Pointer Race Freedom ⋆

Frédéric Haziza[1], Lukáš Holík[2], Roland Meyer[3], and Sebastian Wolff[3]

[1]Uppsala University [2]Brno University of Technology
[3]University of Kaiserslautern

Abstract We propose a novel notion of pointer race for concurrent programs manipulating a shared heap. A pointer race is an access to a memory address which was freed, and it is out of the accessor's control whether or not the cell has been re-allocated. We establish two results. (1) Under the assumption of pointer race freedom, it is sound to verify a program running under explicit memory management as if it was running with garbage collection. (2) Even the requirement of pointer race freedom itself can be verified under the garbage-collected semantics. We then prove analogues of the theorems for a stronger notion of pointer race needed to cope with performance-critical code purposely using racy comparisons and even racy dereferences of pointers. As a practical contribution, we apply our results to optimize a thread-modular analysis under explicit memory management. Our experiments confirm a speed-up of up to two orders of magnitude.

1 Introduction

Today, one of the main challenges in verification is the analysis of concurrent programs that manipulate a shared heap. The numerous interleavings among the threads make it hard to predict the dynamic evolution of the heap. This is even more true if explicit memory management has to be taken into account. With garbage collection as in Java, an allocation request results in a fresh address that was not being pointed to. The address is hence known to be owned by the allocating thread. With explicit memory management as in C, this ownership guarantee does not hold. An address may be re-allocated as soon as it has been freed, even if there are still pointers to it. This missing ownership significantly complicates reasoning against the memory-managed semantics.

In the present paper[1], we carefully investigate the relationship between the memory-managed semantics and the garbage-collected semantics. We show that the difference only becomes apparent if there are programming errors of a particular form that we refer to as pointer races. A pointer race is a situation where a thread uses a pointer that has been freed before. We establish two theorems. First, if the memory-managed semantics is free from pointer races, then it coincides with the garbage-collected semantics. Second, whether or not the memory-managed semantics contains a pointer race can be checked with the garbage-collected semantics.

The developed semantic understanding helps to optimize program analyses. We show that the more complicated verification of the memory-managed semantics can

⋆ This work was supported by the Czech Science Foundation, project 13-37876P, and by the German Science Foundation (DFG), project R2M2.

[1] The full version is available as technical report [9].

often be reduced to an analysis of the simpler garbage-collected semantics — by applying the following policy: check under garbage collection whether the program is pointer race free. If there are pointer races, tell the programmer about these potential bugs. If there are no pointer races, rely on the garbage-collected semantics in all further analyses. In thread-modular reasoning, one of the motivations for our work, restricting to the garbage-collected semantics allows us to use a smaller abstract domain and an optimized fixed point computation. Particularly, it removes the need to correlate the local states of threads, and it restricts the possibilities of how threads can influence one another.

Example 1. We illustrate the idea of pointer race freedom on Treiber's stack [14], a lock-free implementation of a concurrent stack that provides the following methods:

```
// global variables: pTop
void : push(v)
  pnode := malloc;
  pnode.data := v;
  repeat
    ptop := pTop;
    pnode.next := ptop;
  until cas(pTop, ptop, pnode);

bool : pop(&v)
  repeat
    ptop := pTop;
    if (ptop = null) return false;
    pnode := ptop.next;
  until cas(pTop, ptop, pnode);
  v := ptop.data;
  ptop := free; return true;
```

This code is correct (i.e. linearizable and pops return the latest value pushed) in the presence of garbage collection, but it is incorrect under explicit memory management. The memory-managed semantics suffers from a problem known as ABA, which indeed is related to a pointer race. The problem arises as follows. Some thread t executing pop sets its local variable *ptop* to the global top of the stack *pTop*, say address a. The variable *pnode* is assigned the second topmost address b. While t executes pop, another thread frees address a with a pop. Since it has been freed, address a can be re-allocated and pushed, becoming the top of the stack again. However, the stack might have grown in between the free and the re-allocation. As a consequence, b is no longer the second node from the top. Thread t now executes the `cas` (atomic compare-and-swap). The command first tests $pTop = ptop$ (to check for consistency of the program state: has the top of the stack moved?). The test passes since *pTop* has come back to a due to the re-allocation. Thread t then redirects *pTop* to *pnode*. This is a pointer race: t relies on the variable *ptop* where the address was freed, and the re-allocation was not under t's control. At the same time, this causes an error. If *pnode* no longer points to the second address from the top, moving *pTop* loses stack content. □

Performance-critical implementations often intentionally make use of pointer races and employ other mechanisms to protect themselves from harmful effects due to accidental re-allocations. The corrected version of Treiber's stack [10] for example equips every pointer with a version counter logging the updates. Pointer assignments then assign the address together with the value of the associated version counter, and the counters are taken into account in the comparisons within `cas`. That is, the `cas`$(pTop, ptop, pnode)$ command atomically executes the following code:

$\texttt{if}\ (pTop = ptop\ \wedge\ pTop.\texttt{version} = ptop.\texttt{version})\ \{$

$\quad pTop := pnode;\ pTop.\texttt{version} := ptop.\texttt{version} + 1;\ \texttt{return}\ true;$

$\}\ else\ \{\ \texttt{return}\ false;\ \}$

This makes the cas from Example 1 fail and prevents stack corruption. Another pointer race occurs when the pop in Line (10) dereferences a freed pointer. With version counters, this is harmless. Our basic theory, however, would consider the comparison as well as the dereference pointer races, deeming the corrected version of Treiber's stack buggy.

To cope with performance-critical applications that implement version counters or techniques such as hazard pointers [11], reference counting [6], or grace periods [8], we strengthen the notion of pointer race. We let it tolerate assertions on freed pointers and dereferences of freed pointers where the value obtained by the dereference does not visibly influence the computation (e.g., it is assigned to a dead variable). To analyse programs that are only free from strong pointer races, the garbage-collected semantics is no longer sufficient. We define a more general ownership-respecting semantics by imposing an ownership discipline on top of the memory-managed semantics. With this semantics, we are able to show the following analogues of the above results. First, if the program is free from strong pointer races (SPRF) under the memory-managed semantics, then the memory-managed semantics coincides with the ownership-respecting semantics. Second, the memory-managed semantics is SPRF if and only if the ownership-respecting semantics is SPRF.

As a last contribution, we show how to apply our theory to optimize thread-modular reasoning. The idea of thread-modular analysis is to buy efficiency by abstracting from the relationship between the local states of individual threads. The loss of precision, however, is often too severe. For instance, any inductive invariant strong enough to show memory safety of Treiber's stack must correlate the local states of threads. Thread-modular analyses must compensate this loss of precision. Under garbage collection, an efficient way used e.g. in [7,17] is keeping as a part of the local state of each thread information about the ownership of memory addresses. A thread owns an allocated address. No other thread can access it until it enters the shared part of the heap. Unfortunately, this exclusivity cannot be guaranteed under the memory-managed semantics. Addresses can be re-allocated with pointers of other threads still pointing to them. Works such as [15,1] therefore correlate the local states of threads by more expensive means (cf. Section 5), for which they pay by severely decreased scalability.

We apply our theory to put back ownership information into thread-modular reasoning under explicit memory management. We measure the impact of our technique on the method of [1] when used to prove linearizability of programs such as Treiber's stack or Michael & Scott's lock-free queue under explicit memory management. We report on resource savings of about two orders of magnitude.

