow \Gamma'$. *Let* t *be some thread. Let* $p \in PVar$ *and* $a = \mathfrak{m}_\tau(p)$. *Let* $r \in AVar$ *and* $b \in repr_\tau(r)$. *Then,*

$$\begin{array}{rcl} isValid(\Gamma(p)) \implies p \in valid_\tau & \textit{implies} & isValid(\Gamma'(p)) \implies p \in valid_\tau \\ isValid(\Gamma(r)) \implies b \notin freed(\tau) & \textit{implies} & isValid(\Gamma'(r)) \implies b \notin freed(\tau) \\ \mathbb{L} \in \Gamma(p) \implies noalias_\tau(p) & \textit{implies} & \mathbb{L} \in \Gamma'(p) \implies noalias_\tau(p) \\ reach_{O,t,a}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma(p)) & \textit{implies} & reach_{O,t,a}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma'(p)) \\ reach_{O,t,b}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma(r)) & \textit{implies} & reach_{O,t,b}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma'(r)) \end{array}$$

Lemma A.61. *Assume O supports elision, and $inv([\![P]\!]^{\emptyset}_{\emptyset})$. Consider some thread t, some type environments Γ, and some $\tau \in O[\![P]\!]^{\emptyset}_{Adr}$ PRF with $inv(\tau)$ and $\vdash \Gamma^{[t]}_{init}\ \{flat_t(\tau)\}\ \Gamma$. Then, for every $p \in PVar$, we have $reach_{O,t,a}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma(p))$ and $isValid(\Gamma(p)) \implies p \in valid_\tau$.*

Lemma A.62. *Let O supports elision. If $\vdash P$ and $inv([\![P]\!]^{\emptyset}_{\emptyset})$, then $O[\![P]\!]^{\emptyset}_{Adr}$ PRF and $inv(O[\![P]\!]^{\emptyset}_{Adr})$.*

Lemma A.63. *Let O supports elision. If $\vdash P$ and $inv([\![P]\!]^{\emptyset}_{\emptyset})$, $[\![P]\!]^{Adr}_{Adr}$ does not perform double retires.*

B PROOFS

B.1 Reduction

Proof of Lemma A.25. Follows immediately from Assumption 1 as it guarantees that continuations of a history not accepted by O are also not accepted. □

Proof of Lemma A.26. Follows from [Meyer and Wolff 2018, Lemma D.5]. □

Proof of Lemma A.27. Follows from [Meyer and Wolff 2018, Lemma D.7] □

Proof of Lemma A.28. Follows from definition. □

Proof of Lemma A.29. Follows from [Meyer and Wolff 2018, Lemma D.15]. That the semantics from [Meyer and Wolff 2018] sets selectors to $\bot$ for `free` commands does not affect the result. □

Proof of Lemma A.30. To the contrary, assume there is a shortest $\tau.act \in [\![P]\!]^{Adr}_{Adr}$ with some address $a \in freed(\tau.act)$ and $a \in \mathfrak{m}_{\tau.act}(valid_{\tau.act})$. Note that $\tau.act$ is indeed the shortest such computation since the claim is vacuously true for ϵ.

First, consider the case where we have $a \notin freed(\tau)$. Then, act must execute the command $\texttt{free}(a)$. As a consequence, we get $pexp \notin valid_{\tau.act}$ for all $pexp$ with $\mathfrak{m}_\tau(pexp) = a$. So $a \notin \mathfrak{m}_{\tau.act}(valid_{\tau.act})$. Since this contradicts the assumption, we must have $a \in freed(\tau)$.

Now, consider the case where we have $a \in freed(\tau)$. By definition, there is some $pexp \in valid_{\tau.act}$ with $\mathfrak{m}_{\tau.act}(pexp) = a \neq \texttt{seg}$. We get $pexp \notin valid_\tau$ by minimality of $\tau.act$. That is, act validates $pexp$. To do so, act must be an assignment, an allocation, or an assertion:

- If act is of the form $act = (t, pexp := qexp, up)$, then $qexp \in valid_\tau$ and $\mathfrak{m}_\tau(qexp) = a$ must hold in order to establish the desired properties of $pexp$. However, $\mathfrak{m}_\tau(qexp)$ leads to $qexp \notin valid_\tau$ by minimality of $\tau.act$. Hence, act cannot be an assignment.
- If act is of the form $act = (t, pexp := \texttt{malloc}, up)$, then $a \notin freed(\tau.act)$ by definition. Hence, act cannot be an allocation targeting $pexp$.
- If act is of the form $act = (t, p := \texttt{malloc}, up)$ with $\mathfrak{m}_{\tau.act}(p) = b$ and $pexp \equiv b.\texttt{next}$, then $\mathfrak{m}_{\tau.act}(pexp) = \texttt{seg}\neg a$. Hence, act cannot be an allocation.
- If act is of the form $act = (t, \texttt{assume}\ p = q, up)$, then wlog. $pexp \equiv p$ and $q \in valid_\tau$ and $a = \mathfrak{m}_{\tau.act}(pexp) = \mathfrak{m}_\tau(pexp) = \mathfrak{m}_\tau(q)$ must hold. Again by minimality, we get $q \notin valid_\tau$ which contradicts the assumption. Hence, act cannot be an assertion.

The above case distinction is complete and thus concludes the claim. □

Proof of Lemma A.31. Let $\tau \in O[\![P]\!]^{Adr}_{Adr}$, $a \in (fresh(\tau) \cup freed(\tau)) \setminus retired(\tau)$, and $b \in fresh(\tau)$. Let $\mathcal{H}(\tau) = h$. We have $\mathcal{F}_O(h, b)[b/a] = \mathcal{F}_O(h[b/a], a)$ according to [Meyer and Wolff 2018, Lemma D.26]. If $a \in freed(\tau)$, then Definition A.15iii yields $\mathcal{F}_O(h, a) \subseteq \mathcal{F}_O(h[b/a], a)$. Thus, $\mathcal{F}_O(h, a) \subseteq \mathcal{F}_O(h, b)[b/a]$ as desired. Otherwise, we have $a \in fresh(\tau)$. This means $h[b/a] = h$. So, $\mathcal{F}_O(h[b/a], a) = \mathcal{F}_O(h, a)$. Hence, $\mathcal{F}_O(h, a) = \mathcal{F}_O(h, b)[b/a]$ as desired. □

Proof of Lemma A.32. Follows from [Meyer and Wolff 2018, Lemmas D.26, D.27 and D.28]. That the semantics from [Meyer and Wolff 2018] sets selectors to $\bot$ for `free` commands does not affect the result. □

Proof of Lemma A.33. Follows from [Meyer and Wolff 2018, Proofs of Proposition C.14, Lemma C.16, and Lemma D.16]. □

Proof of Lemma A.34. Follows from [Meyer and Wolff 2018, Lemma D.9]. □

Proof of Lemma A.35. The claim holds for the empty computation ϵ. To the contrary, assume the claim does not hold. Then, there must be a shortest computation $\tau.act \in [\![P]\!]^{Adr}_{Adr}$ such that $fresh(\tau.act) \cap retired(\tau.act) \neq \emptyset$. Let $a \in fresh(\tau.act) \cap retired(\tau.act)$. By minimality of $\tau.act$, we must have $a \notin fresh(\tau)$ or $a \notin retired(\tau)$. If $a \notin fresh(\tau)$, then we have $a \notin fresh(\tau.act)$ by definition. Since this contradicts the assumption, we must have $a \in fresh(\tau)$ and $a \notin retired(\tau)$. To arrive at $a \in retired(\tau.act)$, act must be of the form $act = (t, \texttt{enter retire}(p), up)$ with $\mathfrak{m}_\tau(p) = a$. By the contrapositive of Lemma A.34, we have $a \notin fresh(\tau)$. As before, this gives $a \notin fresh(\tau.act)$ and contradicts the assumption. □

Proof of Lemma A.36. The claim holds for the empty computation ϵ by definition. Consider some $\tau.act \in [\![P]\!]^{\emptyset}_{Adr}$ PRF with $act = (t, com, up)$ such that for every $qexp \in PExp$ we have $qexp \notin valid_\tau$ implies $\mathfrak{m}_\tau(qexp) \in freed(\tau)$ or $qexp \notin PVar \wedge \{qexp\} \cap Adr \subseteq fresh(\tau) \cup freed(\tau)$. Let $pexp \in PExp$ be some pointer expression with $pexp \notin valid_{\tau.act}$. We do a case distinction.

- Consider com being an assignment. If com does not contain $pexp$ at the left-hand side, then $pexp \notin valid_\tau$. Hence, we have either $\mathfrak{m}_{\tau.act}(pexp) = \mathfrak{m}_\tau(pexp) \in freed(\tau) = freed(\tau.act)$ or $pexp \notin PVar \wedge \{pexp\} \cap Adr \subseteq fresh(\tau) \cup freed(\tau) = fresh(\tau.act) = freed(\tau.act)$. Otherwise, $com \equiv pexp := qexp$. If $qexp \in PVar$, then $qexp \notin valid_\tau$. Thus, $\mathfrak{m}_\tau(qexp) \in freed(\tau)$. We get $\mathfrak{m}_{\tau.act}(pexp) \in freed(\tau.act)$. Otherwise, $qexp \equiv p.\texttt{next}$ with $\mathfrak{m}_\tau(p) = a$ and $a.\texttt{next} \notin valid_\tau$. By Lemma A.34 we have $a \notin fresh(\tau)$. So we get $\mathfrak{m}_\tau(qexp) \in freed(\tau)$ or $a \in freed(\tau)$. The latter cannot apply since Lemma A.30 gives $p \notin valid_\tau$ which means $\tau.act$ raises a pointer race contradicting the assumption. In the remaining case we get $\mathfrak{m}_{\tau.act}(pexp) \in freed(\tau.act)$ as desired.
- Consider com being an `assume`, an `@inv`, an `enter`, or an `exit`. Then, we know that $pexp \notin valid_\tau$, $\mathfrak{m}_\tau(pexp) = \mathfrak{m}_{\tau.act}(pexp)$, $fresh(\tau) = fresh(\tau.act)$, and $freed(\tau) = freed(\tau.act)$. This implies the desired property.
- Consider com being an allocation. Let $com \equiv p := \texttt{malloc}$. The update is of the form $up = [p \mapsto a, a.\texttt{next} \mapsto \texttt{seg}, dots]$ for some $a \in Adr$. By the semantics, we have $a \in fresh(\tau)$. We get $fresh(\tau.act) = fresh(\tau) \setminus \{a\}$ and $freed(\tau.act) = freed(\tau) \setminus \{a\}$. Since $pexp \notin valid_{\tau.act}$, we have $pexp \notin \{p, a.\texttt{next}\}$. So $\mathfrak{m}_\tau(pexp) = \mathfrak{m}_{\tau.act}(pexp)$. If $\mathfrak{m}_\tau(pexp) \in freed(\tau)$ we get $\mathfrak{m}_\tau(pexp) \notin fresh(\tau)$ and thus $\mathfrak{m}_{\tau.act}(pexp) \in freed(\tau.act)$. Otherwise, $pexp \equiv b.\texttt{next}$ and $b \in fresh(\tau) \cup freed(\tau)$ with $a \neq b$. So we get the desired $b \in fresh(\tau.act) \cup freed(\tau.act)$.
- Consider com being a free. Let $com \equiv \texttt{free}(a)$. If $pexp \in valid_\tau$, then we have $\mathfrak{m}_\tau(pexp) = a$ or $pexp \equiv a.\texttt{next}$ since act invalidates $pexp$. We get either $\mathfrak{m}_{\tau.act}(pexp) \in freed(\tau.act)$ or $pexp \equiv a.\texttt{next} \wedge a \in freed(\tau.act)$. Otherwise, we have $pexp \notin valid_\tau$. If $\mathfrak{m}_\tau(pexp) \in freed(\tau)$, then $\mathfrak{m}_{\tau.act}(pexp) \in freed(\tau.act)$. Otherwise, $pexp \equiv b.\texttt{next} \wedge b \in fresh(\tau) \cup freed(\tau)$. This gives $b \in fresh(\tau.act) \cup freed(\tau.act)$ because $a = b$ yields $b \in freed(\tau.act)$ and $a \neq b$ does not affect the freshness/freedness of b.