Contributions We claim the following contributions, where $[\![P]\!]_{mm}$ denotes the memory-managed semantics, $[\![P]\!]_{own}$ the ownership-respecting semantics, and $[\![P]\!]_{gc}$ the garbage-collected semantics of program P.

(1) We define a notion of pointer race freedom (PRF) and an equivalence $\approx$ among computations such that the following two results hold.

(1.1) If $[\![P]\!]_{mm}$ is PRF, then $[\![P]\!]_{mm} \approx [\![P]\!]_{gc}$.

(1.2) $[\![P]\!]_{mm}$ is PRF if and only if $[\![P]\!]_{gc}$ is PRF.

(2) We define a notion of strong pointer race freedom (SPRF) and an ownership-respecting semantics $[\![P]\!]_{own}$ such that the following two results hold.

(2.1) If $[\![P]\!]_{mm}$ is SPRF, then $[\![P]\!]_{mm} = [\![P]\!]_{own}$.

(2.2) $[\![P]\!]_{mm}$ is SPRF if and only if $[\![P]\!]_{own}$ is SPRF.

(3) Using the Results (2.1) and (2.2), we optimize the recent thread-modular analysis [1] by a use of ownership and report on an experimental evaluation.

The Results (2.1) and (2.2) give less guarantees than (1.1) and (1.2) and hence allow for less simplifications of program analyses. On the other hand, the stronger notion of pointer race makes (2.1) and (2.2) applicable to a wider class of programs which would be racy in the original sense (which is the case for our most challenging benchmarks).

Finally, we note that our results are not only relevant for concurrent programs but apply to sequential programs as well. The point in the definition of pointer race freedom is to guarantee the following: the execution does not depend on whether a malloc has re-allocated an address, possibly with other pointers still pointing to it, or it has allocated a fresh address. However, it is mainly reasoning about concurrent programs where we see a motivation to strive for such guarantees.

Related Work Our work was inspired by the data race freedom (DRF) guarantee [2]. The DRF guarantee can be understood as a contract between hardware architects and programming language designers. If the program is DRF under sequential consistency (SC), then the semantics on the actual architecture will coincide with SC. We split the analogue of the statement into two, coincidence ($[\![P]\!]_{mm}$ PRF implies $[\![P]\!]_{mm} \approx [\![P]\!]_{gc}$) and means of checking ($[\![P]\!]_{mm}$ PRF iff $[\![P]\!]_{gc}$ PRF). There are works that weaken the DRF requirement while still admitting efficient analyses [13,3,4]. Our notion of strong pointer races is along this line.

The closest related work is [8]. Gotsman et al. study re-allocation under explicit memory management. The authors focus on lock-free data structures implemented with hazard pointers, read-copy-update, or epoch-based reclamation. The key observation is that all three techniques rely on a common synchronization pattern called grace periods. Within a grace period of a cell a and a thread t, the thread can safely access the cell without having to fear a free command. The authors give thread-modular reasoning principles for grace periods and show that they lead to elegant and scalable proofs.

The relationship with our work is as follows. If grace periods are respected, then the program is guaranteed to be SPRF (there are equality checks on freed addresses). Hence, using Theorem 3 in this work, it is sufficient to verify lock-free algorithms under the ownership-respecting semantics. Interestingly, Gotsman et al. had an intuitive idea of pointer races without making the notion precise (quote: *...potentially harmful race between threads accessing nodes and those trying to reclaim them is avoided* [8]). Moreover, they did not study the implications of race freedom on the semantics, which is the main interest of this paper. We stress that our approach does not make assumptions about the synchronization strategy. Finally, Gotsman et al. do not consider the problem of checking the synchronization scheme required by grace periods. We show that PRF and SPRF can actually be checked on simpler semantics.

Data refinement in the presence of low-level memory operation is studied in [12]. The work defines a notion of substitutability that only requires a refinement of *error-free computations*. In particular, there is no need to refine computations that dereference dangling pointers. In our terms, these dereferences yield pointer races. We consider [12] as supporting our requirement for (S)PRF.

The practical motivation of our work, thread-modular analysis [5], has already been discussed. We note the adaptation to heap-manipulating programs [7]. Interesting is also the combination with separation logic from [17,16] (which uses ownership to improve precision). There are other works studying shape analysis and thread-modular analysis. As these fields are only a part of the motivation, we do not provide a full overview.

2 Heap-manipulating Programs

Syntax We consider **concurrent heap-manipulating programs**, defined to be sets of threads $P = \{t_1, t_2, \ldots\}$ from a set *Thrd*. We do not assume finiteness of programs. This ensures our results carry over to programs with a parametric number of threads. Threads t are ordinary while-programs operating on data and pointer variables. Data variables are denoted by $x, y \in \mathit{DVar}$. For pointer variables, we use $p, q \in \mathit{PVar}$. We assume $\mathit{DVar} \cap \mathit{PVar} = \emptyset$ and obey this typing. Pointer variables come with selectors $p.\mathtt{next_1}, \ldots, p.\mathtt{next_n}$ and $p.\mathtt{data}$ for finitely many pointer fields and one data field (for simplicity; the generalization to arbitrary data fields is straightforward). We use *pt* to refer to pointers p and $p.\mathtt{next}$. Similarly, by *dt* we mean data variables x and the corresponding selectors $p.\mathtt{data}$. Pointer and data variables are either *local* to a thread, indicated by $p, x \in \mathit{local}_t$, or they are *shared* among the threads in the program. We use *shared* for the set of all shared variables.

The **commands** $\mathit{com} \in \mathit{Com}$ employed in our while-language are

$$\begin{aligned} \mathit{cond} &::= \ p = q \mid x = y \mid \neg \mathit{cond} \\ \mathit{com} &::= \ \mathtt{assert}\ \mathit{cond} \mid p := \mathtt{malloc} \mid p := \mathtt{free} \\ &\quad \mid q := p.\mathtt{next} \mid p.\mathtt{next} := q \mid p := q \\ &\quad \mid x := p.\mathtt{data} \mid p.\mathtt{data} := x \mid x := \mathtt{op}(x_1, \ldots, x_n)\ . \end{aligned}$$

Pointer variables are allocated with $p := \mathtt{malloc}$ and freed via $p := \mathtt{free}$. Pointers and data variables can be used in assignments. These assignments are subject to typing: we only assign pointers to pointers and data to data. Moreover, a thread only uses shared variables and its own local variables. To compute on data variables, we support operations op that are not specified further. We only assume that the program comes with a data domain $(\mathit{Dom}, \mathit{Op})$ so that its operations op stem from *Op*. We support assertions that depend on equalities and inequalities among pointers and data variables. Like in if and while commands, we require assertions to have complements: if a control location has a command assert *cond*, then it also has a command assert $\neg \mathit{cond}$. We use as a running example the program in Example 1, Treiber's stack [14].

Semantics A heap is defined over a set of addresses Adr that contains the distinguished element $\mathtt{seg}$. Value $\mathtt{seg}$ indicates that a pointer has not yet been assigned a cell and thus its data and next selectors cannot be accessed. Such an access would result in a segfault. A heap gives the valuation of pointer variables $PVar \nrightarrow Adr$, the valuation of the next selector functions $Adr \nrightarrow Adr$, the valuation of the data variables $DVar \nrightarrow Dom$, and the valuation of the data selector fields $Adr \nrightarrow Dom$. In Section 3, we will restrict heaps to a subset of so-called valid pointers. To handle such restrictions, it is convenient to let heaps evaluate expressions $a.\mathtt{next}$ rather than next functions. Moreover, with the use of restrictions valuation functions will typically be partial.