This concludes the claim. □

Proof of Lemma A.37. Follows from Lemma A.36. □

Proof of Lemma A.38. By assumption we have

$$\forall h\, \forall a, b.\ a \neq b \implies \mathcal{F}_O(h.\mathtt{free}(a), b) = \mathcal{F}_O(h, b) \tag{A1}$$

$$\forall h\, \forall a.\ \mathcal{F}_O(h.\mathtt{free}(a), a) \subseteq \mathcal{F}_O(h, a) \tag{A2}$$

Consider some $h.\mathtt{free}(a)$. We show $\mathcal{S}(h.\mathtt{free}(a)) \subseteq \mathcal{S}(h)$ as this implies the claim. Towards a contradiction, assume the inclusion does not hold. That is, there is a shortest $h' \in \mathcal{S}(h.\mathtt{free}(a))$ with $h' \notin \mathcal{S}(h)$. If $\mathit{frees}(h') = \emptyset$, then we get $h' \in \mathcal{F}_O(h.\mathtt{free}(a), a)$. By Auxiliary (A2) we have $h' \in \mathcal{F}_O(h, a)$. This means $h' \in \mathcal{S}(h)$ by definition. This contradicts the choice of h'. So we must have $\mathit{frees}(h') \neq \emptyset$.

Consider now $\mathit{frees}(h') \neq \emptyset$. That is, there is a decomposition of h' of the form $h' = h_1.\mathtt{free}(b).h_2$ with $\mathit{frees}(h_2) = \emptyset$. We derive the following.

- We have $h_1.\mathtt{free}(b).h_2 \notin \mathcal{S}(h)$. That is, $h.h_1.\mathtt{free}(b).h_2 \notin \mathcal{S}(O)$. By definition, this means $h_2 \notin \mathcal{F}_O(h.h_1.\mathtt{free}(b), c)$ with $c \neq b$. Now, Auxiliary (A1) yields $h_2 \notin \mathcal{F}_O(h.h_1, c)$. Since $\mathit{frees}(h_2) = \emptyset$, we must have $h.h_1.h_2 \notin \mathcal{S}(O)$. That is, we get $h_1.h_2 \notin \mathcal{S}(h)$.
- We have $h_1.\mathtt{free}(b).h_2 \in \mathcal{S}(h.\mathtt{free}(a))$. So, $h.\mathtt{free}(a).h_1.\mathtt{free}(b).h_2 \in \mathcal{S}(O)$. By $\mathit{frees}(h_2)$, we get $h_2 \in \mathcal{F}_O(h.\mathtt{free}(a).h_1.\mathtt{free}(b), b)$. Then, Auxiliary (A2) yields $h_2 \in \mathcal{F}_O(h.\mathtt{free}(a).h_1, b)$. So, $h.\mathtt{free}(a).h_1.h_2 \in \mathcal{S}(O)$. That is, we get $h_1.h_2 \in \mathcal{S}(h.\mathtt{free}(a))$.

Altogether, this means we have $h_1.h_2 \in \mathcal{S}(h.\mathtt{free}(a))$ and $h_1.h_2 \notin \mathcal{S}(h)$ with $h_1.h_2$ being shorter than $h_1.\mathtt{free}(b).h_2$. This contradicts the minimality of h' and thus concludes the claim. □

Proof of Lemma A.39. Let $a \in \mathit{Adr}$ and $\varphi = \{z_a \mapsto a\}$. The claim holds for ϵ. Towards a contradiction, assume there is a shortest $\tau.\mathit{act} \in O[\![P]\!]^{\emptyset}_{\mathit{Adr}}$ with $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau.\mathit{act})} (L_3, \varphi)$ and $a \notin \mathit{retired}(\tau.\mathit{act})$. If $\mathcal{H}(\tau.\mathit{act}) = \mathcal{H}(\tau)$, then we have $\mathit{retired}(\tau.\mathit{act}) = \mathit{retired}(\tau)$. This contradicts the assumption that $\tau.\mathit{act}$ is the shortest such computation. So $\mathcal{H}(\tau.\mathit{act})$ is of the form $\mathcal{H}(\tau.\mathit{act}) = h.\mathit{evt}$ with $h = \mathcal{H}(\tau)$. We do a case distinction on the state after τ.

- Consider $(L_2, \varphi) \xrightarrow{h} (L_1, \varphi)$. By definition of O_{Base}, there is not step $(L_1, \varphi) \xrightarrow{\mathit{evt}} (L_3, \varphi)$. Hence, this case cannot apply.
- Consider $(L_2, \varphi) \xrightarrow{h} (L_2, \varphi)$. Then we must have $(L_2, \varphi) \xrightarrow{\mathit{evt}} (L_3, \varphi)$. This means evt is of the form $\mathit{evt} = \mathtt{retire}(t, a)$ for some thread t. By definition, $\mathit{act} = (t, \mathtt{retire}(p), \emptyset)$ with $\mathfrak{m}_\tau(p) = a$. Thus, $a \in \mathit{retired}(\tau.\mathit{act})$. This contradicts the assumption.
- Consider $(L_2, \varphi) \xrightarrow{h} (L_3, \varphi)$. By minimality, we have $a \in \mathit{retired}(\tau)$. To arrive at $a \notin \mathit{retired}(\tau)$, we must have $\mathit{evt} = \mathtt{free}(a)$. This, however, leads to $(L_3, \varphi) \xrightarrow{\mathit{evt}} (L_2, \varphi)$. So, $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau.\mathit{act})} (L_2, \varphi)$. This contradicts the assumption.

The above case distinction is complete and thus proves that $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (L_3, \varphi)$ implies $a \in \mathit{retired}(\tau)$. Consider now the reverse direction. To that end, consider some $\tau \in O[\![P]\!]^{\emptyset}_{\mathit{Adr}}$ and some $a \notin \mathit{retired}(\tau)$. Using the contrapositive of the above, we get $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (l, \varphi)$ with $l \neq L_3$. By Assumption 1, $l \neq L_1$ as for otherwise $\tau \notin O[\![P]\!]^{\emptyset}_{\mathit{Adr}}$. Hence, $l = L_2$ must hold. This establishes the first equivalence. The second equivalence follow analogously. The remaining property follows from the second equivalence together with the fact that $a \in \mathit{active}(\tau)$ implies $a \notin \mathit{retired}(\tau)$. □

Proof of Lemma A.40. Let $\tau.\mathit{act} \in O[\![P]\!]^{\emptyset}_{\mathit{Adr}}$ with $\mathit{act} = (t, \mathtt{free}(a), \mathit{up})$. Let $\varphi = \{z_a \mapsto a\}$. We have $\mathcal{H}(\tau.\mathit{act}) = h.\mathtt{free}(a)$ with $h = \mathcal{H}(\tau)$. By definition, $h.\mathtt{free}(a) \in \mathcal{S}(O_{\mathit{Base}})$. So we must have $(L_2, \varphi) \xrightarrow{h} (L_3, \varphi)$ as for otherwise $\mathtt{free}(a)$ would take O_{Base} to L_2 and thus give $\tau.\mathit{act} \notin O[\![P]\!]^{\emptyset}_{\mathit{Adr}}$. Now Lemma A.39 yields the desired $a \in \mathit{retired}(\tau)$. □

Proof of Lemma A.41. For $\tau = \epsilon$ we choose $\epsilon = \sigma \in [\![P]\!]^{\emptyset}_{\emptyset}$. Then, σ satisfies the desired properties. Consider now $\tau.\mathit{act} \in O[\![P]\!]^{\emptyset}_{\mathit{Adr}}$. Assume we already constructed $\sigma \in [\![P]\!]^{\emptyset}_{\emptyset}$ with:

(P1) $\mathit{ctrl}(\tau) = \mathit{ctrl}(\sigma)$,

(P2) $\mathfrak{m}_\tau = \mathfrak{m}_\sigma$,

(P3) $\mathit{fresh}(\tau) \subseteq \mathit{fresh}(\sigma)$,
(P4) $\mathit{retired}(\tau) \subseteq \mathit{retired}(\sigma)$, and
(P5) $\mathit{freed}(\tau) \subseteq \mathit{retired}(\sigma)$.
(P6) $\mathit{inv}(\sigma) \implies \mathit{inv}(\tau)$.

First, assume $\mathit{com}(\mathit{act}) \not\equiv \texttt{free}(a)$. We get $\sigma.\mathit{act} \in [\![P]\!]^{\varnothing}_{\varnothing}$. The reason for this is that σ:

- performs the same updates in assignments due to Property (P2),
- allows for the same `assume` commands due to Property (P2),
- emits the same events due to Property (P2),
- allows for the same `malloc` commands due to Property (P3), and
- nothing can be removed from $\mathit{retired}(\sigma)$.

Moreover, σ satisfies the desired properties. This is because the same update is applied after τ and σ. To see that $\mathit{inv}(\sigma.\mathit{act}) \implies \mathit{inv}(\tau.\mathit{act})$, let act execute an annotation. There are two cases:

- Consider $\mathit{com}(\mathit{act})$ is of the form `@inv angel` r. By definition, $\mathit{inv}(\tau.\mathit{act}) = \exists r.\ F_\tau$ and $\mathit{inv}(\sigma.\mathit{act}) = \exists r.\ F_\sigma$. By induction, we get $\mathit{inv}(\sigma.\mathit{act}) \implies \mathit{inv}(\tau.\mathit{act})$.
- Consider $\mathit{com}(\mathit{act}) \not\equiv$ `@inv angel` r. By definition then, $\mathit{inv}(\tau.\mathit{act}) = \mathit{inv}(\tau) \wedge F_\tau$ and $\mathit{inv}(\sigma.\mathit{act}) = \mathit{inv}(\sigma) \wedge F_\sigma$. If $\mathit{com}(\mathit{act}) \notin \{\texttt{@inv active}(p), \texttt{@inv active}(r)\}$, we have $F_\tau \equiv F_\sigma$ by Property (P2). Otherwise, we have $F_\sigma \implies F_\tau$ because Properties (P4) and (P5) gives $\mathit{active}(\sigma) \subseteq \mathit{active}(\tau)$. By induction, we get $\mathit{inv}(\sigma.\mathit{act}) \implies \mathit{inv}(\tau.\mathit{act})$.

The case distinction is complete and thus concludes the claim.

If $\mathit{com}(\mathit{act}) \equiv \texttt{free}(a)$, then σ already has the desired properties. This is the case because $\texttt{free}(a)$ does not affect the memory nor the control. The $\texttt{free}(a)$ may remove a from $\mathit{fresh}(\tau)$, thus maintaining $\mathit{fresh}(\tau.\mathit{act}) \subseteq \mathit{fresh}(\sigma)$. Similarly, we maintain $\mathit{retired}(\tau.\mathit{act}) \subseteq \mathit{retired}(\sigma)$. Last, we have $\mathit{freed}(\tau.\mathit{act}) = \mathit{freed}(\tau) \cup \{a\}$. It remains to show that $a \in \mathit{retired}(\sigma)$. This follows from Lemma A.40 together with $\mathit{retired}(\tau) \subseteq \mathit{retired}(\sigma)$ from induction. □

Proof of Theorem A.42. Follows from Lemmas A.33 and A.41. □

Proof of Theorem 4.1. Follows from Theorem A.42. □

B.2 Type System

Proof of Lemma A.52. Immediately follows from definition. □

Proof of Lemma A.53. We do an induction over the derivation depth of $\vdash \Gamma_1\ \{\mathit{stmt}\}\ \Gamma_2$.

IB: The derivation is due to one rule application. This means the derivation is not due to one of: (INFER), (SEQ), (CHOICE), or (LOOP). For the remaining, applicable rules we have $\mathit{stmt} \equiv \mathit{com}$ and $(\mathit{com}, \tau) \dashrightarrow_t (\texttt{skip}, \tau.\tau')$. Thus, $\mathit{flat}_t(\tau') = \mathit{stmt}$. We immediately get $\vdash \Gamma_1\ \{\mathit{flat}_t(\tau')\}\ \Gamma_2$. Moreover, $\vdash \Gamma_2\ \{\texttt{skip}\}\ \Gamma_2$ holds by definition. That is, we choose $\Gamma = \Gamma_2$ as it has the desired properties.

IH: If $\vdash \Gamma_1\ \{\mathit{stmt}\}\ \Gamma_2$ and $(\mathit{stmt}, \tau) \dashrightarrow_t (\mathit{stmt}', \tau.\tau')$ hold, then there is some Γ with $\vdash \Gamma_1\ \{\mathit{flat}_t(\tau')\}\ \Gamma$ and $\vdash \Gamma\ \{\mathit{stmt}'\}\ \Gamma_2$.

IS: Consider a composed derivation of $\vdash \Gamma_1\ \{\mathit{stmt}\}\ \Gamma_2$. Let $(\mathit{stmt}, \tau) \dashrightarrow_t (\mathit{stmt}', \tau.\tau')$. We do a case distinction on the first rule of the derivation.

Rule (SEQ), part 1. Consider $\mathit{stmt} \equiv \texttt{skip}; \mathit{stmt}_2$. Then, $\mathit{stmt}' = \mathit{stmt}_2$ and $\tau' = \epsilon$. Moreover, there exists some Γ such that $\vdash \Gamma_1\ \{\texttt{skip}\}\ \Gamma$ and $\vdash \Gamma\ \{\mathit{stmt}_2\}\ \Gamma_2$ due to the type rules. Note that $\mathit{flat}_t(\tau') = \texttt{skip}$. That is, Γ has the desired properties.

Rule (SEQ), part 2. Consider $\mathit{stmt} \equiv \mathit{stmt}_1; \mathit{stmt}_2$. Let $(\mathit{stmt}_1, \tau) \dashrightarrow_t (\mathit{stmt}_1', \tau.\tau'')$. By definition, $\mathit{stmt}' = \mathit{stmt}_1'; \mathit{stmt}_2$ and $\tau' = \tau''$. The type rules give some Γ' with $\vdash \Gamma_1\ \{\mathit{stmt}_1\}\ \Gamma'$ and

$\vdash \Gamma' \ \{stmt_2\} \ \Gamma_2$. By induction, there is Γ with $\vdash \Gamma_1 \ \{flat_t(\tau')\} \ \Gamma$ and $\vdash \Gamma \ \{stmt_1'\} \ \Gamma'$. The latter gives $\vdash \Gamma \ \{stmt_1'; stmt_2\} \ \Gamma_2$. That is, Γ has the desired properties.

Rule (CHOICE). We have $stmt \equiv stmt_1 \oplus stmt_2$. Then, $\tau' = \epsilon$ and $stmt' = stmt_i$ for some $i \in \{1, 2\}$. Due to the type rules, we have $\vdash \Gamma_1 \ \{stmt_i\} \ \Gamma_2$. Moreover, $flat_t(\tau') = \texttt{skip}$ gives $\vdash \Gamma_1 \ \{flat_t(\tau')\} \ \Gamma_1$. That is, $\Gamma = \Gamma_1$ is an adequate choice with the desired properties.

Rule (LOOP). We have $stmt \equiv stmt_1^*$. Then, $\tau' = \epsilon$ and $stmt' = stmt_i$ for some $i \in \{1, 2\}$. Due to the type rules, we have $\vdash \Gamma_1 \ \{stmt_i\} \ \Gamma_2$. Moreover, $flat_t(\tau') = \texttt{skip}$ gives $\vdash \Gamma_1 \ \{flat_t(\tau')\} \ \Gamma_1$. That is, $\Gamma = \Gamma_1$ is an adequate choice with the desired properties.

Rule (INFER). There are type environments Γ_3 and Γ_4 such that $\Gamma_1 \rightsquigarrow \Gamma_3$, $\vdash \Gamma_3 \ \{stmt\} \ \Gamma_4$, and $\Gamma_4 \rightsquigarrow \Gamma_2$. By induction, there is Γ with $\vdash \Gamma_3 \ \{flat_t(\tau')\} \ \Gamma$ and $\vdash \Gamma \ \{stmt'\} \ \Gamma_4$. Applying Rule (INFER) we get $\vdash \Gamma_1 \ \{flat_t(\tau')\} \ \Gamma$ and $\vdash \Gamma \ \{stmt'\} \ \Gamma_2$ as desired.

The above induction concludes the claim. □

PROOF OF LEMMA A.54. Let $\vdash \Gamma_{init} \ \{P\} \ \Gamma$. We proceed by induction over the SOS transitions.

IB: We have $(pc_{init}, \epsilon) \dashrightarrow^0 (pc, \tau)$. That is, $pc = pc_{init}$ and $\tau = \epsilon$. Consider some thread t. We have $flat_t(\tau) = \texttt{skip}$ and $pc(t) = P^{[t]}$. The former gives $\vdash \Gamma_{init}^{[t]} \ \{flat_t(\tau)\} \ \Gamma_{init}^{[t]}$. The latter gives $\vdash \Gamma_{init}^{[t]} \ \{pc(t)\} \ \Gamma$ due to the premise. So we can choose $\Gamma_1 = \Gamma_{init}^{[t]}$ and $\Gamma_2 = \Gamma$.

IH: Let the claim hold for sequences of up to n steps.

IS: Consider now $(pc_{init}, \epsilon) \dashrightarrow^n (pc, \tau) \dashrightarrow_{t'} (pc', \tau.\tau')$. Let $t \neq \bot$ be some arbitrary thread. By induction, there are Γ_1', Γ_2 with

$$\vdash \Gamma_{init}^{[t]} \ \{flat_t(\tau)\} \ \Gamma_1' \qquad \text{and} \qquad \vdash \Gamma_1' \ \{pc(t)\} \ \Gamma_2 \ .$$

First, assume $t \neq t'$. Then, $flat_t(\tau.act) = flat_t(\tau)$ and $pc'(t) = pc(t)$. Thus, the claim follows immediately by induction. So consider $t = t'$ now. We have $(pc(t), \tau) \dashrightarrow_t (pc'(t), \tau.\tau')$ by definition. Lemma A.53 yields Γ_1 with

$$\vdash \Gamma_1' \ \{flat_t(\tau')\} \ \Gamma_1 \qquad \text{and} \qquad \vdash \Gamma_1 \ \{pc'(t)\} \ \Gamma_2 \ .$$

Altogether, we get the desired:

$$\vdash \Gamma_{init}^{[t]} \ \{flat_t(\tau.act)\} \ \Gamma_1 \qquad \text{and} \qquad \vdash \Gamma_1 \ \{pc'(t)\} \ \Gamma_2 \ .$$

The above induction concludes the claim. □

PROOF OF LEMMA A.55. Let $\tau.act \in [\![P]\!]_{Adr}^{\varnothing}$ and t, t' threads with $t \neq t' = thrd(act)$. Consider some $\vdash \Gamma_{init}^{[t]} \ \{flat_t(\tau)\} \ \Gamma$ and $x \in PVar \cup AVar$. We have $lock(\tau) = \{t'\}$ or $lock(\tau.act) = \{t'\}$. Hence, t has not yet contributed any actions to τ or it has finished an atomic section. We do a case distinction.