Let $PExp := PVar \uplus \{a.\mathtt{next} \mid a \in Adr \setminus \{\mathtt{seg}\}$ and $\mathtt{next}$ a selector$\}$ be the set of pointer expressions and $DExp := DVar \uplus \{a.\mathtt{data} \mid a \in Adr \setminus \{\mathtt{seg}\}\}$ be the set of data expressions. A **heap** is a pair $h = (pval, dval)$ with $pval : PExp \nrightarrow Adr$ the valuation of the pointer expressions and $dval : DExp \nrightarrow Dom$ the valuation of the data expressions. We use $pexp$ and $dexp$ for a pointer and a data expression, and also write $h(pexp)$ or $h(dexp)$. The valuation functions are clear from the expression. The addresses inside the heap that are actually in use are

$$adr(h) := (dom(pval) \cup range(pval) \cup dom(dval)) \cap Adr.$$

Here, we use $\{a.\mathtt{next}\} \cap Adr := \{a\}$ and similar for data selectors.

We model heap modifications with **updates** $[pexp \mapsto a]$ and $[dexp \mapsto d]$ from the set Upd. Update $[pexp \mapsto a]$ turns the partial function $pval$ into the new partial function $pval[pexp \mapsto a]$ with $dom(pval[pexp \mapsto a]) := dom(pval) \cup \{pexp\}$. It is defined by $pval[pexp \mapsto a](qexp) := pval(qexp)$ if $qexp \neq pexp$, and $pval[pexp \mapsto a](pexp) := a$. We also write $h[pexp \mapsto a]$ since the valuation that is altered is clear from the update.

We define three semantics for concurrent heap-manipulating programs. All three are in terms of computations, sequences of actions from $Act := Thrd \times Com \times Upd$. An action $act = (t, com, up)$ consist of a thread t, a command com executed in the thread, and an update up. By $thrd(act) := t$, $com(act) := com$, and $upd(act) := up$ we access the thread, the command, and the update in act. To make the heap resulting from a computation $\tau \in Act^*$ explicit, we define $h_\varepsilon := (\emptyset, \emptyset)$ and $h_{\tau.act} := h_\tau[upd(act)]$. So we modify the current heap with the update required by the last action.

The garbage-collected semantics and the memory-managed semantics only differ on allocations. We define a strict form of garbage collection that never re-allocates a cell. With this, we do not have to define unreachable parts of the heap that should be garbage collected. We only model computations that are free from segfaults. This means a transition accessing next and data selectors is enabled only if the corresponding pointer is assigned a cell.

Formally, the **garbage-collected semantics** of a program P, denoted by $[\![P]\!]_{gc}$, is a set of computations in Act^*. The definition is inductive. In the base case, we have single actions $(\perp, \perp, [pval, dval]) \in [\![P]\!]_{gc}$ with $pval : PVar \to \{\mathtt{seg}\}$ and $dval : DVar \to Dom$ arbitrary. No pointer variable is mapped to a cell and the data variables contain arbitrary values. In the induction step, consider $\tau \in [\![P]\!]_{gc}$ where thread t is ready to execute command com. Then $\tau.(t, com, up) \in [\![P]\!]_{gc}$, provided one of the following rules holds.

(Asgn) Let com be $p.\mathtt{next} := q$, $h_\tau(p) = a \neq \mathtt{seg}$, $h_\tau(q) = b$. We set $up = [a.\mathtt{next} \mapsto b]$. The remaining assignments are similar.

(Asrt) Let *com* be $\texttt{assert}\ p = q$. The precondition is $h_\tau(p) = h_\tau(q)$. There are no updates, $up = \emptyset$. The assertion with a negated condition is defined analogously. A special case occurs if $h_\tau(p)$ or $h_\tau(q)$ is $\texttt{seg}$. Then the assert and its negation will pass. Intuitively, undefined pointers hold arbitrary values. Our development does not depend on this modeling choice.

(Free) If *com* is $p := \texttt{free}$, there are no constraints and no updates.

(Malloc1) Let *com* be $p := \texttt{malloc}$, $a \notin adr(h_\tau)$, and $d \in Dom$. Then we define $up = [p \mapsto a, a.\texttt{data} \mapsto d, \{a.\texttt{next} \mapsto \texttt{seg} \mid \text{ for every selector } \texttt{next}\}]$. The rule only allocates cells that have not been used in the computation. Such a cell holds an arbitrary data value and the next selectors have not yet been allocated.

With explicit memory management, we can re-allocate a cell as soon as it has been freed. Formally, the **memory-managed semantics** $[\![P]\!]_{mm} \subseteq Act^*$ is defined like $[\![P]\!]_{gc}$ but has a second allocation rule:

(Malloc2) Let *com* be $p := \texttt{malloc}$ and $a \in freed_\tau$. Then $up = [p \mapsto a]$.

Note that (Malloc2) does not alter the selectors of address a. The set $freed_\tau$ contains the addresses that have been allocated in τ and freed afterwards. The definition is by induction. In the base case, we have $freed_\varepsilon := \emptyset$. The step case is

$$\begin{aligned} freed_{\tau.(t,p:=\texttt{free},up)} &:= freed_\tau \cup \{a\}, && \text{if } h_\tau(p) = a \neq \texttt{seg} \\ freed_{\tau.(t,p:=\texttt{malloc},up)} &:= freed_\tau \setminus \{a\}, && \text{if } \texttt{malloc} \text{ returns } a \\ freed_{\tau.(t,act,up)} &:= freed_\tau, && \text{otherwise.} \end{aligned}$$

3 Pointer Race Freedom

We show that for well-behaved programs the garbage-collected semantics coincides with the memory-managed semantics. Well-behaved means there is no computation where one pointer frees a cell and later a dangling pointer accesses this cell. We call such a situation a **pointer race**, referring to the fact that the free command and the access are not synchronized, for otherwise the access should have been avoided. To apply this equivalence, we continue to show how to reduce the check for pointer race freedom itself to the garbage-collected semantics.

3.1 PRF Guarantee

The definition of pointer races relies on a notion of validity for pointer expressions. To capture the situation sketched above, a pointer is invalidated if the cell it points to is freed. A pointer race is now an access to an invalid pointer. The definition of validity requires care when we pass pointers. Consider an assignment $p := q.\texttt{next}$ where q points to a and $a.\texttt{next}$ points to b. Then p becomes a valid pointer to b only if both q and $a.\texttt{next}$ were valid. In Definition 1, we use *pexp* to uniformly refer to p and $a.\texttt{next}$ on the left-hand side of assignments. In particular, we evaluate pointer variables p to $h_\tau(p) = a$ and write $a.\texttt{next} := q$ for the assignment $p.\texttt{next} := q$.