- Consider the case $flat_t(\tau) = \texttt{skip}$. By the type rules there is Γ' with:

$$\Gamma_{init}^{[t]} \rightsquigarrow \Gamma' \qquad \vdash \Gamma' \ \{\texttt{skip}\} \ \Gamma' \qquad \Gamma' \rightsquigarrow \Gamma$$

  By definition, $\Gamma_{init}^{[t]}(x) = \varnothing$. So $\neg isValid(\Gamma_{init}^{[t]}(x))$. Hence, $\neg isValid(\Gamma'(x))$ and $\neg isValid(\Gamma(x))$ follow from the definition of type inference. So we conclude $\Gamma(x) \cap \{\mathbb{A}, \mathbb{L}, \mathbb{S}\} = \varnothing$.
- Consider the case $flat_t(\tau.act) = stmt; \texttt{endAtomic}$ for some statement $stmt$. By the typing rules there are $\Gamma_1, \Gamma_2, \Gamma_3$ with:

$$\vdash \Gamma_{init}^{[t]} \ \{stmt\} \ \Gamma_1 \qquad \Gamma_1 \rightsquigarrow \Gamma_2 \qquad \vdash \Gamma_2 \ \{\texttt{endAtomic}\} \ \Gamma_3 \qquad \Gamma_3 \rightsquigarrow \Gamma$$

  where the derivation $\vdash \Gamma_2 \ \{\texttt{endAtomic}\} \ \Gamma_3$ is due to Rule (END). This means, $\Gamma_3 = rm(\Gamma_2)$. By definition, $\mathbb{A} \notin \Gamma_3(x)$. Hence, type inference provides $\mathbb{A} \notin \Gamma(p)$ as desired.

Now, consider the case $p \notin \mathit{local}_t$. There are two cases. First, assume $p \in \mathit{shared}$. $\Gamma_3(p) = \varnothing$ by definition. Second, assume $p \notin \mathit{shared}$. That is, x is local to another thread $t'' \neq t$: $p \in \mathit{local}_{t''}$. Then, the claim follows because the initial type binding does not contain local pointers of other threads and the type rules never add type bindings.

The above case distinction is complete and concludes the claim thus. □

Proof of Lemma A.56. Let $\tau.\mathit{act} \in [\![P]\!]^{\varnothing}_{\mathit{Adr}}$ and t, t' threads with $t \neq t' = \mathit{thrd}(\mathit{act}) \neq \bot$. Let $p \in \mathit{PVar} \cap \mathit{local}_t$. By definition, $\mathit{local}_t \cap \mathit{local}_{t'} = \varnothing$. Due to the semantics, p does not occur in $\mathit{com}(\mathit{act})$. Hence, $p \in \mathit{valid}_\tau \iff p \in \mathit{valid}_{\tau.\mathit{act}}$. Moreover, $\mathfrak{m}_\tau(p) = \mathfrak{m}_{\tau.\mathit{act}}(p)$. So every valid alias created by act requires a valid alias in τ. This is not possible since $\mathit{noalias}_\tau(p)$. □

Proof of Lemma A.57. Let $\tau.\mathit{act} \in [\![P]\!]^{\varnothing}_{\mathit{Adr}}$ PRF with $\mathit{act} = (t, \texttt{@inv}\ p = q, \varnothing)$ and $\mathit{inv}(\tau.\mathit{act})$. The latter gives $\mathfrak{m}_\tau(p) = \mathfrak{m}_\tau(q)$. Towards a contradiction, assume the claim does not hold. Wlog. $p \notin \mathit{valid}_\tau$ and $q \in \mathit{valid}_\tau$. Lemma A.29 gives $\mathfrak{m}_\tau(p) \neq \mathfrak{m}_\tau(q)$. This contradicts the premise and thus concludes the claim. □

Proof of Lemma A.58. Let $\tau.\mathit{act} \in [\![P]\!]^{\varnothing}_{\mathit{Adr}}$ PRF with $\mathit{act} = (t, \texttt{@inv active}(p), \mathit{up})$ and $\mathit{inv}(\tau.\mathit{act})$. By definition, this means $\mathfrak{m}_\tau(p) \in \mathit{active}(\tau)$. That is, $\mathfrak{m}_\tau(p) \notin \mathit{freed}(\tau)$. By the contrapositive of Lemma A.37: $p \in \mathit{valid}_\tau$. So $p \in \mathit{valid}_{\tau.\mathit{act}}$ by definition. Moreover, we get $\mathfrak{m}_{\tau.\mathit{act}}(p) \in \mathit{active}(\tau.\mathit{act})$. Hence, the remaining property follows from Lemma A.39. □

Proof of Lemma A.59. Let $\tau.\mathit{act} \in [\![P]\!]^{\varnothing}_{\mathit{Adr}}$ PRF with $\mathit{act} = (t, \texttt{@inv active}(r), \mathit{up})$ and $\mathit{inv}(\tau.\mathit{act})$. By definition, we have $\mathit{repr}_\tau(r) \subseteq \mathit{active}(\tau)$. Hence, $\mathit{repr}_\tau(r) \cap \mathit{freed}(\tau.\mathit{act}) = \varnothing$. Moreover, we have $\mathit{repr}_{\tau.\mathit{act}}(r) \subseteq \mathit{active}(\tau.\mathit{act})$ by definition. Let $a \in \mathit{repr}_{\tau.\mathit{act}}(r)$. This means $a \in \mathit{active}(\tau.\mathit{act})$. Then, the remaining property follows from Lemma A.39. □

Proof of Lemma A.60. Let $\tau \in O[\![P]\!]^{\varnothing}_{\mathit{Adr}}$. Let Γ, Γ' be two type environments with $\Gamma \rightsquigarrow \Gamma'$. Let $p \in \mathit{PVar}$ be a pointer with $a = \mathfrak{m}_\tau(p)$. Let $r \in \mathit{AVar}$ a ghost variable and $b \in \mathit{repr}_\tau(r)$. Consider the possible assumptions:

$$\mathit{isValid}(\Gamma(p)) \implies p \in \mathit{valid}_\tau \tag{1}$$
$$\mathit{isValid}(\Gamma(r)) \implies b \notin \mathit{freed}(\tau) \tag{2}$$
$$\mathbb{L} \in \Gamma(p) \implies \mathit{noalias}_\tau(p) \tag{3}$$
$$\mathit{reach}_{O,t,a}(\mathcal{H}(\tau)) \subseteq \mathit{Loc}(\Gamma(p)) \tag{4}$$
$$\mathit{reach}_{O,t,b}(\mathcal{H}(\tau)) \subseteq \mathit{Loc}(\Gamma(r)) \tag{5}$$

The definition of $\Gamma \rightsquigarrow \Gamma'$ gives:

$$\mathit{isValid}(\Gamma'(p)) \implies \mathit{isValid}(\Gamma(p)) \tag{6}$$
$$\mathit{isValid}(\Gamma'(r)) \implies \mathit{isValid}(\Gamma(r)) \tag{7}$$
$$\mathbb{L} \in \Gamma'(p) \implies \mathbb{L} \in \Gamma(p) \tag{8}$$
$$\mathit{Loc}(\Gamma(p)) \subseteq \mathit{Loc}(\Gamma'(p)) \tag{9}$$
$$\mathit{Loc}(\Gamma(r)) \subseteq \mathit{Loc}(\Gamma'(r)) \tag{10}$$

Then, we combine

- If (1) holds, then (6) gives $\mathit{isValid}(\Gamma'(p)) \implies p \in \mathit{valid}_\tau$,
- If (2) holds, then (7) gives $\mathit{isValid}(\Gamma'(r)) \implies b \notin \mathit{freed}(\tau)$,
- If (3) holds, then (8) gives $\mathbb{L} \in \Gamma'(p) \implies \mathit{noalias}_\tau(p)$, and
- If (4) holds, then (9) gives $\mathit{reach}_{O,t,a}(\mathcal{H}(\tau)) \subseteq \mathit{Loc}(\Gamma'(p))$.
- If (5) holds, then (10) gives $\mathit{reach}_{O,t,b}(\mathcal{H}(\tau)) \subseteq \mathit{Loc}(\Gamma'(r))$.

This concludes the claim. □

Proof of Lemma A.61. Let $\tau \in O[\![P]\!]^{\varnothing}_{Adr}$. We show:

$$freed(\tau) \cap retired(\tau) = \varnothing$$

and $\quad \forall t\ \forall \Gamma.\quad \vdash \Gamma^{[t]}_{init}\ \{flat_t(\tau)\}\ \Gamma$

$$\implies \forall p, r.\ \begin{pmatrix} reach_{O,t,\mathfrak{m}_\tau(p)}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma(p)) \\ \wedge\ isValid(\Gamma(p)) \implies p \in valid_\tau \\ \wedge\ \mathbb{L} \in \Gamma(p) \implies noalias_\tau(p) \\ \wedge\ \forall a \in repr_\tau(r).\ reach_{O,t,a}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma(r)) \\ \wedge\ isValid(\Gamma(r)) \implies repr_\tau(r) \cap freed(\tau) = \varnothing \end{pmatrix}$$

We proceed by induction over the structure of τ.

IB: Let $\tau = \epsilon$. Let t be some thread and let Γ be some type environment such that $\vdash \Gamma^{[t]}_{init}\ \{flat_t(\tau)\}\ \Gamma$. Note that $flat_t(\tau) = \texttt{skip}$. By definition, $freed(\tau) \cap retired(\tau) = \varnothing$. Consider some thread t, some $p \in PVar$ and some $r \in AVar$. By definition, $p \in valid_\tau$. This gives the desired implication $isValid(\Gamma(p)) \implies p \in valid_\tau$. By the type rules we have:

$$\Gamma^{[t]}_{init}, \epsilon \rightsquigarrow \Gamma_1 \qquad \vdash \Gamma_1\ \{\texttt{skip}\}\ \Gamma_1 \qquad \Gamma_1, \epsilon \rightsquigarrow \Gamma$$

Since $\mathbb{L} \notin \Gamma^{[t]}_{init}(p)$, we get $\mathbb{L} \notin \Gamma(p)$ by the definition of type inference. So we satisfy the implication $\mathbb{L} \in \Gamma(p) \implies noalias_\tau(p)$. Moreover, $freed(\tau) = \varnothing$. So $isValid(\Gamma(r)) \implies a \notin freed(\tau)$ is satisfied for every $a \in Adr$ as well. By Lemma A.52 we have $\Gamma^{[t]}_{init} \rightsquigarrow \Gamma$. That is, $Loc(\Gamma^{[t]}_{init}(p)) \subseteq Loc(\Gamma(p))$. By definition, $\Gamma^{[t]}_{init}(p) = \varnothing$. That is, $Loc(\Gamma^{[t]}_{init}(p)) = \top \times \top$. Consequently, $reach_{O,t,\mathfrak{m}_\tau(p)}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma^{[t]}_{init}(p))$. Similarly for r. Altogether, this concludes the base case.