Definition 1. *The* **valid** *pointer expressions in a computation* $\tau \in [\![P]\!]_{mm}$*, denoted by* $valid_\tau \subseteq PExp$*, are defined inductively by* $valid_\varepsilon := PExp$ *and*

$$\begin{aligned}
valid_{\tau.(t,p:=q.\mathtt{next},up)} &:= valid_\tau \cup \{p\}, && \textit{if } q \in valid_\tau \wedge h_\tau(q).\mathtt{next} \in valid_\tau \\
valid_{\tau.(t,p:=q.\mathtt{next},up)} &:= valid_\tau \setminus \{p\}, && \textit{if } q \notin valid_\tau \vee h_\tau(q).\mathtt{next} \notin valid_\tau \\
valid_{\tau.(t,pexp:=q,up)} &:= valid_\tau \cup \{pexp\}, && \textit{if } q \in valid_\tau \\
valid_{\tau.(t,pexp:=q,up)} &:= valid_\tau \setminus \{pexp\}, && \textit{if } q \notin valid_\tau \\
valid_{\tau.(t,p:=\mathtt{free},up)} &:= valid_\tau \setminus invalid_a, && \textit{if } a = h_\tau(p) \\
valid_{\tau.(t,p:=\mathtt{malloc},up)} &:= valid_\tau \cup \{p\}, && \\
valid_{\tau.(t,act,up)} &:= valid_\tau, && \textit{otherwise.}
\end{aligned}$$

If $a \neq \mathtt{seg}$*, then* $invalid_a := \{pexp \mid h_\tau(pexp) = a\} \cup \{a.\mathtt{next_1}, \ldots, a.\mathtt{next_n}\}$*. If* $a = \mathtt{seg}$*, then* $invalid_a := \emptyset$*.*

When we pass a valid pointer, this validates the receiver (adds it to $valid_\tau$). When we pass an invalid pointer, this invalidates the receiver. As a result, only some selectors of an address may be valid. When we free an address $a \neq \mathtt{seg}$, all expressions that point to a as well as all next selectors of a become invalid. This has the effect of isolating a so that the address behaves like a fresh one for valid pointers. A malloc validates the respective pointer but does not validate the next selectors of the allocated address.

Definition 2 (Pointer Race). *A computation* $\tau.(t, com, up) \in [\![P]\!]_{mm}$ *is called a* **pointer race (PR)***, if com is*

(i) a command containing $p.\mathtt{data}$ *or* $p.\mathtt{next}$ *or* $p := \mathtt{free}$ *with* $p \notin valid_\tau$*, or*
(ii) an assertion containing $p \notin valid_\tau$*.*

The last action of a PR is said to **raise a PR**. A set of computations is **pointer race free (PRF)** if it does not contain a PR. In Example 1, the discussed comparison $pTop = ptop$ within `cas` raises a PR since *ptop* is invalid. It is worth noting that we can still pass around freed addresses without raising a PR. This means the memory-managed and the garbage-collected semantics will not yield isomorphic heaps, but only yield isomorphic heaps on the valid pointers. We now define the notion of isomorphism among heaps h.

A function $f_{adr} : adr(h) \to Adr$ is an address mapping, if $f_{adr}(a) = \mathtt{seg}$ if and only if $a = \mathtt{seg}$. Every address mapping induces a function $f_{exp} : dom(h) \to PExp \cup DExp$ on the pointer and data expressions inside the heap by

$$f_{exp}(p) := p \qquad\qquad f_{exp}(x) := x$$
$$f_{exp}(a.\mathtt{next}) := f_{adr}(a).\mathtt{next} \qquad\qquad f_{exp}(a.\mathtt{data}) := f_{adr}(a).\mathtt{data}.$$

Pointer and data variables are mapped identically. Pointers on the heap $a.\mathtt{next}$ are mapped to $f_{adr}(a).\mathtt{next}$ as defined by the address mapping, and similar for the data.

Definition 3. *Two heaps* h_1 *and* h_2 *with* $h_i = (pval_i, dval_i)$ *are* **isomorphic***, denoted by* $h_1 \equiv h_2$*, if there is a bijective address mapping* $iso_{adr} : adr(h_1) \to adr(h_2)$ *where the induced* $iso_{exp} : dom(h_1) \to dom(h_2)$ *is again bijective and satisfies*

$$iso_{adr}(pval_1(pexp)) = pval_2(iso_{exp}(pexp)) \qquad dval_1(dexp) = dval_2(iso_{exp}(dexp)).$$

To prove a correspondence between the two semantics, we restrict heaps to the valid pointers. The restriction operation keeps the data selectors for all addresses that remain. To be more precise, let $h = (pval, dval)$ and $P \subseteq PExp$. The **restriction of** h **to** P is the new heap $h|_P := (pval|_P, dval|_D)$ with

$$D := DVar \cup \{a.\mathtt{data} \mid \exists pexp \in dom(pval) \cap P : pval(pexp) = a\} \ .$$

Restriction and update enjoy a pleasant interplay with isomorphism.

Lemma 1. *Let $h_1 \equiv h_2$ via iso_{adr} and let $P \subseteq PExp$. Then*

$$h_1|_P \equiv h_2|_{iso_{exp}(P)} \tag{14}$$

$$h_1[a.\mathtt{next} \mapsto b] \equiv h_2[a'.\mathtt{next} \mapsto b'] \tag{15}$$

$$h_1[a.\mathtt{data} \mapsto d] \equiv h_2[a'.\mathtt{data} \mapsto d]. \tag{16}$$

Isomorphisms (15) *and* (16) *have a side condition. If $a \in adr(h_1)$ then $a' = iso_{adr}(a)$. If $a \notin adr(h_1)$ then $a' \notin adr(h_2)$, and similar for b.*

Two computations are heap equivalent, if their sequences of actions coincide when projected to the threads and commands, and if the resulting heaps are isomorphic on the valid part. We use $\downarrow$ for projection.

Definition 4. *Computations $\tau, \sigma \in [\![P]\!]_{mm}$ are* **heap-equivalent**, *$\tau \approx \sigma$, if*

$$\tau \downarrow_{Thrd \times Com} = \sigma \downarrow_{Thrd \times Com} \qquad \textit{and} \qquad h_\tau|_{valid_\tau} \equiv h_\sigma|_{valid_\sigma} \ .$$

We also write $[\![P]\!]_{mm} \approx [\![P]\!]_{gc}$ to state that for every computation $\tau \in [\![P]\!]_{mm}$, there is a computation $\sigma \in [\![P]\!]_{gc}$ with $\tau \approx \sigma$, and vice versa.

We are now ready to state the PRF guarantee. The idea is to consider pointer races programming errors. If a program has pointer races, the programmer should be warned. If the program is PRF, further analyses can rely on the garbage-collected semantics:

Theorem 1 (PRF Guarantee). *If $[\![P]\!]_{mm}$ is PRF, then $[\![P]\!]_{mm} \approx [\![P]\!]_{gc}$.*

The memory-managed semantics of Treiber's stack suffers from the ABA-problem while the garbage-collected semantics does not. The two are not heap-equivalent. By Theorem 1, the difference is due to a PR. One such race is discussed in Example 1.

In the proof of Theorem 1, the inclusion from right to left always holds. The reverse direction needs information about the freed addresses: if an address has been freed, it no longer occurs in the valid part of the heap — provided the computation is PRF.

Lemma 2. *Assume $\tau \in [\![P]\!]_{mm}$ is PRF. Then $freed_\tau \cap adr(h_\tau|_{valid_\tau}) = \emptyset$.*

Lemma 1 and 2 allow us to prove Proposition 1. The result implies the missing direction of Theorem 1 and will also be helpful later on.

Proposition 1. *Consider $\tau \in [\![P]\!]_{mm}$ PRF. Then there is $\sigma \in [\![P]\!]_{gc}$ with $\sigma \approx \tau$.*

To apply Theorem 1, one has to prove $[\![P]\!]_{mm}$ PRF. We develop a technique for this.