IH: Let the claim hold for τ.

IS: Consider $\tau' = \tau.act$ with $act = (t', com, up)$, $\tau.act$ PRF, and $inv(\tau.act)$. Let t be some arbitrary thread we establish the claim for. We do a case distinction on t'.

Ad $t = t' \neq \bot$. We have

$$flat_t(\tau.act) = flat_t(\tau); com \qquad \text{and} \qquad freed(\tau.act) \subseteq freed(\tau)\ .$$

Assume that $\tau.act$ can be typed for t as nothing needs to be shown otherwise. That is, assume there are Γ_3 such that

$$\vdash \Gamma^{[t]}_{init}\ \{flat_t(\tau.act)\}\ \Gamma_3\ .$$

Due to the type rules and the above equality, we know that there are some Γ_1, Γ_2 such that:

$$\vdash \Gamma^{[t]}_{init}\ \{flat_t(\tau)\}\ \Gamma_0 \qquad \Gamma_0 \rightsquigarrow \Gamma_1 \qquad \vdash \Gamma_1\ \{com\}\ \Gamma_2 \qquad \Gamma_2 \rightsquigarrow \Gamma_3$$

where $\vdash \Gamma_1\ \{com\}\ \Gamma_2$ is derived by neither Rule (SEQ) nor Rule (CHOICE) nor Rule (LOOP) nor Rule (INFER). By induction, the claim holds for Γ_0. So by Lemma A.60 the claim also holds for Γ_1. If the claim holds for Γ_2, then the claim follow for Γ_3 from Lemma A.60 again. So it remains to show that the claim holds for Γ_2 relying on Γ_1. We do a case distinction over the type rules applied for the derivation $\vdash \Gamma_1\ \{com\}\ \Gamma_2$. To that end, let $p \in PVar$ be some arbitrary pointer variable and let $r \in AVar$ be some arbitrary angel. Let $\mathfrak{m}_\tau(p) = c_p$ and let $c_r \in repr_\tau(r)$. We show

(G1) $reach_{O,t,c_p}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_2(p))$
(G2) $reach_{O,t,c_r}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_2(r))$
(G3) $isValid(\Gamma_2(p)) \implies p \in valid_{\tau.act}$

(G4) $\mathbb{L} \in \Gamma_2(p) \implies noalias_{\tau.act}(p)$
(G5) $isValid(\Gamma_2(r)) \implies c_r \notin freed(\tau.act)$

Case Rule (BEGIN).

By definition, we have $\Gamma_2 = \Gamma_1$, $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$, $valid_\tau = valid_{\tau.act}$, $freed(\tau) = freed(\tau.act)$, $repr_\tau = repr_{\tau.act}$, and $\mathcal{H}(\tau) = \mathcal{H}(\tau.act)$. Hence, the claim follows by induction.

Case Rule (END).

By definition, we have $\Gamma_2 = rm(\Gamma_1)$. By definition, $\Gamma_2(p) \subseteq \Gamma_1(p)$ and $\Gamma_2(r) \subseteq \Gamma_1(r)$. This means we have $Loc(\Gamma_2(p)) \supseteq Loc(\Gamma_1(p))$ and $Loc(\Gamma_2(r)) \supseteq Loc(\Gamma_1(r))$. As in the previous case, we have $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$, $valid_\tau = valid_{\tau.act}$, $freed(\tau) = freed(\tau.act)$, $repr_\tau = repr_{\tau.act}$, and $\mathcal{H}(\tau) = \mathcal{H}(\tau.act)$. So the claim follows by induction.

Case Rule (ASSIGN1).

We have $freed(\tau) = freed(\tau.act)$ and $inv(\tau) \equiv inv(\tau.act)$. Hence, Property (G5) holds by induction. If $\mathbb{L} \in \Gamma_2(p)$, then p does not appear in com by definition of Rule (ASSIGN1). Hence, no alias of p is created by com and Property (G4) continues to hold by induction. If $isValid(\Gamma_2)$ then there are two cases. First, $com \equiv p := q$. Then, $\Gamma_2(p) = \Gamma_1(q) \setminus \{\mathbb{L}\}$. Hence, $q \in valid_\tau$ by induction. This leads to $p \in valid_{\tau.act}$ as desired. Second, p is not assigned to by com. Then, $\Gamma_2(p) \subseteq \Gamma_1(p)$. Hence, $p \in valid_\tau$ by induction and $p \in valid_{\tau.act}$ thus. This concludes Property (G3). Note that $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$. Property (G2) remains to hold by induction since r is not affected by com. It remains to establish Property (G1). If p does not occur in com, nothing needs to be show. So assume p occurs in com. In the first case, p occurs on the right-hand side of the assignment in com. Then, $\Gamma_2(p) = \Gamma_1(p) \setminus \{\mathbb{L}\}$. That is, $Loc(\Gamma_1(p)) \subseteq Loc(\Gamma_2(p))$. Then, Property (G1) follows by induction. Otherwise, p appears on the left-hand side of the assignment in com. So $com \equiv p := q$ for some q. Then, $\Gamma_2(p) = \Gamma_1(q) \setminus \{\mathbb{L}\}$. Moreover, $\mathfrak{m}_{\tau.act}(p) = \mathfrak{m}_\tau(q) = c_p$. By induction, we have $reach_{O,t,c_p}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_1(q))$. Hence, $reach_{O,t,c_p}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_1(q) \setminus \{\mathbb{L}\})$. We get the desired $reach_{O,t,c_p}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_2(q))$ by definition.

Case Rule (ASSIGN2).

Analogous to the previous case for (ASSIGN1).

Case Rule (ASSIGN3).

Analogous to the previous case for (ASSIGN1).

Case Rule (ASSIGN4),(ASSIGN5),(ASSIGN6),(ASSUME2).

We have $\Gamma_2 = \Gamma_1$, $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$, $repr_\tau = repr_{\tau.act}$, $valid_\tau = valid_{\tau.act}$, $freed(\tau) = freed(\tau.act)$, and $\mathcal{H}(\tau) = \mathcal{H}(\tau.act)$. Hence, the claim follows by induction.

Case Rule (ASSUME1).

We have $freed(\tau) = freed(\tau.act)$ and $inv(\tau) \equiv inv(\tau.act)$. Hence, Property (G5) holds by induction. If $\mathbb{L} \in \Gamma_2(p)$, then $\mathbb{L} \in \Gamma_1(p)$ due to the type rule. This means $noalias_\tau(p)$. Note that $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$. Moreover, since $\tau.act$ is PRF we know that the pointers in com are valid. Hence, $valid_\tau = valid_{\tau.act}$. So we get $noalias_{\tau.act}(p)$ by definition. This establishes Property (G4). If $isValid(\Gamma_2(p))$, then there are two cases. First, $isValid(\Gamma_1(p))$ holds. This means we have $p \in valid_\tau$ by induction. As stated above, this results in $p \in valid_{\tau.act}$. Second, $\neg isValid(\Gamma_1(p))$

holds. Then, p is validated by *com*. For this to happen, p must appear in *com*. Since $\tau.act$ is assumed to be PRF, we know $p \in valid_\tau$ must hold. So $p \in valid_{\tau.act}$ as before. This gives Property (G3). Note that we have $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$ and $repr_\tau = repr_{\tau.act}$. So It remains to establish Property (G2) continues to hold by induction since r is not affected. It remains to establish Property (G1). If p does not occur in *com* nothing needs to be show. So assume p appears in *com*. Wlog. *com* is of the form $com \equiv \texttt{assume}\ p = q$. Due to the type rule, we have $\Gamma_2(p) = \Gamma_1(p) \setminus \{\mathbb{L}\}$. By the semantics, we have $c_p = \mathfrak{m}_{\tau.act}(p) = \mathfrak{m}_\tau(p) = \mathfrak{m}_\tau(q) = \mathfrak{m}_{\tau.act}(q)$. So by induction, $reach_{O,t,c_p}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_1(p)) \cap Loc(\Gamma_1(p)) = Loc(\Gamma_1(p) \wedge \Gamma_2(p)) = Loc(\Gamma_2(p))$. This concludes Property (G1).

Case Rule (EQUAL).

If $isValid(\Gamma_2(p))$, then there are two cases. First, $isValid(\Gamma_1(p))$ holds. This means we have $p \in valid_\tau$ by induction. Then, $p \in valid_{\tau.act}$ because $valid_\tau = valid_{\tau.act}$ by definition. Second, $\neg isValid(\Gamma_1(p))$ holds. Then, p is validated by *com*. For this to happen, *com* must be of the form $com \equiv \texttt{@inv}\ p = q$ with $isValid(q)$. By induction, we have $q \in valid_\tau$. Then, Lemma A.57 gives $p \in valid_\tau$. Hence, $p \in valid_{\tau.act}$ as before. This concludes Property (G3). The remaining properties follow analogously to the previous case for (ASSUME1). For Property (G1) note that $repr_\tau \iff repr_{\tau.act}$ by definition together with $inv(\tau.act)$.

Case Rule (ACTIVE) for pointers.

If $isValid(\Gamma_2(p))$, then there are two cases. First, $isValid(\Gamma_1(p))$ holds. This means $p \in valid_\tau$ by induction. Then, $p \in valid_{\tau.act}$ because $valid_\tau = valid_{\tau.act}$. Second, $\neg isValid(\Gamma_1(p))$ holds. Then, p is validated by *com*. So *com* must be of the form $com \equiv \texttt{@inv}\ \texttt{active}(p)$. Since the invariants hold by assumption, $inv(\tau.act)$, we can invoke Lemma A.58. It gives $p \in valid_{\tau.act}$. Altogether, this concludes Property (G3). If $\mathbb{L} \in \Gamma_2(p)$, then $\mathbb{L} \in \Gamma_1(p)$ due to the type rule. Hence, the induction hypothesis together with $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$ and $valid_\tau = valid_{\tau.act}$ gives Property (G4). For Property (G5) note that $\Gamma_2(r) = \Gamma_1(r)$ and that $repr_\tau(r) = repr_{\tau.act}(r)$ because $inv(\tau.act)$ by assumption. So Property (G5) follows by induction. Note that we have $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$. Since r is not affected, Property (G2) follows by induction. We show Property (G1). If *com* does not contain p, nothing needs to be show. Otherwise, *com* is of the form $com \equiv \texttt{@inv}\ \texttt{active}(p)$. From Lemma A.58 we get $reach_{O,t,c_p}(\mathcal{H}(\tau.act)) \subseteq Loc(\mathbb{A})$. From induction, we get $reach_{O,t,c_p}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_1(p))$. By definition, this establishes the desired $reach_{O,t,c_p}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_1(p) \wedge \mathbb{A}) = Loc(\Gamma_2(p))$ and thus concludes Property (G1).

Case Rule (ACTIVE) for angels.

Using Lemma A.59, this case is analogous to the previous, pointer case.

Case Rule (MALLOC).

Recall that $\tau.act \in O[\![P]\!]^{\varnothing}_{Adr}$. So *act* allocates a fresh address. Hence, $freed(\tau) = freed(\tau.act)$ by definition. Moreover, $inv(\tau) \equiv inv(\tau.act)$. Then, Property (G5) holds by induction. Property (G2) remains to hold by induction since r is not affected. Consider Property (G1). If p does not appear in *com*, nothing needs to be shown. Otherwise, $com \equiv p := \texttt{malloc}$. From Lemma A.35 we get $c_p \notin retired(\tau)$. By definition then, $c_p \notin retired(\tau.act)$. Then, Lemma A.39 gives $reachtc_p\mathcal{H}(\tau.act) \in Loc(\mathbb{L})$. And by definition we have $\Gamma_2(p) = \{\mathbb{L}\}$. This concludes Property (G1).

For the remaining properties, we do a case distinction on q. First, consider the case $p \neq q$. The, $\Gamma_2(p) = \Gamma_1(p)$ and $p \in valid_{\tau.act} \iff p \in valid_\tau$. So Property (G3) follows by induction. Also

by induction, we have $noalias_\tau(p)$. Due to *up*, we have $\mathfrak{m}_{\tau.act}(p) = \mathfrak{m}_\tau(p) \neq \texttt{seg}$. Towards a contradiction, assume $\neq noalias_{\tau.act}(p)$. By definition, there is some pointer expression $pexp \in valid_{\tau.act} \setminus \{p\}$ with $\mathfrak{m}_{\tau.act}(p) = \mathfrak{m}_{\tau.act}(pexp)$. Since $\mathfrak{m}_{\tau.act}(q) \in fresh(\tau)$ we have $\mathfrak{m}_{\tau.act}(q) \notin range(\mathfrak{m}_\tau)$ due to Lemma A.34. Hence, $pexp \neq q$. Moreover, $pexp \not\equiv \mathfrak{m}_{\tau.act}(q).\texttt{next}$ because $\mathfrak{m}_{\tau.act}(\mathfrak{m}_{\tau.act}(q).\texttt{next}) = \texttt{seg}$. Consequently, $pexp$ is not affected by *act*. This means we have $\mathfrak{m}_{\tau.act}(pexp) = \mathfrak{m}_\tau(pexp)$ and $pexp \in valid_\tau \setminus \{p\}$. That is, $\neq noalias_\tau(p)$. Since this contradicts induction, we conclude the desired $noalias_{\tau.act}(p)$. This establishes Property (G4). Second, consider the case $p = q$. Then, $\Gamma_2(p) = \{\mathbb{L}\}$. By Lemmas A.30 and A.34 we have $a \notin \mathfrak{m}_\tau(valid_\tau)$. Hence, we get $a \notin \mathfrak{m}_{\tau.act}(valid_{\tau.act} \setminus \{p\})$. This means $noalias_{\tau.act}(p)$ holds by definition. This establishes Property (G4). And by definition $p \in valid_{\tau.act}$. So Property (G3) holds as well.

Case Rule (ENTER).

Note that we have $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$ and $repr_\tau = repr_{\tau.act}$ by definition. By induction, we have $reach_{O,t,c_p}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_1(p))$. Type inference gives $post_{p,com}(Loc(\Gamma_1(p))) \subseteq Loc(\Gamma_2(p))$. Since $\mathcal{H}(\tau.act) \in \mathcal{S}(O)$ due to the semantics, we get $\mathcal{H}(\tau.act) \in postpcomLoc(\Gamma_1(p))$. Hence, $postpcomLoc(\Gamma_2(p)) \subseteq Loc(\Gamma_2(p))$ as desired. This concludes Property (G1). Property (G2) follows along the same lines. If $isValid(\Gamma_2(p))$, then $isValid(\Gamma_1(p))$ by the definition of type inference. Moreover, $valid_\tau = valid_{\tau.act}$. Hence, Property (G3) follows by induction. Similarly, $isValid(\Gamma_2(r))$ implies $isValid(\Gamma_1(r))$. So Property (G5) follows by induction together with $freed(\tau) = freed(\tau.act)$ and $inv(\tau) \equiv inv(\tau.act)$. If $\mathbb{L} \in \Gamma_2(p)$, then $\mathbb{L} \in \Gamma_1(p)$ by definition. Note that $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$. Hence, $noalias_{\tau.act}(p)$ follows by induction. This establishes Property (G4).

Case Rule (EXIT).

Analogously to the previous case for (ENTER).

Case Rule (ANGEL).

By definition, $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$, $valid_\tau = valid_{\tau.act}$, and $freed(\tau) = freed(\tau.act)$. Moreover, the type rule gives $\Gamma_2(p) = \Gamma_1(p)$. So Properties (G3) and (G4) follow by induction. If $isValid(\Gamma_2(r))$, then r does not appear in *com*. So by the type rule we have $\Gamma_2(r) = \Gamma_1(r)$. Hence, Property (G5) follows by induction. Since p is not affected, we get Property (G1) from induction. It remains to consider Property (G2). If r does not appear in *com*, nothing needs to be show because $repr_{\tau.act}(r) = repr_\tau(r)$. Otherwise, we have $\Gamma_2(r) = \emptyset$. This concludes Property (G2).

Case Rule (MEMBER).

By definition, $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$, $valid_\tau = valid_{\tau.act}$, and $freed(\tau) = freed(\tau.act)$. If $isValid(\Gamma_2(r))$, then $isValid(\Gamma_1(r))$ holds since $\Gamma_2(r) = \Gamma_1(r)$. So Property (G5) follows by induction. If $\mathbb{L} \in \Gamma_2(p)$, then $\mathbb{L} \in \Gamma_1(p)$ since angels cannot acquire guarantee $\mathbb{L}$ due to the type rules. Hence, Property (G4) follows by induction. If $isValid(\Gamma_2(p))$, then there are two cases. First, $isValid(\Gamma_1(p))$ holds. Then, $p \in valid_\tau$ by induction and thus $p \in valid_{\tau.act}$. Second, $\neg isValid(\Gamma_1(p))$ holds. Then, p is validated by *com*. For this to happen, we must have $com \equiv \texttt{@inv}\ p\ \texttt{in}\ r'$ with $isValid(\Gamma_1(r'))$. Since $inv(\tau.act)$ hold, we get $\mathfrak{m}_\tau(p) \in repr_{\tau.act}(r')$. Moreover, $isValid(\Gamma_1(r'))$ gives $isValid(\Gamma_2(r'))$. So the already established Property (G5) yields $\mathfrak{m}_\tau(p) \notin freed(\tau.act)$. Hence, $\mathfrak{m}_{\tau.act}(p) \notin freed(\tau.act)$. Now, $p \in valid_{\tau.act}$ by the contrapositive of Lemma A.37. This establishes Property (G3). Consider now Property (G2). If r' does not appear in *com*, nothing needs to be shown. Otherwise, $com \equiv \texttt{@inv}\ q\ \texttt{in}\ r$. Due to assumption of $inv(\tau.act)$, we have $\mathfrak{m}_{\tau.act}(q) \in repr_{\tau.act}(r)$. By definition, $\mathfrak{m}_{\tau.act}(q) = \mathfrak{m}_\tau(q)$. Again by definition,

we get $\mathfrak{m}_\tau(q) \in \mathit{repr}_\tau(r)$ because $\mathit{repr}_\tau(r)$ is defined to be the maximal set. This means $\mathit{repr}_\tau(r) = \mathit{repr}_{\tau.\mathit{act}}(r)$. Hence, we can conclude Property (G2) by induction. It remains to show Property (G1). If p does not occur in com, nothing needs to be shown. Otherwise, $\mathit{com} \equiv \texttt{@inv}\ p\ \texttt{in}\ r'$. Similarly to the above, we get $c_p \in \mathit{repr}_\tau(r')$. So we get by induction:

$$\mathit{reach}_{O,t,c_p}(\mathcal{H}(\tau)) \subseteq \mathit{Loc}(\Gamma_1(p)) \cap \mathit{Loc}(\Gamma_1(r')) = \mathit{Loc}(\Gamma_1(p) \wedge \Gamma_1(r')) = \mathit{Loc}(\Gamma_2(p)) \ .$$

This concludes Property (G1) by induction together with $\mathcal{H}(\tau.\mathit{act}) = \mathcal{H}(\tau)$.

The above case distinction is complete and shows that the claim holds for t and Γ_2. Hence, the claim holds for Γ_3 as reasoned above.

Now, we show that $\mathit{freed}(\tau.\mathit{act}) \cap \mathit{retired}(\tau.\mathit{act}) = \varnothing$ holds. We have $\mathit{freed}(\tau.\mathit{act}) \subseteq \mathit{freed}(\tau)$ by the fact that act cannot be a free due to the fact that $t' \neq \bot$. If $\mathit{retired}(\tau.\mathit{act}) \subseteq \mathit{retired}(\tau)$ holds, then the claim follows by induction. Otherwise, we have $\mathit{retired}(\tau.\mathit{act}) = \mathit{retired}(\tau) \cup \{a\}$ with $\mathit{com} \equiv \texttt{enter retire}(p)$ and $\mathfrak{m}_\tau(p) = a$. Since $\vdash \Gamma_1\ \{\mathit{com}\}\ \Gamma_2$ holds, we know $p \in \mathit{valid}_\tau$. By the contrapositive of Lemma A.30, $a \notin \mathit{freed}(\tau)$. So by induction, $\mathit{freed}(\tau.\mathit{act}) \cap \mathit{retired}(\tau.\mathit{act}) = \varnothing$.