3.2 Checking PRF

We show that checking pointer race freedom for the memory-managed semantics can be reduced to checking pointer race freedom for the garbage-collected semantics. The key argument is that the earliest possible PR always lie in the garbage-collected semantics. Technically, we consider a shortest PR in the memory-managed semantics and from this construct a PR in the garbage-collected semantics.

Theorem 2 (Checking PRF). *$[\![P]\!]_{mm}$ is PRF if and only if $[\![P]\!]_{gc}$ is PRF.*

To illustrate the result, the pointer race in Example 1 belongs to the memory-managed semantics. Under garbage collection, there is a similar computation which does not re-allocate a. Freeing a still renders *ptop* invalid and, as before, leads to a PR in `cas`. The proof of Theorem 2 applies Proposition 1 to mimic the shortest racy computation up to the last action. To mimic the action that raises the PR, we need the fact that an invalid pointer variable does not hold `seg`, as stated in the following lemma.

Lemma 3. *Consider a PRF computation $\sigma \in [\![P]\!]_{gc}$. (i) If $p \notin valid_\sigma$, then $h_\sigma(p) \neq \texttt{seg}$. (ii) If $pexp \in valid_\sigma$, $h_\sigma(pexp) = a \neq \texttt{seg}$, and $a.\texttt{next} \notin valid_\sigma$, then $h_\sigma(a.\texttt{next}) \neq \texttt{seg}$.*

While the completeness proof of Theorem 2 is non-trivial, checking PRF for $[\![P]\!]_{gc}$ is an easy task. One instruments the given program P to a new program P' as follows: P' tags every address that is freed and checks whether a tagged address is dereferenced, freed, or used in an assertion. In this case, P' enters a distinguished goal state.

Proposition 2. *$[\![P]\!]_{gc}$ is PRF if and only if $[\![P']\!]_{gc}$ cannot reach the goal state.*

For the correctness proof, we only need to observe that under garbage collection the invalid pointers are precisely the pointers to the freed cells.

Lemma 4. *Let $\sigma \in [\![P]\!]_{gc}$ and $h_\sigma(pexp) = a \neq \texttt{seg}$. Then $pexp \notin valid_\sigma$ iff $a \in freed_\sigma$.*

The lemma does not hold for the memory-managed semantics. Moreover, the statement turns Lemma 2, which can be read as an implication, into an equivalence. Namely, Lemma 2 says that if a pointer has been freed, then it cannot be valid. Under the assumtpions of Lemma 4, it also holds that if a pointer is not valid, then it has been freed.

4 Strong Pointer Race Freedom

The programing style in which a correct program should be pointer race free counts on the following policy: a memory address is freed only if it is not meant to be touched until its re-allocation, by any means possible.

This simplified treatment of dynamic memory is practical in common programing tasks, but the authors of performance-critical applications are often forced to employ subtler techniques. For example, the version of Treiber's stack equipped with version counters to prevent ABA under explicit memory management contains two violations of the simple policy, both of which are pointer races. (1) The `cas` may compare invalid pointers. This could potentially lead to ABA, but the programmer prevents the harmful effect of re-allocation using version counters, which make the `cas` fail. (2) The command $pnode := ptop.\texttt{next}$ in Line (10) of `pop` may dereference the `next` field of a freed (and therefore invalid) pointer. This is actually correct only under the assumption that neither the environment nor any thread of the program itself may redirect a once valid pointer outside the accessible memory (otherwise the dereference could lead to a segfault). The value obtained by the dereference may again be influenced by that the address was re-allocated. The reason for why this is fine is that the subsequent `cas` is bound to fail, which makes *pnode* a dead variable — its value does not matter.

In both cases, the programmer only prevents side effects of an accidental re-allocation. He uses a subtler policy and frees an address only if its *content* is not meant to be of any relevance any more. Invalid addresses can still be compared, and their pointer fields can even be dereferenced unless the obtained value influences the control.

4.1 SPRF Guarantee

We introduce a stronger notion of pointer race that expresses the above subtler policy. In the definition, we will call strongly invalid the pointer expressions that have obtained their value from dereferencing an invalid/freed pointer.

Definition 5 (Strong Invalidity). *The set of* **strongly invalid** *expressions in* $\tau \in [\![P]\!]_{mm}$*, denoted by* $sinvalid_\tau \subseteq PExp \cup DExp$*, is defined inductively by* $sinvalid_\varepsilon := \emptyset$ *and*

$$
\begin{aligned}
sinvalid_{\tau.(t,p:=q.\texttt{next},up)} &:= sinvalid_\tau \cup \{p\}, && \text{if } q \notin valid_\tau \\
sinvalid_{\tau.(t,pexp:=q,up)} &:= sinvalid_\tau \cup \{pexp\}, && \text{if } q \in sinvalid_\tau \\
sinvalid_{\tau.(t,x:=q.\texttt{data},up)} &:= sinvalid_\tau \cup \{x\}, && \text{if } q \notin valid_\tau \\
sinvalid_{\tau.(t,dexp:=x,up)} &:= sinvalid_\tau \cup \{dexp\}, && \text{if } x \in sinvalid_\tau \\
sinvalid_{\tau.act} &:= sinvalid_\tau \setminus valid_{\tau.act}, && \text{in all other cases.}
\end{aligned}
$$

The value obtained by dereferencing a freed pointer may depend on actions of other threads that point to the cell due to re-allocation. However, by assuming that a once valid pointer can never be set to `seg`, we obtain a guarantee that the actions of other threads cannot prevent the dereference itself from being executed (they cannot make it segfault). Assigning the uncontrolled value to a local variable is therefore not harmful. We only wish to prevent a correct computation from being influenced by that value. We thus define incorrect/racy any attempt to compare or dereference the value. Then, besides allowing for the creation of strongly invalid pointers, the notion of strong pointer race strengthens PR by tolerating comparisons of invalid pointers.

Definition 6 (Strong Pointer Race). *A computation* $\tau.(t,com,up) \in [\![P]\!]_{mm}$ *is a* **strong pointer race (SPR)***, if the command com is one of the following:*

(i) $p.\texttt{next} := q$ *or* $p.\texttt{data} := x$ *or* $p := \texttt{free}$ *with* $p \notin valid_\tau$
(ii) an assertion containing p or x in $sinvalid_\tau$
(iii) a command containing $p.\texttt{next}$ *or* $p.\texttt{data}$ *where* $p \in sinvalid_\tau$.

The last action of an SPR **raises an SPR**. A set of computations is **strong pointer race free (SPRF)** if it does not contain an SPR. An SPR can be seen in Example 1 as a continuation of the race ending at cas. The subsequent $ptop :=$ free raises an SPR as $ptop$ is invalid. The implementation corrected with version counters is SPRF.

Theorems 1 and 2 no longer hold for strong pointer race freedom. It is not possible to verify $[\![P]\!]_{mm}$ modulo SPRF by analysing $[\![P]\!]_{gc}$. The reason is that the garbage-collected semantics does not cover SPRF computations that compare or dereference invalid pointers. To formulate a sound analogy of the theorems, we have to replace $[\![.]\!]_{gc}$ by a more powerful semantics. This, however, comes with a trade-off. The new semantics should still be amenable to efficient thread-modular reasoning.

The idea of our new semantics $[\![P]\!]_{own}$ is to introduce the concept of ownership to the memory-managed semantics, and show that SPRF computations stick to it. Unlike with garbage collection, we cannot use a naive notion of ownership that guarantees the owner exclusive access to an address. This is too strong a guarantee. In $[\![P]\!]_{mm}$, other threads may still have access to an owned address via invalid pointers. Instead, we design ownership such that dangling pointers are not allowed to influence the owner. The computation will thus proceed as if the owner had allocated a fresh address.

To this end, we let a thread own an allocated address until one of the two events happen: either (1) the address is *published*, that is, it enters the shared part of the heap (which consists of addresses reached from shared variables by following valid pointers and of freed addresses), or (2) the address is *compromised*, that is, the owner finds out that the cell is not fresh by comparing it with an invalid pointer. Taking away ownership in this situation is needed since the owner can now change its behavior based on the re-allocation. The owner may also spread the information about the re-allocation among the other threads and change their behavior, too. It can thus no longer be guaranteed that the computation will continue as if a fresh address had been allocated.

Definition 7 (Owned Addresses). *For $\tau \in [\![P]\!]_{mm}$ and a thread t, we define the set of addresses* **owned by** *t, denoted by $own_\tau(t)$, as $own_\varepsilon(t) := \emptyset$ and*

$$\begin{array}{rll}
own_{\tau.(t,p:=\texttt{malloc},up)}(t) := own_\tau(t) \cup \{a\}, & \textit{if } p \in local_t \textit{ and } \texttt{malloc} \textit{ returns } a \\
own_{\tau.(t,p:=\texttt{free},\emptyset)}(t) := own_\tau(t) \setminus \{h_\tau(p)\}, & \textit{if } p \in valid_\tau \\
own_{\tau.(t,p:=q,[p\mapsto a])}(t) := own_\tau(t) \setminus \{a\}, & \textit{if } p \in shared \wedge q \in valid_\tau \\
own_{\tau.(t,p:=q.\texttt{next},[p\mapsto a])}(t) := own_\tau(t) \setminus \{a\}, & \textit{if } p \in shared \wedge q, h_\tau(q).\texttt{next} \in valid_\tau \\
own_{\tau.(\cdot,p:=q.\texttt{next},[p\mapsto a])}(t) := own_\tau(t) \setminus \{a\}, & \textit{if } h_\tau(q) \notin own_\tau(t) \wedge q, h_\tau(q).\texttt{next} \in valid_\tau \\
own_{\tau.(t,\texttt{assert }p=q,\emptyset)}(t) := own_\tau(t) \setminus \{h_\tau(p)\}, & \textit{if } p \notin valid_\tau \vee q \notin valid_\tau \\
own_{\tau.act}(t) := own_\tau(t), & \textit{in all other cases.}
\end{array}$$

The first four cases of losing ownership are due to publishing, the last case is due to the address being compromised by comparing with an invalid pointer.

The following lemma states the intuitive fact that an owned address cannot be pointed to by a valid shared variable or by a valid local variable of another thread, since such a configuration can be achieved only by publishing the address.

Lemma 5. *Let $\tau \in [\![P]\!]_{mm}$ and $p \in valid_\tau$ with $h_\tau(p) \in own_\tau(t)$. Then $p \in local_t$.*

We now define ownership violations as precisely those situations in which the fact that an owned address was re-allocated while an invalid pointer was still pointing to it influences the computation. Technically, the address is freed or its content is altered due to an access via a pointer of another thread or a shared pointer.

Definition 8 (Ownership Violation). *A computation* $\tau.(t, com, up) \in [\![P]\!]_{mm}$ **violates ownership***, if com is one of the following*

$$q.\texttt{next} := p, \quad q.\texttt{data} := x, \quad \textit{or} \quad q := \texttt{free},$$

where $h_\tau(q) \in own_\tau(t')$ *and* ($t' \neq t$ *or* $q \in shared$).

The last action of a computation violating ownership is called an **ownership violation** and a computation which does not violate ownership **respects ownership**. We define the **ownership-respecting semantics** $[\![P]\!]_{own}$ as those computations of $[\![P]\!]_{mm}$ that respect ownership. The following lemma shows that SPRF computations respect ownership.

Lemma 6. *If* $\tau.(t, com, act) \in [\![P]\!]_{mm}$ *violates ownership, then it is an SPR.*

The proof of Lemma 6 is immediate from Lemma 5 and the definitions of ownership violation and strong pointer race. The lemma implies the main result of this section: the memory-managed semantics coincides with the ownership-respecting semantics modulo SPRF.

Theorem 3 (SPRF Guarantee). *If* $[\![P]\!]_{mm}$ *is SPRF, then* $[\![P]\!]_{mm} = [\![P]\!]_{own}$.

4.2 Checking SPRF

This section establishes that checking SPRF may be done in the ownership-respecting semantics. In other words, if $[\![P]\!]_{mm}$ has an SPR, then there is also one in $[\![P]\!]_{own}$. This result, perhaps much less intuitively expected than the symmetrical result of Section 3.2, is particularly useful for optimizing thread-modular analysis of lock-free programs (cf. Section 5). Its proof depends on a subtle interplay of ownership and validity.

Let $ownpntrs_\tau$ be the *owning pointers*, pointers in h_τ to addresses that are owned by threads and the next fields of addresses owned by threads. To be included in $ownpntrs_\tau$, the pointers have to be valid. A set of pointers $O \subseteq ownpntrs_\tau$ is *coherent* if for all $pexp, qexp \in ownpntrs_\tau$ with the same target or source address (in case of $a.\texttt{next}$ or $a.\texttt{data}$) we have $pexp \in O$ if and only if $qexp \in O$.

Lemma 7 below establishes the following fact. For every computation that respects ownership, there is another one that coincides with it but assigns fresh cells to some of the owning pointers. To be more precise, given a computation $\tau \in [\![P]\!]_{own}$ and a coherent set of owning pointers $O \subseteq ownpntrs_\tau$, we can find another computation $\tau' \in [\![P]\!]_{own}$ where the resulting heap coincides with h_τ except for O. These pointers are assigned fresh addresses. The proof of Lemma 7 is nontrivial and can be found in [9].

Lemma 7. *Consider* $\tau \in [\![P]\!]_{own}$ *SPRF and* $O \subseteq ownpntrs_\tau$ *a coherent set. There is* $\tau' \in [\![P]\!]_{own}$ *and an address mapping* $f_{adr} : adr(O) \to Adr$ *that satisfy the following:*

$$(1)\ \tau \downarrow_{Thrd \times Com} = \tau' \downarrow_{Thrd \times Com} \qquad freed_\tau \subseteq freed_{\tau'} \qquad (4)$$

$$(2)\ h_\tau|_{PExp \setminus O} = h_{\tau'}|_{PExp \setminus f_{exp}(O)} \qquad ownpntrs_{\tau'} = (ownpntrs_\tau \setminus O) \cup f_{exp}(O) \qquad (5)$$

$$(3)\ h_\tau|_{valid_\tau} \equiv h_{\tau'}|_{valid_{\tau'}} \ \textit{by}\ f_{adr} \cup id \qquad adr(h_\tau) \cap h_{\tau'}(f_{exp}(O)) = \emptyset. \qquad (6)$$

In this lemma, function f_{adr} specifies the new addresses that τ' assigns to the owning expressions in O. These new addresses are fresh by Point (6). Point (1) says that τ and τ' are the same up to the particular addresses they manipulate, and Point (2) says that the reached states h_τ and $h_{\tau'}$ are the same up to the pointers touched by f_{adr}. Point (3) states that the valid pointers of h_τ stay valid or become valid f_{exp}-images of the originals. Point (5) says that also the owned pointers of h_τ remain the same or become f_{exp}-images of the originals. Finally, Point (4) says that $h_{\tau'}$ re-allocates less cells.