Ad $t \neq t' \neq \bot$. We have

$$\vdash \Gamma_{\mathit{init}}^{[t]}\ \{\mathit{flat}_t(\tau)\}\ \Gamma_4 \quad \text{and} \quad \mathit{flat}_t(\tau.\mathit{act}) = \mathit{flat}_t(\tau) \quad \text{and} \quad \vdash \Gamma_{\mathit{init}}^{[t]}\ \{\mathit{flat}_t(\tau.\mathit{act})\}\ \Gamma_4 \ .$$

The induction hypothesis applies to $\vdash \Gamma_{\mathit{init}}^{[t]}\ \{\mathit{flat}_t(\tau)\}\ \Gamma_4$. We have to show that the desired properties are stable under interference.

Consider some $p \in \mathit{dom}(\Gamma_4) \cap \mathit{PVar}$. If $\mathbb{L} \in \Gamma_4(p)$, then $p \in \mathit{local}_t$ by the contrapositive of Lemma A.55. By induction, we have $\mathit{noalias}_\tau(p)$. Then, Lemma A.56 gives $\mathit{noalias}_{\tau.\mathit{act}}(p)$. If $\mathit{isValid}(\Gamma_4(p))$, then $p \in \mathit{valid}_\tau$ by induction. We invoke Lemma A.56 and get $p \in \mathit{valid}_{\tau.\mathit{act}}$.

Consider some $r \in \mathit{dom}(\Gamma_4) \cap \mathit{AVar}$. If $\mathit{isValid}(\Gamma_4(r))$, then $r \in \mathit{local}_t$ by the contrapositive of Lemma A.55. By induction we get $\mathit{repr}_\tau(r) \cap \mathit{freed}(\tau) = \varnothing$. Due to r being local to a thread other than the one executing act, r cannot occur in com. Consequently, $\mathit{repr}_{\tau.\mathit{act}}(r) = \mathit{repr}_\tau(r)$. So $\mathit{freed}(\tau.\mathit{act}) \subseteq \mathit{freed}(\tau)$ due to $t' \neq \bot$ gives $\mathit{repr}_{\tau.\mathit{act}}(r) \cap \mathit{freed}(\tau.\mathit{act}) = \varnothing$ as desired.

It remains to establish $\mathcal{H}(\tau.\mathit{act}) \subseteq \mathit{Loc}(\Gamma_4)$. If $\mathcal{H}(\tau.\mathit{act}) = \mathcal{H}(\tau)$, then the claim follows by induction. So let $\mathcal{H}(\tau.\mathit{act}) = h.\mathit{evt}$ with $\mathcal{H}(\tau) = h$. We have $\mathit{evt} \downarrow_t = \epsilon$. By definition of closedness under interference, we have for all a and i:

$$\mathit{reach}_{O,t,a}(h) \subseteq \mathit{Loc}(\mathbb{S}) \implies \mathit{reach}_{O,t,a}(h.\mathit{evt}) \subseteq \mathit{Loc}(\mathbb{S})$$
$$\text{and} \qquad \mathit{reach}_{O,t,a}(h) \subseteq \mathit{Loc}(\mathbb{E}_i) \implies \mathit{reach}_{O,t,a}(h.\mathit{evt}) \subseteq \mathit{Loc}(\mathbb{E}_i) \ .$$

Consider some $x \in \mathit{dom}(\Gamma_4)$. By Lemma A.55 we know $\mathbb{A} \notin \Gamma_4(p)$. If $\mathbb{L} \in \Gamma_4(x)$, then we have $x \in \mathit{PVar}$ since the type rules do not allow angels to carry $\mathbb{L}$. Moreover, induction gives $\mathit{reach}_{O,t,\mathfrak{m}_\tau(x)}(h) \subseteq \mathit{Loc}(\mathbb{L})$. To the contrary, assume $\mathit{reach}_{O,t,\mathfrak{m}_\tau(x)}(h.\mathit{evt}) \nsubseteq \mathit{Loc}(\mathbb{L})$. By definition, this means that evt makes O_{Base} leave its initial location. Since $h.\mathit{evt} \in \mathcal{S}(O)$ due to the semantics, we must have $\mathit{evt} = \texttt{enter retire}(t', \mathfrak{m}_\tau(x))$. That is, $\mathit{com} \equiv \texttt{enter retire}(q)$ with $\mathfrak{m}_\tau(q) = \mathfrak{m}_\tau(x)$. Since $\tau.\mathit{act}$ is assumed to be PRF, we must have $q \in \mathit{valid}_\tau$. This results in $\neg \mathit{noalias}_\tau(x)$ and resembles a contradiction. So we conclude $\mathit{reach}_{O,t,\mathfrak{m}_\tau(x)}(h.\mathit{evt}) \subseteq \mathit{Loc}(\mathbb{L})$. By the contrapositive of Lemma A.55 we also know that $x \in \mathit{local}_t$. So $\mathfrak{m}_\tau(x) = \mathfrak{m}_{\tau.\mathit{act}}(x)$. That is, $\mathit{reach}_{O,t,\mathfrak{m}_{\tau.\mathit{act}}(x)}(h.\mathit{evt}) \nsubseteq \mathit{Loc}(\mathbb{L})$. Moreover, if $x \in \mathit{AVar}$, then $x \in \mathit{local}_t$ by Assumption 2. This means $\mathit{repr}_\tau(x) = \mathit{repr}_{\tau.\mathit{act}}(x)$ since x does not appear in com. So we get $\mathit{reach}_{O,t,a}(h.\mathit{evt}) \subseteq \mathit{Loc}(\Gamma_4(x))$ for all $a \in \mathit{repr}_{\tau.\mathit{act}}(x)$ by induction.

The remaining $\mathit{freed}(\tau.\mathit{act}) \cap \mathit{retired}(\tau.\mathit{act}) = \varnothing$ follows as in the previous case for $t = t' \neq \bot$. This concludes the case.

Ad $t' = \bot$. We have $\mathit{act} = (\bot, \texttt{free}(a), \varnothing)$. Consider some thread t and type environment Γ with $\vdash \Gamma_{\mathit{init}}^{[t]}\ \{\mathit{flat}_t(\tau)\}\ \Gamma$. By definition, $\mathit{flat}_t(\tau.\mathit{act}) = \mathit{flat}_t(\tau)$. So $\vdash \Gamma_{\mathit{init}}^{[t]}\ \{\mathit{flat}_t(\tau.\mathit{act})\}\ \Gamma$. We

show that Γ satisfies the claim. Let $\mathcal{H}(\tau) = h$. Then, $\mathcal{H}(\tau.act) = h.\texttt{free}(a)$. By the semantics, we have $h.\texttt{free}(a) \in \mathcal{S}(O)$.
Consider some $p \in dom(\Gamma) \cap PVar$. If $\mathbb{L} \in \Gamma(p)$, then $noalias_\tau(p)$. Since $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$ and $valid_{\tau.act} \subseteq valid_\tau$, we get $noalias_{\tau.act}(p)$. If $isValid(\Gamma(p))$, then $\{\mathbb{A}, \mathbb{L}, \mathbb{S}\} \cap \Gamma(p) \neq \emptyset$. By induction, we have $p \in valid_\tau$. Note that we have $reach_{O,t,\mathfrak{m}_\tau(p)}(h) \subseteq Loc(\Gamma(p))$ by induction. Hence, $a \neq \mathfrak{m}_\tau(p)$ must hold as for otherwise $h.\texttt{free}(a) \notin \mathcal{S}(O)$ by the definition of $\{\mathbb{A}, \mathbb{L}, \mathbb{S}\}$. So, we get $p \in valid_{\tau.act}$ by definition.
Consider some $r \in dom(\Gamma) \cap AVar$. If $isValid(\Gamma(r))$, then $repr_\tau(r) \cap freed(\tau) = \emptyset$ by induction. By definition, we have $repr_\tau(r) = repr_{\tau.act}(r)$. Moreover, $freed(\tau.act) = freed(\tau) \cup \{a\}$. So in order to arrive at $repr_{\tau.act}(r) \cap freed(\tau.act) = \emptyset$, it suffices to establish $a \notin repr_\tau(r)$. To the contrary, assume $a \in repr_\tau(r)$. Then, $reach_{O,t,a}(h) \subseteq Loc(\Gamma(r))$ by induction. However, similar to the pointer case above, this means $h.\texttt{free}(a) \notin \mathcal{S}(O)$ because of $isValid(\Gamma(r))$. Hence, $a \notin repr_\tau(r)$ must hold as desired.
It remains to show that $reach_{O,t,b}(h.\texttt{free}(a)) \subseteq Loc(\Gamma(x))$ for every $x \in dom(\Gamma_4)$ with $a \neq \mathfrak{m}_\tau(x)$ for $x \in PVar$ and $a \notin repr_\tau(x)$ for $x \in AVar$. By definition we have:

$$
\begin{aligned}
reach_{O,t,b}(h) \subseteq Loc(\mathbb{E}_i) &\implies reach_{O,t,b}(h.\texttt{free}(a)) \subseteq Loc(\mathbb{E}_i) \\
reach_{O,t,b}(h) \subseteq Loc(\mathbb{S}) &\implies reach_{O,t,b}(h.\texttt{free}(a)) \subseteq Loc(\mathbb{S}) \\
reach_{O,t,b}(h) \subseteq Loc(\mathbb{A}) &\implies reach_{O,t,b}(h.\texttt{free}(a)) \subseteq Loc(\mathbb{A}) && \text{if } a \neq b \\
reach_{O,t,b}(h) \subseteq Loc(\mathbb{L}) &\implies reach_{O,t,b}(h.\texttt{free}(a)) \subseteq Loc(\mathbb{L}) && \text{if } a \neq b
\end{aligned}
$$

Recall from above that $\{\mathbb{A}, \mathbb{L}\} \cap \Gamma(x) \neq \emptyset$ implies $a \neq \mathfrak{m}_\tau(x)$ for $x \in PVar$ and $a \notin repr_\tau(x)$ for $x \in AVar$. Hence, we conclude the desired $reach_{O,t,b}(h.\texttt{free}(a)) \subseteq Loc(\Gamma(x))$ by induction together with $\mathfrak{m}_\tau = \mathfrak{m}_{\tau.act}$ and $repr_\tau = repr_{\tau.act}$.
Lastly, we show $freed(\tau.act) \cap retired(\tau.act) = \emptyset$. We have $freed(\tau.act) = freed(\tau) \cup \{a\}$ and $retired(\tau.act) = retired(\tau) \setminus \{a\}$. Then the claim follows by induction.

□

Proof of Lemma A.62. Let $\vdash \Gamma_{init}\ \{P\}\ \Gamma_P$ and $inv([\![P]\!]^\emptyset_\emptyset)$. Towards a contradiction, assume the claim does not hold. That is, there is a shortest computation $\tau.act \in O[\![P]\!]^\emptyset_{Adr}$ such that $\tau.act$ raises a pointer race or $\neg inv(\tau.act)$. By minimality, τ is PRF and $inv(\tau)$. Let $act = (t, com, up)$. (Note that com is neither `beginAtomic` nor `endAtomic` due to the assumption.) By Lemma A.54, there is Γ_3 such that $\vdash \Gamma^{[t]}_{init}\ \{flat_t(\tau.act)\}\ \Gamma_3$. We have $flat_t(\tau.act) = flat_t(\tau); com$ by definition. So by the type rules there is $\Gamma_0, \Gamma_1, \Gamma_2$

$$\vdash \Gamma^{[t]}_{init}\ \{flat_t(\tau)\}\ \Gamma_0 \qquad \Gamma_0 \rightsquigarrow \Gamma_1 \qquad \vdash \Gamma_1\ \{com\}\ \Gamma_2 \qquad \Gamma_2 \rightsquigarrow \Gamma_3$$

First, consider the case where $\tau.act$ raises a pointer race. By definition of pointer races, act is on of: an unsafe access, an unsafe assumption, an unsafe `enter`, or an unsafe retire.

- If act is an unsafe access, then com contains $p.\texttt{next}$ or $p.\texttt{data}$ with $p \notin valid_\tau$. That is, $\vdash \Gamma_1\ \{com\}\ \Gamma_2$ is derived using on of the following rules: (ASSIGN2), (ASSIGN3), (ASSIGN5), or (ASSIGN6). Since the derivation is defined, we must have $\Gamma_1(p) = T$ with $isValid(T)$. By Lemma A.61, this means $p \in valid_\tau$. Since this contradicts the assumption of act raising a pointer race, this case cannot apply.
- If act is an unsafe assumption, then $com \equiv \texttt{assume}\ p = q$ with $\{p, q\} \not\subseteq valid_\tau$. That is, $\vdash \Gamma_1\ \{com\}\ \Gamma_2$ is derived using rule (ASSUME1). By definition, we have $\Gamma_1(p) = T, \Gamma_1(q) = T'$ with $isValid(T)$ and $isValid(T')$. From Lemma A.61 we get $\{p, q\} \subseteq valid_\tau$. Since this contradicts the assumption of act raising a pointer race, this case cannot apply.

- Consider an unsafe `enter`, i.e., $com \equiv \texttt{enter}\ func(\bar{p}, \bar{u})$. That is, $\vdash \Gamma_1\ \{com\}\ \Gamma_2$ is derived using rule (ENTER). Let $\mathfrak{m}_\tau(\bar{p}) = \bar{a}$ and $\mathfrak{m}_\tau(\bar{u}) = \bar{d}$. That act is an unsafe enter means that there are $\bar{b}$ and c with:
$$\forall i.\ (a_i = c \vee p_i \in valid_\tau) \implies a_i = b_i$$
$$\text{and} \qquad \mathcal{F}_O(h.func(t, \bar{b}, \bar{d}), c) \not\subseteq \mathcal{F}_O(h.func(t, \bar{a}, \bar{d}), c)\ .$$
Let $\bar{p} = p_1, \ldots, p_n$ with $\Gamma(p_i) = T_i$. From Lemma A.61 we get $p_i \notin valid_\tau \implies \neg isValid(T_i)$. Hence, the following holds:
$$\begin{aligned} & \forall i.\ (a_i = c \vee p_i \in valid_\tau) \implies a_i = b_i \\ \text{implies} \quad & \forall i.\ a_i \neq b_i \implies (a_i \neq c \wedge p_i \notin valid_\tau) \\ \text{implies} \quad & \forall i.\ a_i \neq b_i \implies (a_i \neq c \wedge \neg isValid(T_i)) \\ \text{implies} \quad & \forall i.\ (a_i = c \vee isValid(T_i)) \implies a_i = b_i \end{aligned}$$
So $safeEnter(\Gamma_1, func(\bar{p}, \bar{u})) = false$. As this contradicts $\vdash \Gamma_1\ \{com\}\ \Gamma_2$, this case cannot apply.
- Consider an unsafe retire, i.e., $com \equiv \texttt{enter retire}(p)$ with $p \notin valid_\tau$. As in the previous case, $\vdash \Gamma_1\ \{com\}\ \Gamma_2$ is derived using rule (ENTER). It gives $\mathbb{A} \in \Gamma_1(p)$. That is, $isValid(\Gamma_1(p)) = true$. Then, Lemma A.61 yields $p \in valid_\tau$. Hence, this case cannot apply.

The above case distinction is complete. That is, $\tau.act$ cannot raise a pointer race. Hence, we must have $\neg inv(\tau.act)$. Since we have $inv(\tau)$ as stated before, act must be an annotation that does not hold. By Lemma A.41 for $\tau.act$ there is $\sigma \in [\![P]\!]^{\varnothing}_{\varnothing}$ with $ctrl(\tau.act) = ctrl(\sigma)$, $\mathfrak{m}_{\tau.act} = \mathfrak{m}_\sigma$, and $inv(\sigma) \implies inv(\tau.act)$. The contrapositive of the latter, gives $\neg inv(\tau.act) \implies \neg inv(\sigma)$. That is, we must have $\neg inv(\sigma)$. This contradicts the assumption of $inv([\![P]\!]^{\varnothing}_{\varnothing})$, concluding the claim thus. □

Proof of Lemma A.63. Let O_{SMR} supports elision. Furthermore, let $\vdash P$ and $inv([\![P]\!]^{\varnothing}_{\varnothing})$. By Lemma A.62 we know that $O[\![P]\!]^{\varnothing}_{Adr}$ is PRF and that $inv(O[\![P]\!]^{\varnothing}_{Adr})$ holds. Now, to the contrary, assume the overall claim does not hold. Then there is $\tau.act \in [\![P]\!]^{Adr}_{Adr}$ with $a \in retired(\tau)$, $act = (t, com, up)$, $com \equiv \texttt{enter retire}(p)$, and $\mathfrak{m}_\tau(p) = a$. Then, Lemma A.33 yields $\sigma \in O[\![P]\!]^{\varnothing}_{Adr}$ such that $\tau \sim \sigma$ and $retired(\tau) \subseteq retired(\sigma)$. From the former we get $\sigma.act \in O[\![P]\!]^{\varnothing}_{Adr}$. Moreover, together with $O[\![P]\!]^{\varnothing}_{Adr}$ PRF and thus $p \in valid_\sigma$, we get $\mathfrak{m}_\tau(p) = a = \mathfrak{m}_\sigma(p)$. The latter gives $a \in retired(\sigma)$.

From Lemma A.54 we get some Γ_3 with $\vdash \Gamma^{[t]}_{init}\ \{flat_t(\sigma.act)\}\ \Gamma_3$. Then, $flat_t(\sigma.act) = flat_t(\sigma); com$ by definition. So the typing rules give some $\Gamma_0, \Gamma_1, \Gamma_2$ with
$$\vdash \Gamma^{[t]}_{init}\ \{flat_t(\sigma)\}\ \Gamma_0 \qquad \Gamma_0 \rightsquigarrow \Gamma_1 \qquad \vdash \Gamma_1\ \{com\}\ \Gamma_2 \qquad \Gamma_2 \rightsquigarrow \Gamma_3$$
where the derivation for $\vdash \Gamma_1\ \{com\}\ \Gamma_2$ is due to Rule (ENTER). By definition, this means we have $\mathbb{A} \in \Gamma_1(p)$. Note that σ is PRF and $inv(\sigma)$. Moreover, $\vdash \Gamma^{[t]}_{init}\ \{flat_t(\sigma)\}\ \Gamma_1$ holds due to Rule (INFER). Now, Lemma A.61 yields $reach_{O,t,a}(\mathcal{H}(\sigma)) \subseteq Loc(\Gamma_1(p))$. In particular, this means $reach_{O,t,a}(\mathcal{H}(\sigma)) \subseteq Loc(\mathbb{A})$. Hence, $(L_2, \varphi) \xrightarrow{h} (L_2, \varphi)$ with $\varphi = \{z_a \mapsto a\}$ and $h = \mathcal{H}(\sigma)$. Then, Lemma A.39 gives $a \notin retired(\sigma)$. This contradicts the previous $a \in retired(\sigma)$. □

Proof of Theorem 5.1. Follows from Lemmas A.62 and A.63. □

C TYPE CHECKING

Lemma C.1. *$AntiChainTypes := (Types/_{\rightsquigarrow \cap \rightsquigarrow^{-1}}, \rightsquigarrow)$ is a complete lattice.*

Proof. The least element is the most precise type containing every guarantee. The join of two guarantees $\mathbb{E}_L \sqcup \mathbb{E}_{L'}$ is characterized by the union $L \cup L'$, which is again closed under interferences. The meet of two guarantees $\mathbb{E}_L \sqcap \mathbb{E}_{L'}$ requires care. The intersection $L \cap L'$ may not be closed under interferences, in which case a corresponding guarantee $\mathbb{E}_{L \cap L'}$ does not exist. Instead, we

take the largest set of locations that is closed under interferences and lives inside the intersection. It is guaranteed to exist. □

Lemma C.2. *$sp(\cdot, com)$ is monotonic.*

Proof. The case that requires care is Rule (ENTER). A larger enriched environment has less guarantees, and eventually $safeEnter(\Gamma, func(\bar{p}, \bar{u}))$ may fail. In this case, however, we return $\top$. □

Proof of Proposition 6.2. For $lsol(X) \sqsupseteq \bigsqcap_{\vdash \Gamma_{init}\ \{stmt\}\ \Gamma} \Gamma$, observe that the least solution to the constraint system yields a type derivation $\vdash \Gamma_{init}\ \{stmt\}\ lsol(X)$. Indeed, as the solution assigns a type environment to every control point, it already gives the intermediary environments we should assume for a Rule (SEQ). For the reverse direction, consider $\vdash \Gamma_{init}\ \{stmt\}\ \Gamma$. One can show that by assigning the environments encountered in the type derivation to the variables of the corresponding control points, we obtain a solution to the constraint system. Since $lsol(X)$ is the least solution, $lsol(X) \sqsubseteq \Gamma$. As the reasoning holds for every derivation, $lsol(X)$ will lower bound the meet of the environments. □