Lemma 7 is a cornerstone in the proof of the main result in this section, namely that SPRF is equivalent for the memory-managed and the ownership-respecting semantics.

Theorem 4 (Checking SPRF). *$[\![P]\!]_{mm}$ is SPRF if and only if $[\![P]\!]_{own}$ is SPRF.*

Proof. If $[\![P]\!]_{mm}$ is SPRF, by $[\![P]\!]_{own} \subseteq [\![P]\!]_{mm}$ this carries over to the ownership-respecting semantics. For the reverse direction, assume $[\![P]\!]_{mm}$ has an SPR. In this case, there is a shortest computation $\tau.act \in [\![P]\!]_{mm}$ where *act* raises an SPR. In case $\tau.act \in [\![P]\!]_{own}$, we obtain the same SPR in the ownership-respecting semantics.

Assume $\tau.act \notin [\![P]\!]_{own}$. We first argue that *act* violates ownership. By prefix closure, $\tau \in [\![P]\!]_{mm}$. By minimality, τ is SPRF. Since ownership violations are SPR by Lemma 6, τ does not contain any, $\tau \in [\![P]\!]_{own}$. Hence, if *act* respected ownership we could extend τ to the computation $\tau.act \in [\![P]\!]_{own}$ — a contradiction to our assumption.

We turn this ownership violation in the memory-managed semantics into an SPR in the ownership-respecting semantics. To this end, we construct a new computation $\tau'.act' \in [\![P]\!]_{own}$ that mimics $\tau.act$, respects ownership, but suffers from SPR. Since $\tau.act$ is an ownership violation, *act* takes the form (t, com, up) with *com* being

$$q.\texttt{next} := p, \quad q.\texttt{data} := x, \quad \text{or} \quad q := \texttt{free}.$$

Here, $h_\tau(q) \in own_\tau(t')$ and ($t' \neq t$ or $q \in shared$). Since the address is owned, Lemma 5 implies $q \notin valid_\tau$.

As a first step towards the new computation, we construct τ'. Let $O := ownpntrs_\tau$ be the (coherent) set of all owning pointers in all threads (with $q \notin O$). With this choice of O, we apply Lemma 7. It returns $\tau' \in [\![P]\!]_{own}$ with $\tau' \downarrow_{Thrd \times Com} = \tau \downarrow_{Thrd \times Com}$ and

$$h_{\tau'}|_{PExp \setminus f_{exp}(O)} = h_\tau|_{PExp \setminus O} \quad \text{and} \quad h_{\tau'}|_{valid_{\tau'}} \equiv h_\tau|_{valid_\tau}.$$

Address $h_{\tau'}(q)$ is not owned by any thread. This follows from

$$ownpntrs_{\tau'} = (ownpntrs_\tau \setminus O) \cup f_{exp}(O) = f_{exp}(O)$$

and $q \notin f_{exp}(O)$. Finally, $q \notin valid_{\tau'}$ by the isomorphism $h_{\tau'}|_{valid_{\tau'}} \equiv h_\tau|_{valid_\tau}$.

As a last step, we mimic $act = (t, com, up)$ by an action $act' = (t, com, up')$. If *com* is $q := \texttt{free}$, then we free the invalid pointer $q \notin valid_{\tau'}$ and obtain an SPR in $[\![P]\!]_{own}$. Assume *com* is an assignment $q.\texttt{next} := p$ (the case of $q.\texttt{data} := x$ is similar). Since *act* is enabled after τ and $h_{\tau'}(q) = h_\tau(q)$, we have $h_{\tau'}(q) \neq \texttt{seg}$. Hence, the command is also enabled after τ'. Since $q \notin valid_{\tau'}$, the assignment is again to an invalid pointer. It is thus an SPR according by Definition 6.(i). $\square$

5 Improving Thread-Modular Analyses

We now describe how the theory developed so far can be used to increase the efficiency of thread-modular analyses of pointer programs under explicit memory management.

Thread-modular reasoning abstracts a program state into a set of states of individual threads. A thread's state consists of the local state, the part of the heap reachable from the local variables, and the shared state, the heap reachable from the shared variables.

The analysis saturates the set of reachable thread states by a fixpoint computation. Every step in this computation creates new thread states out of the existing ones by applying the following two rules. (1) Sequential step: a thread's state is modified by an action of this thread. (2) Interference: a state of a victim thread is changed by an action of another, interfering thread. This is accounted for by creating combined two-threads states from existing pairs of states of the victim and the interferer thread. The states that are combined have to agree on the shared part. The combined state is constructed by deciding which addresses in the two local states coincide. It is then observed how an action of the interferer changes the state of the victim within the combined state.

Pure thread-modular reasoning does not keep any information about what thread states can appear simultaneously during a computation and what identities can possibly hold between addresses of local states of threads. This brings efficiency, but also easily leads to false positives. To see this, consider in Treiber's stack a state s of a thread that is just about to perform the `cas` in `push`. Variable *pnode* points to an address a allocated in the first line of `push`, *pTop*, *ptop*, and *pnode*.`next` are at the top of the stack. Consider an interference step where the states s_v of the victim and s_i of the interferer are isomorphic to s, with *pnode* pointing to the newly allocated addresses a_v and a_i, respectively. Since the shared states conincide, the interference is triggered. The combination must account for all possible equalities among the local variables. Hence, there is a combined state with $a_v = a_i$, which does not occur in reality. This is a crucial imprecision, which leads to false positives. Namely, the interferer's `cas` succeeds, resulting in the new victim's state s'_v with *pTop* on a_i (which is equal to a_v). The victim's `cas` then fails, and the thread continues with the commands $ptop := pTop; pnode.\texttt{next} := ptop$. This results in a_v.`next` pointing back to a_v, and a loss of the stack content.

Methods based on thread-modular reasoning must prevent such false positives by maintaining the necessary information about correlations of local states. An efficient technique commonly used under garbage collection is based on ownership: a thread's state records that a has just been allocated and hence no other thread can access the address, until it enters the shared state. This is enough to prevent false positives such as the one described above. Namely, the addresses a_i and a_v are owned by the respective threads and therefore they cannot be equal. Interference may then safely ignore the problematic case when $a_v = a_i$. Moreover, besides the increased precision, the ability to avoid interference steps due to ownership significantly improves the overall efficiency. This technique was used for instance to prove safety (and linearizability) of Treiber's stack and other subtle lock-free algorithms in [17].

Under explicit memory management, ownership of this form cannot be guaranteed. Addresses can be freed and re-allocated while still being pointed to. Other techniques must be used to correlate the local states of threads. The solution chosen in [15,1] is to replace the states of individual threads by states of pairs of threads. Precision is thus restored at the cost of an almost quadratic blow-up of the abstract domain that in turn manifests itself in a severe decrease of scalability.

5.1 Pointer Race Freedom Saves Ownership

Using the results from Sections 3 and 4, we show how to apply the ownership-based optimization of thread-modular reasoning to the memory-managed semantics. To this end, we split the verification effort into two phases. Depending on the notion of pointer race freedom, we first check whether the program under scrutiny is (S)PRF. If the check fails, we report pointer races as potential errors to the developer. If the check succeeds, the second phase verifies the property of interest (here, linearizability) assuming (S)PRF.

When the notion of PRF from Section 3 is used, the second verification phase can be performed in the garbage-collected semantics due to Theorem 1. This allows us to apply the ownership-based optimization discussed above. Moreover, Theorem 2 says that the first PR has to appear in the garbage-collected semantics. Hence, even the first phase, checking PRF, can rely on garbage collection and ownership. The PRF check itself is simple. Validity of pointers is kept as a part of the individual thread states and updated at every sequential and interference step. Based on this, every computation step is checked for raising a PR according to Definition 2. Our experiments suggest that the overhead caused by the recorded validity information is low.

For SPRF, we proceed analogously. Due to the Theorems 3 and 4, checking SPRF in the first phase and property verification in the second phase can both be done in the ownership-respecting semantics. The SPRF check is similar to the PRF check. Validity of pointers together with an information about strong invalidity is kept as a part of a thread's state, and every step is checked for raising an SPR according to Definition 6.

The surprising good news is that both phases can again use the ownership-based optimization. That is, also in the ownership-respecting semantics, interferences on the owned memory addresses can be skipped. We argue that this is sound. Due to Lemma 5, if a thread t owns an address a, other threads may access a only via invalid pointers. Therefore, (1) modifications of a by t need not be considered as an interference step for other threads. Indeed, if a thread $t' \neq t$ was influenced by such a modification (t' reads a next or the data field of a), then the corresponding variable of t' would become strongly invalid, Definition 5. Hence, either this variable is never used in an assertion or in a dereference again (it is effectively dead), or the first use raises an SPR, Cases (ii) and (iii) in Definition 6. (2) In turn, in the ownership-respecting semantics, another thread t' cannot make changes to a, by Definition 8 of ownership violations. This means we can also avoid the step where t' interferes with the victim t.

5.2 Experimental Results

To substantiate our claim for a more efficient analysis with practical experiments, we implemented the thread-modular analysis from [1] in a prototype tool. This analysis is challenging for three reasons: it checks linearizability as a non-trivial requirement, it handles an unbounded number of threads, and it supports an unbounded heap. Our tool covers the garbage-collected semantics, the new ownership-respecting semantics of Section 4, and the memory-managed semantics. For the former two, we use the abstract domain where local states refer to single threads. Moreover, we support the ownership-based pruning of interference steps from Section 5.1. For the memory-managed semantics, to restore precision as discussed above, the abstract domain needs local states with pairs of threads. Rather than running two phases, our tool combines the PRF check and the actual analysis. We tested our implementation on lock-free data structures from the literature and verified linearizability following the approach in [1].

Table 1. Experimental results for thread-modular reasoning using different memory semantics.

Program		time in seconds	explored state count	sequential step count	interference step count	pruned interferences	correctness established
Single lock stack	GC	0.053	328	941	3276	10160	yes
	OWN	0.21	703	1913	6983	22678	yes
	GC^-	0.20	507	1243	19321	–	yes
	OWN^-	0.60	950	2474	38117	–	yes
	MM^-	5.34	16117	25472	183388	–	yes
Single lock queue	GC	0.034	199	588	738	5718	yes
	OWN	0.56	520	1336	734	31200	yes
	GC^-	0.19	331	778	9539	–	yes
	OWN^-	2.52	790	1963	65025	–	yes
	MM^-	31.7	27499	60263	442306	–	yes
Treiber's lock free stack (with version counters) [10]	GC	0.052	269	779	3516	15379	yes
	OWN	2.36	744	2637	43261	95398	yes
	GC^-	0.16	296	837	11530	–	yes
	OWN^-	4.21	746	2158	73478	–	yes
	MM^-	602	116776	322057	7920186	–	yes
Michael & Scott's lock free queue [10] (with hints)	GC	2.52	3134	6607	46838	1237012	yes
	OWN	10564	19553	43305	6678240	20747559	yes
	GC^-	9.08	3309	7753	187349	–	yes
	OWN^-	51046	31329	64234	35477171	–	yes
	MM^-	*aborted*	$\geq$69000	$\geq$90000	–	–	false positive

The experimental results are listed in Table 1. The experiments were conducted on an Intel Xeon E5-2650 v3 running at 2.3 GHz. The table includes the following: (1) runtime taken to establish correctness, (2) number of explored thread states (i.e. size of the search space), (3) number of sequential steps, (4) number of interference steps, (5) number of interference steps that have been pruned by the ownership-based optimization, and (6) the result of the analysis, i.e. whether or not correctness could be established. For a comparison, we also include the results with the ownership-based optimization turned off (suffix $^-$). Recall that the optimization does not apply to the memory-managed semantics. We elaborate on our findings.

Our experiments confirm the usefulness of pointer race freedom. When equipped with pruning (OWN), the ownership-respecting semantics provides a speed-up of two orders of magnitude for Treiber's stack and the single lock data structures compared to the memory-managed semantics (MM^-). The size of the explored state space is close to the one for the garbage-collected semantics (GC) and up to two orders of magnitude smaller than the one for explicit memory management. We also stress tested our tool by purposely inserting pointer races, for example, by discarding the version counters. In all cases, the tool was able to detect those races.

For Michael & Scott's queue we had to provide hints in order to eliminate certain patterns of false positives. This is due to an imprecision that results from joins over a large number of states (we are using the joined representation of states from [1] based on Cartesian abstraction). Those hints are sufficient for the analysis relying on the ownership-respecting semantics to establish correctness. The memory-manged semantics produces more false positives, the elimination of which would require more hinting, as also witnessed by the implementation of [1]. Regarding the stress tests from above, note that we ran those tests with the same hints and were still able to find the purposely inserted bugs.

References

1. P.A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, volume 7795 of *LNCS*, pages 324–338. Springer, 2013.
2. S. V. Adve and M. D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, 1993.
3. J. Alglave and L. Maranget. Stability in Weak Memory Models. In *CAV*, volume 6806 of *LNCS*, pages 50–66. Springer, 2011.
4. A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding Robustness against Total Store Ordering. In *ICALP*, volume 6756 of *LNCS*, pages 428–440. Springer, 2011.
5. S. Qadeer C. Flanagan. Thread-modular model checking. In *SPIN*, pages 213–224, 2003.
6. David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. Lock-free reference counting. *Distrib. Comput.*, 15(4):255–271, December 2002.
7. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277. ACM, 2007.
8. A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, volume 7792 of *LNCS*, pages 249–269. Springer, 2013.
9. Frédéric Haziza, Lukáš Holík, Roland Meyer, and Sebastian Wolff. Pointer race freedom. Technical Report FIT-TR-2015-05, Brno University of Technology, FIT, 2015.
10. M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Dist. Comp.*, 51(1):1–26, 1998.
11. Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.
12. I. Mijajlovic and H. Yang. Data refinement with low-level pointer operations. In *APLAS*, volume 3780 of *LNCS*, pages 19–36. Springer, 2005.
13. S. Owens. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *ECOOP*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.
14. R.K.Treiber. Systems Programming: Coping with Parallelism. RJ5118, 1986.
15. M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Abstract transformers for thread correlation analysis. In *APLAS*, LNCS, pages 30–46. Springer, 2009.
16. V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.
17. V. Vafeiadis. RGSep action inference. In *VMCAI*, volume 5944 of *LNCS*, pages 345–361. Springer, 2010.